JAVA™ 2

Manual de usuario y tutorial

5ª edición



Agustín Froufe Quintas

Incluye CD-ROM
con el código completo
de los ejemplos

Ra-Ma®

JAVA 2

Manual de usuario y tutorial

5ª edición

J2SE 6

JAVA 2

Manual de usuario y tutorial

5ª edición

J2SE 6

Agustín Froufe Quintas

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com



La ley prohíbe
Copiar o Imprimir este libro

JAVA 2. Manual de Usuario y Tutorial. 5ª Edición. (E-Book)
© Agustín Froufe Quintas

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-7897-875-5
Todos los derechos reservados © RA-MA, S.A. Editorial y Publicaciones, Madrid, España.

MARCAS COMERCIALES. Las designaciones utilizadas por las empresas para distinguir sus productos (hardware, software, sistemas operativos, etc.) suelen ser marcas registradas. RA-MA ha intentado a lo largo de este libro distinguir las marcas comerciales de los términos descriptivos, siguiendo el estilo que utiliza el fabricante, sin intención de infringir la marca y solo en beneficio del propietario de la misma. Los datos de los ejemplos y pantallas son ficticios a no ser que se especifique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso ni tampoco de cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa o de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes, intencionadamente, reprodujeren o plagiaren, en todo o en parte, una obra literaria, artística o científica.

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Jesus Ramirez Galan
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-354-0

E-Book desarrollado en España en Septiembre de 2014

A Patri y a mi hermana,
porque siempre han confiado en mí.

ÍNDICE

PRÓLOGO ..... **XIII**

Prólogo a la Segunda edición XIII
Prólogo a la Tercera edición XV
Prólogo a la Cuarta edición XV

PREFACIO ..... **XVII**

CAPÍTULO 1: INTRODUCCIÓN A JAVA **1**

Origen de Java 3

CAPÍTULO 2: CARACTERÍSTICAS DE JAVA **7**

Características principales de Java 7
Java Community Process 11

CAPÍTULO 3: PRIMEROS PASOS EN JAVA **13**

Una mínima aplicación en Java 13
Un Applet básico en Java 18
Argumentos en la línea de comandos 21

CAPÍTULO 4: LENGUAJE JAVA **23**

Comentarios 23
Identificadores 24
Separadores 27
Operadores 28
Variables 34

Expresiones 34
Arrays 35
Strings 38
Tipos Enumerados 38
Anotaciones 39
Control de flujo 43
Tipos Genéricos 50

CAPÍTULO 5: PROGRAMAS BÁSICOS EN JAVA 55

El visor de Applets (*appletviewer*) 55
Escribir Applets Java 63

CAPÍTULO 6: CONCEPTOS BÁSICOS DE JAVA 67

Objetos 67
Clases 71
Control de acceso 89
Herencia 93
Subclases 94
Sobrescritura de métodos 94
Conversiones Implícitas 96
Clase Object 97
Clases abstractas 101
Interfaces 101
Paquetes 107

CAPÍTULO 7: CLASES JAVA 109

La clase Math 109
La clase Character 111
Las clases de tipos numéricos 111
La clase String 112
La clase StringBuffer 114
La clase StringTokenizer 116
Expresiones regulares 118
La clase Formatter 120
La clase Scanner 123
La clase Properties 123
La clase Runtime 125
La clase System 126

CAPÍTULO 8: ALMACENAMIENTO DE DATOS 131

Arrays 131
Colecciones 132
Enumeraciones 133
Iteradores 134

Tipos de Colecciones 136
Colecciones y Mapas 141
Operaciones No Soportadas 149
Ordenación y Búsqueda 151
Colecciones o Mapas de Sólo Lectura 156
Colecciones o Mapas Sincronizados 156

CAPÍTULO 9: FICHEROS EN JAVA 159

La clase File 159
Aplicaciones 162
Operaciones 164
Excepciones 166
Applets 167
Seguridad 167
Servlets 174
Acceso aleatorio 174
La arquitectura NIO 176

CAPÍTULO 10: EXCEPCIONES EN JAVA 181

Clasificación de excepciones 181
Manejo de excepciones 182
Generar excepciones en Java 183
Crear excepciones propias 186
Capturar excepciones 187
Propagación de excepciones 194
Aserciones 196

CAPÍTULO 11: TAREAS Y MULTITAREA 201

Programas de flujo único 202
Programas de flujo múltiple 203
Creación y control de tareas 204
Grupos de tareas 211
Arrancar y parar tareas 212
Suspender y reanudar tareas 214
Estados de una tarea 215
Scheduling 218
Comunicaciones Entre Tareas 219
Utilidades de Concurrencia 224

CAPÍTULO 12: DELEGACIÓN DE EVENTOS 231

Receptores de Eventos 236
Fuentes de Eventos 236
Adaptadores 238
Eventos de bajo nivel y semánticos 242

Eventos generados por el usuario....245
Eventos en Swing....258

CAPÍTULO 13: AWT....267

Interfaz de usuario....268
Estructura del AWT....269
Componentes y Contenedores....270
Componentes....270
Contenedores....289
Menús....301
Layouts....308
Escritorio....324
Imprimir con el AWT....330

CAPÍTULO 14: SWING....337

Arquiectura de Swing....340
Bordes....342
Etiquetas....343
Botones....344
Grupos de botones....345
Listas y cajas combinadas....347
Texto....352
Tool Tips....354
Iconos....357
Menús....359
Menús Popup....362
Escalas y barras de progreso....364
Árboles....367
Tablas....382
Pestañas....387
Selector de ficheros....390
Diálogos Predefinidos....391
Teclado....393
Paneles desplazables....394
Arrastrar y soltar....397
Look and Feel....400

CAPÍTULO 15: GRÁFICOS....415

El sistema de coordenadas....415
Pintado en AWT....417
Pintado en Swing....423
La clase Graphics....431
Animación....463
Uso avanzado de imágenes....469

CAPÍTULO 16: MÉTODOS NATIVOS 477

Escribir código Java 478
Compilar el código Java 480
Crear el fichero de cabecera 480
Escribir la función C 481
Crear la librería dinámica 482
Ejecutar el programa 484
Usar Librerías del Sistema 484

CAPÍTULO 17: COMUNICACIONES EN RED 487

Sockets 488
La clase InetAddress 491
La clase NetworkInterface 494
La clase URL 495
La clase URLConnection 500
La clase Socket 502
La clase DatagramPacket 520
La clase DatagramSocket 522
Comunicaciones seguras 529
Multicast 532

CAPÍTULO 18: SERVLETS 537

Servlets y CGI 538
API Servlet 539
La clase HttpServlet 540
Información adicional 568

CAPÍTULO 19: JDBC 573

Bases de Datos 573
Conectividad JDBC 576
Transacciones 587
Información de la Base de Datos 588
Tipos SQL en Java 590
Modelo relacional de objetos 592
Modelo de conexión 593
JDBC RowSets 596
JDBC y Servlets 600
JavaDB 608
Código independiente y portable 613

CAPÍTULO 20: RMI 617

HolaMundo Remoto 618
RMI con múLtiples objetos del mismo tipo 624

RMI con múltiples objetos de distinto tipo 626
Comunicación entre objetos 627
Rendimiento RMI 632
Comunicación RMI 634
Serialización de objetos 639
Validación de objetos 643

CAPÍTULO 21: JMX 649

Arquitectura JMX 649
Instrumentación de recursos 651
Utilización de jconsole 656

APÉNDICE A: BIBLIOGRAFÍA 659

APÉNDICE B: CONTENIDO DEL CD-ROM 663

ÍNDICE ALFABÉTICO 667

PRÓLOGO

PRÓLOGO A LA SEGUNDA EDICIÓN

Entre los excelentes libros de informática que se están escribiendo actualmente en castellano y por autores españoles se notaba la falta de un libro sobre Java que fuera a la vez didáctico y riguroso. Con el actual, el autor nos sorprende con uno que además de ser todo eso, es ameno y fácil de leer.

Para quienes conocemos la trayectoria del autor no nos sorprende. Ha demostrado su capacidad desde que era un inquieto estudiante hasta llegar a ser el experimentado desarrollador de complejos sistemas en que se ha convertido. Además no hay mejor maestro que quien ama la materia que enseña. El germen de este libro hay que buscarlo en su famoso *Tutorial de Java*, que convirtió la página web del autor en la más premiada (y visitada) en su momento. Este libro es más y menos. Mucho ha llovido y mucho ha crecido la experiencia del autor. Y eso se nota. Además de un texto profundo y muy completo, quiero destacar algunos aspectos: quien ha estudiado mucha informática sabe de la importancia de los buenos ejemplos y el libro está lleno de ellos; el libro es voluminoso, pero se avanza rápido: se hace corto. Además, desde el principio los ejemplos funcionan, ¡y hacen cosas interesantes!

Java es un lenguaje (o quizás más que un lenguaje, como el autor se encarga de convencernos), que ya no necesita presentación. Los profesionales de la informática saben encontrar Java en todas partes, aunque para el usuario normal pase desapercibido. Y esto es una prueba de la madurez de Java. Ahora nadie discute que Internet sería diferente si no existiera Java. Más aún, no sólo es importante por sí mismo, sino por la influencia que ha tenido en la informática. Es referencia obligada. Java ha sido un motor que ha impulsado y favorecido no sólo muchos elementos complementarios, sino también a sus competidores. Nadie puede afirmar cuál será su

futuro, pero sí su presente: hacer desaparecer ahora Java significaría un colapso difícil de imaginar. Como el aire, se nota mucho más su ausencia que su presencia.

No me resisto a destacar una importantísima característica de Java que muchos autores resaltan insuficientemente: su belleza. Cuando daba mis primeros pasos en informática quedé enamorado del PASCAL. Devoré los libros de N. Wirth y fui un converso. Era estructurado y modular. Era expresivo y potente. Para quien había lidiado con los horrores del FORTRAN o con los primeros BASIC, era ideal. Era el lenguaje en el que se debería iniciar el estudiante de informática, para formar buenos hábitos desde el principio. Pero el C era mucho más potente. Y fui un curtido programador de C. Pero había perdido algo: es fácil hacer cosas horribles en C. Afortunadamente C++ acudió al rescate: podía ser tan bueno y expresivo como PASCAL, tan potente como C, y cuando había que ser perverso..., bueno, se podía. Las cosas estaban de nuevo en su sitio. Sin embargo, algo iba mal, y Windows acabó de estropearlo: cualquier cosa necesitaba un millón de líneas de código (también en PASCAL). Se dice que los programadores de C++ tenemos una ventaja sobre otros programadores: si nuestro código se compila ya tenemos una satisfacción. Si además funcionan bien algunas cosas, ¡oh maravilla! Cuando el programa completo está depurado, ya tenemos la mayor joya de la informática moderna. Pero el ciclo de desarrollo de tal sistema es complejo y costoso y, sobre todo, lento. Y puede que difícil de mantener. Pero apareció Java. Si PASCAL era ideal, Java es casi perfecto. Es tan estructurado como se pueda desear. Es orientado a objetos (casi) como C++. Y es fácil. No hacen falta miles de líneas de código. Para programas sencillos, es empezar y hacerlo funcionar. Además, prohíbe ser malo. Se parece a C++, pero en realidad es pariente cercano de Visual Basic o (algo menos) de Delphi. Los punteros están ocultos: funcionan, pero no son asunto del programador. No hay que ocuparse de la asignación y liberación de memoria. Las variables globales están prohibidas. El tratamiento de errores, si se sabe hacer, es estupendo. Es el lenguaje que debería ser obligatorio para iniciarse en la informática del siglo XXI. ¿Y la eficiencia? Hablemos de ello.

Java es esencialmente portable. La independencia del sistema es su alma. En cada implementación debe haber un motor que entienda de Java. Pero eso tiene un precio: la eficiencia depende de ese intermediario. Tenemos entonces todo un espectro que va desde la lentitud de pobres implementaciones de intérpretes de Java, hasta máquinas diseñadas expresamente que entienden su código como instrucciones nativas. Esperemos al futuro; ya el presente es bueno.

Queremos programas de calidad. Esperamos programas fáciles de depurar. Queremos sistemas portables y extensibles. Java puede ayudar. Sólo cabe esperar que un libro de la calidad del presente lo impulse. Yo estoy convencido.

Gerardo Valeiras
Sevilla, mayo de 2000

PRÓLOGO A LA TERCERA EDICIÓN

Es para mí una satisfacción poder escribir unas notas sobre esta nueva edición. Estuve presente en la gestación de esta obra y siempre aposté por su calidad; creo que el tiempo me está dando la razón. La continua innovación tecnológica y concretamente la velocidad a la que esto sucede en el mundo de la Informática, hace que cualquier libro necesite actualizaciones constantes y sólo los buenos libros pueden mantener este ritmo, y únicamente los buenos autores, que además sean profesionales, pueden mantener sus conocimientos al día. Es claro que en el caso de la obra que nos ocupa se dan ambas circunstancias.

De entre la distinción que Sun ha establecido entre J2ME, J2EE y J2SE, este libro sólo trata del último. La lista de actualizaciones de los contenidos estaría fuera del alcance de este prólogo, pero no puedo resistirme a comentar las que me parecen más significativas, como la completa orientación a Java 2 (dejando las versiones anteriores como referencias históricas), la profundización en el tema de las colecciones, el estudio de nuevos elementos de Swing, como las barras de progreso, los spin, y algunos otros, la comunicación a través de sockets seguros, la comunicación Multicast, etc. Me gustaría destacar el profundo estudio de los servlets y su adaptación completa a la especificación actual, con ejemplos y explicaciones difíciles de encontrar reunidas, y mucho menos en un libro de carácter general como éste.

Gerardo Valeiras
Doctor en Matemáticas, Catedrático de Escuela Universitaria
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

Sevilla, abril de 2002

PRÓLOGO A LA CUARTA EDICIÓN

Es para mí una satisfacción poder prologar la *Cuarta Edición* de este Manual de Usuario y Tutorial de ***Java 2***. Abordar la tarea implica una reflexión académica y a la vez profana, pues aunque soy Rector, soy a la vez profesor de Historia. El contenido del libro me permite alegrarme por la relevancia, el rigor y la amenidad en los temas tratados, de ahí su interés en destacar sus cualidades.

En primer lugar, la gran habilidad que tiene el autor para escribir sobre los asuntos más sofisticados y complejos de manera fluida y accesible. El texto se lee (y se entiende) con suma facilidad, sin esfuerzo, está escrito con una ligereza más propia de una buena novela que de un libro que aborda con precisión científica temas de tecnología muy avanzada. Los vocablos, sincretismos y siglas se van introduciendo y explicando progresivamente, con naturalidad, y esa característica es algo que el lector agradece, pues en ningún momento esta fluidez del texto compromete la calidad técnica del mismo.

Otra característica, como no podíamos esperar que fuera de otra forma, es que pone de manifiesto a lo largo de toda la obra la fe y el conocimiento que el autor tiene sobre lo que escribe. Podemos decir que Agustín Froufe es un apasionado del Java, y eso se nota en su obra, pero un apasionado honesto, serio desde un punto de vista tecnológico. No en vano, en los primeros capítulos del libro no escatima comparaciones con los principales rivales de Java en el campo de la programación de propósito general, como C++, *Visual Basic* o *Delphi*. Nos deja claro las grandes ventajas de Java como lenguaje para Internet, principalmente la robustez, neutralidad, portabilidad y seguridad, derivadas de esa genialidad que ha sido introducir el *ByteCode*, que si bien lo hace más lento al tener que ser compilado e interpretado, resuelve otros muchísimos problemas.

Otra característica a destacar de este libro, quizás la fundamental, es que se trata de una obra completa y rigurosa. Una vez introducidos al mundo de Java, especialmente a la versión J2SE de la plataforma Java 2, versión de mayor difusión, nos introduce de manera gradual al lenguaje, empezando por esa otra maravilla que son los *applets* o miniaplicaciones que corren en la mayoría de los navegadores de Internet y continúa con los conceptos y programas básicos, el tratamiento de los ficheros y las excepciones. Pero la obra no finaliza ahí, sino que una vez asentados los niveles medios, da paso a conceptos más avanzados como multitarea, delegación de eventos, AWT, *Swing*, gráficos o los métodos nativos. Finalmente, dedica cuatro capítulos a las aplicaciones de Java en red, comunicaciones, *servlets*, JDBC y RMI. En suma, una obra muy completa.

Pero eso no es todo, casi cada concepto o aplicabilidad de Java está ilustrada con ejemplos útiles y que funcionan, que podemos encontrar descritos tanto en el texto y como su código completo el CD que acompaña al libro, el cual además de contener el código fuente de cada uno de estos ejemplos, se acompaña de numerosas herramientas software y documentación adicional de gran utilidad para el lector usuario.

Resumiendo, podemos decir que se trata de un libro completo, útil para quien desee introducirse o profundizar en el mundo Java, lleno de ejemplos útiles, muy bien escrito, que va de menos a más y que resulta muy ameno de leer sin perder ni un ápice de calidad técnica.

Por todo ello, quiero felicitar al autor, animarle en su tarea, y apoyarle para que persista en dar a conocer a los menos avezados en el tema, contenidos tan sugerentes y tan bien planteados.

Manuel Lobo Cabrera
Rector de la Universidad de Las Palmas de Gran Canaria

Las Palmas de Gran Canaria, mayo de 2005

PREFACIO

Este Tutorial va dirigido a todos aquellos que intentan entrar en el mundo Java, que han oído o leído cosas pero que no saben a ciencia cierta qué es eso de Java, pero sí tienen conocimientos de otros lenguajes de programación.

Java es ya un leguaje de programación maduro, utilizado en los más diversos campos; desde el estudiante para implementar sus prácticas en las que muestra su nombre en la pantalla, hasta los más expertos programadores para desarrollar el software de robots médicos o el de vehículos no tripulados en Marte. Java ha supuesto una revolución. Aunque no fuese una idea escandalosamente genial en sus inicios, ahora ya se ha constituido en un lenguaje muy completo, ha aportado muchas características nuevas a los programadores y ha adaptado muchas de otros lenguajes. A lo largo del Tutorial, el lector podrá ir descubriendo todas estas aportaciones.

La creación de este Tutorial partió en sus inicios de la necesidad de aprendizaje del lenguaje Java para implantarlo en aplicaciones críticas. Se necesitaba una evaluación del lenguaje para comprobar si podía emplearse en el desarrollo de pequeñas aplicaciones (no necesariamente con Internet por medio, aunque también). En aquel momento, hace más de una docena de años, si el interés del programador no se centraba en Internet, Java era simplemente un lenguaje más, incluso con porciones poco definidas o discutibles. Ahora Java es un lenguaje imprescindible, que se utiliza en todos los campos y que todo desarrollador debe conocer.

Este libro tiene su origen en el Tutorial de Java que el autor ha difundido a través de Internet desde hace algunos años a través del Web de la Facultad de Informática de Sevilla, *http://www.cica.es/formacion/JavaTut*. Esta edición es una reescritura completa del libro, en la que se han eliminado las partes correspondientes a los conceptos más básicos de Java, se ha revisado todo el código fuente, para adaptarlo a

la versión actual de la plataforma Java 2, y se han añadido ejemplos adicionales en las partes en que más consultas se han realizado.

En esta quinta edición actualizada del Tutorial se han reescrito algunas de las partes en las que más consultas se han realizado al autor, para extender más los conceptos o aclararlos con nuevos y más representativos ejemplos. Se han eliminado casi todos los ejemplos completos de código en el libro, reproduciendo solamente aquellas partes del código de los ejemplos necesarias para la comprensión de los conceptos a tratar; remitiendo al lector al soporte digital que acompaña al libro. Se han eliminado los capítulos en los que se trataban los temas más básicos de Java, tanto por ser comunes con muchos lenguajes de programación, como por ser temas que el lector puede consultar en muchos de los recursos disponibles en Internet que se indican en el apéndice de Bibliografía.

También se ha actualizado todo el código fuente y referencias al API de Java para adaptarlo a la versión J2SE 6 de la plataforma Java 2. En la compilación y ejecución de los ejemplos, se utiliza la línea de comandos, huyendo del uso de herramientas tipo IDE (*Entorno Integrado de Desarrollo*) como **Netbeans** o **Eclipse**, porque dichas herramientas suponen al lector un aprendizaje previo de dicha herramienta. Si el lector es ya usuario de alguna de ellas, no tendrá dificultad alguna en compilar y ejecutar los ejemplos que se desarrollan en el Tutorial, creando proyectos sobre ellas.

En la conferencia JavaOne desarrollada en San Francisco (EE.UU.) a finales de 1999, *Sun Microsystems* presentó su estrategia en torno a la plataforma Java 2, que se resume en la diversificación de Java en tres grandes ramas, atendiendo al mercado al que va dirigido: grandes ordenadores, ordenadores de sobremesa y microordenadores o dispositivos de memoria limitada. Fruto de esta declaración de intenciones *Sun Microsystems* ha definido el lenguaje Java simplemente como ***Plataforma Java 2*** y ha redistribuido dicha plataforma en tres vertientes, atendiendo al sector al que va dirigida la edición correspondiente, cada una de ellas con su conjunto de APIs y herramientas de desarrollo propias. *Sun Microsystems* distingue pues:

1. **Java 2 Standard Edition** (J2SE), orientada a ordenadores de sobremesa. Comprende el JDK hasta ahora distribuido por *Sun*, en donde **Swing** se ha convertido en pieza clave y al que se han incorporado clases adicionales para facilitar el desarrollo de aplicaciones Java en donde la interfaz de usuario tiene una importancia muy especial. Ésta es la versión en la que la mayoría de la gente habla cuando piensa en Java.

2. **Java 2 Enterprise Edition** (J2EE), engloba al J2SE y lo potencia añadiéndole clases para el desarrollo en entornos corporativos. Esta edición de la plataforma Java 2 está orientada al desarrollo de aplicaciones para servidores utilizando Enterprise JavaBeans, aplicaciones web, Servlets, JavaServer Pages, CORBA y *Extensible Markup Language* (XML). Es decir, esta edición está más orientada al

desarrollo de componentes y distribución de aplicaciones, luciendo toda la parafernalia asociada a las aplicaciones al nivel de negocio.

3. **Java 2 Micro Edition** (J2ME), es un subconjunto de J2SE orientado al desarrollo de aplicaciones Java destinadas a dispositivos con pocos recursos, con capacidades restringidas, tanto con respecto a la capacidad de memoria disponible, limitaciones de la pantalla gráfica como con respecto a la capacidad de procesamiento; características típicas de cualquier equipo de electrónica de consumo; por ejemplo, teléfonos celulares, PDAs, buscapersonas, mensáfonos, dispositivos de navegación de coches, etc.

La mayoría de los programadores Java estarán interesados en la versión J2SE, porque es la que ofrece todas las ventajas de la plataforma Java 2, sin los inconvenientes de dificultad (y alto coste) que suelen acompañar a todo lo que se desarrolla para un entorno empresarial, ni las limitaciones que vienen impuestas por la carencia de recursos de los dispositivos móviles, aunque los desarrollos sobre J2ME están adquiriendo un auge inusitado, de la mano de la facilidad de acceso a dispositivos con dicha tecnología. La versión 1.4 del JDK fue la primera que vio la luz realmente adaptada a esta nueva estrategia de *Sun*, por ello este Tutorial fue adaptado para abrazar esa visión de la plataforma J2SE. En esta edición se han revisado todos los capítulos acerca de las nuevas características incorporadas a J2SE y al lenguaje Java desde la publicación de la edición anterior del Tutorial y se han eliminado otras para las cuales el lector puede encontrar abundante literatura acerca de ellas en muchos recursos en Internet, siendo la fuente de consulta principal el ***Java Tutorial*** de **Sun Microsystems** en donde se describen todas las características pasadas y actuales de Java.

Las características de Java descritas en la anterior edición de este libro siguen siendo válidas, porque la versión 1.6.0 del JDK no es más que la incorporación de algunas características nuevas al lenguaje, como el uso extendido de las anotaciones o la inclusión de lenguaje *script*, pero no representan una revisión completa del lenguaje como fue la versión 1.1 del JDK, en donde se incluían por primera vez las clases anidadas y el nuevo modelo de eventos, o la versión 1.4, en donde se incorporaban las aserciones y expresiones regulares, o la versión 1.5, en la que se añadían al lenguaje los tipos genéricos y algunas de las reclamaciones hechas por los programadores durante años, como el tan famoso *printf()* del lenguaje C, que ya ha pasado a formar parte de Java.

El lector no debe esperar que este libro le proporcione ungüentos milagrosos que por arte de magia traspasen el conocimiento. El estudio de Java, sus *applets*, sus aplicaciones y su funcionamiento a través de este Tutorial no será sino el examen de una particular forma de ver las cosas, con un poco de estructuración en la presentación y un cierto trasfondo de Internet; el resto es, como siempre, tarea del programador. Es decir, uno puede aprender a construir un *applet*, o bien dejar que alguna de las herramientas lo construya automáticamente, igual que puede enseñarse a codificar un

diálogo en un entorno gráfico, pero... la inteligencia de esa pieza siempre dependerá de la habilidad y experiencia del programador respecto del lenguaje usado y de sus recursos. En fin, un buen *applet* o una buena aplicación será únicamente resultado del trabajo de un buen programador Java.

CAPÍTULO 1

INTRODUCCIÓN A JAVA

El uso principal que se hace de Internet e incluso de las redes internas (corporativas) es el del correo electrónico (*e-mail*) y la navegación *Web*, aunque cada vez hay más aplicaciones web corporativas que utilizan Internet como base de comunicación y transmisión de datos. Los documentos Web pueden contener variedad de texto y gráficos de todas clases, así como proporcionar enlaces hipertexto hacia cualquier lugar de la red. Los navegadores utilizan documentos escritos en lenguaje *HTML*. La combinación actual de navegadores HTML/WWW está limitada, pues, a texto y gráficos. Si se quiere reproducir un sonido o ejecutar un programa de demostración, primero hay que descargar (*download*) el fichero en cuestión y luego utilizar un programa en el ordenador propio capaz de entender el formato de ese fichero, o bien cargar un módulo (*plug-in*) en el navegador para que pueda interpretar el fichero que se ha descargado.

Hasta hace bien poco, la única forma de realizar una página Web con contenido interactivo era mediante la interfaz *CGI* (*Common Gateway Interface*), que permite pasar parámetros entre formularios definidos en lenguaje HTML y programas escritos en Perl o en C. Esta interfaz resulta muy incómoda de programar y es pobre en sus posibilidades. Actualmente está en auge el uso de plug-ins *Flash* que permiten incorporar contenido dinámico e interactivo a las páginas web. Sin embargo, Flash no es exactamente un lenguaje de programación y el contenido a presentar depende en gran medida de la herramienta utilizada en el desarrollo.

El lenguaje Java y los navegadores con soporte Java proporcionan una forma diferente de hacer que ese navegador sea capaz de ejecutar programas. Con Java se puede reproducir sonido directamente desde el navegador, se pueden visitar *home pages* con animaciones, se puede *enseñar* al navegador a manejar nuevos formatos de

ficheros, e incluso (a pesar de la mala calidad) se puede transmitir vídeo por las líneas telefónicas, el navegador está preparado para mostrar esas imágenes.

Utilizando Java se pueden eliminar los inconvenientes de la interfaz CGI o las limitaciones de los plug-ins Flash, así como añadir aplicaciones que vayan desde experimentos científicos interactivos de propósito educativo a juegos o aplicaciones especializadas para la *televenta*. Es posible implementar publicidad interactiva y periódicos personalizados. Por ejemplo, alguien podría escribir un programa Java que implementara una simulación química interactiva —una cadena de ADN—. Utilizando un navegador con soporte Java, un usuario podría recibir fácilmente esa simulación e interaccionar con ella en lugar de conseguir simplemente un dibujo estático y algo de texto. Lo recibido cobra vida. Además, con Java el usuario puede estar seguro de que el código que hace funcionar el experimento químico no contiene ningún trozo de código malicioso que dañe al sistema. El código que intente actuar destructivamente o que contenga errores no podrá traspasar los muros defensivos colocados por las características de seguridad y robustez de Java.

Java proporciona una nueva forma de acceder a las aplicaciones. El software viaja transparentemente a través de la red. No hay necesidad de instalar las aplicaciones, ellas mismas vienen cuando se necesitan. Por ejemplo, la mayoría de los navegadores Web pueden procesar un reducido número de formatos gráficos (generalmente GIF, PNG y JPEG). Si se encuentran con otro tipo de formato, el navegador estándar no tiene capacidad para procesarlo, tendría que ser actualizado para poder aprovechar las ventajas del nuevo formato. Sin embargo, un navegador con soporte Java puede enlazar con el servidor que contiene el algoritmo que procesa ese nuevo formato y mostrar la imagen. Por lo tanto, si alguien inventa un nuevo algoritmo de compresión para imágenes, el inventor sólo necesitará estar seguro de que hay una copia en código Java de ese algoritmo instalada en el servidor que contiene las imágenes que quiere publicar. Es decir, los navegadores con soporte Java se actualizan a sí mismos sobre la marcha, cuando encuentran un nuevo tipo de fichero o algoritmo.

En esta filosofía es en la que se basan los **NC** (*Network Computer*), que son ordenadores sin disco y con mucha memoria. Sus programas residen en un servidor que los envía cuando se los solicita. Es quizá un guiño al pasado y una versión futurista de lo que ha sido un *Terminal-X* en otros tiempos, salvando las diferencias, evidentemente (no sea que alguien tilde al autor de irreverente con las nuevas tecnologías).

Java fue el primer lenguaje con la virtud de ser compilado e interpretado de forma simultánea. Cuando un programador realiza una aplicación o un applet en Java y lo compila, en realidad, el compilador no trabaja como un compilador de un lenguaje al uso. El compilador Java únicamente genera el denominado *ByteCode*. Este código es un código intermedio entre el lenguaje máquina del procesador y Java. Evidentemente este código no es ejecutable por sí mismo en ninguna plataforma hardware, pues no se corresponde con el lenguaje de ninguno de los procesadores que actualmente se

conocen (habrá que esperar a ver qué ocurre con los procesadores Java si alguna vez salen al mundo comercial). Por lo tanto, para ejecutar una aplicación Java es necesario disponer de un mecanismo que permita ejecutar el ByteCode. Este mecanismo es la denominada *Máquina Virtual Java* (JVM). En cada plataforma (Solaris, Linux, Windows 95/98/NT/2000/XP/Vista, HP-UX, MacOS, etc.) existe una Máquina Virtual específica. Así que cuando el ByteCode llega a la máquina virtual, ésta lo interpreta pasándolo a código máquina del procesador donde se esté trabajando, así como ejecutando las instrucciones en lenguaje máquina que se deriven de la aplicación Java. De este modo, cuando el mismo ByteCode llega a diferentes plataformas, éste se ejecutará de forma correcta, pues en cada una de esas plataformas existirá la máquina virtual adecuada. Con este mecanismo se consigue la famosa multiplataforma de Java, en la que con sólo codificar una vez, podemos ejecutar en varias plataformas.

En realidad la Máquina Virtual Java (JVM) desempeña otras funciones, como la de aislar los programas Java al entorno de la JVM, consiguiendo una gran seguridad.

Sin embargo, como podrá estar deduciendo el lector, esto tiene algunas desventajas, y la más clara es la velocidad de ejecución. Puesto que la JVM debe estar interpretando constantemente el ByteCode, se consume demasiado tiempo de procesador en realizar esta interpretación, que por otra parte no aporta nada a la aplicación, obteniendo así un bajo rendimiento. Para solucionarlo se han adoptado soluciones intermedias. Una de las más útiles son los compiladores JIT (*Just-In-Time*). Estos compiladores están situados a la entrada de la JVM, de forma que según llega el ByteCode lo van compilando al lenguaje máquina del procesador. A diferencia de la interpretación, el compilador no ejecuta el ByteCode, únicamente lo *traduce* y lo almacena en código nativo dentro de la JVM. Así, una vez que la aplicación está dentro de la JVM, ya se encuentra en lenguaje máquina y, por lo tanto, será directamente ejecutable, sin necesidad de interpretaciones, consiguiendo dotar de mayores rendimientos a la aplicación.

Muy breve y de forma muy genérica, éste es el funcionamiento básico de Java. Todas las mejoras al lenguaje se centran básicamente en conseguir mejores tiempos de ejecución y dotar de mayores prestaciones a la Máquina Virtual Java.

ORIGEN DE JAVA

Sun Microsystems, líder en servidores para Internet, uno de cuyos lemas desde hace mucho tiempo ha sido "*the network is the computer*" (lo que quiere dar a entender que el verdadero ordenador es la red en su conjunto y no cada máquina individual), es quien ha inventado el lenguaje Java, en un intento de resolver simultáneamente todos los problemas que se planteaban a los desarrolladores de software por la proliferación de arquitecturas incompatibles, tanto entre las diferentes máquinas como entre los diversos sistemas operativos y sistemas de ventanas que funcionan sobre una misma máquina, añadiendo la dificultad de crear aplicaciones distribuidas en una red como Internet.

El autor ha podido leer en su día más de cinco versiones distintas sobre el origen, concepción y desarrollo de Java, desde la que dice que éste fue un proyecto que rebotó durante mucho tiempo por distintos departamentos de Sun sin que nadie le prestara ninguna atención, hasta que finalmente encontró su nicho de mercado en la *aldea global* que es Internet; hasta la más difundida, que justifica a Java como lenguaje de pequeños electrodomésticos.

Hace bastantes años *Sun Microsystems* decidió intentar introducirse en el mercado de la electrónica de consumo y desarrollar programas para pequeños dispositivos electrónicos. Tras unos comienzos dudosos, *Sun* decidió crear una filial, denominada **FirstPerson Inc.**, para dar margen de maniobra al equipo responsable del proyecto.

El mercado inicialmente previsto para los programas de *FirstPerson* eran los equipos domésticos: microondas, tostadoras y, fundamentalmente, televisión interactiva. Este mercado, dada la falta de sofisticación de los usuarios, requería unas interfaces mucho más cómodas e intuitivas que los sistemas de ventanas que proliferaban en el momento.

Otros requisitos importantes eran la fiabilidad del código y la facilidad de desarrollo. James Gosling, el miembro del equipo con más experiencia en lenguajes de programación, decidió que las ventajas de eficiencia de C++ (el lenguaje más usado en aquel momento) no compensaban el gran coste de pruebas y depuración. Gosling había estado trabajando en su tiempo libre en un lenguaje de programación que él había llamado *Oak*, según parece el nombre se debe al roble que se veía por la ventana de su casa (¡originalidad ante todo!), el cual, aun partiendo de la sintaxis de C++, intentaba remediar las deficiencias que había observado.

Los lenguajes al uso en aquellos días, como C o C++, debían ser compilados para un chip concreto, y si se cambiaba el chip, todo el software debía compilarse de nuevo. Esto encarecía mucho los desarrollos y el problema era especialmente acusado en el campo de la electrónica de consumo. La aparición de un chip más barato y, generalmente, más eficiente, conduce inmediatamente a los fabricantes a incluirlo en las nuevas series de sus cadenas de producción, por pequeña que sea la diferencia en precio, ya que, multiplicada por la tirada masiva de los aparatos, supone un ahorro considerable. Por tanto, Gosling decidió mejorar las características de Oak y utilizarlo.

El primer proyecto en que se aplicó este lenguaje recibió el nombre de proyecto *Green* y consistía en un sistema de control completo de los aparatos electrónicos y el entorno de un hogar. Para ello se construyó un ordenador experimental denominado ***7** (*Star Seven*). El sistema presentaba una interfaz basada en la representación de la casa de forma animada y el control se llevaba a cabo mediante una pantalla sensible al tacto. En el sistema aparecía *Duke*, la actual mascota de Java. Posteriormente se aplicó a otro proyecto denominado **VOD** (*Video On Demand*) en el que se empleaba como interfaz para la televisión interactiva. Ninguno de estos proyectos se convirtió nunca en un sistema comercial, pero fueron desarrollados enteramente en un Java primitivo y fueron como su bautismo de fuego.

Una vez que en *Sun* se dieron cuenta de que a corto plazo la televisión interactiva no iba a ser un gran éxito, urgieron a *FirstPerson* a desarrollar con rapidez nuevas estrategias que produjeran beneficios. No lo consiguieron y *FirstPerson* cerró en la primavera de 1994.

Pese a lo que parecía ya un olvido definitivo, Bill Joy, cofundador de *Sun* y uno de los desarrolladores principales del Unix de Berkeley, juzgó que Internet podría llegar a ser el campo de juego adecuado para disputar a *Microsoft* su primacía casi absoluta en el terreno del software, y vio en Oak el instrumento idóneo para llevar a cabo estos planes. Tras un cambio de nombre y algunos de diseño, el lenguaje Java fue presentado en sociedad en agosto de 1995. *Sun Microsystems* creó entonces la unidad de negocio **JavaSoft**, dedicada exclusivamente a todo lo referente a Java. Actualmente, esta unidad ha sido absorbida por la propia *Sun*.

No obstante, lo mejor será hacer caso omiso de las historias que pretenden dar carta de naturaleza a la clarividencia industrial de sus protagonistas, porque la cuestión es si independientemente de su origen y entorno comercial, Java ofrece soluciones a las expectativas del lector. Porque tampoco es cuestión de desechar la penicilina aunque su origen haya sido fruto de la casualidad.

CAPÍTULO 2

CARACTERÍSTICAS DE JAVA

Java, al igual que cualquier otro lenguaje de programación, dispone de sus propias particularidades, que representarán una ventaja o una desventaja dependiendo de la aplicación que se vaya a realizar. Por ello, este capítulo está dedicado a presentar las principales características propias de Java y proporcionar una visión desde la perspectiva de desarrollador sobre las expectativas que se colocan sobre Java.

CARACTERÍSTICAS PRINCIPALES DE JAVA

Las características principales que ofrece Java respecto a cualquier otro lenguaje de programación, teniendo muy presentes los lenguajes C y C++ que son su caldo de cultivo, se podrían resumir en las siguientes:

Simple

Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos. C++ no es un lenguaje conveniente por razones de seguridad, pero C y C++ eran los lenguajes más difundidos en el nacimiento de Java, por ello Java se diseñó para ser parecido a C++ y así facilitar un rápido y fácil aprendizaje.

Java elimina muchas de las características de otros lenguajes como C++, para mantener reducidas las especificaciones del lenguaje y añadir características muy útiles como el *garbage collector* (reciclador de memoria dinámica). No es necesario preocuparse de liberar memoria, el reciclador se encarga de ello y como es de baja prioridad, cuando entra en acción, permite liberar bloques de memoria muy grandes, lo que limita mucho la fragmentación de la memoria.

Java reduce en un 50% los errores más comunes de programación con lenguajes como C y C++ al eliminar muchas de las características de éstos.

Orientado a objetos

Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo. Las plantillas de objetos son llamadas *clases* y sus copias, *instancias*. Estas instancias necesitan ser construidas y destruidas en espacios de memoria.

Distribuido

Java se ha construido con extensas capacidades de interconexión TCP/IP. Existen librerías de rutinas para acceder e interactuar con protocolos como *http* y *ftp*. Esto permite a los programadores acceder a la información a través de la red con tanta facilidad como a los ficheros locales.

Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan ser distribuidos, es decir, que se ejecuten en varias máquinas, interactuando.

Robusto

Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores, lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de métodos, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de memoria.

Además, para asegurar el funcionamiento de la aplicación, realiza una verificación de los *ByteCodes*, que son el resultado de la compilación de un programa Java.

Arquitectura neutral

Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto en formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (*run-time*) puede ejecutar ese código objeto, independientemente de la máquina en que ha sido generado. Actualmente existen sistemas run-time para Solaris, SunOs 4.1.x, Windows '95/'98, Windows NT/2000/XP/Vista, Linux, HP-UX, Irix, Aix, MacOS, Apple y probablemente haya grupos de desarrollo trabajando en el *porting* a cualquier otra plataforma, porque el código fuente de Java está accesible.

Seguro

La seguridad en Java tiene dos facetas. En el lenguaje, características como los punteros o el *casting* implícito que hace el compilador de C y C++ se eliminan para prevenir el acceso ilegal a la memoria. Cuando se usa Java para crear un navegador, se combinan las características del lenguaje con protecciones de sentido común aplicadas al propio navegador.

El código Java pasa muchas comprobaciones antes de ejecutarse en una máquina. El código se pasa a través de un verificador de ByteCode que comprueba el formato de los fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal —código que falsea punteros, viola derechos de acceso sobre objetos o intenta cambiar el tipo o clase de un objeto—.

El *Cargador de clases* también ayuda a Java a mantener su seguridad, separando el espacio de nombres del sistema de ficheros local del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo Caballo de Troya, ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior.

Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase, primero se busca en las clases predefinidas (del sistema local) y luego en el espacio de nombres de la clase que hace la referencia. Esto imposibilita que una clase suplante a una predefinida.

Dada, pues, la concepción del lenguaje y si todos los elementos se mantienen dentro del estándar marcado por *Sun*, no hay peligro. En el caso de los applets, Java imposibilita, también, abrir ficheros de la máquina local (siempre que se realizan operaciones con archivos, éstas trabajan sobre el disco duro de la máquina de donde partió el applet), no permite ejecutar ninguna aplicación nativa de una plataforma e impide que se utilicen otros ordenadores como puente, es decir, nadie puede utilizar una máquina para hacer peticiones o realizar operaciones con otra. Además, los intérpretes que incorporan los navegadores Web son aún más restrictivos. Bajo estas condiciones (y dentro de la filosofía de que el único ordenador seguro es el que está apagado, desenchufado, dentro de una cámara acorazada en un búnker y rodeado por mil soldados de los cuerpos especiales del ejército), se puede considerar que Java es un lenguaje seguro y que los applets están libres de virus.

Respecto a la seguridad del código fuente, no ya del lenguaje, el propio JDK proporciona un desensamblador de ByteCode, que hace que cualquier programa pueda ser convertido a código fuente, lo que para el programador significa una vulnerabilidad total a su código. Utilizando *javap* no se obtiene el código fuente original, pero sí desmonta el programa mostrando el algoritmo que se utiliza, que es lo realmente interesante. La protección de los programadores ante esto es utilizar llamadas a programas nativos, externos (incluso en C o C++) de forma que no sea

descompilable todo el código, aunque así se pierda portabilidad. También se puede recurrir al uso de *ofuscadores* de código.

Portable

Más allá de la portabilidad básica por ser de arquitectura independiente, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Los enteros son siempre enteros y, además, enteros de 32 bits en complemento a 2. Además, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que éstas puedan ser implantadas en entornos Unix, Pc o Mac.

Interpretado

El intérprete Java (sistema *run-time*) puede ejecutar directamente el código objeto. Enlazar (*linkar*) un programa normalmente consume menos recursos que compilarlo, por lo que los desarrolladores Java pasarán más tiempo desarrollando y menos esperando por el ordenador. No obstante, el compilador actual del JDK es bastante lento. Por ahora, en que todavía no hay compiladores específicos de Java para las diversas plataformas, Java es más lento que otros lenguajes de programación ya que debe ser interpretado y no ejecutado como sucede en cualquier programa tradicional. No obstante, aunque Java sigue siendo básicamente un lenguaje interpretado, la situación se acerca mucho a la de los programas compilados, sobre todo en lo que a la rapidez en la ejecución del código se refiere.

Multitarea

Al ser Multitarea o Multihilo (o multihilvanado, mala traducción de *multithreaded*), Java permite realizar muchas actividades simultáneas en un programa. El término *multithreaded* es de difícil traducción, aunque actualmente parece ser que la palabra *multitarea* para expresar lo mismo es la que está más comúnmente aceptada. Las tareas —a veces llamadas, procesos ligeros, o hilos de ejecución— son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar estas tareas construidas en el mismo lenguaje, son muy fáciles de usar.

El beneficio de ser multitarea consiste en un mejor rendimiento interactivo y mejor comportamiento en tiempo real. Aunque el comportamiento en tiempo real está limitado a las capacidades del sistema operativo subyacente (Unix, Windows, etc.) de la plataforma, aún supera a los entornos de flujo único de programa (*single-threaded*) tanto en facilidad de desarrollo como en rendimiento.

Dinámico

Java se beneficia todo lo posible de la tecnología orientada a objetos y no intenta conectar todos los módulos que comprenden una aplicación hasta el mismo tiempo de

ejecución. Las librerías nuevas o actualizadas no paralizarán la ejecución de las aplicaciones actuales siempre que mantengan el API anterior.

Difundido

Java, en la actualidad, ya no es un lenguaje recién nacido que solamente se conoce en unos cuantos centros de investigación. Java se ha convertido en el lenguaje más difundido en este momento. Es el lenguaje que cuenta con una legión de programadores más numerosa. Esta circunstancia, independientemente de las características intrínsecas al propio lenguaje, hace que sea muy fácil encontrar documentación, código de ejemplo y muchos otros recursos referentes al lenguaje en Internet, en revistas especializadas, en multitud de libros y, en resumen, en cualquier tipo de lugar o foro de desarrolladores. Además, Java es un lenguaje que se utiliza tanto para grandes aplicaciones empresariales, televisión interactiva, juegos para teléfonos móviles y, en realidad, para cualquier tipo de programa. Esto proporciona una versatilidad al lenguaje que hace que se convierta en la mejor de sus características.

JAVA COMMUNITY PROCESS

Aunque *Sun Microsystems* es la máxima autoridad y tiene la última palabra en todo lo que se refiere a la plataforma Java, gran parte de las especificaciones y extensiones que se incorporan a la plataforma Java se realizan bajo el *Java Community Process* (JCP). Este programa permite que tanto las grandes corporaciones como las empresas pequeñas e incluso personas individuales sean partícipes y tengan decisión en la definición y revisión de las diferentes partes de la plataforma Java y puedan influir en la dirección en la que se dirige Java.

El proceso que se sigue es simple: se crea en primer lugar una petición de especificación, *Java Specification Request* (JSR), con alguna característica nueva o con mejoras a alguna de las partes ya definidas de Java, al objeto de extender la plataforma Java. Si la JSR es aceptada para su desarrollo, se procederá a la creación de un grupo de expertos para definir formalmente esa JSR. Este grupo de expertos está constituido por miembros del JCP con experiencia contrastada en el área cubierta por la JSR, liderado por una de las empresas más relevante del sector al que se dirige la JSR y cualquier otra persona voluntaria que quiera aportar su tiempo y esfuerzo a la comunidad Java. Cuando la definición formal de la especificación está lista, se publica para que sea revisada por el resto de los miembros del JCP y por el público en general. La especificación entonces es modificada con todos los comentarios aportados por quienes la hayan revisado antes de ser votada y aceptada como parte integrante de la plataforma Java cerrándose su desarrollo y desintegrando el grupo de expertos que la gestionó.

Todos los estándares J2SE están definidos en base al JCP. Si el lector desea más información, puede consultar la dirección de Internet: *http://www.jcp.org* en donde

encontrará una lista de todas las JSR que han sido definidas o están en proceso de ser definidas, entre las cuales se incluyen todas las que integran la edición J2SE 6.0 de la plataforma Java y las previstas para su incorporación a posteriores revisiones de la plataforma Java 2.

Como se puede comprobar, *Sun* lidera gran parte del desarrollo de Java pero no tiene la llave de la tecnología, lo cual es mucho más evidente en el mundo de los dispositivos móviles, en el cual hay otras empresas o grupos de desarrollo ajenos a *Sun* que han realizado implementaciones de las especificaciones promulgadas como JSR.

CAPÍTULO 3

PRIMEROS PASOS EN JAVA

Como cualquier otro lenguaje, Java se usa para crear aplicaciones pero, además, tiene la particularidad especial de poder crear aplicaciones muy especiales, son los *applets*, que es una mini (*let*) aplicación (*app*) diseñada para ejecutarse en un navegador. A continuación, se verá en detalle lo mínimo que se puede hacer en ambos casos, lo que permitirá presentar la secuencia de edición, compilación, ejecución en Java, que será imprescindible a la hora de estudiar detalles más concretos de Java, para que los ejemplos que se muestren sean mejor comprendidos.

Hay que hacer una aclaración antes de entrar a ver nada, porque muchos programadores que se introducen en Java piensan solamente en applets, porque les llama la atención en su navegación por Internet. Pero Java va mucho más allá, y no hay por qué establecer una distinción entre applet como *aplicación gráfica* y aplicación independiente como *aplicación de consola*. No hay nada que impida a una aplicación independiente funcionar como una aplicación en modo gráfico, lo único que hay que hacer es tomarse la molestia de inicializar la ventana de la aplicación a mano y añadirle el evento de que se cierre cuando el mensaje que se lo indique le llegue, cosas que en el caso de los applets están a cargo del navegador o visualizador que se esté empleando. Esto es importante, porque en este Tutorial se utilizarán fundamentalmente aplicaciones Java independientes, porque los applets tienen unas medidas de seguridad tan estrictas que no permiten muchas de las opciones del lenguaje, como por ejemplo el acceso a ficheros o la impresión de documentos.

UNA MÍNIMA APLICACIÓN EN JAVA

La aplicación más pequeña posible es la que simplemente imprime un mensaje en la pantalla. Tradicionalmente, el mensaje suele ser "Hola Mundo!". Esto es justamente lo que hace el siguiente fragmento de código:

```
//
// Aplicación HolaMundo de ejemplo
//
class HolaMundoApp {
  public static void main( String args[] ) {
    System.out.println( "Hola Mundo!" ) ;
    }
  }
```

HolaMundo

Hay que ver en detalle la aplicación anterior, línea a línea. Esas líneas de código contienen los componentes mínimos para imprimir *Hola Mundo!* en la pantalla. Es un ejemplo muy simple, que no instancia objetos de ninguna otra clase; sin embargo, accede a una de las clases incluidas en la plataforma Java.

```
//
// Aplicación HolaMundo de ejemplo
//
```

Estas tres primeras líneas son comentarios. Hay tres tipos de comentarios en Java, (`//`) es un comentario orientado a línea.

```
class HolaMundoApp {
```

Esta línea declara la clase **HolaMundoApp**. El nombre de la clase especificado en el fichero fuente se utiliza para crear un fichero *`nombredeclase.class`* en el directorio en el que se compila la aplicación. En este caso, el compilador creará un fichero llamado `HolaMundoApp.class`.

```
    public static void main( String args[] ) {
```

Esta línea especifica el método que el intérprete Java busca para ejecutar en primer lugar. Igual que en otros lenguajes, Java utiliza una palabra clave, `main`, para especificar la primera función a ejecutar. En este ejemplo tan simple no se pasan argumentos.

`public` significa que el método *main()* puede ser llamado por cualquiera, incluyendo el intérprete Java.

`static` es una palabra clave del lenguaje que le dice al compilador que *main* se refiere a la propia clase **HolaMundoApp** y no a ninguna instancia de la clase. De esta forma, si alguien intenta hacer otra instancia de la clase, el método *main()* no se ejecutaría.

`void` indica que *main()* no devuelve nada. Esto es importante, ya que Java realiza una estricta comprobación de tipos, incluyendo aquellos que se ha declarado que devuelven los métodos.

args[] es la declaración de un array de **Strings**. Éstos son los argumentos escritos tras el nombre de la clase en la línea de comandos:

```
% java HolaMundoApp arg1 arg2 ...
```

```
System.out.println( "Hola Mundo!" );
```

Ésta es la funcionalidad de la aplicación. Esta línea muestra el uso de un nombre de clase y método. Se usa el método *println()* de la clase **out** que está en el paquete **System**.

A una variable de tipo **class** se puede acceder sin necesidad de instanciar ningún objeto de esa clase. Por ello ha de ser un tipo básico o primitivo, o bien puede ser una referencia que apunta a otro objeto. La variable **out** es una referencia que apunta a un objeto de otro tipo, en este caso una instancia de la clase **PrintStream** (un objeto **PrintStream**), que es automáticamente instanciado cuando la clase **System** es cargada en la aplicación.

El método *println()* toma una cadena como argumento y la escribe en el *canal* de salida estándar; en este caso, la ventana donde se lanza la aplicación. La clase **PrintStream** tiene un método instanciable llamado *println()*, que permite presentar en la salida estándar del Sistema el argumento que se le pase. En este caso, se utiliza la variable o instancia de **out** para acceder al método.

```
    }
  }
```

Finalmente, se cierran las llaves que limitan el método *main()* y la clase **HolaMundoApp**.

Compilación y Ejecución de HolaMundo

A continuación, se puede ver el resultado de esta primera y sencilla aplicación Java en pantalla. Se genera un fichero con el código fuente de la aplicación, se compila y se ejecuta mediante el intérprete *Java*.

FICHEROS FUENTE JAVA

Los ficheros fuente en Java terminan con la extensión "*.java*". El lector puede crear un fichero utilizando cualquier editor de texto ASCII que tenga como contenido el código de las ocho líneas de la pequeña aplicación, y salvarlo en un fichero con el nombre de `HolaMundoApp.java`. Para crear los ficheros con código fuente Java no es necesario un procesador de textos, sino que es suficiente con cualquier editor que tenga salida a fichero de texto plano o ASCII, aunque actualmente se utilizan los Entornos Integrados de Desarrollo que proporcionan editores específicos para Java con características de edición avanzadas, como coloración de sintaxis, autocompletado de sentencias, compilación al vuelo, ayuda interactiva, etc.

COMPILACIÓN

El compilador ***javac*** se encuentra en el directorio *bin* por debajo del directorio donde se haya instalado la plataforma Java 2. Este directorio *bin*, si se han seguido las instrucciones de instalación, debería formar parte de la variable de entorno `PATH` del sistema. Si no es así, el lector tendría que revisar la *Instalación del JDK*. El compilador de Java traslada el código fuente Java a byte-codes, que son los componentes que entiende la *Máquina Virtual Java* que está incluida en los navegadores con soporte Java y en *appletviewer*.

Una vez creado el fichero fuente `HolaMundoApp.java`, se puede compilar introduciendo la siguiente línea de comandos:

```
% javac HolaMundoApp.java
```

Si no se han cometido errores al teclear ni se han tenido problemas con el `PATH` a la hora de especificar el directorio del fichero fuente y del compilador, no debería aparecer mensaje alguno en la pantalla, y cuando vuelva a aparecer el *prompt* del sistema, se debería ver un fichero `HolaMundoApp.class` nuevo en el directorio donde se encuentra el fichero fuente.

Si aparece algún problema, en *Problemas de compilación* en la sección siguiente, se muestran los que más frecuentemente se suelen dar, y que el lector puede consultar por si pueden aportar un poco de luz al error que haya aparecido.

EJECUCIÓN

Para ejecutar la aplicación **HolaMundoApp**, se recurre al intérprete ***java***, que también se encuentra en el directorio *bin*, bajo el directorio en donde se haya instalado la plataforma Java 2 (alias JDK). Se ejecutará la aplicación con la línea:

```
% java HolaMundoApp
```

y debería aparecer en pantalla la respuesta de Java:

```
% Hola Mundo!
```

El símbolo `%` representa al *prompt* del sistema, y se utilizará aquí para presentar las respuestas que devuelva el sistema como resultado de la ejecución de los comandos que se introduzcan por teclado o para indicar las líneas de comandos a introducir por pantalla.

Cuando se ejecuta una aplicación Java, el intérprete Java busca e invoca el método *main()* de la clase cuyo nombre coincide con el nombre del fichero `.class` que se indique en la línea de comandos. En el ejemplo se indica al Sistema Operativo que arranque el intérprete Java y luego se indica a éste que busque y ejecute el método *main()* de la aplicación Java almacenada en el fichero `HolaMundoApp.class`.

Problemas de compilación

A continuación, se encuentra una lista de los errores más frecuentes que se presentan a la hora de compilar un fichero con código fuente Java, tomando como base los errores provocados sobre la mínima aplicación Java que se está utilizando como ejemplo, aunque también podría generalizarse sin demasiados problemas.

```
% javac: Command not found
```

No se ha establecido correctamente la variable `PATH` del sistema para el compilador *javac*. El compilador *javac* se encuentra en el directorio *bin*, que cuelga del directorio donde se haya instalado la plataforma Java, o JDK (*Java Development Kit*), según se le quiera llamar.

```
% HolaMundoApp.java:3: Method printl(java.lang.String) not found in class
java.io.PrintStream.
    System.out.printl( "HolaMundo! );
               ^
```

Error tipográfico, el método es *println* no `printl`.

```
% In class HolaMundoApp: main must be public and static
```

Error de ejecución, se olvidó colocar la palabra `static` en la declaración del método *main()* de la aplicación.

```
% Can´t find class HolaMundoApp
```

Éste es un error muy sutil. Generalmente significa que el nombre de la clase es distinto al del fichero que contiene el código fuente, con lo cual el fichero *nombre_fichero.class* que se genera es diferente del que cabría esperar. Por ejemplo, si en el fichero de código fuente de la aplicación `HolaMundoApp.java` se coloca en vez de la declaración actual de la clase **HolaMundoApp**, la línea:

```
class HolaMundoapp {
```

se creará un fichero `HolaMundoapp.class`, que es diferente del `HolaMundoApp.class`, que es el nombre esperado de la clase; la diferencia se encuentra en la **a** minúscula y mayúscula.

```
% Note: HolaMundoApp.java uses a deprecated API. Recompile with "-
deprecation" for details. 1 Warning
```

Esto es originado por otra de las novedades que ha introducido la plataforma Java a partir del JDK 1.1 como son los elementos obsoletos (*deprecated*), es decir, aquellas clases o métodos que no se recomienda utilizar, aunque sigan siendo válidos, porque están destinados a desaparecer de la faz de la Tierra a partir de alguna de las versiones posteriores de la plataforma Java. Si se compila un programa que hace uso de una de estas clases, o bien utiliza o sobrecarga un método obsoleto, el compilador mostrará un mensaje de este tipo.

Solamente se genera un aviso por módulo, independientemente del número de métodos obsoletos que se estén utilizando, por eso hay que seguir la recomendación del aviso si se quieren saber los detalles completos de todas las clases y métodos obsoletos que se están utilizando. La llamada a estos métodos rara vez tiene excusa, aunque haya casos especiales en que se escriba código para que sea llamado tanto por programas generados con la versión 1.0 del JDK como con versiones posteriores. En este caso *Sun* recomienda indicar al compilador que se está haciendo uso intencionado del método obsoleto, y esto se consigue colocando el comentario `/** @deprecated */` justo antes del método sobrecargado, por ejemplo:

```
/** @deprecated */
public boolean handleEvent( Event evt ) {
  if( evt.id == Event.WINDOW_DESTROY )
    System.exit( 0 );
  return( false );
  }
```

No obstante, en este caso y a pesar de estos avisos, el compilador genera código perfectamente ejecutable.

UN APPLET BÁSICO EN JAVA

A continuación, se trata de crear el código fuente de un applet que sea sencillo y reproducir el ciclo de desarrollo de escritura del código fuente Java, compilación, ejecución y visualización de resultados de un applet, es decir, un applet que *presente el mismo mensaje de saludo en la pantalla*.

La programación de applets Java difiere significativamente de la programación de aplicaciones Java, aunque se puedan compatibilizar. Se puede prever que cuando haya necesidades de una interfaz gráfica, la programación de applets será bastante más sencilla que la programación de aplicaciones que realicen los mismos cometidos, aunque los applets necesiten de un visualizador de ficheros HTML con soporte Java para poder ejecutarse; y hay que tener en cuenta las restricciones a que están sometidos.

HolaMundo

Ahora se presenta el código fuente del applet *HolaMundo*, que es la versión applet de la mínima aplicación Java que antes se ha creado y desarrollado antes. Este código se guardará en un fichero fuente Java como `HolaMundo.java`, y que como todos los ejemplos de código que se incorporan en este Tutorial, se pueden encontrar ya pre-escritos y se pueden cargar directamente en el navegador (los *.html*) o compilar (los *.java*), a partir del soporte digital que acompaña al libro.

```
import java.awt.Graphics;
import java.applet.Applet;

public class HolaMundo extends Applet {
```

```
public void paint( Graphics g ) {
  // Pinta el mensaje en la posición indicada
  g.drawString( "Hola Mundo!",25,25 ) ;
  }
}
```

Componentes básicos de un applet

El lenguaje Java implementa un modelo de Programación Orientada a Objetos. Los objetos sirven de bloques centrales de construcción de los programas Java. De la misma forma que otros lenguajes de programación, Java tiene variables de estado y métodos.

La descomposición de un applet en sus piezas/objetos sería la que se muestra a continuación:

```
/*
Sección de importaciones
*/

public class NombreDelNuevoApplet extends Applet {
  /*
  Aquí se declaran las variables de estado (public y private)
  */

  /*
  Los métodos para la interacción con los objetos se
  declaran y definen aquí
  */
  public void MetodoUno( parámetros ) {
    /*
    Aquí viene para cada método, el código Java que
    desempeña la tarea.
    Qué código se use depende del applet
    */
    }
  }
```

Para **HolaMundo** se importan las dos clases que necesita. No hay variables de estado, y sólo se tiene que definir un método para que el applet tenga el comportamiento esperado.

CLASES INCLUIDAS

El comando `import` carga otras clases dentro del código fuente. El importar una clase desde un paquete de Java hace que esa clase importada esté disponible para todo el código incluido en el fichero fuente Java que la importa. Por ejemplo, en el applet *HolaMundo* que se está presentando aquí, se importa la clase **java.awt.Graphics**, y se podrá llamar a los métodos de esta clase desde cualquiera de los métodos que se encuentren incluidos en el fichero `HolaMundo.java`.

Esta clase define un área gráfica y métodos para poder dibujar dentro de ella. La función, método, *paint()* declara a `g` como un objeto de tipo **Graphics**; luego, *paint()* usa el método *drawString()* de la clase **Graphics** para generar su salida.

LA CLASE APPLET

Se puede crear una nueva clase, en este caso **HolaMundo**, extendiendo la clase básica de Java: **Applet**. De esta forma, se hereda todo lo necesario para crear un applet. Modificando determinados métodos del applet, se puede lograr que lleve a cabo las funciones que se deseen.

```
import java.applet.Applet;
. . .
public class HolaMundo extends Applet {
```

MÉTODOS DE APPLET

La parte del applet a modificar es el método *paint()*. En la clase **Applet**, se llama al método *paint()* cada vez que el método arranca o necesita ser refrescado, pero no hace nada. En este caso, del applet básico que se está gestando, lo que se hace es:

```
public void paint( Graphics g ) {
  g.drawString( "Hola Mundo!",25,25 ) ;
  }
```

De acuerdo a las normas de sobrecarga, se ejecutará este último *paint()* y no el *paint()* vacío de la clase **Applet**. Luego aquí se ejecuta el método *drawString()*, que le dice al applet cómo debe aparecer un texto en el área de dibujo.

Compilación de un applet

Ahora que ya está el código del applet básico escrito y el fichero fuente Java que lo contiene guardado en disco, es necesario compilarlo y obtener un fichero `.class` ejecutable a través del intérprete de Java. Se utiliza el compilador Java, *javac*, para realizar la tarea, y el comando de compilación a usar será:

```
% javac HolaMundo.java
```

Eso es todo. El compilador *javac* generará un fichero `HolaMundo.class` que podrá ser llamado desde cualquier navegador con soporte Java y, por tanto, capaz de ejecutar applets Java.

Llamada a applets

¿Qué tienen de especial Firefox, Mozilla, Opera, Internet Explorer o Netscape Communicator con respecto a otros navegadores? Con ellos se puede ver código

escrito en lenguaje HTML básico y acceder a todo el texto, gráfico, sonido e hipertexto que se pueda ver con cualquier otro navegador.

Pero además, y esto es lo que tienen de especial, pueden ejecutar applets, que no es HTML estándar. Estos navegadores entienden código HTML que lleve la marca `<APPLET>`:

```
<APPLET CODE="MiCodigo.class" WIDTH=100 HEIGHT=50>
</APPLET>
```

Esta marca HTML llama al applet `MiCodigo.class` y establece su anchura y altura inicial. Cuando se acceda a la página Web donde se encuentre incluida la marca, se ejecutará el ByteCode contenido en `MiCodigo.class`, obteniéndose el resultado de la ejecución del applet en la ventana del navegador, con soporte Java, que se esté utilizando.

Si no se dispone de ningún navegador, se puede utilizar el visor de applets que proporciona Sun con la plataforma Java 2, el *appletviewer*, que además requiere muchos menos recursos de la máquina en que se esté ejecutando que cualquier otro de los navegadores que se acaban de citar. Para el desarrollador, en el momento de probar sus applets, resulta mucho más cómodo el uso del visor *appletviewer* que el de un navegador.

Prueba de un applet

La ejecución de un applet sobre *appletviewer* se realiza a través de la llamada:

```
% appletviewer fichero.html
```

En este caso el fichero con el código HTML que ejecutará el applet *HolaMundo* es `HolaMundo.html` que generará la salida que muestra la figura 3.1.

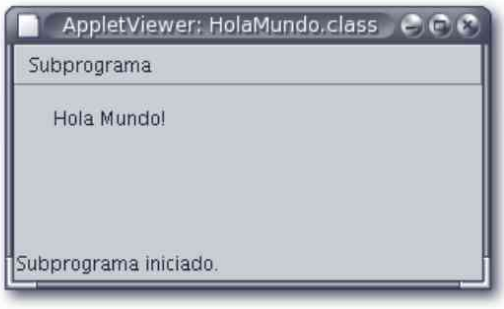


Figura 3.1

ARGUMENTOS EN LA LÍNEA DE COMANDOS

Al ejecutar aplicaciones, se pueden especificar argumentos en la línea de llamada, y si los programas están preparados para aceptarlos, podrán tomarlos en cuenta. Los

usuarios del sistema DOS o de Linux, estarán muy familiarizados con comandos del tipo

```
copy ficheroA ficheroB
```

En donde `copy` es el nombre del programa y `ficheroA` y `ficheroB` son los argumentos de la línea de comandos.

Sin embargo, Java, a diferencia de otros lenguajes como C++, trata de forma diferente los argumentos que se pasan en la línea de llamada. La sintaxis utilizada por Java es la que se ha visto en la miniaplicación inicial desarrollada al comienzo de este capítulo:

```
public static void main( String args[] ) {
  . . .
  }
```

En el lenguaje Java los parámetros siempre deben aparecer en la declaración del método *main()*, aunque el programa Java no esté preparado para aceptar ninguno.

El nombre del programa no se proporciona en Java y el número de elementos en el array se puede obtener mediante el atributo *length* del array, con lo que no es necesario que el sistema operativo indique cuántos parámetros hay en la línea de comandos.

Solamente como curiosidad, y también como muestra de una aplicación un poco más complicada que la impresión de un mensaje, el siguiente ejemplo, `Java301.java`, imprime la lista de argumentos que se pasa a una aplicación Java.

```
class Java301 {
  public static void main( String args[] ) {
    for ( int i=0; i < args.length; i++ )
      System.out.println( args[i] );
    }
  }
```

Si se compila y, posteriormente, ejecuta esta aplicación con la llamada que se indica, lo que se obtiene en pantalla es lo que se muestra a continuación:

```
% javac Java301.java
% java Java301 estos son argumentos
estos
son
argumentos
```

CAPÍTULO 4

LENGUAJE JAVA

Ahora que ya se ha visto a grandes rasgos lo que Java puede ofrecer, y tras la generación del primer código Java, se supone que el gusanillo de comenzar con Java ya estará satisfecho, por lo que es el momento de entrar a pelearse con el lenguaje en sí. En el aprendizaje de todo lenguaje de programación el primer paso suele ser entrar en conocimiento de los conceptos fundamentales, como son las variables, los tipos de datos, expresiones, flujo de control, etc. Por ello, en este capítulo se tratarán estos conceptos. Lo básico resultará muy familiar a quienes tengan conocimientos de C/C++ o cualquier lenguaje orientado a objetos. Los programadores con experiencia en otros lenguajes procedurales reconocerán la mayor parte de las construcciones.

El lector encontrará en los apéndices recursos para poder profundizar en cada uno de los conceptos que se introducen o describen en éste y posteriores capítulos. Sin embargo, como referencia del lenguaje, a continuación se comentan los que son fundamentales para Java.

Cuando se programa en Java, se coloca todo el código en métodos, de la misma forma que se escriben funciones en otros lenguajes como C.

COMENTARIOS

En Java hay tres tipos de comentarios:

```
// comentarios para una sola línea

/* comentarios de una o
    más líneas
*/
```

```
/** comentario de documentación, de una o más líneas
*/
```

Los dos primeros tipos de comentarios son los que todo programador conoce y se utilizan del mismo modo. Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java, *javadoc*, no disponible en otros lenguajes de programación. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación de las clases que se va construyendo al mismo tiempo que se genera el código de la aplicación.

La plataforma Java 2 permite la creación de *doclets*, que son pequeñas aplicaciones que la herramienta *javadoc* puede ejecutar para generar cualquier tipo de documentación especial que se desee. A pesar de la posible complejidad de los *doclets*, la salida estándar es un fichero HTML con el mismo formato que el resto de la documentación de Java, por lo que resulta muy fácil navegar a través de las clases.

IDENTIFICADORES

Los identificadores nombran variables, funciones, clases y objetos; cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de dólar ($). Los siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay una longitud máxima establecida para el identificador. La forma básica de una declaración de variable, por ejemplo, sería:

```
tipo identificador [ = valor][,identificador [= valor] ...];
```

Serían identificadores válidos:

```
identificador
nombre_usuario
Nombre_Usuario
_variable_del_sistema
$transaccion
```

y su uso sería, por ejemplo:

```
int contador_principal;
char _lista_de_ficheros;
float $cantidad_en_Euros;
```

Palabras clave

Las siguientes son las palabras clave que están definidas en Java y que no se pueden utilizar como identificadores:

```
abstract    continue    for         new         switch
assert      default     goto        null        synchronized
boolean     do          if          package     this
break       double      implements  private     threadsafe
byte        else        import      protected   throw
byvalue     enum        instanceof  public      transient
case        extends     int         return      true
catch       false       interface   short       try
char        final       long        static      void
class       finally     native      super       while
const       float
```

Palabras reservadas

Además, el lenguaje se reserva unas cuantas palabras más, pero que hasta ahora no tienen un cometido específico. Son:

```
cast        future      generic     inner
operator    outer       rest        var
```

Literales

Un valor constante en Java se crea utilizando una representación literal de él. Java utiliza cinco tipos de elementos: *enteros*, *reales en coma flotante*, *booleanos*, *caracteres* y *cadenas*, que se pueden poner en cualquier lugar del código fuente de Java. Cada uno de estos literales tiene un tipo correspondiente asociado con él.

Enteros:

`byte`	8 bits		complemento a dos
`short`	16 bits		complemento a dos
`int`	32 bits		complemento a dos
`long`	64 bits		complemento a dos
Por ejemplo:	`221`	`077`	`0xDC00`

Reales en coma flotante:

`float`	32 bits		IEEE 754
`double`	64 bits		IEEE 754
Por ejemplo:	`3.14`	`2e12`	`3.1E12`

Booleanos:

```
true
false
```

Caracteres:

Por ejemplo: `a` `\t` `\u???? [????] número unicode`

Cadenas:

Por ejemplo: `"Esto es una cadena literal"`

Cuando se inserta un literal en un programa, el compilador normalmente sabe exactamente de qué tipo se trata. Sin embargo, hay ocasiones en la que el tipo es ambiguo y hay que guiar al compilador proporcionándole información adicional para indicarle exactamente de qué tipo son los caracteres que componen el literal que se va a encontrar. En el ejemplo siguiente se muestran algunos casos en que resulta imprescindible indicar al compilador el tipo de información que se le está proporcionando:

```
class Literales {
    char c = 0xffff;    // mayor valor de char en hexadecimal
    byte b = 0x7f;      // mayor valor de byte en hexadecimal
    short s = 0x7fff;   // mayor valor de short en hexadecimal
    int i1 = 0x2f;      // hexadecimal en minúsculas
    int i2 = 0X2F;      // hexadecimal en mayúsculas
    int i3 = 0177;      // octal (un cero al principio)
    long l1 = 100L;
    long l2 = 100l;
    long l3 = 200;
    // long l4(200);    // no está permitido
    float f1 = 1;
    float f2 = 1F;
    float f3 = 1f;
    float f4 = 1e-45f;  // en base 10
    float f5 = 1e+9f;
    double d1 = 1d;
    double d2 = 1D;
    double d3 = 47e47d; // en base 10
    }
```

Un valor hexadecimal (base 16), que funciona con todos los tipos enteros de datos, se indica mediante un `0x` o `0X` seguido por `0-9` y `a-f`, bien en mayúsculas o minúsculas. Los números octales (base 8) se indican colocando un cero a la izquierda del número que se desee. No hay representación para números binarios. Si se intenta inicializar una variable con un valor mayor que el que puede almacenar, el compilador generará un error.

Se puede colocar un carácter al final del literal para indicar su tipo, ya sea una letra mayúscula o minúscula. La letra **L** se usa para indicar un `long`, la letra **F** significa `float` y la letra **D** mayúscula o minúscula es lo que se emplea para indicar que el literal es un `double`.

La exponenciación se indica con la letra `e`, tomando como referente la base 10. Es decir, que hay que realizar una traslación mental al ver estos números de tal forma que **1.3e-45f** en Java, en la realidad es $1.3x10^{-45}$.

No es necesario indicarle nada al compilador cuando se puede conocer el tipo sin ambigüedades. Por ejemplo, en:

```
long l3 = 200;
```

no es necesario colocar la L después de 200 porque resultaría superflua. Sin embargo, en el caso:

```
float f4 = 1e-45f;  // en base 10
```

sí es necesario indicar el tipo, porque el compilador trata normalmente los números exponenciales como `double`, por lo tanto, si no se coloca la **f** final, el compilador generará un error indicando que se debe colocar un *moldeador* para convertir el `double` en `float`.

Cuando se realizan operaciones matemáticas o a nivel de bits con tipos de datos básicos más pequeños que `int` (`char`, `byte` o `short`), esos valores son promocionados a `int` antes de realizar las operaciones y el resultado es de tipo `int`. Si se quiere seguir teniendo el tipo de dato original, hay que colocar un moldeo, teniendo en cuenta que al pasar de un tipo de dato mayor a uno menor, es decir, al hacer un *moldeo estrecho*, se puede perder información. En general, el tipo más grande en una expresión es el que determina el tamaño del resultado de la expresión; si se multiplica un `float` por un `double`, el resultado será un `double`, y si se suma un `int` y un `long`, el resultado será un `long`.

SEPARADORES

Sólo hay un par de secuencias con otros caracteres que pueden aparecer en el código Java; son los separadores simples, que van a definir la forma y función del código. Los separadores admitidos en Java son:

() - *paréntesis*. Para contener listas de parámetros en la definición y llamada a métodos. También se utilizan para definir precedencia en expresiones, contener expresiones para control de flujo y rodear las conversiones de tipo.

{} - *llaves*. Para contener los valores de matrices inicializadas automáticamente. También se utilizan para definir un bloque de código, para clases, métodos y ámbitos locales.

[] - *corchetes*. Para declarar tipos matriz. También se utilizan cuando se referencian valores de matriz.

; - *punto y coma*. Separa sentencias.

, - *coma*. Separa identificadores consecutivos en una declaración de variables. También se utiliza para encadenar sentencias dentro de una sentencia `for`.

. - *punto*. Se emplea para separar los nombres de paquetes, de subpaquetes y de clases. También se utiliza a la hora de separar una variable o método de una variable de referencia.

OPERADORES

Java proporciona un conjunto de operadores para poder realizar acciones sobre uno o dos operandos. Un operador que actúa sobre un solo operando es un operador **unario**, y un operador que actúa sobre dos operandos es un operador **binario**.

Algunos operadores pueden funcionar como unarios y como binarios, el ejemplo más claro es el operador - (signo menos). Como operador binario, el signo menos hace que el operando de la derecha sea sustraído al operando de la izquierda; como operador unario hace que el signo algebraico del operando que se encuentre a su derecha sea modificado.

En la siguiente tabla aparecen los operadores que se utilizan en Java, por orden de precedencia:

```
      .      []     ()
      ++     --
      !      ~      instanceof
*     /      %
      +      -
      <<     >>     >>>
      <      >      <=     >=     ==     !=
      &      ^      |
      &&     ||
      ? :
      =      op=    (*=    /=     %=     +=     -=     etc.)  ,
```

Los operadores numéricos se comportan como es de esperar:

```
int + int = int
```

Los operadores relacionales devuelven un valor booleano.

Para las cadenas, se pueden utilizar los operadores relacionales para comparaciones además de + y += para la concatenación:

```
String nombre = "nombre" + "Apellido";
```

El operador = siempre hace copias de objetos, marcando los antiguos para que sean borrados, encargándose el *garbage collector* de devolver al sistema la memoria ocupada por el objeto eliminado.

Java, a diferencia de otros lenguajes, no soporta la sobrecarga de operadores. Esto significa que no es posible redefinir el entorno en el que actúa un operador con

respecto a los objetos de un nuevo tipo que el programador haya definido como propios.

Operadores aritméticos

Java soporta varios operadores aritméticos que actúan sobre números enteros y números en coma flotante. Los operadores **binarios** soportados por Java son:

- `+` suma los operandos
- `-` resta el operando de la derecha al de la izquierda
- `*` multiplica los operandos
- `/` divide el operando de la izquierda entre el de la derecha
- `%` resto de la división del operando izquierdo entre el derecho

Como se ha indicado anteriormente, el operador *más* (+) se puede utilizar para concatenar cadenas, como se observa en el ejemplo siguiente:

```
"miVariable tiene el valor " + miVariable + " en este programa"
```

Esta operación hace que se tome la representación como cadena del valor de `miVariable` para su uso exclusivo en la expresión. De ninguna forma se altera el valor que contiene la variable.

El operador *módulo* (%), que devuelve el resto de una división, funciona con tipos en coma flotante además de con tipos enteros. Cuando se ejecuta el programa que se muestra en el siguiente ejemplo, `Java401.java`:

```
class Java401 {
  public static void main( String args[] ) {
    int x=33;
    double y=33.3;
    System.out.println( "x mod 10 = " + x%10 );
    System.out.println( "y mod 10 = " + y%10 );
    }
  }
```

la salida que se obtiene por pantalla es:

```
% java Java401
x mod 10 = 3
y mod 10 = 3.299999999999997
```

Los operadores **unarios** que soporta Java son:

- `+` indica un valor positivo
- `-` negativo, o cambia el signo algebraico
- `++` suma 1 al operando, como prefijo o sufijo
- `--` resta 1 al operando, como prefijo o sufijo

En los operadores de *incremento* (++) y *decremento* (--), en la versión prefijo, el operando se coloca a la derecha del operador, ++x; mientras que el operando aparece a la izquierda del operador, x++, en la versión sufijo. La diferencia entre estas versiones es el momento en el tiempo en que se realiza la operación representada por el operador si éste y su operando aparecen en una expresión larga. Con la versión *prefijo*, la variable se incrementa (o decrementa) antes de que sea utilizada para evaluar la expresión en que se encuentre, mientras que en la versión *sufijo*, se utiliza la variable para realizar la evaluación de la expresión y luego se incrementa (o decrementa) en una unidad su valor.

Operadores relacionales y condicionales

> el operando izquierdo es mayor que el derecho
>= el operando izquierdo es mayor o igual que el derecho
< el operando izquierdo es menor que el derecho
<= el operando izquierdo es menor o igual que el derecho
== el operando izquierdo es igual que el derecho
!= el operando izquierdo es distinto del derecho

Los operadores relacionales combinados con los operadores condicionales se utilizan para obtener expresiones más complejas. Los operadores condicionales que soporta el lenguaje Java son:

&& expresiones izquierda y derecha son `true`
|| o la expresión izquierda o la expresión de la derecha son `true`
! la expresión de la derecha es `false`

Los operandos de estos operadores deben ser tipos booleanos, o expresiones que devuelvan un tipo booleano.

Una característica importante del funcionamiento de los operadores && y ||, que es pasada a veces por alto incluso por programadores experimentados, es que las expresiones se evalúan de izquierda a derecha y que la evaluación de la expresión finaliza tan pronto como se pueda determinar el valor de la expresión. Por ejemplo, en la expresión siguiente:

```
( a < b ) || ( c < d )
```

si la variable a es menor que la variable b, no hay necesidad de evaluar el operando izquierdo del operador || para determinar el valor de la expresión entera. En casos de complicadas, complejas y largas expresiones, el orden en que se realizan estas comprobaciones puede ser fundamental, y cualquier error en la colocación de los operandos puede dar al traste con la evaluación que se desea realizar, estos errores son harto difíciles de detectar, ya que se debe estudiar concienzudamente el resultado de las expresiones en tiempo de ejecución para poder detectar el problema y no siempre es fácil hacer pasar la ejecución del código por todas las condiciones en las que se generen este tipo de errores.

Operadores a nivel de bits

Tanto Java como otros lenguajes comparten un conjunto de operadores que realizan operaciones sobre un solo bit cada vez. Java también soporta el operador >>>, no disponible en otros lenguajes y, además, el entorno en el que se realizan algunas de las operaciones no siempre es el mismo en Java que en los otros lenguajes.

Los operadores de bits que soporta Java son los que aparecen en la siguiente tabla:

Operador	Uso	Operación
>>	Operando >> Despl	Desplaza bits del operando hacia la derecha las posiciones indicadas (con signo)
<<	Operando << Despl	Desplaza bits del operando hacia la izquierda las posiciones indicadas
>>>	Operando >>> Despl	Desplaza bits del operando hacia la derecha las posiciones indicadas (sin signo)
&	Operando & Operando	Realiza una operación AND lógica entre los dos operandos
|	Operando | Operando	Realiza una operación OR lógica entre los dos operandos
^	Operando ^ Operando	Realiza una operación lógica OR Exclusiva entre los dos operandos
~	~Operando	Complementario del operando (unario)

En Java, el operador de desplazamiento hacia la derecha sin signo rellena los bits que pueden quedar vacíos con ceros. Los bits que son desplazados fuera del entorno se pierden.

En el desplazamiento a la izquierda, hay que ser precavidos cuando se trata de desplazar enteros positivos pequeños, porque el desplazamiento a la izquierda tiene el efecto de multiplicar por 2 para cada posición de bit que se desplace; y esto es peligroso porque si se desplaza un bit 1 a la posición más alta, el bit 31 o el 63, el valor se convertirá en negativo.

Operadores de asignación

El operador = es un operador binario de asignación de valores. El valor almacenado en la memoria y representado por el operando situado a la derecha del operador es copiado en la memoria indicada por el operando de la izquierda. Ni el operador de asignación, ni ningún otro, se pueden sobrecargar.

Java soporta toda la panoplia de operadores de asignación que se componen con otros operadores para realizar la operación que indique ese operador y luego asignar el valor obtenido al operando situado a la izquierda del operador de asignación. De este modo se pueden realizar dos operaciones con un solo operador.

```
+=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=   >>>=
```

Por ejemplo, las dos sentencias que siguen realizan la misma función:

```
x += y;
x = x + y;
```

Y las otras comprobaciones siguen el mismo patrón.

Operador ternario if-then-else

El lenguaje Java soporta este operador **ternario**. Aunque la construcción utilizada por este operador es algo confusa, se puede llegar a entender perfectamente cuando uno lee en el pensamiento lo que está escrito en el código. La forma general del operador es:

```
expresion ? sentencia1 : sentencia2
```

en donde `expresion` puede ser cualquier expresión de la que se obtenga como resultado un valor booleano, y si es `true` entonces se ejecuta la `sentencia1` y en caso contrario se ejecuta la `sentencia2`. La limitación que impone el operador es que `sentencia1` y `sentencia2` deben devolver el mismo tipo, y éste no puede ser `void`.

Puede resultar útil para evaluar algún valor que seleccione una expresión a utilizar, como en la siguiente sentencia:

```
cociente = denominador == 0 ? 0 : numerador / denominador
```

Cuando Java evalúa la asignación, primero mira la expresión que está a la izquierda del interrogante. Si `denominador` es cero, entonces evalúa la expresión que está entre el interrogante y los dos puntos, y se utiliza como valor de la expresión completa. Si `denominador` no es cero, entonces evalúa la expresión que está después de los dos puntos y se utiliza el resultado como valor de la expresión completa, que se asigna a la variable que está a la izquierda del operador de asignación, `cociente`.

En el ejemplo `Java402.java`, se utiliza este operador para comprobar que el denominador no es cero antes de evaluar la expresión que divide por él, devolviendo un cero en caso contrario. Hay dos expresiones, una que tiene un denominador cero y otra que no.

```
class Java402 {
  public static void main( String args[] ) {
    int a = 28;
    int b = 4;
    int c = 45;
    int d = 0;

    // Utilizamos el operador ternario para asignar valores a las dos
    // variables e y f, que son resultado de la evaluación realizada
    // por el operador
```

```
    int e = (b == 0) ? 0 : (a / b);
    int f = (d == 0) ? 0 : (c / d);
    // int f = c / d;

    System.out.println( "a = " + a );
    System.out.println( "b = " + b );
    System.out.println( "c = " + c );
    System.out.println( "d = " + d );
    System.out.println();
    System.out.println( "a / b = " + e );
    System.out.println( "c / d = " + f );
    }
  }
```

El programa se ejecuta sin errores, y la salida que genera por pantalla es:

```
% java Java402
a = 28
b = 4
c = 45
d = 0

a / b = 7
c / d = 0
```

Si ahora se cambia la línea que asigna un valor a la variable `f`, y se elimina este operador, dejándola como:

```
int f = c / d;
```

se producirá una excepción en tiempo de ejecución de división por cero, deteniéndose la ejecución del programa en esa línea.

```
% java Java402
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Java402.main(Java402.java:42)
```

Moldeo de operadores

Es lo que se conoce como ***casting***, refiriéndose a "*colocar un molde*". Java automáticamente convierte el tipo de un dato en otro cuando es pertinente. Por ejemplo, si se asigna un valor entero a una variable declarada como flotante, el compilador convierte automáticamente el `int` a `float`. El *casting*, o moldeo, permite hacer esto explícitamente, o forzarlo cuando normalmente no se haría.

Para realizar un moldeo, se coloca el tipo de dato que se desea (incluyendo todos los modificadores) dentro del paréntesis a la izquierda del valor. Por ejemplo:

```
void moldeos() {
  int i = 100;
  long l = (long)i;
  long l2 = (long)200;
  }
```

Como se puede observar, es posible realizar el moldeo de un valor numérico del mismo modo que el de una variable. No obstante, en el ejemplo el moldeo es superfluo, porque el compilador ya promociona los enteros a flotantes, cuando es necesario, automáticamente, sin necesidad de que se le indique. Aunque hay otras ocasiones en que es imprescindible colocar el moldeo para que el código compile.

Java permite el moldeo de cualquier tipo primitivo en otro tipo primitivo, excepto en el caso de los booleanos, en que no se permite moldeo alguno. Tampoco se permite el moldeo de clases. Para convertir una clase en otra hay que utilizar métodos específicos. La clase **String** es un caso especial que se verá más adelante.

VARIABLES

Las variables se utilizan en la programación Java para almacenar datos que varían durante la ejecución del programa. Para usar una variable, hay que indicarle al compilador el tipo y nombre de esa variable, *declaración de la variable*. El tipo de la variable determinará el conjunto de valores que se podrán almacenar en la variable y el tipo de operaciones que se podrán realizar con ella.

Por ejemplo, el tipo `int` solamente puede contener números completos (enteros). En Java, todas las variables de tipo `int` contienen valores con signo.

EXPRESIONES

Los programas en Java se componen de *sentencias*, que a su vez están compuestas en base a *expresiones*. Una expresión es una determinada combinación de operadores y operandos que se evalúan para obtener un resultado particular. Los operandos pueden ser variables, constantes o llamadas a métodos.

Una llamada a un método evalúa el valor devuelto por el método y el tipo de una llamada a un método es el tipo devuelto por ese método.

Java soporta *constantes* con nombre y la forma de crearlas es:

```
final float PI = 3.14159265358979323846;
```

Esta línea de código produce un valor que se puede referenciar en el programa, pero no puede ser modificado. La palabra clave `final` es la que evita que esto suceda.

En ciertos casos, el orden en que se realizan las operaciones es determinante para el resultado que se obtenga. Al igual que en C++, Pascal y otros lenguajes, en Java se puede controlar el orden de evaluación de las expresiones mediante el uso de paréntesis. Si no se proporcionan estos paréntesis, el orden estará determinado por la precedencia de operadores, de tal modo que las operaciones en las que estén involucrados operadores con más alta precedencia serán los que primero se evalúen.

ARRAYS

Java cuenta con un tipo de datos verdadero de posiciones secuenciales, *array*, que dispone de comprobaciones exhaustivas para su correcta manipulación; por ejemplo, de la comprobación de sobrepasar los límites definidos para el array, en evitación de desbordamiento o corrupción de memoria.

En Java hay que declarar un array antes de poder utilizarlo. Y en la declaración hay que incluir el nombre del array y el tipo de datos que se van a almacenar en él. La sintaxis general para declarar e instanciar un array es:

```
tipoDeElementos[] nombreDelArray =
  new tipoDeElemetos[tamañoDelArray]
```

Se pueden declarar en Java arrays de cualquier tipo:

```
char s[];
int iArray[];
```

Incluso se pueden construir arrays de arrays:

```
int tabla[][] = new int[4][5];
```

Al igual que los demás objetos en Java, la declaración del array no localiza, o reserva, memoria para contener los datos. En su lugar, simplemente localiza memoria para almacenar una referencia al array. La memoria necesaria para almacenar los datos que componen el array se buscará en memoria dinámica a la hora de instanciar y crear realmente el array. Cuando se crea un array, Java lo inicializa automáticamente, utilizando `false` para un array que almacene objetos **boolean**, `null` para un array de objetos de tipo **Object**, o el equivalente a `0` en cualquier otro caso.

Para crear un array en Java hay dos métodos básicos: se puede crear un array vacío:

```
int lista[] = new int[50];
```

o bien crear ya el array con sus valores iniciales:

```
String nombres[] = {
  "Juan","Pepe","Pedro","Maria"
  };
```

lo que es equivalente a:

```
String nombres[];
nombres = new String[4];
nombres[0] = new String( "Juan" );
nombres[1] = new String( "Pepe" );
nombres[2] = new String( "Pedro" );
nombres[3] = new String( "Maria" );
```

No se pueden crear arrays estáticos en tiempo de compilación:

```
int lista[50]; // generará un error en tiempo de compilación
```

Tampoco se puede rellenar un array sin declarar el tamaño con el operador new:

```
int lista[];
for( int i=0; i < 9; i++ )
  lista[i] = i;
```

En las sentencias anteriores simultáneamente se declara el nombre del array y se reserva memoria para contener sus datos. Sin embargo, no es necesario combinar estos procesos. Se puede ejecutar la sentencia de declaración del array y posteriormente, otra sentencia para asignar valores a los datos.

Una vez que se ha instanciado un array, se puede acceder a los elementos de ese array utilizando un índice, de forma similar a la que se accede en otros lenguajes de programación. Los índices de un array siempre empiezan por 0.

Todos los arrays en Java disponen del atributo: *length*, que se puede utilizar para conocer la longitud del array.

```
int a[][] = new int[10][3];
a.length;            /* 10 */
a[0].length;         /*  3 */
```

El ejemplo `Java403.java` intenta ilustrar un aspecto interesante del uso de arrays en Java. Se trata de que Java puede crear arrays multidimensionales, que se verían como arrays de arrays. Aunque ésta es una característica común a muchos lenguajes de programación en Java, sin embargo, los arrays secundarios no necesitan tener todos el mismo tamaño. En el ejemplo se declara e instancia un array de enteros, de dos dimensiones, con un tamaño inicial (tamaño de la primera dimensión) de 3. Los tamaños de las dimensiones secundarias (tamaño de cada uno de los subarrays) son de 2, 3 y 4, respectivamente.

En este ejemplo también se puede observar el hecho de que cuando se declara un array de dos dimensiones no es necesario indicar el tamaño de la dimensión secundaria a la hora de la declaración del array, sino que puede declararse posteriormente el tamaño de cada uno de los subarrays. También se puede observar el resultado de acceder a elementos que se encuentran fuera de los límites del array; Java protege la aplicación contra este tipo de errores de programación.

En lenguajes como C o C++, cuando se intenta acceder y almacenar datos en un elemento del array que está fuera de límites, siempre se consigue, aunque los datos se almacenen en una zona de memoria que no forme parte del array. En Java, si se intenta hacer esto, el sistema generará una excepción, tal como se muestra en el ejemplo. En él, la excepción simplemente hace que el programa termine, pero se podría capturar

esa excepción e implementar un controlador de excepciones (*exception handler*) para corregir automáticamente el error, sin necesidad de abortar la ejecución del programa.

La salida por pantalla de la ejecución del ejemplo sería:

```
% java Java403
00
012
0246
Acceso a un elemento fuera de limites
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at Java403.main(Java403.java:55)
```

y el código del ejemplo es el que se reproduce a continuación:

```
class Java403 {
  public static void main( String args[] ) {
    // Declaramos un array de dos dimensiones con un tamaño de 3 en la
    // primera dimensión y diferentes tamaños en la segunda
    int miArray[][] = new int[3][];  // No especificamos el tamaño de
                                     // la segunda dimensión
    miArray[0] = new int[2]; // El tamaño de la segunda dimensión es 2
    miArray[1] = new int[3]; // El tamaño de la segunda dimensión es 3
    miArray[2] = new int[4]; // El tamaño de la segunda dimensión es 4
    // Rellenamos el array con datos
    for( int i=0; i < 3; i++ ) {
      for( int j=0; j < miArray[i].length; j++ )
        miArray[i][j] = i * j;
      }
    // Visualizamos los datos que contiene el array
    for( int i=0; i < 3; i++ ) {
      for( int j=0; j < miArray[i].length; j++ )
        System.out.print( miArray[i][j] );
      System.out.println();
     }
    // Intentamos acceder a un elemento que se encuentre
    // fuera de los límites del array
    System.out.println( "Acceso a un elemento fuera de limites" );
    miArray[4][0] = 7;
    // Esa sentencia originará el lanzamiento de una excepción de
    // tipo ArrayIndexOutOfBounds
    }
  }
```

El ejemplo `Java404.java` ilustra otro aspecto del uso de arrays en Java. Se trata, en este caso, de que Java permite la asignación de un array a otro. Sin embargo, hay que extremar las precauciones cuando se hace esto, porque lo que en realidad está pasando es que se está haciendo una copia de la referencia a los mismos datos en memoria. Y tener dos referencias a los mismos datos no parece ser una buena idea, porque los resultados pueden despistar.

STRINGS

Un **String** o cadena se considera a toda secuencia de caracteres almacenados en memoria y accesibles como una unidad. Java implementa cadenas a través de las clases **String** y **StringBuffer**, a las que se dedicará un amplio estudio.

TIPOS ENUMERADOS

El patrón para crear tipos enumerados desde el comienzo de Java estaba implementado en base a tipos `int`, tal como se muestra a continuación:

```
public class Estaciones {
  public static final int PRIMAVERA = 0;
  public static final int VERANO = 0;
  public static final int OTONO = 0;
  public static final int INVIERNO = 0;
  }
```

Esta aproximación presenta una serie de problemas, como son el no permitir comprobación de tipos, no pertenecer a un espacio de nombres definido, las constantes han de ser compiladas en los clientes y, sobre todo, los valores que imprimen no proporcionan ninguna información útil.

Un tipo enumerado es un tipo cuyos valores son un conjunto determinado de constantes. Las enumeraciones proporcionan a Java tipos enumerados, que definen clases representando un único elemento enumerado. Estas clases no proporcionan ningún constructor público, solamente métodos estáticos y finales. Permiten la comprobación de tipos en tiempo de compilación y, debido a que son objetos, se pueden incorporar a colecciones y recabar cualquier tipo de campos y métodos. Es decir, resuelven todas las desventajas del patrón `int` y además añaden nuevas características. Utilizando los tipos enumerados, es posible escribir el siguiente código:

```
  static public enum Estacion {
    Primavera, Verano, Otono, Invierno
    };
```

En este ejemplo se introduce la palabra clave `enum` para declarar los tipos enumerados que limitan el conjunto de valores con tipos definidos, permitiendo incluso su utilización en la sentencia `switch`. Es decir, ya no sólo se pueden utilizar constantes enteras, sino que pueden emplearse clases reales. La aplicación `Java405.java` imprime las cuatro estaciones del año y su código es el siguiente:

```
class Java405 {
  static public enum Estacion {
    Primavera, Verano, Otono, Invierno
    };
  public static void main( String args[] ) {
    for( Estacion e : Estacion.values() )
      System.out.println( "La estacion es "+e );
```

```
    }
  }
```

La ejecución de esta aplicación genera la salida que se muestra a continuación:

```
% java Java405
La estacion es Primavera
La estacion es Verano
La estacion es Otono
La estacion es Invierno
```

Como puede observar el lector, la salida por pantalla no muestra números sin información alguna, sino que aparece la descripción correspondiente al tipo enumerado.

Todos los tipos enumerados implementan el método estático *values()* para devolver todos los valores de la enumeración en una colección, tal como mostraba el bucle `for` del ejemplo anterior.

También implementan el método *valueOf()*, que devuelve la constante de la enumeración que corresponde a un parámetro de tipo **String**. Por ejemplo, la invocación a *estacion.valueOf("Primavera")* devolverá `Estacion.Privamera`.

Hay otros métodos útiles a la hora de manejar tipos enumerados. El método *name()* devuelve el nombre de la constante en la enumeración, tal como se haya declarado; por ejemplo, la invocación *Estacion.Verano.name()* devolverá `Verano`. El método *ordinal()* devuelve la posición de la constante declarada, siendo la inicial 0; por ejemplo, *Estacion.Verano.ordinal()* devolverá `1`. Finalmente, el método *compareTo()* compara el objeto actual con el objeto de la enumeración que se especifique, devolviendo un número negativo, cero o un número positivo dependiendo de que el objeto actual sea menor, igual o mayor que el objeto especificado de la enumeración, respectivamente. La comparación se basa en el ordinal correspondiente a la posición de la declaración en la enumeración; por ejemplo,

```
Estacion.Verano.compareTo( Estacion.Invierno );
```

Devolverá -2, porque `Verano` está dos posiciones más atrás que `Invierno` en la declaración de la enumeración.

La aplicación `Java406.java` muestra un ejemplo más elaborado que presentará un mensaje más explicativo sobrescribiendo el método *toString()* y añadiendo un método para el control de los colores que integran la bandera de Canarias.

ANOTACIONES

Las *anotaciones* corresponden a un concepto introducido en J2SE 5.0. Ofrecen una forma de asociar *metadatos* con elementos de programación, como pueden ser las

clases, interfaces o métodos. Se pueden considerar como modificadores adicionales, aunque no cambien el *ByteCode* que se genera para dichos elementos.

El concepto de *metadato* no es nuevo en Java. Desde el principio se pueden utilizar etiquetas `@deprecated` en los comentarios de los métodos para que *javadoc* y el compilador traten esta etiqueta como dato adicional acerca de un método. El concepto de anotación es el mismo, al menos en lo que corresponde a la parte "`@`" de la etiqueta anterior. Sin embargo, la colocación o el lugar de posicionamiento sí ha cambiado; ahora estas etiquetas no solamente se pueden colocar en los comentarios, sino que las anotaciones se pueden incluir en el código fuente. La característica principal que añaden las anotaciones es que proporcionan una forma sistemática para soportar el modelo de programación declarativa.

La primera anotación que se incluyó en J2SE 5 fue `@Deprecated` (observe el lector la D mayúscula para diferenciarla de la anotación usada por *javadoc*). La funcionalidad de esta etiqueta en el código es la misma que `@deprecated` en los comentarios. Si se asigna `@Deprecated` a un método o a una clase, se estará indicando al compilador que proporcione un aviso al usuario cuando utilice ese método o esa clase.

La clase **Obsoleto** proporciona el método *metodoDeprecated()*, al que se ha asignado la etiqueta `@Deprecated` y se ha incluido `@deprecated` en los comentarios.

```
public class Obsoleto {
  /**
   * @deprecated Obsoleto. Usar System.metodo() en su lugar.
   */
  @Deprecated
  public static void metodoDeprecated() {
    System.out.println( "No me invoques.." );
    }
  }
```

La clase con anotaciones se compila de la misma forma que una clase sin anotaciones. Si ahora se crea una nueva clase, **Java411**, desde la que se invoca el método anterior, el compilador generará un aviso en tiempo de compilación, de la misma forma que hace el uso de la etiqueta `@deprecated` en *javadoc*.

```
public class Java411 {
  public static void main( String args[] ) {
    Obsoleto.metodoDeprecated();
    }
  }
```

Si se compila el fichero `Java411.java`, la salida por pantalla que genera el compilador será:

```
> javac Java411.java
Note: Java411.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Y si al comando de compilación se añade la opción `-Xlint:deprecation`, el compilador indicará de forma específica qué es lo que sucede:

```
> javac -Xlint:deprecation Java411.java
Java411.java:34: warning: [deprecation] metodoDeprecated() in Obsoleto has
been deprecated
    Obsoleto.metodoDeprecated();
             ^
1 warning
```

El cambio de utilizar la etiqueta `@deprecated` en los comentarios a la anotación `@Deprecated` no introduce nada nuevo en el lenguaje, sino que es otra forma de hacer lo mismo. Sin embargo, otras anotaciones como `@Override` o `@SupressWarnings` sí que incorporan nuevas funcionalidades a la plataforma Java.

La anotación `@Override` se puede utilizar en las declaraciones de los métodos para indicar que un método sobrescribe al método correspondiente de su superclase. La razón de su uso es que los errores aparezcan cuanto antes. Es frecuente a la hora de sobrescribir un método que se cometan errores tipográficos o que se especifiquen argumentos erróneos o que se devuelva un tipo de dato distinto del que devuelve el método que se intenta sobrescribir. Es decir, que no es infrecuente que en vez de sobrescribir un método, en realidad se esté creando un método nuevo, lo que hará que sea muy difícil encontrar el problema que ocasiona su presencia, en tiempo de ejecución. El uso de `@Override` permite que el compilador pueda comprobar que el nuevo método sobrescribe al de la superclase y presentar el error, si lo hubiera, ya en tiempo de compilación.

```
public class Java412 {
  @Override
  public int hashcode() {
    return 0;
    }
  @Override
  public boolean equals( Object obj ) {
    return true;
    }
  }
```

En el código anterior, el problema que presenta es que la signatura del método debería ser *hash**C**ode()* y no *hash**c**ode()*. Si la declaración anterior estuviese (como es habitual) incluida en el código fuente de una clase mucho mayor, sería muy complicado darse cuenta del error que genera la circunstancia de que *hashcode()* tenga su nombre todo en minúsculas, con lo cual, no está sobrescribiendo al método *hashCode()*, sino que se trata de un método nuevo. Utilizando la anotación `@Override`, a la hora de compilar la clase se genera el aviso acerca del problema:

```
> javac Java412.java
Java412.java:31: method does not override or implement a method from a
supertype

  @Override
  ^
1 error
```

La anotación `@SuppressWarnings` quizás sea aún más interesante porque permite indicar al compilador que no proporcione avisos sobre cosas que normalmente sí generarían avisos. El compilador define varias opciones de supresión de avisos, en función de sus tipos: *all*, *deprecation*, *unchecked*, *fallthrough*, *path*, *serial* y *finally*.

Para mostrar un ejemplo, en el siguiente código se utiliza la anotación anterior para suprimir los avisos de tipo *fallthrough*. Si el lector observa la clase, apreciará la ausencia de la sentencia `break` en cada una de las opciones `switch`.

```
public class Java413 {
  public static void main( String args[] ) {
    int i = args.length;
    switch( i ) {
      case 0:
        System.out.println( "0" );
      case 1:
        System.out.println( "1" );
      case 2:
        System.out.println( "2" );
      case 3:
        System.out.println( "3" );
      default:
        System.out.println( "Default" );
      }
    }
  }
```

Si se compila la clase con el comando:

```
> javac Java413.java
```

la salida por pantalla no indica ningún problema. Si se quiere que el compilador avise de que las cláusulas `switch` caen en un *fallthrough*, es decir, que faltan una o varias sentencias break, se debe incluir la opción `-Xlint:fallthrough` en el comando de compilación:

```
> javac -Xlint:fallthrough Java413.java
```

que si se ejecuta, ahora generará la siguiente salida por pantalla:

```
Java413.java:39: warning: [fallthrough] possible fall-through into case
      case 1:
      ^
Java413.java:41: warning: [fallthrough] possible fall-through into case
      case 2:
      ^
```

```
Java413.java:43: warning: [fallthrough] possible fall-through into case
      case 3:
      ^
Java413.java:45: warning: [fallthrough] possible fall-through into case
      default:
      ^
4 warnings
```

Pero en el caso de que se quieran ignorar los avisos anteriores, es decir, ignorar el hecho de que no haya sentencias `break`, es donde entra en juego la anotación `@SuppressWarnings`, que en este caso se puede añadir en el método *main()* de la forma:

```
@SuppressWarnings("fallthrough")
```

y se compila con el comando anterior con la opción `Xlint`, entonces se generará el fichero *.class* correspondiente a la clase sin indicar ningún aviso en la salida.

El lector debe usar esta anotación con juicio; es decir, no debe utilizarse para que el resultado de la compilación aparezca limpio, sino para evitar que en la salida aparezca todo tipo de información cuando se está resolviendo un tipo de problema concreto. De este modo, se pueden evitar todos los avisos que no interesen o no deban tenerse en cuenta en ese instante y solamente contribuyen a complicar la salida.

CONTROL DE FLUJO

El control del flujo es la manera que tiene un lenguaje de programación de hacer que el flujo de la ejecución avance y se ramifique en función de los cambios de estado de los datos. Java, en este aspecto, no utiliza los principios de diseño orientado a objetos, sino que las sentencias de control del flujo del programa se han tomado del C/C++. A continuación, se tratan todos los mecanismos que proporciona Java para conseguir este control y decidir qué partes del código ejecutar.

Sentencias de salto

if/else

```
if( expresión-booleana ) {
  sentencias;
  }
[else {
  sentencias;
  }]
```

Esta construcción hace que el programa atraviese un conjunto de estados booleanos que determinan la ejecución de distintos fragmentos de código. La cláusula `else` es opcional. Cada una de las *sentencias* puede ser una compuesta y la *expresión-booleana* podría ser una variable simple declarada como *boolean*, o bien una

expresión que utilice operadores relacionales para generar el resultado de una comparación.

switch

```
switch( expresión ) {
  case valor1:
    sentencias;
    break;
  case valor2:
    sentencias;
    break;
  [default:
    sentencias;]
  }
```

La sentencia `switch` proporciona una forma limpia de enviar la ejecución a partes diferentes del código en base al valor de una única variable o expresión. La *expresión* puede devolver cualquier tipo básico, y cada uno de los *valores* especificados en las sentencias `case` debe ser de un tipo compatible.

La sentencia `switch` funciona de la siguiente manera: el valor de la *expresión* se compara con cada uno de los *valores* literales de las sentencias `case`. Si coincide con alguno, se ejecutará el código que sigue a la sentencia `case`. Si no coincide con ninguno de ellos, entonces se ejecutará la sentencia `default` (por defecto), que es opcional. Si no hay sentencia `default` y no coincide con ninguno de los valores, no hará nada. Al igual que en otros lenguajes, cada constante en una sentencia `case` debe ser única.

El uso de la cláusula `default`, a pesar de ser opcional, siempre debería utilizarse, porque en el caso de desarrollo de aplicaciones en entornos colaborativos, algún programador puede haber cambiado el conjunto de valores utilizado en la sentencia `switch`, sin haber recordado actualizar las cláusulas `case`, en cuyo caso la aplicación no funcionará correctamente, siendo difícil detectar un error tan sutil. Por ello, es importante utilizar la cláusula `default` indicando qué valor de `switch` ha hecho llegar ahí el flujo de la aplicación.

El compilador de Java inspeccionará cada uno de los valores que pueda tomar la *expresión* en base a las sentencias `case` que se proporcionen, y creará una tabla eficiente que utiliza para ramificar el control del flujo al `case` adecuado dependiendo del valor que tome la *expresión*. Por lo tanto, si se necesita seleccionar entre un gran grupo de *valores*, una sentencia `switch` se ejecutará mucho más rápido que la lógica equivalente codificada utilizando sentencias `if-else`.

La palabra clave `break` se utiliza habitualmente en sentencias `switch` sin etiqueta, para que la ejecución salte tras el final de la sentencia `switch`. Si no se pone el `break`, la ejecución continuará en el siguiente `case`. A la hora de programar es un error habitual olvidar un `break`; dado que el compilador no avisa de dichas omisiones, es

aconsejable poner un comentario en los sitios en los que normalmente se pondría el `break`, diciendo que la intención es que el `case` continúe en el siguiente, por convenio este comentario es simplemente, "*Continúa*". También se puede hacer uso de la opción `Xlint` del compilador que en este caso generaría avisos de tipo *fallthrough*.

Sentencias de bucle

Bucles for

```
for( inicialización; terminación; iteración ) {
  sentencias;
  }
```

Un bucle `for`, normalmente, involucra a tres acciones en su ejecución:

- Inicialización de la variable de control
- Comprobación del valor de la variable de control en una expresión condicional
- Actualización de la variable de control

La cláusula de *inicio* y la cláusula de *incremento* pueden estar compuestas por varias expresiones separadas mediante el operador *coma* (,), que en estos bucles Java también soporta:

```
for( a=0,b=0; a < 7; a++,b+=2 )
```

El operador *coma* garantiza que el operando de su izquierda se ejecutará antes que el operando de su derecha. Las expresiones de la cláusula de *inicio* se ejecutan una sola vez, cuando arranca el bucle. Cualquier expresión legal se puede emplear en esta cláusula, aunque generalmente se utiliza para inicialización. Las variables se pueden declarar e inicializar al mismo tiempo en esta cláusula:

```
for( int cnt=0; cnt < 2; cnt++ )
```

La segunda cláusula, de comprobación o *test*, consiste en una única expresión que debe evaluarse a `false` para que el bucle concluya. Esta segunda expresión debe ser de tipo booleano, de tal modo que se pueden utilizar únicamente expresiones relacionales o expresiones relacionales y condicionales. El valor de la segunda cláusula se comprueba cuando la sentencia comienza la ejecución y en cada una de las iteraciones posteriores.

La tercera cláusula, de *incremento*, aunque aparece físicamente en la declaración de bucle, no se ejecuta hasta que se han ejecutado todas las sentencias que componen el cuerpo del bucle `for`; por ello, se utiliza para actualizar la variable de control. Es importante tener en cuenta que si se usan variables incluidas en esta tercera cláusula en las sentencias del cuerpo del bucle, su valor no se actualizará hasta que la ejecución de todas y cada una de las sentencias del cuerpo del bucle se haya completado. En esta

cláusula pueden aparecer múltiples expresiones separadas por el operador *coma*, que serán ejecutadas de izquierda a derecha.

La primera y tercera cláusulas del bucle `for` pueden encontrarse vacías, pero deben estar separadas por punto y coma (`;`). Hay autores que sugieren incluso que la cláusula de testeo puede estar vacía, aunque salvando el caso de que se trate de implementar un bucle infinito, si esta cláusula de comprobación se encuentra vacía, el método de terminación del bucle no es nada obvio, al no haber una expresión condicional que evaluar, por lo que debería recurrirse a otro tipo de sentencia en vez de utilizar un bucle `for`.

La forma que se acaba de presentar del bucle `for` es muy eficiente y poderosa, pero no está optimizada cuando se trata de recorrer arrays y colecciones, porque cuando se recorre una colección de datos hay que utilizar un objeto de tipo **Iterator** para poder recorrerla, no sirviendo este objeto para ningún otro cometido. Con el uso de las clases genéricas, que se presentarán seguidamente, para indicar el tipo de las colecciones, el uso de iteradores podría evitarse. Por ello, a partir de la versión J2SE 5, el lenguaje Java incorpora otra forma de utilización del bucle `for` en la cual no es necesario indicar el índice para recuperar un elemento, haciendo mucho más claro el código.

La forma del bucle for ahora puede ser tal como se muestra a continuación:

```
for( parámetro : expresión ) sentencias
```

donde `expresión` debe ser un array o una instancia de la interfaz **Iterable**, que permite el uso de esta nueva forma del bucle `for`. Todas las colecciones extienden esta interfaz. Por ejemplo, si se quieren recuperar los valores de un array, se puede hacer de la siguiente forma clásica:

```
int impares[] = {1,3,5,7,9,11,13 };
for( int i=0; i < 7; i++ )
  System.out.println( "Numero impar: "+impares[i] );
```

y con la nueva forma del bucle `for` introducida por J2SE 5.0, se puede escribir el mismo código de forma más clara:

```
for( int num : impares )
  System.out.println( "Numero impar: "+num );
```

Como se ha indicado, esta nueva forma del bucle `for` también se puede utilizar con colecciones, tal como se mostrará en capítulos posteriores. De este modo, es posible prescindir del uso de iteradores que son una fuente de errores, porque aparecen al menos tres veces en cada bucle y pueden generar fallos en tiempo de ejecución muy difíciles de detectar. Para evitarlo, se deja al compilador que realice todo el trabajo, tal como se muestra en el código siguiente:

```
Vector<Integer> v new Vector<Integer>;  // colección genérica
v.addElement( 1 );  // conversiones implícitas
```

```
v.addElement( 2 );
v.addElement( 3 );
for( Integer i : v )
  System.out.println( "Numero: "+i );
```

Se crea un iterador de tipo **Integer** que recorre todos los valores de la colección de tipo **Vector**. El iterador `i` es local al bucle `for` y la nueva sintaxis se encarga de actualizar el iterador y realizar la conversión a **Integer**. La clase **Java407** utiliza las dos formas del bucle `for` para recorrer la colección que se pasa como argumento a cada uno de los métodos de la clase.

```
class Java407 {
  // Método que utiliza la forma clásica de recorrer una colección
  // utilizando un iterador para ello
  private void forAntiguo( Collection c ) {
    System.out.println( "For al estilo de siempre.." );
    for( Iterator i = c.iterator(); i.hasNext(); ) {
      String elemento = (String)i.next();
      System.out.print( elemento );
      }
    }

  // Método que utiliza la nueva forma del bucle for para recorrer la
  // colección de semejante manera que el método anterior
  private void forNuevo( Collection c ) {
    System.out.println( "For en formato nuevo.." );
    for( Object obj : c ) {
      System.out.print( obj );
      }
    }

  public static void main( String args[] ) {
    Java407 af = new Java407();
    // Creamos una colección y utilizando clases genéricas,
    // restringimos su contenido a elementos de tipo String
    ArrayList<String> lista = new ArrayList<String>();
    // Incorporamos algunos elementos a la lista
    lista.add( "Tutorial " );
    lista.add( "de " );
    lista.add( "Java " );
    lista.add( "adaptado " );
    lista.add( "al " );
    lista.add( "J2SE " );
    lista.add( "6" );
    // Ahora invocamos a los métodos que nos presentan el contenido
    // de la lista
    af.forAntiguo( lista );
    System.out.println();
    af.forNuevo( lista );
    }
  }
```

Hay que tener en cuenta, no obstante, que esta nueva forma del bucle `for` no puede utilizarse cuando se recorre una colección para modificarla; por ejemplo, para eliminar elementos.

Bucles while

```
[inicialización;]
while( terminación-expresión-booleana ) {
  sentencias;
  [iteración;]
  }
```

El bucle `while` es la sentencia de bucle más básica en Java. Ejecuta repetidamente una vez tras otra una sentencia mientras una expresión booleana sea verdadera. Las partes de inicialización e iteración, que se presentan entre corchetes, son opcionales.

Esta sentencia `while` se utiliza para crear una *condición de entrada*. El significado de esta condición de entrada es que la expresión condicional que controla el bucle se comprueba antes de ejecutar cualquiera de las sentencias que se encuentran situadas en el interior del bucle, de tal modo que si esta comprobación es `false` la primera vez, el conjunto de las sentencias no se ejecutará nunca.

Bucles do/while

```
[inicialización;]
do {
  sentencias;
  [iteración;]
  } while( terminación-expresión-booleana );
```

A veces, se puede desear el ejecutar el cuerpo de un bucle `while` al menos una vez, incluso si la expresión booleana de terminación tiene el valor `false` la primera vez; es decir, si se desea evaluar la expresión de terminación al final del bucle en vez de al principio como en el bucle `while`. Esta construcción `do-while` hace eso exactamente.

Excepciones

try-catch-throw

```
try {
  sentencias;
} catch( Exception ) {
  sentencias;
  }
```

Java implementa excepciones para facilitar la construcción de código robusto. Cuando ocurre un error en un programa, el código que encuentra el error lanza una excepción, que se puede capturar y recuperarse de ella. Java proporciona muchas excepciones predefinidas.

El manejo y control de excepciones es tan importante en Java que debe ser tratado en un capítulo aparte, aunque sirva este punto como mención previa de su existencia.

Control general del flujo

break

```
break [etiqueta];
```

La sentencia `break` puede utilizarse en una sentencia `switch` o en un bucle. Cuando se encuentra en una sentencia `switch`, `break` hace que el control del flujo del programa pase a la siguiente sentencia que se encuentre fuera del entorno del `switch`. Si se encuentra en un bucle, hace que el flujo de ejecución del programa deje el ámbito del bucle y pase a la siguiente sentencia que venga a continuación del bucle.

Java incorpora la posibilidad de etiquetar la sentencia `break`, de forma que el control pasa a sentencias que no se encuentran inmediatamente después de la sentencia `switch` o del bucle, es decir, saltará a la sentencia en donde se encuentre situada la etiqueta. La sintaxis de una sentencia etiquetada es la siguiente:

```
etiqueta: sentencia;
```

continue

```
continue [etiqueta];
```

La sentencia `continue` no se puede utilizar en una sentencia `switch`, sino solamente en bucles. Cuando se encuentra esta sentencia en el discurrir normal de un programa Java, la iteración en que se encuentre el bucle finaliza y se inicia la siguiente.

Java permite el uso de etiquetas en la sentencia `continue`, de forma que el funcionamiento normal se ve alterado y el salto en la ejecución del flujo del programa se realizará a la sentencia en la que se encuentra colocada la etiqueta.

Por ejemplo, ante la presencia de bucles anidados, se pueden utilizar etiquetas para poder salir de ellos:

```
uno: for( ) {
  dos: for( ) {
    continue;      // seguiría en el bucle interno
    continue uno;  // seguiría en el bucle principal
    break uno;     // se saldría del bucle principal
    }
  }
```

Hay autores que sugieren que el uso de sentencias `break` y `continue` etiquetadas proporciona una alternativa al infame `goto` (que Java no soporta, aunque reserva la palabra). Quizá sea así, pero el entorno de uso de las sentencias etiquetadas `break` y `continue` es mucho más restrictivo que un `goto`. Concretamente, un `break` o `continue` con etiqueta compilan con éxito solamente si la sentencia en que se

encuentra colocada la etiqueta es una sentencia a la que se pueda llegar con un `break` o `continue` normal.

return

```
return expresión;
```

La sentencia `return` se utiliza para terminar un método o función y opcionalmente devolver un valor al método de llamada.

En el código de una función siempre hay que ser consecuentes con la declaración que se haya hecho de ella. Por ejemplo, si se declara una función para que devuelva un entero, es imprescindible colocar un `return` final para salir de esa función, independientemente de que haya otros en medio del código que también provoquen la salida de la función. En caso de no hacerlo se generará un Aviso (*Warning*), y el código Java no se debe compilar con *Warnings*.

```
int func() {
  if( a == 0 )
    return 1;
  return 0;    // es imprescindible porque se retorna un entero
  }
```

Si el valor a retornar es `void`, se podrá omitir ese valor de retorno, con lo que la sintaxis se queda en un sencillo:

```
return;
```

y se usaría simplemente para finalizar el método o función en que se encuentra, así como devolver el control al método o función de llamada.

TIPOS GENÉRICOS

Los tipos genéricos representan un concepto difícil de ubicar en un capítulo determinado, porque representan un concepto básico a la hora de la comprobación de tipos; sin embargo, su uso principal es con colecciones. El motivo primordial de la existencia de tipos genéricos (***generics***) es proporcionar a una clase la capacidad de trabajar con una amplia variedad de tipos de objetos. Por ejemplo, la clase **ArrayList** permite almacenar una lista de cualquier tipo de objetos; sin embargo, **ArrayList** siempre fuerza a que sea necesario colocar un moldeo a los objetos que contiene al sacarlos de la lista para convertirlos al tipo que verdaderamente les corresponde:

```
String cadena = (Swing)arrayList.get( 0 );
```

Desde el J2SE 5 se introducen los tipos genéricos en el lenguaje Java para evitar la presencia del moldeo y que no se produzca el error que generaba la ausencia del moldeo en tiempo de ejecución, una excepción de tipo **ClassCastException**. Este soporte a tipos genéricos ha sido una de las peticiones que más insistentemente se han

hecho y se había pospuesto porque requería cambios en la especificación del lenguaje Java e incluso actualizaciones en la especificación de la Máquina Virtual Java. Finalmente, a partir de la versión J2SE 5.0 los tipos genéricos forman parte del lenguaje Java.

Con la inclusión de los tipos genéricos en las colecciones Java, se asegura en tiempo de compilación la asignación de tipos. Para ello incluso se cambia la notación, aunque no resulte demasiado significativa:

```
Coleccion<Tipo[,Tipo,…]> identificador =
  new Coleccion<Tipo[,Tipo,…]>();
```

En donde `Coleccion` es el tipo de la colección de que se trate: **Vector**, **ArrayList**, **HashMap**, etc., y `tipo` es el tipo de objetos que podrá almacenar esa colección. Por ejemplo, un **Vector** que almacene objetos de tipo **String**, se declarará de la siguiente forma:

```
Vector<String> v = new Vector<String>();
```

La aplicación del ejemplo `Java408.java` crea una lista de enteros y otra de cadenas e imprime el contenido de ambas en bucles separados, introduciendo la funcionalidad que proporciona el uso de tipos genéricos. El código completo del ejemplo se reproduce seguidamente.

```
class Java408 {
  public static void main( String args[] ) {
    // Creamos la colección de enteros e introducimos elementos
    Vector<Integer> v = new Vector<Integer>();
    v.add( 1 );
    v.add( 3 );
    v.add( 5 );
    // Creamos la colección de cadenas e introducimos elementos
    List<String> al = new ArrayList<String>();
    al.add( "Tutorial" );
    al.add( "de" );
    al.add( "Java" );
    // Imprimimos el contenido de la colección de enteros
    System.out.println( "Contenido de la coleccion de enteros:" );
    for( Integer i : v )
      System.out.println( i );
    // Imprimimos el contenido de la colección de cadenas
    System.out.println( "Contenido de la coleccion de cadenas:" );
    for( String s : al )
      System.out.println( s );
    }
  }
```

La sintaxis también permite a los desarrolladores indicar los parámetros que extienden una clase o implementan una interfaz. Esto ayuda a restringir el tipo de parámetros que acepta una clase, una interfaz o un método. También se pueden utilizar tipos parametrizables, tal como se muestra en el ejemplo `Java409.java`.

```
class Java409 {
  public static class Clase<T1,T2> {
    private T1 tipo1;
    private T2 tipo2;

    public Clase( T1 tipo1, T2 tipo2 ) {
      this.tipo1 = tipo1;
      this.tipo2 = tipo2;
      }
    public T1 getTipo1() {
      return( this.tipo1 );
      }
    public T2 getTipo2() {
      return( this.tipo2 );
      }
    };

  public static void main( String[] args ) {
    Clase<String,Integer> cCadInt =
      new Clase<String,Integer>( "Primero",1 );
      String t1CadInt = cCadInt.getTipo1();
      Integer t2CadInt = cCadInt.getTipo2();

    Clase<Integer,Boolean> cIntBol =
      new Clase<Integer,Boolean>( 1,true );
      Integer t1IntBol = cIntBol.getTipo1();
      Boolean t2IntBol = cIntBol.getTipo2();
    }
  }
```

En el método *main()*, puede observar el lector la creación de dos clases de tipo **Clase**, una destinada a contener objetos de tipo **String** e **Integer** y otra que contendrá objetos de tipo **Integer** y **Boolean**.

Del mismo modo se pueden utilizar tipos genéricos a la hora de parametrizar una interfaz o un método de una clase.

El uso de tipos genéricos no aumenta el tamaño del código, ni reduce la complejidad del desarrollo, sino que permite al compilador comprobar el tipo de datos que integran la colección y también evita la necesidad de colocar moldeos explícitos a la hora de manipular los datos de las colecciones.

La utilización de tipos genéricos en lenguaje Java permite el empleo de *comodines*. Para entender por qué se han incorporado, considere el lector la siguiente clase que define un objeto de tipo **Vehiculo** y las clases que extienden el objeto **Vehiculo** particularizándolo en objetos de tipo **Automovil** y **Motocicleta**, cada uno de ellos con los métodos *arrancar()* y *parar()*.

```
public abstract class Vehiculo {
  abstract public void arrancar();
  abstract public void parar();
  }
```

```
class Automovil extends Vehiculo {
  public void arrancar() {
    System.out.println( "El coche esta en movimiento..." );
    }
  public void parar() {
    System.out.println( "El coche esta detenido..." );
    }
  }

class Motocicleta extends Vehiculo {
  public void arrancar() {
    System.out.println( "La moto esta en movimiento..." );
    }
  public void parar() {
    System.out.println( "La moto esta detenida..." );
    }
  }
```

Si ahora se crea una aplicación, `Java410.java`, en la que se declaran colecciones de estos objetos, tal como se reproduce en el código siguiente:

```
public class Java410 {
  public static void main( String[] args ) {
    // Definimos un Vector de objetos de tipo Automovil
    Vector<Automovil> coches = new Vector<Automovil>();
    // Añadimos algunos elementos al vector
    coches.add( new Automovil() );
    coches.add( new Automovil() );
    // Definimos un Vector de objetos de tipo Motocicleta
    Vector<Motocicleta> motos = new Vector<Motocicleta>();
    // Añadimos algunos elementos al vector
    motos.add( new Motocicleta() );
    motos.add( new Motocicleta() );
    // Ahora arrancamos todos los vehículos
// Esta línea código es la que genera el error que se indica de
// incompatibilidad de tipos
    Vector<Vehiculo> misVehiculos = coches;
    // Arrancamos los dos coches y las dos motocicletas
    for( Vehiculo v : misVehiculos )
      v.arrancar();
    misVehiculos = motos;
    for( Vehiculo v : misVehiculos )
      v.arrancar();
    }
  }
```

en tiempo de compilación se producen los errores que se muestran:

```
% javac Java410.java
Java410.java:57: incompatible types
found   : java.util.Vector<Automovil>
required: java.util.Vector<Vehiculo>
    Vector<Vehiculo> misVehiculos = coches;
                                    ^
Java410.java:65: incompatible types
found   : java.util.Vector<Motocicleta>
required: java.util.Vector<Vehiculo>
```

```
misVehiculos = motos;
               ^
```

Para que la anterior aplicación funcione, hay que recurrir al uso de comodines, simplemente creando el **Vector** tal como se muestra a continuación.

```
Vector<? extends Vehiculo> misVehiculos = coches;
```

El símbolo del interrogante (?) es un comodín, que junto con la palabra clave `extends` declara a `misVehiculos` como un **Vector** de cualquier tipo que derive de **Vehiculo**. Lo cual difiere de la línea que figuraba en el código inicial del ejemplo, en donde se declaraba a `misVehiculos` como un **Vector** que solamente podía contener objetos de tipo directo **Vehiculo**.

Para que un **Vector** acepte cualquier tipo de objeto pues, se puede utilizar la declaración `Vector<?>`, que es distinta de `Vector<Object>` por el mismo motivo indicado en el párrafo anterior.

CAPÍTULO 5

PROGRAMAS BÁSICOS EN JAVA

En este capítulo se amplía el estudio de los applets, que ya se han presentado en capítulos anteriores. Pero antes de entrar en materia en cuestión de applets, quizá sea más conveniente aprender los entresijos de la herramienta que proporciona *Sun Microsystems* para su visualización, porque no siempre es seguro que el navegador que se utilice para ejecutar el código HTML que contenga la marca `APPLET` tenga el soporte de todas las características que se hayan incorporado a la versión de JDK con que se haya compilado, dada la vertiginosidad con que evoluciona la plataforma Java, cosa que sí se puede asegurar tratándose del visor de applets, *appletviewer*, amén de que para el desarrollador es mucho más cómodo comprobar el funcionamiento de un applet en esta herramienta que en un navegador.

EL VISOR DE APPLETS (*APPLETVIEWER*)

El visor de applets (*appletviewer*) es una aplicación que permite ver en funcionamiento applets, sin necesidad de utilizar de un navegador World-Wide-Web (*www*) como *Firefox*, *Mozilla*, *Opera*, *Internet Explorer* o *Netscape Navigator*. En adelante, se recurrirá muchas veces a él, ya que el objetivo de este Tutorial es el lenguaje Java y, como se ha indicado antes, es la forma más sencilla, rápida, cómoda y económica de poder ver un applet en ejecución.

Applet

La definición más extendida de ***applet***, muy bien resumida en su día por Patrick Naughton, indica que un applet es *"una pequeña aplicación accesible en un servidor Internet, que se transporta por la red, se instala automáticamente y se ejecuta in situ como parte de un documento web"*. Claro que así la definición establece el entorno

(Internet, Web, etc.). En realidad, un applet es una aplicación pretendidamente corta (nada impide que ocupe más de un gigabyte, a no ser el pensamiento de que se va a transportar por la red y una mente sensata) basada en un formato gráfico sin representación independiente: es decir, se trata de un elemento a embeber en otras aplicaciones; es un *componente* en su sentido estricto.

Un ejemplo en otro ámbito de cosas podría ser el siguiente: imagínese el lector una empresa que cansada de empezar siempre a codificar desde cero, diseña un formulario con los datos básicos de una persona (nombre, dirección, etc.). Tal formulario no es un diálogo por sí mismo, pero se podría integrar en diálogos de clientes, proveedores, empleados, etc. El hecho de que se integre estática (embebido en un ejecutable) o dinámicamente (intérpretes, librerías, DLLs, etc.) no afecta en absoluto a la esencia de su comportamiento como componente con el cual construir diálogos con sentido autónomo.

Pues bien, así es un applet. Lo que ocurre es que, dado que no existe una base adecuada para soportar aplicaciones industriales Java en las que insertar estas miniaplicaciones (aunque todo se va andando y ya se pueden encontrar aplicaciones de este tipo en intranets), los applets se han construido mayoritariamente, y con gran acierto comercial (a las pruebas hay que remitirse), como pequeñas aplicaciones interactivas, con movimiento, luces y sonido... en *Internet*.

Llamadas a applets con *appletviewer*

Al ser un applet una mínima aplicación Java diseñada para ejecutarse en un navegador Web, no necesita por tanto preocuparse por un método *main()* ni en dónde se realizan las llamadas. El applet asume que el código se está ejecutando desde dentro de un navegador. El *appletviewer* se asemeja al mínimo navegador. Espera como argumento el nombre del fichero HTML a cargar, no se le puede pasar directamente un programa Java. Este fichero HTML debe contener una marca que especifica el código que cargará el *appletviewer*:

```
<HTML>
<APPLET CODE=HolaMundo.class WIDTH=300 HEIGHT=100>
</APPLET>
</HTML>
```

El *appletviewer* creará un espacio de navegación, incluyendo un área gráfica, donde se ejecutará el applet, entonces llamará a la clase applet apropiada. En el ejemplo anterior, el *appletviewer* cargará una clase de nombre **HolaMundo** y le permitirá trabajar en su espacio gráfico.

La marca APPLET de HTML

Dado que los applets están generalmente destinados a ser ejecutados en navegadores Web, había que preparar el lenguaje HTML para soportar Java, o mejor,

los applets. El esquema de marcas de HTML y la evolución del estándar marcado por *Netscape* hicieron fácil la adición de una nueva marca que permitiera, una vez añadido el correspondiente código gestor en los navegadores, la ejecución de programas Java en ellos.

La etiqueta APPLET fue definida fundamentalmente por el consorcio *World-Wide-Web*; sin embargo, actualmente este consorcio ha recomendado que en su lugar se utilice la etiqueta OBJECT. A continuación se describe la etiqueta APPLET, porque para mostrar la forma de incorporar applets a un documento web, es la forma más sencilla y directa. Con el JDK de Java, *Sun Microsystems* proporciona una herramienta que convierte de forma automática las etiquetas APPLET a la combinación de etiquetas OBJECT y EMBED, para aplicar las recomendaciones del *World-Wide-Web*.

La sintaxis de las etiquetas <APPLET> y <PARAM> es la que se muestra a continuación y que se irá explicando en detalle a través de los párrafos posteriores:

```
<APPLET CODE=  WIDTH=  HEIGTH=  [CODEBASE=] [ALT=] [ARCHIVE=]
[NAME=] [ALIGN=] [VSPACE=] [HSPACE=]>
<PARAM NAME=  VALUE= >
</APPLET>
```

Atributos obligatorios:
CODE: Nombre de la clase principal que implementa el applet
WIDTH: Anchura inicial del applet en el documento Web
HEIGHT: Altura inicial del applet en el documento Web

Atributos opcionales:
CODEBASE: URL base del applet
ARCHIVE: archivo JAR que contiene el applet empaquetado
ALT: Texto alternativo
NAME: Nombre de la instancia para referenciar en otros applets
ALIGN: Justificación del applet
VSPACE: Espaciado vertical
HSPACE: Espaciado horizontal

Los applets se incluyen en las páginas Web a través de la marca <APPLET>, que para quien conozca HTML resultará muy similar a otras marcas de ese lenguaje, como la marca <IMG>, por ejemplo. Ambas necesitan la referencia a un fichero fuente que no forma parte de la página en que se encuentran embebidos. IMG hace esto a través de SRC=parámetro y APPLET lo hace a través de CODE=parámetro. El parámetro de CODE indica al navegador dónde se encuentra el fichero con el código Java compilado .class. Es una localización relativa al documento fuente.

Para proporcionar parámetros a un applet, que al fin y al cabo es un programa ejecutable, se define otra nueva etiqueta HTML: <PARAM>. Estas etiquetas no son estándares, por lo que son ignoradas por aquellos navegadores que no sean *Java Compatibles*.

Por razones que el autor no entiende muy bien, pero posiblemente relacionadas con los *packages* y *classpaths*, si un applet reside en un directorio diferente del que contiene a la página en que se encuentra embebido, entonces no se indica una URL a esta localización, sino que se apunta al directorio del fichero `.class` utilizando el parámetro `CODEBASE`, aunque todavía se puede usar `CODE` para proporcionar el nombre del fichero `.class`.

Al igual que `IMG`, `APPLET` tiene una serie de parámetros que lo posicionan en la página. `WIDTH` y `HEIGHT` especifican el tamaño del rectángulo que contendrá al applet, se indican en píxeles. `ALIGN` funciona igual que con `IMG` (en los navegadores que lo soportan), definiendo cómo se posiciona el rectángulo del applet con respecto a los otros elementos de la página. Los valores posibles a especificar son: `LEFT`, `RIGHT`, `TOP`, `TEXTTOP`, `MIDDLE`, `ABSMIDDLE`, `BASELINE`, `BOTTOM` y `ABSBOTTOM`. Y, finalmente, del mismo modo que con `IMG`, se puede especificar un `HSPACE` y un `VSPACE` en píxeles para indicar la cantidad de espacio vacío que habrá de separación entre el applet y el texto que le rodea.

`APPLET` tiene una marca `ALT`. La utilizaría un navegador que entendiese la marca `APPLET`, pero que por alguna razón, no pudiese ejecutarlo. Por ejemplo, si un applet necesita escribir en el disco duro del ordenador, pero en las características de seguridad está bloqueada esa posibilidad, entonces el navegador presentaría el texto asociado a `ALT`.

`ALT` no es utilizado por los navegadores que no tratan la marca `APPLET`, por ello se ha definido la marca `</APPLET>`, que finaliza la descripción del applet. Un navegador con soporte Java ignorará todo el texto que haya entre las marcas `<APPLET>` y `</APPLET>`, sin embargo, un navegador que no soporte Java ignorará las marcas y presentará el texto que haya entre ellas.

Se comentó en párrafos anteriores que con el JDK se dispone de un conversor para la etiqueta `APPLET`, que adapta las llamadas a las recomendaciones del consorcio *World-Wide-Web*. Ejecutando este conversor sobre el applet **HolaMundo**, el resultado que se obtiene es el que se muestra a continuación.

```
<HTML>
<!--"CONVERTED_APPLET"-->
<!-- HTML CONVERTER -->
<object
    classid = "clsid:CAFEEFAC-0016-0000-0005-ABCDEFFEDCBA"
    codebase = "http://java.sun.com/update/1.6.0/jinstall-6u50-windows-
i586.cab#Version=6,0,50,13"
    WIDTH = 300 HEIGHT = 100 >
    <PARAM NAME = CODE VALUE = "HolaMundo.class" >
    <param name = "type" value = "application/x-java-applet;jpi-
version=1.6.0_05">
    <param name = "scriptable" value = "false">
    <comment>
       <embed
            type = "application/x-java-applet;jpi-version=1.6.0_05" \
```

```
            CODE = "HolaMundo.class" \
            WIDTH = 300 \
            HEIGHT = 100
           scriptable = false
           pluginspage =
"http://java.sun.com/products/plugin/index.html#download">
           <noembed>
            </noembed>
       </embed>
    </comment>
</object>

<!--
<APPLET CODE = "HolaMundo.class" WIDTH = 300 HEIGHT = 100>
</APPLET>
-->
<!--"END_CONVERTED_APPLET"-->
</HTML>
```

En el listado anterior se puede comprobar el uso de las etiquetas `OBJECT` y `EMBED`, cuya combinación proporciona soporte tanto para *Internet Explorer* como para cualquier navegador que cumpla las recomendaciones del *World-Wide-Web* como *Netscape Navigator* o *Firefox*.

PASO DE PARÁMETROS A APPLETS

El espacio que queda entre las marcas de apertura y cierre de la definición de un applet se utiliza para el paso de parámetros al applet. Para ello se utiliza la marca `PARAM` en la página HTML para indicar los parámetros y el método *getParameter()* de la clase **java.applet.Applet** para leerlos en el código interno del applet. La construcción puede repetirse cuantas veces se quiera, una tras otra.

Los atributos que acompañan a la marca `PARAM` son los siguientes:

NAME
Nombre del parámetro que se desea pasar al applet.

VALUE
Valor que se desea transmitir en el parámetro que se ha indicado antes.

Texto HTML
Texto HTML que será interpretado por los navegadores que no entienden la marca `APPLET` en sustitución del applet mismo.

Para mostrar esta posibilidad, se recurre de nuevo al conocido applet básico *HolaMundo*, para modificarlo y que pueda saludar a cualquiera. Lo que se hará será pasarle al applet el nombre de la persona a quien se desea saludar. Se genera el código para ello y se guarda en el fichero `HolaTal.java`

```
import java.awt.Graphics;
import java.applet.Applet;

public class HolaTal extends Applet {
  String nombre;
  public void init() {
    // Obtiene la cadena del parámetro "nombre" que se fija en la
    // llamada al applet desde la página html
    nombre = getParameter( "Nombre" );
    }
  public void paint( Graphics g ) {
    // Pinta el saludo en la posición indicada
    g.drawString( "Hola "+nombre+"!",25,25 );
    }
  }
```

Al compilar el ejemplo se obtiene el fichero `HolaTal.class` que se incluirá en la siguiente página Web, que se genera a partir del fichero `HolaTal.html`, en el que se incluye el applet, y que tendrá el siguiente contenido:

```
<HTML>
<APPLET CODE=HolaTal.class WIDTH=300 HEIGHT=100>
<PARAM NAME="Nombre" VALUE="Agustin">
</APPLET>
</HTML>
```

Evidentemente el lector puede sustituir el nombre por el suyo propio. Este cambio no afectará al código Java, no será necesario recompilarlo para que el applet tenga la delicadeza de saludar al amable lector.

Los parámetros no se limitan a uno solo. Se puede pasar al applet cualquier número de parámetros y siempre hay que indicar un *nombre* y un *valor* para cada uno de ellos.

El método *getParameter()* es fácil de entender. El único argumento que necesita es el *nombre* del parámetro cuyo valor se quiere recuperar. Todos los parámetros se pasan como objetos de tipo **String**, en caso de necesitar pasarle al applet un valor entero, se ha de pasar como **String**, recuperarlo como tal y luego convertirlo al tipo que se desee. Tanto el argumento de `NAME` como el de `VALUE` deben ir colocados entre dobles comillas ("), ya que son **String**.

El hecho de que las marcas `<APPLET>` y `<PARAM>` sean ignoradas por los navegadores que no entienden Java, es inteligentemente aprovechado a la hora de definir un contenido alternativo a ser mostrado en este último caso. Así la etiqueta es doble:

```
<APPLET atributos>
parámetros
contenido alternativo
</APPLET>
```

El fichero anterior para mostrar el applet de ejemplo se puede modificar para que pueda ser visualizado en cualquier navegador y en unos casos presente la información alternativa y, en otros, ejecute el applet:

```
<HTML>
<APPLET CODE=HolaTal.class WIDTH=300 HEIGHT=100>
<PARAM NAME="Nombre" VALUE="Agustin">
No verás lo bueno hasta que consigas un navegador
<I>Compatible con Java</I>
</APPLET>
</HTML>
```

EL PARÁMETRO *ARCHIVE*

El parámetro `ARCHIVE` ha sido una de las mejores aportaciones de *Sun Microsystems* desde las últimas versiones del JDK, así que a continuación se profundiza un poco más en el porqué de su incorporación y en su funcionamiento.

Una de las cosas que se achacan a Java es la lentitud. El factor principal en la percepción que tiene el usuario de la velocidad y valor de los applets es el tiempo que tardan en cargarse todas las clases que componen el applet. Algunas veces hay que estar esperando más de un minuto para ver una triste animación, ni siquiera buena. Y, desafortunadamente, esta percepción de utilidad negativa puede recaer también sobre applets que realmente sí son útiles.

Para entender el porqué de la necesidad de un nuevo método de carga para acelerarla, es necesario comprender por qué el método actual es lento. Normalmente un applet se compone de varias clases, es decir, varios ficheros `.class`. Por cada uno de estos ficheros `.class`, el cargador de clases debe abrir una conexión individual entre el navegador y el servidor donde reside el applet. Así, si un applet se compone de 20 ficheros `.class`, el navegador necesitará abrir 20 conexiones para transmitir cada uno de los ficheros. La sobrecarga que representa cada una de estas conexiones es relativamente significativa. Por ejemplo, cada conexión necesita un número de paquetes adicionales que incrementan el tráfico en la Red.

Ya el avispado lector habrá pensado la solución al problema: poner todos los ficheros en uno solo, con lo cual solamente sería necesaria una conexión para descargar todo el código del applet. Bien pensado. Esto es lo mismo que han pensado en un principio los dos grandes competidores en el terreno de los navegadores, *Netscape* y *Microsoft*, y que posteriormente *Sun Microsystems* ha recogido ya oficialmente.

Desafortunadamente, las soluciones que han implementado ambas compañías no son directamente compatibles. *Microsoft*, en su afán de marcar diferencias, crea su propio formato de ficheros **CAB**. La solución de *Netscape* es utilizar el archiconocido formato **ZIP**. Y *Sun Microsystems*, saliéndose por la tangente, ha definido un nuevo formato de ficheros, que incorpora desde el JDK 1.1, para incluir juntos todos los

ficheros de imágenes, sonido y *.class*, que ha llamado formato ***JAR*** (**J**ava **Ar**chive), por suerte también basado en tecnología *ZIP*. No obstante, es fácil escribir código HTML de forma que maneje cualquiera de los formatos en caso necesario. Esto es así porque se puede especificar cada uno de estos formatos de ficheros especiales en extensiones separadas de la marca <APPLET>.

No es de interés el contar la creación de ficheros *CAB*; si el lector está interesado, puede consultar la documentación que proporciona *Microsoft.* Una vez que se dispone de este fichero, se puede añadir un parámetro CABBASE a la marca <APPLET>:

```
<APPLET NAME="Hola" CODE="HolaMundo" WIDTH=50 HEIGHT=50 >
  <PARAM NAME=CODEBASE VALUE="http://www.ejemplo.es/classes">
  <PARAM NAME=CABBASE VALUE="hola.cab">
</APPLET>
```

El VALUE del parámetro CABBASE es el nombre del fichero *CAB* que contiene los ficheros `.class` que componen el conjunto de applet.

Crear un archivo *ZIP* para utilizarlo con Netscape o Firefox es muy fácil. Se deben agrupar todos los ficheros `.class` necesarios en un solo fichero `.zip`. Lo único a tener en cuenta es que solamente hay que almacenar los ficheros `.class` en el archivo; es decir, no hay que comprimir.

Si se está utilizando **pkzip**, se haría:

```
pkzip -e0 archivo.zip listaFicherosClass
```

El parámetro que se indica en la línea de comandos anterior es el número *cero*, no la "O" mayúscula.

Para utilizar un fichero *.zip* hay que indicarlo expresamente en la marca ARCHIVE de la sección <APPLET>:

```
<APPLET NAME="Hola" CODE="HolaMundo" WIDTH=50 HEIGHT=50
  CODEBASE VALUE="http://www.ejemplo.es/classes"
  ARCHIVE="hola.zip">
</APPLET>
```

Pero hay más. Es posible crear ambos tipos de ficheros y hacer que tanto los usuarios de *Netscape Navigator*, como los de *Firefox*, como los de *Internet Explorer* puedan realizar descargas rápidas del código del applet. No hay que tener en cuenta los usuarios de otros navegadores, o de versiones antiguas de estos navegadores, porque ellos todavía podrán seguir cargando los ficheros a través del método lento habitual. Para compatibilizarlo todo, se juntan todas las piezas anteriores:

```
<APPLET NAME="Hola" CODE="HolaMundo" WIDTH=50 HEIGHT=50
  CODEBASE VALUE="http://www.ejemplo.es/classes"
  ARCHIVE="hola.zip">
  <PARAM NAME=CABBASE VALUE="hola.cab">
  <PARAM NAME=CODEBASE VALUE="http://www.ejemplo.es/classes">
```

```
  <PARAM NAME=CABBASE VALUE="hola.cab">
</APPLET>
```

Ahora que se puede hacer esto con ficheros `.cab` y `.zip`, es tarea del lector el trabajo de incorporar los ficheros `.jar` y poner los tres formatos juntos bajo el mismo paraguas de la marca `<APPLET>`.

ESCRIBIR APPLETS JAVA

Para escribir applets Java, hay que utilizar una serie de métodos. Incluso para el applet más sencillo. Son los que se usan para arrancar (*start*) y detener (*stop*) la ejecución del applet, para pintar (*paint*) y actualizar (*update*) la pantalla y para capturar la información que se pasa al applet desde el fichero *html* a través de la marca `APPLET` (*getParameter*).

Los applets no necesitan un método *main()* como las aplicaciones Java, sino que deben implementar (redefinir) al menos uno de los tres métodos siguientes: *init()*, *start()* o *paint()*.

init()

Esta función miembro es llamada una única vez al crearse el applet. La clase **Applet** no hace nada en *init()*. Las clases derivadas deben sobrescribir este método para cambiar el tamaño durante su inicialización, y cualquier otra inicialización de los datos que solamente deba realizarse una vez. Deberían realizarse al menos las siguientes acciones:

- Carga de imágenes y sonido.
- El redimensionado del applet para que tenga su tamaño correcto.
- Asignación de valores a las variables globales.

Por ejemplo:

```
public void init() {
  if( width < 200  ||  height < 200 )
    resize( 200,200 );
  valor_global1 = 0;
  valor_global2 = 100;
  // cargaremos imágenes en memoria sin mostrarlas
  // cargaremos música de fondo en memoria sin reproducirla
  }
```

destroy()

Esta función miembro es llamada cuando el applet no se va a usar más. La clase **Applet** no hace nada en este método. Las clases derivadas deberían sobrescribirlo para hacer una limpieza final. Los applets multitarea deberán usar *destroy()* para "*matar*"

cualquier tarea que pudiese quedar activa, antes de concluir definitivamente la ejecución del applet.

start()

Llamada para activar el applet. Esta función miembro es llamada cuando se visita el applet. La clase **Applet** no hace nada en este método. Las clases derivadas deberían sobrescribirlo para comenzar una animación, sonido, etc.

```
public void start() {
  estaDetenido = false;
  // comenzar la reproducción de la música
  musicClip.play();
  }
```

También se puede utilizar *start()* para eliminar cualquier tarea que se necesite.

stop()

Llamada para detener el applet. Se llama cuando el applet desaparece de la pantalla. La clase **Applet** no hace nada en este método. Las clases derivadas deberían sobrescribirlo para detener la animación, el sonido, etc.

```
public void stop() {
  estaDetenido = true;
  if( /* ¿se está reproduciendo música? */ )
    musicClip.stop();
  }
```

resize(int width,int height)

El método *init()* debería llamar a esta función miembro para establecer el tamaño del applet. Puede utilizar las variables anchura y altura, pero no es necesario. Cambiar el tamaño en otro sitio que no sea *init()* produce un reformateo de todo el documento y no se recomienda.

width

Variable entera, su valor es el ancho definido en el parámetro `WIDTH` de la marca HTML del `APPLET`. Por defecto es la anchura del icono.

height

Variable entera, su valor es la altura definida en el parámetro `HEIGHT` de la marca HTML del `APPLET`. Por defecto es la altura del icono. Tanto `width` como `height` están siempre disponibles para que se pueda comprobar el tamaño del applet.

Se puede retomar el ejemplo de *init()*:

```
public void init() {
  if( width < 200  ||  height < 200 )
    resize( 200,200 );
  ...
```

paint(Graphics g)

Este método se llama cada vez que se necesita refrescar el área de dibujo del applet. El método *paint()* pertenece a la clase **Component**, que es heredado por varias clases intermedias y, finalmente, es heredado por la clase **Applet**. La clase **Applet** simplemente dibuja una caja en el área. Obviamente, la clase derivada debería sobrescribir este método para representar algo inteligente en la pantalla.

Para repintar toda la zona del applet cuando llega un evento `Paint`, se pide el rectángulo sobre el que se va a aplicar *paint()* y si es más pequeño que el tamaño real del applet se invoca a *repaint()*, que como va a hacer un *update()*, actualizará toda la zona.

Se puede utilizar *paint()* para imprimir el mensaje de bienvenida:

```
void public paint( Graphics g ) {
  g.drawString( "Hola Java!",25,25 );
  // Dibujaremos las imágenes que necesitemos
  }
```

update(Graphics g)

Ésta es la función que se llama realmente cuando se necesita actualizar la pantalla. La clase **Applet** simplemente limpia el área y llama al método *paint()*. Esta funcionalidad es suficiente en la mayoría de los casos. De cualquier forma, las clases derivadas pueden sustituir esta funcionalidad para sus propósitos.

Se puede, por ejemplo, utilizar *update()* para modificar selectivamente partes del área gráfica sin tener que pintar el área completa:

```
public void update( Graphics g ) {
  if( estaActualizado ) {
    g.clear();  // garantiza la pantalla limpia
    repaint();  // podemos usar el método padre:
                //   super.update()
    }
  else
    // Información adicional
    g.drawString( "Otra información",25,50 );
  }
```

repaint()

A esta función debe llamarse cuando el applet necesite ser repintado. No debería sobrescribirse, sino dejar que Java repinte completamente el contenido del applet.

Al llamar a *repaint()* sin parámetros, internamente se llama a *update()* que borrará el rectángulo sobre el que se redibujará y luego se llama a *paint()*. Como a *repaint()* se le pueden pasar parámetros, se puede modificar el rectángulo a repintar.

getParameter(String attr)

Este método carga los valores pasados al applet vía la marca `APPLET` de HTML. El argumento **String** es el nombre del parámetro que se quiere obtener. Devuelve el valor que se le haya asignado al parámetro; en caso de que no se le haya asignado ninguno, devolverá `null`.

Para usar *getParameter()*, se define una cadena genérica. Una vez que se ha capturado el parámetro, se utilizan métodos de cadena o de números para convertir el valor obtenido al tipo adecuado.

```
public void init() {
  String pv;
  pv = getParameter( "velocidad" );
  if( pv == null )
    velocidad = 10;
  else
    velocidad = Integer.parseInt( pv );
  }
```

getDocumentBase()

Indica la ruta *http*, o el directorio del disco, de donde se ha obtenido la página HTML que contiene el applet, es decir, el lugar donde está el documento en todo Internet o en el disco.

print(Graphics g)

Para imprimir en impresora, al igual que *paint()* se puede utilizar *print()*, que pintará en la impresora el mapa de bits del dibujo.

CAPÍTULO 6

CONCEPTOS BÁSICOS DE JAVA

En este capítulo se presentan de forma más completa algunos de los conceptos que se introducían en el capítulo anterior de forma más práctica. También se revisan otros conceptos fundamentales de la programación orientada a objetos que tienen amplia repercusión en el lenguaje Java.

El elemento básico de la programación orientada a objetos en Java es una *Clase*. Una *Clase* define la forma y comportamiento de un *Objeto*. Todo en Java gira alrededor de estos conceptos, por ello, una vez que se han repasado los mecanismos básicos del lenguaje en cuestiones ya conocidas, este capítulo se adentra en la ampliación de los conceptos teóricos enfocados hacia el estilo de programación orientada a objetos de Java.

OBJETOS

Antes de entrar en el estudio de las Clases en Java, hay que presentar a los *Objetos*, que son las instanciaciones de una clase. En este caso, haciendo un símil con la vida real, se estaría colocando el carro delante del caballo, pero se supone que el lector tiene el bagaje suficiente como para entender lo que se expondrá, sin necesidad de conocer en profundidad lo que es y cómo funciona una clase. Además, esto permitirá obviar muchas cosas cuando se entre en el estudio de las clases. Y, en última instancia, el lector siempre puede pasar al capítulo siguiente y luego volver aquí.

Un objeto es la instanciación de una clase y tiene un estado y un funcionamiento. El *estado* está contenido en sus **variables**, *variables miembro*, y el *funcionamiento* viene determinado por sus **métodos**. Las variables miembro pueden ser *variables de instancia* o *variables de clase*. Se activa el funcionamiento del objeto invocando a uno

de sus métodos, que realizará una acción o modificará su estado, o bien ambas cosas. Cuando un objeto ya no se usa, en otros lenguajes como C++, se destruye y la memoria que ocupaba se libera y es devuelta al Sistema. Cuando un objeto ya no se usa en Java, simplemente se anula. Posteriormente, el reciclador de memoria (*garbage collector*) puede recuperar esa memoria para devolverla al Sistema.

Las afirmaciones anteriores son un compendio de lo que se puede esperar de un objeto. A continuación, se verá más en detalle cómo es la creación, el uso y la destrucción de un objeto.

Creación de objetos

Un objeto es (insistiendo) una instancia de una clase. La creación de un objeto se realiza en tres pasos (aunque se pueden combinar):

- *Declaración*, proporcionar un nombre al objeto.
- *Instanciación*, asignar memoria al objeto.
- *Inicialización*, opcionalmente se pueden proporcionar valores iniciales a las variables de instancia del objeto.

La declaración e instanciación de una variable de tipo básico utiliza una sentencia que asigna un nombre a la variable y reserva memoria para almacenar su contenido:

```
int miVariable;
```

Se puede inicializar la variable con un valor determinado en el momento de declararla, es decir, se pueden resumir los tres pasos anteriormente citados de declaración, instanciación e inicialización en una sola sentencia:

```
int miVariable = 7;
```

Y, lo más importante, esto sucede en tiempo de compilación.

En Java no siempre es necesaria la declaración de un objeto (darle un nombre). En el siguiente ejemplo, `Java601.java`, se instancia un nuevo objeto que se usa en una expresión, sin haberlo declarado previamente.

```
class Java601 {
  public static void main( String args[] ) {
    System.out.println( new Date() );
    }
  }
```

La inicialización de un objeto se puede realizar utilizando las constantes de la clase, de forma que un objeto de un mismo tipo puede ser declarado e inicializado de formas diferentes.

Utilización de objetos

Una vez que se tiene declarado un objeto con sus variables y sus métodos, se puede acceder a ellos a fin de activar la funcionalidad para la que se ha creado. Sin embargo, algunos métodos o variables pueden estar ocultos y el acceso a ellos resultar imposible.

Para acceder a variables o métodos se especifica el nombre del objeto y el nombre de la variable, o método, separados por un punto (.).

En las siguientes sentencias, se muestran las formas de acceso que permite el lenguaje Java:

```
System.out.println( "miObjeto apunta a "+miObjeto.getData() );
fecha.displayFecha( dateContainer( "29/07/97" );
```

Destrucción de objetos

El programador Java no necesita preocuparse de devolver la memoria, ese bien tan preciado, al sistema operativo. Eso se realizará automáticamente de la mano del *reciclador de basura*, otro nombre para el imprescindible *garbage collector*.

Sin embargo, hay una mala noticia. Y es que en Java no hay soporte para algo parecido a un destructor, como existe en C++, que pueda garantizar su ejecución cuando el objeto deje de existir. Por lo tanto, excepto la devolución de memoria al sistema operativo, la liberación de los demás recursos que el objeto utilizase vuelve a ser responsabilidad del programador. Pero, algo se ha ganado, ya no debería aparecer más el fatídico mensaje de "*No hay memoria suficiente*", o su versión inglesa de "*Out of Memory*".

LIBERACIÓN DE MEMORIA

El único propósito para el que se ha creado el reciclador de memoria es para devolver al sistema operativo la memoria ocupada por objetos que ya no es necesaria. Un objeto es blanco del *garbage collector* para su reciclado cuando deja de haber referencias a ese objeto. Sin embargo, el que un objeto sea elegido para su reciclado, no significa que eso se haga inmediatamente.

El *garbage collector* es una tarea, o *thread*, o hilo de ejecución, de baja prioridad; que puede ejecutarse síncrona o asíncronamente, dependiendo de la situación en que se encuentre el sistema Java. Se ejecutará *síncronamente* cuando el sistema detecta poca memoria o en respuesta a un programa Java. El programador puede activar el *garbage collector* en cualquier momento llamando al método *System.gc()*. Se ejecutará *asíncronamente* cuando el sistema se encuentre sin hacer nada (*idle*) en aquellos sistemas operativos en los que para lanzar una tarea haya que interrumpir la ejecución de otra, como es el caso de Windows 95/98/NT/2000.

No obstante, la llamada al sistema para que se active el *garbage collector* no garantiza que la memoria que consumía sea liberada.

A continuación, se muestra un ejemplo sencillo de funcionamiento del *garbage collector*:

```
String s;                   // no se ha asignado memoria todavía
s = new String( "abc" );    // memoria asignada
s = "def";                  // se ha asignado nueva memoria
// (nuevo objeto)
```

Más adelante se verá en detalle la clase **String**, pero una breve descripción de lo que hace esto es: crear un objeto **String**, rellenarlo con los caracteres "abc", crear otro (nuevo) **String** y colocarle los caracteres "def".

En esencia se crean dos objetos:

- Objeto **String** "abc"
- Objeto **String** "def"

Al final de la tercera sentencia, el primer objeto creado de nombre s que contiene "abc" se ha salido de alcance. No hay forma de acceder a él. Ahora se tiene un nuevo objeto llamado s y contiene "def". Es marcado y eliminado en la siguiente iteración de la tarea que recicla la memoria.

EL MÉTODO PARA FINALIZAR

Antes de que el reciclador de memoria reclame la memoria ocupada por un objeto, se puede invocar el método *finalize()*. Este método es miembro de la clase **Object**, por lo tanto, todas las clases lo contienen. La declaración, por defecto, de este método es:

```
protected void finalize() {}
```

Para utilizar este método, hay que sobrescribirlo, proporcionando el código que contenga las acciones que se desee ejecutar antes de liberar la memoria consumida por el objeto. Aunque se puede asegurar que el método *finalize()* se invocará antes de que el *garbage collector* reclame la memoria ocupada por el objeto, no se puede asegurar cuándo se liberará esa memoria; e incluso, si esa memoria será liberada.

Además del método *System.gc()*, que permite invocar al *garbage collector* en cualquier instante, también puede ser posible forzar la ejecución de los métodos *finalize()* de los objetos marcados para reciclar, llamando al método:

```
System.runFinalization();
```

Pero la circunstancia explicada anteriormente sigue vigente: no se puede garantizar que esto libere todos los recursos y, por lo tanto, que se produzca la finalización del objeto.

CLASES

Las *Clases* son lo más simple de Java. Todo en Java forma parte de una *Clase*, es una *Clase* o describe cómo funciona una *Clase*. El conocimiento de las *Clases* es fundamental para poder entender los programas Java.

Las acciones de los programas Java se colocan dentro del bloque de una clase o un *Objeto*. Un *Objeto* es una instancia de una clase. Todos los métodos se definen dentro del bloque de la clase, Java no soporta funciones o variables globales. Así pues, el esqueleto de cualquier aplicación Java se basa en la definición de una clase.

Todos los datos básicos, como los *enteros*, se deben declarar en las clases antes de hacer uso de ellos. En lenguajes procedurales como C la unidad fundamental son los ficheros con código fuente, en Java son las clases. De hecho son pocas las sentencias que se pueden colocar fuera del bloque de una clase. La palabra clave `import` puede colocarse al principio de un fichero, fuera del bloque de la clase. Sin embargo, el compilador reemplazará esa sentencia con el contenido del fichero que se indique, que consistirá, como es de suponer, en más clases.

La definición de una clase consta de dos partes, la declaración y el cuerpo, según la siguiente sintaxis:

```
DeclaracionClase {
  CuerpoClase
  }
```

La declaración de la clase indica al compilador el nombre de la clase, la clase de la que deriva (su *superclase*), los privilegios de acceso a la clase (pública, abstracta, final) y si la clase implementa o no, una o varias *interfaces*. El nombre de la clase debe ser un identificador válido en Java. Por convención, el nombre de las clases Java empieza con una letra mayúscula.

Cada clase Java deriva, directa o indirectamente, de la clase **Object**. La clase padre inmediatamente superior a la clase que se está declarando se conoce como ***superclass*** (superclase). Si no se especifica la superclase de la que deriva una clase, se entiende que deriva directamente de la clase **Object** (definida en el paquete *java.lang*).

En la declaración de una clase se utiliza la palabra clave `extends` para especificar la superclase, de la forma:

```
class MiClase extends SuperClase {
  // cuerpo de la clase
  }
```

La herencia de los métodos de acceso a una clase no se puede utilizar en Java para modificar el control de acceso asignado a un miembro de la clase padre.

Una clase hereda las variables y métodos de su superclase y también de la superclase de esa clase, etc.; es decir, de todo el árbol de jerarquía de clases desde la clase que se está declarando hasta la raíz superior del árbol: **Object**. En otras palabras, un objeto que es instanciado desde una clase determinada, contiene todas las variables y métodos de instancia definidos para esta clase y sus antecesores, aunque los métodos pueden ser modificados (sobrescritos) en algún lugar.

Una clase puede implementar una o más interfaces, declarándose esto utilizando la palabra clave `implements`, seguida de la lista de interfaces que implementa, separadas por coma (,), de la forma:

```
class MiClase extends SuperClase
  implements MiInterfaz,TuInterfaz {
  // cuerpo de la clase
  }
```

Cuando una clase indique que implementa una interfaz, se puede asegurar que proporciona una definición para todos y cada uno de los métodos declarados en esa interfaz; en caso contrario, el compilador generará errores al no poder resolver los métodos de la interfaz en la clase que lo implementa.

Hay cierta similitud entre una interfaz y una clase abstracta, aunque las definiciones de métodos no están permitidas en una interfaz y sí se pueden definir en una clase abstracta. El propósito de las interfaces es proporcionar nombres, es decir, solamente declara lo que necesita implementar la interfaz, pero no cómo se ha de realizar esa implementación; es una forma de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Cuando se implementa una interfaz, los nombres de los métodos de la clase deben coincidir con los nombres de los métodos que están declarados en esa interfaz, todos y cada uno de ellos.

El *cuerpo* de la clase contiene las declaraciones, y posiblemente la inicialización, de todos los datos miembros, tanto variables de clase como variables de instancia, así como la definición completa de todos los métodos.

Las variables pueden declararse dentro del cuerpo de la clase o dentro del cuerpo de un método de la clase. Sin embargo, estas últimas no son variables miembro de la clase, sino variables locales del método en el que están declaradas.

Un objeto instanciado de una clase tiene un *estado* definido por el valor actual de las variables de instancia y un *entorno* definido por los métodos de instancia. Es típico de la programación orientada a objetos el restringir el acceso a las variables y

proporcionar métodos que permitan el acceso a esas variables, aunque esto no es estrictamente necesario.

Alguna o todas las variables pueden declararse para que se comporten como si fuesen constantes, utilizando la palabra clave `final`.

Los objetos instanciados desde una clase contienen todos los métodos de instancia de esa clase y también todos los métodos heredados de la superclase y sus antecesores. Por ejemplo, todos los objetos en Java heredan, directa o indirectamente, de la clase **Object**; luego todo objeto contiene los miembros de la clase **Object**, es decir, la clase **Object** define el estado y entorno básicos para todo objeto Java.

Las características más importantes que la clase **Object** cede a sus descendientes son las siguientes:

- Posibilidad de cooperación consigo o con otro objeto.
- Posibilidad de conversión automática a **String**.
- Posibilidad de espera sobre una variable de condición.
- Posibilidad de notificación a otros objetos del estado de la variable de condición.

La sintaxis general de definición de clase se podría extender tal como se muestra en el siguiente esquema, que debe tomarse sólo como referencia y no al pie de la letra:

```
NombreDeLaClase {
  // declaración de las variables de instancia
  // declaración de las variables de la clase
  metodoDeInstancia() {
    // variables locales y código
    }
  metodoDeLaClase() {
    // variables locales y código
    }
  }
```

Tipos de clases

Hasta ahora sólo se ha utilizado la palabra clave `public` para calificar el nombre de las clases que se han visto, pero hay tres modificadores más. Los tipos de clases que se pueden definir son:

public

Las clases `public` son accesibles desde otras clases, bien sea directamente o por herencia, desde clases declaradas fuera del paquete que contiene a esas clases públicas, ya que, por defecto, las clases solamente son accesibles por otras clases declaradas dentro del mismo paquete en el que se han declarado. Para acceder desde otros paquetes, primero tienen que ser importadas. La sintaxis es:

```
public class miClase extends SuperClase
  implements miInterfaz,TuInterfaz {
  // cuerpo de la clase
  }
```

Aquí la palabra clave `public` se utiliza en un contexto diferente del que se emplea cuando se define internamente la clase, junto con `private` y `protected`.

abstract

Una clase `abstract` tiene al menos un método abstracto. Una clase abstracta no se instancia, sino que se utiliza como clase base para la herencia.

final

Una clase `final` se declara como la clase que termina una cadena de herencia, es lo contrario a una clase abstracta. Nadie puede heredar de una clase `final`. Por ejemplo, la clase **Math** es una clase `final`. Aunque es técnicamente posible declarar clases con varias combinaciones de `public`, `abstract` y `final`, la declaración de una clase abstracta y a la vez final no tiene sentido, y el compilador no permitirá que se declare una clase con esos dos modificadores juntos.

synchronizable

Este modificador especifica que todos los métodos definidos en la clase son sincronizados, es decir, que no se puede acceder al mismo tiempo a ellos desde distintas *tareas*; el sistema se encarga de colocar los *flags* necesarios para evitarlo. Este mecanismo hace que desde tareas diferentes se puedan modificar las mismas variables sin que haya problemas de que se sobrescriban.

Si no se utiliza alguno de los modificadores expuestos, por defecto, Java asumirá que una clase es:

- No final.
- No abstracta.
- Subclase de la clase **Object**.
- No implementa interfaz alguna.

Variables miembro

Una clase en Java puede contener variables y métodos. Las variables pueden ser tipos primitivos como `int`, `char`, etc. Los métodos son funciones.

Por ejemplo, esto se puede observar en el siguiente segmento de código:

```
public class MiClase {
  int i;
```

```
  public MiClase() {
    i = 10;
    }
  public void Suma_a_i( int j ) {
    int suma;
    suma = i + j;
    }
  }
```

La clase **MiClase** contiene una variable (`i`) y dos métodos: *MiClase()* que es el constructor de la clase y *Suma_a_i(`int j`)*.

La *declaración* de una variable miembro aparece dentro del cuerpo de la clase, pero fuera del cuerpo de cualquier método de esa clase. Si se declara dentro de un método, será una variable local del método y no una variable miembro de la clase. En el ejemplo anterior, `i` es una variable miembro de la clase y `suma` es una variable local del método *Suma_ a_i()*.

El *tipo* de una variable determina los valores que se le pueden asignar y las operaciones que se pueden realizar con ella.

El *nombre* de una variable ha de ser un identificador válido en Java. Por convenio, los nombres de variables comienzan con una letra minúscula, pero no es imprescindible. Los nombres de las variables han de ser únicos dentro de la clase y se permite que haya variables y métodos con el mismo nombre.

La sintaxis completa de la declaración de una variable miembro de una clase en Java sería:

```
[especificador_de_acceso][static][final][transient][volatile]
  tipo nombreVariable [= valor_inicial];
```

Ámbito de una variable

Los bloques de sentencias compuestas en Java se delimitan con dos llaves. Las variables de Java sólo son válidas desde el punto donde están declaradas hasta el final de la sentencia compuesta que las engloba. Se pueden anidar estas sentencias compuestas, y cada una puede contener su propio conjunto de declaraciones de variables locales. Sin embargo, no se puede declarar una variable con el mismo nombre que una de ámbito exterior.

En el siguiente ejemplo intenta declarar dos variables separadas con el mismo nombre. En lenguaje Java, esto es ilegal.

```
class Ambito {
  int i = 1;          // ámbito exterior
    {                 // crea un nuevo ámbito
    int i = 2;        // error de compilación
    }
```

```
}
```

Variables de instancia

La declaración de una variable miembro dentro de la definición de una clase sin anteponerle la palabra clave `static`, hace que sea una *variable de instancia* en todos los objetos de la clase. El significado de variable de instancia sería, más o menos, que cualquier objeto instanciado de esa clase contiene su propia copia de toda variable de instancia. Si se examinara la zona de memoria reservada a cada objeto de la clase, se encontraría la reserva realizada para todas las variables de instancia de la clase. En otras palabras, como un objeto es una instancia de una clase, y como cada objeto tiene su propia copia de un dato miembro particular de la clase, entonces se puede denominar a ese dato miembro como variable de instancia.

En Java se accede a las variables de instancia asociadas a un objeto determinado utilizando el nombre del objeto, el operador punto (.) y el nombre de la variable:

```
miObjeto.miVariableDeInstancia;
```

Variables estáticas

La declaración de un dato miembro de una clase usando `static`, crea una *variable de clase* o *variable estática* de la clase. El significado de variable estática es que todas las instancias de la clase (todos los objetos instanciados de la clase) contienen las mismas variables de clase o estáticas. En otras palabras, en un momento determinado se puede querer crear una clase en la que el valor de una variable de instancia sea el mismo (y de hecho sea la misma variable) para todos los objetos instanciados a partir de esa clase. Es decir, que exista una única copia de la variable de instancia, entonces es cuando debe usarse la palabra clave `static`.

```
class Documento extends Pagina {
  static int version = 10;
  }
```

El valor de la variable `version` será el mismo para cualquier objeto instanciado de la clase **Documento**. Siempre que un objeto instanciado de **Documento** cambie la variable `version`, ésta cambiará para todos los objetos.

Si se examinara en este caso la zona de memoria reservada por el sistema para cada uno de los objetos, se encontraría con que todos los objetos comparten la misma zona de memoria para cada una de las variables estáticas, por ello se llaman también *variables de clase*, porque son comunes a la clase, a todos los objetos instanciados de la clase.

Se puede acceder a las variables de clase utilizando el nombre de la clase y el nombre de la variable, no es necesario instanciar ningún objeto de la clase para acceder a las variables de clase.

Se accede a las variables de clase utilizando el nombre de la clase, el nombre de la variable y el operador punto (`.`). La siguiente línea de código, ya archivista, se utiliza para acceder a la variable `out` de la clase **System**. En el proceso, se accede al método *println()* de la variable de clase que presenta una cadena en el dispositivo estándar de salida.

```
System.out.println( "Hola, Mundo" );
```

Es importante recordar que todos los objetos de la clase comparten las mismas variables de clase, porque si alguno de ellos modifica alguna de esas variables de clase, quedarán modificadas para todos los objetos de la clase. Esto puede utilizarse como una forma de comunicación entre objetos.

Constantes

En lenguaje Java se utiliza la palabra clave `final` para indicar que una variable debe comportarse como si fuese *constante*, significando con esto que no se permite su modificación una vez que haya sido declarada e inicializada.

Como es una constante, se ha de proporcionar un valor en el momento de la declaración, por ejemplo:

```
class Elipse {
  final float PI = 3.14159265358979323846;
  . . .
  }
```

Si se intenta modificar el valor de una variable `final` desde el código de la aplicación, se generará un error de compilación.

Si se usa la palabra clave `final` con una variable o clase estática, se pueden crear constantes de clase, haciendo de este modo un uso altamente eficiente de la memoria, porque no se necesitarían múltiples copias de las constantes.

La palabra clave `final` también se puede aplicar a métodos, significando en este caso que los métodos no pueden ser sobrescritos.

Métodos

Los métodos son funciones que pueden ser llamadas dentro de la clase o por otras clases. La implementación de un método consta de dos partes, una *declaración* y un *cuerpo*. La declaración en Java de un método se puede expresar esquemáticamente como:

```
tipoRetorno nombreMetodo( [lista_de_argumentos] ) {
  cuerpoMetodo
  }
```

Los métodos pueden tener numerosos atributos a la hora de declararlos, incluyendo el control de acceso, si es estático o no estático, etc.

La *lista de argumentos* es opcional, puede limitarse a su mínima expresión consistente en dos paréntesis, sin parámetro alguno en su interior. Los parámetros, o argumentos, se utilizan para pasar información al cuerpo del método.

La sintaxis de la declaración completa de un método es la que se muestra a continuación con los ítems opcionales en itálica y los ítems requeridos en negrilla:

especificadorAcceso `static abstract final native synchronized`
`tipoRetorno nombreMetodo(` lista_de_argumentos **`)`**
`throws` *listaExcepciones*

especificadorAcceso, determina si otros objetos pueden acceder al método y cómo pueden hacerlo.

`static`, indica que los métodos pueden ser accedidos sin necesidad de instanciar un objeto del tipo que determina la clase.

`abstract`, indica que el método no está definido en la clase, sino que se encuentra declarado ahí para ser definido en una subclase (sobrescrito).

`final`, evita que un método pueda ser sobrescrito.

`native`, son métodos escritos en otro lenguaje. Java soporta actualmente los lenguajes C y C++.

`synchronized`, se usa en el soporte de la multitarea.

lista_de_argumentos, es la lista opcional de parámetros a pasar al método.

`throws` *listaExcepciones*, excepciones que puede generar y manipular el método.

ARGUMENTOS VARIABLES

Nadie pone en duda la conveniencia de disponer de métodos que admitan un número variable de argumentos. En las versiones anteriores de Java, estos argumentos se pasaban en términos de un contenedor, como un array o un objeto de tipo **Collection**. Las siguientes líneas de código ilustran esta aproximación, sabiendo que la clase **FormatoMensaje** dispone del método:

```
public static formato( String patron,Object args[] );
```

```
String patron = "Hay {0,number,integer} días en un/a {1}.";
Object args = { new Integer(365),"año" };
String cadena = FormatoMensaje.formato( patron,args );
```

```
// La cadena generada es:
// "Hay 365 días en un/a año."
```

Desde la versión J2SE 5, el método anterior se define también de la forma:

```
public static formato( String patron,Object… args );
```

La sintaxis `Object…` indica que el método puede ser invocado pasando un array de elementos de tipo **Object** o pasando una secuencia de elementos de tipo **Object** de cualquier longitud. Es decir, la invocación al método *formato()* ahora puede realizarse de la siguiente forma:

```
String patron = "Hay {0,number,integer} días en un/a {1}.";
String cadena = MensajeFormato.formato(patron,new Integer(365),"año");
// La cadena generada es:
// "Hay 365 días en un/a año."
```

El ejemplo **Java602** muestra el uso de un método que admite un número variable de argumentos en su invocación. Observe el lector que el método no necesita conocer la longitud de la lista de argumentos.

```
class Java602 {
  // Método que imprime un nombre y una lista de argumentos
  public static void ver( String nombre,String... args ) {
    System.out.println( "Nombre: "+nombre );
    for( String s: args ) {
      System.out.println( " -"+s );
      }
    System.out.println( "--------------------------------" );
    }
  public static void main( String args[] ) {
    ver( "Estaciones","Primavera","Verano","Otoño","Invierno" );
    ver( "Año","Enero","Febrero","Marzo","Abril","Mayo","Junio",
      "Julio","Agosto","Septiembre","Octubre","Noviembre",
      "Diciembre" );
    ver( "Semana", "Lunes","Martes","Miércoles","Jueves","Viernes",
      "Sábado","Domingo" );
    }
  }
```

VALOR DE RETORNO DE UN MÉTODO

En Java es imprescindible que a la hora de la declaración de un método, se indique el tipo de dato que ha de devolver. Si no devuelve ningún valor, se indicará el tipo `void` como retorno.

Todos los tipos primitivos se devuelven por valor y todos los objetos se devuelven por referencia. El retorno de la referencia a un objeto es la dirección de la posición en memoria dinámica donde se encuentra almacenado el objeto.

Para devolver un valor se utiliza la palabra clave `return`. Ésta va seguida de una expresión que será evaluada para saber el valor de retorno. Esta expresión puede ser

compleja o puede ser simplemente el nombre de un objeto, una variable de tipo primitivo o una constante.

En el siguiente ejemplo, Java603.java, se ilustra el retorno por valor y por referencia.

```
// Un objeto de esta clase será devuelto por referencia
class miClase {
  int varInstancia = 10;
  }

class Java603 {
  // Método que devuelve por Valor
  int retornoPorValor() {
    // Devuelve un tipo primitivo por valor
    return( 5 );
    }
  // Método que devuelve por Referencia
  miClase retornoPorReferencia() {
    // Devuelve un objeto por referencia
    return( new miClase() );
    }

  public static void main( String args[] ) {
    // Instancia un objeto
    Java603 obj = new Java603();
    System.out.println( "El Valor devuelto es "+
      obj.retornoPorValor() );
    System.out.println(
      "Valor de la variable de instancia en el objeto: "+
      obj.retornoPorReferencia().varInstancia ); // Atención a los
    }                                            // dos puntos
  }
```

Si un programa Java devuelve una referencia a un objeto y esa referencia no es asignada a ninguna variable, o utilizada en una expresión, el objeto se marcará inmediatamente para que el reciclador de memoria en su siguiente ejecución devuelva la memoria ocupada por el objeto al sistema, asumiendo que la dirección no se encuentra ya almacenada en ninguna otra variable.

El tipo del valor de retorno debe coincidir con el tipo de retorno que se ha indicado en la declaración del método; aunque en Java, el tipo actual de retorno puede ser una subclase del tipo que se ha indicado en la declaración del método. Esto es posible porque todas las clases heredan desde un objeto raíz común: **Object**.

En general, se permite almacenar una referencia a un objeto en una variable de referencia que sea una superclase de ese objeto. También se puede utilizar una interfaz como tipo de retorno, en cuyo caso, el objeto devuelto debe implementar esa interfaz.

NOMBRE DEL MÉTODO

El nombre del método puede ser cualquier identificador legal en Java. Java soporta el concepto de *sobrecarga de métodos*, es decir, permite que dos métodos compartan el mismo nombre pero con diferente lista de argumentos, de forma que el compilador pueda diferenciar claramente cuándo se invoca a uno o a otro en función de los parámetros que se utilicen en la llamada al método.

El siguiente fragmento de código muestra una clase Java con cuatro métodos sobrecargados, el último no es legal porque tiene el mismo nombre y lista de argumentos que otro previamente declarado:

```
class MiClase {
  . . .
  void miMetodo( int x,int y ) { . . . }
  void miMetodo( int x ) { . . . }
  void miMetodo( int x,float y ) { . . . }
  // void miMetodo( int a,float b ) { . . . }  // no válido
  }
```

Todo lenguaje de programación orientado a objetos debe soportar las características de encapsulación, herencia y polimorfismo. La sobrecarga de métodos es considerada por algunos autores como polimorfismo en tiempo de compilación. Los métodos sobrecargados siempre deben devolver el mismo tipo.

MÉTODOS DE INSTANCIA

Cuando se incluye un método en una definición de una clase Java sin utilizar la palabra clave `static`, se está generando un *método de instancia*. Aunque cada objeto de la clase no contiene su propia copia de un método de instancia (no existen múltiples copias del método en memoria), el resultado final es como si fuese así, como si cada objeto dispusiese de su propia copia del método.

Cuando se invoca un método de instancia a través de un objeto determinado, si este método está referenciando a variables de instancia de la clase, en realidad se estarán referenciando variables de instancia específicas del objeto que se está invocando.

La llamada a los métodos de instancia en Java se realiza utilizando el nombre del objeto, el operador punto y el nombre del método.

```
miObjeto.miMetodoDeInstancia();
```

Los métodos de instancia tienen acceso tanto a las variables de instancia como a las variables de clase.

MÉTODOS ESTÁTICOS

Cuando una función es incluida en una definición de una clase Java, y se utiliza la palabra `static`, se obtiene un método estático o método de clase.

Lo más significativo de los métodos de clase es que pueden ser invocados sin necesidad de que haya que instanciar ningún objeto de la clase. En Java se puede invocar un método de clase utilizando el nombre de la clase, el operador punto y el nombre del método.

```
MiClase.miMetodoDeClase();
```

En Java, los métodos de clase operan solamente como variables de clase; no tienen acceso a variables de instancia de la clase, a no ser que se cree un nuevo objeto y se acceda a las variables de instancia a través de ese objeto.

Si se observa el siguiente trozo de código de ejemplo:

```
class Documento extends Pagina {
  static int version = 10;
  int numero_de_capitulos;
  static void annade_un_capitulo() {
    numero_de_capitulos++;      // esto no funciona
    }
  static void modifica_version( int i ) {
    version++;                 // esto si funciona
    }
  }
```

la modificación de la variable `numero_de_capitulos` no funciona porque se está violando una de las reglas de acceso al intentar acceder desde un método estático a una variable no estática.

Todas las clases que se derivan, cuando se declaran estáticas, comparten la misma página de variables; es decir, todos los objetos derivados que se generen comparten la misma zona de memoria. Los métodos estáticos se usan para acceder sólo a variables estáticas.

```
class UnaClase {
  int var;
  UnaClase() {
    var = 5;
    }
  unMetodo() {
    var += 5;
    }
  }
```

En el código anterior, si se llama al método *unMetodo()* a través de un puntero a función, no se podría acceder a `var`, porque al utilizar un puntero a función no se pasa implícitamente el puntero al propio objeto (`this`). Sin embargo, sí se podría acceder a

var si fuese estática, porque siempre estaría en la misma posición de memoria para todos los objetos que se creasen de la clase **UnaClase**.

PASO DE PARÁMETROS

En Java todos los métodos deben estar declarados y definidos dentro de la clase, y hay que indicar el tipo y nombre de los argumentos o parámetros que acepta. Los argumentos son como variables locales declaradas en el cuerpo del método que están inicializadas al valor que se pasa como parámetro en la invocación del método.

Todos los argumentos de tipos primitivos en Java deben pasarse por valor, mientras que los objetos deben pasarse por referencia. Cuando se pasa un objeto por referencia, se está pasando la dirección de memoria en la que se encuentra almacenado el objeto.

Si se modifica una variable que haya sido pasada por valor, no se modificará la variable original que se haya utilizado para invocar al método, mientras que si se modifica una variable pasada por referencia, la variable original del método de llamada se verá afectada de los cambios que se produzcan en el método al que se le ha pasado como argumento.

En el ejemplo siguiente, Java604.java, se ilustra el paso de parámetros de tipo primitivo y también el paso de objetos, por valor y por referencia, respectivamente.

```
// Esta clase se usa para instanciar un objeto referencia
class MiClase {
  int varInstancia = 100;
  }

// Clase principal
class Java604 {
  // Método para ilustrar el paso de parámetros
  void pasoVariables( int varPrim,MiClase varRef ) {
    System.out.println( "--> Entrada en la funcion pasoVariables" );
    System.out.println( "Valor de la variable primitiva: "+varPrim );
    System.out.println( "Valor contenido en el objeto: "+
      varRef.varInstancia );
    System.out.println( "-> Modificamos los valores" );
    varRef.varInstancia = 101;
    varPrim = 201;
    System.out.println( "--> Todavia en la funcion pasoVariables" );
    System.out.println( "Valor de la variable primitiva: "+varPrim );
    System.out.println( "Valor contenido en el objeto: "+
      varRef.varInstancia );
    }

  public static void main( String args[] ) {
    // Instanciamos un objeto para acceder a sus métodos
    Java604 aObj = new Java604();
    // Instanciamos un objeto normal
    MiClase obj = new MiClase();
    // Instanciamos una variable de tipo primitivo
```

```
    int varPrim = 200;
    System.out.println( "> Estamos en main()" );
    System.out.println( "Valor de la variable primitiva: "+varPrim );
    System.out.println( "Valor contenido en el objeto: "+
      obj.varInstancia );
    // Llamamos al método del objeto
    aObj.pasoVariables( varPrim,obj );
    System.out.println( "> Volvemos a main()" );
    System.out.println( "Valor de la variable primitiva, todavia : "+
      varPrim );
    System.out.println( "Valor contenido ahora en el objeto: "+
      obj.varInstancia );
    }
}
```

Los métodos tienen acceso directo a las variables miembro de la clase. El nombre de un argumento puede tener el mismo nombre que una variable miembro de la clase. En este caso, la variable local que resulta del argumento del método, *oculta* a la variable miembro de la clase.

Cuando se instancia un método se pasa siempre una referencia al propio objeto que ha llamado al método, es la referencia `this`.

Constructor

El lenguaje Java soporta la sobrecarga de métodos, es decir, que dos o más métodos puedan tener el mismo nombre, pero distinta lista de argumentos en su invocación. Si se sobrecarga un método, el compilador determinará ya en tiempo de compilación, en base a lista de argumentos con que se llame al método, cuál es la versión del método que debe utilizar.

Java también soporta la noción de *constructor*. El constructor es un tipo específico de método que siempre tiene el mismo nombre que la clase y se utiliza para construir objetos de esa clase. No tiene tipo de dato específico de retorno, ni siquiera `void`. Esto se debe a que el tipo específico que debe devolver un constructor de clase es el propio tipo de la clase.

En este caso, pues, no se puede especificar un tipo de retorno, ni se puede colocar ninguna sentencia que devuelva un valor. Los constructores pueden sobrecargarse, y aunque puedan contener código, su función primordial es inicializar el nuevo objeto que se instancia de la clase. En Java, ha de hacerse una llamada explícita al constructor para instanciar un nuevo objeto.

Cuando se declara una clase, se pueden declarar uno o más constructores opcionales que realizan la inicialización cuando se instancia (se crea una ocurrencia) un objeto de dicha clase.

Utilizando el código de la sección anterior, cuando se crea una nueva instancia de **MiClase**, se crean (instancias) todos los métodos y variables, y se llama al constructor de la clase:

```
MiClase mc;
mc = new MiClase();
```

La palabra clave `new` se usa para crear una instancia de la clase. Antes de ser instanciada con `new` no consume memoria, simplemente es una declaración de tipo. Después de ser instanciado un nuevo objeto `mc`, el valor de `i` en el objeto `mc` será igual a 10. Se puede referenciar la variable (de instancia) `i` con el nombre del objeto:

```
mc.i++;       // incrementa la instancia de i de mc
```

Al tener `mc` todas las variables y métodos de **MiClase**, se puede usar la primera sintaxis para llamar al método *Suma_a_i()* utilizando el nuevo nombre de clase **mc**:

```
mc.Suma_a_i( 10 );
```

y ahora la variable `mc.i` vale 21.

Luego cuando se instancia un objeto, siempre se hace una llamada directa al constructor como argumento del operador `new`. Este operador se encarga de que el sistema proporcione memoria para contener al objeto que se va a crear.

Si no se proporciona explícitamente un constructor, el sistema proporcionará uno por defecto que inicializará automáticamente todas las variables miembro a cero o su equivalente, en Java.

Se puede pensar en el constructor de defecto como un método que tiene el mismo nombre que la clase y una lista de argumentos vacía.

Si se proporciona uno o más constructores, el constructor de defecto no se proporcionará automáticamente y si fuese necesaria su utilización, tendría que proporcionarlo explícitamente el programa.

Las dos sentencias siguientes muestran cómo se utiliza el constructor en Java para declarar, instanciar y, opcionalmente, inicializar un objeto:

```
MiClase miObjeto = new MiClase();
MiClase miObjeto = new MiClase( 1,2,3 );
```

Las dos sentencias devuelven una referencia al nuevo objeto que es almacenada en la variable `miObjeto`. También se puede invocar al constructor sin asignar la referencia a una variable. Esto es útil cuando un método requiere un objeto de un tipo determinado como argumento, ya que se puede incluir una llamada al constructor de este objeto en la llamada al método:

```
miMetodo( new MiConstructor( 1,2,3 ) );
```

Aquí se instancia e inicializa un objeto y se pasa a la función. Para que el programa compile adecuadamente debe existir una versión de la función que espere recibir un objeto de ese tipo como parámetro.

Cuando un método o una función comienza su ejecución, todos los parámetros se crean como variables locales automáticas. En este caso, el objeto es instanciado en conjunción con la llamada a la función que será utilizada para inicializar esas variables locales cuando comience la ejecución y luego serán guardadas. Como son automáticas, cuando el método concluye su ejecución, será marcado para su destrucción.

En el siguiente ejemplo, `Java605.java`, se ilustran algunos de los conceptos sobre constructores que se han planteado en esta sección.

```
class MiClase {
  int varInstancia;

  // Éste es el constructor parametrizado
  MiClase( int dato ) {
    // Rellenamos la variable de instancia con los datos que se pasan
    // al constructor
    varInstancia = dato;
    }

  void verVarInstancia() {
    System.out.println( "El Objeto contiene " + varInstancia );
    }
  }

class Java605 {
  public static void main( String args[] ) {
    System.out.println( "Lanzando la aplicacion" );
    // Instanciamos un objeto de este tipo llamando al constructor
    // de defecto
    Java605 obj = new Java605();
    // Llamamos a la función pasándole un constructor parametrizado
    // como parámetro
    obj.miFuncion( new MiClase(100) );
    }

  // Esta función recibe un objeto y llama a uno de sus métodos para
  // presentar en pantalla el dato que contiene el objeto
  void miFuncion( MiClase objeto ){
    objeto.verVarInstancia();
    }
  }
```

HERENCIA

En casos en que se vea involucrada la *herencia*, los constructores toman un significado especial porque lo normal es que la subclase necesite que se ejecute el constructor de la superclase antes que su propio constructor, para que se inicialicen correctamente aquellas variables que deriven de la superclase. La sintaxis para

conseguir esto es sencilla y consiste en incluir en el cuerpo del constructor de la subclase como primera línea de código la siguiente sentencia:

```
super( parametros_opcionales );
```

Esto hará que se ejecute el constructor de la superclase, utilizando los parámetros que se pasen para la inicialización. En el ejemplo `Java606.java`, se ilustra el uso de esta palabra clave para llamar al constructor de la superclase desde una subclase.

Si `super` no aparece como primera sentencia del cuerpo de un constructor, el compilador Java insertará una llamada implícita, *super()*, al constructor de la superclase inmediata. Es decir, el constructor por defecto de la superclase es invocado automáticamente cuando se ejecuta el constructor para una nueva subclase si no se especifica un constructor parametrizado para llamar al constructor de la superclase.

CONTROL DE ACCESO

El control de acceso también tiene un significado especial cuando se trata de constructores. Aunque más adelante se trata a fondo el tema del control de acceso, con referencia a los constructores se puede decir que el control de acceso que se indique determina la forma en que otros objetos van a poder instanciar objetos de la clase. En la siguiente descripción, se indica cómo se trata el control de acceso cuando se tienen entre manos a los constructores:

private

Ninguna otra clase puede instanciar objetos de la clase. La clase puede contener métodos públicos, y estos métodos pueden construir un objeto y devolverlo, pero nadie más puede hacerlo.

protected

Solamente las subclases de la clase pueden crear instancias de ella.

public

Cualquier otra clase puede crear instancias de la clase.

package

Nadie desde fuera del paquete puede construir una instancia de la clase. Esto es útil si se quiere tener acceso a las clases del paquete para crear instancias de la clase, pero sin que nadie más pueda hacerlo, con lo cual se restringe quién puede crear instancias de la clase.

Una instancia de una clase, un objeto, contiene todas las variables y métodos de instancia de la clase y de todas sus superclases. Sin embargo, se puede sobrescribir un método declarado en una superclase, indicando el mismo nombre y misma lista de argumentos.

Si una clase define un método con el mismo nombre, mismo tipo de retorno y misma lista de argumentos que un método de una superclase, el nuevo método sobrescribirá al método de la superclase, utilizándose en todos los objetos que se creen en donde se vea involucrado el tipo de la subclase que sobrescribe el método.

Finalizadores

Java tiene una forma de recuperar automáticamente todos los objetos que se salen del ámbito de la clase. No obstante, proporciona un método que, cuando se especifique en el código de la clase, llamará al reciclador de memoria (*garbage collector*):

```
// Cierra el canal cuando este objeto es reciclado
protected void finalize() {
  close();
  }
```

Cada objeto tiene el método *finalize()*, que es heredado de la clase **Object**. Si se necesitase realizar alguna limpieza asociada con la memoria, se podrá sobrescribir el método *finalize()* y colocar en él el código que sea necesario.

El método *finalize()* siempre se invocará antes de que el reciclador de memoria libere la zona de memoria ocupada por el objeto, pero no hay garantía alguna de que el reciclador de memoria reclame la memoria de un determinado objeto, es decir, no hay garantía de que el método *finalize()* sea invocado.

La regla de oro a seguir es que no se debe poner ningún código que deba ser ejecutado en el método *finalize()*. Por ejemplo, si se necesita concluir la comunicación con un servidor cuando ya no se va a usar un objeto, no debe ponerse el código de desconexión en el método *finalize()*, porque puede que *nunca* sea llamado. Luego, en Java, es responsabilidad del programador escribir métodos para realizar limpieza que no involucre a la memoria ocupada por el objeto y ejecutarlos en el instante preciso. El método *finalize()* y el reciclador de memoria son útiles para liberar la memoria de la pila y debería restringirse su uso solamente a eso, y no depender de ellos para realizar ningún otro tipo de limpieza.

No obstante, Java dispone de dos métodos que permiten asegurarse de la ejecución de los finalizadores. Los dos métodos habilitan la finalización a la salida de la aplicación, haciendo que los finalizadores de todos los objetos que tengan finalizador y que todavía no hayan sido invocados automáticamente, se ejecuten antes de que la Máquina Virtual Java concluya la ejecución de la aplicación. Estos dos métodos son:

runFinalizersOnExit(boolean), método estático de **java.lang.Runtime**, y
runFinalizersOnExit(boolean), método estático de **java.lang.System**

Una clase también hereda de su superclase el método *finalize()*, y en caso necesario, debe llamarse una vez que el método *finalize()* de la clase haya realizado las tareas que se le hayan encomendado, de la forma:

```
super.finalize();
```

En la construcción de un objeto, hay que desplazarse por el árbol de jerarquía, de herencia, desde la raíz del árbol hacia las ramas, y en la finalización, hacerlo al revés, los desplazamientos por la herencia deben ser desde las ramas hacia las superclases hasta llegar a la clase raíz.

CONTROL DE ACCESO

Java implementa cuatro especificadotes de acceso: `private`, `public`, `protected` y `package`.

Por lo tanto, cuando se crea una nueva clase en Java, se puede especificar el nivel de acceso que se quiere para las variables de instancia y los métodos definidos en la clase: `private`, `protected`, `public` y `package`.

La tabla siguiente muestra el nivel de acceso que está permitido a cada uno de los especificadores:

Nivel de Acceso	clase	subclase	paquete	todos
`private`	X			
`protected`	X	X*	X	
`Public`	X	X	X	X
`package`	X		X	

Si se profundiza en el significado de la tabla, se podrá observar que la columna ***clase*** indica que todos los métodos de una clase tienen acceso a todos los otros miembros de la misma clase, independientemente del nivel de acceso especificado.

La columna ***subclase*** se aplica a todas las clases heredadas de la clase, independientemente del paquete en que residan. Los miembros de una subclase tienen acceso a todos los miembros de la superclase que se hayan designado como `public`. El asterisco (`*`) en la intersección *subclase-`protected`* quiere decir que si una clase es a la vez subclase y está en el mismo paquete que la clase con un miembro `protected`, entonces la clase tiene acceso a ese miembro protegido.

En general, si la subclase no se encuentra en el mismo paquete que la superclase, no tiene acceso a los miembros protegidos de la superclase. Los miembros de una subclase no tienen acceso a los miembros de la superclase catalogados como `private` o `package`, excepto a los miembros de una subclase del mismo paquete, que tienen acceso a los miembros de la superclase designados como `package`.

La columna ***paquete*** indica que las clases del mismo paquete tienen acceso a los miembros de una clase, independientemente de su árbol de herencia. La tabla indica que todos los miembros `protected`, `public` y `package` de una clase pueden ser accedidos por otra clase que se encuentre en el mismo paquete.

Colocando dos o más clases en el mismo paquete se hace que la relación `friend`, de amistad, se extienda a todos los métodos de las clases, es decir, si eres amigo de uno de los miembros de una familia, serás amigo automáticamente de todos y cada uno de los componentes de esa familia.

La columna ***todos*** indica que los privilegios de acceso para métodos que no están en la misma clase, ni en una subclase, ni en el mismo paquete, se encuentran restringidos a los miembros públicos de la clase.

Si se observa la misma tabla desde el punto de vista de las filas, se pueden describir los calificadores de los métodos.

private

```
private String NumeroDelDocumentoDeIdentidad;
```

Las variables y métodos de instancia privados sólo pueden ser accedidos desde dentro de la clase. No son accesibles desde las subclases de esa clase. Hay que resaltar, una vez más, que un método de instancia de un objeto de una clase puede acceder a todos los miembros privados de ese objeto, o miembros privados de cualquier otro objeto de la misma clase. Es decir, que en Java el control de acceso existe a nivel de clase, pero no a nivel de objeto de la clase.

Una cuestión interesante puede plantearse a la hora de declarar una clase `private` o `final`, porque en ninguno de los casos se permite que las clases sean sobrescritas. Sin embargo, hay diferencias entre ambas posibilidades. Las clases derivadas no pueden sobrescribir un método `private` por diseño, mientras que la palabra clave `final` indica al compilador que las clases derivadas no pueden sobrescribir un método a partir de ese nivel. Como `private` indica que ya no se pueden sobrescribir métodos, el uso de `private` y `final` juntos es redundante; no causará problemas, pero no implica nada, ya que todo método `private` es considerado automáticamente como `final`.

public

```
public void CualquieraPuedeAcceder(){}
```

Cualquier clase desde cualquier lugar puede acceder a las variables y métodos de instancia públicos.

protected

```
protected void SoloSubClases(){}
```

Sólo las subclases de la clase y nadie más pueden acceder a las variables y métodos de instancia protegidos. Todas las clases de un paquete pueden ver los métodos protegidos de ese paquete.

package

```
void MetodoDeMiPaquete(){}
```

Por defecto, si no se especifica el control de acceso, las variables y métodos de instancia se declaran `package`, lo que significa que son accesibles por todos los objetos dentro del mismo paquete, pero no por los externos al paquete. Aparentemente, parece lo mismo que `protected`; la diferencia estriba en que la designación de `protected` es heredada por las subclases de un paquete diferente, mientras que la designación `package` no es heredada por subclases de paquetes diferentes.

Debido a la complejidad y posible confusión respecto a los niveles de protección que proporciona Java para permitir el control preciso de la visibilidad de variables y métodos, se puede generar otra tabla en base a cuatro categorías de visibilidad entre los elementos de clase:

	private	sin modificador	protected	public
Misma clase	SI	SI	SI	SI
Misma subclase de paquete	NO	SI	SI	SI
Misma no-subclase de paquete	NO	SI	SI	SI
Subclase de diferente paquete	NO	NO	SI	SI
No-subclase de diferente paquete	NO	NO	NO	SI

Y una guía de uso indicaría tener en cuenta lo siguiente:

- Usar `private` para métodos y variables que solamente se utilicen dentro de la clase y que deberían estar ocultas para todo el resto.
- Usar `public` para métodos, constantes y otras variables importantes que deban ser visibles para todo el mundo.
- Usar `protected` si se quiere que las clases del mismo paquete puedan tener acceso a estas variables o métodos.
- Usar la sentencia `package` para poder agrupar las clases en paquetes.
- No usar nada, dejar la visibilidad por defecto (`default`, `package`) para métodos y variables que deban estar ocultas fuera del paquete, pero que deban estar disponibles al acceso desde dentro del mismo paquete. Utilizar `protected` en su lugar si se quiere que esos componentes sean visibles fuera del paquete.

El objeto *this*

Al acceder a variables de instancia de una clase, la palabra clave `this` hace referencia a los miembros de la propia clase. Volviendo al ejemplo de **MiClase**, se puede añadir otro constructor de la forma siguiente:

```
public class MiClase {
  int i;
  public MiClase() {
    i = 10;
    }
  // Este constructor establece el valor de i
  public MiClase( int valor ) {
    this.i = valor;         // i = valor
    }
  public void Suma_a_i( int j ) {
    i = i + j ;
    }
  }
```

Aquí `this.i` se refiere al entero `i` en la clase **MiClase**.

El objeto *super*

Si se necesita llamar al método padre dentro de una clase que ha reemplazado ese método, se podrá hacer referencia al método padre con la palabra clave `super`:

```
import MiClase;
public class MiNuevaClase extends MiClase {
  public void Suma_a_i( int j ) {
    i = i + ( j/2 );
    super.Suma_a_i( j );
    }
  }
```

En el siguiente código, el constructor establecerá el valor de `i` a 10, después lo cambiará a 15 y finalmente el método *Suma_a_i()* de la clase padre **MiClase** lo dejará en 25:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

El concepto `super` es un concepto que no existe en otros lenguajes, al menos no con una implementación similar a Java. Si un método sobrescribe un método de su superclase, se podrá utilizar la palabra clave `super` para eludir la versión sobrescrita de la clase e invocar a la versión original del método en la superclase. Del mismo modo, se puede utilizar `super` para acceder a variables miembro de la superclase.

HERENCIA

La ***herencia*** es el mecanismo por el que se crean nuevos objetos definidos en términos de objetos ya existentes. Por ejemplo, si se tiene la clase **Ave**, se puede crear la subclase **Pato**, que es una especialización de **Ave**.

```
class Pato extends Ave {
  int numero_de_patas;
  }
```

La palabra clave `extends` se usa para generar una subclase (especialización) de un objeto. Un **Pato** es una subclase de **Ave**. Cualquier cosa que contenga la definición de **Ave** será copiada a la clase **Pato**, además, en **Pato** se pueden definir sus propios métodos y variables de instancia. Se dice que **Pato** deriva o hereda de **Ave**.

Además, se pueden sustituir los métodos proporcionados por la clase base. Utilizando el anterior ejemplo de `MiClase`, aquí se presenta una clase derivada sustituyendo a la función *Suma_a_i()*:

```
import MiClase;
public class MiNuevaClase extends MiClase {
  public void Suma_a_i( int j ) {
    i = i + ( j/2 );
    }
  }
```

Ahora cuando se crea una instancia de `MiNuevaClase`, el valor de `i` también se inicializa a 10, pero la llamada al método *Suma_a_i()* produce un resultado diferente:

```
MiNuevaClase mnc;
mnc = new MiNuevaClase();
mnc.Suma_a_i( 10 );
```

Java se diseñó con la idea de que fuera un lenguaje sencillo y, por tanto, se le denegó la capacidad de la herencia múltiple tal como es conocida por los programadores C++, por ejemplo. En este lenguaje se añade cierta complejidad sintáctica cuando se realiza herencia múltiple de varias clases, las cuales comparten una clase base común (hay que declarar dicha clase como virtual y tener bastante cuidado con los constructores), o también cuando las clases base tienen miembros de nombre similar (entonces hay que utilizar especificadores de acceso).

Por ejemplo, de la clase *aparato con motor* y de la clase *animal* no se puede derivar nada, sería como obtener el objeto *toro mecánico* a partir de una *máquina motorizada* (aparato con motor) y un *toro* (animal). En realidad, lo que se pretende es copiar los métodos, es decir, pasar la funcionalidad del toro de verdad al toro mecánico, con lo cual no sería necesaria la herencia múltiple sino simplemente que compartieran la funcionalidad que se encuentra implementada en Java a través de *interfaces*.

SUBCLASES

Como se ha indicado en múltiples ocasiones en este capítulo, cuando se pueden crear nuevas clases por herencia de clases ya existentes, las nuevas clases se llaman ***subclases***, mientras que las clases de donde hereda se llaman ***superclases***.

Cualquier objeto de la subclase contiene todas las variables y todos los métodos de la superclase y sus antecesores.

Todas las clases en Java derivan de alguna clase anterior. La clase raíz del árbol de la jerarquía de clases de Java es la clase **Object**, definida en el paquete **java.lang**. Cada vez que se desciende en el árbol de jerarquía, las clases van siendo más especializadas.

Cuando se desee que nadie pueda derivar de una clase, se indica que es final; y lo mismo con los métodos, si no se desea que se puedan sobrescribir, se les antepone la palabra clave `final`.

Lo contrario de `final` es `abstract`. Una clase marcada como abstracta, únicamente está diseñada para crear subclases a partir de ella, no siendo posible instanciar ningún objeto a partir de una clase abstracta.

SOBRESCRITURA DE MÉTODOS

Para entender en su totalidad el código fuente escrito en lenguaje Java, es imprescindible tener muy claro el concepto de la *redefinición* o *sobrescritura de métodos*. Tanto es así que a continuación se vuelve a insistir sobre la cuestión.

La sobrescritura de métodos es una característica más de la herencia en Java. Es decir, las nuevas clases Java se pueden definir extendiendo clases ya existentes. Aquí surgen los conceptos de **subclase** que sería la clase obtenida, y **superclase**, que sería la clase que está siendo extendida, tal como ya se ha explicado.

Cuando una nueva clase se extiende desde otra que ya existía, todas las variables y métodos que son miembros de la superclase (y todos aquellos miembros de los antecesores de la superclase) serán también miembros de la subclase.

En el supuesto de que en el entorno en que se va a mover la nueva subclase, alguno de los métodos de la superclase (o alguno de sus antecesores) no sea adecuado para los objetos originados de la subclase, es posible reescribir el método en la subclase para que se adapte en su funcionamiento a los objetos del tipo de la subclase.

La reescritura del método dentro de la subclase es lo que se conoce por **sobrescritura de métodos** (*overriding methods*). Todo lo que se requiere para poder sobrescribir un método es utilizar el mismo nombre del método en la subclase, el

mismo tipo de retorno y la misma lista de argumentos de llamada, y en el cuerpo de ese método en la subclase proporcionar un código diferente y específico para las acciones que vaya a realizar sobre objetos del tipo que origina la nueva subclase.

Aunque no se ha discutido todavía la capacidad multitarea o multihilo del lenguaje Java, marcada por la clase **Thread**; sí se puede indicar que hay un método en esta clase, *run()*, que es de vital importancia. La implementación de este método *run()* en la clase **Thread** está completamente vacía, indicando que no se hace nada sino que se limita a definir un método interfaz. No tiene sentido para el método *run()* definir nada por defecto, porque en último lugar será utilizado para realizar lo que la tarea de ejecución necesite y los programadores de la librería de Java no pueden anticipar esas necesidades.

Pero, por otro lado, este método no se puede declarar como abstracto, porque esto podría influir en la instanciación de objetos de la clase **Thread**, y la instanciación de estos objetos es un aspecto sumamente crítico en la programación multitarea en Java. Luego el resultado ha sido que los programadores de *Sun Microsystems* han definido a *run()* como un método vacío.

Como se ha indicado antes, se puede reemplazar completamente la implementación de un método heredado indicando el mismo nombre, la misma lista de argumentos y el mismo tipo de retorno; y colocar en el cuerpo del método el código que realice la función que sea menester en su nueva situación. En el caso del método *run()*, se podría hacer algo como:

```
class MiThread extends Thread {
  void run() {
    // código del método
    }
  }
```

En este fragmento de código, el método *run()* de la clase **MiThread** sobrescribe al método *run()* de la clase **Thread** y proporciona una nueva implementación.

Hay que tener cuidado con la diferencia entre sobrecargar y sobrescribir un método, y entenderla correctamente. Para *sobrecargar un método* hay que duplicar el nombre el método y el tipo que devuelve, pero utilizar una lista de argumentos diferente al original. Para *sobrescribir un método*, no solamente el nombre y el tipo de retorno deben ser iguales, sino que ha de serlo también la lista de argumentos. Es decir, que hay que estar atentos a la lista de argumentos que se indica para un método, en evitación de la sobrecarga cuando en realidad se cree que se está sobrescribiendo, o viceversa. La anotación `@Override` viene a contribuir a la solución de este error, bastante común y muy difícil de detectar en tiempo de ejecución.

Cuando se implementa un método que sobrescribe a otro, es importante el uso de `@Override`, para permitir al compilador la comparación de los dos métodos y asegurar que todas las características necesarias para la sobrescritura de métodos que se indican

en el párrafo anterior se cumplen. En caso de que no sea así, ya en tiempo de compilación, aparecerán los avisos correspondientes al error que se cometa en la sobrescritura del método.

CONVERSIONES IMPLÍCITAS

Con la inclusión de esta característica en el lenguaje Java se trata de dotar al lenguaje de la capacidad de ***autoboxing*** y ***auto-unboxing***, que permiten la ejecución automática de conversiones entre tipos auxiliares, como **Integer**, y los tipos nativos que representan, `int` en este caso.

Las conversiones entre estos tipos son necesarias, por ejemplo, cuando se añade un tipo primitivo a una colección, o cuando se pretende su transferencia a otra máquina virtual mediante RMI.

En el caso de las colecciones, por ejemplo, también se presenta el problema de que las asignaciones no pueden ser comprobadas en tiempo de compilación para detectar errores de moldeo (*casting*), por lo que si se producen errores, será en tiempo de ejecución y se lanzará una excepción de tipo **ClassCastException**.

La solución viene de la mano de las conversiones implícitas, porque si se indica al compilador el tipo de la colección, circunstancia que es imprescindible porque todas las colecciones son *genéricas*, el compilador será capaz de realizar las conversiones por sí solo y detectar los problemas en tiempo de compilación.

El siguiente trozo de código muestra la inserción de elementos en una lista de tipo ArrayList en el formato sin uso de tipos genéricos:

```
List numeros = new ArrayList();
numeros.add( 0,new Integer(5) );
```

Con el uso de colecciones con tipos genéricos y el uso de las conversiones implícitas, el código se reescribiría como se incluye en la clase **Java607**, que se reproduce a continuación:

```
List<Integer> numeros = new ArrayList<Integer>();
numeros.add( 0,5 );   // Uso de autoboxing
```

El compilador automáticamente añadirá el código necesario para realizar la conversión del valor entero `5`, en la clase **Integer** que le corresponde, utilizando la característica de *autoboxing* de Java.

El programa `Java608.java` hace uso de las conversiones implícitas para asignar valores a objetos correspondientes a las clases auxiliares de los tipos primitivos, para luego presentarlos en la salida al ejecutar el programa.

Mediante las conversiones implícitas se elimina la necesidad de conversión explícita de tipos primitivos a las clases correspondientes y viceversa, no siendo necesaria esa cantidad de código extra para llevar a cabo las conversiones. Las conversiones implícitas hacen la programación más sencilla, pero el desarrollador debe tener presente que oculta la creación de objetos. Por ejemplo, en la ejecución del siguiente código, la creación de objetos se realizará solamente la primera vez que se ejecute, en las siguientes se utilizará la copia, porque con los valores entre -127 y 128 (`byte`, `short` e `int`) se utiliza un caché y no se crearán objetos nuevos con el mismo valor:

```
for( int i=0; i < 20; i++ ) {
  lista.add( i );
  }
```

En estas conversiones no se llama a `new` sino al método *valueOf()* de las clases auxiliares, con lo cual los programas son menos intensivos en el uso de la memoria. Además, esta capacidad añadida a Java es una característica de conveniencia que permite un código más claro y aumenta la productividad.

CLASE OBJECT

La clase **Object**, como se ha indicado en ocasiones anteriores, es la clase raíz de todo el árbol de la jerarquía de clases Java, y proporciona un cierto número de métodos de utilidad general que pueden utilizar todos los objetos. La lista completa se puede ver en la documentación del API de Java, aquí solamente se tratarán algunos de ellos.

El método *equals()*

```
public boolean equals( Object obj );
```

Todas las clases que se definan en lenguaje Java heredarán el método *equals()*, que se puede utilizar para comparar dos objetos. Esta comparación no es la misma que proporciona el operador ==, que solamente compara si dos referencias a objetos apuntan al mismo objeto.

El método *equals()* se utiliza para saber si dos objetos distintos son del mismo tipo y contienen los mismos datos. El método devuelve `true` si los objetos son iguales y `false` en caso contrario.

El sistema ya sabe de antemano cómo aplicar el método a todas las clases estándar y a todos los objetos de los que el compilador tiene conocimiento. Por ejemplo, se puede usar directamente para saber si dos objetos **String** son iguales.

Las subclases pueden sobrescribir el método *equals()* para realizar la adecuada comparación entre dos objetos de un tipo que haya sido definido por el programador.

En la aplicación de ejemplo `Java609.java`, se sobrescribe el método para comparar dos objetos de la nueva clase que crea la aplicación.

En la lista de argumentos del método *equals()* hay que pasarle un argumento de tipo **Object**. Si se define un método con un argumento de tipo diferente, se estará sobrecargando el método, no sobrescribiéndolo.

En el ejemplo, una vez que se ejecuta, es necesario hacer un moldeo del argumento al tipo de la clase antes de intentar realizar la comparación. Se utiliza el operador `instanceof` para confirmar que el objeto es del tipo correcto. Uno de los objetos proporcionados para comprobar su equivalencia es de tipo erróneo (**String**) y el método sobrescrito *equals()* indicará que no es un objeto equivalente.

```
class Java609 {
  int miDato;
  // Constructor parametrizado
  Java609( int dato ) {
    miDato = dato;
    }

  public static void main(String args[] ) {
    // Se instancian los objetos que se van a testear
    Java609 obj1 = new Java609( 2 );
    Java609 obj2 = new Java609( 2 );
    Java609 obj3 = new Java609( 3 );
    String obj4 = "Un objeto String";
    // Se realizan las comprobaciones y se presenta por pantalla
    // el resultado de cada una de ellas
    System.out.println( "obj1 equals obj1: "+obj1.equals( obj1 ) );
    System.out.println( "obj1 equals obj2: "+obj1.equals( obj2 ) );
    System.out.println( "obj1 equals obj3: "+obj1.equals( obj3 ) );
    System.out.println( "obj1 equals obj4: "+obj1.equals( obj4 ) );
    }

  // Se sobrescribe el método equals()
  @Override
  public boolean equals( Object arg ) {
    // Se comprueba que el argumento es del tipo adecuado y
    // que no es nulo. Si lo anterior se cumple se realiza
    // la comprobación de equivalencia de los datos.
    // Obsérvese que se ha empleado el operador instanceof
    if ( (arg != null) && (arg instanceof Java609) ) {
      // Hacemos un moldeado del Object general a tipo Java609
      Java609 temp = (Java609)arg;
      // Se realiza la comparación y se devuelve el resultado
      return( this.miDato == temp.miDato );
      }
    else {
      // No es del tipo esperado
      return( false );
      }
    }
  }
```

El método *getClass()*

```
public final native Class getClass();
```

En Java existe la clase **Class**, cuyas instancias representan las clases e interfaces que está ejecutando la aplicación Java. No hay un constructor para la clase **Class**, sus objetos son construidos automáticamente por la *Máquina Virtual Java* cuando las clases son cargadas, o por llamadas al método *defineClass()* del cargador de clases.

El método *getClass()* de la clase **Object** se puede utilizar para determinar la clase de un objeto. Es decir, devuelve un objeto de tipo **Class**, que contiene información importante sobre el objeto que crea la clase. Una vez determinada la clase del objeto, se pueden utilizar los métodos de la clase **Class** para obtener información acerca del objeto como cuál es su nombre o cómo se llama su superclase.

Además, habiendo determinado la clase del objeto, el método *newInstance()* de la clase **Class** puede invocarse para instanciar otro objeto del mismo tipo. El resultado es que el operador `new` será utilizado con un constructor de una clase conocida.

Hay que hacer notar que la última afirmación del párrafo anterior es una situación que el compilador no conoce en tiempo de compilación, es decir, no sabe el tipo del objeto que va a ser instanciado. Por lo tanto, si se necesita referenciar al nuevo objeto, es necesario declarar la variable de referencia del tipo genérico **Object**, aunque el objeto actual tomará todos los atributos de la subclase actual por la que será instanciado.

El método *getClass()* es un método final y no puede ser sobrescrito. Devuelve un objeto de tipo **Class** que permite el uso de los métodos definidos en la clase **Class** sobre ese objeto.

El programa `Java610.java`, ilustra alguna de estas características. Primero, instancia un objeto, mira la clase de ese objeto y utiliza alguno de los métodos de la clase **Class** para obtener y presentar información acerca del objeto. Luego, pregunta al usuario si quiere instanciar un nuevo objeto, instanciando un objeto de tipo **String** en un caso o, en el otro caso, se aplica el método *getClass()* a un objeto existente y utilizando el método *newInstance()* se instancia un nuevo objeto del mismo tipo.

El método *toString()*

```
public String toString();
```

La clase **Object** dispone de este método que puede usarse para convertir todos los objetos conocidos por el compilador a algún tipo de representación de cadena, que dependerá del objeto.

Por ejemplo, el método *toString()* extrae el entero contenido en un objeto **Integer**. De forma similar, si se aplica el método *toString()* al objeto **Thread**, se puede obtener información importante acerca de las tareas y presentarla como cadena.

Este método también se puede sobrescribir, o redefinir, para convertir los objetos definidos por el programador a cadenas. El programa `Java611.java` redefine el método *toString()* de una clase recién definida para que pueda utilizarse en la conversión de objetos de esta clase a cadenas.

```
class Java611 {
  // Se definen las variables de instancia para la clase
  String uno;
  String dos;
  String tres;

  // Constructor de la clase
  Java611( String a,String b,String c ) {
    uno = a;
    dos = b;
    tres = c;
    }

  public static void main( String args[] ) {
    // Se instancia un objeto de la clase
    Java611 obj = new Java611( "Tutorial","de","Java" );
    // Se presenta el objeto utilizando el método sobrescrito
    System.out.println( obj.toString() );
    }

  // Sobrescritura del método toString() de la clase Object
  @Override
  public String toString() {
    // Convierte un objeto a cadena y lo devuelve
    return( uno+" "+dos+" "+tres );
    }
  }
```

Otros métodos

Hay otros métodos útiles en la clase **Object** que se tratan en otras secciones de este Tutorial. Por ejemplo, el método

```
protected void finalize();
```

que se cita en el apartado de *finalizadores*. O también, los métodos que se utilizan en la programación de tareas para hacer que varias de ellas se sincronicen, como son

```
public final void wait();
public final native void wait( long timeout );
public final native void notify();
public final native void notifyAll();
```

que se tratarán cuando se describa la multitarea (*multithreading*) en Java.

CLASES ABSTRACTAS

Una de las características más útiles de cualquier lenguaje orientado a objetos es la posibilidad de declarar clases que definen cómo se utilizan solamente, sin tener que implementar métodos, son las *clases abstractas*. Mediante una clase abstracta se intenta fijar un conjunto mínimo de métodos (el comportamiento) y de atributos, que permitan modelar un cierto concepto, que será refinado y especializado mediante el mecanismo de la herencia. Como consecuencia, la implementación de la mayoría de los métodos de una clase abstracta podría no tener significado. Para resolverlo, Java proporciona los *métodos abstractos*. Estos métodos se encuentran incompletos, sólo cuentan con la declaración y no poseen cuerpo de definición. Esto es muy útil cuando la implementación es específica para cada usuario pero todos los usuarios tienen que utilizar los mismos métodos. Un ejemplo de clase abstracta en Java es la clase **Graphics**:

```
public abstract class Graphics {
public abstract void drawLine( int x1,int y1,int x2,int y2 );
public abstract void drawOval( int x,int y,int width,int height );
public abstract void drawArc( int x,int y,int width,
  int height,int startAngle,int arcAngle );
  . . .
  }
```

Los métodos se declaran en la clase **Graphics**, pero el código que ejecutará el método está en algún otro sitio:

```
public class MiClase extends Graphics {
  public void drawLine( int x1,int y1,int x2,int y2 ) {
    <código para pintar líneas -específico de la arquitectura->
    }
  }
```

Cuando una clase contiene un método abstracto tiene que declararse abstracta. No obstante, no todos los métodos de una clase abstracta tienen que ser abstractos. Las clases abstractas no pueden tener métodos privados (no se podrían implementar) ni tampoco estáticos. Una clase abstracta tiene que derivarse obligatoriamente, no se puede hacer un `new` de una clase abstracta.

INTERFACES

Los métodos abstractos son útiles cuando se quiere que cada implementación de la clase parezca y funcione igual, pero necesita que se cree una nueva clase para utilizar esos métodos abstractos. Las interfaces proporcionan un mecanismo para abstraer los métodos a un nivel superior, lo que permite simular la herencia múltiple de otros lenguajes de programación.

Una interfaz sublima el concepto de clase abstracta hasta su grado más alto. Una interfaz podrá verse simplemente como una forma, es como un molde, solamente

permite declarar nombres de métodos, listas de argumentos, tipos de retorno y adicionalmente miembros datos (los cuales podrán ser únicamente tipos básicos y serán tomados como constantes en tiempo de compilación, es decir, `static` y `final`).

Una interfaz contiene una colección de métodos que se implementan en otro lugar. Los métodos de una interfaz son `public`, `static` y `final`.

La principal diferencia entre *`interface`* y *`abstract`* es que una interfaz proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia. Por ejemplo:

```
public interface VideoClip {
  // comienza la reproducción del vídeo
  void play();
  // reproduce el clip en un bucle
  void bucle();
  // detiene la reproducción
  void stop();
  }
```

Las clases que quieran utilizar la interfaz **VideoClip** utilizarán la palabra `implements` y proporcionarán el código necesario para implementar los métodos que se han definido para la interfaz:

```
class MiClase implements VideoClip {
  void play() {
    <código>
    }
  void bucle() {
    <código>
    }
  void stop() {
    <código>
    }
```

Al utilizar `implements` para la `interface` es como si se hiciese una acción de *copiar-y-pegar* del código de la interfaz, con lo cual no se hereda nada, solamente se pueden usar los métodos.

La ventaja principal del uso de interfaces es que una `interface` puede ser implementada por cualquier número de clases, permitiendo a cada clase compartir la interfaz de programación sin tener que ser consciente de la implementación que hagan las otras clases que implementen la `interface`.

```
class MiOtraClase implements VideoClip {
  void play() {
    <código nuevo>
    }
  void bucle() {
    <código nuevo>
    }
  void stop() {
```

```
<código nuevo>
}
```

Es decir, el aspecto más importante del uso de interfaces es que múltiples objetos de clases diferentes pueden ser tratados como si fuesen de un mismo tipo común, en donde este tipo viene indicado por el nombre de la interfaz.

Aunque se puede considerar el nombre de la interfaz como un tipo de prototipo de referencia a objetos, no se pueden instanciar objetos en sí del tipo interfaz. La definición de una interfaz no tiene constructor, por lo que no es posible invocar el operador `new` sobre un tipo interfaz.

Una interfaz puede heredar de varios interfaces sin ningún problema. Sin embargo, una clase solamente puede heredar de una clase base, aunque puede implementar varias interfaces. La plataforma Java 2 ofrece la posibilidad de definir una interfaz vacía, como es el caso de **Serialize**, que permite *serializar* un objeto. Una interfaz vacía se puede utilizar como un marcador para marcar a una clase con una propiedad determinada.

La aplicación `Java612.java` ilustra algunos de los conceptos referentes a las interfaces. Se definen dos interfaces, en una de ellas se definen dos constantes y en la otra se declara un método *put()* y un método *get()*. Las constantes y los métodos se podrían haber colocado en la misma definición de la interfaz, pero se han separado para mostrar que una clase simple puede implementar dos o más interfaces utilizando el separador coma (,) en la lista de interfaces.

También se definen dos clases, implementando cada una de ellas las dos interfaces. Esto significa que cada clase define el método *put()* y el método *get()*, declarados en una interfaz y hace uso de las constantes definidas en la otra interfaz. Estas clases se encuentran en ficheros separados por exigencias del compilador, los ficheros son `Constantes.java` y `MiInterfaz.java`, y el contenido de ambos ficheros es el que se muestra a continuación:

```
public interface Constantes {
  public final double pi = 6.14;
  public final int constanteInt = 125;
  }

public interface MiInterfaz {
  void put( int dato );
  int get();
  }
```

Es importante observar que en la definición de los dos métodos de la interfaz, cada clase los define de la forma más adecuada para esa clase, sin tener en cuenta cómo estarán definidos en las otras clases.

Una de las clases también define el método *show()*, que no está declarado en la interfaz. Este método se utiliza para demostrar que un método que no está declarado en la interfaz no puede ser accedido utilizando una variable referencia de tipo interfaz.

El método *main()* en la clase principal ejecuta una serie de instanciaciones, invocaciones de métodos y asignaciones destinadas a mostrar las características de las interfaces descritas anteriormente. Si se ejecuta la aplicación, las sentencias que se van imprimiendo en pantalla son autoexplicativas de lo que está sucediendo en el corazón de la aplicación.

Las interfaces son útiles para recuperar las similitudes entre clases no relacionadas, forzando una relación entre ellas. También para declarar métodos que forzosamente una o más clases han de implementar. Y también, para tener acceso a un objeto y para permitir el uso de un objeto sin revelar su clase, son los llamados *objetos anónimos*, que son muy útiles cuando se proporciona o vende un paquete de clases a otros desarrolladores.

Definición

La definición de una interfaz es semejante a la de una clase. La definición de interfaz tiene dos componentes, *declaración* y *cuerpo*. En forma esquemática sería:

```
DeclaracionInterfaz {
  // cuerpoInterfaz
  }
```

Declaración

La mínima declaración consiste en la palabra clave `interface` y el nombre de la interfaz. Por convenio, los nombres de interfaces comienzan con una letra mayúscula, como los nombres de las clases, pero no es obligatorio.

La declaración de interfaz puede tener dos componentes adicionales, el especificador de acceso `public` y la lista de *superinterfaces*.

Una interfaz puede extender otras interfaces. Sin embargo, mientras que una clase solamente puede extender otra clase, una interfaz puede extender cualquier número de interfaces. En el ejemplo se muestra la definición completa de una interfaz, declaración y cuerpo.

```
public interface MiInterfaz extends InterfazA, InterfazB {
  public final double PI = 6.14;
  public final int entero = 125;
  void put( int dato );
  int get();
  }
```

El especificador de acceso `public` indica que la interfaz puede ser utilizada por cualquier clase de cualquier paquete. Si se omite, la interfaz solamente será accesible a aquellas clases que estén definidas en el mismo paquete.

La cláusula `extends` es similar a la de la declaración de clase. Aunque una interfaz puede extender múltiples interfaces, una interfaz no puede extender clases.

La lista de superinterfaces es una lista separada por comas de todas las interfaces que la nueva interfaz va a extender. Una interfaz hereda todas las constantes y métodos de sus superinterfaces, excepto si la interfaz *oculta* una constante con otra del mismo nombre, o si *redeclara* un método con una nueva declaración de ese método.

El cuerpo de la interfaz contiene las declaraciones de los métodos, que terminan en un punto y coma y no contienen código alguno en su cuerpo. Todos los métodos declarados en una interfaz son implícitamente, `public` y `abstract`, y no se permite el uso de `transient`, `volatile`, `private`, `protected` o `syncronized` en la declaración de miembros en una interfaz. En el cuerpo de la interfaz se pueden definir constantes, que serán implícitamente `public`, `static` y `final`.

Implementación

Una interfaz se utiliza definiendo una clase que *implemente* la interfaz a través de su nombre. Cuando una clase implementa una interfaz, debe proporcionar la definición completa de todos los métodos declarados en la interfaz y, también, la de todos los métodos declarados en todas las superinterfaces de esa interfaz.

Una clase puede implementar más de una interfaz, incluyendo varios nombres de interfaces separados por comas. En este caso, la clase debe proporcionar la definición completa de todos los métodos declarados en todas las interfaces de la lista y de todas las superinterfaces de esas interfaces.

En el anterior ejemplo, `Java612.java`, se puede observar una clase que implementa dos interfaces, **Constantes** y **MiInterfaz**.

```
class ClaseA implements Constantes,MiInterfaz {
  double dDato;

  // Define las versiones de put() y get() que utiliza la ClaseA
  public void put( int dato ) {
    // Se usa "pi" de la interfaz Constantes
    dDato = (double)dato * pi;
    System.out.println(
      "En put() de ClaseA, usando pi del interfaz Constantes: "
      + dDato );
    }

  public int get() {
    return( (int)dDato );
    }
```

```
// Se define un método show() para la ClaseA que no está declarado
// en la interfaz MiInterfaz
void show() {
  System.out.println(
    "En show() de ClaseA, dDato = " + dDato );
  }
}
```

Como se ve, esta clase proporciona la definición completa de los métodos *put()* y *get()* de la interfaz **MiInterfaz**, a la vez que utiliza las constantes definidas en la interfaz **Constantes**. Además, la clase proporciona la definición del método *show()* que no está declarado en ninguna de las interfaces, sino que es propio de la clase.

Constantes sin interfaces

Hay ocasiones en las que se declara una interfaz solamente para tener agrupadas una serie de variables; haciendo que una clase implemente esa interfaz se dispondrá de acceso a esas variables.

Esto no es académicamente correcto, porque se está acoplando una clase con una interfaz de modo innecesario, lo que se conoce como *anti-patrón*, pues si se añade un método a la interfaz, todas las clases que la implementen se verán obligadas a implementar ese método, sin razón aparente.

Java dispone a partir del J2SE 5 de la característica ***import static***, que es una forma de conveniencia para extender el modo en que Java importa clases y paquetes. Utilizando esta capacidad se puede indicar al compilador que solamente importe las porciones estáticas de la clase, sin necesidad de heredar de ella. Solamente se puede utilizar con métodos estáticos y tipos enumerados.

Así, por ejemplo a la hora de importar la clase **Color**, que se verá posteriormente al hablar del AWT, para solamente acceder a las constantes que definen los colores, se puede utilizar:

```
import static java.awt.color;
```

y todas las sentencias que normalmente serían, por ejemplo:

```
Color.WHITE;
Color.BLACK;
```

O cualquier otra constante, se podría acceder a ellas directamente: `WHITE`, `BLACK` o la constante que corresponda al color de que se trate.

Lo cierto es que esto tira un poco por tierra el paradigma de los objetos, pero hace el código más legible.

PAQUETES

La palabra clave `package` permite agrupar clases e interfaces. Los paquetes se utilizan en lenguaje Java de forma similar a cómo se utilizan las librerías en otros lenguajes, para agrupar funciones y clases. Los nombres de los paquetes son palabras separadas por puntos y se almacenan en directorios que coinciden con esos nombres.

Por ejemplo, los ficheros siguientes, que contienen código fuente Java: `Applet.java`, `AppletContext.java`, `AppletStub.java`, `AudioClip.java`

contienen en su código la línea:

```
package java.applet;
```

Y las clases que se obtienen de la compilación de los ficheros anteriores se encuentran con el nombre `nombre_de_clase.class`, en el directorio:

```
java/applet
```

En el *Tutorial de Java* de *Sun Microsystems*, se indica que "*los ficheros `.class` del paquete `java.util` están en un directorio llamado `util` de un directorio `java`, situado en algún lugar apuntado por CLASSPATH*".

CLASSPATH es una variable de entorno que indica al sistema dónde debe buscar los ficheros `.class` que necesite. Sin embargo, lo que dice el *Tutorial de Java* de Sun, normalmente no es así, lo cual puede ocasionar confusión. Cuando se utiliza la plataforma Java 2, no existe el directorio que se indica; es más, no es necesaria la declaración de esta variable de entorno.

La no existencia se debe a que Java tiene la capacidad de buscar ficheros comprimidos que utilicen la tecnología **zip** (los ficheros con extensión JAR utilizan este tipo de tecnología de compresión). Esto redunda en un gran ahorro de espacio en disco y además, mantiene la estructura de directorios en el fichero comprimido. Por tanto, se podría parafrasear lo indicado por *Sun* escribiendo que "*en algún lugar del disco, se encontrará un fichero comprimido (zip o jar) que contiene una gran cantidad de ficheros `.class`. Antes de haber sido comprimidos, los ficheros `.class` del paquete `java.util` estaban situados en un directorio llamado `util` de un directorio `java`. Estos ficheros, junto con sus estructuras, se almacenan en el fichero comprimido que debe encontrarse en algún lugar apuntado por CLASSPATH*".

A la hora de crear un paquete hay que tener presente una serie de ideas:

1. La palabra clave `package` debe ser la primera sentencia que aparezca en el fichero, exceptuando, claro está, los espacios en blanco y comentarios.
2. Es aconsejable que todas las clases que vayan a ser incluidas en el paquete se encuentren en el mismo directorio. Como se ha visto, esta recomendación se la puede uno saltar a la torera, pero se corre el riesgo de que aparezcan determinados

problemas difíciles de resolver a la hora de compilar, en el supuesto caso de que no se hile muy fino.

3. Ante todo, recordar que en un fichero únicamente puede existir, como máximo, una clase con el especificador de acceso `public`, debiendo coincidir el nombre del fichero con el nombre de la clase.

Import

Los paquetes de clases se cargan con la palabra clave `import`, especificando el nombre del paquete como ruta y nombre de clase. Se pueden cargar varias clases utilizando un asterisco.

```
import java.Date;
import java.awt.*;
```

El uso del asterisco debe hacerse con cautela, porque al ya de por sí lento compilador, si se pone un asterisco, se cargarán todos los paquetes, lo que hará todavía más lenta la compilación (aunque el asterisco no tiene impacto alguno a la hora de la ejecución, solamente en tiempo de compilación).

Si un fichero fuente Java no contiene ningún *package*, se colocará en el paquete por defecto sin nombre. Es decir, en el mismo directorio que el fichero fuente, y la clase puede ser cargada con la sentencia `import`:

```
import MiClase;
```

Paquetes de Java

El lenguaje Java proporciona una serie de paquetes que incluyen ventanas, utilidades, un sistema de entrada/salida general, herramientas y comunicaciones. Los paquetes básicos más importantes que proporciona la implementación de Java de *Sun Microsystems*, se introducen en el desarrollo de otros apartados del Tutorial. Si el lector desea tener una visión completa de lo que contienen todos los paquetes de la plataforma Java2, debe recurrir a la documentación del API que se proporciona con el JDK, en donde se listan todos los paquetes de la distribución.

CAPÍTULO 7

CLASES JAVA

En un lenguaje orientado a objetos, las clases definen cualquier objeto que se pueda manipular. Java tiene muchas clases útiles, no sólo aquellas que se utilizan para gráficos y sonido, usadas en la construcción de applets y mucho más conocidas.

En este capítulo se describen algunas de las clases que integran el lenguaje Java y que proporcionan características especiales, como el acceso directo a funciones matemáticas o a las propiedades del sistema operativo en el que se ejecuta la aplicación Java. En la descripción de las funcionalidades que proporciona cada una de las clases descritas, se incluyen aplicaciones de ejemplo, de las cuales solamente se reproducen las partes de código relevantes a dichas funcionalidades, o se describe el funcionamiento de la aplicación. El lector debe recurrir al soporte digital que acompaña a este Tutorial para ver el código fuente de los ejemplos en su totalidad, que se encuentran profusamente comentados para ayudar a la comprensión de cada una de las sentencias.

LA CLASE MATH

La clase **Math** representa la librería matemática de Java. Las funciones que contiene son las de todos los lenguajes, parece que se han metido en una clase solamente a propósito de agrupación, por eso se encapsulan en **Math**, y lo mismo sucede con las demás clases que corresponden a objetos que tienen un tipo equivalente (`Character`, `Float`, etc.). El constructor de la clase es privado, por lo que no se pueden crear instancias de la clase. Sin embargo, **Math** es `public` para que se pueda llamar desde cualquier sitio y `static` para que no haya que inicializarla.

Funciones matemáticas

Si se importa la clase, se tiene acceso al conjunto de funciones matemáticas estándar:

`Math.abs( x )`	para `int`, `long`, `float` y `double`
`Math.sin( double a )`	devuelve el seno del ángulo `a` en radianes
`Math.cos( double a )`	devuelve el coseno del ángulo `a` en radianes
`Math.tan( double a )`	devuelve la tangente del ángulo `a` en radianes
`Math.asin( double r )`	devuelve el ángulo cuyo seno es `r`
`Math.acos( double r )`	devuelve el ángulo cuyo coseno es `r`
`Math.atan( double r )`	devuelve el ángulo cuya tangente es `r`
`Math.atan2(double a, Double b)`	devuelve el ángulo cuya tangente es `a/b`
`Math.exp( double x )`	devuelve `e` elevado a `x`
`Math.log( double x )`	devuelve el logaritmo natural de `x`
`Math.sqrt( double x )`	devuelve la raíz cuadrada de `x`
`Math.ceil( double a )`	devuelve el número completo más pequeño mayor o igual que `a`
`Math.floor( double a )`	devuelve el número completo más grande menor o igual que `a`
`Math.rint( double a )`	devuelve el valor `double` truncado de `a`
`Math.pow( double x, Double y )`	devuelve `y` elevado a `x`
`Math.round( x )`	para `double` y `float`
`Math.random()`	devuelve un `double`
`Math.max( a,b )`	para `int`, `long`, `float` y `double`
`Math.min( a,b )`	para `int`, `long`, `float` y `double`
`Math.E`	para la base exponencial, aproximadamente 2,72
`Math.PI`	para `PI`, aproximadamente 3,14

El ejemplo `Java701.java` muestra la utilización de algunas funciones de **Math**:

```
class Java701 {
  public static void main( String args[] ) {
    int x;
    double rand,y,z;
    float max;
    rand = Math.random();
    x = Math.abs( -123 );
    y = Math.round( 123.567 );
    z = Math.pow( 2,4 );
    max = Math.max( (float)1e10,(float)3e9 );
    // Se imprimen en consola los números obtenidos de las operaciones
    // anteriores para comprobar los resultados de la aplicación de
    // los métodos definidos en la clase Math
    System.out.println( rand );
    System.out.println( x );
    System.out.println( y );
    System.out.println( z );
```

```
    System.out.println( max );
    }
  }
```

LA CLASE CHARACTER

Al trabajar con caracteres se necesitan muchas funciones de comprobación y traslación. Estas funciones están englobadas en la clase **Character**. De esta clase sí que se pueden crear instancias, al contrario de lo que sucedía con la clase **Math**.

La primera de las siguientes sentencias creará una variable carácter y la segunda un objeto **Character**:

```
char c;
Character C;
```

A partir del J2SE 5, el lenguaje Java dispone de soporte *Unicode 4.0*. Los caracteres adicionales se codifican como pares de valores *UTF16*, que generan un carácter distinto o *codepoint*.

En general, cuando se utilice una secuencia de caracteres o un objeto **String**, el API de Java manejará los caracteres de modo transparente a la aplicación. Sin embargo, el tipo `char` todavía permanece con 16 bits, con lo cual aquellos métodos que utilizan `char` como argumentos, disponen de versiones complementarias en las que se acepta un valor `int` para poder representar los caracteres *Unicode*.

La clase **Character** tiene métodos para recuperar el carácter actual y el siguiente, para así poder recuperar el carácter *Unicode*. Por ejemplo,

```
String u = "\uD840\uDC08";
System.out.println(  u+ " - " +u.length() );
System.out.println( Character.isHighSurrogate( u.charAt(0) ) );
System.out.println( (int)u.charAt(1) );
```

LAS CLASES DE TIPOS NUMÉRICOS

Cada tipo numérico tiene su propia clase de objetos. Así el tipo `float` tiene el objeto **Float**, al igual que `double` dispone de su propia clase **Double**, `int` de **Integer**, `long` de **Long** y `boolean` de **Boolean**. De la misma forma que con la clase **Character**, se han codificado muchas funciones útiles dentro de los métodos de estas clases.

La clase **Float** dispone de métodos para realizar comprobaciones:

```
boolean b = Float.isNaN( f );
boolean b = Float.isInfinite( f );
```

El método *isNaN()* comprueba si `f` es un *No-Número*. Un ejemplo de no-número es raíz cuadrada de -2.

La especificación del IEEE para la coma flotante trata a estos dos valores de forma muy especial, y en el siguiente ejemplo, `Java704.java`, se crean dos objetos **Double**, uno es *infinito* y el otro es un *no-número*.

```
class Java704 {
  public static void main( String args[] ) {
    Double d1 = new Double( 1/0. );
    Double d2 = new Double( 0/0. );
    System.out.println( d1 + ": " + d1.isInfinite() + ", " +
      d1.isNaN() );
    System.out.println( d2 + ": " + d2.isInfinite() + ", " +
      d2.isNaN() );
    }
  }
```

Si se ejecuta este programa, la salida obtenida es la siguiente:

```
% java Java704
Infinity: true, false
NaN: false, true
```

LA CLASE STRING

Java posee gran capacidad para el manejo de cadenas dentro de sus clases **String** y **StringBuffer**. Un objeto **String** representa una cadena alfanumérica de un valor constante que no puede ser cambiada después de haber sido creada. Un objeto **StringBuffer** representa una cadena cuyo tamaño puede variar, o bien puede ser modificada por programa.

Los objetos de tipo **String** son objetos constantes y, por lo tanto, muy baratos para el sistema. La mayoría de las funciones relacionadas con cadenas esperan valores **String** como argumentos y devuelven valores **String**.

Hay que tener en cuenta que las funciones estáticas no consumen memoria del objeto, con lo cual es más conveniente usar **Character** que `char`. No obstante, `char` se usa, por ejemplo, para leer ficheros que están escritos desde otro lenguaje.

Existen varios constructores para crear nuevas cadenas:

```
String();
String( String value );
String( char value[] );
String( char value[],int offset,int count );
String( byte ascii[],int hibyte );
String( byte ascii[],int hibyte,int offset,int count );
String( StringBuffer buffer );
```

Tal como uno puede imaginarse, las cadenas pueden ser muy complejas, existiendo muchos métodos útiles para trabajar con ellas y, afortunadamente, la mayoría están codificados en la clase **String**, por lo que sería conveniente que el programador tuviese una copia de la declaración de la clase **String** sobre su mesa de

trabajo, para determinar el significado de los parámetros en los constructores y métodos de la clase. Es más, esta necesidad puede extenderse a todas las demás clases, pero claro, teniendo en cuenta el espacio disponible sobre la mesa de trabajo.

Hay que resaltar el hecho de que mientras que el contenido de un objeto **String** no puede modificarse, una referencia a un objeto **String** puede hacerse que apunte a otro objeto **String**, de tal forma que parece que el primer objeto ha sido modificado. Esta característica es la que ilustra el ejemplo `Java706.java`.

```
class Java706 {
  String cadena1 = "ESTA CADENA SE LLAMA cadena1";
  String cadena2 = "Esta cadena se llama cadena2";

  public static void main( String args[] ) {
    Java706 obj = new Java706();
    System.out.println(
      "Los valores originales de las cadenas son:" );
    System.out.println( obj.cadena1 );
    System.out.println( obj.cadena2 );
    System.out.println( "Reemplaza cadena1 con otra cadena" );
    obj.cadena1 = obj.cadena1 + " " + obj.cadena2;
    System.out.println( "Presenta el nuevo valor de cadena1:" );
    System.out.println( obj.cadena1 );
    System.out.println( "Finaliza el programa" );
    }
  }
```

Es importante resaltar que la siguiente sentencia no modifica el objeto original referenciado por la variable `str1`:

```
thisObj.str1 = thisObj.str1 + " " + thisObj.str2;
```

sino que esta sentencia crea un nuevo objeto que es la concatenación de los objetos existentes, y hace que la variable de referencia `str1` apunte al nuevo objeto en lugar de apuntar al objeto original. Ese objeto original queda marcado para que el reciclador de memoria en su siguiente pasada devuelva la memoria que ocupaba al sistema.

Una reflexión especial merecen los arrays de objetos **String**. La siguiente sentencia declara e instancia un array de referencias a cinco objetos de tipo **String**:

```
String miArray = new String[5];
```

Este array no contiene los datos de las cadenas. Sólo reserva una zona de memoria para almacenar las cinco referencias a cadenas. No se guarda memoria para almacenar los caracteres que van a componer esas cinco cadenas, por lo que hay que reservar expresamente esa memoria a la hora de almacenar los datos concretos de las cadenas, tal como se hace en la siguiente línea de código:

```
miArray[0] = new String( "Esta es la primera cadena" );
miArray[1] = new String( "Esta es la segunda cadena" );
```

Con frecuencia se necesita convertir un objeto cualquiera a un objeto **String**, porque sea necesario pasarlo a un método que solamente acepte valores **String**, o por cualquier otra razón. Todas las clases heredan el método *toString()* de la clase **Object** y muchas de ellas lo sobrescriben para proporcionar una implementación que tenga sentido para esas clases. Además, puede haber ocasiones en que sea necesario sobrescribir el método *toString()* para clases propias y proporcionarles un método de conversión adecuado.

La clase **String** posee numerosas funciones para transformar valores de otros tipos de datos a su representación como cadena. Todas estas funciones tienen el nombre de *valueOf*, estando el método sobrecargado para todos los tipos de datos básicos. Ahora bien, la conversión contraria de valor numérico de un tipo básico a cadena no tiene métodos en Java. Las clases que encapsulan a los tipos básicos disponen del método *toString()* heredado de **Object**, que convierte el valor que representan a un objeto de tipo **String**.

LA CLASE STRINGBUFFER

La clase **StringBuffer** dispone de muchos métodos para modificar el contenido de los objetos de tipo **StringBuffer**. Si el contenido de una cadena va a ser modificado en un programa, habrá que sacrificar el uso de objetos **String** en beneficio de **StringBuffer**, que aunque consumen más recursos del sistema, permiten ese tipo de manipulaciones.

La clase **StringBuffer** puede utilizarse en un entorno multitarea sin problemas, pero si en la ejecución de la aplicación Java se puede asegurar que solamente habrá una tarea, un único *thread*, entonces puede utilizarse la clase **StringBuilder**, que es básicamente idéntica a **StringBuffer**, con la diferencia de que no es *thread safe*, con lo cual no tiene la sobrecarga que implica el control de la concurrencia necesario en un entorno multitarea.

Al estar la mayoría de las características de los objetos **StringBuffer** basadas en su tamaño variable, se necesita un nuevo método de creación:

```
StringBuffer();
StringBuffer( int length );
StringBuffer( String str );
```

Se puede crear un **StringBuffer** vacío de cualquier longitud y también utilizar un **String** como punto de partida para un **StringBuffer**.

```
StringBuffer Dos = new StringBuffer( 20 );
StringBuffer Uno = new StringBuffer( "Hola Mundo" );
StringBuffer Cero = new StringBuffer();
```

Parece, aparentemente, más efectivo si se conoce la longitud final del objeto, indicarla cuando se instancia el objeto, que dejar que el sistema instancie el objeto con

una longitud por defecto y luego hacer que se incremente, en tiempo de ejecución, cuando se manipule el objeto. Esto se muestra en las siguientes sentencias, que utilizan el método *length()* de la clase **String** para hacer las cosas más interesantes (un simple entero indicando el tamaño del objeto **StringBuffer** habría funcionado igualmente).

```
StringBuffer str = new StringBuffer("StringBuffer de prueba".length());
Str.append( "StringBuffer de prueba" );
```

Cambio de tamaño

El cambio de tamaño de un **StringBuffer** necesita varias funciones específicas para manipular la longitud de las cadenas:

```
int length();
char charAt( int index );
void getChars( int srcBegin,int srcEnd,char dst[],int dstBegin );
void setLength( int newlength );
void setCharAt( int index,char ch );
int capacity();
void ensureCapacity( int minimumCapacity );
int reverse();
```

El método *capacity()* es particularmente interesante, ya que devuelve la cantidad de espacio que hay actualmente *reservado* para el objeto **StringBuffer**, en contraposición al método *length()*, que devuelve la cantidad de espacio *ocupado* por el objeto **StringBuffer**.

Modificación del contenido

Para cambiar el contenido de un **StringBuffer**, se pueden utilizar dos métodos: *append()* e *insert()*.

En el ejemplo `Java707.java`, se muestra el uso de estos dos métodos:

```
class Java707 {
  public static void main( String args[] ) {
    // Crea un StringBuffer inicializado a un texto por defecto
    StringBuffer str = new StringBuffer( "Tutorial" );
    // Le concatena otro texto, String
    str.append( " de Java" );
    // Imprime en consola el resultado de la concatenación
    System.out.println( str );
    }
  }
```

Operadores de concatenación

Hay que recordar que los operadores "+" y "+=" también se pueden aplicar a cadenas. Ambos realizan una concatenación y están implementados con objetos **StringBuffer**.

Por ejemplo, la sentencia:

```
String s = "¿Qué" + " tal?";
```

es interpretada por el compilador como:

```
String s =
  new StringBuffer().append( "¿Qué" ).append( " tal?" ).toString();
```

y se marcaría el **StringBuffer** para borrarlo, ya que el contenido pasa al objeto **String**. También, la sentencia:

```
s += " por ahí!";
```

sería interpretada por el sistema como:

```
String s =
  new StringBuffer().append( s ).append( " por ahí!" ).toString();
```

y volvería a marcar para borrar el nuevo **StringBuffer** utilizado para crear el objeto **String**.

LA CLASE STRINGTOKENIZER

La clase **StringTokenizer** proporciona uno de los primeros pasos para realizar un análisis gramatical de una cadena de entrada, extrayendo los símbolos que se encuentren en esa cadena. Si se tiene una cadena de entrada cuyo formato es regular y se desea extraer la información que está codificada en ella, **StringTokenizer** es el punto de partida.

Para utilizar esta clase, se necesita un **String** de entrada y un **String** que indique el delimitador a utilizar. Los delimitadores marcan la separación entre los símbolos que se encuentran en la cadena de entrada. Se considera que todos los caracteres de la cadena son delimitadores válidos; por ejemplo, para `<,;:>` el delimitador puede ser una coma, un punto y coma o dos puntos. El conjunto de delimitadores por defecto son los caracteres de espacio habituales: espacio, tabulador, línea nueva y retorno de carro.

Una vez que se ha creado un objeto **StringTokenizer**, se utiliza el método *nextToken()* para ir extrayendo los símbolos consecutivamente. El método *hasMoreTokens()* devuelve `true` cuando todavía quedan símbolos por extraer.

En el ejemplo `Java708.java`, se crea un objeto **StringTokenizer** para analizar gramaticalmente parejas del tipo "`clave=valor`" de un **String**. Los conjuntos consecutivos de parejas *`clave=valor`*, van separados por dos puntos (`:`).

```
class Java708 {
  static String cadena = "titulo=Tutorial de Java:" +
    "idioma=castellano:" +
    "editor=RA-MA:" +
    "autor=Agustin Froufe:" +
    "e-mail=froufe@servidorcorreo.es";

  public static void main( String args[] ) {
    StringTokenizer st = new StringTokenizer( cadena,"=:" );
    while( st.hasMoreTokens() ) {
      String clave = st.nextToken();
      String valor = st.nextToken();
      System.out.println( clave + "\t" + valor );
        }
      }
    }
```

Y la salida de este programita tan sencillo, una vez ejecutado se presenta en la reproducción siguiente:

```
% java Java708
titulo  Tutorial de Java
idioma  castellano
editor  RA-MA
autor   Agustin Froufe
e-mail  froufe@servidorcorreo.es
```

El uso de **StringTokenizer** presenta un problema cuando intervienen cadenas vacías. Por ejemplo, el siguiente trozo de código:

```
// Hay dos comas consecutivas separando cada cadena
StringTokenizer st = new StringTokenizer("Esto,es,una,,cadena","," );
while( st.hasMoreTokens() ) {
  System.out.println( st.nextToken() );
  }
```

generará la salida que se reproduce a continuación:

```
Esto
es
una
cadena
```

Es decir, no se ha respetado la cadena vacía, lo cual puede representar un problema cuando esas cadenas vacías deban ser preservadas. Utilizar **StringTokenizer** para detectarlas hace el código más complejo, por ello, la clase **String** proporciona el método *split()*. Utilizando la invocación a este método, el código anterior se podría reescribir de la forma:

```
String[] tokens = String.split(",","Esto,es,una,,cadena" );
for( String s: tokens ) {
  System.out.println( s );
  }
```

y la salida generada por la ejecución de este código sí respetaría la presencia de la cadena vacía.

El método *split()* tiene la ventaja añadida de que soporta expresiones regulares como argumento para indicar el delimitador de separación entre los elementos a obtener de la cadena origen.

EXPRESIONES REGULARES

En las aplicaciones que se desarrollan en cualquier lenguaje, y por tanto también en Java, suele ser necesario algún tipo de procesamiento de texto, ya sea para buscar palabras, para validar direcciones de correo electrónico, para comprobar claves, para asegurar el formato correcto de datos que se vayan a introducir en una base de datos, o incluso para tareas mucho más complejas como puede ser el comprobar la integridad de un documento XML. Todas estas tareas se realizan a través del reconocimiento y comprobación de *patrones* en las cadenas de texto; tarea en la que la clase **StringTokenizer** dispone de métodos como *charAt()* que permiten ese procesamiento. No obstante, el código que se obtiene es complejo y muy difícil de seguir.

Para subsanar esta circunstancia, la plataforma Java 2 dispone de un paquete destinado al uso de expresiones regulares, **java.util.regex**; que permite realizar todo lo anteriormente expuesto y que incluso permite la utilización de *metacaracteres*, o *comodines*, proporcionando una gran versatilidad a la manipulación de las expresiones.

Una *expresión regular* es un patrón de caracteres que describe a un conjunto de cadenas. El paquete **regex** se puede emplear para buscar, presentar o modificar alguna, o todas, las apariciones de un determinado patrón de caracteres en la cadena de texto que se especifique.

La forma más simple de una expresión regular es una cadena de texto, por ejemplo "Tutorial" o "Java". La comprobación del texto utilizando expresiones regulares permite también comprobar si ese texto está escrito siguiendo una forma sintáctica específica; por ejemplo, en el caso de una dirección de correo electrónico es imprescindible la presencia del carácter `@` con una palabra (al menos) a cada lado de ese carácter.

Para crear expresiones regulares se utilizan caracteres especiales y ordinarios. Los caracteres especiales permitidos son:

```
\$  ^  .  *  +  ?  [' ']  \.
```

cualquier otro carácter que aparezca en una expresión regular y no vaya precedido de la barra invertida, \, se considera un carácter ordinario.

Los caracteres especiales se utilizan para propósitos específicos. Por ejemplo, el carácter punto (.) indica cualquier carácter, excepto el de *nueva línea*. De este modo, si aparece la expresión regular `p.pa` en una búsqueda, se encontrarían correctas palabras como *papa*, *pipa*, *popa* o *pupa*.

Las expresiones que se utilizan en el paquete **regex** siguen la sintaxis del lenguaje *Perl*, por lo que en Java se puede utilizar esa misma sintaxis. Si el lector no está familiarizado con este lenguaje, deberá recurrir a la documentación del API de Java, en donde se describen todas las construcciones admitidas en la creación de expresiones regulares.

Una expresión regular se representa en Java a través de una instancia de la clase **Pattern**; por tanto, una expresión regular, especificada en la cadena correspondiente creada en base a caracteres ordinarios y especiales, debe ser inicialmente compilada en una instancia de la clase **Pattern**. El resultado obtenido se usará para crear un objeto de tipo **Matcher** que será el encargado de comprobar cualquier secuencia arbitraria de caracteres contra esa expresión regular.

El método *compile()* de la clase **Pattern** es el encargado de compilar la cadena correspondiente a la expresión regular y convertirla en un patrón. El método *matcher()* crea un objeto que será el que compruebe la entrada que se le pase contra ese patrón. El método *pattern()* devuelve el patrón resultante de la compilación de la expresión regular. Éstos son los métodos más interesantes de la clase **Pattern**.

Las instancias de la clase **Matcher** se utilizan para comprobar secuencias de caracteres en busca de aquellas que coincidan con el patrón correspondiente a la expresión regular. La entrada de estas instancias se proporciona a través de la interfaz **CharSequence** que soporta una gran variedad de fuentes de datos, de modo que todas esas fuentes de datos dispongan de un formato uniforme y de acceso en modo sólo lectura para ser accesibles mediante objetos de tipo **Matcher**.

Un objeto de tipo **Matcher** se crea a partir de un patrón invocando al método *matcher()* de la clase **Pattern**. Una vez creado, se pueden utilizar los métodos de la clase **Matcher** para realizar operaciones como la búsqueda de la coincidencia completa de la secuencia de caracteres de entrada contra el patrón mediante el método *matches()*. También se puede buscar la primera aparición del patrón a partir del inicio de la secuencia de caracteres, utilizando el método *lookingAt()*, para seguir luego buscando la siguiente aparición del patrón a través del método *find()*.

Cada uno de los métodos anteriores devuelve un *booleano* indicando el éxito o fracaso de la búsqueda de coincidencias, aunque también se pueden invocar métodos específicos del objeto **Matcher** para consultar el estado.

La clase **Matcher** también define métodos para permitir reemplazar los patrones de las expresiones regulares que se encuentren por nuevas cadenas de texto. El ejemplo `Java709.java` realiza la comprobación de una dirección de correo electrónico, asegurándose de que la cadena de caracteres que se pasa es una dirección de correo electrónico válida. No se trata de un ejemplo exhaustivo que compruebe todas las posibles incoherencias en las direcciones de correo electrónico, sino una simple muestra para que el lector pueda familiarizarse con el uso de las expresiones regulares.

El ejemplo `Java710.java` imprime los comentarios de una sola línea que se encuentran incluidos en un fichero de código. Para utilizarlo, el lector debe indicar el fichero que contiene el código fuente del que quiere extraer los comentarios; por defecto, se usa el propio fichero de código del ejemplo.

El código del ejemplo es muy simple. Crea el patrón de búsqueda en base a los caracteres utilizados en Java para delimitar los comentarios de una línea, las dos barras inclinadas, `//`; abre el fichero de código fuente, utiliza un canal para recuperar el contenido y lo contrasta con el patrón. Cada vez que se encuentre un comentario, lo enviará a la pantalla.

LA CLASE FORMATTER

La clase **Formatter** proporciona al lenguaje Java la posibilidad de formatear la salida de los programas, soportando formatos numéricos, de cadena, de fechas y horas, de mensajes localizados para el país de ejecución de la aplicación e incluso permite la justificación o alineación del texto. Si el lector es programador desde hace años, probablemente la descripción anterior le recuerde a la maravillosa función del lenguaje C *printf()*. Y así es, Java recupera las características de esa función y también su nombre, de forma que todo lo que la función *printf()* podía hacer en C, ahora lo puede hacer en Java.

```
System.out.printf( "Número decimal: %.2f",numero );
```

Asumiendo que `numero` es un número en coma flotante, esta sentencia indica al compilador que debe presentarlo con al menos dos dígitos en la parte decimal del número (colocando ceros si fuese necesario).

El método *printf()* también ha sido incorporado a la clase **PrintStream**, una de cuyas implementaciones más comunes es **System.out**. La clase **Java711** muestra una serie de ejemplos en donde se utilizan las características que proporciona esta clase.

```
class Java711 {
  public static void main( String args[] ) {
    // Enviamos todos los mensajes a un objeto de tipo Appendable
    StringBuilder sb = new StringBuilder();
    // Creamos un objeto de tipo Formatter, fijando GERMANY como
    // Locale para que se formatee el número con la coma "," como
    // separador decimal
```

```
    Formatter fmt = new Formatter( sb,Locale.GERMANY );
    // Enviamos un número decimal a la salida
    float num = 123456.07F;
    fmt.format( "El número es: %(,.2f",num );
    System.out.println( sb );
    // Enviamos la fecha actual. %n se utiliza para saltar a la
    // siguiente línea
    System.out.format( "Fecha actual: %1$te de %1$tB de %1$tY%n",
      Calendar.getInstance() );
    // Enviamos la hora actual en formato de 12 horas. %n se utiliza
    // para saltar a la siguiente línea
    System.out.format( "Hora actual: %tr%n",Calendar.getInstance() );
    // Enviamos la hora actual en formato de 24 horas zulú
    System.out.format( "Hora actual: %tT",Calendar.getInstance() );
    }
  }
```

El uso de la clase permite que en lugar de concatenar cadenas, como en:

```
String nombreCompleto = nombre + " " + apellido;
```

sea posible describir la salida y proporcionar argumentos que se coloquen en los lugares convenientes de esa cadena de salida:

```
nombreCompleto = System.out.printf( "%1%s %2$s",nombre,apellido );
```

El rango de opciones de formato va desde algo como `%7.4f` para indicar la precisión y longitud de un número decimal, a `%tT` para dar formato a una hora o `%3$s` para indicar el tercer argumento de tipo cadena. Para indicar una nueva línea, aunque se acepta el carácter estándar de Unix "`\n`", para facilitar la ejecución en cualquier plataforma, se recomienda utilizar `%n`. La documentación del API de Java indica todos los formatos disponibles.

Por tanto, el método *printf()* viene a reemplazar a *println()*, de modo que en lugar de utilizar la concatenación para construir la cadena de salida, se colocan los marcadores de formato que posteriormente serán sustituidos por los valores correspondientes de los argumentos. Estos marcadores incluyen *meta-información* para especificar el formato y los valores que tomen, colocándose después de la cadena que indica el formato, como en el caso de una secuencia variable de argumentos en la invocación de un método.

La utilización de la clase **Formatter** implica la creación de un objeto de tipo **Appendable** para almacenar la salida y la invocación del método *printf()*, o del método *format()*, para colocar el contenido formateado en ese objeto.

Hay muchos objetos que implementan la interfaz **Appendable** y para utilizarlos con **Formatter** basta con pasarlos como argumento al constructor de **Formatter**. En el ejemplo anterior se utiliza la clase **StringBuilder** en lugar de **StringBuffer**, porque la ejecución de la aplicación se produce en una única tarea. Al constructor de

Formatter también hay que pasarle el **Locale** que se utilizará para que formatee el contenido de acuerdo a las características del país que corresponda a ese **Locale**.

Una vez creado el objeto **Formatter**, ya se puede invocar a su método *printf()* con una cadena conteniendo el formato y la lista de argumentos. Si se necesita utilizar un **Locale** diferente al indicado en el constructor, el método *printf()* dispondrá de una forma en la que se puede fijar el **Locale** a utilizar en la llamada.

En secciones anteriores se ha indicado que la clase **PrintStream** contiene las definiciones de objetos como **System.out** y **System.err** para poder escribir en la salida estándar y en la salida de error, respectivamente. Ahora Java introduce más constructores y métodos para soportar el formateo de la salida. Proporciona el método *append()* que implementa la interfaz **Appendable**, por lo que no se puede llamar directamente, aunque sí se pueden invocar directamente los métodos *format()* y *printf()*, que permite el uso de la posibilidad de indicar un número variable de argumentos.

La clase **String** dispone de dos métodos estáticos *format()*, que funcionan de forma semejante a *printf()*; envían una cadena indicando un formato, los argumentos y, opcionalmente, un **Locale**, con lo que se genera una salida en la cual los argumentos son convertidos al formato que se especifica. En este caso, como se trata de métodos de la clase **String**, devuelven un objeto de tipo **String**, mientras que en el caso anterior los métodos enviaban la salida a cualquier objeto de tipo **Stream**. El método *format()* de la clase **String** no hace nada espectacular, simplemente evita tener que utilizar un objeto **Formatter** directamente y la creación de un objeto intermedio de tipo **StringBuilder**.

Si el desarrollador quiere incorporar capacidades de formateo a sus propias clases mediante **Formatter**, entonces es cuando entra en juego la interfaz **Formattable**, que obliga a implementar el método *formatTo()*, cuya definición es:

```
void formatTo( Formatter formatter,int flags,int width,int precision )
```

El ejemplo `Java712.java` muestra el uso de la interfaz **Formattable** en el que se crea una clase con la propiedad `mensaje`, que se imprime en la salida estándar, controlando la anchura y justificación de la cadena de salida.

La ejecución de esta aplicación genera las líneas que se reproducen seguidamente. En las dos primeras se puede observar la diferencia entre utilizar el método *toString()* o recurrir al uso de **Formatter** y en las siguientes se muestra el efecto que producen las opciones de control de anchura y justificación.

```
% java Java712
Corta: Java712@12981f
Cadena Corta: 'Tutorial'
Cadena Larga: 'Tutorial de Java'
Cadena Larga: '     Tutorial*'
Cadena Larga: 'Tutorial*     '
```

LA CLASE SCANNER

Esta clase facilita la tarea de leer y analizar cadenas y tipos básicos de datos en las aplicaciones, incorporando para ello el uso de expresiones regulares.

La clase **Java713** presenta en pantalla el texto del fichero que se pase como argumento en la invocación. Si se compila el fichero `Java713.java` y se ejecuta, la aplicación solicitará el nombre de un fichero de texto y presentará en pantalla su contenido completo.

En el código de la clase se puede observar que la parte más importante es la creación de un objeto de tipo **Scanner** a partir del objeto **File**, originado por la carga del fichero de texto que se indique en la línea de comando. El objeto **Scanner** divide el contenido del fichero en base a un delimitador, que en el ejemplo es el fin de línea; si no se indica ninguno, por defecto se utilizará el espacio en blanco.

Observe el lector que hay otras opciones para detectar el final de una línea, por ejemplo, se podría comprobar mediante los caracteres de retorno de carro y nueva línea usando la expresión "`\r\n|\n`" y también se podrían utilizar los métodos *nextLine()* y *hasNextLine()* de la propia clase **Scanner**.

Un cambio muy simple en el patrón utilizado en el delimitador empleado por **Scanner** proporciona mucha flexibilidad. Por ejemplo, si se indica el delimitador "`\\z`", **Scanner** leerá el fichero de una sola vez, circunstancia que puede ser conveniente en ciertas ocasiones, como a la hora de leer el contenido completo de una página web sin crear varios objetos intermedios. El código siguiente correspondiente a la clase **Java714**, que permite imprimir el contenido de una página web, leerá el contenido completo de la página principal de la web de *Sun* dedicada a Java.

```
URL url = new URL( "http://java.sun.com" );
URLConnection conn = url.openConnection();
Scanner sc = new Scanner( conn.getInputStream() );
sc.useDelimiter( "\\z );
sc.next();
```

En resumen, una vez creado un objeto de tipo **Scanner**, se pueden utilizar los métodos *nextXxx()* para extraer el tipo de variable que se espera, aunque hay dos apreciaciones. La primera es que la clase **Pattern** se puede utilizar para crear una expresión regular utilizada como *token*. La segunda es que el método *nextLine()* avanza hasta pasada la línea actual y devuelve todos los datos de la línea en un objeto de tipo **String**.

LA CLASE PROPERTIES

Hay ocasiones en que es necesario que el sistema sea capaz de leer atributos determinados de ese sistema, así como de leer y/o modificar atributos específicos de la aplicación. Los mecanismos que Java proporciona para resolver estas tareas son tres:

argumentos en la línea de comandos, parámetros en la llamada a applets y las propiedades.

El tópico de Propiedades, **Properties**, es uno de los que nunca cambian técnicamente y que son terriblemente insufribles hasta que se necesitan. Y, cuando son necesarios, lo son de verdad.

Las Propiedades definen un entorno persistente, es decir, se pueden fijar atributos a través de las Propiedades que sean necesarios en la invocación de programas. En otras palabras, si hay alguna información que deba proporcionarse a un programa cada vez que se ejecute, entonces las Propiedades puede ser la solución. Evidentemente, esto implica que haya acceso disponible a sistemas de almacenamiento fijo. Por ahora, las aplicaciones Java tienen acceso a los ficheros de disco en la máquina en que se están ejecutando, mientras que los applets tienen este acceso denegado (siempre en condiciones normales, sin el uso de applets *firmados*).

Para que un applet pueda utilizar de las Propiedades, sería necesario que el applet tuviese acceso a un dispositivo de almacenamiento fijo en el servidor desde el que ha sido descargado; porque, por ahora, el acceso a servidores de red de un applet está limitado al servidor desde el que ha sido descargado.

Las Propiedades se almacenan en variables de instancia del objeto en forma de pares *clave/valor*. Cada propiedad individual se identifica a través de una `clave` y el valor que se asigna a una clave viene determinado por el miembro `valor` del par. Tanto la `clave` como el `valor` son cadenas.

El programa `Java715.java` instancia un objeto **Properties** para presentar en pantalla las Propiedades por defecto del sistema donde se ejecute. Su código completo es el que sigue:

```
class Java715 {
  public static void main( String args[] ) {
    // Instancia y presenta un objeto Properties para presentar
    // las características del sistema
    Properties obj = new Properties( System.getProperties() );
    obj.list( System.out );
    }
  }
```

Como se puede observar, lo fundamental en este programa son las dos líneas del cuerpo del método en donde se instancia un objeto **Properties** y se listan las Propiedades por defecto del sistema que contiene ese objeto.

Las Propiedades del sistema en un momento dado se pueden obtener llamando al método *getProperties()* de la clase **System**. El método *list()* de la clase **Properties** sería luego el utilizado para visualizar el contenido del objeto.

Una vez que se ha creado un objeto **Properties** para el programa, se pueden guardar en un dispositivo de almacenamiento fijo utilizando el método *save()* y posteriormente recuperarlos a través del método *load()*. Las Propiedades del sistema son mantenidas a través de la clase **System**.

LA CLASE RUNTIME

Esta clase encapsula el proceso del intérprete Java que se ejecute. No se puede crear una instancia de **Runtime**; sin embargo, se puede obtener una referencia al objeto **Runtime** que se está ejecutando actualmente llamando al método estático *Runtime.getRuntime()*. Los applets y otros fragmentos de código de los que se desconfíe habitualmente no pueden llamar a ninguno de los métodos de la clase **Runtime** sin activar una **SecurityException**, excepción de seguridad. Algo sencillo que se puede realizar sobre el proceso en ejecución es provocar que se detenga con un código de salida, utilizando el método *exit(int)* de la clase **System**, donde `int` es el código que devolverá el programa.

Aunque Java dispone de su sistema de recogida de basura automática, o liberación de memoria automática para expresarlo con más propiedad y sin términos coloquiales, se podría desear el conocer la cantidad de objetos y el espacio libre que hay para comprobar la eficiencia del código escrito. Para proporcionar esta información, la clase **Runtime** dispone de los métodos *totalMemory()*, que devuelve la memoria total en la Máquina Virtual Java, y *freeMemory()*, que devuelve la cantidad de memoria libre disponible.

También se puede ejecutar el liberador de memoria bajo demanda, llamando al método *gc()* de la clase **System**. Algo aconsejable que se puede intentar es llamar a *gc()*, y después llamar a *freeMemory()* para obtener la utilización de memoria base. A continuación, se ejecuta el código desarrollado y se llama a *freeMemory()* de nuevo para ver cuánta memoria está asignando.

En entornos seguros se puede hacer que Java ejecute otros procesos intensivos en un sistema operativo multitarea. Hay varios constructores del método *exec()* que permiten que se indique el nombre del programa que se va a ejecutar, junto con los parámetros de entrada. El método *exec()* devuelve un objeto **Process**, que se puede utilizar para controlar la interacción del programa Java con el nuevo proceso en ejecución. El problema a la hora de documentar *exec()* es que los programas que se ejecutan son muy dependientes del sistema. Se podría hacer *exec("/usr/bin/ls")* en Solaris/Linux y *exec("notepad")* en Windows 95/98/NT/2000/XP/Vista.

En el ejemplo `Java716.java`, se lanza una aplicación específica de Windows, el editor de textos simple, *Bloc de Notas*, en uno de los archivos fuente de Java. Tómese nota de que *exec()* convierte automáticamente el carácter "/" en el separador de directorios de Windows "\".

```
class Java716 {
  public static void main( String args[] ) {
    Runtime r = Runtime.getRuntime();
    Process p = null;
    String comando[] = { "notepad","Java716.java"};
    // Datos de la memoria del Sistema
    System.out.println( "Memoria Total = "+ r.totalMemory() +
      " Memoria Libre = "+ r.freeMemory() );
    // Intenta ejecutar el comando que se le indica, en este caso
    // lanzar el bloc de notas
    try {
      p = r.exec( comando );
    } catch( Exception e ) {
      System.out.println( "Error ejecutando "+comando[0] );
      }
    }
  }
```

LA CLASE SYSTEM

Hay ocasiones en que se necesita acceder a recursos del sistema, como son los dispositivos de entrada/salida, el reloj del sistema, etc. Java dispone de la clase **System**, que proporciona acceso a estos recursos, independientemente de la plataforma. Es decir, que si se ejecuta un programa en una plataforma diferente a la que se ha desarrollado, no es necesaria ninguna modificación para tener en cuenta las peculiaridades de la nueva plataforma.

La clase **System** es miembro del paquete **java.lang** y en ella se definen los dispositivos estándar de entrada/salida,

```
static PrintStream err;
static InputStream in;
static PrintStream out;
```

y dispone de varios métodos, algunos de los cuales ya se han utilizado en secciones anteriores, sin saber muy bien lo que se estaba haciendo, cosa que se intentará remediar ahora.

No se puede instanciar ningún objeto de la clase **System**, porque es una clase final y todos sus contenidos son privados. Es decir, la clase **System** siempre está ahí disponible para que se pueda invocar cualquiera de sus métodos utilizando la sintaxis de punto (`.`) ya conocida

```
System.out.println( "Hola Java" );
```

Entorno de ejecución

Uno de los métodos útiles de la clase **System** es *getenv()* que devuelve las variables del sistema y su contenido. Este método estaba en las primeras versiones del JDK, siendo posteriormente eliminado y ahora recuperado de nuevo para la plataforma

Java 2. La aplicación `Java717.java` muestra el contenido de la variable de entorno que se pasa como parámetro.

```
class Java717 {
  public static void main( String args[] ) {
    if( args.length > 0 )
      System.out.println( System.getenv( args[0] ) );
    }
  }
```

Ejecutando la aplicación pasando como parámetro la variable de entorno `PROCESSOR_IDENTIFIER`, la aplicación realiza la llamada al método *getenv()* e imprime el valor actual de la variable. La salida producida por la ejecución de la aplicación en el ordenador del autor se reproduce a continuación:

```
% java Java717 PROCESSOR_IDENTIFIER
x86 Family 15 Model 3 Stepping 4, GenuineIntel
```

La plataforma Java 2 proporciona dos versiones del método *getenv()*, la primera es la vista anteriormente y la segunda es la que devuelve la pareja formada por la variable de entorno y el valor que contiene. La aplicación `Java718.java` devuelve todas las variables de entorno del sistema.

```
class Java718 {
  public static void main( String args[] ) {
    for( Map.Entry variable: System.getenv().entrySet() ) {
      System.out.println(variable.getKey()+" / "+variable.getValue());
      }
    }
  }
```

Entrada/Salida estándar

La clase **System** proporciona automáticamente, cuando comienza la ejecución de un programa, un canal para leer del dispositivo estándar de entrada (normalmente, el teclado), un canal para presentar información en el dispositivo estándar de salida (normalmente, la pantalla) y otro canal donde presentar mensajes de error, que es el dispositivo estándar de error (normalmente, la pantalla).

Los tres canales o streams de entrada/salida están controlados por esta clase y se referencian como:

`System.in`	entrada estándar
`System.out`	salida estándar
`System.err`	salida de error estándar

Las variables internas de la clase **System** `out` y `err` son de tipo **PrintStream**, es decir, que tienen acceso a los métodos de la clase **PrintStream**. La clase **PrintStream** proporciona tres métodos para poder visualizar información: *print()*, *println()*, *printf()* y *write()*.

Los dos primeros ya se han utilizado en el Tutorial ampliamente, con lo que no resultan extrañas sentencias como:

```
System.out.print( … );
System.out.println( … );
```

Los métodos *print()* y *println()* son semejantes, la única diferencia es que *println()* coloca automáticamente un carácter *nueva línea* en el *stream*, tras la lista de argumentos que se le pase.

El método *printf()*, que ya se ha descrito al tratar la clase **Formatter**, permite indicar una cadena patrón y una lista de argumentos que serán formateados en la salida de acuerdo al patrón especificado.

El método *write()* se utiliza para escribir bytes en el *stream*, es decir, para escribir datos que no pueden interpretarse como texto, como pueden ser los datos que componen un gráfico.

Los métodos *print()* y *println()* aceptan un argumento de cualquiera de los siguientes tipos: **Object**, **String**, `char[]`, `int`, `long`, `float`, `double` o `boolean`. En cada caso, el sistema convierte el dato en un conjunto de caracteres que transfiere al dispositivo estándar de salida. Si se invoca al método *println()* sin argumentos, simplemente se insertará un carácter *nueva línea* en el stream.

Además, hay versiones sobrecargadas de estos métodos para visualizar adecuadamente objetos de varias clases estándar. Por ejemplo, las siguientes sentencias:

```
Thread obj = new Thread;
System.out.println( obj );
```

producirían la siguiente salida en pantalla:

```
Thread[Thread-4,5,main]
```

Cuando se utilizan *print()* y *println()* sobre un objeto, la salida dependerá de ese objeto; por ejemplo, si se imprime un objeto **String**, se visualizará el contenido de la cadena, y si se imprime un objeto **Thread**, se obtendrá una salida en formato:

```
claseThread[nombre,prioridad,grupo]
```

como en el ejemplo anterior.

Salida del Sistema

Se puede abandonar el intérprete Java llamando al método *exit()* y pasándole un `int` como código de salida. Sin embargo, la invocación de este método está sujeta a restricciones de seguridad, Así, por ejemplo, dependiendo del navegador sobre el que

se esté ejecutando un applet, una llamada a *exit()* desde dentro del applet puede originar una excepción de seguridad, **SecurityException**.

Seguridad

El controlador de seguridad es un objeto que asegura una cierta política de seguridad a la aplicación Java. Se puede fijar el controlador de seguridad para las aplicaciones utilizando el método *setSecurityManager()*, y se puede recuperar el que esté actualmente definido utilizando el método *getSecurityManager()*.

El controlador de seguridad para una aplicación solamente se puede fijar una vez. Normalmente, un navegador fija su controlador de seguridad al arrancar, con lo cual, en acciones posteriores los applets no pueden fijarlo de nuevo, o bien se originará una excepción de seguridad si el applet lo intenta.

CAPÍTULO 8

ALMACENAMIENTO DE DATOS

En este capítulo se mostrarán algunas de las clases Java que permiten almacenar colecciones de objetos, bien de una forma u otra, para facilitar ciertas operaciones que son habituales en la programación.

ARRAYS

Mucho de lo que se podría decir de los arrays se ha comentado antes, aquí sólo interesa el array como almacén de objetos. Hay dos características que diferencian a los arrays de cualquier otro tipo de colección: *eficiencia* y *tipo*. El array es la forma más eficiente que Java proporciona para almacenar y acceder a una secuencia de objetos. El array es una simple secuencia lineal que hace que el acceso a los elementos sea muy rápido, pero el precio que hay que pagar por esta velocidad es que cuando se crea un array su tamaño se fija y no se puede cambiar a lo largo de la vida del objeto. Se puede sugerir la creación de un array de tamaño determinado y luego, ya en tiempo de ejecución, crear otro más grande, mover todos los objetos al nuevo y borrar el antiguo. Esto es lo que hace la clase **Vector**, que se verá posteriormente, pero debido a la carga que supone esta flexibilidad, un **Vector** es menos eficiente que un array en cuestiones de velocidad.

Los otros tipos de colecciones disponibles en Java: **Vector**, **Stack** y **Hashtable** hasta la introducción de los tipos genéricos en el J2SE 5, podían contener cualquier tipo de objeto sin necesidad de que fuese de un tipo definido. Esto era así porque trataban a sus elementos como si fuesen **Object**, la clase raíz de todas las clases Java. Esto era perfecto desde el punto de vista de que se construye solamente una colección, y cualquier tipo de objeto puede ser almacenado en ella. Pero aquí es donde los arrays

eran mucho más eficientes que las colecciones genéricas, porque a la hora de crear un array, siempre hay que indicar el tipo de objetos que va a contener.

El uso de tipos genéricos en las colecciones introducido en el J2SE 5, evita que se creen colecciones genéricas, acercando la eficiencia de cualquier colección a la de los arrays. Es decir, ahora tanto en los arrays como en las colecciones, ya en tiempo de compilación se realizan comprobaciones para que no se almacene en esa colección o array ningún objeto de tipo diferente al que está destinado a contener, ni que se intente extraer un objeto diferente. Desde luego, Java controlará que no se envíe un mensaje inadecuado a un objeto, ya sea en tiempo de compilación como en tiempo de ejecución.

Las colecciones de clases manejan solamente los identificadores, *handles*, de los objetos. Un array, sin embargo, puede crearse para contener tipos básicos directamente, o también identificadores de objetos. Es posible utilizar las clases correspondientes a los tipos básicos, como son **Integer**, **Double**, etc., para colocar tipos básicos dentro de una colección. El colocar una cosa u otra es cuestión de eficiencia, porque es mucho más rápida la creación y acceso en un array de tipos básicos que en uno de objetos del tipo básico.

Desde luego, si se está utilizando un tipo básico y se necesita la flexibilidad que ofrece una colección de expandirse cuando sea preciso, el array no sirve y habrá que recurrir a la colección de objetos del tipo básico.

COLECCIONES

Cuando se necesitan características más sofisticadas para almacenar objetos que las que proporciona un simple array, Java pone a disposición del programador las clases colección: **Vector**, **BitSet**, **Properties**, **Stack** y **Hashtable**, que mejora en la plataforma Java 2 con las nuevas colecciones: **Map**, **List** y **Set** y, a partir del J2SE 5, **Queue**.

El diagrama de la figura 8.1 muestra la jerarquía de clases que integran las colecciones en la plataforma Java 2.

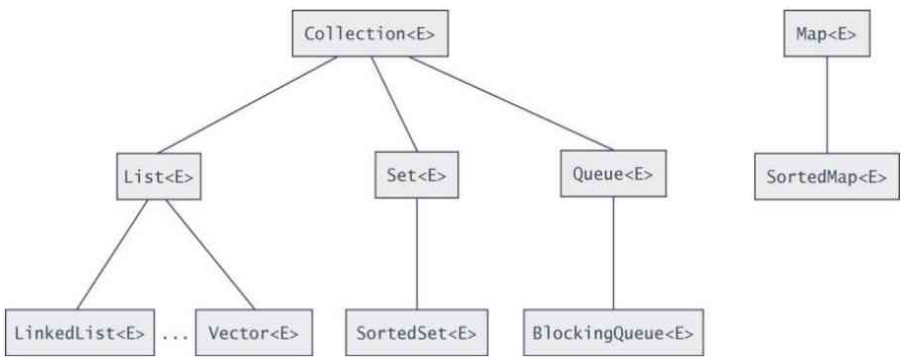


Figura 8.1

En la figura destaca la separación entre **Collection** y **Map**. Además, a la hora de diseñar aplicaciones que utilicen colecciones Java, es importante que el programador recuerde los siguientes puntos:

- La interfaz **Collection** comparte un grupo de objetos y permite duplicarlos.
- La interfaz **Set** extiende a **Collection**, pero prohíbe la presencia de duplicados.
- La interfaz **List** extiende a **Collection**, permite duplicados e introduce un índice posicional.
- La interfaz **Map** es independiente de **Set** y **Collection**, y está orientada a su uso con parejas de tipo *clave-valor*.

Entre otras características, las clases **Collection** se redimensionan automáticamente, por lo que se puede colocar en ellas cualquier número de objetos sin necesidad de tener que ir controlando continuamente en el programa la longitud de la colección.

La gran *desventaja* del uso de las colecciones en Java hasta ahora es que se perdía la información de tipo cuando se colocaba un objeto en una colección, lo cual no se permite a partir de J2SE 5, porque en todas las colecciones es necesario indicar expresamente el tipo de objetos que está destinado a contener cada colección.

ENUMERACIONES

En cualquier clase de colección, debe haber una forma de introducir cosas y otra de sacarlas; después de todo, la principal finalidad de una colección es almacenar cosas. En un **Vector**, el método *addElement()* es la manera en que se colocan objetos dentro de la colección y llamando al método *elementAt()* es cómo se sacan. **Vector** es muy flexible, se puede seleccionar cualquier cosa en cualquier momento así como múltiples elementos utilizando diferentes índices.

Si se quiere empezar a pensar desde un nivel más alto, se presenta un inconveniente: la necesidad de saber el tipo exacto de la colección para utilizarla. Esto no parece que sea malo en principio, pero si se empieza implementando un **Vector** a la hora de desarrollar el programa, y posteriormente se decide cambiarlo a **List**, por eficiencia, entonces sí es problemático.

El concepto de *enumerador*, o *iterador*, que es su nombre más común en OOP, puede utilizarse para alcanzar el nivel de abstracción que se necesita en este caso. Es un objeto cuya misión consiste en moverse a través de una secuencia de objetos y seleccionar aquellos objetos adecuados sin que el programa cliente tenga que conocer la estructura de la secuencia. Además, un *iterador* es normalmente un objeto ligero, *lightweight*, es decir, que consume muy pocos recursos, por lo que hay ocasiones en que presenta ciertas restricciones; por ejemplo, algunos iteradores solamente se pueden mover en una dirección. Y también son fuente de errores, porque aparecen muchas veces en un programa.

La **Enumeration** en Java es un ejemplo de un iterador con esas características, y las cosas que se pueden hacer son:

- Crear una colección para manejar una **Enumeration** utilizando el método *elements()*. Esta **Enumeration** estará lista para devolver el primer elemento en la secuencia cuando se llame por primera vez al método *nextElement()*.
- Obtener el siguiente elemento en la secuencia a través del método *nextElement()*.
- Ver si hay más elementos en la secuencia con el método *hasMoreElements()*.

Con la **Enumeration** no hay que preocuparse del número de elementos que contenga la colección, ya que del control sobre ellos se encargan los métodos *hasMoreElements()* y *nextElement()*.

ITERADORES

Un iterador es un mecanismo que permite recorrer los elementos de una estructura de datos. La interfaz **Iterator** es parte del entorno de las colecciones en Java, que es extendida por la interfaz **ListIterator**, que incorpora métodos para desplazarse hacia delante y hacia atrás y realizar modificaciones en una lista durante las iteraciones.

Para recorrer los elementos de una colección, basta con utilizar la forma del bucle `for` que permite indicar el tipo de objetos que contiene la colección, pero sólo permite recorrer dicha colección, no siendo posible realizar cambios en la colección dentro del bucle `for`.

El uso de iteradores se hace necesario, por tanto, a la hora de manipular colecciones y estructuras de datos complejas en donde no sea obvia la forma de desplazarse por ellas; por ejemplo, estructuras de tipo árbol. Además, los iteradores proporcionan una interfaz uniforme para acceder a tipos de datos diferentes. El ejemplo `Java801.java` es una aplicación muy simple que presenta los datos contenidos en cualquier tipo de colección o mapa. El método *muestraElementos()* es el encargado de recorrer toda la colección de datos:

```
static void muestraElementos( Object obj ) {
  // En el caso de una colección de tipo Map, hay que recuperar las
  // claves y los valores correspondientes a los elementos, para
  // poder utilizar el iterador sobre la lista de elementos que se
  // obtenga
  if( obj instanceof Map )
    obj = ((Map)obj).entrySet();
  // Mostramos los elementos de la colección, siempre que se trate
  // de una colección
  if( obj instanceof Collection ) {
    Collection coleccion = (Collection)obj;
    Iterator iterador = coleccion.iterator();
    while( iterador.hasNext() )
      System.out.println( iterador.next() );
    }
```

```
  // En caso de que no estemos ante una colección, indicamos
  // el fallo
  else {
    System.out.println( "No es una colección válida" );
    }
  }
```

Colecciones de tipo **List** y **Set** extienden la interfaz **Collection**, pero **Map** no, aunque es posible extraer un conjunto de datos de un objeto **Map** y luego iterar sobre él.

Los métodos que proporciona la interfaz **Iterator** son:

hasNext(), devuelve `true` si hay más elementos,
next(), devuelve el siguiente elemento,
remove(), elimina el último elemento devuelto

El ejemplo `Java802.java` intenta modificar una lista mientras se están realizando iteraciones sobre ella. El código del ejemplo es el que se reproduce seguidamente.

```
public class Java802 {
  public static void main( String args[] ) {
    // Creamos la lista de elementos de prueba
    List<String> lista = new ArrayList<String>();
    lista.add( "linea 1" );
    lista.add( "linea 2" );
    lista.add( "linea 3" );
    lista.add( "linea 4" );
    // Creamos el iterador que utilizaremos para recorrer la lista
    Iterator iterador = lista.iterator();
    for( int i=0; iterador.hasNext(); i++ ) {
      String elemento = (String)iterador.next();
      // Intentamos eliminar el tercer elemento de la lista
      if( elemento.equals("linea 3") ) {
        // Esta línea provoca una excepción al eliminar el elemento de
        // tipo ConcurrentModificationException
        lista.remove( i );
        // Si se comenta la línea anterior y se descomenta la
        // siguiente, se elimina la excepción
        // iterador.remove();
        }
      // Imprimimos la lista original
      System.out.println( elemento );
      }
    // Imprimimos la lista sin el elemento eliminado antes
    iterador = lista.iterator();
    while( iterador.hasNext() )
      System.out.println( "--"+iterador.next() );
    }
  }
```

Si el lector ejecuta la aplicación, obtendrá una excepción de tipo **ConcurrentModificationException**, provocada al intentar eliminar un elemento de la lista mientras se iteraba sobre ella. Tanto las implementaciones de la lista como el iterador permiten detectar el problema. Sin embargo, es posible eliminar de forma

segura el elemento utilizando el método *remove()* que proporciona la implementación del iterador. Si el lector comenta la línea de código que invoca al método *remove()* de la lista, y descomenta la línea que llama al método *remove()* del iterador, ya no aparecerá la excepción cuando se elimine el elemento de la lista.

También es posible utilizar un iterador para filtrar la salida de otro iterador. El ejemplo `Java803.java` permite que un iterador filtre el contenido de la lista para obtener solamente aquellos elementos de esa lista que implementan la interfaz **Number**, como pueden ser **Integer** o **Double**. Si el lector observa el código de la clase, el método *main()* resultará interesante porque en él se crea una colección de tipo **List** que admite cualquier tipo de objeto al estar declarada como colección para almacenar objetos de tipo **Object**, que es el objeto más genérico de Java. Utilizando colecciones de **Object**, se estaría obviando el chequeo de tipos que fue introducido con J2SE 5.

```
public static void main( String args[] ) {
  // Creamos la lista de ejemplo
  List<Object> lista = new ArrayList<Object>();
  // La rellenamos con algunos elementos
  lista.add( null );
  lista.add( "linea 1" );
  lista.add( new Double(98.76) );
  lista.add( new Byte((byte)64) );
  lista.add( null );
  lista.add( "linea 2" );
  lista.add( new Long(3245) );
  lista.add( new Integer(69) );
  lista.add( "linea 3" );
  lista.add( new Double(54.32) );
  lista.add( null );
  lista.add( "ultima linea" );
  // Filtramos la lista utilizando el iterador y presentamos en
  // pantalla solamente los elementos de tipo Number
  Iterator iterador = new FiltroNumero( lista.iterator() );
  while( iterador.hasNext() )
    System.out.println( iterador.next() );
  }
```

TIPOS DE COLECCIONES

En la plataforma Java 2 se proporcionan librerías de colecciones que satisfacen casi cualquier proyecto que se emprenda en donde sea necesario el uso de un sistema propio de almacenamiento de datos. A continuación, se verán cada una de estas librerías de colecciones por separado para dar una idea del potencial de que se dispone la plataforma Java.

Vector

El **Vector** es muy simple y fácil de utilizar. Aunque los métodos más habituales en su manipulación son *addElement()* para insertar elementos en el **Vector**,

elementAt() para recuperarlos y *elements()* para obtener una **Enumeration** con el número de elementos del **Vector**, lo cierto es que hay más métodos, y al igual que sucede con todas las librerías de Java, se remite al lector a que consulte la documentación que proporciona *Sun Microsystems* para conocer todos los métodos que componen esta clase.

Las colecciones estándar de Java contienen el método *toString()*, que permite obtener una representación en forma de **String** de sí mismas, incluyendo los objetos que contienen. Dentro de **Vector**, por ejemplo, *toString()* va saltando a través de los elementos del **Vector** y llama al método *toString()* para cada uno de esos elementos. En caso, por poner un ejemplo, de querer imprimir la dirección de la clase, parecería lógico referirse a ella simplemente como **this**, tal como muestra el ejemplo `Java804.java` y que se reproduce en las siguientes líneas.

```
public class Java804 {
  public String toString() {
    return( "Direccion del objeto: "+this+"\n" );
    }

  public static void main( String args[] ) {
    Vector<Java804> v = new Vector<Java804>();
    for( int i=0; i < 10; i++ )
      v.addElement( new Java804() );
    System.out.println( v );
    }
  }
```

El ejemplo es muy sencillo, simplemente crea un objeto de tipo **Java804** y lo imprime; sin embargo, a la hora de ejecutar el programa lo que se obtiene es una secuencia infinita de excepciones. Lo que está pasando es que cuando se le indica al compilador:

```
"Direccion del objeto: "+this
```

el compilador ve un **String** seguido del operador + y otra cosa que no es un **String**, así que intenta convertir **this** en un **String**. La conversión la realiza llamando al método *toString()* que genera una llamada recursiva, llegando a llenarse la pila.

Si realmente se quiere imprimir la dirección del objeto en este caso, la solución pasa por llamar al método *toString()* de la clase **Object**. Así, si en vez de **this** se coloca *super.toString()*, el ejemplo funcionará. En otros casos, este método también funcionará siempre que se esté heredando directamente de **Object** o, aunque no sea así, siempre que ninguna clase padre haya sobrescrito el método *toString()*.

BitSet

Se llama así lo que en realidad es un **Vector** de bits. Lo que ocurre es que está optimizado para uso de bits, en cuanto a tamaño, porque en lo que respecta al tiempo

de acceso a los elementos, es bastante más lento que el acceso a un array de elementos del mismo tipo básico.

Además, el tamaño mínimo de un **BitSet** es de 64 bits. Es decir, que si se está almacenando cualquier otra cosa menor, por ejemplo de 8 bits, se estará desperdiciando espacio.

En un **Vector** normal la colección se expande cuando se añaden más elementos. En el **BitSet** ocurre lo mismo pero ordenadamente. En el ejemplo `Java805.java` muestra el uso de esta colección. Se utiliza el generador de números aleatorios para obtener un `byte`, un `short` y un `int`, que son convertidos a su patrón de bits e incorporados al **BitSet**.

Stack

Un **Stack** es una ***Pila***, o una colección de tipo *LIFO* (*last-in, first-out*). Es decir, lo último que se coloque en la pila será lo primero que se saque.

Los diseñadores de Java, en vez de utilizar un array como bloque para crear un **Stack**, han hecho que **Stack** derive directamente de **Vector**, así que tiene todas las características de un **Vector** más alguna otra propia ya del **Stack**. El ejemplo `Java806.java` es una demostración muy simple del uso de una ***Pila*** que consiste en leer cada una de las líneas de un array y colocarlas en un **String**.

```
public class Java806 {
  static String diasSemana[] = {
    "Lunes", "Martes", "Miercoles", "Jueves",
    "Viernes", "Sabado", "Domingo"};

  public static void main( String args[] ) {
    Stack<String> pila = new Stack<String>();
    for( int i=0; i < diasSemana.length; i++ )
      pila.push( diasSemana[i]+" " );
    System.out.println( "pila = "+pila );
    // Tratando la Pila como un Vector:
    pila.addElement( "Esta es la ultima linea" );
    // Se imprime el elemento 5 (sabiendo que la cuenta empieza por 0)
    System.out.println( "Elemento 5 -> "+pila.elementAt( 5 ) );
    System.out.println( "Elementos introducidos:" );
    while ( !pila.empty() )
      System.out.println( pila.pop() );
    }
  }
```

Cada línea en el array `diasSemana` se inserta en el **Stack** con *push()* y posteriormente se retira con *pop()*. Para ilustrar una afirmación anterior, también se utilizan métodos propios de **Vector** sobre el **Stack**. Esto es posible ya que en virtud de la herencia un **Stack** es un **Vector**, así que todas las operaciones que se realicen sobre un **Vector** también se podrán realizar sobre un **Stack**, como por ejemplo, *elementAt()*.

Hashtable

Un **Vector** permite selecciones desde una colección de objetos utilizando un número, luego parece lógico pensar que hay números asociados a los objetos. Entonces, ¿qué es lo que sucede cuando se realizan selecciones utilizando otros criterios? Un **Stack** podría servir de ejemplo: su criterio de selección es "lo último que se haya colocado en el **Stack**". Si se riza la idea de "selección desde una secuencia", aparece un *mapa*, un *diccionario* o un *array asociativo*. Conceptualmente, todo parece ser un vector, pero en lugar de acceder a los objetos a través de un número, en realidad se utiliza *otro objeto*. Esto lleva a utilizar *claves* y al procesado de claves en el programa. Este concepto se expresa en Java a través de la clase abstracta **Dictionary**. La interfaz para esta clase es muy simple:

`size()`, indica cuántos elementos contiene,
`isEmpty()`, es `true` si no hay ningún elemento,
`put(Object clave,Object valor)`, añade un `valor` y lo asocia con una `clave`
`get(Object clave)`, obtiene el `valor` que corresponde a la `clave` que se indica
`remove(Object clave)`, elimina el par `clave-valor` de la lista
`keys()`, genera una **Enumeration** de todas las claves de la lista
`elements()`, genera una **Enumeration** de todos los valores de la lista

Todo esto es lo que corresponde a un *Diccionario* (**Dictionary**), que no es excesivamente difícil de implementar.

La librería estándar de Java solamente incorpora una implementación de un **Dictionary**, la **Hashtable**. **Hashtable** tiene la misma interfaz básica que **Dictionary**, pero difiere en algo muy importante: la *eficiencia*. Si en un **Dictionary** se realiza un *get()* para obtener un valor, la búsqueda es bastante lenta a través del vector de claves. Aquí es donde la **Hashtable** acelera el proceso, ya que en vez de realizar la tediosa búsqueda línea a línea a través del vector de claves, utiliza un valor especial llamado *código hash*. El *código hash* es una forma de conseguir información sobre el objeto en cuestión y convertirlo en un `int` relativamente único para ese objeto. Todos los objetos tienen un *código hash* y *hashCode()* es un método de la clase **Object**. Una **Hashtable** toma el *hashCode()* del objeto y lo utiliza para cazar rápidamente la clave. El resultado es una impresionante reducción del tiempo de búsqueda. La forma en que funciona una tabla **Hash** se escapa del Tutorial, hay muchos libros que lo explican en detalle, por ahora es suficiente con saber que la tabla **Hash** es un **Dictionary** muy rápido y que un **Dictionary** es una herramienta muy útil.

Para ver el funcionamiento de la tabla **Hash** está el ejemplo `Java807.java`, que intenta comprobar la aleatoriedad del método *Math.random()*. Normalmente, debería producir una distribución perfecta de números aleatorios, pero para poder comprobarlo sería necesario generar una buena cantidad de números aleatorios y comprobar los rangos en que caen. Una **Hashtable** es perfecta para este propósito al asociar objetos

con objetos, en este caso, los valores producidos por el método *Math.random()* con el número de veces en que aparecen esos valores.

```
class Contador {
  int i = 1;
  public String toString() {
    return( Integer.toString( i ) );
    }
  }

class Java807 {
  public static void main( String args[] ) {
    // Declaramos los objetos que contendrá la tabla
    Hashtable<Integer,Contador> ht =
      new Hashtable<Integer,Contador>();
    for( int i=0; i < 10000; i++ ) {
      // Genera un número cuasi-aleatorio entre 0 y 20
      Integer r = new Integer( (int)( Math.random()*20 ) );
      if( ht.containsKey( r ) )
        ( (Contador)ht.get( r ) ).i++;
      else
        ht.put( r,new Contador() );
      }
    System.out.println( ht );
    }
  }
```

En el método *main()*, cada vez que se genera un número aleatorio, se convierte en objeto **Integer** para que pueda ser manejado por la tabla **Hash**, ya que no se pueden utilizar tipos básicos con una colección, porque solamente manejan objetos. El método *containsKey()* comprueba si la clave se encuentra ya en la colección. En caso afirmativo, el método *get()* obtiene el valor asociado a la clave, que es un objeto de tipo **Contador**. El valor `i` dentro del contador se incrementa para indicar que el número aleatorio ha aparecido una vez más.

Si la clave no se encuentra en la colección, el método *put()* colocará el nuevo par clave-valor en la tabla **Hash**. Como **Contador** inicializa automáticamente su variable `i` a 1 en el momento de crearla, ya se indica que es la primera vez que aparece ese número aleatorio concreto.

Para presentar los valores de la tabla **Hash**, simplemente se imprimen. El método *toString()* de **Hashtable** navega a través de los pares clave-valor y llama a método *toString()* de cada uno de ellos. El método *toString()* de **Integer** está predefinido, por lo que no hay ningún problema en llamar a *toString()* para **Contador**. Un ejemplo de ejecución del programa sería la salida que se muestra a continuación:

```
% java Java807
{19=526, 18=533, 17=460, 16=513, 15=521, 14=495, 13=512, 12=483,
  11=488, 10=487, 9=514, 8=523, 7=497, 6=487, 5=489, 3=509, 2=503,
  1=475, 0=505}
```

Al lector le puede parecer superfluo el uso de la clase **Contador**, parece que no hace nada que no haga ya la clase **Integer**. ¿Por qué no utilizar `int` o **Integer**? Pues bien, `int` no puede utilizarse porque como ya se ha indicado, las colecciones solamente manejan objetos, por ello están las clases que envuelven a esos tipos básicos y los convierten en objetos. Sin embargo, lo único que pueden hacer estas clases es inicializar los objetos a un valor determinado y leer ese valor. Es decir, no hay modo alguno de cambiar el valor de un objeto correspondiente a un tipo básico una vez que se ha creado. Esto hace que la clase **Integer** sea inútil para resolver el problema que plantea el ejemplo, así que la creación de la clase **Contador** es imprescindible.

COLECCIONES Y MAPAS

En la plataforma Java 2 hay cambios de diseño respecto a la versión original de Java en las colecciones para hacer su uso más simple y para poner herramientas más poderosas en manos de los programadores. El equipo de desarrollo de *Sun Microsystems*, en aras de seguir manteniendo un diseño simple, en vez de proporcionar una serie de interfaces separadas para soportar capacidades opcionales, define todos los métodos que la implementación de la clase pueda proporcionar.

Sin embargo, hay métodos que son opcionales con lo cual lo anterior presenta el problema de que la implementación de la interfaz debe proporcionar implementaciones para todos los métodos de la interfaz, lo que hace necesario proporcionar al método invocante la información de si alguno de los métodos opcionales no está soportado. Esto se hace lanzando una excepción de tipo **UnsupportedOperationException**.

Además del control de operaciones opcionales en tiempo de ejecución, los iteradores para implementaciones concretas de colecciones son de *fallo rápido*. Es decir, que si se utiliza un objeto de tipo **Iterator** para recorrer una colección mientras esa colección está siendo modificada por otra tarea, se lanza inmediatamente una excepción de tipo **ConcurrentModificationException**, de forma que la próxima vez que el **Iterator** sea invocado esta excepción pueda ser recogida y la tarea involucrada sabrá que la colección ha sido modificada.

Las nuevas colecciones incorporadas en la plataforma Java 2 han sido dotadas de métodos de conveniencia para convertir las colecciones antiguas a las nuevas, de forma que las clases e interfaces utilizadas en versiones anteriores del JDK puedan ser portadas fácilmente al nuevo conjunto de clases e interfaces, y el programador no se vea abrumado ante la necesidad de reescribir código para utilizar las colecciones nuevas. Estos métodos de conveniencia sirven pues como puente cuando se necesita una colección nueva pero ya se dispone de una colección antigua, es decir, cuando se necesita pasar de **Vector** a **List**, una colección de tipo **Hashtable** a **Map** o cualquier **Enumeration** a cualquier tipo de **Collection**.

El siguiente código muestra un ejemplo de uso de uno de estos métodos de conveniencia en el que se pasa de un array a un objeto de tipo **List** del nuevo conjunto de colecciones.

```
String libros[] = {
  "Rescate en el tiempo","Manual de Usuario de Java 2",
  "Comunicaciones Serie","Manual de Modelismo" };
List lista = Arrays.asList( libros );
```

La nueva librería de colecciones de la plataforma Java 2 también parte de la premisa de *almacenar objetos*, y diferencia claramente dos conceptos en base a ello: **Collection** y **Map**.

Las **Colecciones** y los **Mapas** pueden ser implementados de muy diversas formas, en función de las necesidades concretas de programación, pero en realidad todo se reduce a cuatro colecciones: **Map**, **List**, **Set** y **Queue**; y solamente dos o tres implementaciones de cada una de ellas.

Colección (**Collection**): un grupo de elementos individuales, siempre con alguna regla que se les puede aplicar. Una colección **List** almacenará objetos en una secuencia determinada y una colección **Set** no permitirá elementos duplicados.

Mapa (**Map**): un grupo de parejas de objetos clave-valor, como la **Hastable** ya vista. En principio podría parecer que esto es una **Collection** de parejas, pero cuando se intenta implementar, este diseño se vuelve confuso, por lo que resulta mucho más claro tomarlo como un concepto separado. Además, es conveniente consultar porciones de un **Map** creando una **Collection** que represente a esa porción; de este modo, un **Map** puede devolver un **Set** de sus claves, una **List** de sus valores, o una **List** de sus parejas clave-valor. Los **Mapas**, al igual que los arrays, se pueden expandir fácilmente en múltiples dimensiones sin la incorporación de nuevos conceptos: simplemente se monta un **Map** cuyos valores son **Map**, que a su vez pueden estar constituidos por **Map**, etc.

Las interfaces que tienen que ver con el almacenamiento de datos son: **Collection**, **Set**, **List**, **Map** y **Queue**. Un programador deberá crear casi todo su código para entenderse con estas interfaces y solamente necesitará indicar específicamente el tipo de datos que se están usando en el momento de la creación. Por ejemplo, una **Lista** se puede crear de la siguiente forma, indicando el tipo de objetos que contendrá:

```
List<String> lista = new LinkedList<String>();
```

Desde luego, también se puede decidir que `lista` sea una lista enlazada en vez de una lista genérica, así como precisar más el tipo de información de la lista. Mediante el uso de las interfaces se facilita el cambio de la implementación de la lista, de forma que solamente sea necesario cambiar el punto de creación, por ejemplo:

```
List<String> lista = new ArrayList<String>();
```

el resto del código permanece invariable.

En la jerarquía de clases hay algunas clases abstractas que implementan parcialmente una interfaz, a semejanza de las clases adaptadoras de Swing. Si el programador quiere hacer su propio **Set**, por ejemplo, no tendría que empezar con la interfaz **Set** e implementar todos los métodos, sino que podría derivar directamente de **AbstractSet** y ya el trabajo para crear la nueva clase es mínimo.

COLECCIONES

A continuación, se indican los métodos que están disponibles para las colecciones, es decir, lo que se puede hacer con un **Set** o una **List**, aunque las listas tengan funcionalidad añadida que ya se verá, y **Map** no hereda de **Collection**, así que se tratará aparte.

`boolean add(Object)`, asegura que la colección contiene el argumento. Devuelve `false` si no se puede añadir el argumento a la colección.

`boolean addAll(Collection)`, añade todos los elementos que se pasan en el argumento. Devuelve `true` si es capaz de incorporar a la colección cualquiera de los elementos del argumento.

`void clear()`, elimina todos los elementos que componen la colección.

`boolean contains(Object)`, verdadero si la colección contiene el argumento que se pasa como parámetro.

`boolean isEmpty()`, verdadero si la colección está vacía, no contiene elemento alguno.

`Iterator iterator()`, devuelve un **Iterator** que se puede utilizar para desplazamientos a través de los elementos que componen la colección.

`boolean remove(Object)`, si el argumento está en la colección, se eliminará una instancia de ese elemento y se devuelve `true` si se ha conseguido.

`boolean removeAll(Collection)`, elimina todos los elementos que están contenidos en el argumento. Devuelve `true` si consigue eliminar cualquiera de ellos.

`boolean retainAll(Collection)`, mantiene solamente los elementos que están contenidos en el argumento, es lo que sería una intersección en la teoría de conjuntos. Devuelve verdadero en caso de que se produzca algún cambio.

`int size()`, devuelve el número de elementos que componen la colección.

`Object[] toArray()`, devuelve un array que contiene todos los elementos que forman parte de la colección. Éste es un método opcional, lo cual significa que no está implementado para una **Collection** determinada. Si no puede devolver el array, lanzará una excepción de tipo **UnsupportedOperationException**.

List

Hay varias implementaciones de **List**, siendo **ArrayList** la que debería ser la elección por defecto, en caso de no tener que utilizar las características que proporcionan las demás implementaciones.

List (interfaz)

La ordenación es la característica más importante de una Lista, asegurando que los elementos siempre se mantendrán en una secuencia concreta. La Lista incorpora una serie de métodos a la Colección que permite insertar y borrar elementos en medio de la Lista. Además, en la Lista Enlazada se puede generar un **ListIterator** para moverse a través de las lista en ambas direcciones.

ArrayList

Es una Lista volcada en un Array. Se debe utilizar en lugar de **Vector** como almacenamiento de objetos de propósito general. Permite un acceso aleatorio muy rápido a los elementos, pero realiza con bastante lentitud las operaciones de inserción y borrado de elementos en medio de la Lista. Se puede utilizar un **ListIterator** para moverse hacia atrás y hacia delante en la Lista, pero no para insertar y eliminar elementos.

LinkedList

Lista enlazada proporciona un óptimo acceso secuencial, permitiendo inserciones y borrado de elementos en medio de la Lista muy rápidos. Sin embargo, es bastante lento el acceso aleatorio, en comparación con la **ArrayList**. Dispone además de los métodos *addLast()*, *getFirst()*, *getLast()*, *removeFirst()* y *removeLast()*, que no están definidos en ninguna interfaz o clase base y que permiten utilizar la Lista Enlazada como una Pila, una Cola o una Cola Doble.

Set

Set tiene exactamente la misma interfaz que **Collection**, y no hay ninguna funcionalidad extra, como en el caso de las Listas. Un **Set** es exactamente una Colección, pero utilizada en un entorno determinado, es ideal para el uso de la herencia o el polimorfismo. Un **Set** sólo permite que exista una instancia de cada objeto, prohíbe expresamente la existencia de duplicados.

A continuación, se muestran las diferentes implementaciones de **Set**, debiendo utilizarse **HashSet** en general, a no ser que se necesiten las características proporcionadas por alguna de las otras implementaciones.

Set (interfaz)

Cada elemento que se añada a un **Set** debe ser único, ya que el otro caso no se añadirá porque el **Set** no permite almacenar elementos duplicados. Los elementos incorporados al Conjunto deben tener definido el método *equals()*, en aras de

establecer comparaciones para eliminar duplicados. **Set** tiene la misma interfaz que **Collection**, y no garantiza el orden en que se encuentren almacenados los objetos que contenga.

HashSet

Es la elección más habitual, excepto en **Sets** que sean muy pequeños. Para optimizar el uso de espacio, puede indicarse una *capacidad inicial* y un *factor de carga*. Debe tener definido el método *hashCode()*.

LinkedHashSet

Esta implementación mantiene dos listas. La habitual del orden marcado por los códigos *hash*. Y una lista enlazada adicional que mantiene el orden de inserción de elementos en la colección.

ArraySet

Un **Set** encajonado en un **Array**. Esto es útil para **Sets** muy pequeños, especialmente aquellos que son creados y destruidos con frecuencia. Para estos pequeños **Sets**, la creación e iteración consume muchos menos recursos que en el caso del **HashSet**. Sin embargo, el rendimiento es muy malo en el caso de **Sets** con gran cantidad de elementos.

SortedSet

Es un **Set** ordenado, aunque con bastantes limitaciones, por ejemplo, solamente se puede recorrer en orden ascendente.

TreeSet

Es un **Set** ordenado, almacenado en un árbol balanceado. En este caso es muy fácil extraer una secuencia ordenada a partir de un **Set** de este tipo.

NavigableSet

Esta interfaz se introduce en J2SE 6 para subsanar las limitaciones de **SortedSet**. Así, un **NavigableSet** se puede recorrer tanto en orden ascendente como descendente. El ejemplo `Java819.java` muestra cómo se recorre una colección en ambos sentidos. En la clase, se utilizan los métodos *iterator()* y *descendingIterator()* para recorrer la misma colección en orden ascendente y descendente, respectivamente.

Map

Los Mapas almacenan información en base a parejas de valores, formados por una clave y el valor que corresponde a esa clave.

Map (interfaz)

Mantiene las asociaciones de pares clave-valor, de forma que se puede encontrar cualquier valor a partir de la clave correspondiente.

HashMap

Es una implementación basada en una tabla *hash*. Proporciona un rendimiento muy constante a la hora de insertar y localizar cualquier pareja de valores; aunque este rendimiento se puede ajustar a través de los constructores que permite fijar la *capacidad inicial* y el *factor de carga* de la tabla *hash*.

WeakHashMap

Es una implementación específica de **Map** que guarda solamente referencias a claves. Esto permite que se pueda eliminar una pareja clave-valor cuando esa clave ya no esté referenciada fuera del **WeakHashMap**. Utilizando este mapa se pueden mantener fácilmente estructuras de tipo *registro*, donde es importante eliminar una entrada cuando su clave ya no es alcanzada desde ninguna tarea.

LinkedHashMap

Es esta implementación la colección mantiene dos índices a través de sus elementos. Además del índice propio de la relación *hash*, se dispone de una lista enlazada a través de la colección que se encarga de seguir el orden de la inserción de elementos en esa colección; es decir, que un iterador sobre este segundo índice devolverá los elementos en el mismo orden en que se han insertado, y no en el orden que indiquen sus códigos *hash*.

Además, también es posible obtener los elementos de la colección en el orden de acceso, de forma que el primer elemento sea el más recientemente utilizado; en otras palabras, los elementos se añaden al final y las búsquedas en el Mapa comienzan desde el final de la lista enlazada. Esta última circunstancia es muy importante, porque el acceso al Mapa puede cambiar el orden, y si hay varias tareas leyendo del Mapa, es imprescindible la sincronización del acceso.

Los tiempos de iteración utilizando **LinkedHashMap** no se ven afectados por la capacidad del Mapa, al contrario de lo que ocurre con el **HashMap** normal. Elegir la implementación **LinkedHashMap** con gran capacidad no restará en absoluto velocidad de acceso a los elementos del Mapa a través de iteradores; sin embargo, la incorporación de nuevos elementos al Mapa implica añadir cada par de elementos separadamente.

El ejemplo `Java810.java` construye varias colecciones de los tipos aquí referenciados en base a los nombres de los meses en español e inglés, presentando la lista de meses de forma ordenada, desordenada o en función del orden de acceso a algunos elementos de la lista.

ArrayMap

Es un Mapa circunscrito en un **Array**. Proporciona un control muy preciso sobre el orden de iteración. Está diseñado para su utilización con Mapas muy pequeños, especialmente con aquellos que se crean y destruyen muy frecuentemente. En este tipo de Mapas la creación e iteración consume muy pocos recursos del sistema, y muchos menos que el **HashMap**. El rendimiento cae estrepitosamente cuando se intentan manejar Mapas grandes.

TreeMap

Es una implementación basada en un árbol balanceado. Cuando se observan las claves o los valores, se comprueba que están colocados en un orden concreto, determinado por **Comparable** o **Comparator**. Lo importante de un **TreeMap** es que se pueden recuperar los elementos en un determinado orden. **TreeMap** es el único mapa que define el método *subMap()*, que permite recuperar una parte del árbol solamente.

Queue

Las Colas son colecciones que generalmente, aunque no necesariamente, ordenan sus elementos en una estructura *FIFO* (*first-in, first-out*), excepto las Colas ordenadas por prioridad.

Queue (interfaz)

La interfaz **Queue**, que permite implementar Colas, proporciona métodos como *poll()* o *remove()* para permitir eliminar un elemento de la Cola. También proporciona el método *offer()* que permite añadir un elemento, si es posible. Este método *offer()* difiere del método *add()* de **Collection** en que se ha diseñado para situaciones en las que un fallo es una situación normal, en lugar de una excepción como sucede, por ejemplo, en las Colas de capacidad fija.

Las implementaciones de **Queue** normalmente no definen métodos *equals()* y *hashCode()* en base a los elementos de la Cola, sino que los heredan de la clase **Object**. Esto es así porque las comparaciones basadas en los elementos de la Cola no siempre están bien definidas, ya que puede haber Colas con el mismo tipo de elementos pero distinto criterio de ordenación.

AbstractQueue

Es una clase abstracta que no puede instanciarse directamente. Proporciona el esqueleto de algunas de las operaciones que define la interfaz **Queue**. Las implementaciones son adecuadas para las ocasiones en que no se permiten elementos nulos en la Cola.

Una implementación de una Cola extendiendo esta clase, debe definir el método *offer()* no permitiendo la inserción de elementos `null`. También debe definir los métodos *peek()*, *poll()*, *size()* e *iterator()* que soporte la acción *remove()*. Los métodos

adicionales serán sobrescritos. Si estos requisitos no se dan, deberá considerarse la extensión de la clase **AbstractCollection** en lugar de **AbstractQueue**.

BlockingQueue (interfaz)

La interfaz **BlockingQueue** define métodos bloqueantes, comunes en la programación concurrente. Estos métodos esperan a que los elementos aparezcan en la Cola o a que haya espacio disponible en la misma.

Las Colas generalmente están restringidas a un tamaño determinado, por ello hay implementaciones que proporcionan métodos como *add()* y *poll()* para añadir y quitar elementos de la Cola, pero su existencia no es imperativa. Por ejemplo, la clase **ArrayBlockingQueue** tiene una capacidad de almacenamiento fija, mientras que la clase **LinkedBlockingQueue** permite que se especifique el tamaño máximo, por defecto no tiene límite. Ambas clases implementan la interfaz **BlockingQueue**.

PriorityQueue

Implementa la interfaz **Queue** para crear una Cola en la que los elementos están ordenados de forma automática. En función del constructor que se utilice, el orden de los elementos puede estar basado en el orden natural o en el orden establecido mediante la clase **Comparator**. Por ejemplo, cuando se ordena una Cola en la que se almacenen elementos de tipo **String**, el orden natural de ordenación que se utiliza es alfabéticamente.

DelayQueue

Hay ocasiones en las que los elementos colocados en una Cola deben permanecer en esa Cola una determinada cantidad de tiempo antes de poder ser retirados de ella. Este comportamiento es el que proporciona la clase **DelayQueue**.

SynchronousQueue

La implementación **SynchronousQueue** crea una Cola que no tiene capacidad interna. Esto es intencionado y significa que en este tipo de Cola, una petición para incorporar un elemento debe esperar a que se produzca una petición de retirada de un elemento desde otra tarea.

Esta clase se utiliza cuando un objeto que se está ejecutando en una tarea debe sincronizarse con otro objeto de una tarea distinta, para obtener algún tipo de información o evento.

El ejemplo `Java811.java` ilustra el uso de la clase **BlockingQueue** creando las clases anidadas correspondientes a la clásica relación productor/consumidor, de modo que el objeto **Productor** coloca elementos en la cola a intervalos aleatorios de tiempo mientras que las instancias de **Consumidor** van retirando elementos de la Cola.

En la ejecución del ejemplo, las cuatro instancias de **Consumidor** van retirando los elementos de la Cola que va insertando **Productor**, de forma secuencial. Un objeto **Consumidor** bloquea a los otros el acceso a la Cola hasta que es capaz de retirar el objeto que le corresponde.

OPERACIONES NO SOPORTADAS

Es posible convertir un array en una Lista utilizando el método estático *Arrays.toList()*, tal como se muestra en el ejemplo `Java812.java`.

```
public class Java812 {
  private static String s[] = {
    "uno", "dos", "tres", "cuatro", "cinco",
    "seis", "siete", "ocho", "nueve", "diez",
    };
  static List<String> a = Arrays.asList( s );
  static List<String> a2 =
    Arrays.asList( new String[] { s[3],s[4],s[5]} );

  // Se desplaza a través de una Lista utilizando un Iterador
  public static void print( Collection c ) {
    for( Iterator x=c.iterator(); x.hasNext(); )
      System.out.print( x.next()+" " );
    System.out.println();
    }

  public static void main( String args[] ) {
    print( a );        // Iteración
    System.out.println(
      "a.contains("+s[0]+") = "+a.contains(s[0]) );
    System.out.println(
      "a.containsAll(a2) = "+a.containsAll(a2) );
    System.out.println( "a.isEmpty() = "+a.isEmpty() );
    System.out.println(
      "a.indexOf("+s[5]+") = "+a.indexOf(s[5]) );
    // Movimientos hacia atrás
    ListIterator<String> lit = a.listIterator( a.size() );
    while( lit.hasPrevious() )
      System.out.print( lit.previous() );
    System.out.println();
    // Fija algunos elementos a valores diferentes
    for ( int i=0; i < a.size(); i++ )
      a.set( i,"47" );
    print( a );
    // Lo siguiente compila, pero no funciona
    lit.add( "X" );            // Operación no soportada
    a.clear();                 // No soportada
    a.add( "once" );           // No soportada
    a.addAll( a2 );            // No soportada
    a.retainAll( a2 );         // No soportada
    a.remove( s[0] );          // No soportada
    a.removeAll( a2 );         // No soportada
    }
  }
```

El lector podrá descubrir que solamente una parte de las interfaces de **Collection** y **List** están actualmente implementadas. El resto de los métodos provocan la aparición del desagradable mensaje la excepción **UnsupportedOperationException**. La causa es que la interfaz **Collection**, al igual que las otras interfaces en la nueva librería de colecciones, contienen métodos *opcionales*, que pueden estar o no *soportados* en la clase concreta que implemente esa interfaz. La llamada a un método no soportado hace que aparezca la excepción anterior para indicar el error de programación. Las líneas siguientes reproducen la ejecución del ejemplo, en donde se observa el mensaje que genera el intérprete Java a la hora de realizar la operación no soportada sobre la Colección.

```
% java Java812
uno dos tres cuatro cinco seis siete ocho nueve diez
a.contains(uno) = true
a.containsAll(a2) = true
a.isEmpty() = false
a.indexOf(seis) = 5
dieznueveochosieteseiscincocuatrotresdosuno
47 47 47 47 47 47 47 47 47 47
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.AbstractList.add(AbstractList.java:131)
        at java.util.AbstractList$ListItr.add(AbstractList.java:423)
        at Java812.main(Java812.java:67)
```

Lo anterior echa por tierra todas las promesas que se hacían con las interfaces, ya que se supone que todos los métodos definidos en interfaces y clases base tienen algún significado y están dotados de código en sus implementaciones; pero es más, en este caso no solamente no hacen nada, sino que detienen la ejecución del programa. Lo cierto es que no todo está tan mal; con **Collection**, **List**, **Set**, **Map** o **Queue**, el compilador todavía restringe la llamada a métodos, permitiendo solamente la llamada a métodos que se encuentren en la interfaz correspondiente, y además, muchos de los métodos que tienen una **Collection** como argumento solamente leen desde esa colección, y todos los métodos de *lectura* son *no* opcionales.

Esta posibilidad, o característica, evita una explosión de interfaces en el diseño. Otros diseños de librerías de colecciones siempre parecen tener al final una plétora de interfaces que describen cada una de las variaciones del tema principal, lo que hace que sean difíciles de aprender. Amén de que no es posible tratar todos los casos especiales en interfaces, ya que siempre hay alguien dispuesto a inventarse una nueva interfaz. La posibilidad de la *operación no soportada* consigue una meta importante con su incorporación en las nuevas colecciones: las hace muy fáciles de aprender y su uso es muy simple. Pero para que esto funcione, hay que tener en cuenta dos cuestiones:

1. La **UnsupportedOperationException** debe ser un evento muy raro. Es decir, en la mayoría de las clases todos los métodos deben funcionar, y solamente en casos muy especiales una operación podría no estar soportada. Esto es cierto en las

nuevas colecciones, ya que en el 99 por ciento de las veces, tanto **ArrayList**, **LinkedList**, **HashSet** o **HashMap**, así como otras implementaciones concretas, soportan todas las operaciones. El diseño proporciona una puerta falsa si se quiere crear una nueva **Collection** sin asignar significado a todos los métodos de la interfaz **Collection**, y todavía está en la librería.

2. Cuando una operación no esté soportada, sería muy adecuado que apareciese en tiempo de compilación la **UnsupportedOperationException** antes de que se genere en el programa que se vaya a entregar al cliente. Después de todo, una excepción indica un error de programación; es decir, hay una clase que se está utilizando de forma incorrecta.

En el ejemplo anterior, *Arrays.toList()* genera una Lista que está basada en un array de un tamaño fijo de elementos; por lo tanto, tiene sentido el que solamente estén soportadas aquellas operaciones que no alteran el tamaño del array. Si, de otro modo, fuese necesaria una interfaz nueva para tratar este distinto comportamiento, llamado **TamanoFijoList**, por ejemplo; se estaría dejando la puerta abierta al incremento de la complejidad de la librería, que se volvería insoportable cuando alguien ajeno intentase utilizarla al cabo del tiempo, por el montón de interfaces que tendría que aprenderse.

La documentación de todo método que tenga como argumento **Collection**, **List**, **Set**, **Map** o **Queue** debería especificar cuáles de los métodos opcionales deben ser implementados. Por ejemplo, la ordenación necesita los métodos *set()* e *Iterator.set()*, pero no necesita para nada los métodos *add()* y *remove()*.

ORDENACIÓN Y BÚSQUEDA

La plataforma Java 2 incorpora utilidades para realizar ordenaciones y búsquedas tanto en arrays como en listas. Estas utilidades son métodos *estáticos* de dos nuevas clases: **Arrays** para la ordenación y búsqueda en arrays y **Collections** para la ordenación y búsqueda en Listas.

Arrays

La clase **Arrays** tiene los métodos *sort()* y *binarySearch()* sobrecargados para todos los tipos básicos de datos, incluidos **String** y **Object**. El ejemplo `Java813.java` muestra la ordenación y búsqueda en un array de elementos de tipo `byte`, para todos los demás tipos básicos se haría de forma semejante, incluso para un array de **String**.

```
public class Java813 {
  static Random r = new Random();
  static String sorigen =
  "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
  "abcdefghijklmnopqrstuvwxyz";
  static char fuente[] = sorigen.toCharArray();
```

```
  // Se crea un String aleatorio a partir de la cadena que contiene
  // todas las letras
  public static String cadAleat( int longitud ) {
    int rnd;
    char buffer[] = new char[longitud];
    for( int i=0; i < longitud; i++ ) {
      rnd = Math.abs( r.nextInt() ) % fuente.length;
      buffer[i] = fuente[rnd];
      }
    return( new String( buffer ) );
    }

  // Crea un array aleatorio de Strings
  public static String[] cadsAleat( int longitud,int tamano ) {
    String s[] = new String[tamano];
    for( int i=0; i < tamano; i++ )
      s[i] = cadAleat( longitud );
    return( s );
    }

  public static void print( byte b[] ) {
    for( int i=0; i < b.length; i++ )
      System.out.print( b[i]+" " );
    System.out.println();
    }

  public static void print( String s[] ) {
    for( int i=0; i < s.length; i++ )
      System.out.print( s[i]+" " );
    System.out.println();
    }

  public static void main( String args[] ) {
    byte b[] = new byte[15];
    r.nextBytes( b ); // Se rellena el array de bytes
    print( b );
    Arrays.sort( b );
    print( b );
    int loc = Arrays.binarySearch( b,b[10] );
    System.out.println("Posicion de "+b[10]+" = "+loc );
    // Ahora ya se prueban la ordenación y la búsqueda
    String s[] = cadsAleat( 4,10 );
    print( s );
    Arrays.sort( s );
    print( s );
    loc = Arrays.binarySearch( s,s[4] );
    System.out.println( "Posicion de "+s[4]+" = "+loc );
    }
  }
```

La primera parte de la clase contiene utilidades para la generación de objetos aleatorios de tipo **String**, utilizando un array de caracteres desde el cual se pueden seleccionar las letras. El método *cadAleat()* devuelve una cadena de cualquier longitud, y *cadsAleat()* crea un array de cadenas aleatorias, dada la longitud de cada **String** y el número de elementos que va a tener el array. Los dos métodos *print()* simplifican la presentación de los datos. El método *Random.nextBytes()* rellena el

array argumento con bytes seleccionados aleatoriamente; en los otros tipos de datos básicos no hay un método semejante.

Una vez construido el array se puede observar que se hace una sola llamada al *sort()* o a *binarySearch()*. Hay una cuestión respecto a *binarySearch()*, y es que, si no se llama al método *sort()* antes de llamar a *binarySearch()* para realizar la búsqueda, se pueden producir resultados impredecibles, incluso la entrada en un bucle infinito.

La ordenación y búsqueda en el caso de **String** parece semejante, pero a la hora de ejecutar el programa se observa una cosa interesante: la ordenación es alfabética y lexicográfica, es decir, las letras mayúsculas preceden a las letras minúsculas. Así, todas las letras mayúsculas están al principio de la lista, seguidas de las letras minúsculas, luego 'Z' está antes que 'a'.

Comparable y Comparator

Si se quiere generar el índice de un libro, esa ordenación no es admisible, ya que lo lógico es que la 'a' vaya después de la 'A'.

Y, ¿qué pasa en un array de elementos de tipo **Object**? ¿Qué determina el orden en que se encuentran dos elementos de este tipo? Desgraciadamente, los diseñadores originales de Java no consideraron que éste fuese un problema importante, porque sino lo habrían incluido en la clase raíz **Object**. Como resultado, el orden en que se encuentran los elementos **Object** ha de ser impuesto desde fuera, por lo que la nueva librería de colecciones proporciona una forma estándar de hacerlo que por suerte es casi tan buena como si la hubiesen incluido en **Object**.

La tabla siguiente muestra la forma natural de ordenación, en donde algunas clases comparten el mismo orden. Solamente se pueden realizar ordenaciones entre clases que sean *mutuamente compatibles*.

Clase	Orden
BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short	Numérico
Carácter	Numérico por su valor *Unicode*
CollationKey	En función de la configuración local
Date	Cronológico
File	Numérico por el valor *Unicode* de los caracteres del nombre
ObjectStreamField	Numérico por el valor *Unicode* de los caracteres del nombre
String	Numérico por el valor *Unicode* de los caracteres de la cadena

Hay un método *sort()* para arrays de elementos **Object** (y **String**, desde luego, es un **Object**) que tiene un segundo argumento: un objeto que implementa la interfaz **Comparator**, que forma parte de la nueva librería, y realiza comparaciones a través de su único método *compare()*. Este método toma los dos objetos que van a compararse como argumentos y devuelve un entero negativo si el primer argumento es menor que el segundo, cero si son iguales y un entero positivo si el primer argumento es más grande que el segundo. Sabiendo esto, en el ejemplo `Java814.java`, se vuelve a implementar la parte correspondiente al array de elementos **String** del ejemplo anterior para que la ordenación sea alfabética.

```
public class Java814 implements Comparator<Object> {
  public int compare( Object obj1,Object obj2 ) {
    // Suponemos que solamente vamos a utilizar cadenas
    String s1 = ( (String)obj1).toLowerCase();
    String s2 = ( (String)obj2).toLowerCase();
    return( s1.compareTo( s2 ) );
    }

  public static void main( String args[] ) {
    String s[] = Java813.cadsAleat( 4,10 );
    Java813.print( s );
    Java814 ac = new Java814();
    Arrays.sort( s,ac );
    Java813.print( s );
    // Se debe utilizar un Comparador para realizar la búsqueda
    int loc = Arrays.binarySearch( s,s[3],ac );
    System.out.println( "Posicion de "+s[3]+" = "+loc );
    }
  }
```

Haciendo el moldeo a **String**, el método *compare()* implícitamente se asegura que solamente se están utilizando objetos de tipo **String**. Luego se fuerzan los dos **Strings** a minúsculas y el método *String.compareTo()* devuelve el resultado que se desea. Obsérvese también que es necesario indicar el tipo de datos que va comparar **Comparator**, en este caso se indica **Object** para ver el caso más genérico, pero se puede restringir a un tipo determinado de objetos.

A la hora de utilizar un **Comparator** propio para llamar a *sort()*, se debe utilizar el mismo **Comparator** cuando se vaya a llamar a *binarySearch()*. La clase **Arrays** tiene otro método *sort()* que toma un solo argumento: un array de **Object**, sin ningún **Comparator**. Este método también puede comparar dos **Object**, utilizando el *método natural de comparación* que es comunicado a la clase a través de la interfaz **Comparable**. Esta interfaz tiene un único método, *compareTo()*, que compara el objeto con su argumento y devuelve negativo, cero o positivo dependiendo de que el objeto sea menor, igual o mayor que el argumento. El ejemplo `Java815.java` es un programa muy sencillo que ilustra esta circunstancia.

Desde luego, el método *compareTo()* puede hacerse tan complicado como sea necesario. Sin embargo, *compareTo()* por defecto utiliza el orden lexicográfico, lo cual significa que los valores numéricos de los caracteres en el texto no corresponden

necesariamente al orden alfabético en todos los lenguajes. Para la ordenación en función de la configuración local se utiliza **Collator** con **CollationKey**. El ejemplo `Java816.java` muestra un ejemplo de este tipo de ordenación.

Listas

Una Lista puede ser ordenada y se puede buscar en ella del mismo modo que en un array. Los métodos estáticos para ordenar y buscar en una Lista están contenidos en la clase **Collections** y tienen un formato similar a los correspondientes de la clase **Arrays**: *sort(List)* para ordenar una Lista de objetos que implementen la interfaz **Comparable**, *binarySearch(List,Object)* para buscar un objeto en una lista, *sort(List,Comparator)* para ordenar una Lista utilizando un **Comparator** y *binarySearch(List,Object,Comparator)* para buscar un objeto en esa Lista. El uso de estos métodos es idéntico a los correspondientes de la clase **Arrays**, pero utilizando una Lista en lugar de un Array. El **TreeMap** también puede ordenar sus objetos en función de **Comparable** y **Comparator**.

Utilidades

Hay otras utilidades que pueden resultar interesantes en la clase **Collections**, como por ejemplo:

`enumeration(Collection)`, devuelve una **Enumeration** al viejo estilo a partir del argumento que se le pasa.

`max(Collection)`
`min(Collection)`, devuelven el elemento máximo o el mínimo de la colección que se le pasa como argumento, utilizando el método natural de comparación entre los objetos de la Colección.

`max(Collection,Comparator)`
`min(Collection,Comparator)`, devuelven el elemento máximo o el mínimo de la Colección que se le pasa como argumento, utilizando el **Comparator**.

`nCopies(int n,Object o)`, devuelve una Lista de tamaño `n` en la cual los *handles* apuntan a `o`.

`subList(List,int min,int max)`, devuelve una nueva Lista a partir de aquella que se le pasa como argumento que es una porción de esta última cuyos índices comienzan en el elemento `min` y terminan en el elemento `max`.

Tanto *min()* como *max()* trabajan con objetos de tipo **Collection**, no con **List**, por lo tanto es indiferente si la Colección está ordenada o no. Como ya se indicó antes, es necesario llamar a *sort()* en una Lista o Array antes de realizar una búsqueda con *binarySearch()*.

COLECCIONES O MAPAS DE SÓLO LECTURA

A menudo, es conveniente crear una versión de sólo lectura de una Colección o un Mapa. La clase **Collections** permite que se haga esto pasando el contenedor original a un método que devuelve una versión de sólo lectura. Hay cuatro variaciones sobre este método, una para cada tipo de **Collection** (en caso de no querer tratar la Colección de forma más específica), **List**, **Set** y **Map**. El ejemplo `Java817.java` muestra la forma en que se pueden construir versiones de sólo lectura de cada uno de ellos.

En cada caso, se debe rellenar el contenedor con datos significativos antes de hacerlo de sólo lectura. Una vez que está cargado, el mejor método es reemplazar el *handle* existente con el que genera la llamada al *no modificable*. De esta forma, no se asume el riesgo de cambiar accidentalmente el contenido. Por otro lado, esta herramienta permite que se pueda mantener un contenedor modificable como *privado* dentro de la clase y devolver un *handle* de sólo lectura hacia él a través de un método. Así se puede realizar cualquier cambio desde la propia clase, pero el resto del mundo solamente puede leer los datos.

La llamada al método *no modificable* para un tipo determinado no genera ninguna comprobación en tiempo de compilación, pero una vez que se haya realizado la transformación, la llamada a cualquier método que intente modificar el contenido de cualquiera de los elementos del contenedor obtendrá por respuesta una excepción de tipo **UnsupportedOperationException**.

Si en el ejemplo `Java817.java` se descomenta la línea que añade un elemento a la primera de las Colecciones

```
c.add( "uno" );                    // No se puede cambiar
```

y se vuelve a compilar el código fuente resultante, a la hora de ejecutar el programa se obtendrá un resultado semejante al que reproducen las líneas siguientes

```
% java Java817
0 1 2 3 4 5 6 7 8 9
Exception in thread "main" java.lang.UnsupportedOperationException
        at java.util.Collections$UnmodifiableCollection.add(
          Collections.java:1018)
        at Java817.main(Java817.java:57)
```

COLECCIONES O MAPAS SINCRONIZADOS

La palabra reservada `synchronized` forma una parte muy importante de la programación multitarea, o *multithread*, que se tratará abundantemente en otro capítulo; aquí, en lo que respecta a las colecciones baste decir que la clase **Collections** contiene una forma de sincronizar automáticamente un contenedor entero, utilizando una sintaxis parecida a la del caso anterior de hacer un contenedor *no modificable*. La

clase siguiente es una muestra de su uso, en el que se crean todas las colecciones para contener elementos de tipo **String** a modo de ilustración.

```
public class CreaColecciones {
  public static void main( String args[] ) {
    Collection<String> c = Collections.synchronizedCollection(
      new ArrayList<String>() );
    List<String> list = Collections.synchronizedList(
      new ArrayList<String>() );
    Set<String> s = Collections.synchronizedSet(
      new HashSet<String>() );
    Map<String,String> m = Collections.synchronizedMap(
      new HashMap<String,String>() );
    }
  }
```

En esta clase, el nuevo contenedor se pasa inmediatamente a través del método adecuado de sincronización, para evitar que se produzca algún destrozo por estar expuesto a acciones concurrentes.

Las nuevas Colecciones también tienen un mecanismo de prevenir que más de un proceso esté modificando de forma simultánea los objetos que contiene un contenedor. El problema se presenta en el movimiento de elementos a través del contenedor llevado a cabo por un proceso, mientras otro proceso se encuentra realizando acciones de inserción, borrado o cambio de alguno de los objetos que está almacenado en el contenedor. Puede que ya se haya pasado sobre el objeto en cuestión, o puede que todavía no se haya alcanzado, o puede que el tamaño del contenedor varíe inmediatamente después que se haya llamado a *size()*, en fin, que hay mil y un casos en que se puede producir una situación que desemboque en un completo desastre. La nueva librería de colecciones incorpora un mecanismo que llaman *fail fast*, que está monitorizando continuamente los cambios que se producen en el contenedor por parte de otros procesos, de tal modo que si detecta que se está modificando el contenedor, inmediatamente genera una excepción de tipo **ConcurrentModificationException**.

Colecciones o Mapas Singleton

La clase **Collections** permite crear objetos **Set** de un único elemento en un solo paso, evitando tener que crear un objeto **Set** y luego rellenarlo. El objeto **Set** resultante es inmutable, persistente.

```
Set<String> set = Collections.singleton( "Tutorial de Java" );
```

También se pueden crear de la misma forma otros objetos del tipo de las Colecciones Java, como Listas y Mapas, de tipo también *Singleton*.

```
List<E> singletonList( Object elemento )
Map<E,E> singletonMap( Object clave,Object valor )
```

CAPÍTULO 9

FICHEROS EN JAVA

En otros capítulos se muestra cómo recuperar e introducir cadenas de texto desde el teclado en una interfaz de usuario; pero hay que reconocer que lo interesante es hacerlo en ficheros y bases de datos, que permiten un almacenamiento permanente de la información. No obstante, la aproximación que se va a realizar de los ficheros no va a ser tan exhaustiva como la realizada con otros conceptos de Java, ya que se supone que el lector, a estas alturas, conoce las bases del funcionamiento de los sistemas de ficheros, así que se van a presentar ejemplos de acceso a ficheros y también algunos conceptos específicos de Java, como la forma de permitir que Java acceda a ciertos ficheros a través de applets o servlets y cómo restringir a una aplicación el acceso a ficheros. Todas estas operaciones, presentes desde la creación de Java, se han visto mejoradas, e incluso reescritas, mediante el uso más optimizado de tareas en la versión actual de la plataforma Java 2. Incluso algunas de las operaciones que los programadores echaban en falta, se han incorporado en la arquitectura de entrada/salida de Java, **NIO**.

LA CLASE FILE

Antes de realizar acciones sobre un fichero, es necesario introducir la clase **File** que proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre los mismos.

Creación de un objeto File

Para crear un objeto **File** nuevo, se puede utilizar cualquiera de los constructores que se enumeran:

```
File miFichero;
miFichero = new File( "/home/kk" );
    o
miFichero = new File( "/home","kk" );
    o
File miDirectorio = new File( "/home" );
miFichero = new File( miDirectorio,"kk" );
```

El constructor utilizado depende a menudo de otros objetos **File** necesarios para el acceso. Por ejemplo, si sólo se utiliza un fichero en la aplicación, el primer constructor es el mejor. Si, en cambio, se utilizan muchos ficheros desde un mismo directorio, el segundo o tercer constructor será más cómodo. En caso de que el directorio o fichero sean variables, el tercer constructor será el más indicado de los tres.

Comprobaciones y utilidades

Una vez creado un objeto **File**, ya se puede reunir información sobre el fichero, la plataforma Java 2 proporciona muchos métodos: para obtener el nombre del fichero, para comprobar los permisos de lectura y escritura, para obtener las últimas modificaciones, para recuperar información general sobre el fichero, etc.

El ejemplo `Java901.java` es un programa muy simple que muestra información sobre los ficheros pasados como argumentos en la línea de comandos.

```
class Java901 {
  public static void main( String args[] ) throws IOException {
    // Se comprueba que nos han indicado algún fichero
    if( args.length > 0 ) {
      // Vamos comprobando cada uno de los ficheros que se hayan
      // pasado en la línea de comandos
      for( int i=0; i < args.length; i++ ) {
        // Se crea un objeto File para tener una referencia al
        // fichero físico del disco
        File f = new File( args[i] );
        // Se presenta el nombre y directorio donde se encuentra
        System.out.println( "Nombre: "+f.getName() );
        System.out.println( "Camino: "+f.getPath() );
        // Si el fichero existe se presentan los permisos de lectura
        // y escritura y su longitud en bytes
        if( f.exists() ) {
          System.out.print( "Fichero existente" );
          System.out.print( (f.canRead() ?
            " y se puede Leer" : "" ) );
          System.out.print( (f.canWrite() ?
            " y se puede Escribir" : "" ) );
          System.out.println( "." );
          System.out.println( "La longitud del fichero es de "+
            f.length()+" bytes" );
          }
        else
          System.out.println( "El fichero no existe." );
        }
      }
```

```
    else
      System.out.println( "Debe indicar un fichero." );
    }
  }
```

Permisos y control de acceso

Java proporciona varios métodos en la clase **File** que permiten modificar los permisos de acceso a ficheros. Si el lector es habitual de sistemas Unix, estos métodos le resultarán familiares, porque son semejantes a la funcionalidad que proporciona el comando *chmod* de Unix.

Los permisos de acceso a ficheros son los que corresponden a la lectura, escritura y ejecución. El sistema de ficheros puede asignar cualquiera de estos permisos a un único objeto, y así mismo, un conjunto de estos permisos al propietario del fichero o a otros usuarios. Los métodos que proporciona Java corresponden a cada uno de los tres permisos sobre los ficheros y tienen dos signaturas: la primera que recibe un único parámetro de tipo booleano para indicar si ese permiso se asigna o no y, la segunda que recibe dos parámetros, el primero igual que en el caso anterior y además, un segundo parámetro para indicar si el permiso afecta solamente al propietario del fichero, si se pasa a `true`, o a todos los usuarios si se fija como `false`. En el caso de sistemas de ficheros que no distinguen entre permisos de propietario y genéricos, como es el caso de los sistemas de ficheros de Windows, este parámetro no tiene efecto y los permisos que se fijen afectarán a todos los usuarios.

Los métodos para asignar permisos, en sus dos formas, son los siguientes:

public boolean setExecutable(boolean ejecutar);
public boolean setExecutable(boolean ejecutar, boolean propietario);
public boolean setReadable(boolean leer);
public boolean setReadable(boolean leer, boolean propietario);
public boolean setWritable(boolean escribir);
public boolean setWrittable(boolean escribir, boolean propietario);

La invocación de los métodos anteriores devuelve `false` si el usuario que los ejecuta no tiene autorización para cambiar los permisos de acceso al fichero. Cada uno de los métodos puede generar una excepción de tipo **SecurityException** en el caso de que se haya definido un controlador de seguridad, en el cual los métodos *checkExecute()*, *checkRead()* y *checkWrite()*, controlan el acceso al fichero.

La clase **File** también proporciona los métodos *canExecute()*, *canRead()* y *canWrite()* para poder comprobar los permisos asignados a un fichero y poder conocer las acciones que se pueden realizar sobre él.

En el ejemplo `Java910.java` se muestra el uso de los métodos *setWritable()* y *canWrite()* para modificar y consultar el estado del permiso de escritura, respectivamente, sobre el fichero que se pase como argumento en la ejecución.

APLICACIONES

La plataforma Java 2 proporciona una gran cantidad de clases para leer tanto caracteres como bytes en un programa y, por supuesto, para guardarlos en un fichero externo, dispositivo de almacenamiento o programa. La fuente o el destino pueden estar situados en el ordenador local en que se ejecuta la aplicación o en cualquier otro lugar de la red. En los ejemplos que siguen, se supone que la lectura de datos y la escritura en fichero se realizan sobre ficheros situados en el disco local del ordenador.

Para *leer*, un programa abre un *stream de entrada* sobre el fichero y lee los datos en el mismo orden en que fueron escritos en el fichero. Un *stream* es un canal que solamente permite el flujo de datos en una dirección, es decir, si en este caso se abre un stream de entrada, o *canal de entrada*, solamente se podrán leer datos a través de él, no permite operaciones de escritura.

Para *escribir*, un programa abre un *stream de salida* hacia el fichero y escribe los datos secuencialmente. Este *stream* de salida, o *canal de salida*, también es de dirección única, por lo que no permite operaciones de lectura a través de él.

El ejemplo `Java902.java` presenta en pantalla una ventana que contiene un campo de texto y un botón. Si se introduce una cadena de texto en el campo y se pulsa el botón, esa cadena se grabará en un fichero. Si se vuelve a pulsar el botón, se recuperará el contenido de otro fichero de texto que se presentará en pantalla. Pulsando de nuevo el botón se puede reproducir el proceso.

La figura 9.1 reproduce los dos estados en que se puede encontrar la aplicación, a la izquierda la ventana lista para grabar texto, y a la derecha la ventana una vez ha recuperado el texto del fichero.



Figura 9.1

Las partes más interesantes del código del ejemplo es lo que se trata a continuación. Lo primero es la declaración de las variables globales correspondientes a los elementos de la ventana, de los cuales el realmente importante es el campo de texto que va a permitir introducir la cadena que se va a grabar en el fichero y cuyo comportamiento se tratará con profundidad en posteriores secciones.

```
JTextField campoTexto;
```

El constructor se limita a colocar los distintos componentes *Swing* sobre el panel, así que no merece la pena detenerse en su descripción, porque en capítulos posteriores de tratarán en profundidad. Pero sí resulta interesante el método *actionPerformed()*, que es el encargado de controlar las pulsaciones del ratón. En este método se utilizan objetos de tipo **FileInputStream** y **FileOutputStream** para leer y escribir datos a fichero; estas clases manejan los datos como bytes en vez de como caracteres.

Las dos líneas siguientes muestran la declaración de la variable que obtiene la cadena de texto introducida por el usuario en la ventana y el array de bytes en la que se almacena.

```
String texto = campoTexto.getText();
byte b[] = texto.getBytes();
```

A continuación, se crea un objeto **File** para el fichero que se va a grabar y que será el que se utilice para crear el objeto **FileOutputStream** a través del cual se enviarán los datos al fichero.

```
File fichSalida = new File( dir +
  File.separatorChar+ "textoe.txt" );
FileOutputStream canalSalida = new FileOutputStream( fichSalida );
```

Para concluir la grabación de la cadena en el dispositivo de almacenamiento, se escribe el array de bytes a través del canal de salida y se cierra la conexión con el fichero; tal como reproducen las siguientes dos líneas de código.

```
canalSalida.write( b );
canalSalida.close();
```

El código encargado de la lectura del fichero de texto es similar. Primero se crea un objeto de tipo **File** que es utilizado para crear un canal de entrada desde él. El código siguiente reproduce esa situación en el ejemplo.

```
File fichEntrada = new File( dir +
  File.separatorChar+ "textol.txt" );
FileInputStream canalEntrada = new FileInputStream( fichEntrada );
```

Luego hay que crear un array de bytes del mismo tamaño que el fichero sobre el cual se va a leer el contenido de ese fichero.

```
byte bt[] = new byte[(int)fichEntrada.length()];
int numBytes = canalEntrada.read( bt );
```

Y ya, el array de bytes es el origen de un objeto **String** que es el que se utiliza para crear la etiqueta que se presenta en la ventana. El canal de comunicación con el fichero se cierra una vez que se ha pasado el contenido del array a la etiqueta.

```
String cadena = new String( bt );
etiqueta.setText( cadena );
```

```
canalEntrada.close();
```

El código comentado que aparece en el ejemplo, y que se reproduce a continuación, merece una revisión un poco más detallada.

```
  String dir = System.getProperty( "user.home" );
```

Corresponde al directorio raíz del usuario, que siempre es dependiente del Sistema Operativo en el cual se ejecuta la aplicación, pero que representa un lugar definido para cualquier sistema.

```
File fichEntrada = new File( dir +File.separatorChar+ "textol.txt" );
```

En esa línea de código se utiliza la variable `separatorChar` de la clase **File** a la hora de construir el nombre del directorio donde se encuentra el fichero que se va a recuperar. Esta variable es inicializada con el contendido del separador que se haya indicado en el fichero de propiedades del sistema como `file.separator`, proporcionando de este modo una forma muy cómoda de construir aplicaciones independientes de plataforma. En el caso concreto de que estuvieran directorio y fichero en *Unix* sería:

```
/tmp/tutorial/texto.txt
```

mientras que en entornos *Windows* sería:

```
\tmp\tutorial\texto.txt
```

y los dos representados en el formato independiente de plataforma que se ha indicado antes se indicarían de la forma:

```
File.separatorChar+ "tmp" + File.separatorChar+ "tutorial"
  +File.separatorChar+ "texto.txt";
```

OPERACIONES

En toda aplicación, las operaciones más costosas en tiempo de ejecución son las de entrada/salida, las que implican tareas de lectura y escritura en ficheros.

La peor solución será la lectura, o escritura, carácter a carácter, tal como se muestra en el ejemplo `Java903.java`, en el cual se ha añadido un contador para cuantificar el tiempo empleado en realizar la acción, en este caso, de lectura. El resultado de ejecutar el programa sobre un fichero de prueba de 7 kbytes arroja un tiempo de 40 msgs en el ordenador del autor.

```
public class Java903 {
  public static void main( String args[] ) {
    long intervalo = System.currentTimeMillis();
    FileReader fReader = null;
    int c;
```

```
    try {
      fReader = new FileReader("prueba.txt");
      while( (c = fReader.read()) != -1 ) {
        }
    } catch( Exception e ) {
      e.printStackTrace();
    } finally {
      try {
        if( fReader != null )
          fReader.close();
      } catch( Exception e ) {
        e.printStackTrace();
        }
      }
    System.out.println( "Tiempo: "+
      (System.currentTimeMillis()-intervalo)+ " msgs." );
    }
  }
```

Java proporciona herramientas de control de los canales de entrada y salida que permiten reducir de forma drástica la carga que suponen este tipo de operaciones. La más importante de todas ellas es la posibilidad de incorporar un buffer de lectura, o escritura. El ejemplo `Java904.java` es el mismo ejemplo anterior, modificado para utilizar un buffer de lectura. Su código es el que se reproduce.

```
public class Java904 {
  public static void main( String args[] ) {
    long intervalo = System.currentTimeMillis();
    BufferedReader bReader = null;
    int c;
    try {
      bReader = new BufferedReader( new FileReader("prueba.txt") );
      while( (c = bReader.read()) != -1 ) {
        }
    } catch( Exception e ) {
      e.printStackTrace();
    } finally {
      try {
        if( bReader != null )
          bReader.close();
      } catch( Exception e ) {
        e.printStackTrace();
        }
      }
    System.out.println( "Tiempo: "+
      (System.currentTimeMillis()-intervalo)+ " msgs." );
    }
  }
```

La clase **BufferedReader** es semejante en funcionalidad a la clase **FileReader**, pero utiliza un buffer de lectura interno de forma que no lee el fichero carácter a carácter, sino en bloques de caracteres que almacena en ese buffer interno, transparente para el programador; posteriormente, cada carácter es leído del buffer en lugar del fichero. Si el lector ejecuta la aplicación, comprobará que el tiempo de

lectura del fichero de prueba anterior sufre una drástica reducción, siendo de unos 16 milisegundos, un tercio más o menos.

En general, se puede afirmar que toda operación que implique lectura y escritura de ficheros es mucho más eficaz cuando se utilizan bloques de datos, y se emplean las clases que el API de Java proporciona para encargarse de esa labor.

EXCEPCIONES

Como el lector comprobará en el capítulo siguiente, una *excepción* es una clase que desciende de la clase padre **Exception** o **RuntimeException**, y que define condiciones de error de tipo moderado que se puede encontrar la aplicación durante su ejecución.

En el caso de tratamiento de ficheros, es imprescindible capturar las excepciones que puede lanzar el sistema. Por ello, en el ejemplo anterior, el código que involucraba a las operaciones de entrada y salida a fichero en el método *actionPerformed()*, estaban encerradas en un bloque `try-catch`, que sería en este caso el encargado de capturar las excepciones de tipo **IOException** que pudiesen generarse durante la manipulación de los ficheros, tanto en grabación como en lectura.

La excepción **IOException** se considera una excepción de comprobación. La plataforma Java requiere que todas las excepciones de este tipo sean capturadas, generando un error en tiempo de compilación si no se hace así. También este tipo de excepciones se pueden relanzar, pero ha de ser dentro del ámbito del método en que se producen; es decir, si el método no captura la excepción en un bloque `try-catch`, debe especificar que puede lanzarla, debe hacer que pase a formar parte de su interfaz pública, porque los métodos que lo llamen deben conocer las excepciones que han de capturar para tomar las medidas oportunas. Este tipo de excepciones descienden de la clase **Throwable**.

Sin embargo, a pesar de lo dicho anteriormente, puede ocurrir que no se pueda cambiar la interfaz del método; por ejemplo, el método *actionPerformed()* del ejemplo anterior ya tiene una definición de interfaz pública que no puede alterarse para indicar que es capaz de lanzar excepciones de tipo **IOException**, por lo que en este caso, la única solución es capturar y tratar en el propio método la excepción.

Los métodos que defina el programador pueden especificar excepciones o capturarlas y tratarlas; mientras que los métodos que se sobrescriban deben, inexcusablemente, capturar y tratar las excepciones de comprobación. El código siguiente muestra un ejemplo de un método definido por el programador que especifica una excepción a los otros métodos que lo llamen, para que sean ellos los que capturen ese tipo de excepción y obren en consecuencia.

```
public int MiMetodo( int _var1,int _var2 )
  throws IllegalValueException {
  // ...Cuerpo del método
  }
```

APPLETS

En el ejemplo `Java905.java`, el lector podrá comprobar que el código de acceso a fichero es semejante al visto en el ejemplo `Java902.java`, pero muestra el uso de canales de tipo carácter en lugar de canales de bytes. El que se use un tipo en aplicaciones y otro en applets es puramente ilustrativo, se pueden utilizar cualquier tipo de canales tanto en aplicaciones como en applets, la decisión de uso de uno u otro debe estar basada en los requisitos específicos del programa a realizar.

En el código del ejemplo, el lector podrá observar dos cambios respecto al ejemplo anterior. A la hora de *escribir* la cadena en el fichero, el contenido del campo de texto es pasado directamente al método que lo escribe, *write()*. Y a la hora de *leer*, se crea un array de caracteres para almacenar los datos que se lean del fichero.

Para poder almacenar datos en *streams*, el paquete **java.applet** proporciona tres métodos a través de la clase **AppletContext**. Estos métodos permiten el almacenamiento de datos en *streams* donde cada *stream* es asignado a un identificador y con un valor determinado para ese identificador, siguiendo la estructura de las colecciones **Map**. Este mapeo es específico del código del applet, es decir, que un applet que provenga de un *host* determinado no podrá acceder a los *streams* de otro *host*, para no violar las normas de seguridad impuestas a los applets.

El método principal para almacenar información en un stream es *setStream()*. Luego se puede utilizar el método *getStream()* para recuperar un stream y el método *getStreamKeys()* para recuperar todos los streams, utilizando un iterador para recuperar la colección de streams, de la siguiente forma:

```
Iterator iter = getAppletContext().getStreamKeys();
if( iter != null ) {
  while( iter.hasNext() ) {
    String nombre = iter.next();
    InputStream is = getAppletContext().getStream( nombre );
    // código de lectura del contenido del stream
    }
  }
```

SEGURIDAD

Probablemente el lector esté cansado de ejecutar applets y obtener mensajes de error a la hora de pulsar un botón, de tipo de falta de permiso para ejecutar las acciones relacionadas con ficheros. Estos errores se deben a que el control de seguridad de la plataforma Java 2 no permite que un applet o un servlet puedan escribir en un fichero o leer datos de él sin disponer de permiso explícito para ello.

Un applet no tiene acceso a ningún recurso del ordenador local, a no ser que expresamente se le proporcione el permiso requerido, luego para que el programa pueda escribir en el fichero *textoe.txt* y leer del fichero *textol.txt*, hay que proporcionarle los adecuados permisos de escritura y lectura para cada fichero.

Los permisos se conceden a través de un fichero de normas o pólizas o políticas (mala traducción de *policy file*) de seguridad, que se debe pasar a *appletviewer* para que el applet lo conozca.

Creación de un fichero de políticas

La plataforma Java 2 proporciona varias herramientas de seguridad, entre ellas está la herramienta *PolicyTool*, cuya utilización se explicará en un capítulo posterior, mediante la cual se pueden crear los ficheros de políticas o *normas* de seguridad. En este caso concreto, para crear el fichero de políticas se puede utilizar la herramienta *PolicyTool* o copiar directamente el texto que sigue en un fichero ASCII.

```
grant {
  permission java.util.PropertyPermission "user.home","read";
  permission java.io.FilePermission "${user.home}/textoe.txt","write";
  permission java.io.FilePermission "${user.home}/textol.txt","read";
  };
```

Ejecución con ficheros de políticas

Suponiendo que el lector ha llamado al fichero de normas de seguridad, *java.politicas*, y que se encuentra en el mismo directorio que el fichero HTML que permite lanzar el applet, cuyo código podría ser el que se reproduce a continuación:

```
<HTML>
<BODY>
<APPLET CODE="Java905.class" WIDTH=200 HEIGHT=75>
</APPLET>
</BODY>
</HTML>
```

el comando necesario para lanzar el *appletviewer* utilizando el fichero de normas, pólizas o políticas de seguridad que se ha creado para permitir el acceso a ficheros, sería el siguiente:

```
appletviewer -J-Djava.security.policy=java.politicas java905.html
```

Si el lector no utiliza *appletviewer* y dispone de un navegador con soporte para la plataforma Java 2, o si ha instalado el *Java Plug-In*, podrá ejecutar el applet desde del propio navegador si coloca el fichero de políticas de seguridad en su directorio raíz local y le asigna el nombre *.java.policy*, o puede firmar el applet como se indica en una sección posterior.

Restricción de accesos a aplicaciones

Se puede utilizar el controlador de seguridad por defecto y un fichero de políticas de seguridad para restringir los accesos de las aplicaciones, a través del comando siguiente (en una sola línea):

```
java -Djava.security.manager
  -Djava.security.policy=java.politicas Java902
```

Como las aplicaciones se ejecutan sin el controlador de seguridad, el cual deshabilita todo tipo de acceso, el fichero de políticas necesita dos permisos adicionales: uno para el controlador de seguridad, para que se pueda acceder a la cola de eventos y cargar los componentes de la interfaz de usuario; y el otro para la aplicación, para que no presente el mensaje de aviso de que la ventana ha sido creada por otro programa (el controlador de seguridad). El fichero de políticas de seguridad quedaría, por tanto, como sigue

```
grant {
  permission java.awt.AWTPermission "accessEventQueue";
  permission java.awt.AWTPermission "showWindowWithoutWarningBanner";
  permission java.util.PropertyPermission "user.home","read";
  permission java.io.FilePermission "${user.home}/textoe.txt","write";
  permission java.io.FilePermission "${user.home}/textol.txt","read";
  };
```

De igual modo que en el caso anterior, este fichero puede crearse con la herramienta *PolicyTool*, o bien copiando directamente las líneas anteriores en un fichero ASCII.

Firma de applets

Para la ejecución del applet en un navegador, de forma que se pueda invocar desde cualquier plataforma, es necesario el uso del *Java Plug-In* de Java 2, que es el que proporciona la independencia necesaria. Para que el navegador, a través del Plug-In pueda realizar o asignar diferentes niveles de seguridad a un applet, es necesario que éste sea *firmado*. A continuación, se indican los pasos a seguir para el firmado del applet que se implementa en el archivo fuente `Java905.java`, de forma que se pueda cargar la página `java905s.html` en un navegador y sea posible salvar en el propio disco del usuario la cadena que se introduzca en el campo de texto, operación normalmente vedada a un applet normal.

En las siguientes indicaciones se utilizan las herramientas estándar que proporciona *Sun Microsystems* en la distribución habitual de la plataforma Java 2, sin embargo, es posible utilizar herramientas proporcionadas por los propios creadores de los navegadores para conseguir el mismo fin. No obstante, a la hora de utilizar certificados de validez de firmas para la distribución de applets, será necesario el concurso de una empresa de verificación, como *Verisign*, *Thawte*, etc. Para efectos de

desarrollo y prueba, *Sun* proporciona su propia herramienta que hace que no sea necesario el concurso de terceros para comprobar la validez de un certificado.

La compilación del applet se realiza de la forma habitual, el proceso de firmado es posterior y no tiene nada que ver con el funcionamiento interno del applet.

Una vez obtenido el fichero *.class* correspondiente a la compilación del código del applet, es necesario generar una clave, un fichero de certificado que contenga una clave que permita la verificación de quién es el autor del applet. Para ello se utiliza la herramienta **KeyTool**. Desde el directorio en donde se encuentra el fichero *.class*, se invoca la herramienta de la forma que se indica a continuación, respondiendo a todas las preguntas que van apareciendo en la pantalla:

```
-> keytool -genkey -keyalg rsa -alias clave
Escriba la contraseña del almacén de claves:  cualquiera
¿Cuáles son su nombre y su apellido?
  [Unknown]:  Agustin Froufe
¿Cuál es el nombre de su unidad de organización?
  [Unknown]:  Documentacion
¿Cuál es el nombre de su organización?
  [Unknown]:  Tutorial de Java
¿Cuál es el nombre de su ciudad o localidad?
  [Unknown]:  Las Palmas de Gran Canaria
¿Cuál es el nombre de su estado o provincia?
  [Unknown]:  Islas Canarias
¿Cuál es el código de país de dos letras de la unidad?
  [Unknown]:  ES
¿Es correcto CN=Agustin Froufe, OU=Documentacion, O=Tutorial de Java,
L=Las Palmas de Gran Canaria, ST=Islas Canarias, C=ES?
  [no]:  si

(esperar un ratito)

Escriba la contraseña clave para <clave>
      (INTRO si es la misma contraseña que la del almacén de claves):
```

La información que se introduce es como ejemplo, el lector debe introducir la correspondiente a sus propias circunstancias. A continuación, se genera el fichero de certificado, utilizando la misma herramienta, invocándola de la forma siguiente:

```
-> keytool -export -alias clave -file clave.crt
Escriba la contraseña del almacén de claves:  cualquiera
Certificado almacenado en el archivo <clave.crt>
```

Así se genera el fichero `clave.crt` que contiene el certificado con el cual será posible firmar el applet.

No es posible firmar archivos *.class*, solamente es posible hacerlo a través de archivos JAR, que contienen información acerca del certificado y autor del applet, además de los archivos que forman parte del applet. En este caso, para crear el archivo JAR correspondiente al ejemplo, invocamos a la herramienta **jar** del siguiente modo:

```
-> jar cvf Java905.jar Java905.class
manifest agregado
agregando: Java905.class (entrada = 3153) (salida = 1784)
  (desinflado 43%)
```

Es posible verificar el contenido del archivo JAR. El lector comprobará que la herramienta ha incorporado archivos extra. Se verifica un archivo JAR invocando de nuevo a **jar** de la forma:

```
-> jar tvf Java905.jar
    0 Sun Apr 06 11:36:30 BST 2008 META-INF/
   71 Sun Apr 06 11:36:30 BST 2008 META-INF/MANIFEST.MF
 3244 Sun Apr 06 11:14:18 BST 2008 Java905.class
```

Ahora sí es posible *firmar* el archivo JAR, que es el que contiene el código del applet. Para hacerlo se utiliza la herramienta **JarSigner**, invocándola de la forma:

```
-> jarsigner Java905.jar clave
Enter Passphrase for keystore: cualquiera

Warning: This jar contains entries whose signer certificate will
  expire within six months.
```

La última frase es el aviso de la caducidad del certificado, porque por defecto, la herramienta *keytool* genera certificados válidos durante seis meses.

Por supuesto, también es posible la verificación de la firma, ejecutando de nuevo *jarsigner* de la forma:

```
-> jarsigner -verify -verbose -certs Java905.jar
         137 Sun Apr 06 11:42:30 BST 2008 META-INF/MANIFEST.MF
         258 Sun Apr 06 11:42:30 BST 2008 META-INF/CLAVE.SF
        1070 Sun Apr 06 11:42:30 BST 2008 META-INF/CLAVE.RSA
           0 Sun Apr 06 11:41:10 BST 2008 META-INF/
smk     3244 Sun Apr 06 11:14:18 BST 2008 Java905.class

      X.509, CN=Agustin Froufe, OU=Documentacion, O=Tutorial de Java,
      L=Las Palmas de Gran Canaria, ST=Islas Canarias, C=ES (clave)
      [certificate will expire on 5/07/08 11:40]
  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope
jar verified.

Warning: This jar contains entries whose signer certificate will
  expire within six months.
```

El siguiente paso consiste en la modificación de la página Web, o archivo HTML que realiza la invocación del applet, el que contiene la etiqueta <APPLET>. En este archivo es necesario indicar el parámetro ARCHIVE, al cual se le ha de asignar como valor el nombre del fichero JAR que se ha creado. La página usada anteriormente, modificada de este modo se muestra a continuación:

```
<HTML><BODY>
<APPLET CODE="Java905" ARCHIVE="Java905.jar" WIDTH=200 HEIGHT=75>
</APPLET>
<BODY></HTML>
```

Como se ha indicado anteriormente, para que el applet pueda utilizarse en cualquier navegador a través de este archivo HTML es necesario obtener la verificación del certificado por un tercero, por una autoridad de certificación. Las opciones son dos: recurrir a una empresa de pueda emitir esa verificación o utilizar el almacén `cacerts` que *Sun* proporciona para facilitar el desarrollo de aplicaciones.

Para solicitar la verificación de un certificado a una empresa especializada, es necesario realizar una petición de validación exportando el certificado generado. Para ello se utiliza la herramienta **KeyTool** de la siguiente forma:

```
-> keytool -certreg -alias clave -file clave.crt
```

Y una vez obtenida la respuesta de la autoridad de certificación, es necesario importar el nuevo certificado en el almacén de claves, sobrescribiendo el certificado original que se había generado. Para ello también se utiliza **KeyTool**, pero con el comando `import`, de la siguiente forma:

```
-> keytool -import -trustcacerts -alias clave -file respuesta.crt
```

Cualquier applet firmado mediante un certificado verificado por una autoridad de certificación, será automáticamente reconocido por el *Java Plug-In*.

Sin embargo, este proceso es lento en tiempo y, en ocasiones gravoso, porque todas las entidades de certificación suelen requerir pago por la utilización de sus servicios y por la emisión de certificados reales, aunque también suelen emitir certificados de prueba de forma gratuita. Por ello, para propósitos de desarrollo es posible generar certificados temporales, de forma que el Plug-In los reconozca. El Java Plug-In reconocerá todos los certificados que estén localizados a través del almacén `cacerts`. Esto significa que es posible importar el certificado de prueba que se había generado anteriormente en este almacén y hacer que el Plug-In reconozca el applet que se había incorporado al archivo `Java905.jar`.

Para importar el certificado es necesario copiar el archivo que lo contiene, en este caso `clave.crt`, en el directorio de seguridad en donde Java 2 almacena las claves, que es:

```
JAVA_HOME/jre/lib/security
```

En donde `JAVA_HOME` es el directorio de instalación del JDK. Se copia el fichero y se incorpora al almacén de claves `cacerts`, de la siguiente forma:

```
-> keytool -import -keystore cacerts -storepass changeit -file clave.crt
Propietario: CN=Agustin Froufe, OU=Documentacion, O=Tutorial de Java,
  L=Las Palmas de Gran Canaria, ST=Islas Canarias, C=ES
```

```
Emisor: CN=Agustin Froufe, OU=Documentacion, O=Tutorial de Java,
  L=Las Palmas de Gran Canaria, ST=Islas Canarias, C=ES
Número de serie: 47f8a8a4
Válido desde: Sun Apr 06 11:40:36 2008 hasta: Sat Jul 05 11:40:36 2008
Huellas digitales del certificado:
     MD5:  B6:A4:80:B5:1D:64:3E:C2:78:58:7C:2F:29:3E:A0:0F
     SHA1: 9A:B0:A0:09:80:D2:96:CF:10:62:64:50:3A:BE:3C:C7:C8:B2:11:0C
     Nombre del algoritmo de firma: SHA1withRSA
     Versión: 3
¿Confiar en este certificado? [no]:  si
Se ha añadido el certificado al almacén de claves
```

En este momento, independientemente del método utilizado, el applet ya debería ser reconocido a través del archivo JAR firmado. Si el lector utiliza archivos JAR adicionales en el funcionamiento del applet, por ejemplo el archivo JAR correspondiente a un analizador XML también ha de firmarse, porque es necesario que se firmen todos aquellos archivos JAR que contengan clases que se ejecuten en el navegador cliente, ya que de otro modo, solamente aquellas clases que estén contenidas en el archivo firmadas trabajarán con el criterio de seguridad fijado a `java.security.AllPermission`.

La figura 9.2 muestra la ventana de confirmación de aceptación del certificado anterior cuando se carga el archivo HTML en un navegador.



Figura 9.2

Si no se utiliza el atributo `-keystore` con las herramientas **KeyTool** y **JarSigner**, el fichero de almacén que se utiliza es `.keystore`, localizado en el directorio raíz del usuario. Este fichero se crea la primera vez que se accede a un almacén de claves y se protege con la primera contraseña que se haya proporcionado. Si el lector olvida la contraseña, no hay forma de recuperar el contenido, aunque sí es posible eliminar el

fichero y volver a crearlo. Si se pretende un cierto grado de seguridad, el uso del atributo `-keystore` es mucho más adecuado.

SERVLETS

El caso de los servlets, que se tratarán de forma exhaustiva en un capítulo posterior, es semejante al de los applets en cuestiones de acceso a ficheros. En lo que respecta a los permisos de acceso a esos ficheros, aunque los servlets sean invocados por un navegador, ellos siempre se ejecutan bajo la política de seguridad que establezca el servidor Web sobre el que ejecutan. Por ejemplo, el servlet que se presenta en el ejemplo `Java906.java`, se ejecuta sin restricción alguna bajo *Java WebServer*. El lector debe colocar en el `CLASSPATH` el paquete que contiene las clases de los servlets para poder compilarlo, tal como se explica en el capítulo dedicado al desarrollo de servlets.

El servlet, cuando es invocado, presenta al usuario una página muy simple con un formulario que permite introducir una cadena de texto y pulsar un botón que será el que invoque al método *doPost()* del servlet, que genera otra página de respuesta consistente en la misma cadena que se ha introducido en el formulario de la página, que graba en un fichero, y otra cadena de texto que lee de un fichero. La figura 9.3 reproduce la página de respuesta del servlet.

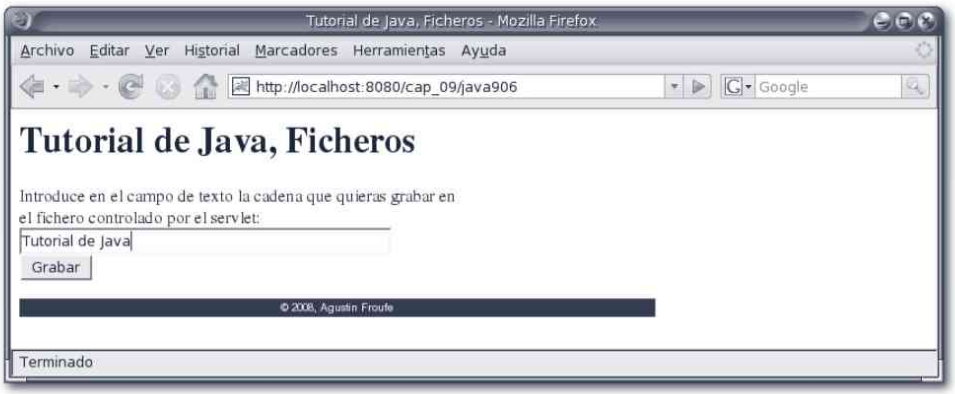


Figura 9.3

ACCESO ALEATORIO

Los ejemplos que se han mostrado en este capítulo son muy simples y solamente muestran al lector la forma en que se abren, leen, escriben y cierran ficheros, manipulando los arrays de datos completos. Evidentemente, una de las primeras cosas que querrá investigar el lector será la incorporación a un fichero existente de más datos y la lectura de cierta información de un fichero ya existente. Aunque el lector debería estar en condiciones de afrontar por sí solo un programa de estas características, el ejemplo `Java907.java`, del que se reproduce a continuación el bloque `try-catch` encargado de la manipulación y acceso a los ficheros, es una

aproximación a la realización de esas tareas, tomando como base el ejemplo Java902.java.

```
try {
  // Porción de código encargada de grabar en fichero el contenido
  // que se introduzca en el campo de texto
  // Primero se recupera el texto del campo de texto y se convierte
  // a un array de bytes
  String texto = campoTexto.getText();
  byte b[] = texto.getBytes();

  // Se crea un objeto File correspondiente al fichero donde se va a
  // grabar el texto
  File fichSalida = new File( dir +
    File.separatorChar+ "textoe.txt" );
  // Se crea el canal de salida conectado a ese fichero, en este caso
  // como fichero de acceso aleatorio y preparado para lectura y para
  // escritura
  RandomAccessFile canalSalida =
    new RandomAccessFile( fichSalida,"rw" );
  // Nos colocamos al final del fichero
  canalSalida.seek( fichSalida.length() );
  // Se escribe el contenido del array de bytes en el fichero
  canalSalida.write( b );
  // Escribimos un carácter de nueva línea, para que las nuevas
  // incorporaciones de texto al fichero se inicien en la línea
  // siguiente a la que se acaba de introducir
  canalSalida.writeByte( 10 );
  // Se cierra el canal
  canalSalida.close();

  // Porción de código encargada de recuperar del fichero el
  // contenido que se presenta en la zona correspondiente al campo de
  // texto, una vez que se ha grabado en fichero el array de bytes. Se
  // crea un objeto File para referenciar al fichero del que se va a
  // recuperar el contenido
  File fichEntrada = new File( dir +
    File.separatorChar+ "textol.txt" );
  // Se crea el canal de entrada para leer el texto
  FileInputStream canalEntrada = new FileInputStream( fichEntrada );
  // Creamos un array de bytes para almacenar el contenido del fichero
  byte bt[] = new byte[(int)fichEntrada.length()];
  // Se lee el fichero
  int numBytes = canalEntrada.read( bt );
  // Se convierte el array de bytes a la cadena que se presenta en la
  // ventana
  String cadena = new String( bt );
  etiqueta.setText( cadena );
  // Se cierra el canal de comunicación con el fichero
  canalEntrada.close();
} catch( IOException e ) {
  e.printStackTrace();
  }
```

LA ARQUITECTURA NIO

La funcionalidad de entrada/salida en Java siempre ha estado en manos de *streams*, bien de tipo *byte* o de tipo *character*. Sin embargo, los *streams* no proporcionan toda la funcionalidad necesaria para los programadores, sino solamente para operaciones básicas de lectura y escritura y, además, son bloqueantes; es decir, una lectura de un **InputStream** siempre intentará devolver un byte, esperando incluso a que esté disponible si es necesario, o estén disponibles el mismo número de bytes que se hayan solicitado. De igual modo, las operaciones de salida también escriben siempre todos los datos que se pasen.

Este comportamiento no siempre es el deseado, porque es posible querer leer datos si están disponibles, pero seguir haciendo cosas en caso contrario, esperando la llegada de esos datos. Por ejemplo, si se trata de un reproductor de música, mientras espera a que llegue otro paquete de datos, puede ir ejecutando lo que ya se haya leído. El uso de *buffers* para lectura y escritura supone un paso más allá de las limitaciones anteriores, pero tienen el inconveniente de que no es fácil su programación cuando se necesitan realizar operaciones complejas, porque los buffers están ocultos al programador. Es decir, o el programador implementa la lógica de procesado de buffers en base a la lectura de arrays de bytes o utiliza un **BufferedReader**, teniendo en cuenta que solamente podrá extraer objetos de tipo **String**.

La arquitectura de entrada/salida de Java **nio** intenta en parte solucionar esos problemas. Para ello proporciona una serie de clases de tipo **Buffer** que tratan los arrays de bytes y se aprovechan de las características de los nuevos sistemas operativos que son capaces de mapear el contenido de un fichero en un buffer de memoria interna, circunstancia que permite accesos aleatorios muy rápidos.

La transferencia de datos hacia y desde el buffer se realiza a través de *canales*. Un canal, o **Channel**, representa el origen y/o el destino de datos, por ejemplo **SocketChannel** y **FileChannel**. También están disponibles objetos de tipo **Pipe** que permiten mover datos entre diferentes partes de un mismo programa.

Una diferencia entre *streams* y *canales* es que estos últimos permiten realizar operaciones de lectura y escritura al mismo tiempo. Y también que los canales pueden ser no bloqueantes. En el caso de que los canales se abran sobre un fichero siempre son bloqueantes, pero en el caso de canales sobre sockets es posible seleccionar cualquiera de las dos características. Si se activa la transferencia no bloqueante, cuando se invoca al método de lectura es posible que se realice ésta cuando todavía no se han recibido todos los datos; y lo mismo sucede con la escritura, en cuyo caso puede que no se transfiera el contenido completo del buffer. Probablemente esto suene anárquico al lector, pero hay situaciones en las que resulta la única solución posible, por ejemplo en las transmisiones de vídeo, en donde se va escribiendo en los canales todo el contenido del buffer, pero no a expensas del procesamiento de ese vídeo.

En general, toda la funcionalidad ofrecida por los objetos de tipo **InputStream** y **OutputStream** se proporciona ahora a través de canales, de un modo más optimizado. No obstante, la clase **Channel** proporciona métodos estáticos para interacturar con el antiguo API de entrada/salida de Java, que no está descartado en absoluto; al contrario, hay algunas partes que han sido mejoradas e incluso reescritas. La más notable es la correspondiente a las operaciones de entrada/salida a través de *bloques*, que ahora funciona en un entorno de tareas mucho más eficiente y más sencillo de controlar.

Buffers

Los buffers corresponden fundamentalmente a una forma sencilla de representar un array de bytes. Para añadir datos a un buffer se utiliza el método *put()* del propio buffer, o el método *read()* de los canales. La lectura de datos se realiza a través del método *get()*, o llamando al método *write()* de los canales.

Todos los buffers tienen cuatro propiedades básicas, que se describen a continuación.

1. *Capacidad*. Ésta es una propiedad que no se puede modificar una vez que se haya creado el buffer. Indica el contenido máximo de datos asignado a ese buffer.

2. *Límite*. Es una marca de fin de buffer, que se puede desplazar dentro de un buffer con capacidad ya determinada para permitir cambiar dinámicamente el tamaño de la parte utilizable del buffer. Por defecto, corresponde al mismo valor que la capacidad. No se puede leer y escribir en el buffer más allá del límite y no se puede mover el límite más allá, obviamente, de la capacidad del buffer.

3. *Posición*. Corresponde a la localización dentro del buffer donde se realizará la siguiente acción relativa de lectura o escritura. Todas estas operaciones de lectura/escritura pueden realizarse de forma *absoluta*, especificando un desplazamiento desde el origen del buffer, o de forma *relativa*, referida a la posición actual. Funciona de forma semejante a un *puntero* en un archivo o a un *cursor* en una tabla de base de datos.

4. *Marca*. Es una posición etiquetada en el buffer. Siempre debe estar situada antes de la posición actual y puede utilizarse el método *reset()* para volver a ella. Por defecto, no hay ninguna marca definida, entendiendo que siempre existe una marca implícita asignada a la posición cero.

Un problema siempre presente con los procesos de entrada/salida tradicionales en Java es el no poder interactuar con los flujos de datos binarios de fuentes que no sean Java. El mapeo de caracteres es adecuado para aplicaciones basadas en texto, como un servidor HTTP, pero cuando se trata de imágenes, sonido, vídeo o incluso datos numéricos, Java siempre ha tenido problemas en el control, fundamentalmente debido a la característica multiplataforma de Java. Un ejemplo claro es el orden de los bytes; por ejemplo, a la hora de manipular un dato de tipo short, formado por dos bytes, para

Java el primer byte es el más significativo, lo cual no es cierto en todas las plataformas; por lo tanto si se codifica la lectura de un fichero diseñado para una plataforma en la cual el segundo byte es el más significativo, el programador debería suplir la carencia de Java.

Ahora **NIO** ofrece una solución, ya que permite indicar el orden de los bytes mediante la clase **ByteOrder**, que define las constantes `BIG_ENDIAN` y `LITTLE_ENDIAN`, de forma que se consiga un mapeado correcto en Java cuando se escriben o extraen datos de cualquier archivo o flujo de datos.

El uso de buffers tiene especial importancia cuando se trata de analizar archivos XML utilizando analizadores de tipo DOM, permitiendo mapear grandes cantidades de datos del archivo en buffers de memoria y luego en objetos **DOM**, cuando se requiera, en lugar de hacerlo en un único paso.

Canales

La interfaz **Channel** es muy simple, solamente dispone de los métodos *isOpen()* y *close()*. Un canal se abre en el momento que se crea y nunca puede ser vuelto a abrir una vez que se haya cerrado. Para leer y escribir datos en un canal se utilizan los métodos *read()*, definido en la interfaz **ReadableByteChannel**, y *write()*, definido en la interfaz **WritableByteChannel**.

La interfaz de conveniencia **ByteChannel** consolida las dos anteriores, y es la que implementan los dos canales más importantes: **SocketChannel** y **FileChannel**.

Un canal de tipo **FileChannel** es un canal de lectura o escritura dependiendo de cómo se abra. Será un canal de escritura si se ha creado desde un objeto **FileOutputStream** o un objeto **RandomAccessFile** abierto como lectura/escritura, y será un canal de lectura si se ha creado a partir de un **FileInputStream** o desde un **RandomAccessFile**.

El tamaño del fichero está disponible desde el canal a través del método *size()*. El tamaño puede incrementarse si se escribe más allá del tamaño actual del fichero y se puede reducir llamando al método *truncate()*. El canal también dispone de un *puntero* cuya posición se puede conocer y fijar mediante el método *position()*. La posición en donde se encuentre el puntero será la siguiente en leer o escribir, según la acción que se realice.

El ejemplo `Java908.java` es muy sencillo y solamente trata de copiar un fichero, para lo cual se deben indicar tanto el archivo de origen como el archivo de destino.

La línea más interesante del programa es la llamada al método *transferFrom()*, que ejecuta una transferencia optimizada de bytes entre los dos canales que se han abierto hacia el archivo de entrada, lectura, y hacia el archivo de salida, escritura.

```
// Éste es el método que realiza la copia del contenido
// del archivo origen en el archivo destino
salida.transferFrom( entrada,0L,(int)entrada.size() );
```

Otras partes que incorpora el API NIO, además de buffers y canales, corresponden a **Charsets**, que permiten el tratamiento de flujos de caracteres, disponiendo de codificadores y decodificadores asociados que son capaces realizar la traslación entre bytes y caracteres Unicode. Además, NIO también incorpora **Selectors**, que si se asocian a los canales permiten la implementación de un sistema de entrada/salida multiplexado y no bloqueante. Por último, NIO también proporciona soporte para expresiones regulares que permiten trabajar con patrones de reconocimiento, búsqueda, sustitución, etc.

En este último caso, los desarrolladores pueden utilizar la posibilidad de tratamiento de expresiones regulares, por ejemplo, para imprimir todas las líneas de un fichero de texto que contengan una cadena patrón que describe a su vez un conjunto de cadenas. El ejemplo `Java909.java` realiza esta tarea, leyendo cada línea del fichero de texto que se pasa como argumento en la línea de comandos e imprime aquellas que coinciden con la expresión que también debe indicarse en la línea de comandos. Por ejemplo, para imprimir todas las líneas que contengan la palabra "Buscar" en el fichero `Java909.java`, se ejecutaría el comando:

```
% java Java909 Buscar Java909.java
```

Y para imprimir las líneas que contengan "Buscar" y también "buscar", es decir, independientemente de si la primera letra de la palabra es mayúscula o minúscula, el comando de invocación sería:

```
% java Java909 "Buscar|buscar" Java909.java
```

Las partes más interesantes del código del ejemplo son aquellas en las que NIO utiliza las capacidades de reconocimiento de patrones, en base al tratamiento de expresiones regulares. Para realizar esta tarea, la clase utiliza varias sentencias.

```
Pattern patron = Pattern.compile( args[0] );
```

Esta sentencia compila la expresión regular que se pasa desde la línea de comandos, en un patrón más eficiente para realizar la búsqueda, en base al reconocimiento de patrones, en el fichero de texto. Devuelve una referencia al patrón ya compilado.

```
Matcher m = patron.matcher( "" );
```

Aquí se obtiene un objeto de tipo **Matcher** que se encargará de realizar las acciones de comprobación del texto. Se le pasa el argumento vacío ("") al método del objeto **Pattern**, para indicar que el patrón real de búsqueda se proporcionará en una sentencia posterior.

A continuación, el bucle `while` se encarga de recorrer todas las líneas del fichero.

```
while( (linea=br.readLine()) != null ) {
  // Reseteamos el objeto Matcher para que comience la búsqueda
  // al principio de la línea
  m.reset( linea );
  // Buscamos el patrón. Obtenemos true si se localiza
  if( m.find() ) {
    // Imprimimos la línea de texto que contiene el patrón
    System.out.println( linea );
    }
  }
```

Dentro del bucle se realizan dos acciones. La primera invoca al método *reset()* del objeto **Matcher**. Al contrario de lo que ocurre con los objetos de tipo **Pattern**, los objetos **Matcher** mantienen información de estado, incluyendo la línea de texto en que buscar y la información de dónde comienza la cadena que está buscando. El método *reset()* salva la línea de texto y asegura que el objeto **Matcher** siempre comience a buscar en la posición inicial.

La otra sentencia del bucle es la invocación del método *find()* del objeto **Matcher**, que es el que realmente realiza la búsqueda de la cadena patrón en la línea salvada por *reset()*. Si la encuentra, devuelve `true` y se imprime en la salida estándar.

CAPÍTULO 10

EXCEPCIONES EN JAVA

Una excepción es un evento que ocurre durante la ejecución de un programa y detiene el flujo normal de la secuencia de instrucciones de ese programa; en otras palabras, una excepción es una condición anormal que surge en una secuencia de código durante su ejecución.

Las excepciones en Java están destinadas, al igual que en el resto de los lenguajes que las soportan, para la detección y corrección de errores. Si hay un error, la aplicación no debería morirse y generar un *core* (o un *crash* en caso del DOS o una *pantalla azul* en el caso de Windows). Se debería lanzar (`throw`) una excepción que a su vez debería capturarse (`catch`) y resolver la situación de error, o bien poder ser tratada finalmente (`finally`) por un gestor por defecto. Utilizadas en forma adecuada, las excepciones aumentan en gran medida la robustez de las aplicaciones.

La gestión de excepciones en Java proporciona un mecanismo *excepcionalmente* poderoso para controlar programas que tengan características dinámicas durante su ejecución. Las excepciones son formas muy limpias de manejar errores y problemas inesperados en la lógica del programa, y no deberían considerarse como un mecanismo general de ramificaciones o un tipo de sentencias de salto.

La utilización adecuada de las excepciones proporcionará un refinamiento profesional al código que cualquier usuario futuro de las aplicaciones que salgan de la mano del programador agradecerá con toda seguridad.

CLASIFICACIÓN DE EXCEPCIONES

Las excepciones se clasifican en función de quién es el originador. Se dividen en tres clases:

1. Excepciones de la Máquina Virtual Java. Casi nunca se puede hacer nada cuando se producen errores fatales, como un fallo de memoria o cosas similares. Lo único que se puede hacer ante este tipo de excepciones es detener el sistema y reiniciarlo.

2. Excepciones de aplicación. Son excepciones generadas por la aplicación o por alguna de las librerías que ésta incluye. También se llaman excepciones ***checked***, porque es necesario capturarlas en un bloque `try-catch` o relanzarlas mediante el uso de `throw`. En tiempo de compilación se detecta si se están capturando o no. En este tipo de excepciones, quien invoca al método que está generando la excepción está obligado a capturarla y por tanto, debe implementar su control y presentar un flujo alternativo en la aplicación. Estas excepciones derivan directamente de **Exception**.

3. Excepciones del Sistema. La mayoría de excepciones de este tipo derivan de **RuntimeException** y son lanzadas por código defectuoso, como **NullPointerException** o **ArrayOutOfBoundsException**. Reciben también el nombre de excepciones ***unchecked***, porque no es obligatorio atraparlas, al no saber a ciencia cierta cuándo van a producirse.

Las excepciones no deben ignorarse nunca. Cuando un método genera una excepción de tipo *checked*, está intentando comunicar que algo puede suceder y aunque la excepción no tenga sentido aparentemente, no se debe ignorar con una sentencia vacía ({}) y continuar como si nada hubiese pasado. En todo caso, enviarla a la salida estándar o convertida en una excepción *unchecked* y relanzarla.

No es conveniente capturar en primer lugar excepciones de alto nivel, del tipo **Exception**, porque en este caso se están capturando tanto las excepciones de aplicación como las excepciones del sistema. No indica un buen nivel de programación capturar excepciones de alto nivel, demuestra poco conocimiento de lo que el código de la aplicación puede provocar. Sin embargo, sí es útil cerrar siempre el ciclo de captura de excepciones atrapando la excepción del alto nivel **Exception** y enviarla a la salida estándar o de error.

MANEJO DE EXCEPCIONES

A continuación, se muestra cómo se utilizan las excepciones, reconvirtiendo en primer lugar el applet de saludo en una versión iterativa en `HolaIte.java`:

```
import java.awt.*;
import java.applet.Applet;

public class HolaIte extends Applet {
  private int i = 0;
  private String Saludos[] = {
    "Hola Mundo!",
    "HOLA Mundo!",
    "HOLA MUNDO!!"
```

```
    };

  public void paint( Graphics g ) {
    try {
      g.drawString( Saludos[i],25,25 );
      } catch( ArrayIndexOutOfBoundsException e ) {
        g.drawString( "Saludos desbordado",25,25 );
    } catch( Exception e ) {
      // Cualquier otra excepción
      System.out.println( e.toString() );
    } finally {
      System.out.println( "Esto se imprime siempre!" );
      }
    i++;
    }
  }
```

Normalmente, un programa termina con un mensaje de error cuando se lanza una excepción. Sin embargo, Java tiene mecanismos para excepciones que permiten ver qué excepción se ha producido e intentar recuperarse de ella.

La palabra clave `finally` define un bloque de código que se quiere que sea ejecutado siempre, tanto si se capturó la excepción como si no. En el ejemplo anterior, la salida en la consola siempre mostrará el mensaje siguiente:

```
¡Esto se imprime siempre!
```

Realizando la ejecución con el *appletviewer*, y seleccionado la opción *Reiniciar* del menú superior *Subprograma* varias veces, aparecerán en la ventana los distintos saludos, hasta que se desborde el índice del array, en cuyo caso aparecerá el mensaje "*Saludos desbordado*" en su lugar, producido al saltar la excepción.

GENERAR EXCEPCIONES EN JAVA

Cuando se produce una condición excepcional en el transcurso de la ejecución de un programa, se debería generar, o lanzar, una excepción. Esta excepción es un objeto derivado directa, o indirectamente, de la clase **Throwable**. Tanto el intérprete Java como muchos métodos de las múltiples clases de Java pueden lanzar excepciones.

La clase **Throwable** tiene dos subclases: **Error** y **Exception**. Un **Error** indica que se ha producido un fallo no recuperable, del que no se puede recuperar la ejecución normal del programa, por lo tanto, en este caso no hay nada que hacer. Los errores, normalmente, hacen que el intérprete Java presente un mensaje en el dispositivo estándar de salida y concluya la ejecución del programa. El único caso en que esto no es así, es cuando se produce la muerte de una tarea, en cuyo caso se genera el error **ThreadDead**, que lo que hace es concluir la ejecución de esa tarea, pero ni presenta mensajes en pantalla ni afecta a otras tareas que se estén ejecutando.

Una **Exception** indicará una condición anormal que puede ser subsanada para evitar la terminación de la ejecución del programa. Hay nueve subclases de la clase **Exception** ya predefinidas, y cada una de ellas, a su vez, tiene numerosas subclases.

Para que un método Java lance excepciones, es necesario indicarlo expresamente.

```
void MetodoAsesino() throws NullPointerException,CaidaException
```

Se pueden definir excepciones propias explícitamente, no hay por qué limitarse a las que Java predefine y a sus subclases; bastará con extender la clase **Exception** y proporcionar la funcionalidad extra que requiera el tratamiento de esa excepción. También pueden producirse excepciones no de forma explícita como en el caso anterior, sino de forma implícita cuando se realiza alguna acción ilegal o no válida.

Las excepciones, pues, pueden originarse de dos modos: el programa hace algo ilegal (caso normal), o el programa explícitamente genera una excepción ejecutando la sentencia `throw` (caso menos normal). La sentencia `throw` tiene la siguiente forma:

```
throw ObtejoException;
```

El objeto **ObjetoException** es un objeto de una clase que extiende la clase **Exception**.

El ejemplo `Java1001.java` origina una excepción de división por cero:

```
class Java1001 {
  public static void main( String a[] ) {
    int i=0, j=0, k;
    k = i/j;    // Origina un error de division-by-zero
    }
  }
```

Si se compila y ejecuta esta aplicación, se obtendrá la salida por pantalla:

```
% javac Java1001.java
% java Java1001
    Exception in thread "main" java.lang.ArithmeticException: / by zero
            at Java1001.main(Java1001.java:28)
```

Las excepciones *predefinidas*, como **ArithmeticException**, pertenecen al grupo de las excepciones *unchecked*, que sería mejor llamarlas excepciones *irrecuperables*. En contraste con las excepciones que se generan explícitamente, a petición del programador, mucho menos severas y de las cuales, en la mayoría de los casos, no resulta complicado recuperarse. Por ejemplo, si un fichero no puede abrirse, se puede preguntar al usuario que indique otro fichero; o si una estructura de datos se encuentra completa, siempre se podrá sobrescribir algún elemento que ya no se necesite.

Todas las excepciones deben llevar un mensaje asociado a ellas al que se puede acceder utilizando el método *getMessage()*, que presentará un mensaje describiendo el error o la excepción que se ha producido.

Si se desea, se pueden invocar otros métodos de la clase **Throwable** que presentan un *traceado* de la pila en donde se ha producido la excepción, o también se pueden invocar para convertir el objeto **Exception** en una cadena, que siempre es más inteligible y agradable a la vista.

El compilador Java obliga al programador a proporcionar el código de manejo o control de algunas de las excepciones predefinidas por el lenguaje. Por ejemplo, el programa `Java1002.java`, no compilará porque no se captura la excepción **InterruptedException** que puede lanzar el método *sleep()*.

```
class Java1002 {
  public static void main( String args[] ) {
    Java1002 obj = new Java1002();
    obj.miMetodo();
    }

  void miMetodo() {
    // Aquí se produce el segundo error de compilación, porque no se
    // está declarando la excepción que genera este método
    Thread.currentThread().sleep( 1000 ); // currentThread() genera
                                          // una excepción
    }
  }
```

Éste es un programa muy simple, que al intentar compilar, producirá el siguiente error de compilación:

```
% javac Java1002.java
Java1002.java:43: unreported exception java.lang.InterruptedException;
   must be caught or declared to be thrown
   Thread.currentThread().sleep( 1000 ); // currentThread() genera
                                ^
```

Como no se ha previsto la captura de la excepción, el programa no compila. El error identifica la llamada al método *sleep()* como origen del problema, así que la siguiente versión del programa, `Java1003.java` soluciona el problema generado por esta llamada.

```
class Java1003 {
  public static void main( String args[] ) {
    // Se instancia un objeto
    Java1003 obj = new Java1003();
    // Se crea la secuencia try/catch que llamará al método que lanza
    // la excepción
    try {
      // Llamada al método que genera la excepción
      obj.miMetodo();
    } catch( InterruptedException e ) {
```

```
      } // Procesa la excepción
    }

  // Éste es el método que va a lanzar la excepción
  void miMetodo() throws InterruptedException {
    Thread.currentThread().sleep( 1000 ); // currentThread() genera
                                          // una excepción
    }
  }
```

Lo único que se ha hecho es indicar al compilador que el método *miMetodo()* puede lanzar excepciones de tipo **InterruptedException**. Con ello se consigue propagar la excepción que genera el método *sleep()* al nivel siguiente de la jerarquía de clases. Es decir, en realidad no se resuelve el problema, sino que se está pasando a otro método para que lo resuelva él.

En el método *main()* se proporciona la estructura que resuelve el problema de compilación, aunque no haga nada, por el momento. Esta estructura consta de un bloque `try` y un bloque `catch`, que se puede interpretar como que intentará ejecutar el código del bloque `try` y si hubiese una nueva excepción del tipo que indica el bloque `catch`, se ejecutaría el código de este bloque, sin ejecutar nada del `try`.

La transferencia de control al bloque `catch` no es una llamada a un método, es una transferencia incondicional, es decir, no hay un retorno de un bloque `catch`.

CREAR EXCEPCIONES PROPIAS

También el programador puede lanzar sus propias excepciones, como se ha indicado anteriormente, extendiendo la clase **System.exception**. Por ejemplo, considérese un programa cliente/servidor. El código cliente se intenta conectar al servidor, y durante 5 segundos se espera a que conteste el servidor. Si el servidor no responde, el servidor lanzaría la excepción de exceso de tiempo, *time-out*:

```
class ServerTimeOutException extends Exception {}
public void conectame( String nombreServidor ) throws Exception {
    int exito;
    int puerto = 80;
    exito = open( nombreServidor,puerto );
    if( exito == -1 )
      throw ServerTimeOutException;
    }
```

Si se quieren capturar las excepciones propias, se deberá utilizar la sentencia `try`:

```
public void encuentraServidor() {
 ...
 try {
   conectame( servidorDefecto );
 } catch( ServerTimeOutException e ) {
   g.drawString(
     "Time-out del Servidor, intentando alternativa",5,5 );
```

```
   conectame( servidorAlterno );
   }
...
}
```

Cualquier método que lance una excepción también debe capturarla, o declararla como parte de la interfaz del método. Cabe preguntarse entonces el porqué de lanzar una excepción si hay que capturarla en el mismo método. La respuesta es que las excepciones no simplifican el trabajo del control de errores. Tienen la ventaja de que se puede tener muy localizado el control de errores y no hay que controlar miles de valores de retorno, pero no van más allá.

Y todavía se puede plantear una pregunta más, al respecto de cuándo crear excepciones propias y no utilizar las múltiples que ya proporciona Java. Como guía, se pueden plantear las siguientes cuestiones, y si la respuesta es afirmativa, lo más adecuado será implementar una clase **Exception** nueva y, en caso contrario, utilizar una del sistema.

- ¿Se necesita un tipo de excepción no representado en la que proporciona el entorno de desarrollo Java?
- ¿Ayudaría a los usuarios si pudiesen diferenciar las excepciones propias de las que lanzan las clases de otros desarrolladores?
- ¿Si se lanzan las excepciones propias, los usuarios tendrán acceso a esas excepciones?
- ¿El *package* propio debe ser independiente y auto-contenido?

CAPTURAR EXCEPCIONES

Las excepciones lanzadas por un método que pueda hacerlo deben capturarse en un bloque `try/catch` o `try/finally`.

```
int valor;
try {
  for( x=0,valor = 100; x < 100; x ++ )
    valor /= x;
} catch( ArithmeticException e ) {
  System.out.println( "Matemáticas locas!" );
} catch( Exception e ) {
  System.out.println( "Se ha producido un error" );
  }
```

try

Es el bloque de código donde se prevé que se genere una excepción. Es como decir "*intenta estas sentencias y mira a ver si se produce una excepción*". El bloque `try` tiene que ir seguido, al menos, por una cláusula `catch` o una cláusula `finally`.

La sintaxis general del bloque `try` consiste en la palabra clave `try` y una o más sentencias entre llaves

```
try {
  // Sentencias Java
  }
```

Puede haber más de una sentencia que genere excepciones, en cuyo caso habría que proporcionar un bloque `try` para cada una de ellas. Algunas sentencias, en especial aquellas que invocan a otros métodos, pueden lanzar, potencialmente, muchos tipos diferentes de excepciones, por lo que un bloque `try` consistente en una sola sentencia requeriría varios controladores de excepciones.

También se puede dar el caso contrario, en que todas las sentencias, o varias de ellas, que puedan lanzar excepciones se encuentren en un único bloque `try`, con lo que habría que asociar múltiples controladores a ese bloque. Aquí la experiencia del programador es la que cuenta y es el propio programador el que debe decidir qué opción tomar en cada caso.

Los controladores de excepciones deben colocarse inmediatamente después del bloque `try`. Si se produce una excepción dentro del bloque `try`, esa excepción será manejada por el controlador que esté asociado con el bloque `try`.

catch

Es el código que se ejecuta cuando se produce la excepción. Es como decir "*controlo cualquier excepción que coincida con mi argumento*". No hay código alguno entre un bloque `try` y un bloque `catch`, ni entre bloques `catch`. La sintaxis general de la sentencia `catch` en Java es la siguiente:

```
catch( UnTipoThrowable nombreVariable ) {
  // sentencias Java
  }
```

El argumento de la sentencia declara el tipo de excepción que el controlador, el bloque `catch`, va a manejar.

En este bloque hay que asegurarse de colocar código que no genere excepciones. Se pueden colocar sentencias `catch` sucesivas, cada una controlando una excepción diferente. No se debería intentar capturar todas las excepciones con una sola cláusula, como ésta:

```
catch( Excepcion e ) { ...
```

Esto representaría un uso demasiado general, podrían llegar muchas más excepciones de las esperadas. En este caso es mejor dejar que la excepción se propague hacia arriba y dar un mensaje de error al usuario.

Se pueden controlar grupos de excepciones, es decir, se pueden controlar, a través del argumento, excepciones semejantes. Por ejemplo:

```
class Limites extends Exception {}
class demasiadoCalor extends Limites {}
class demasiadoFrio extends Limites {}
class demasiadoRapido extends Limites {}
class demasiadoCansado extends Limites {}
  . . .
try {
    if( temp > 40 )
      throw( new demasiadoCalor() );
    if( dormir < 8 )
      throw( new demasiado Cansado() );
  } catch( Limites lim ) {
    if( lim instanceof demasiadoCalor ) {
      System.out.println( "Capturada excesivo calor!" );
      return;
      }
    if( lim instanceof demasiadoCansado ) {
      System.out.println( "Capturada excesivo cansancio!" );
      return;
      }
  } finally
    System.out.println( "En la clausula finally" );
```

La cláusula `catch` comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecuta el bloque. El operador `instanceof` se utiliza para identificar exactamente cuál ha sido la identidad de la excepción.

Cuando se colocan varios controladores de excepción, es decir, varias sentencias `catch`, el orden en que aparecen en el programa es importante, especialmente si alguno de los controladores engloba a otros en el árbol de jerarquía. Se deben colocar primero los controladores que manejen las excepciones más alejadas en el árbol de jerarquía, porque de otro modo, estas excepciones podrían no llegar a tratarse si son capturadas por un controlador más general colocado anteriormente.

finally

Es el bloque de código que se ejecuta siempre, haya o no excepción. Hay una cierta controversia entre su utilidad, pero, por ejemplo, podría servir para servir de bitácora (*log*) o hacer un seguimiento de lo que está pasando, porque como se ejecuta siempre, puede dejar grabado si se producen excepciones y si el programa se ha recuperado de ellas o no.

Este bloque `finally` puede ser útil cuando no hay ninguna excepción. Es un trozo de código que se ejecuta independientemente de lo que se haga en el bloque `try`.

A la hora de tratar una excepción, se plantea el problema de qué acciones se van a tomar. En la mayoría de los casos, bastará con presentar una indicación de error al

usuario y un mensaje avisándolo de que se ha producido un error y que decida si quiere o no continuar con la ejecución del programa.

Por ejemplo, se podría disponer de un diálogo como el que se presenta en el código siguiente:

```
public class DialogoError extends Dialog {
  DialogoError( Frame padre ) {
    super( padre,true );
    setLayout( new BorderLayout() );
    // Presentamos un panel con continuar o salir
    Panel p = new Panel();
    p.add( new Button( "¿Continuar?" ) );
    p.add( new Button( "Salir" ) );
    add( "Center",new Label(
      "Se ha producido un error. ¿Continuar?" ) )
    add( "South",p );
    }

  public boolean action( Event evt,Object obj ) {
    if( "Salir".equals( obj ) ) {
      dispose();
      System.exit( 1 );
      }
    return false;
    }
  }
```

Y la invocación, desde algún lugar en que se suponga que se generarán errores, podría ser como sigue:

```
try {
  // Código peligroso
} catch( AlgunaExcepcion e ) {
  VentanaError = new DialogoError( this );
  VentanaError.show();
  }
```

Lo cierto es que hay autores que indican la inutilidad del bloque `finally`, mientras que desde el *Java Tutorial* de **Sun** se justifica plenamente su existencia. El lector deberá revisar todo el material que esté a su alcance y crearse su propia opinión al respecto.

En el siguiente programa, `Java1004.java`, se intenta demostrar el poder del bloque `finally`. En él, un controlador de excepciones intenta terminar la ejecución del programa ejecutando una sentencia `return`. Antes de que la sentencia se ejecute, el control se pasa al bloque `finally` y se ejecutan todas las sentencias de este bloque. Luego el programa termina. Es decir, quedaría demostrado que el bloque `finally` no tiene la última palabra.

```
class MiExcepcion extends Exception {
  int datoInformacion;
```

```
  // Constructor
  MiExcepcion( int datos ) {
    // Guarda la información de diagnóstico en el objeto
    datoInformacion = datos;
    }

  // Sobrescribe el método de Throwable
  public String getMessage() {
    return( "La maxima es: Compra Barato, Vende Caro\n"
      + "El valor de datoInformacion es: " + datoInformacion );
    }
  }

class Java1004 {
  public static void main( String args[] ) {
    try {
      for( int cnt=0; cnt < 5; cnt++ ) {
        // Lanza la excepción propia y pasa la información
        // si "cnt" es 3
        if( cnt == 3 )
          throw new MiExcepcion( 3 );
        // Transfiere el control antes de que "cnt" sea 3
        System.out.println( "Procesando datos para cnt = :"
          + cnt );
        }
      System.out.println( "Esta linea no debe ejecutarse nunca." );
    } catch( MiExcepcion e ) {
      System.out.println(
        "Controlador de Excepciones, captura el mensaje\n"+
        e.getMessage() );
      System.out.println( "Controlador de Excepciones, "+
        "intentando finalizar la ejecucion.\n"+
        "Ejecutando la sentencia return." );
      return;
    } finally {
      System.out.println( "Bloque finally, para probar que "+
        "se entra en el independientemente\n"+
        "de la sentencia return del Controlador de Excepciones." );
      }
    System.out.println(
      "Esta sentencia nunca se ejecutara debido a la sentencia "+
      "return del controlador de excepciones." );
    }
  }
```

El programa redefine el método *getMessage()* de la clase **Throwable**, porque este método devuelve `null` si no es adecuadamente redefinido por la nueva clase excepción.

throw

La sentencia `throw` se utiliza para lanzar explícitamente una excepción. En primer lugar se debe obtener un descriptor de un objeto **Throwable**, bien mediante un parámetro en una cláusula `catch` o bien creándolo mediante el operador `new`. La forma general de la sentencia `throw` es:

```
throw ObjetoThrowable;
```

El flujo de la ejecución se detiene inmediatamente después de la sentencia `throw`, y nunca se llega a la sentencia siguiente. Se inspecciona el bloque `try` que la engloba más cercano para ver si tiene la cláusula `catch` cuyo tipo coincide con el del objeto o instancia **Throwable**. Si se encuentra, el control se transferirá a esa sentencia. Si no, se inspeccionará el siguiente bloque `try` que la engloba, y así sucesivamente, hasta que el gestor de excepciones más externo detiene el programa y saca por pantalla la *traza* de lo que hay en la pila hasta que se alcanzó la sentencia `throw`. En el programa `Java1005.java`, se demuestra cómo se hace el lanzamiento de una nueva instancia de una excepción, y también cómo dentro del gestor se vuelve a lanzar la misma excepción al gestor más externo.

```
class Java1005 {
  static void demoproc() {
    try {
      throw new NullPointerException( "demo" );
    } catch( NullPointerException e ) {
      System.out.println( "Capturada la excepcion en demoproc" );
      throw e;
      }
    }

  public static void main( String args[] ) {
    try {
      demoproc();
    } catch( NullPointerException e ) {
      System.out.println( "Capturada de nuevo: " + e );
      }
    }
  }
```

Este programa dispone de dos oportunidades para tratar el mismo error. Primero, *main()* establece un contexto de excepción y después se llama al método *demoproc()*, que establece otro contexto de gestión de excepciones y lanza inmediatamente una nueva instancia de la excepción, que se captura en la línea siguiente. La salida que se obtiene tras la ejecución de esta aplicación es la que se reproduce:

```
% java Java1005
Capturada la excepcion en demoproc
Capturada de nuevo: java.lang.NullPointerException: demo
```

throws

Si un método es capaz de provocar una excepción que no maneja él mismo, debería especificar este comportamiento para que todos los métodos que lo llamen puedan colocar protecciones frente a esa excepción. La palabra clave `throws` se utiliza para identificar la lista posible de excepciones que un método puede lanzar. Para la mayoría de las subclases de la clase **Exception**, el compilador Java obliga a declarar qué tipos podrá lanzar un método. Si el tipo de excepción es **Error** o **RuntimeException**, o cualquiera de sus subclases, no se aplicará esta regla, dado que

no se espera que se produzcan como resultado del funcionamiento normal del programa. Si un método lanza explícitamente una instancia de **Exception** o de sus subclases, excepto de la excepción de *runtime*, se deberá declarar su tipo con la sentencia `throws`. La declaración del método sigue ahora la sintaxis siguiente:

```
type NombreMetodo( argumentos ) throws excepciones { }
```

En el ejemplo `Java1006.java`, el programa intenta lanzar una excepción sin tener código para capturarla, y tampoco utiliza `throws` para declarar que se lanza esta excepción. Por tanto, el código no será posible compilarlo.

```
class Java1006 {
  static void demoproc() {
    System.out.println( "Capturada la excepcion en demoproc" );
    throw new IllegalAccessException( "demo" );
    }
  public static void main( String args[] ) {
    demoproc();
    }
  }
```

El error de compilación que se produce es lo suficientemente explícito:

```
% javac Java1006.java
Java1006.java:32: unreported exception java.lang.
IllegalAccessException; must be caught or declared to be thrown
    throw new IllegalAccessException( "demo" );
    ^
```

Para hacer que este código compile, se convierte en el programa `Java1007.java`, en donde se declara que el método puede lanzar una excepción de *acceso ilegal*, con lo que el problema asciende un nivel más en la jerarquía de llamadas. Ahora *main()* llama a *demoproc()*, que se ha declarado para lanzar una **IllegalAccessException**, por lo tanto se coloca un bloque `try` que pueda capturar esa excepción.

```
class Java1007 {
  static void demoproc() throws IllegalAccessException {
    System.out.println( "Dentro de demoproc" );
    throw new IllegalAccessException( "demo" );
    }

  public static void main( String args[] ) {
    try {
      demoproc();
    } catch( IllegalAccessException e ) {
      System.out.println( "Capturada de nuevo: " + e );
      }
    }
  }
```

PROPAGACIÓN DE EXCEPCIONES

La cláusula `catch` comprueba los argumentos en el mismo orden en que aparezcan en el programa. Si hay alguno que coincida, se ejecutará el bloque y seguirá el flujo de control por el bloque `finally` (si lo hay) y concluirá el control de la excepción.

Si ninguna de las cláusulas `catch` coincide con la excepción que se ha producido, entonces se ejecutará el código de la cláusula `finally` (en caso de que la haya). Lo que ocurre en este caso es exactamente lo mismo que si la sentencia que lanza la excepción no se encontrase encerrada en el bloque `try`.

El flujo de control abandona este método y retorna prematuramente al método que lo llamó. Si la llamada estaba dentro del ámbito de una sentencia `try`, entonces se volverá a intentar el control de la excepción, y así continuamente.

Cuando una excepción no es tratada en la rutina en donde se produce, lo que sucede es que el sistema Java busca un bloque `try-catch` más allá de la llamada, pero dentro del método que lo trajo aquí. Si la excepción se propaga de todas formas hasta la parte superior de la pila de llamadas sin encontrar un controlador específico para la excepción, entonces la ejecución se detendrá dando un mensaje. Es decir, se puede suponer que Java está proporcionando un bloque `catch` por defecto que imprime un mensaje de error, indica las últimas entradas en la pila de llamadas y sale.

No hay ninguna sobrecarga en el sistema por incorporar sentencias `try` al código. La sobrecarga se produce cuando se genera la excepción.

Se ha indicado ya que un método debe capturar las excepciones que genera, o en todo caso, declararlas como parte de su llamada, indicando a todo el mundo que es capaz de generar excepciones. Esto debe ser así para que cualquiera que escriba una llamada a ese método esté advertido de que le puede llegar una excepción, en lugar del valor de retorno normal. Esto permite al programador que llama a ese método, elegir entre controlar la excepción o propagarla hacia arriba en la pila de llamadas. La siguiente línea de código muestra la forma general en que un método declara excepciones que se pueden propagar fuera de él, tal como se ha visto a la hora de tratar la sentencia `throws`:

```
tipo_de_retorno( parametros ) throws e1,e2,e3 { }
```

Los nombres `e1,e2,...` deben ser nombres de excepciones, es decir, cualquier tipo que sea asignable al tipo predefinido **Throwable**. Como en la llamada al método se especifica el tipo de retorno, se está especificando el tipo de excepción que puede generar (en lugar de un objeto **Exception**).

En el diagrama de la figura 10.1 se muestra gráficamente cómo se propaga la excepción que se genera en el código a través de la pila de llamadas durante la ejecución del código:

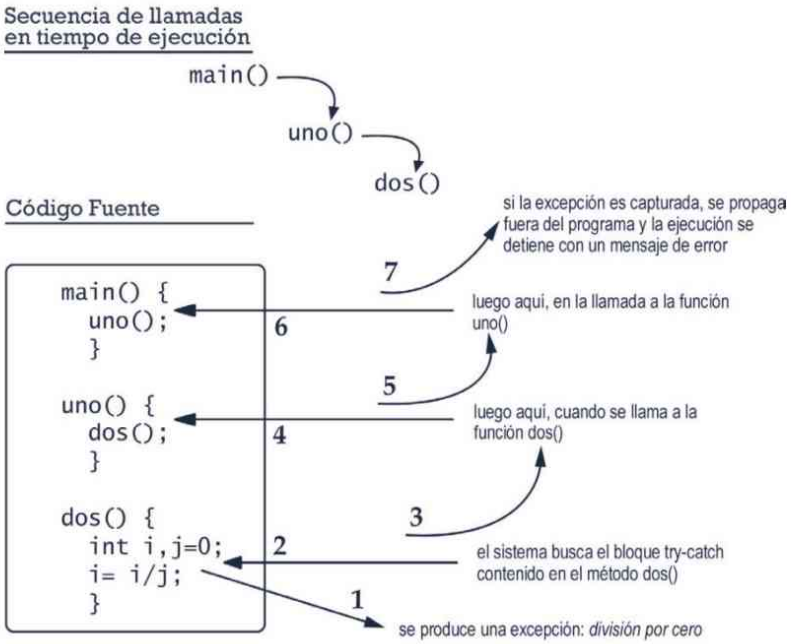


Figura 10.1

Cuando se crea una nueva excepción, derivando de una clase **Exception** ya existente, se puede cambiar el mensaje que lleva asociado. Normalmente, el texto del mensaje proporcionará información para resolver el problema o sugerirá una acción alternativa. Por ejemplo:

```
class SinGasolina extends Exception {
  SinGasolina( String s ) {    // constructor
    super( s );
    }
  ....
  // Cuando se use, aparecerá algo como esto
  try {
    if( j < 1 )
      throw new SinGasolina( "Usando deposito de reserva" );
  } catch( SinGasolina e ) {
    System.out.println( o.getMessage() );
    }
```

Esto, en tiempo de ejecución, originaría la siguiente salida por pantalla:

```
> Usando deposito de reserva
```

Otro método que es heredado de la superclase **Throwable** es *printStackTrace()*. Invocando a este método sobre una excepción se volcarán a pantalla todas las llamadas hasta el momento en donde se generó la excepción (no donde se maneje la excepción). Por ejemplo:

```
// Capturando una excepción en un método
class testcap {
  static int slice0[] = { 0,1,2,3,4 };
  public static void main( String a[] ) {
    try {
      uno();
    } catch( Exception e ) {
      System.out.println( "Captura de la excepcion en main()" );
      e.printStackTrace();
      }
    }

  static void uno() {
    try {
      slice0[-1] = 4;
    } catch( NullPointerException e ) {
      System.out.println( "Captura una excepcion diferente" );
      }
    }
  }
```

Cuando se ejecute ese código, en pantalla observaremos la siguiente salida:

```
> Captura de la excepcion en main()
> java.lang.ArrayIndexOutOfBoundsException: -1
        at testcap.uno(test5p.java:19)
        at testcap.main(test5p.java:9)
```

Con todo el manejo de excepciones la plataforma Java proporciona un método más seguro para el control de errores, además de representar una excelente herramienta para organizar en sitios concretos todo el manejo de los errores y, además, se pueden proporcionar mensajes de error más decentes al usuario indicando qué es lo que ha fallado y por qué, e incluso es posible, a veces, recuperar al programa automáticamente de los errores.

La degradación que se produce en la ejecución de programas con manejo de excepciones está ampliamente compensada por las ventajas que representa en cuanto a seguridad de funcionamiento de esos mismos programas.

ASERCIONES

Las aserciones (***assertions***) fueron introducidas por *Sun Microsystems* en el lenguaje Java desde el JDK 1.4, añadiendo la palabra clave **`assert`** al lenguaje. Las aserciones se utilizan para lanzar un error cuando una expresión booleana definida por el programador es falsa. Se trata pues, fundamentalmente, de expresiones utilizadas para depurar código.

Las aserciones implican nuevas comprobaciones que debe realizar el código, que no redundan además en un beneficio para la ejecución normal de la aplicación; más bien al revés, suponen una carga extra para el programa. Sin embargo, son una de las herramientas más útiles a la hora de detectar fallos en programas. Para evitar su uso

cuando no son necesarias, las aserciones pueden habilitarse o deshabilitarse varias veces en un programa, e incluso se pueden activar a nivel de paquete o clase. No obstante, lo normal es activarlas todas cuando se trata de depurar un programa.

Las aserciones suponen una de las mejores técnicas de depuración en manos del desarrollador, del cual depende que sean útiles o no, en función de dónde las coloque en su código, porque pueden proporcionar información valiosa antes de que se pierda el control del programa. Por ejemplo, no deben utilizarse para detectar errores en los datos de entrada al programa, porque ese programa debe garantizar la comprobación de los argumentos de entrada, proporcionando información al usuario del error, para que introduzca los argumentos correctos. Sin embargo, sí deben utilizarse para asegurar que los datos de entrada a métodos privados son los esperados.

Las aserciones, por tanto, se colocarán en el código para indicar acciones que ni deben ni pueden ocurrir; en teoría, evidentemente, porque luego la práctica hará que ocurran (el señor Murphy se encargará de ello).

En resumen, las aserciones con una herramienta para el programador y no para el usuario. Por ello, en el funcionamiento normal de las aplicaciones estarán desactivadas y se activarán cuando se pretenda detectar algún error.

Las aserciones se pueden colocar para comprobar precondiciones, postcondiciones, invariantes de bucle y cálculos intermedios.

1. Precondición. Cuando empieza a ejecutarse un fragmento de código. Por ejemplo, si el método a comprobar consiste en la división de dos números, se exigirá que el denominador no sea cero, como precondición.

2. Postcondición. Cuando termina la ejecución de un fragmento de código.

3. Invariante de bucle. Corresponde a lo que debe ser cierto mientras un trozo de código se está ejecutando. Un fallo aquí significa que el algoritmo completo no ha funcionado.

4. Cálculos intermedios. Se colocan en el código para comprobar circunstancias puntuales.

Las aserciones en Java tienen dos formas. Una forma sencilla en la que se indica la expresión booleana que debe ser cierta:

```
assert expresion_booleana;
```

En este caso, cuando la expresión booleana sea falsa, se lanzará un **AssertionError**. Tenga en cuenta el lector que se trata de un error, no de una excepción; es decir, se trata como una excepción de tipo *unchecked*.

El código siguiente, correspondiente a la clase **Java1008**, muestra el uso de esta forma de utilización de `assert`.

```
class Java1008 {
  public static void main( String args[] ) {
    int i = 2;
    ++i;
    assert i == 2;
    }
  }
```

La ejecución del programa con las aserciones habilitadas producirá la siguiente salida por pantalla:

```
% java -ea Java1008
Exception in thread "main" java.lang.AssertionError
        at Java1008.main(Java1008.java:31)
```

```
assert expresion_booleana : expresion_o_mensaje_de_error;
```

Esta forma parece más útil porque permite indicar, por ejemplo, el valor de alguna variable que proporcione pistas de qué es lo que ha ido mal en la ejecución del código para que la aserción saltase. Este segundo parámetro puede ser cualquier objeto, incluso `null`.

El código que se reproduce a continuación, clase **Java1009**, utiliza esta segunda forma de `assert` para presentar un mensaje cuando la condición de la aserción falla.

```
class Java1009 {
  public static int miMetodo () {
    int i = 2;
    // Lanzamos la aserción cuando i sea 2 precisamente
    assert i != 2 : "i es igual a 2";
    return( i );
    }

  public static void main( String args[] ) {
    try {
      miMetodo();
    } catch( AssertionError e ) {
      e.printStackTrace();
      }
    }
  }
```

Si se ejecuta el programa activando las aserciones, en la pantalla se observará la salida que se reproduce a continuación:

```
% java -ea Java1009
java.lang.AssertionError: i es igual a 2
        at Java1009.miMetodo(Java1009.java:34)
        at Java1009.main(Java1009.java:40)
```

Tenga en cuenta el lector que, como se ha indicado anteriormente, las aserciones constituyen una herramienta de depuración, por lo cual el código que se escribe en una expresión `assert` no debe ser necesario para la correcta ejecución de la aplicación y, por supuesto, no puede modificar el valor de variable alguna del programa.

En el ejemplo anterior, la captura de la excepción **AssertionError** no es necesaria, se ha incluido a efectos ilustrativos, porque ya se ha indicado que es una excepción de tipo *unchecked*, las cuales no es necesario capturar.

Las aserciones se han incluido como una característica del lenguaje Java, no como la integración o mejora a través de un API. Esto significa problemas de compatibilidad con las versiones del lenguaje anteriores al JDK 1.4. Lo mismo ocurre con los tipos enumerados, introducidos en el lenguaje Java a través de la palabra clave `enum` a partir del J2SE 5.

Por defecto, Java tiene deshabilitadas las aserciones. Para activarlas es necesario indicar expresamente lo que se desea hacer mediante los parámetros de invocación del intérprete `-enableassertions` (`-ea` en forma breve) y `-disableassertions` (`-da`). Estos parámetros permiten varias acepciones:

- Sin argumentos, para activar o desactivar todas las aserciones de la aplicación.
- Indicando un nombre de paquete, para activar o desactivar las aserciones de las clases que contiene ese paquete y sus paquetes dependientes.
- Indicando un nombre de clase, para activar o desactivar las aserciones de la clase que se indique.

Por ejemplo, los comandos utilizados en la ejecución de los dos últimos ejemplos, activan todas las aserciones incluidas en las aplicaciones. El comando siguiente:

```
java -ea:miPaquete.. -da:miPaquete.subpaquete MiAplicacion
```

habilitará las aserciones para el paquete `miPaquete` y para todos sus paquetes dependientes, excepto para el paquete `miPaquete.subpaquete`.

En resumen, se pueden incluir activaciones y desactivaciones, una tras otra, que se ejecuten en el orden que aparecen en la línea de comandos, permitiendo un control exhaustivo de las aserciones que se puedan lanzar. No obstante, salvo casos en que se trate de aplicaciones muy grandes, lo normal es activar todas las aserciones cuando se trata de encontrar o localizar errores o funcionamientos anómalos en esas aplicaciones.

CAPÍTULO 11

TAREAS Y MULTITAREA

Considerando el entorno *multithread* (multihilo, multitarea, multihilvanado), cada *thread* (hilo, tarea, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. A veces se les llama *procesos ligeros* o *contextos de ejecución*.

El autor recuerda al lector que cuando se refiere a ***tarea***, se refiere a un *hilo de ejecución*, a un ***thread*** y del mismo modo, cuando se refiere a ***multitarea*** se refiere a ***multithread***.

Generalmente, cada tarea controla un único aspecto dentro de un programa, como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todas las tareas comparten los mismos recursos, al contrario que los *procesos*, en donde cada uno tiene su propia copia de código y datos (separados unos de otros). Gráficamente, las tareas (*threads*) se parecen en su funcionamiento a lo que muestra la figura 11.1.

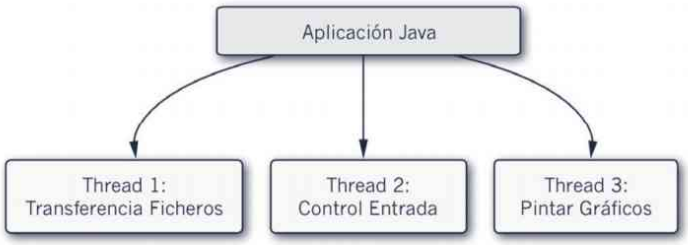


Figura 11.1

Hay que distinguir *multitarea* de *multiproceso*. El *multiproceso* se refiere a dos programas que se ejecutan "*aparentemente*", a la vez, bajo el control del Sistema Operativo. Los programas no necesitan tener relación entre sí, simplemente el hecho

de que el usuario desee que se ejecuten a la vez. *Multitarea* se refiere a que dos o más tareas se ejecutan "*aparentemente*" a la vez, dentro de un mismo programa.

Se usa "*aparentemente*" en ambos casos, porque normalmente las plataformas tienen una sola CPU, con lo cual, los procesos no se ejecutan en realidad "*concurrentemente*", sino que comparten la CPU. En plataformas con varias CPU, sí es posible que los procesos se ejecuten realmente a la vez.

Tanto en multiproceso como en multitarea (*multihilo*), el Sistema Operativo se encarga de que se genere la ilusión de que todo se ejecuta a la vez. Sin embargo, la multitarea puede producir programas que realicen más trabajo en la misma cantidad de tiempo que el multiproceso, debido a que la CPU está compartida entre tareas de un mismo proceso. Además, como el multiproceso está implementado a nivel de sistema operativo, el programador no puede intervenir en el planteamiento de su ejecución; mientras que en el caso de la multitarea, como el programa debe ser diseñado expresamente para que pueda soportar esta característica, es imprescindible que el desarrollador tenga que planificar adecuadamente la ejecución de cada tarea.

Actualmente hay diferencias en la especificación del intérprete de Java, porque el intérprete de *Windows* conmuta las tareas de igual prioridad mediante un algoritmo circular (*round-robin*), mientras que el de *Solaris* deja que una tarea ocupe la CPU indefinidamente, lo que implica la inanición de las demás.

PROGRAMAS DE FLUJO ÚNICO

Un programa de flujo único, tarea única o mono-hilo (*single-thread*) utiliza un único flujo de control (*thread*) para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples tareas. Sin necesidad de especificar explícitamente que se quiere un único flujo de control, muchos de los applets y aplicaciones son de flujo único.

Por ejemplo, en la aplicación estándar de saludo para aprender cualquier lenguaje de programación:

```
public class HolaMundo {
  static public void main( String args[] ) {
    System.out.println( "Hola Mundo!" );
    }
  }
```

Aquí, cuando se llama a *main()*, la aplicación imprime el mensaje y termina. Esto ocurre dentro de una única tarea (*thread*).

Debido a que la mayor parte de los entornos operativos no solían ofrecer un soporte razonable para múltiples tareas, los lenguajes de programación tradicionales, tales como C++, no incorporaron mecanismos para describir de manera elegante situaciones de este tipo. La sincronización entre las múltiples partes de un programa se

llevaba a cabo mediante un bucle de suceso único. Estos entornos son de tipo síncrono, gestionados por sucesos. Entornos tales como el de *MacOS* de Apple, *Windows* de Microsoft y *X11/Motif* fueron diseñados en torno al modelo de bucle de suceso.

PROGRAMAS DE FLUJO MÚLTIPLE

En la aplicación de saludo, no se ve la tarea que está ejecutando el programa. Sin embargo, Java posibilita la creación y control de tareas explícitamente. La utilización de tareas (*threads*) en Java permite una enorme flexibilidad a los programadores a la hora de plantearse el desarrollo de aplicaciones. La simplicidad para crear, configurar y ejecutar tareas permite que se puedan implementar muy poderosas y portables aplicaciones/applets que no se pueden crear con otros lenguajes de tercera generación. En un lenguaje orientado a Internet como es Java, esta herramienta es vital.

Si el lector ha utilizado un navegador con soporte Java, ya habrá visto el uso de múltiples tareas en Java. Habrá observado que dos applets se pueden ejecutar al mismo tiempo, o bien que puede desplazar la página del navegador mientras el applet continúa ejecutándose. Esto no significa que el applet utilice múltiples tareas, sino que el navegador es multitarea, multihilo, multihilvanado o *multithreaded*.

Los navegadores utilizan diferentes tareas ejecutándose en paralelo para realizar varias tareas, "*aparentemente*" de forma concurrente. Por ejemplo, en muchas páginas Web se puede *desplazar* la página e ir *leyendo* el texto antes de que todas las imágenes estén presentes en la pantalla. En este caso, el navegador está descargando las imágenes en una tarea y soportando el desplazamiento de la página en otra tarea diferente.

Las aplicaciones (y *applets*) multitarea utilizan muchos contextos de ejecución para cumplir su trabajo. Se aprovechan del hecho de que muchas tareas contienen subtareas distintas e independientes. Se puede utilizar un hilo de ejecución para cada subtarea.

Mientras que los programas de flujo único pueden realizar su tarea ejecutando las subtareas secuencialmente, un programa multitarea permite que cada tarea comience y termine tan pronto como sea posible. Este comportamiento presenta una mejor respuesta a la entrada en tiempo real.

A continuación, se va a modificar el programa de saludo creando tres tareas individuales, que imprimen cada una de ellas su propio mensaje de saludo, `MultiHola.java`:

```
class TestTh extends Thread {
  private String nombre;
  private int retardo;

  // Constructor para almacenar nuestro nombre y el retraso
```

```
  public TestTh( String s,int d ) {
    nombre = s;
    retardo = d;
    }

  // El método run() es similar al main(), pero para threads. Cuando
  // run() termina el thread muere
  public void run() {
    // Retasamos la ejecución el tiempo especificado
    try {
      sleep( retardo );
    } catch( InterruptedException e ) {
      ;
      }
    // Ahora imprimimos el nombre
    System.out.println( "Hola Mundo! "+nombre+" "+retardo );
    }
  }

public class MultiHola {
  public static void main( String args[] ) {
    TestTh t1,t2,t3;
    // Creamos los threads
    t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
    t2 = new TestTh( "Thread 2",(int)(Math.random()*2000) );
    t3 = new TestTh( "Thread 3",(int)(Math.random()*2000) );
    // Arrancamos los threads
    t1.start();
    t2.start();
    t3.start();
    }
  }
```

CREACIÓN Y CONTROL DE TAREAS

Antes de entrar en más profundidades en las tareas en Java, se propone al lector una referencia rápida de la clase **Thread**.

La clase Thread

Es la clase que encapsula todo el control necesario sobre las tareas o hilos de ejecución (*threads*). Hay que distinguir claramente un objeto **Thread** de una tarea o *thread*. Esta distinción resulta complicada, aunque se puede simplificar si se considera al objeto **Thread** como el panel de control de una tarea o hilo de ejecución (*thread*). La clase **Thread** es la única forma de controlar el comportamiento de las tareas y para ello se sirve de los métodos que se exponen en las secciones siguientes.

MÉTODOS DE CLASE

Corresponden a los métodos estáticos que deben invocarse de manera directa en la clase **Thread**.

`currentThread()`, este método devuelve el objeto **Thread** que representa a la tarea que se está ejecutando actualmente.

`yield()`, este método hace que el intérprete cambie de contexto entre la tarea actual y la siguiente tarea ejecutable disponible. Es una manera de asegurar que las tareas de menor prioridad no sufran inanición.

`sleep(long)`, el método *sleep()* provoca que el intérprete detenga la tarea en curso durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicha tarea volverá a estar disponible para su ejecución. Se advierte al lector de que los relojes asociados a la mayor parte de los intérpretes de Java no serán capaces de obtener precisiones mayores de 10 milisegundos, por mucho que se permita indicar hasta nanosegundos en la llamada alternativa a este método.

MÉTODOS DE INSTANCIA

Aquí no están recogidos todos los métodos de la clase **Thread**, sino sólo los más interesantes, porque los demás corresponden a áreas en donde el estándar de Java no está completo, y puede que se queden obsoletos cuando una nueva versión de Java vea la luz, por ello, para completar la información que aquí se expone, se ha de recurrir a la documentación de la interfaz de programación de aplicación (API) del JDK,

`start()`, este método indica al intérprete de Java que cree un contexto de la tarea del sistema y comience a ejecutarla. A continuación, el método *run()* de esta tarea será invocado en el nuevo contexto de la tarea. Hay que tener precaución de no llamar al método *start()* más de una vez sobre una tarea determinada.

`run()`, el método *run()* constituye el cuerpo de una tarea o hilo en ejecución. Éste es el único método de la interfaz **Runnable**. Es llamado por el método *start()* después de que la tarea apropiada del sistema se haya inicializado. Siempre que el método *run()* devuelva el control, la tarea actual se detendrá.

`stop()`, este método provoca que la tarea se detenga de manera inmediata. A menudo, constituye una manera brusca de detener una tarea, especialmente si este método se ejecuta sobre la tarea en curso. En tal caso, la línea inmediatamente posterior a la llamada al método *stop()* no llega a ejecutarse jamás, pues el contexto de la tarea *muere* antes de que *stop()* devuelva el control. Una forma más elegante de detener una tarea es utilizar alguna variable que ocasione que el método *run()* termine de manera ordenada. En realidad, nunca se debería recurrir al uso del método *stop()*.

`interrupt()`, este método también provoca la detención de la tarea, aunque solamente cuando el método que se está ejecutando pueda lanzar una excepción del tipo **InterruptedException**, que puede ser capturada y ser tratada para salir correctamente de la tarea o hilo de ejecución.

`suspend()`, es distinto de *stop()*. El método *suspend()* toma la tarea y hace que se detenga su ejecución sin destruir la tarea de sistema subyacente ni el estado de la

tarea anteriormente en ejecución. Si la ejecución de una tarea se suspende, podrá llamarse a *resume()* sobre esa tarea para lograr que vuelva a ejecutarse de nuevo.

`resume()`, el método *resume()* se utiliza para revivir una tarea suspendida. No hay garantías de que la tarea comience a ejecutarse inmediatamente, ya que puede haber una tarea de mayor prioridad en ejecución, pero *resume()* hace que la tarea vuelva a ser una candidata a ser ejecutada.

`setPriority(int)`, el método *setPriority()* asigna a la tarea la prioridad indicada por el valor pasado como parámetro. Hay bastantes constantes predefinidas en la clase **Thread** para establecer la prioridad, tales como `MIN_PRIORITY`, `NORM_PRIORITY` y `MAX_PRIORITY`, que toman los valores 1, 5 y 10, respectivamente. Como guía aproximada de utilización, se puede indicar que la mayor parte de los procesos a nivel de usuario deberían tomar una prioridad en torno a `NORM_PRIORITY`. Las tareas en segundo plano, como una entrada/salida a red o el nuevo dibujo de la pantalla, deberían tener una prioridad cercana a `MIN_PRIORITY`. Con las tareas a las que se fije la máxima prioridad, en torno a `MAX_PRIORITY`, hay que ser especialmente cuidadosos, porque si no se hacen llamadas a *sleep()* o *yield()*, se puede provocar que el intérprete Java quede totalmente fuera de control.

`getPriority()`, este método devuelve la prioridad de la tarea en curso, que es un valor comprendido entre uno y diez.

`setName(String)`, este método permite identificar la tarea con un nombre mnemónico. De esta manera se facilita la depuración de programas multitarea. El nombre mnemónico aparecerá en todas las líneas de trazado que se muestran cada vez que el intérprete Java imprime excepciones no capturadas.

`getName()`, este método devuelve el valor actual, de tipo cadena, asignado como nombre a la tarea en ejecución mediante *setName()*.

Creación de una tarea

Hay dos modos de conseguir tareas (*threads*) en Java. Una es implementando la interfaz **Runnable**, la otra es extender la clase **Thread**.

La implementación de la interfaz **Runnable** es la forma habitual de crear tareas. Las interfaces proporcionan al programador una forma de agrupar el trabajo de infraestructura de una clase. Se utilizan para diseñar los requisitos comunes al conjunto de clases que se implementan. La interfaz define el trabajo y la clase, o clases, que implementan la interfaz para realizar ese trabajo. Los diferentes grupos de clases que implementen la interfaz tendrán que seguir las mismas reglas de funcionamiento.

El primer método para crear una tarea o hilo de ejecución es simplemente extender la clase **Thread**:

```
class MiThread extends Thread {
  public void run() {
    . . .
    }
```

El ejemplo anterior crea una nueva clase **MiThread** que extiende la clase **Thread** y sobrescribe el método *Thread.run()* por su propia implementación. Con el método *run()* es con el que se realizará todo el trabajo de la clase. Extendiendo la clase **Thread**, se pueden heredar los métodos y variables de la clase padre. En este caso, solamente se puede extender o derivar una vez de la clase padre. Esta limitación de Java puede ser superada a través de la implementación de **Runnable**:

```
public class MiThread implements Runnable {
  Thread t;
  public void run() {
    // Ejecución del thread una vez creado
    }
  }
```

En este caso es necesario crear una instancia de **Thread** antes de que el sistema pueda ejecutar el proceso como una tarea. Además, el método abstracto *run()* está definido en la interfaz **Runnable** y tiene que ser implementado. La única diferencia entre los dos métodos es que este último es mucho más flexible. En el ejemplo anterior, todavía está la oportunidad de extender la clase **MiThread**, si fuese necesario. La mayoría de las clases creadas que necesiten ejecutarse como una tarea, implementarán la interfaz **Runnable**, ya que lo más probable es que necesiten extender alguna de sus funcionalidades a otras clases.

No piense el lector que la interfaz **Runnable** implica acciones en las clases que la implementan cuando la tarea se está ejecutando. Solamente contiene métodos abstractos, con lo cual proporciona una guía sobre el diseño de la clase **Thread**, y su misión concluye en ese punto. De hecho, si se observan los fuentes de Java, se puede comprobar que sólo contiene un método abstracto:

```
package java.lang;
public interface Runnable {
  public abstract void run() ;
  }
```

Y esto es todo lo que hay sobre la interfaz **Runnable**. Como se ha indicado en capítulos anteriores, una interfaz sólo proporciona un diseño para las clases que vayan a ser implementadas. En el caso de **Runnable**, fuerza a la definición del método *run()*, por lo tanto, la mayor parte del trabajo se hace en la clase **Thread**. Un vistazo un poco más profundo a la definición de la clase **Thread** da idea de lo que realmente está pasando:

```
public class Thread implements Runnable {
  ...
  public void run() {
    if( thread != null )
      thread.run() ;
```

```
      }
    }
  ...
  }
```

De estas pocas líneas de código se desprende que la clase **Thread** implementa la interfaz **Runnable**. El método *thread.run()* se asegura de que la clase con que trabaja (la clase que va a ejecutarse como tarea) no sea nula y ejecuta el método *run()* de esa clase. Cuando esto suceda, el método *run()* de la clase hará que corra como una tarea.

A continuación, se presenta el ejemplo `Java1101.java`, que implementa la interfaz **Runnable** para crear un programa multitarea.

```
class Java1101 {
  static public void main( String args[] ) {
    // Se instancia dos nuevos objetos Thread
    Thread hiloA = new Thread( new MiHilo(),"hiloA" );
    Thread hiloB = new Thread( new MiHilo(),"hiloB" );

    // Se arrancan las dos tareas, para que comiencen su ejecución
    hiloA.start();
    hiloB.start();
    // Aquí se retrasa la ejecución un segundo y se captura la posible
    // excepción que genera el método, aunque no se hace nada en el
    // caso de que se produzca
    try {
      Thread.currentThread().sleep( 1000 );
    } catch( InterruptedException e ){}
    // Presenta información acerca del Thread principal del programa
    System.out.println( Thread.currentThread() );
    }
  }

// Esta clase existe solamente para que sea heredada por la clase
// MiHilo, para evitar que esta clase sea capaz de heredar la clase
// Thread, y se pueda implementar la interfaz Runnable en su lugar
class NoHaceNada {}

class MiHilo extends NoHaceNada implements Runnable {
  public void run() {
    // Presenta en pantalla información sobre esta tarea en particular
    System.out.println( Thread.currentThread() );
    }
  }
```

El programa define la clase **MiHilo** que extiende a la clase **NoHaceNada** e implementa la interfaz **Runnable**. Se redefine el método *run()* en la clase **MiHilo** para presentar información sobre la tarea.

La única razón de extender la clase **NoHaceNada** es puramente ilustrativa, para proporcionar un ejemplo de situación en que haya que extender alguna otra clase, además de implementar la interfaz.

En el ejemplo Java1102.java, se muestra básicamente el mismo programa, pero en este caso extendiendo la clase **Thread**, en lugar de implementar la interfaz **Runnable** para crear el programa multitarea.

```
class Java1102 {
  . . .
class MiHilo extends Thread {
  public void run() {
    // Presenta en pantalla información sobre este hilo en particular
    System.out.println( Thread.currentThread() );
    }
  }
```

En este caso, la nueva clase **MiHilo** extiende la clase **Thread** y no implementa la interfaz **Runnable** directamente (la clase **Thread** implementa la interfaz **Runnable**, por lo que indirectamente **MiHilo** también está implementando esa interfaz). El resto del programa es similar al anterior.

Y todavía se puede presentar un ejemplo más simple, utilizando un constructor de la clase **Thread** que no necesita parámetros, tal como se presenta en el ejemplo Java1103.java. En los ejemplos anteriores, el constructor utilizado para **Thread** necesitaba dos parámetros, el primero un objeto de cualquier clase que implemente la interfaz **Runnable** y el segundo una cadena que indica el nombre de la tarea (este nombre es independiente del nombre de la variable que referencia al objeto **Thread**).

```
class Java1103 {
  static public void main( String args[] ) {
    // Se instancia dos nuevos objetos Thread
    Thread hiloA = new MiHilo();
    Thread hiloB = new MiHilo();
  . . .
  }
```

Las sentencias en este ejemplo para instanciar objetos **Thread**, son mucho menos complejas, siendo el programa, en esencia, el mismo de los ejemplos anteriores.

Arranque de una tarea

Las aplicaciones ejecutan *main()* tras arrancar. Ésta es la razón de que *main()* sea el lugar natural para crear y arrancar otras tareas. La línea de código

```
t1 = new TestTh( "Thread 1",(int)(Math.random()*2000) );
```

crea una nueva tarea o hilo de ejecución. Los dos argumentos pasados representan el nombre de la tarea y el tiempo de espera antes de imprimir el mensaje.

Al tener control directo sobre las tareas, hay que arrancarlas explícitamente. En el ejemplo con:

```
t1.start();
```

start(), en realidad es un método oculto en la tarea que llama a *run()*.

Manipulación de una tarea

Si todo fue bien en la creación de la tarea, `t1` debería contener un objeto de tipo **Thread** válido, que se controlará en el método *run()*.

Una vez dentro de *run()*, se pueden comenzar las sentencias de ejecución como en otros programas. El método *run()* sirve como rutina *main()* para las tareas; cuando *run()* termina, también lo hace la tarea. Todo lo que se quiera que haga la tarea ha de estar dentro de *run()*, por eso cuando se dice que un método es **Runnable**, es obligatorio escribir un método *run()*.

En este ejemplo se intenta inmediatamente esperar durante una cantidad de tiempo aleatoria (pasada como parámetro a través del constructor):

```
sleep( retardo );
```

El método *sleep()* simplemente le dice a la tarea que *duerma* durante los milisegundos especificados. Se debe utilizar *sleep()* cuando se pretenda retrasar la ejecución de la tarea. El método *sleep()* no consume recursos del sistema mientras la tarea duerme. De esta forma otras tareas pueden seguir funcionando. Una vez hecho el retraso, se imprime el mensaje "Hola Mundo!" con el nombre de la tarea y el retardo.

Suspensión de una tarea

Puede resultar útil suspender la ejecución de una tarea sin marcar un límite de tiempo. Si, por ejemplo, está construyendo un applet con una tarea de animación, seguramente se querrá permitir al usuario la opción de detener la animación hasta que quiera continuar. No se trata de terminar la animación, sino desactivarla. Para este tipo de control de las tareas se puede utilizar el método *suspend()*.

```
t1.suspend();
```

Este método no detiene la ejecución permanentemente. La tarea es suspendida indefinidamente, y para volver a activarla de nuevo se necesita realizar una invocación al método *resume()*:

```
t1.resume();
```

Parada de una tarea

El último elemento de control que se necesita sobre las tareas o hilos de ejecución es el método *stop()*. Se utiliza para terminar la ejecución de una tarea:

```
t1.stop();
```

Esta llamada no destruye la tarea, sino que detiene su ejecución. La ejecución no se puede reanudar ya con *t1.start()*. Cuando se desasignen las variables que se usan en la tarea, el objeto **Thread** (creado con new) quedará marcado para eliminarlo y el *garbage collector* se encargará de liberar la memoria que utilizaba.

En el ejemplo, no se necesita detener explícitamente la tarea, simplemente se la deja terminar. Los programas más complejos necesitarán un control sobre cada una de las tareas que lancen, el método *stop()* puede utilizarse en esas situaciones, aunque hay que tener en cuenta que en la ejecución normal de una aplicación, no puede saberse de forma exacta en qué punto se encuentra la tarea en el momento de invocarlo, y esto puede originar que haya recursos que no se liberen adecuadamente; por ejemplo, si se está ejecutando una sección `finalize` y se invoca a *stop()*, la ejecución de todas las sentencias que componen esa sección `finalize` no se llevará a cabo, por lo que se quedarán recursos sin liberar.

Si se necesita, se podrá comprobar si una tarea está viva o no; considerando *viva* una tarea que ha comenzado y no ha sido detenida.

```
t1.isAlive();
```

Este método devolverá *true* en caso de que la tarea `t1` esté activa, es decir, ya se haya llamado a su método *run()* y no haya sido parada con un *stop()* ni haya terminado el método *run()* en su ejecución.

En el ejemplo no hay problemas de realizar una parada incondicional, al estar todas las tareas activas. Pero si a una tarea, que puede no estar activa, se le invoca su método *stop()*, se generará una excepción. En este caso, en los que el estado de la tarea no puede conocerse de antemano, es donde se requiere el uso del método *isAlive()*.

GRUPOS DE TAREAS

Toda tarea o hilo de ejecución en Java debe formar parte de un *grupo*. La clase **ThreadGroup** define e implementa la capacidad de un grupo de tareas.

Los grupos de tareas permiten que sea posible agrupar varias tareas en un solo objeto y manipularlo como un grupo en vez de individualmente. Por ejemplo, se pueden regenerar las tareas de un grupo mediante una sola sentencia.

Cuando se crea una nueva tarea, se coloca en un grupo, bien indicándolo explícitamente, o bien dejando que el sistema la coloque en el grupo por defecto. Una vez creada la tarea y asignada a un grupo, ya no se podrá cambiar a otro grupo.

Si no se especifica un grupo en el constructor, el sistema colocará la tarea en el mismo grupo en que se encuentre la tarea que la haya creado, y si no se especifica el grupo para ninguna de las tareas, entonces todas serán miembros del grupo "*main*", que es creado por el sistema cuando arranca la aplicación Java.

En la ejecución de los ejemplos de este capítulo, se ha podido observar la circunstancia anterior. Por ejemplo, el resultado en pantalla de uno de esos ejemplos es el que se reproduce a continuación:

```
% java Java1102
Thread[hiloA,5,main]
Thread[hiloB,5,main]
Thread[main,5,main]
```

Como resultado de la ejecución de sentencias del tipo

```
System.out.println( Thread.currentThread() );
```

se obtiene información sobre la tarea. Se puede observar que aparece el nombre de la tarea, su prioridad y el nombre del grupo en que se encuentra englobada.

La clase **Thread** proporciona constructores en los que se puede especificar el grupo de la tarea que se está creando en el mismo momento de instanciarla, y también métodos como *setThreadGroup()*, que permiten determinar el grupo en que se encuentra una tarea.

ARRANCAR Y PARAR TAREAS

Ahora que se ha visto por encima cómo se arrancan, paran, manipulan y agrupan las tareas, se presenta un ejemplo un poco más gráfico, se trata de un contador, cuyo código, `Java1104.java`, es el siguiente:

```
public class Java1104 extends Applet implements Runnable {
  Thread t;
  int contador;

  // Creamos el thread y lo arrancamos
  public void init() {
    ProcesoRaton procesoRaton = new ProcesoRaton();
    addMouseListener( procesoRaton );
    contador = 0;
    t = new Thread( this );
    t.start();
    }

  // Corazón del applet, incrementa el contador, lo pinta en la
  // pantalla y tiene su tiempo de espera, tanto para incrementar de
  // nuevo el contador como para dejar tiempo a la CPU para que
  // atienda a otros applets o aplicaciones que pudiesen convivir
  public void run() {
    Thread miThread = Thread.currentThread();
    while( t == miThread ) {
      contador++;
      repaint();
      // Forzosamente tenemos que capturar esta interrupción
      try {
        miThread.sleep( 10 );
      } catch( InterruptedException e ) {}
      }
```

```
    }

  // Actualizamos un contador en la ventana del applet y otro en la
  // consola
  public void paint( Graphics g ) {
    g.drawString( Integer.toString( contador ),10,10 );
    System.out.println( "Contador= "+contador );
    }

  // Paramos el applet, pero sin llamar al método stop(), por el
  // peligro de caer en un punto de no retorno
  public void stop() {
    t = null;
    }

  // Cuando se pulsa el ratón dentro del dominio del applet se detiene
  // la ejecución. Ésta es una Clase Anidada
  class ProcesoRaton extends MouseAdapter {
    public void mousePressed( MouseEvent evt ) {
      t.stop();
      }
    }
  }
```

Este applet arranca un contador en 0 y lo incrementa, presentando su salida en la pantalla gráfica y en la consola. Una primera ojeada al código puede dar la impresión de que el programa empezará a contar y presentará cada número, pero no es así. Una revisión más profunda del flujo de ejecución del applet revelará su verdadera identidad.

En este caso, la clase **Java1104** está forzada a implementar **Runnable** sobre la clase **Applet** que extiende. Como en todos los applets, el método *init()* es el primero que se ejecuta. En *init()*, la variable `contador` se inicializa a cero y se crea una nueva instancia de la clase **Thread**. Pasándole `this` al constructor de **Thread**, la nueva tarea podrá conocer al objeto que va a correr. En este caso `this` es una referencia a **Java1104**. Después de que se haya creado la tarea, es necesario arrancarla. La llamada a *start()*, llamará a su vez al método *run()* de la clase, es decir, a *Java1104.run()*. La llamada a *start()* retornará con éxito y la tarea comenzará a ejecutarse en ese instante. Observe el lector que el método *run()* es un bucle infinito. Es infinito porque una vez que se sale de él, la ejecución de la tarea se detiene. En este método se incrementa la variable `contador`, se *duerme* 10 milisegundos y envía una petición de refresco del nuevo valor al applet.

Es muy importante utilizar *sleep()* en algún lugar de la tarea, porque si no, esa tarea consumirá todo el tiempo de la CPU para su proceso y no permitirá que entren métodos de otras tareas a ejecutarse. En el `contador`, la tarea se detiene cuando se pulsa el ratón mientras el cursor se encuentre sobre el applet. Dependiendo de la velocidad del ordenador, se presentarán los números consecutivos o no, porque el incremento de la variable `contador` es independiente del refresco en pantalla. El applet no se refresca a cada petición que se le hace, sino que el sistema operativo encolará las peticiones y las que sean sucesivas las convertirá en un único refresco.

Así, mientras los refrescos se van encolando, la variable `contador` se estará todavía incrementando, pero no se visualiza en pantalla.

El uso y la conveniencia de utilización del método *stop()*, como ya se indicó anteriormente, es un poco dudoso y algo que debería evitarse, porque puede haber objetos que dependan de la ejecución de varias tareas, y si se detiene una de ellas, puede que el objeto en cuestión estuviese en un estado no demasiado consistente, y en el caso de que se mate la tarea de control es probable que ese objeto definitivamente se dañe. Una solución alternativa es el uso de una variable de control que permita saber si la tarea se encuentra en ejecución o no, por ello, en el ejemplo se utiliza la variable `miThread` que controla cuándo la tarea está en ejecución o parada.

La clase anidada **ProcesoRaton** es la que se encarga de implementar un objeto receptor de los eventos de ratón, para detectar cuándo el usuario pulsa alguno de los botones sobre la zona de influencia del applet.

SUSPENDER Y REANUDAR TAREAS

Una vez que se para una tarea invocando al método *stop()*, ya no se puede rearrancar con el comando *start()*, debido a que *stop()* concluirá la ejecución de esa tarea. Por ello, en vez de parar la tarea, lo que se puede hacer es *dormirla*, llamando al método *sleep()*. La tarea estará suspendida un cierto tiempo y luego reanudará su ejecución cuando el límite fijado se alcance. Pero esto no es útil cuando se necesite que la tarea reanude su ejecución ante la presencia de ciertos eventos. En estos casos, el método *suspend()* permite que cese la ejecución de la tarea y el método *resume()* permite que un método suspendido reanude su ejecución. En la siguiente versión modificada del ejemplo anterior, `Java1105.java`, se modifica el applet para que utilice los métodos *suspend()* y *resume()*.

El uso de *suspend()* es critico en ocasiones, sobre todo cuando la tarea que se va a suspender está utilizando recursos del sistema, porque en el momento de la suspensión los va a bloquear, y esos recursos seguirán bloqueados hasta que no se reanude la ejecución de la tarea con *resume()*. Por ello, deben utilizarse métodos alternativos a éstos, por ejemplo, implementando el uso de variables de control que vigilen periódicamente el estado en que se encuentra la tarea actual y obren en consecuencia.

```
public class Java1105 extends Applet implements Runnable {
    ...
class ProcesoRaton extends MouseAdapter {
  boolean suspendido;
  public void mousePressed( MouseEvent evt ) {
    if( suspendido )
      t.resume();
    else
      t.suspend();
    suspendido = !suspendido;
    }
  }
```

...

Para controlar el estado del applet, se ha modificado el funcionamiento del objeto **Listener** que recibe los eventos del ratón, en donde se ha introducido la variable `suspendido`. Diferenciar los distintos estados de ejecución del applet resulta importante porque algunos métodos pueden generar excepciones si son llamados desde un estado erróneo. Por ejemplo, si el applet ha sido arrancado y se detiene con *stop()*, y en ese momento se intenta ejecutar el método *start()*, se generará una excepción **IllegalThreadStateException**.

Aquí se puede poner de nuevo en cuarentena la idoneidad del uso de estos métodos para el control del estado de la tarea, tanto por lo comentado del posible bloqueo de recursos vitales del sistema, como porque se puede generar un punto muerto en el sistema si la tarea que va a intentar revivir la tarea suspendida necesita del recurso bloqueado. Por ello, es más seguro el uso de una variable de control como `suspendido`, de tal forma que sea ella quien controle el estado de la tarea y utilizar el método *notify()* para indicar cuándo la tarea vuelve a la vida.

ESTADOS DE UNA TAREA

Durante el ciclo de vida de una tarea, ésta se puede encontrar en diferentes estados. La figura 11.2 muestra estos estados y los métodos que provocan el paso de un estado a otro. Este diagrama no es una máquina de estados finita, pero es lo que más se aproxima al funcionamiento real de una tarea.

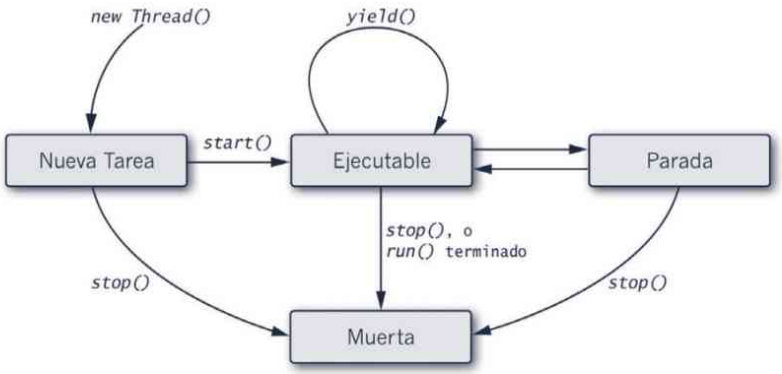


Figura 11.2

Nueva tarea

La siguiente sentencia crea una nueva tarea pero no la arranca, la deja en el estado de *Nueva Tarea*:

```
Thread MiThread = new MiClaseThread();
Thread MiThread = new Thread( new UnaClaseThread,"hiloA" );
```

Cuando una tarea está en este estado, es simplemente un objeto **Thread** vacío. El sistema no ha destinado ningún recurso para él. Desde este estado solamente puede

arrancarse llamando al método *start()*, o detenerse definitivamente, llamando al método *stop()*; la llamada a cualquier otro método carece de sentido y lo único que provocará será la generación de una excepción de tipo **IllegalThreadStateException**.

Ejecutable

Ahora observe el lector las dos líneas de código que se presentan a continuación:

```
Thread MiThread = new MiClaseThread();
MiThread.start();
```

La llamada al método *start()* creará los recursos del sistema necesarios para que la tarea pueda ejecutarse, la incorpora a la lista de procesos disponibles para ejecución del sistema y llama al método *run()* de la tarea. En este momento se encuentra en el estado *Ejecutable* del diagrama. Y este estado es *Ejecutable* y no *En Ejecución*, porque cuando la tarea está aquí no está corriendo. Muchos ordenadores tienen solamente un procesador, lo que hace imposible que todas las tareas estén corriendo al mismo tiempo. Java implementa un tipo de *scheduling* o lista de procesos que permite que el procesador sea compartido entre todos los procesos o tareas que se encuentran en la lista. Sin embargo, para el propósito que aquí se persigue, y en la mayoría de los casos, se puede considerar que este estado es realmente un estado *En Ejecución*, porque la impresión que produce ante el usuario es que todos los procesos se ejecutan al mismo tiempo.

Cuando la tarea se encuentra en este estado, todas las instrucciones de código que se encuentren dentro del bloque declarado para el método *run()*, se ejecutarán secuencialmente.

Parada

La tarea entra en estado *Parada* cuando alguien llama al método *suspend()*, cuando se llama al método *sleep()*, cuando la tarea está bloqueada en un proceso de entrada/salida o cuando la tarea utiliza su método *wait()* para esperar a que se cumpla una determinada condición. Cuando ocurra cualquiera de las cuatro cosas anteriores, la tarea estará en estado *Parada*.

Por ejemplo, en el trozo de código siguiente

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
  MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
  e.printStackTrace();
  }
```

la llamada al método *sleep()* hace que la tarea se *duerma* durante 10 segundos. Durante ese tiempo, incluso aunque el procesador estuviese totalmente libre,

`MiThread` no correría. Después de esos 10 segundos. `MiThread` volvería a estar en estado *Ejecutable* y ahora sí que el procesador podría hacerle caso cuando se encuentre disponible.

Para cada uno de los cuatro modos de entrada en estado *Parada*, hay una forma específica de volver a estado *Ejecutable*. Cada forma de recuperar ese estado es exclusiva; por ejemplo, si la tarea ha sido puesta a *dormir*, una vez transcurridos los milisegundos que se especifiquen, ella sola se despierta y vuelve a estar en estado *Ejecutable*. Llamar al método *resume()* mientras esté la tarea durmiendo no serviría para nada.

Los métodos de recuperación del estado *Ejecutable*, en función de la forma de llegar al estado *Parada* de la tarea, son los siguientes:

- Si una tarea está dormida, pasado el lapso de tiempo.
- Si una tarea está suspendida, después de una llamada a su método *resume()*.
- Si una tarea está bloqueada en una entrada/salida, una vez que el comando de entrada/salida concluya su ejecución.
- Si una tarea está esperando por una condición, cada vez que la variable que controla esa condición varíe debe llamarse al método *notify()* o *notifyAll()*.

Muerta

Una tarea se puede *morir* de dos formas: por causas naturales o porque la maten. Una tarea muere normalmente cuando concluye de forma habitual su método *run()*. Por ejemplo, en el siguiente trozo de código, el bucle `while` es un bucle finito, ya que realiza la iteración 20 veces y termina:

```
public void run() {
  int i=0;
  while( i < 20 ) {
    i++;
    System.out.println( "i = "+i );
    }
  }
```

Una tarea que contenga a este método *run()*, morirá naturalmente después de que se complete el bucle y *run()* concluya.

También se puede matar en cualquier momento una tarea, invocando a su método *stop()*. En el trozo de código siguiente

```
Thread MiThread = new MiClaseThread();
MiThread.start();
try {
  MiThread.sleep( 10000 );
} catch( InterruptedException e ) {
  e.printStackTrace();
```

```
  }
MiThread.stop();
```

se crea y arranca la tarea `MiThread`, se duerme durante 10 segundos y en el momento de despertarse, la llamada a su método *stop()*, la mata.

El método *stop()* envía un objeto **ThreadDeath** a la tarea que quiere detener. Así, cuando una tarea es parada de este modo, muere asíncronamente. La tarea morirá en el mismo instante en que reciba ese objeto **ThreadDeath**.

Los applets utilizarán el método *stop()* para matar a todas sus tareas cuando el navegador con soporte Java en el que se están ejecutando le indica al applet que se detengan, por ejemplo, cuando se minimiza la ventana del navegador o cuando se cambia de página.

Hay que recordar que en la plataforma Java 2 el método *stop()* de la clase **Thread** ya no está implementado como tal, sino que debe ser implementado por el programador en cada una de las subclases que cree, preferiblemente para el control de una variable booleana que indique si la tarea está corriendo o parada.

SCHEDULING

Java tiene un **Scheduler**, una lista de procesos, que monitoriza todas las tareas que se están ejecutando en todos los programas y decide cuáles deben ejecutarse y cuáles deben encontrarse preparadas para su ejecución. Hay dos características de las tareas que el *scheduler* identifica en este proceso de decisión. Una, la más importante, es la prioridad de la tarea; la otra, es el indicador de *demonio*. La regla básica del *scheduler* es que si solamente hay *tareas demonio* ejecutándose, la *Máquina Virtual Java* (JVM) concluirá. Las nuevas tareas heredan la prioridad y el indicador de demonio de las tareas que los han creado. El *scheduler* determina qué tareas deberán ejecutarse comprobando la prioridad de todas ellas, aquéllas con prioridad más alta dispondrán del procesador antes de las que tienen prioridad más baja.

El *scheduler* puede seguir dos patrones, *preemptivo* y *no-preemptivo*. Los *schedulers preemptivos* proporcionan un segmento de tiempo a todas las tareas que están corriendo en el sistema. El *scheduler* decide cuál será la siguiente tarea a ejecutarse y llama al método *resume()* para darle vida durante un período fijo de tiempo. Cuando la tarea ha estado en ejecución ese período de tiempo, se llama a *suspend()* y la siguiente tarea en la lista de procesos será relanzada (con *resume()*). Los *schedulers no-preemptivos* deciden la tarea que debe correr y la ejecutan hasta que concluye. La tarea tiene control total sobre el sistema mientras esté en ejecución. El método *yield()* es la forma en que una tarea fuerza al *scheduler* a comenzar la ejecución de otra tarea que esté esperando. Dependiendo del sistema en que esté corriendo Java, el *scheduler* será de un tipo u otro, preemptivo o no-preemptivo.

Prioridades

El *scheduler* determina la tarea que debe ejecutarse en función de la prioridad asignada a cada una de ellas. El rango de prioridades oscila entre 1 y 10. La prioridad por defecto de una tarea es `NORM_PRIORITY`, que tiene asignado un valor de 5. Hay otras dos variables estáticas disponibles, que son `MIN_PRIORITY`, fijada a 1, y `MAX_PRIORITY`, que tiene un valor de 10. El método *getPriority()* puede utilizarse para conocer el valor actual de la prioridad de una tarea.

Tareas demonio

Las tareas *demonio* también se llaman *servicios*, porque se ejecutan, normalmente, con prioridad baja y proporcionan un servicio básico a un programa o programas cuando la actividad de la máquina es reducida.

Son útiles cuando una tarea debe ejecutarse en segundo plano durante largos períodos de tiempo. Un ejemplo de *tarea demonio* que está en continua ejecución es el recolector de basura (*garbage collector*). Esta tarea, proporcionada por la Máquina Virtual Java, comprueba las variables de los programas a las que no se accede nunca y libera estos recursos, devolviéndolos al sistema.

Una tarea puede fijar su indicador de demonio pasando un valor `true` al método *setDaemon()*. Si se pasa `false` a este método, la tarea será devuelta por el sistema como una tarea de usuario. No obstante, esto último debe realizarse antes de que se arranque la tarea (mediante la llamada al método *start()*). Si se quiere saber si una tarea es una tarea demonio, se utilizará el método *isDaemon()*.

COMUNICACIONES ENTRE TAREAS

Otra clave del éxito y la ventaja de la utilización de múltiples tareas en una aplicación, o aplicación *multithreaded*, es que pueden comunicarse entre sí. Se pueden diseñar tareas para utilizar objetos comunes, que cada tarea puede manipular independientemente de las otras tareas.

Como se ha indicado en párrafos anteriores, una de las formas de controlar el estado de una tarea es mediante el uso de condiciones de estado, invocando al método *wait()* para que la tarea se inactive hasta que la condición establecida varíe y se invoque al método *notify()* para reanudar la ejecución de la tarea. La utilización de estas condiciones de control resulta útil en muchas circunstancias, por ejemplo, cuando una tarea no quiere concluir, sino solamente esperar hasta que un objeto determinado se desbloquee. Y, concretamente, los mensajes, eventos, pulsaciones de teclado, peticiones web, etc., son ejemplos en los que las condiciones de control constituyen la base de su funcionamiento.

El ejemplo clásico de comunicación entre tareas es un modelo de tipo productor/consumidor, como el ya implementado en capítulos anteriores en base a Colas. En este caso se implementa mediante un buffer y un monitor que actuará de controlador de las peticiones que se realicen entre productor y consumidor. Una tarea produce una salida, que otra tarea usa (consume), sea lo que sea esa salida. Por ejemplo, se crea un *productor*, que será una tarea que irá sacando caracteres por su salida; y se crea también un *consumidor* que irá recogiendo los caracteres que vaya sacando el productor y un *monitor* que controlará el proceso de sincronización entre las tareas. Funcionará como una tubería, insertando el productor caracteres en un extremo y leyéndolos el consumidor en el otro, con el monitor siendo la propia tubería.



Figura 11.3

Productor

El productor extenderá la clase **Thread**, y su código es el siguiente:

```
class Productor extends Thread {
  private Tuberia tuberia;
  private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  public Productor( Tuberia t ) {
    // Mantiene una copia propia del objeto compartido
    tuberia = t;
    }

  public void run() {
    char c;
    // Mete 10 letras en la tubería
    for( int i=0; i < 10; i++ ) {
      c = alfabeto.charAt( (int)(Math.random()*26 ) );
      tuberia.lanzar( c );
      // Imprime un registro con lo añadido
      System.out.println( "Lanzado "+c+" a la tuberia." );
      // Espera un poco antes de añadir más letras
      try {
        sleep( (int)(Math.random() * 100 ) );
      } catch( InterruptedException e ) {
        e.printStackTrace();
        }
      }
    }
  }
```

Obsérvese que se crea una instancia de la clase **Tuberia**, y que se utiliza el método *tuberia.lanzar()* para que se vaya construyendo la tubería, en principio de 10 caracteres.

Consumidor

Ahora se reproduce el código del consumidor, que extiende la clase **Thread**:

```
class Consumidor extends Thread {
  private Tuberia tuberia;

  public Consumidor( Tuberia t ) {
    // Mantiene una copia propia del objeto compartido
    tuberia = t;
    }

  public void run() {
    char c;
    // Consume 10 letras de la tubería
    for( int i=0; i < 10; i++ ) {
      c = tuberia.recuperar();
      // Imprime las letras retiradas
      System.out.println( "Recogido el caracter "+c );
      // Espera un poco antes de capturar más letras
      try {
        sleep( (int)(Math.random() * 2000 ) );
      } catch( InterruptedException e ) {
        e.printStackTrace();
        }
      }
    }
  }
```

En este caso, como en el del productor, se cuenta con un método en la clase **Tuberia**, *tuberia.recuperar()*, para manejar la información.

Monitor

Una vez vistos el productor de la información y el consumidor, lo que realiza la clase **Tuberia**, es una función de supervisión de las transacciones entre las dos tareas, el productor y el consumidor. Los monitores, en general, son piezas muy importantes de las aplicaciones multitarea, porque mantienen el flujo de comunicación entre las tareas.

```
class Tuberia {
  private char buffer[] = new char[6];
  private int siguiente = 0;
  // Flags para saber el estado del buffer
  private boolean estaLlena = false;
  private boolean estaVacia = true;
  // Método para retirar letras del buffer
  public synchronized char recuperar() {
    // No se puede consumir si el buffer está vacío
```

```
    while( estaVacia == true ) {
      try {
        wait(); // Se sale cuando estaVacia cambia a false
      } catch( InterruptedException e ) {
        e.printStackTrace();
        }
      }
    // Decrementa la cuenta, ya que va a consumir una letra
    siguiente--;
    // Comprueba si se retiró la última letra
    if( siguiente == 0 )
      estaVacia = true;
    // El buffer no puede estar lleno, porque acabamos de consumir
    estaLlena = false;
    notify();
    // Devuelve la letra al thread consumidor
    return( buffer[siguiente] );
    }

  // Método para añadir letras al buffer
  public synchronized void lanzar( char c ) {
    // Espera hasta que haya sitio para otra letra
    while( estaLlena == true ) {
      try {
        wait(); // Se sale cuando estaLlena cambia a false
      } catch( InterruptedException e ) {
        e.printStackTrace();
      }
    }
    // Añade una letra en el primer lugar disponible
    buffer[siguiente] = c;
    // Cambia al siguiente lugar disponible
    siguiente++;
    // Comprueba si el buffer está lleno
    if( siguiente == 6 )
      estaLlena = true;
    estaVacia = false;
      notify();
    }
  }
```

En la clase **Tuberia** se pueden observar dos características importantes: los miembros dato (`buffer[]`) son privados, y los métodos de acceso (*lanzar()* y *recuperar()*) son sincronizados.

Aquí se observa que la variable `estaVacia` es un semáforo. La naturaleza privada de los datos evita que el productor y el consumidor accedan directamente a éstos. Si se permitiese el acceso directo de ambas tareas a los datos, se podrían producir problemas; por ejemplo, si el consumidor intenta retirar datos de un buffer vacío, obtendrá excepciones innecesarias, o bien se bloqueará el proceso.

Los métodos de acceso sincronizado impiden que los productores y consumidores corrompan un objeto compartido. Mientras el productor está añadiendo una letra a la tubería, el consumidor no la puede retirar, y viceversa. Esta sincronización es vital

para mantener la integridad de cualquier objeto compartido. No sería lo mismo sincronizar la clase en vez de los métodos, porque esto significaría que nadie puede acceder a las variables de la clase en paralelo, mientras que al sincronizar los métodos, sí pueden acceder a todas las variables que están fuera de los métodos que pertenecen a la clase.

Se pueden sincronizar incluso variables, para realizar alguna acción determinada sobre ellas, por ejemplo:

```
sincronized( p ) {
  // Aquí se colocaría el código. Los threads que estén intentando
  // acceder a p se pararán y generarán una InterruptedException
  }
```

El método *notify()* al final de cada método de acceso avisa a cualquier proceso que esté esperando por el objeto, entonces el proceso que ha estado esperando intentará acceder de nuevo al objeto. En el método *wait()* se hace que la tarea se quede a la espera de que le llegue un *notify()*, ya sea enviado por la tarea en ejecución o por el sistema.

Ahora que se dispone de un productor, un consumidor y un objeto compartido, se necesita una aplicación que arranque las tareas y que consiga que todos hablen con el mismo objeto que están compartiendo. Esto es lo que hace el siguiente trozo de código, del fuente `Java1106.java`:

```
class Java1106 {
  public static void main( String args[] ) {
    Tuberia t = new Tuberia();
    Productor p = new Productor( t );
    Consumidor c = new Consumidor( t );
    p.start();
    c.start();
    }
  }
```

Compilando y ejecutando esta aplicación, se podrá observar el modelo que se ha diseñado en pleno funcionamiento.

Monitorización del productor

Los programas productor/consumidor a menudo emplean monitorización remota, que permite al consumidor observar la tarea del productor interaccionando con un usuario o con otra parte del sistema. Por ejemplo, en una red, un grupo de tareas productores podrían trabajar cada uno en una estación de trabajo. Los productores imprimirían documentos, almacenando una entrada en un registro (*log*). Un consumidor (o múltiples consumidores) podría procesar el registro y realizar durante la noche un informe de la actividad de impresión del día anterior.

Otro ejemplo a pequeña escala podría ser el uso de varias ventanas en una estación de trabajo. Una ventana se puede usar para la entrada de información (el productor), y otra ventana reaccionaría a esa información (el consumidor).

Peer es un observador general del sistema.

UTILIDADES DE CONCURRENCIA

El uso de `synchronize` y la pareja de métodos *wait()* y *notify()*, no son suficientes para el control exhaustivo de la ejecución de tareas. Por ello, desde la versión J2SE 5, Java dispone de una librería de utilidades que proporcionan potentes construcciones de alto nivel para ejecutores, colas, temporizadores, bloqueos y otras primitivas de sincronización.

Los ***semáforos***, por ejemplo, pueden ser utilizados de la misma forma que el método *wait()* descrito anteriormente, pero para restringir el acceso a un bloque de código. Los semáforos son más flexibles y pueden permitir el acceso concurrente de varias tareas, a la vez que permiten que las tareas puedan conocer su estado antes de pedir el bloqueo.

La clase **PoolRecursos** proporciona un *pool* de recursos, que aunque para efectos de simplicidad de la aplicación, son simples objetos de tipo **Integer**, podría tratarse en aplicaciones reales de conexiones JDBC, por ejemplo.

```
public class PoolRecursos {
  // Tamaño del pool de recursos
  private int tamPool;
  private Semaphore semaforo;
  // Creamos dos arrays. Uno que contiene los elementos que constituyen
  // los recursos del pool y otro que indica si cada uno de esos
  // recursos está siendo utilizado o no
  private Integer[] recursos;
  private boolean[] utilizado;

  // Constructor en el que creamos el pool de recursos
  public PoolRecursos( int tamPool ) {
    this.tamPool = tamPool;
    // Creamos un semáforo para controlar el acceso a los recursos
    semaforo = new Semaphore( tamPool );
    // Creamos el array de indicadores de uso de los recursos
    utilizado = new boolean[tamPool];
    // Creamos el array de recursos y le asignamos valores a cada uno
    recursos = new Integer[tamPool];
    for( int i=0; i < tamPool; i++ )
      recursos[i] = new Integer(i);
    }

  // Este método devuelve un recurso. Si todos los recursos están
  // siendo utilizados, se producirá el bloqueo hasta que alguno de
  // ellos se libere. Es un método sincronizado para que el código sea
  // totalmente "thread safe"
```

```
public synchronized Integer adquiereRecurso() {
  try {
    // Solicitamos el semáforo
    semaforo.acquire();
  } catch( InterruptedException e ) {
    System.out.println( e.getLocalizedMessage() );
    }
  // Buscamos un recurso libre
  for( int i=0; i < tamPool; i++ ) {
    if( utilizado[i] == false ) {
      // Si el método no fuese sincronizado, varias tareas podrían
      // alcanzar este punto y se les devolvería el mismo recurso a
      // cada una de ellas
      utilizado[i] = true;
      return( recursos[i] );
      }
    }
  return( null );
  }

// Método utilizado para devolver un recurso al pool
public void liberaRecurso( Integer recurso ) {
  // Utilizamos auto-unboxing
  utilizado[recurso] = false;
  semaforo.release();
  }
}
```

El constructor de la clase crea un semáforo indicando el número de recursos disponibles, en el caso del ejemplo, dos.

Cuando alguien necesita uno de los recursos, invoca al método *adquiereRecurso()* que llama al método de la clase **Semaphore** para adquirirlo. Si el contador del semáforo es menor que el utilizado en el constructor, el método continuará. Si todos los recursos están en uso, el método bloqueará el acceso hasta que alguno de ellos quede disponible.

Una de las características más útiles de los semáforos es que el método para adquirir el recurso y el método para devolverlo, pueden estar en la misma clase y ser accedidos concurrentemente por tareas distintas.

La aplicación `Java1107.java` crea tres tareas que intentan acceder a los recursos del *pool*, utilizando un semáforo para restringir el acceso a esos recursos. En la ejecución se demuestra que una de las tres tareas siempre queda bloqueada cuando intenta acceder al recurso, porque solamente hay dos disponibles.

Otro de los elementos que incorpora la librería de concurrencia es la interfaz **Executor**, que permite iniciar la ejecución de una tarea separada de su aplicación, en lugar de invocar al método *start()* de la clase **Thread**.

Para crear una instancia de la clase **ThreadPoolExecutor**, que es una de las implementaciones disponibles de la interfaz **Executor**, se utiliza la invocación al

método *newFixedThreadPool()*, indicando el número de tareas que constituyen el *pool*. También se puede conseguir creando un objeto de tipo **Queue** para almacenar la información de las tareas e invocar al constructor **ThreadPoolExecutor**. El primer método es más simple y es el que se ha utilizado en el ejemplo.

Si solamente se quiere crear una única tarea, las implementaciones de **Executor** también deben proporcionar el método *newSingleThreadExecutor()* para permitir esta posibilidad.

La aplicación `Java1108.java` crea una clase que actúa como un registro de eventos sobre un puerto determinado, presentando en pantalla los caracteres que recibe cuando se establece una conexión a ese puerto. Como es necesario controlar muchas conexiones simultáneamente que duran muy poco tiempo, es imprescindible tener una tarea separada para cada conexión.

Como la creación de objetos **Thread** es costosa para la Máquina Virtual Java, el mejor modo de conseguirlo es utilizar un pool de tareas que pueda ser reutilizado por las nuevas conexiones cuando las anteriores vayan finalizando.

Ejecutar la aplicación en una ventana de comandos del sistema operativo. En esa ventana aparecerá un mensaje indicando que la aplicación queda a la espera de que se establezcan conexiones con ella.

La clase **Eco** es una versión simplificada de telnet, envía los caracteres que se tecleen a un socket. La secuencia de caracteres tecleada aparece en pantalla y se envía en el momento de pulsar la tecla *Retorno*. Ejecutar en una ventana de comandos diferente esta aplicación, teclear una frase y enviarla pulsando la tecla *Retorno*.

En la ventana de comandos aparecerán los mensajes de conexión con el servidor, los correspondientes al uso de la aplicación **Eco** y la frase tecleada. En la ventana de ejecución del servidor **Java1108** se mostrará la frase enviada por **Eco**.

En una ventana de comandos diferente de las anteriores, volver a ejecutar la aplicación **Eco** y repetir las acciones indicadas en el párrafo anterior. El resultado obtenido es semejante, porque el pool de conexiones definido en la clase **Java1108** es dos.

Sin embargo, las cosas no son iguales si se ejecuta de nuevo por tercera vez la aplicación **Eco** en otra ventana de comandos diferente de las anteriores. Aparecerá el mensaje de que la conexión con el servidor se ha establecido, a pesar de que en la ventana del servidor no hay indicación alguna de que dicha conexión se haya iniciado. Es más, si se teclea una frase en esta ventana y se envía al servidor pulsando *Retorno*, en la ventana del servidor no ocurre nada.

Ahora, en una de las dos primeras ventanas de ejecución de **Eco**, teclear `SALIR` y pulsar *Retorno*. La ejecución concluirá y en la ventana del servidor se indicará dicha

circunstancia, pero además, se iniciará la conexión con la aplicación **Eco** que estaba detenida y se mostrará la frase que se había introducido en la tercera ventana de **Eco**.

En la aplicación anterior se ha mostrado el uso de **Executor**, que evita la utilización del método *start()*, a favor de *execute()*. Otra interfaz que proporciona una alternativa, en este caso al método *run()*, es **Callable**. La interfaz **Callable** define el método *call()* y proporciona una forma de obtener un resultado o capturar una excepción desde una tarea separada.

Los objetos **Callable** son enviados a un **Executor**, pero no ejecutados. El método *submit()* devuelve un objeto de tipo **Future**, que es quien proporciona el método *get()* para recuperar un resultado. A la hora de recuperar este resultado, si ya está listo en la tarea se devuelve, y en caso contrario, la tarea se bloqueará hasta que esté disponible.

Para mostrar el funcionamiento de **Callable** y **Future**, se crea la clase **CallableImp**, que implementa la interfaz **Callable** y es capaz de devolver un objeto de tipo **String** cuando es invocada por una tarea.

```
public class CallableImp implements Callable<String> {
  // Punto de entrada de la llamada que es invocado cuando un nuevo
  // hilo entra en ejecución
  public String call() {
    System.out.println(
      "[2] Invocacion de call() en la segunda tarea.." );
    try {
      Thread.sleep( 500 );
    } catch( InterruptedException e ) {
      System.out.println( e.getLocalizedMessage() );
    }
    System.out.println(
      "[2] Completada la llamada a call() en la segunda tarea.." );
    return( "Concluido" );
    }
  }
```

La clase **Java1109** utiliza la clase **CallableImp** para simular acciones en una tarea separada mientras concluye las suyas propias. En este caso tan simple no se realiza acción alguna, solamente se espera un lapso de tiempo invocando al método *sleep()*.

```
public class Java1109 {
  // Ejecución de la prueba
  public void prueba() {
    // Utilizamos los métodos de utilidad de Executor para recuperar
    // un objeto ExecutorService desde un hilo de ejecución separado
    ExecutorService es = Executors.newSingleThreadExecutor();
    System.out.println( "[1] Iniciando prueba en la primera tarea..");
    // Ahora lanzamos la ejecución de la segunda tarea,
    // correspondiente ala implementación de Callable en este caso
    try {
      Future<String> f = es.submit( new CallableImp() );
      // Fijamos el intervalo a 1000, para poder ver los efectos de la
```

```
      // sincronización
      Thread.sleep( 1000 );
      System.out.println(
        "[1] Primera tarea completada. Consultando a Future..");
      String salida = f.get();
      System.out.println( "El resultado desde Future es " +salida );
    } catch( InterruptedException e ) {
      System.out.println( e.getMessage() );
    } catch( ExecutionException e ) {
      System.out.printf( "Error en la ejecucion de la prueba: %s",
        e.getLocalizedMessage() );
      }
    // Cortamos la segunda tarea para que el programa concluya
    es.shutdown();
    }

  public static void main( String[] args ) {
    Java1109 ej = new Java1109();
    ej.prueba();
    }
  }
```

Ejecutando la aplicación, aparecerán mensajes indicando lo que ocurre con las dos tareas en lo que respecta a su inicio y funcionamiento. El lector puede experimentar modificando los valores de *sleep()* en las dos tareas para que la principal se complete más rápidamente, lo cual cambiaría el orden de los mensajes.

Observe el lector que tanto en un caso como en otro, o en cualquier otra circunstancia, las dos tareas permanecerán siempre correctamente sincronizadas.

Para finalizar esta sección, indicar que Java, desde el J2SE 5, dispone de la interfaz **BlockingQueue** para el control de las acciones entre tareas. En el ejemplo `Java1110.java`, se implementa un sistema de bitácora, de registro o de log, basado en una Cola de este tipo para sincronizar la grabación de mensajes procedentes de distintas tareas.

El código de la clase **Logs** que se reproduce a continuación, toma como argumento en el constructor un objeto de tipo **BlockingQueue** para utilizarlo como almacén de los mensajes que las distintas tareas quieran registrar. Esta cola asegura que las operaciones son seguras entre todas las tareas que intenten añadir elementos a la lista de mensajes.

```
public class Logs implements Runnable {
  // Cola que se utilizará para pasar los mensajes entre dos tareas,
  // la que genera los mensajes y ésta que los imprime
  private BlockingQueue<String> cola;

  public Logs(BlockingQueue<String> cola) {
    this.cola = cola;
    }

  // Este método obtiene los mensajes de la cola y los envía a la
  // salida estándar
```

```
public void run() {
  try {
    while( true )
      System.out.printf( "Mensaje: %s%n",cola.take() );
  } catch( InterruptedException e ) {
    System.out.println( e.getLocalizedMessage() );
    }
  }
}
```

La clase **Java1110** crea una Cola de tipo **ArrayBlockingQueue**, que es la implementación más simple de la interfaz **BlockingQueue**. Utiliza este objeto para enviar mensajes a la instancia de **Logs**. Cuando se ejecuta la aplicación, se crean varias instancias de la clase con identificadores diferentes, que se encargan de enviar mensajes. Se definen intervalos de pausa entre los mensajes de cada tarea, para que no envíen todas a la vez y, se crean en base a **ThreadPoolExecutor**, ya visto en párrafos anteriores.

Cuando se compila y ejecuta el fichero `Java1110.java`, la salida de la aplicación no es muy impresionante, pero en ella se puede observar que las tareas van incorporando mensajes sucesivamente sin problemas de concurrencia entre ellas, al estar bajo el control de **BlockingQueue**.

CAPÍTULO 12

DELEGACIÓN DE EVENTOS

El lenguaje Java originalmente manejaba los eventos siguiendo el modelo de *Propagación*, o *Herencia*, pero desde el JDK 1.1 *Sun Microsystems* ha implantado el modelo de *Delegación*.

De acuerdo con Sun, en el **AWT**, que es la librería de clases básicas para el desarrollo de interfaces de usuario, y en *Swing*, que es la librería de componentes de interfaz de usuario que solamente soporta el modelo de *delegación* de eventos, las principales características de partida que han originado el diseño e implantación del modelo de *Delegación* para el manejo de eventos, son:

- Que sea simple y fácil de aprender.
- Que soporte una clara separación entre el código de la aplicación y el código de la interfaz.
- Que facilite la creación de robustos controladores de eventos, con menos posibilidad de generación de errores (chequeo más potente en tiempo de compilación).
- Suficientemente flexible para permitir el flujo y propagación de eventos.
- Para herramientas visuales, permitir en tiempo de ejecución ver cómo se generan estos eventos y quién lo hace.
- Que soporte compatibilidad binaria con el modelo anterior.

A continuación, se muestra una revisión por encima del modelo de *Delegación*, antes de entrar en el estudio detallado, y también se expone un ejemplo sencillo para poder entender de forma más fácil ese estudio detallado del modelo de Delegación de eventos.

Los eventos están organizados en jerarquías de clases de eventos. El modelo de *Delegación* hace uso de *fuentes* de eventos (**Source**) y *receptores* de eventos (**Listener**). Una fuente de eventos es un objeto que tiene la capacidad de detectar eventos y notificar a los receptores de eventos que se han producido esos eventos. Aunque el programador puede establecer el entorno en que se producen esas notificaciones, siempre hay un escenario por defecto.

Un objeto receptor de eventos es una clase (o una subclase de una clase) que implementa una interfaz receptora específica. Hay definido un determinado número de interfaces receptoras, donde cada interfaz declara los métodos adecuados al tratamiento de los eventos de su clase. Luego, hay un emparejamiento natural entre clases de eventos y definiciones de interfaces. Por ejemplo, hay una clase de eventos de ratón que incluye muchos de los eventos asociados con las acciones del ratón, y hay una interfaz que se utiliza para definir los receptores de esos eventos.

Un objeto receptor puede estar registrado con un objeto fuente para ser notificado de la ocurrencia de todos los eventos de la clase para los que el objeto receptor está diseñado. Una vez que el objeto receptor está registrado para ser notificado de esos eventos, el suceso de un evento en esta clase automáticamente invocará al método sobrescrito del objeto receptor. El código en el método sobrescrito debe estar diseñado por el programador para realizar las acciones específicas que desee cuando suceda el evento.

Algunas clases de eventos, como los de ratón, involucran a un determinado conjunto de eventos diferentes. Una clase *Receptor* (**Listener**) que implemente la interfaz que recoja estos eventos debe sobrescribir todos los métodos declarados en la interfaz. Para prevenir esto, de forma que no sea tan tedioso y no haya que sobrescribir métodos que no se van a utilizar, se han definido un conjunto de clases intermedias, conocidas como *Adaptadores* (**Adapter**).

Estas clases Adaptadores implementan las interfaces receptoras y sobrescriben todos los métodos de la interfaz con métodos vacíos. Una clase Receptor puede estar definida como clase que extiende una clase **Adapter** en lugar de una clase que implemente la interfaz. Cuando se hace esto, la clase Receptor solamente necesita sobrescribir aquellos métodos que sean de interés para la aplicación, porque todos los otros métodos serán resueltos por la clase **Adapter**. Por ejemplo, en el programa que se muestra a continuación, `Java1201.java`, los dos objetos Receptor instanciados desde dos clases diferentes, están registrados para recibir todos los eventos involucrados en la manipulación de un objeto de tipo **Frame** (apertura, cierre, minimización, etc.).

Uno de los objetos Receptor implementa la interfaz **WindowListener**, por lo que debe sobrescribir los seis métodos de la interfaz. La otra clase Receptor extiende la clase **WindowAdapter** en vez de implementar la interfaz **WindowListener**. La clase **WindowAdapter** sobrescribe los seis métodos de la interfaz con métodos vacíos, por lo que la clase receptor no necesita sobrescribir esos seis métodos.

El ejemplo es tan sencillo que los métodos sobrescritos solamente presentan un mensaje en pantalla indicando cuándo han sido invocados.

Una de las cosas en las que debe reparar el lector es en la forma en que los objetos receptores son registrados para la notificación de los eventos; lo cual se hace en las sentencias del ejemplo que se reproducen:

```
ventana.addWindowListener( ventanaProceso1 );
ventana.addWindowListener( ventanaProceso2 );
```

La interpretación de este fragmento de código es que dos objetos receptores llamados `ventanaProceso1` y `ventanaProceso2` se añaden a la lista de objetos receptores que serán automáticamente notificados cuando se produzca un evento de la clase **Window** con respecto al objeto **Frame** llamado `ventana`.

Estos objetos receptores son notificados invocando los métodos sobrescritos de los objetos que capturan el tipo específico de evento (apertura de la ventana, minimización de la ventana, cierre de la ventana, etc.).

Los párrafos que siguen introducirán al lector en una visión más detallada del modelo de Delegación de Eventos.

En el modelo de *Delegación* los eventos se encapsulan en una jerarquía de clases donde la clase raíz es **EventObject**; en la plataforma Java 2 bajo esta misma clase se encapsulan las clases de los eventos de otros APIs como son **BeanContextEvent**, **DragSourceEvent**, **DropTargetEvent** y **PropertyChangeEvent**. La propagación de un evento desde un objeto Fuente hasta un objeto Receptor involucra la llamada a un método en el objeto receptor y el paso de una instancia de una subclase de eventos (un objeto) que define el tipo de evento generado. Cada subclase de eventos puede incluir más de un tipo de eventos.

Un objeto receptor es una instancia de una clase que implementa una interfaz específica **EventListener** extendida desde el receptor genérico **EventListener**. Una interfaz **EventListener** define uno o más métodos que deben ser invocados por la fuente de eventos en respuesta a cada tipo de evento controlado por la interfaz.

La invocación de estos métodos redefinidos es el mecanismo por el cual el objeto Fuente notifica al Receptor que uno o más eventos han sucedido. El objeto fuente mantiene una lista de objetos receptores y los tipos de eventos a los que están suscritos. El programador crea esa lista utilizando llamadas a los métodos *add<TipoEvento>Listener()*.

Una vez que la lista de receptores está creada, el objeto fuente utiliza esta lista para notificar a cada receptor que ha sucedido un evento del tipo que controla, sin esfuerzo alguno por parte del programador. Esto es lo que se conoce como *registrar* Receptores específicos para recibir la notificación de eventos determinados.

La *fuente de eventos* es generalmente un componente de la interfaz gráfica. Un *receptor de eventos* es normalmente un objeto de una clase que implementa la adecuada interfaz del receptor. El objeto receptor también puede ser otro componente del AWT que implementa una o más interfaces receptoras, con el propósito de comunicar unos objetos de la interfaz gráfica con otros.

Los eventos en el modelo de *Delegación* constituyen cada uno de ellos un miembro específico de la clase de tipos de eventos y son estas clases las que constituyen la jerarquía completa.

Como una sola clase de evento se puede utilizar para representar más de un tipo de evento, algunas clases pueden contar con un identificador (único para cada clase) que designa a cada uno de los eventos específicos; por ejemplo, **MouseEvent** representa el movimiento del cursor, la pulsación de un botón, el arrastre del ratón, el soltar un botón, etc.

No hay campos públicos en las clases. En su lugar, los datos del evento están encapsulados y solamente se puede acceder a ellos a través de los adecuados métodos *set..()* y *get..()*; los primeros sólo existen para modificar atributos de un evento y sólo pueden ser utilizados por un receptor.

El AWT define un conjunto determinado de eventos, aunque el programador también puede definir sus propios tipos de eventos, derivando de **EventObject**, o desde una de las clases de eventos del AWT.

El AWT proporciona dos tipos conceptuales de eventos: de *bajo nivel* y *semánticos*, que se introducen a continuación, aunque luego se verán con detalle.

Un evento de bajo nivel es aquel que representa una entrada de bajo nivel o un suceso sobre un componente visual de un sistema de ventanas sobre la pantalla. Eventos de este tipo son:

`Java.util.EventObject`	
`java.awt.AWTEvent`	
`java.awt.event.ComponentEvent`	Componente redimensionado, desplazado
`java.awt.event.FocusEvent`	Pérdida, ganancia del foco por un componente
`java.awt.event.InputEvent`	
`java.awt.event.KeyEvent`	El componente captura una pulsación de teclado
`java.awt.event.MouseEvent`	El componente captura movimientos del ratón, pulsación de botones
`java.awt.event.ContainerEvent`	
`java.awt.event.WindowEvent`	

Como ya se ha indicado, algunas clases de eventos engloban a varios tipos distintos de eventos. Normalmente, hay una interfaz correspondiente a cada clase de evento y hay métodos de la interfaz para cada tipo distinto de evento en cada clase de evento.

Un evento semántico es un evento que se define a alto nivel y encapsula una acción de un componente de la interfaz de usuario. Algunos eventos de este tipo son:

```
Java.util.EventObject
  java.awt.AWTEvent
    java.awt.event.ActionEvent          Ejecución de un comando
    java.awt.event.AdjustmentEvent      Ajuste de un valor
    java.awt.event.ItemEvent            Cambio de estado de un ítem
    java.awt.event.TextEvent            Cambio de valor de un texto
```

Estos eventos no están pensados para atender a componentes específicos de la pantalla, sino para aplicarlos a un conjunto de eventos que implementen un modelo semántico similar. Por ejemplo, un objeto **Button** generará un evento **Action** cuando sea pulsado y un objeto **List** generará un evento **Action** cuando se pulse dos veces con el ratón sobre uno de los elementos que componen la lista.

En el caso concreto de *eventos de teclado*, la plataforma Java 2 proporciona capacidades para diferenciar las teclas, o incluso diferenciar entre diferentes versiones de una misma tecla, por ejemplo una tecla normal y su correspondiente del teclado numérico; también permite diferenciar secuencias de teclas y botones de ratón. Por ejemplo, hasta ahora no se podía diferenciar entre pulsar la tecla de mayúsculas y pulsar un botón del ratón, porque todos ellos producían el mismo evento. A partir del JDK 1.4, se incorporaron métodos para detectar el botón del ratón que se ha pulsado y se añadieron constantes a la clase **InputEvent** para estos menesteres.

También se incluye el método *setButton()* en la clase **MouseEvent** para conocer el botón que ha cambiado de estado y proporciona el método *getKeyLocation()* en la clase **KeyEvent** para poder distinguir entre las diferentes versiones de una misma tecla, así se definen las siguientes constantes:

```
KEY_LOCATION_LEFT                KEY_LOCATION_STANDARD
KEY_LOCATION_NUMPAD              KEY_LOCATION_UNKNOWN
KEY_LOCATION_RIGHT
```

El ejemplo `Java1202.java` es un pequeño compendio de las posibilidades incorporadas a la plataforma Java 2 en el manejo de eventos. El programa fija el tamaño de la ventana al máximo, utiliza las nuevas constantes para definir colores y luego permite cambiar el color de fondo de la ventana mediante la rueda superior del ratón, o mediante las teclas de mayúsculas situadas a derecha e izquierda en el teclado.

RECEPTORES DE EVENTOS

Una interfaz **EventListener** tendrá un método específico para cada tipo de evento distinto que trate la clase de evento. Por ejemplo, la interfaz **FocusEventListener** define los métodos *focusGained()* y *focusLost()*, uno para cada tipo de evento que trata la clase **FocusEvent**.

Las interfaces de bajo nivel definidas por la plataforma Java 2 son las siguientes:

```
Java.util.EventListener
  java.awt.event.ComponentListener
  java.awt.event.ContainerListener
  java.awt.event.FocusListener
  java.awt.event.KeyListener
  java.awt.event.MouseListener
  java.awt.event.MouseMotionListener
  java.awt.event.MouseWheelListener
  java.awt.event.WindowListener
```

Si se compara esta lista con la lista anterior de clases de eventos de bajo nivel, se verá claramente que hay definida una interfaz receptora por cada una de las clases más bajas en jerarquía de las clases de eventos, excepto para la clase **MouseEvent** en que hay dos interfaces receptoras diferentes.

Las interfaces de nivel semántico que define el AWT en la plataforma Java 2 son:

```
Java.util.EventListener
  java.awt.event.ActionListener
  java.awt.event.AdjustmentListener
  java.awt.event.ItemListener
  java.awt.event.TextListener
```

La correspondencia aquí entre interfaces de nivel semántico y clases evento de nivel semántico es una a una.

Como los receptores se registran para manejar tipos de eventos determinados, solamente serán notificados de esos tipos de eventos y no llegarán a ellos otros tipos de eventos para los que no estén registrados. Esto es justamente lo contrario a lo que se utilizaba en el modelo de Propagación, en donde todos los eventos se pasaban a un controlador de eventos, fuesen de su interés o no. Este filtrado de eventos mejora el rendimiento, especialmente con los eventos que se producen con mucha frecuencia, como son los de movimiento del ratón.

FUENTES DE EVENTOS

Todas las fuentes de eventos del AWT soportan el multienvío a receptores. Esto significa que se pueden añadir o quitar múltiples receptores de una sola fuente; es

decir, la notificación de que se ha producido un mismo evento se puede enviar a uno o más objetos receptores simultáneamente.

El API de Java no garantiza el orden en que se enviarán los eventos a los receptores que están registrados en un objeto fuente, para ser informados de esos eventos. En caso de que el orden en que se distribuyen los eventos sea un factor importante en el programa, se deberían encadenar los receptores de un solo objeto receptor registrado sobre el objeto fuente; el hecho de que los datos del evento estén encapsulados en un solo objeto hace que la propagación del evento sea extremadamente simple.

Como en el caso de los receptores, se puede hacer una distinción entre los eventos de bajo nivel y los eventos de tipo semántico. Las fuentes de eventos de bajo nivel serán las clases de elementos o componentes visuales de la interfaz gráfica (botones, barras de desplazamiento, cajas de selección, etc.), porque cada componente de la pantalla generará sus eventos específicos. La plataforma Java 2 permite registrar receptores sobre fuentes de eventos de los siguientes tipos:

```
Java.awt.Component
  addComponentListener
  addFocusListener
  addKeyListener
  addMouseListener
  addMouseMotionListener
  addMouseWheelListener
java.awt.Container
  addContainerListener
java.awt.Dialog
  addWindowListener
java.awt.Frame
  addWindowListener
```

Para determinar todos los tipos de eventos que se pueden comunicar desde un objeto fuente a un receptor, hay que tener en cuenta la herencia. Por ejemplo, un objeto puede detectar eventos del ratón sobre un objeto **Frame** y notificar a un objeto **MouseListener** la ocurrencia de estos eventos, aunque en la lista anterior no se muestre un **MouseListener** sobre un **Frame**. Esto es posible porque un objeto **Frame** extiende indirectamente la clase **Component** y **MouseListener** está definido en la clase **Component**.

Los receptores de eventos que se pueden registrar de tipo semántico sobre objetos fuentes, generadores de eventos, en la plataforma Java 2 son:

```
Java.awt.Button
  addActionListener
Java.awt.Choice
  addItemListener
Java.awt.Checkbox
  addItemListener
Java.awt.CheckboxMenuItem
  addItemListener
Java.awt.List
  addActionListener
  addItemListener
java.awt.MenuItem
  addActionListener
java.awt.Scrollbar
  addAdjustmentListener
java.awt.TextArea
  addTextListener
java.awt.TextField
  addActionListener
  addTextListener
```

ADAPTADORES

Muchas interfaces **EventListener** están diseñadas para recibir múltiples clases de eventos, por ejemplo, la interfaz **MouseListener** puede recibir los eventos de pulsación de un botón, al soltar el botón, a la recepción del cursor, etc. La interfaz declara un método para cada uno de estos subtipos. Cuando se implementa una interfaz, es necesario redefinir todos los métodos que se declaran en esa interfaz, incluso aunque se haga con métodos vacíos. En la mayoría de las ocasiones, no es necesario redefinir todos los métodos declarados en la interfaz porque no son útiles para la aplicación.

Por ello, el AWT proporciona un conjunto de clases abstractas adaptadores (**Adapter**) que coinciden con las interfaces. Cada clase Adaptador implementa una interfaz y redefine todos los métodos declarados por la interfaz con métodos vacíos, con lo cual se satisface el requisito de la redefinición de todos los métodos.

Se pueden definir clases Receptor *extendiendo* clases Adaptadores, en vez de *implementar* la interfaz receptora correspondiente. Esto proporciona libertad al programador para redefinir solamente aquellos métodos de la interfaz que intervienen en la aplicación que desarrolla.

De nuevo, hay que recordar que todos los métodos declarados en una interfaz corresponden a los tipos de eventos individuales de la clase de eventos correspondiente, y que el objeto **Fuente** notifica al **Receptor** la ocurrencia de un evento de un tipo determinado invocando al método redefinido de la interfaz.

Las clases **Adaptadores** que se definen en la plataforma Java 2 son las que se indican a continuación:

```
java.awt.ComponentAdapter
java.awt.FocusAdapter
java.awt.KeyAdapter
java.awt.MouseAdapter
java.awt.MouseMotionAdapter
java.awt.WindowAdapter
```



Figura 12.1

En el ejemplo `Java1203.java`, se modifica el primer programa de este capítulo, en que la ejecución no terminaba cuando se cerraba la ventana; ahora el programa

termina cuando el usuario cierra la ventana, ejecutando la sentencia de salida en el controlador de eventos adecuado. El programa implementa un objeto **EventSource** que notifica a un objeto **Listener** la ocurrencia de un evento en la clase **Window**, y notifica a otro objeto **Listener** la ocurrencia de un evento en la clase **Mouse**.

Si se compila y ejecuta el ejemplo, cada vez que se pulse el botón del ratón con la flecha del cursor dentro de la ventana, aparecerán las coordenadas en las que se encuentra el cursor, tal como muestra la figura 12.1.

En el caso más simple, los eventos de bajo nivel del modelo de Delegación de Eventos, se pueden controlar siguiendo los pasos que se indican a continuación:

- Definir una clase **Listener**, Receptor, para una determinada clase de evento que implemente la interfaz receptora que coincida con la clase de evento, o extender la clase adaptadora correspondiente.
- Redefinir los métodos de la interfaz receptora para cada tipo de evento específico de la clase evento, para poder implementar la respuesta deseada del programa ante la ocurrencia de un evento. Si se implementa la interfaz receptora, hay que redefinir todos los métodos de la interfaz. Si se extiende la clase adaptadora, se pueden redefinir solamente aquellos métodos que son de interés.
- Definir una clase **Source**, fuente, que instancie un objeto de la clase Receptor y registrarla para la notificación de la ocurrencia de eventos generados por cada componente específico.

Por ejemplo, esto se consigue utilizando código como:

```
objetoVentana.addMouseListener( procesoRaton );
```

en donde

`objetoVentana`, es el objeto que genera el evento
`procesoRaton`, es el nombre del objeto receptor del evento, y
`addMouseListener`, es el método que registra el objeto receptor para recibir eventos de ratón desde el objeto llamado `objetoVentana`

Esta sentencia hará que el objeto `procesoRaton` sea notificado de todos los eventos que se produzcan sobre el `objetoVentana` que formen parte de la clase de eventos del ratón. La notificación tendrá lugar invocando al método redefinido en el objeto `procesoRaton` que corresponda con cada tipo específico de evento en la clase de eventos de ratón, aunque algunos de estos métodos pueden estar vacíos porque no interese tratar los eventos a que corresponden.

Todo lo anterior en el caso más simple, porque es posible complicar la situación a gusto, por ejemplo, si se quiere notificar a dos objetos receptor diferentes la ocurrencia de un determinado evento sobre un mismo objeto de la pantalla, tal como muestra el

código del ejemplo `Java1204.java`, en donde un objeto receptor es compartido por dos componentes visuales diferentes del mismo tipo. El programa detecta los eventos de ratón sobre dos objetos **Frame** diferentes, que distingue en base a su nombre, y presenta las coordenadas del cursor en cada pulsación sobre el objeto en que se encontraba el cursor.

El código del ejemplo es realmente simple. La única parte críptica es la que trata de obtener el nombre del componente visual que ha generado el evento *mousePressed()*.

El método *main()* instancia un objeto de tipo **IHM**, que sirve para dos propósitos, por un lado proporcionar la interfaz visual y, por otro, actuar como una fuente de eventos que notificará su ocurrencia a los objetos receptor que se registren con él.

La clase **Frame** es extendida en una nueva clase llamada **MiFrame**, para permitir la redefinición del método *paint()* de la clase; lo cual es imprescindible para presentar las coordenadas en donde se encuentra el cursor sobre el **Frame**, utilizando el método *drawString()*.

El constructor de la clase **IHM** instancia dos objetos de tipo **MiFrame** y los hace visibles. Cuando son instanciados, se les asignan los nombres `Frame1` y `Frame2` a través del método *setName()*. Estos nombres serán los que permitan determinar posteriormente cuál ha sido el objeto que ha generado el evento.

También en ese mismo constructor se instancia un solo objeto receptor a través del cual se procesarán todos los eventos de bajo nivel que se produzcan en cualquiera de los dos objetos visuales:

```
ProcesoRaton procesoRaton = new ProcesoRaton( miFrame1,miFrame2 );
miFrame1.addMouseListener( procesoRaton );
miFrame2.addMouseListener( procesoRaton );
```

La primera sentencia solamente instancia el nuevo objeto receptor `procesoRaton`, pasándole las referencias de los dos elementos visuales como parámetro. Las dos sentencias siguientes incorporan este objeto receptor (lo registran) a la lista de objetos receptor que serán automáticamente notificados cuando suceda cualquier evento de ratón sobre los objetos visuales referenciados como `miFrame1` y `miFrame2`, respectivamente. Y no se requiere ningún código extra para que se produzca esa notificación.

Las notificaciones se realizan invocando métodos de instancia específicos redefinidos del objeto receptor ante la ocurrencia de tipos determinados de eventos del ratón. Las declaraciones de todos los métodos deben coincidir con todos los posibles eventos del ratón que estén definidos en la interfaz **MouseListener**, que deben coincidir con los definidos en la clase **MouseEvent**. La clase desde la que el objeto

receptor es instanciado debe redefinir, bien directa o indirectamente, todos los métodos declarados en la interfaz **MouseListener**.

Además de registrar el objeto **MouseListener** para recibir objetos de ratón, el programa también instancia y registra un objeto receptor que monitoriza los eventos de la ventana, y termina la ejecución del programa cuando el usuario cierra uno cualquiera de los objetos visuales.

```
Proceso1 procesoVentana1 = new Proceso1();
miFrame1.addWindowListener( procesoVentana1 );
miFrame2.addWindowListener( procesoVentana1 );
```

La parte más complicada de la programación involucra al objeto receptor del ratón, y aun así, es bastante sencilla. Lo que intenta el código es determinar cuál de los dos elementos visuales ha sido el que ha generado el evento. En este caso el objeto receptor solamente trata eventos *mousePressed*, aunque lo que se explica a continuación se podría aplicar a todos los eventos del ratón y a muchos de los eventos de bajo nivel.

La clase **ProcesoRaton** (receptor) en este programa extiende la clase **MouseAdapter** y redefine el método *mousePressed()* que está declarado en la interfaz **MouseListener**. Cuando es invocado el método *mousePressed()*, se le pasa un objeto de tipo **MouseEvent**, llamado `evt`, como parámetro.

Para determinar si el objeto que ha generado el evento ha sido `Frame1`, se utiliza la siguiente sentencia:

```
if( evt.getComponent().getName().compareTo( "Frame1" ) == 0 ) {
```

El método *getComponent()* es aquel que devuelve el objeto donde se ha generado el evento. En este caso devuelve un objeto de tipo **Component** sobre el cual actúa el método *getName()*. Este último método devuelve el nombre del componente como un objeto **String**, sobre el cual actúa el método *compareTo()*. Este método, ya visto en capítulos anteriores, es estándar de la clase **String** y se utiliza para comparar dos objetos **String**. En este caso se utiliza para comparar el nombre del componente con la cadena "Frame1"; si coincide, el código que se ejecuta es el que presenta las coordenadas del ratón sobre el objeto visual `Frame1`; si no coincide se ejecuta el código de la cláusula `else` que presentará las coordenadas sobre el objeto visual `Frame2`.

También es posible realizar la comparación directamente sobre el objeto **MouseEvent**, tal como se verá más adelante. Por ahora, baste hacer ver al lector que el paquete **java.awt.event** es diferente del paquete **java.awt**.

Aunque en el ejemplo anterior se utilizan dos objetos visuales del mismo tipo, no hay razón alguna para que eso sea así, ya que todos los objetos visuales comparten el

mismo objeto receptor y son capaces de generar eventos para los que el receptor esté registrado.

También se incorporan eventos de ratón para detectar movimientos de la rueda superior que incorporan la mayoría de estos dispositivos. Por lo tanto, es posible incorporar un receptor de eventos de tipo **MouseWheelListener** a cualquier componente para que éste reaccione.

El receptor dispone solamente del método *mouseWheelMoved()*, que captura un evento de tipo **MouseWheelEvent**. Entre la información que proporciona el evento se puede obtener la cantidad de desplazamiento mediante *getScrollAmount()*, si se está desplazando una unidad o un bloque con *getScrollType()*, la dirección y el número de vueltas de la rueda con *getScrollRotation()* y, por conveniencia, el número de unidades a desplazar mediante *getUnitsScroll()*.

EVENTOS DE BAJO NIVEL Y SEMÁNTICOS

Aunque el conjunto de eventos semánticos es utilizado para propósitos diferentes que el conjunto de eventos de bajo nivel, desde el punto de vista de la programación la diferencia es muy poca.

La principal diferencia reside en la naturaleza del objeto evento que es pasado al controlador de eventos en el momento en que algo sucede. Utilizando la información del objeto evento, los eventos de bajo nivel pueden acceder al componente específico que ha generado el evento, porque todas las clases de eventos de bajo nivel son subclases de la clase **ComponentEvent**. Una vez que la referencia a ese componente está disponible, hay docenas de métodos de la clase **Component** que pueden ser invocados sobre el objeto, como *getLocation()*, *getLocationOnScreen()*, *getName()*, *getMaximumSize()*, *getMinimumSize()*, *getMouseListeners()*, *getWidth()*, *getHeight()*, *getSize()*, *getPreferredSize()*, etc.

Los eventos de tipo semántico, por otro lado, no son subclases de la clase **ComponentEvent**, sino que son subclase de la superclase de **ComponentEvent**, es decir, están al mismo nivel, son hermanos de **ComponentEvent**.

Al no ser subclases de **ComponentEvent**, los objetos evento pasados a controladores de eventos semánticos proporcionan un método para obtener una referencia al objeto que ha generado el evento y, por lo tanto, no pueden invocar los métodos de la clase **Component** sobre ese objeto.

Que lo anterior tenga importancia o no, depende de las necesidades del programa. Por ejemplo, si se necesita determinar la posición del objeto que ha generado un evento, sí se podría hacer procesando un evento de bajo nivel y, probablemente, no se pueda determinar esa posición procesando un evento semántico; y se indica

probablemente, porque nunca se puede decir *nunca*, si no queremos que alguien demuestre que estamos equivocados.

Quitando la posibilidad de acceder al objeto que ha generado el evento, el nombre del objeto está disponible tanto para los eventos de bajo nivel como para los eventos semánticos. En ambos casos, el nombre del objeto está encapsulado en el objeto evento pasado como parámetro y puede extraerse y comprobarse utilizando métodos de la clase **String**. En muchas ocasiones, saber el nombre del objeto es suficiente para conseguir el resultado apetecido.

En el ejemplo `Java1205.java`, se incluyen controladores de bajo nivel y semánticos. Se coloca un objeto **Button** y un objeto **TextField** sobre un objeto **Frame**. El controlador de nivel semántico trata los eventos **Action** y el controlador de bajo nivel trata los eventos **mousePressed** y los eventos **Focus** sobre los mismos componentes.

A continuación, se verán en detalle los eventos y objetos que intervienen en la aplicación.

Eventos de foco

En Java, cuando se dice que un componente tiene el foco, significa que las entradas de teclado se dirigen a ese componente. Hay muchas razones por las que pasa el foco de un componente a otro, y cuando esto sucede, se genera un evento *focusLost()* en el componente que pierde el foco y en el que recibe el foco se genera un evento *focusGained()*. En base a esta pequeña explicación, es fácil comprender que haya muchos tipos de componentes que pueden generar este tipo de eventos, ya que cualquier componente que pueda ganar el foco también podrá perderlo y generará esos eventos.

Hay algunos componentes como son los *Botones* y los *Campos de Texto*, que ganan el foco automáticamente cuando se pulsa sobre ellos con el ratón. En otros componentes, sin embargo, esto no ocurre, como por ejemplo, en las *Etiquetas*, aunque estos componentes pueden ganar el foco si lo solicitan.

Eventos de acción

Un evento **Action** puede ser generado por muchos componentes. Por ejemplo, pulsando con el ratón o pulsando la tecla *Retorno* cuando el foco está sobre un campo de texto, se generará un evento de este tipo. La terminología deriva de que las acciones de usuario generan acciones hacia el programa para realizar algo específico, en función de la naturaleza del componente del que ha surgido el mensaje. Por ejemplo, si un botón tiene la etiqueta "*Salir*" y es pulsado con el ratón, esto significa que el usuario está esperando que el programa realice alguna acción que él interpreta como una salida del programa.

Objeto ActionListener

En este ejemplo, un objeto **ActionListener** es instanciado y registrado para monitorizar de forma semántica eventos de tipo *actionPerformed()* sobre el botón y el campo de texto. Cuando se genera un evento de este tipo, hay cierta información acerca de este evento que es encapsulada en un objeto que se le pasa al método *actionPerformed()* del objeto **Receptor**. La documentación oficial de la plataforma Java llama a esto ***command name***. A esta información puede accederse a través de la invocación al método *getActionCommand()* sobre el objeto. En este ejemplo, se accede al *command name* y se presenta en la pantalla.

El *nombre del comando* asociado con el botón es simplemente el texto o etiqueta que figura sobre el botón. El *nombre del comando* asociado al campo de texto es el texto actual que contiene el campo. Esta información puede ser utilizada de diferente forma por los componentes; por ejemplo, puede ser utilizada para distinguir entre varios botones si no se permite cambiar sus etiquetas durante la ejecución del programa. También se puede utilizar para capturar la entrada del usuario desde un objeto campo de texto.

El objeto de tipo **ActionEvent** que se pasa al método *actionPerformed()* también incluye el nombre del componente, que en este ejemplo es utilizado en comparaciones para identificar el componente que está generando el evento. Se puede hacer utilizando el método *indexOf()* de la clase **String** para determinar si un componente determinado se encuentra incluido en un objeto específico.

En este programa, cada vez que se invoca al método *actionPerformed()*, el código en el cuerpo del método utiliza precisamente la llamada al método *indexOf()* para identificar el componente que ha generado el evento y presenta un mensaje en pantalla indicando el nombre del componente, su *command name*.

Objeto FocusListener

Se instancia un objeto **FocusListener** y también se registra, para monitorizar a bajo nivel los eventos *focusGained()* y *focusLost()*, sobre el botón y el campo de texto.

Cuando se produce un evento *focusGained()*, se presenta en pantalla un mensaje indicando el objeto que ha ganado el foco. De la misma forma, cuando ocurre un evento *focusLost()*, también se presenta un mensaje que indica el objeto que ha perdido el foco.

El objeto que gana o pierde el foco se identifica a través de comprobaciones condicionales sobre el objeto **FocusEvent** pasado como parámetro, del mismo modo que se hace con el objeto **ActionEvent** utilizado en los eventos de acción.

Objeto MouseListener

Un objeto **MouseListener** es instanciado y registrado para monitorizar a bajo nivel eventos de tipo *mousePressed()* sobre los tres objetos que conforman la interfaz. No se controlan otros de los muchos eventos que se producen por mantener una cierta simplicidad en el ejemplo.

El objeto **MouseListener** distingue entre los tres objetos: **Frame**, **Button** y **TextField**, en base al nombre del componente que se asigna a cada objeto en el momento de su instanciación. Si el programador no asigna nombres a los componentes cuando son instanciados, el sistema le asignará nombres por defecto. Estos nombres que asigna el sistema tienen el formato `frame0`, `frame1`, `frame2`, etc., con la parte principal del nombre indicando el tipo de componente y el dígito final se va asignando en el mismo orden en que se van instanciando los objetos.

En este ejemplo, se utiliza el método *indexOf()* sobre el objeto **MouseEvent** para determinar el nombre del componente. Esto es un poco menos complejo que el procedimiento utilizado en ejemplos anteriores, basado en recuperar el objeto e invocar su método *getName()*.

Cuando se produce un evento *mousePressed()* sobre cualquiera de los tres objetos visuales, el objeto **MouseListener** presenta un mensaje en pantalla identificando el objeto que ha generado el evento.

Objeto WindowListener

Por último, para terminar la descomposición del programa, un objeto **WindowListener** es instanciado y registrado para terminar la ejecución de la aplicación cuando el usuario cierra el objeto **Frame**.

Para mantener la simplicidad, la respuesta a los eventos en este programa se limita a presentar información en pantalla. Obviamente, cuando el flujo de ejecución se encuentra dentro del código que controla el evento, el programador puede implementar respuestas diferentes.

También hay que hacer notar al lector, que en el ejemplo se da el hecho de que una sola acción del usuario origina varios tipos diferentes de eventos.

EVENTOS GENERADOS POR EL USUARIO

Hasta ahora se han descrito los eventos que pueden generar los componentes que se integran en una interfaz gráfica. Ahora se va a abordar la creación y lanzamiento de eventos bajo el control del programa que se está ejecutando, que producirán las mismas respuestas que si los eventos tuviesen su origen en alguno de los componentes de la interfaz.

Aunque esta técnica no es imprescindible para entender el funcionamiento del Modelo de Delegación de Eventos, sí es un material crítico a la hora de entrar en el estudio de los componentes *Lightweight*, o *componentes ligeros*, que son aquellos que pueden existir sin necesidad de que haya componentes nativos semejantes en la plataforma en que se ejecuta la aplicación que los usa, por lo que es necesario sentar sólidamente las bases del entendimiento de los *eventos generados por programa* para luego atacar el uso de los componentes *Lightweight*. Además, también esto servirá para entender más fácilmente lo que está sucediendo realmente cuando se implemente el Modelo de Delegación de Eventos utilizando componentes visuales desde el AWT.

Para poder utilizar los eventos generados por programa con las técnicas que se van a describir, será necesario definir una clase que sea capaz de generar este tipo de eventos. Aquí se centrará el estudio en los eventos de tipo **Action**, aunque no hay razón alguna para que la misma técnica se aplique a eventos de bajo nivel como puedan ser los eventos de ratón o del teclado.

La clase en cuestión debe ser una subclase de **Component** y, al menos, debe incluir los siguientes tres miembros:

- Una *variable de instancia* que es una referencia a la lista de objetos **Listener** registrados. En el ejemplo `Java1206.java`, estos objetos receptores de eventos son de tipo **ActionListener**. La variable de instancia es de tipo **ActionListener** y puede contener una referencia a un solo objeto de este tipo o una referencia a una lista de objetos de este tipo.
- Un *método* para crear la lista anterior, que en el ejemplo es *generaListaReceptores()*, que se llama así para ilustrar que el nombre no es técnicamente importante, aunque por consistencia con la documentación del Modelo de Delegación de Eventos, debiera llamarse *addActionListener()*. Esta lista ha de ser generada a través de una llamada al método *AWTEventMulticaster.add()*, que devuelve una referencia a la lista.
- Un *método* que invocará al método correspondiente al tipo de evento en la clase **Listener** de la lista de objetos receptores de los eventos. En el ejemplo, los objetos receptores son de tipo **ActionListener**, así que el método en cuestión es *actionPerformed()*, y el método que lo invoca es *generaEventoAction()*. En este ejemplo solamente hay un objeto receptor.

En la clase **Java1206** se instancia un solo objeto de la clase **NoVisualizable**. También se define un solo objeto de la clase **ActionListener**, que es instanciado y registrado para recibir objetos de tipo **Action** generados sobre el objeto **NoVisualizable**.

La generación del evento **Action** se produce al invocar el método *generaEventoAction()* del objeto de la clase **NoVisualizable**. Este evento es atrapado y procesado por el objeto **ActionListener**, y como resultado del procesado aparecerá en pantalla la información siguiente:

```
% java Java1206
Tutorial de Java, Eventos
metodo actionPerformed() invocado sobre ObjetoNoVisual
```

Es interesante repasar un poco más detalladamente el funcionamiento del programa y el código que lo implementa. Empezando por el constructor, se observa que instancia un objeto del tipo **NoVisualizable**, registra un objeto receptor de eventos sobre ese objeto **NoVisualizable** e invoca al método que hace que el objeto **NoVisualizable** genere un evento de tipo **Action**. Todo esto se hace en las líneas siguientes:

```
NoVisualizable objNoVisual = new NoVisualizable( "ObjetoNoVisual" );
objNoVisual.generaListaReceptores( new ClaseActionListener() );
objNoVisual.generaEventoAction();
```

La siguiente sentencia es la declaración de la variable de instancia en la definición de la clase **NoVisualizable** que referenciará la lista de objetos **Listener** registrados.

```
ActionListener receptorAction;
```

En la línea de código que se reproduce a continuación se encuentra la sentencia que construye la lista de objetos **Listener** registrados añadiendo un nuevo objeto a esa lista. La primera vez que se ejecuta la sentencia en el programa, devuelve una referencia al objeto que se añade. Cuando se ejecute posteriormente, devolverá una referencia a la lista de objetos que está siendo mantenida separadamente.

```
receptorAction = AWTEventMulticaster.add( receptorAction,listener );
```

En este caso, solamente se añade un objeto a la lista y si se examina la referencia que devuelve el método *add()* se observará que es la referencia a un solo objeto.

La última sentencia interesante de este ejemplo es la invocación del método *actionPerformed()* del objeto **ActionListener**, o más propiamente, la invocación de este método en todos los objetos que se encuentren registrados en la lista. Afortunadamente, todo lo que hay que hacer es invocar el método sobre la referencia y el sistema se encarga de realizar la invocación en cada uno de los objetos que están integrados en la lista. Ésta es la característica principal de la clase **AWTEventMulticaster**.

```
receptorAction.actionPerformed(
  new ActionEvent( this,ActionEvent.ACTION_PERFORMED,id ) );
```

Y a esto es a lo que se reduce, en esencia, la generación de eventos desde programa. Aunque se puede complicar todo lo complicable, como se muestra en el ejemplo `Java1207.java`, que ya tiene un poco más de consistencia y está destinado a ilustrar la capacidad de la clase **AWTEventMulticaster** de despachar eventos a más de un objeto a la vez, en concreto a todos los que se encuentren registrados en la lista de objetos **Listener**.

En este ejemplo se registran dos objetos diferentes **ActionListener** sobre un solo objeto **NoVisualizable**, con lo cual la clase **AWTEventMulticaster** deberá lanzar eventos **Action** a dos objetos **Listener** distintos.

La mayor parte del código del ejemplo lo constituyen simples sentencias de impresión de información por pantalla, para explicar qué es lo que está sucediendo durante la ejecución del programa, así que solamente se revisarán las sentencias que resultan interesantes para comprender el funcionamiento y utilización de la clase **AWTEventMulticaster**.

El primer fragmento interesante del código del ejemplo es la instanciación de los dos objetos de la clase **NoVisualizable**, en donde se les asignan también nombres únicos a cada uno de ellos.

```
NoVisualizable aNoVisualizable =
  new NoVisualizable( "ObjNoVisualizable A" );
aNoVisualizable.setName( "ObjNoVisualizableA" );
NoVisualizable bNoVisualizable =
  new NoVisualizable( "ObjNoVisualizable B" );
bNoVisualizable.setName( "ObjNoVisualizableB" );
```

Las siguientes sentencias registran objetos **ActionListener** sobre los dos objetos de la clase **NoVisualizable**. Al contrario que en el ejemplo visto antes, este programa utiliza el nombre convencional de *addActionListener()* para el método que genera la lista de objetos registrados para recibir eventos de tipo **Action**.

```
aNoVisualizable.addActionListener( new PrimerReceptorAction() );
bNoVisualizable.addActionListener( new PrimerReceptorAction() );
bNoVisualizable.addActionListener( new SegundoReceptorAction() );
```

El fragmento de código que se reproduce a continuación es el que hace que cada objeto **NoVisualizable** genere un evento **Action** (se ha eliminado el código superfluo).

```
aNoVisualizable.generaEventoAction();
.
. // Sentencias de impresión por pantalla
.
bNoVisualizable.generaEventoAction();
```

A este código le siguen las definiciones estándar de la clase **ActionListener** y el código en donde comienza la clase **NoVisualizable**, que extiende la clase **Component**, así como la declaración de dos variables de instancia de la clase.

```
class NoVisualizable extends Component {
  String id;
  ActionListener receptorAction;
```

La primera variable de instancia es una referencia al identificador del objeto, que se pasa como parámetro cuando el objeto es instanciado. En el ejemplo, se trata de un objeto de tipo **String** que se pasa como parámetro al constructor en el momento de

instanciar el objeto **NoVisualizable**. El constructor consiste en una sola sentencia que asigna su parámetro de entrada a la variable de instancia.

La siguiente variable de instancia contiene el identificador del objeto **ActionListener** y es vital en el programa, ya que una vez que los objetos **ActionListener** son registrados sobre el objeto **NoVisualizable**, esta variable de instancia contendrá una referencia a un objeto de tipo **AWTEventMulticaster**, en el caso de que haya más de un objeto **ActionListener**.

En este último caso, cuando posteriormente se invoque el método *actionPerformed()* sobre la variable de instancia `receptorAction`, se estará invocando de hecho a este método sobre un objeto de tipo **AWTEventMulticaster**, que invocará a su vez al mismo método de todos los objetos de tipo **ActionListener** contenidos en la lista de objetos **ActionListener** registrados sobre el objeto **NoVisualizable**. Ésta es la característica principal de la clase **AWTEventMulticaster**.

El siguiente código en el que hay que reparar es el que construye la lista de objetos registrados como objetos receptores de eventos sobre un objeto **NoVisualizable** específico.

A la lista se añaden nuevos objetos llamando al método estático *add()* de la clase **AWTEventMulticaster** y pasándole la variable de instancia que referencia la lista, junto con el nuevo objeto **Listener** que se quiere añadir a la lista. Cuando se añade el primer elemento a la lista, se devuelve una referencia al objeto **Listener**, por lo tanto, en el caso de que la lista contenga solamente un objeto receptor de eventos, la referencia a la lista será simplemente una referencia a un objeto **Listener**. Cuando se añaden objetos **Listener** adicionales a la lista, el método *add()* devuelve una referencia a un objeto de tipo **AWTEventMulticaster**. La clase **AWTEventMulticaster** maneja la estructura de una cadena de receptores de eventos y envía eventos a esos receptores.

Cuando posteriormente se invoque el método *actionPerformed()* sobre la referencia a la lista, el método será invocado en realidad sobre un objeto de tipo **ActionListener** o sobre un objeto de tipo **AWTEventMulticaster**. En este segundo caso, el método *actionPerformed()* de la clase **AWTEventMulticaster** asume la responsabilidad de invocar al método *actionPerformed()* de todos los objetos **ActionListener** de la lista.

Con esta larga introducción parece que el código ha de ser muy complejo y, en realidad, se limita a una sola sentencia.

```
receptorAction = AWTEventMulticaster.add( receptorAction,receptor );
```

Especialmente interesante es la salida generada por el programa cuando son registrados los objetos **ActionListener** sobre un solo objeto **NoVisualizable**, ya que en este caso hay que prestar atención al identificador del objeto **Listener** pasado como parámetro al método *addActionListener()* y al identificador del objeto referenciado por

la variable de instancia que referencia la lista. Cuando el primer objeto **Listener** es añadido a la lista, la referencia apuntaba a este mismo objeto. Cuando el segundo objeto **Listener** es incorporado también a la lista, la referencia apunta a `java.awt.AWTEventMulticaster@2087fb`. Esto es lo que se deduce de la reproducción de la salida por pantalla de la ejecución del programa.

```
Invocado el metodo addActionListener()
ObjNoVisualizable A:   El Receptor incorporado es:
  PrimerReceptorAction@20878a
Invocado AWTEventMulticaster.add() para recuperar la referencia a
  ActionListener
ObjNoVisualizable A:   La Ref a ActionListener es:
  PrimerReceptorAction@20878a

Invocado el metodo addActionListener()
ObjNoVisualizable B:   El Receptor incorporado es:
  PrimerReceptorAction@2087c3
Invocado AWTEventMulticaster.add() para recuperar la referencia a
  ActionListener
ObjNoVisualizable B:   La Ref a ActionListener es:
  PrimerReceptorAction@2087c3
```

La última sentencia interesante es la que se encuentra en el método *generaEventoAction()* que instancia un objeto **ActionEvent** e invoca al método *actionPerformed()* sobre la referencia a la lista de objetos **ActionListener**, pasándole el objeto **ActionEvent** como parámetro.

```
receptorAction.actionPerformed( new ActionEvent(
  this,ActionEvent.ACTION_PERFORMED,id ) );
```

Crear eventos propios

Tras la amplia introducción a la clase **AWTEventMulticaster** que se ha visto, se puede atacar la creación y envío de eventos propios, que esencialmente consta de las siguientes fases:

- Definir una clase para que los objetos del nuevo tipo de evento puedan ser instanciados. Debería extender la clase **EventObject**.
- Definir una interfaz **EventListener** para el nuevo tipo de evento. Esta interfaz debería ser implementada por las clases **Listener** del nuevo tipo y debería extender a **EventListener**.
- Definir una clase **Listener** que implemente la interfaz para el nuevo tipo de evento.
- Definir una clase para instanciar objetos capaces de generar el nuevo tipo de evento. En el ejemplo que se verá a continuación, se extiende la clase **Component**.
- Definir un método *add<nombre_del_evento>Listener()* para que pueda mantener la lista de objetos **Listener** registrados para recibir eventos del nuevo tipo.

- Definir un método que envíe un evento del nuevo tipo a todos los objetos **Listener** registrados en la lista anterior, invocando al método de cada uno de esos objetos. Se trata de un método declarado en la interfaz **EventListener** para el nuevo tipo de evento.

El modelo de *Delegación de Eventos* puede soportar el registro de múltiples objetos **Listener** sobre un solo objeto generador de eventos. El modelo también puede soportar el registro de objetos desde una sola clase **Listener** sobre múltiples objetos generadores de eventos. Y también puede soportar combinaciones de los dos.

En la sección anterior se presentó el método *add()* de la clase **AWTEventMulticaster** para crear y mantener una lista de objetos **Listener** registrados. Desafortunadamente, este método no puede utilizarse directamente para mantener una lista de objetos registrados para los eventos creados por el usuario, porque no hay una versión sobrecargada del método cuya signatura coincida con el evento nuevo.

Una forma de crear y mantener estas listas de objetos registrados es embarcarse en un ejercicio de programación de estructuras de datos en una lista. Hay ejemplos de esto en algunos libros, pero aquí no se va a ver, porque parece más interesante una aproximación distinta, que consistirá en la construcción de subclases de **AWTEventMulticaster** y sobrecargar el método *add()* para que tenga una *signature* que coincida con el evento que se está creando. Por supuesto, el cuerpo del método sobrecargado todavía tiene que proporcionar código para procesar la lista, lo cual lo hace de nuevo complicado. Así que el siguiente ejemplo se limita a tener un solo objeto registrado, para evitar la dificultad añadida que supondría, en un primer contacto por parte del lector con este tipo de eventos, el encontrarse con el código de proceso de la lista de eventos.

El programa de ejemplo, `Java1208.java`, muestra cómo se crean, se capturan y se procesan eventos creados por el programador. Se define una clase **NoVisual**, cuyos objetos son capaces de generar eventos del nuevo tipo que se va a crear siguiendo los pasos que se han indicado en párrafos anteriores. En el programa se instancian dos objetos de la clase **NoVisual** y se define una clase **MiClaseEventListener** que implementa la interfaz **MiEventoListener** y define un método para procesar los objetos de tipo **MiEvento** llamando al método *capturarMiEvento()*.

Los objetos de la clase **MiClaseEventListener** son instanciados y registrados para recibir eventos propios desde los objetos **NoVisual**, es decir, se instancian y registran objetos desde una sola clase **Listener** sobre múltiples objetos generadores de eventos. El objeto **NoVisual** contiene un método de instancia llamado *generarMiEvento()*, diseñado para enviar un evento del nuevo tipo al objeto **Listener** registrado sobre el objeto fuente al invocar el método *capturarMiEvento()* del objeto **Listener**.

El método *generarMiEvento()* es invocado sobre ambos objetos **NoVisual**, haciendo que se generen los eventos y que sean capturados y procesados por los objetos registrados de la clase **MiClaseEventListener**. El procesado del evento es muy simple. El identificador y la fuente de la información se extraen del objeto **MiEvento** que es pasado como parámetro al método *capturarMiEvento()* y se presenta esta información en la pantalla.

El constructor de la clase recibe dos parámetros. El primero es una referencia al objeto que ha generado el evento, que es pasada al constructor de la superclase, **EventObject**, donde se almacena para poder acceder a ella a través del método *getSource()* de esa superclase **EventObject**. El segundo parámetro es una cadena de identificación proporcionada al constructor cuando el objeto **MiEvento** es instanciado. En este ejemplo, este identificador es simplemente un **String** almacenado en una variable de instancia en el objeto fuente cuando se instancia, pero se puede pasar cualquier identificación o comando de este modo. La clase también proporciona un método para poder recuperar esa identificación de la variable de instancia del objeto.

El siguiente código interesante es la definición de la nueva interfaz **Listener** llamada **MiEventoListener** que extiende a **EventListener**. Esta interfaz declara el método *capturarMiEvento()* que es el principal en el procesado de eventos de este tipo.

```
interface MiEventoListener extends EventListener {
  void capturarMiEvento( MiEvento evt );
  }
```

El código que se reproduce a continuación es el que implementa la interfaz anterior y define una clase receptora de eventos del nuevo tipo. La clase **Listener** sobrescribe el evento *capturarMiEvento()* declarado en la interfaz y utiliza los métodos de la superclase para acceder y presentar en pantalla la identificación del evento y del objeto que lo ha generado.

```
class MiClaseEventListener implements MiEventoListener {
  public void capturarMiEvento( MiEvento evt ) {
    System.out.println(
      "Metodo capturarMiEvento() invocado sobre " +
      evt.getEventoID() );
    System.out.println(
      "El origen del evento fue " + evt.getSource() );
    }
  }
```

En el fragmento que sigue es donde se define la clase **NoVisual** que extiende la clase **Component** y mantiene dos variables de instancia; una de ellas es el identificador **String** inicializado por el constructor de la clase y la otra es una referencia al objeto **Listener** registrado sobre la fuente de eventos.

```
class NoVisual extends Component {
  String id;
  MiClaseEventListener miReceptor;
```

El código siguiente es el que registra un objeto **Listener** y que, como se puede observar, es muy similar al de ejemplos anteriores, excepto en que no se utiliza el método *add()* de la clase **AWTEventMulticaster** para crear y mantener la lista de objetos registrados. Este código soporta solamente un objeto **Listener** registrado y contiene la lógica para concluir la ejecución si se intenta registrar más de un objeto receptor de eventos.

```
public void addMiEventoListener( MiClaseEventListener receptor ) {
  if( miReceptor == null )
    miReceptor = receptor;
  else {
    System.out.println( "No se soportan multiples Receptores" );
    System.exit( 0 );
    }
  }
```

Y ya sólo resta repasar el código que corresponde al método que genera el evento del nuevo tipo. En este caso, la generación de eventos va acompañada de la llamada al método *capturarMiEvento()* del objeto **Listener** registrado, al que se le pasa un objeto de tipo **MiEvento** como parámetro, que es la forma típica de lanzar eventos en el *Modelo de Delegación* de eventos de la plataforma Java.

```
public void generarMiEvento() {
  miReceptor.capturarMiEvento( new MiEvento( this,ID ) );
  }
```

La cola de eventos del Sistema

Una de las cosas más complicadas de entender del nuevo modelo de gestión de eventos es la creación de eventos y el envío de estos eventos a la *cola de eventos del Sistema*, para que lo lance a un determinado componente. El concepto en sí no es difícil, sin embargo, como la documentación es escasa, se vuelve una ardua tarea el entender el funcionamiento del sistema bajo estas circunstancias.

En este modelo para la creación y envío de eventos a la cola del Sistema, *solamente* es necesario invocar a un método, tal como muestra la siguiente sentencia:

```
Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
  new MouseEvent( miComponente,MouseEvent.MOUSE_CLICKED,0,0,
  -1,-1,2,false ) );
```

Si se examina la sentencia de dentro hacia fuera, se observará que hay una instanciación de un nuevo objeto evento del tipo que se desea; se envía el objeto a la cola de eventos del Sistema que es devuelta por la invocación al método *getSystemEventQueue()*, que es un método de **Toolkit** devuelto por la invocación al método *getDefaultToolkit()*, que es un método estático de la clase **Toolkit**.

En el ejemplo `Java1209.java`, se utiliza la cola de eventos del Sistema para interceptar eventos del teclado generados sobre el botón que ocupa la ventana y convertirlos en eventos del ratón.

El objeto principal de la interfaz gráfica es instanciado desde la clase que extiende a la clase **Frame**. Se crea un componente propio extendiendo la clase **Button**. Los objetos de esta nueva clase así creada son capaces de responder a eventos del teclado y del ratón. Extendiendo la clase **Button** es posible sobrescribir el método *processMouseEvent()* de la clase **Label** que se proporciona a la clase **MiComponente**.

Los eventos del ratón son habilitados sobre los objetos de esta clase para que cualquier evento del ratón sea enviado al método *processMouseEvent()* y, como siempre en estos casos, hay que pasar el objeto **MouseEvent** al método del mismo nombre de la superclase antes de que termine el flujo del programa en el método.

Las pulsaciones sobre el ratón con el cursor posicionado en el objeto botón que se ha creado son enviados al método *processMouseEvent()*, en el cual lo que se hace es presentar la información que contiene el evento en pantalla.

Los eventos del teclado, **KeyEvent**, son capturados por el receptor **KeyListener**. Cuando se captura un evento del teclado se genera un objeto **MouseEvent** sintético y se coloca en la cola de eventos del Sistema. La documentación de la plataforma Java 2 para la creación de un objeto **MouseEvent** es la siguiente:

```
public MouseEvent(Component source,
                  int id,
                  long when,
                  int modifiers,
                  int x,
                  int y,
                  int clickCount,
                  boolean popupTrigger)
```

En el programa, `source` es una referencia al objeto que se ha creado. Se asignan valores negativos a los parámetros `x` e `y` para que el objeto sea fácilmente reconocible cuando se genere, ya que una pulsación real de ratón nunca podrá tener valores negativos en las coordenadas. Para el resto de parámetros se asignan valores aleatorios, excepto para `id`. Este parámetro es crítico a la hora de crear el objeto **MouseEvent**, ya que debe coincidir con alguna de las constantes simbólicas que están definidas en la clase **MouseEvent** para los diferentes eventos del ratón que reconoce el Sistema. Estas constantes son:

`MOUSE_FIRST`	Primer entero usado como `id` en el rango de eventos
`MOUSE_LAST`	Último entero usado como `id` en el rango de eventos
`MOUSE_CLICKED`	
`MOUSE_PRESSED`	
`MOUSE_RELEASED`	

```
MOUSE_ENTERED
MOUSE_EXITED
MOUSE_DRAGGED
```

Si el identificador `id` asignado al nuevo evento no se encuentra en el rango anterior, no se producirá ningún aviso ni se genera ninguna excepción, ni en la compilación ni en la ejecución del programa, simplemente no se envía ningún evento.

Cuando se ejecuta el programa aparece una ventana en la pantalla totalmente ocupada con el componente propio creado al extender la etiqueta. Si se mueve el ratón, se producirán mensajes normales en la pantalla del sistema generados por el método *processMouseEvent()*; pero si se pulsa una tecla, aparecerá un mensaje indicando la tecla pulsada y, a continuación, se indica que hay una llamada al método de procesado de eventos del ratón, al haberse generado un evento sintético provocado por la pulsación de la tecla. En las líneas siguientes se observa la salida por pantalla que se acaba de describir.

```
Metodo processMouseEvent(), MiComponente ID = 504
  java.awt.Point[x=239,y=28]
Metodo keyPressed(), tecla pulsada -> x
Metodo processMouseEvent(), MiComponente ID = 500
  java.awt.Point[x=-1,y=-1]
Metodo processMouseEvent(), MiComponente ID = 505
  java.awt.Point[x=246,y=47]
Metodo processMouseEvent(), MiComponente ID = 504
  java.awt.Point[x=227,y=23]
Metodo keyPressed(), tecla pulsada -> y
Metodo processMouseEvent(), MiComponente ID = 500
  java.awt.Point[x=-1,y=-1]
```

Echando un vistazo al código del programa se observa que hay gran parte ya vista en ejemplos anteriores; así que, siguiendo la tónica de otras secciones, se revisan a continuación los trozos de código que se han introducido nuevos o que merece la pena volver a ver. El primer fragmento son las sentencias en que se instancia un objeto del tipo **MiComponente** y se registra un objeto **KeyListener** para recibir los eventos del teclado sobre ese componente. Posteriormente, será el código de ese receptor **KeyListener** el que atrapará los objetos **KeyEvent**, creará objetos **MouseEvent** sintéticos y los enviará a la cola del Sistema como provenientes de **MiComponente**.

```
MiComponente miComponente = new MiComponente();
  this.add( miComponente );
  . . .
  miComponente.addKeyListener( new MiKeyListener( miComponente ) );
```

La sentencia siguiente es la ya vista al comienzo, en la que se proporcionan todos los parámetros necesarios para generar el evento del ratón sintético y enviarlo a la cola de eventos del Sistema. Entre esos parámetros está la referencia al componente creado, que en la descripción anterior era el parámetro `source`, luego está el parámetro `id` del nuevo evento, que es uno de la lista presentada antes y, finalmente, están las coordenadas `x` e `y`, que se fijan a `-1` para poder reconocer fácilmente el evento sintético entre los mensajes que aparezcan por la pantalla.

```
Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
  new MouseEvent( miComponente,MouseEvent.MOUSE_CLICKED,0,0,
  -1,-1,2,false ) );
```

Lo que ocurre aquí, si se piensa un poco en ello, es que hay un gran problema de seguridad si se permite a applets que no estén certificados manipular libremente la cola de eventos del Sistema. Por ello, el método *getSystemEventQueue()* está protegido por comprobaciones de seguridad que impiden a los applets acceso directo a la cola de eventos.

Ahora se encuentra la sentencia que permite la generación de eventos y que está contenida en el constructor de la clase **MiComponente**. Esta sentencia es necesaria para poder invocar al método sobrescrito *processMouseEvent()* siempre que un evento de ratón sea enviado a un objeto de la clase **MiComponente**.

```
enableEvents( AWTEvent.MOUSE_EVENT_MASK );
```

Finalmente, para concluir el repaso al código del ejemplo, está el método sobrescrito *processMouseEvent()* que anuncia la invocación del método y presenta en pantalla información del `id` del evento y las coordenadas que contiene. La sentencia final es la llamada al mismo método de la superclase, si no se hace esta llamada, la salida probablemente no sea del todo correcta.

```
public void processMouseEvent( MouseEvent evt ) {
  System.out.println(
    "Metodo processMouseEvent(), MiComponente ID = " +
    evt.getID() + " " + evt.getPoint() );
  super.processMouseEvent( evt );
  }
```

Y el resto del código es el mismo que el de muchos de los ejemplos anteriores.

Intercambio de componentes

Todavía se verán algunos ejemplos más, porque nunca por mucho fue mal año y es mejor reforzar lo que se sabe, o decir lo mismo de otro modo, para que el lector pueda reconocer lo mismo desde diferentes puntos de vista si no se ha aclarado con explicaciones anteriores y, además, en todos los ejemplos habrá algo que pueda fijarse de mejor forma que lo que haya hecho otro anterior, y eso es lo que se pretende.

En este caso, se va a tratar de hacer que un objeto **Label** funcione como un objeto **Button**, provocando que genere eventos de tipo **ActionEvent** atribuibles a un botón; es decir, los eventos serán enviados a un objeto receptor **ActionListener** registrado sobre un **Button**, donde serán procesados como si fuesen eventos originados por ese objeto **Button**, cuando en realidad su origen es el objeto **Label**.

Al contrario que en el ejemplo anterior, en este ejemplo, `Java1210.java`, no se va a sobrescribir ningún método *processXxxEvent()*, sino que se centrará el código completamente al modelo fuente/receptor del Modelo de Delegación de Eventos. Dos

objetos **Label** y un objeto **Button** son instanciados y añadidos a un objeto **Frame**. Cuando se pulsa sobre el botón, se genera un evento de tipo **ActionEvent** que es atrapado por un objeto **ActionListener** registrado sobre el objeto **Button**. El código del método *actionPerformed()* del objeto **ActionListener** cambia el color de fondo del objeto **Label** `Muestra` entre azul y amarillo, y viceversa.

Hasta aquí todo es normal. Sin embargo, se registra un objeto **MouseListener** sobre el objeto **Label**, para que cuando se pulse sobre la etiqueta, el código del método *mouseClicked()* del objeto **MouseListener** genere un evento **ActionEvent** y lo coloque en la cola de eventos del Sistema. Lo que hace es simular que ha sido pulsado el botón, colocando la identificación del objeto **Button** en el parámetro `source` del objeto **ActionEvent**. El Sistema enviará este objeto **ActionEvent** al objeto **ActionListener** registrado sobre el objeto **Button**. El resultado final es que pulsando el ratón con el cursor colocado sobre la etiqueta, se invoca al método *actionPerformed()* registrado sobre el objeto **Button**, produciéndose exactamente el mismo resultado que cuando se pulsa el botón.

Durante la ejecución del programa se puede observar un efecto colateral curioso, y es que la etiqueta responde más rápido al ratón que el propio botón. Esto probablemente sea debido a los repintados que tiene que hacer el botón para simular la animación de la pulsación; quizá si se utilizasen simulaciones de los eventos *pulsar* y *soltar* sobre la etiqueta y se fuesen cambiando los colores de fondo, por ejemplo, se produjese una degradación similar.

En la revisión del ejemplo, el primer fragmento de código sobre el que hay que reparar en el ejemplo, son las dos sentencias en el constructor que registran objetos **Listener** sobre el botón y la etiqueta. Una cosa importante es que la referencia al objeto **Button** es pasada al constructor del objeto **MouseListener** del objeto **Label**, siendo éste el enlace que permite crear el evento **ActionEvent** desde la etiqueta y atribuírselo al botón.

También es interesante observar que el objeto **Listener** de la etiqueta no sabe nada sobre la otra etiqueta, `Muestra`, cuyo color se cambia durante la ejecución del programa. Esto es porque el objeto **Label** que va a simular al botón, no es el que cambia directamente el color del otro objeto **Label**; sino que va a ser el botón el encargado de realizar esa tarea.

```
miClickLabel.addMouseListener( new MiMouseListener( miBoton ) );
miBoton.addActionListener( new MiActionListener( miColorLabel ) );
```

La siguiente sentencia resultará conocida. Es aquella que va en el método *mouseClicked()* del objeto **MouseListener** que crea y envía el objeto **ActionEvent** y lo atribuye al objeto **Button**. Esto es posible, ya que tiene acceso a la referencia del objeto **Button**. Esta referencia es pasada como parámetro cuando el objeto **MouseListener** es construido, aunque puede hacerse de otras formas, por ejemplo, se podría consultar periódicamente la cola de eventos del Sistema en espera de cazar un evento de tipo **ActionEvent** y obtener la referencia del botón. Ésta es una de las

razones por las que se aplican normas estrictas de seguridad en el acceso de los applets a la cola de eventos del Sistema, porque podría ser consultada y obtener todo tipo de información si no se hiciese así.

Como el objeto **ActionEvent** es atribuido al objeto **Button**, será enviado al objeto **ActionListener** registrado sobre ese **Button**, donde será procesado como si su origen estuviese realmente en el botón.

```
Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(
  new ActionEvent( miBoton,
  ActionEvent.ACTION_PERFORMED,"evento" ) );
```

Lo único que resta es llamar la atención sobre la sintaxis del objeto **ActionEvent**, en donde se utiliza la referencia al objeto **Button** como primer parámetro, `source`, y hay que recordar que se debe indicar un valor correcto para el parámetro `id`, si no el evento no será tratado por el Sistema.

EVENTOS EN SWING

El modelo de eventos en **Swing** es el modelo de Delegación. Los componentes Swing no soportan el modelo de eventos de Propagación, sino solamente el modelo de Delegación incluido desde el JDK 1.1; por lo tanto, si se van a utilizar componentes Swing, se debe programar exclusivamente en el nuevo modelo. Aunque hay componentes de Swing que se corresponden (o reemplazan) a componentes de AWT, hay otros que no tienen una contrapartida en el AWT.

Desde el punto de vista del manejo de eventos, los componentes de Swing funcionan del mismo modo que los componentes del AWT, a diferencia de los nuevos eventos que incorpora Swing. Desde otros puntos de vista, los componentes Swing pueden parecerse ya más o menos a su contrapartida del AWT. Además, hay una serie de componentes muy útiles en Swing, como son los árboles, la barra de progreso, los *tooltips*, los *spins*, que no tienen ningún componente semejante en el AWT.

En este capítulo solamente se van a tratar los componentes Swing desde el punto de vista del control de eventos, dejando para un capítulo individual el estudio de los componentes en sí mismos y de las características que aportan a Java.

En el ejemplo `Java1211.java`, que es equivalente al ejemplo `Java1201.java`, solamente se reemplaza cada instancia de **Frame** por una instancia de **JFrame**, además de incorporar la sentencia `import` que hace que las clases sean accesibles al compilador y al intérprete. Por lo demás, el control de eventos en este nuevo ejemplo es el mismo que se ejercía en el ejemplo del AWT.

En el ejemplo se puede comprobar la utilización de fuentes de eventos, receptores de eventos y adaptadores del Modelo de Delegación de Eventos para componentes Swing. Sucintamente, la aplicación instancia un objeto que crea una interfaz de

usuario consistente en un **JFrame**. Este objeto es una fuente de eventos que notificará a dos objetos diferentes, receptores de eventos, de eventos de tipo **Window**.

Uno de los objetos **Listener**, receptores de eventos, implementa la interfaz **WindowListener** y define todos los métodos que se declaran en esa interfaz. El otro objeto **Listener**, extiende la clase **Adapter**, adaptador, llamada **WindowAdapter**, que ya no tiene por qué sobrescribir todos los métodos de la interfaz, sino solamente aquellos que le resultan interesantes.

La aplicación no termina y devuelve el control al sistema operativo, sino que esto debe forzarse. El código del ejemplo se reproduce a continuación:

```
import javax.swing.*;    // Éste es el paquete de Swing

public class Java1211 {
  public static void main( String args[] ) {
. . .
  // Constructor de la clase
  public IHM() {
    // Se crea un objeto JFrame
    JFrame ventana = new JFrame();
. . .
class Proceso1 implements WindowListener {
  // Variable utilizada para guardar una referencia al objeto JFrame
  JFrame ventanaRef;

  // Constructor que guarda la referencia al objeto JFrame
  Proceso1( JFrame vent ){
    this.ventanaRef = vent;
    }
. . .
```

Lo único destacable del código es el fragmento del comienzo del constructor de la clase **IHM** en que se utiliza la clase **JFrame** para instanciar al contenedor principal de la interfaz gráfica.

```
class IHM {
  // Constructor de la clase
  public IHM() {
    // Se crea un objeto JFrame
    JFrame ventana = new JFrame();

    // El método setSize() reemplaza al método resize() del JDK 1.0
    ventana.setSize( 300,200 );
    ventana.setTitle( "Tutorial de Java, Eventos" );
```

Si se compila y ejecuta el programa, el lector podrá comprobar que no hay diferencias aparentes con el ejemplo `Java1201.java`, presentado en la sección anterior.

Pero tampoco las cosas son tan sencillas como parecen, porque hay ocasiones en que la conversión de un programa de AWT a Swing no es tan simple como una

sustitución e importar la librería. Esto es lo que se muestra en el ejemplo `Java1212.java`, que es la contrapartida en Swing de la clase **Java1202**, ya vista al tratar del AWT. Se ha hecho lo mismo que en el caso anterior, sustituir **Frame** por **JFrame** e incorporar la sentencia que importa la librería de Swing.

La intención del programa era presentar las coordenadas del cursor en la posición en que se pulsase, dentro del área de influencia del **Frame**, en este caso del **JFrame**.

```
import javax.swing.*;   // Éste es el paquete de Swing
. . .
class MiFrame extends JFrame {
  int ratonX;
  int ratonY;
. . .
```

Si se compila y ejecuta el programa, inicialmente todo parece ir bien. Sin embargo, cuando se pulsan varias veces, se observa que las indicaciones de las coordenadas anteriores no desaparecen de la pantalla (ver figura 12.2). Si se hace que el foco pase a otra de las aplicaciones que estén corriendo, entonces sí desaparecen esos textos antiguos de la ventana. Quizá sea un ejercicio interesante para el lector el descubrir por sí mismo el porqué de este funcionamiento aparentemente anómalo.

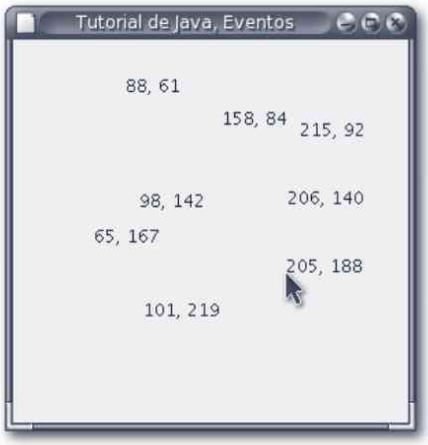


Figura 12.2

La única parte destacable del código del programa, por destacar algo, es la que extiende la clase **JFrame** para dar origen a la clase **MiFrame**, en aras de hacer posible la sobrescritura del método *paint()* de la clase **JFrame**.

```
class MiFrame extends JFrame {
  int ratonX;
  int ratonY;
  public void paint( Graphics g ) {
    g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
    }
  }
```

El lector recordará que en el caso del ejemplo `Java1202.java`, la clase **Frame** se extendía de una forma similar para dar origen a otra clase donde poder sobrescribir el método *paint()*.

Nuevos eventos en Swing

Aunque en Swing la forma de manejar los eventos es similar a la del AWT del Modelo de Delegación, las clases Swing proporcionan una serie de nuevos tipos de eventos.

Una de las formas más fáciles de identificar estos nuevos tipos de eventos que incorpora Swing, es consultar las interfaces definidas en Swing o, también, echar una ojeada a la definición de las clases de los eventos en Swing. Ahora se presentan dos tablas, la de la izquierda muestra una lista de las interfaces receptoras definidas en el paquete Swing y la de la derecha muestra la lista de clases Evento definidas en ese mismo paquete.

AncestorListener	**AncestorEvent**
CaretListener	**CaretEvent**
CellEditorListener	**ChangeEvent**
ChangeListener	**EventListenerList**
DocumentEvent	**HyperlinkEvent**
DocumentListener	**InternalFrameAdapter**
HyperlinkListener	**ListDataEvent**
InternalFrameListener	**ListSelectionEvent**
ListDataListener	**MenuEvent**
ListSelectionListener	**PopMenuEvent**
MenuListener	**TableColumnModelEvent**
PopMenuListener	**TableModelEvent**
TableColumnModelListener	**TreeExpansionEvent**
TableModelListener	**TreeModelEvent**
TreeExpansionListener	**TreeSelectionEvent**
TreeModelListener	**UndoableEditEvent**
TreeSelectionListener	
UndoableEditListener	

Se puede observar que no hay una correspondencia obvia entre las interfaces receptores y las clases de evento en todos los casos. En los dos ejemplos que siguen, se verán la clase **AncestorEvent** y la interfaz **AncestorListener**.

El primero de estos programas, `Java1213.java`, muestra el uso de *getContentPane()* para añadir un objeto Swing de tipo **JButton** a un **JFrame**, de la misma forma que se ha visto en ejemplos de otras secciones, pero en este caso se profundiza en lo que a los eventos se refiere, ilustrando el uso de la interfaz **AncestorListener** sobre un objeto Swing **JButton**.

A la hora de repasar el código de la aplicación, en este caso hay que empezar desde las sentencias `import`, porque ya en ellas está el código que permite que tanto compilador como intérprete Java puedan acceder a las clases Swing.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.swing.*;
import java.awt.swing.event.*;
```

El método *main()*, en este caso es tan simple, que no merece comentario alguno. No obstante, ahí se construye un objeto de la clase **IHM** que será el que se presente en pantalla. El constructor de esta clase es muy sencillo, instancia un objeto Swing de tipo **JFrame**, fijando su tamaño, proporcionándole un título, etc. Además, también se añade un objeto **WindowListener** para capturar los eventos de la ventana y terminar el programa cuando el usuario cierre el objeto **JFrame**.

```
JFrame ventana = new JFrame();
ventana.setSize( 300,300 );
ventana.setTitle( "Tutorial de Java, Swing" );
ventana.addWindowListener( new Conclusion() );
```

Luego se instancia un objeto Swing de tipo **JButton** y se registra un objeto de tipo **AncestorListener** sobre ese botón. A continuación, se añade el objeto **JButton** al objeto **JFrame** ventana, invocando al método *getContentPane()* y luego al método *add()* sobre el contenido anterior. Por fin, se presenta un mensaje y se hace visible el objeto **JFrame**, concluyendo el constructor.

```
JButton boton = new JButton( "Boton" );
boton.addAncestorListener( new MiAncestorListener() );
ventana.getContentPane().add( boton );
System.out.println( "Se hace visible el JFrame" );
ventana.setVisible( true );
```

El método *getContentPane()* no existe en el AWT, donde la única forma de añadir componentes es manipular directamente el área cliente del objeto **Frame**. En Swing, sin embargo, algunos *paneles* son colocados automáticamente sobre el área cliente de un objeto **JFrame**, con lo cual se pueden añadir componentes, o manipular, estos *paneles* en vez de manipular el área cliente del objeto **JFrame** directamente.

La definición de la clase **JFrame** es la siguiente:

```
public class JFrame extends Frame implements WindowConstants,
  Accesible,RootPaneContainer
```

Es decir, es una extensión de la clase **Frame** del AWT que le añade soporte para recibir entradas y pintar sobre objetos **Frame** hijos, soporte para hijos especiales controlados por un **LayeredPanel** y soporte para las barras de menú de Swing. Esta clase **JFrame** es ligeramente incompatible con la clase **Frame** del AWT. **JFrame** contiene un objeto **JRootPane** como único hijo. El **contentPane** debería ser el padre

de cualquier hijo del **JFrame**. Esto difiere con respecto al **Frame** del AWT; por ejemplo, para añadir un hijo a un **Frame**, se escribiría

```
frame.add( hijo );
```

mientras que en el caso del **JFrame**, es necesario añadir el hijo al **contentPane**, de la siguiente forma:

```
frame.getContentPane().add( hijo );
```

Esto mismo es válido a la hoja de fijar el controlador de posicionamiento de los componentes, eliminar componentes, listar los hijos, etc. Todos estos métodos normalmente deberían ser enviados al **contentPane** en vez de directamente al **JFrame**. El **contentPane** siempre será distinto de nulo y, por defecto, tendrá un **BorderLayout** como controlador de posicionamiento. Si se intenta hacer que **contentPane** sea nulo, el sistema generará una excepción. En este ejemplo es suficiente con insertar una llamada al método entre la referencia al objeto **JFrame** y la llamada a *add()*, *setLayout()*, etc. En programas más complejos, las ramificaciones probablemente creasen mayores problemas.

La interfaz de usuario del programa tiene dos clases anidadas. Una de ellas es una clase **WindowListener** que se utiliza para concluir la ejecución del programa cuando el usuario cierra el **JFrame**. Es muy simple, y ya se ha visto en otras secciones, aquí se incluye como anidada para mostrar la versatilidad de Java y ver que se puede hacer lo mismo de varias formas diferentes. Java también dispone de acciones por defecto sobre **JFrame**, cuando se cierra la ventana. La acción predeterminada es simplemente ocultar la ventana, y si se quiere cerrar, es necesario incluir la línea de código siguiente, que reemplaza a la detección por parte del programador del cierre de la ventana, porque será el propio **JFrame** quien actúe.

```
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

La segunda clase anidada se utiliza para instanciar un objeto de tipo **AncestorListener** que será registrado sobre el objeto **JButton**. Esto ya es un poco más interesante. La interfaz **AncestorListener** declara tres métodos, así que la clase debe implementar estos tres métodos, que son:

`ancestorAdded(AncestorEvent)`, llamado cuando el origen o uno de sus antecesores se hace visible, bien porque se llame al método *setVisible()* pasándole el parámetro `true`, o porque se haya añadido el componente a la jerarquía.

`ancestorMoved(AncestorEvent)`, llamado cuando el origen o uno de sus antecesores es movido.

`ancestorRemoved(AncestorEvent)`, llamado cuando el origen o uno de sus antecesores se hace invisible, bien porque se llame al método *setVisible()* pasándole el parámetro `false`, o porque se haya eliminado el componente de la jerarquía.

Como se puede observar, cuando alguno de estos métodos es llamado, se le pasa un objeto de tipo **AncestorEvent** como parámetro. Cuando se llama al primero de ellos, invoca a su vez a métodos del **AncestorEvent** que se le pasa para presentar en pantalla información sobre el antecesor.

```
class MiAncestorListener implements AncestorListener{
  public void ancestorAdded( AncestorEvent evt ) {
    System.out.println( "Llamada al metodo ancestorAdded" );
    System.out.println( "Origen Evento: " + evt.getSource() );
    System.out.println( "Ancestor: " + evt.getAncestor() );
    System.out.println( "Padre: " + evt.getAncestorParent() );
    System.out.println( "Componente: " + evt.getComponent() );
    System.out.println( "ID: " + evt.getID() );
    }
```

La verdad es que si se compila y ejecuta el programa, se obtendría algo como la salida de pantalla capturada a continuación:

```
% java Java1213
Se hace visible el JFrame
Llamada al metodo ancestorAdded
Origen Evento: java.awt.swing.JButton[,0,0,0x0,invalid,
    layout=java.awt.swing.OverlayLayout]
Ancestor: java.awt.swing.JButton[,0,0,0x0,invalid,
    layout=java.awt.swing.OverlayLayout]
Padre: java.awt.swing.JPanel[null.contentPane,0,0,0x0,invalid,
    layout=java.awt.swing.JRootPane$1]
Componente: java.awt.swing.JButton[,0,0,0x0,invalid,
    layout=java.awt.swing.OverlayLayout]
ID: 1
Metodo ancestorMoved
```

y esto no parece coincidir con las descripciones que *Sun Microsystems* proporciona en su documentación sobre los métodos *getAncestor()* y *getAncestorParent()*. La salida parece referirse al objeto **JButton** como antecesor y al objeto **JRootPane** como padre del antecesor. Pero lo cierto es que el objeto **JButton** es un hijo, no un antecesor, aunque es de imaginar que esto dependerá de la interpretación que se le quiera dar. El caso es que no está demasiado claro en la documentación.

El fragmento final del código es el que muestra la definición de los otros dos métodos de la interfaz **AncestorListener**, y que se reproduce a continuación:

```
public void ancestorRemoved( AncestorEvent evt ) {
  System.out.println( "Metodo ancestorRemoved" );
  }
public void ancestorMoved( AncestorEvent evt ) {
  System.out.println( "Metodo ancestorMoved" );
  }
```

Si se ejecuta el programa, aparte de observar la salida referida anteriormente, se observa que cuando se mueve, iconiza o desiconiza la ventana, se generan llamadas al método *ancestorMoved()*. Cuando se hace visible el **JFrame**, se llaman a los dos métodos, *ancestorAdded()* y *ancestorMoved()*.

A continuación, se muestra otro ejemplo, `Java1214.java`, que ilustra el uso de un receptor de tipo **AncestorListener** sobre un **JButton**, y lo que es más importante, ilustra el hecho de que objetos como **JButton** pueden ser contenedores de otros objetos, incluyendo entre ellos a otros objetos **JButton**.



Figura 12.3

El programa apila tres objetos **JButton** y los coloca sobre un objeto **JFrame**, tal como muestra la figura 12.3. Los objetos **ActionListener** se registran sobre cada uno de los botones para atrapar los eventos de tipo **Action** cuando se pulsa el botón y poder presentar la información correspondiente al origen del evento. Los objetos **AncestorListener** también son registrados sobre los objetos **JButton**.

La mayor parte del código de este programa es semejante al de los anteriores. Cuando se instancian los tres botones, se apilan añadiendo el `segundoBoton` al `primerBoton` y añadiendo el `tercerBoton` al `segundoBoton`.

Se registra un **AncestorListener** sobre los tres botones y luego un objeto **ActionListener** también sobre ellos. La clase **AncestorListener** es muy similar a la del ejemplo anterior; no obstante, es de destacar la necesidad de moldeo en esta versión del método. Esto se debe a la invocación del método *getSource()* que devuelve un objeto de tipo **Object**, que debe ser moldeado a un **JButton** para que pueda ser utilizado.

```
class MiAncestorListener implements AncestorListener {
  // Se definen los tres métodos declarados en la interfaz
  // AncestorListener, incorporando el moldeo necesario para
  // que nadie se queje
  public void ancestorAdded( AncestorEvent evt ) {
    System.out.println( "Metodo ancestorAdded" );
    System.out.println( " Origen del evento: " +
      ( (JButton)evt.getSource() ).getActionCommand() );
    }
```

Y ya lo más interesante es ver cómo la clase **ActionListener** atrapa los eventos de tipo **Action** que se producen en los botones cuando se pulsan.

```
class MiActionListener implements ActionListener {
  public void actionPerformed( ActionEvent evt ) {
    System.out.println( "Metodo actionPerformed" );
    System.out.println( " Origen del evento: " +
      ( (JButton)evt.getSource() ).getActionCommand() );
    }
  }
```

CAPÍTULO 13

AWT

AWT es el acrónimo del **X** **W**indow **T**oolkit para Java, donde **X** puede ser cualquier cosa: Abstract, Alternative, Awkward, Another o *Asqueroso*; aunque parece que *Sun Microsystems* se decanta por *Abstracto* (seriedad por encima de todo). Se trata de una biblioteca de clases Java para el desarrollo de Interfaces Gráficas de Usuario. La versión del AWT que *Sun Microsystems* proporciona con el JDK se desarrolló en sólo dos meses y, aunque se ha retocado mucho posteriormente, sigue siendo la parte más débil de todo lo que representa Java como lenguaje. El entorno que ofrece es demasiado simple, no se han tenido en cuenta las ideas de entornos gráficos novedosos. Quizá la presión de tener que lanzar algo al mercado haya tenido mucho que ver en la pobreza de AWT, que luego se intentó paliar con ***Swing*** y se ha conseguido de forma más que sobrada.

Sun Microsystems, en vista de la precariedad mostrada por AWT, y para asegurarse de que los elementos que desarrolla para generar interfaces gráficas sean fácilmente transportables entre plataformas, constituyó un grupo de trabajo con *Netscape*, *IBM* y *Lighthouse Design* para crear un conjunto de clases que proporcionen una sensación visual agradable y sean más fáciles de utilizar por el programador. Fruto de los trabajos de ese grupo surgió un conjunto de clases. Esta colección de clases son las *Java Foundation Classes* (JFC), que en la plataforma Java 2 están constituidas por cinco grupos de clases: AWT, Java 2D, Accesibilidad, Arrastrar y Soltar y Swing.

- **AWT**, engloba a todos los componentes del AWT que existían en la versión 1.1.2 del JDK y los que se han incorporado en versiones posteriores. Se ha modificado de modo que sobre entornos ***X*** ahora está implementado utilizando *Xlib*, no *Motif* como era originalmente.

- **Java 2D** es un conjunto de clases gráficas bajo licencia de *IBM/Taligent*. Utiliza *OpenGL* para aumentar la velocidad de renderizado en todos los entornos, hasta J2SE 5 en entornos Windows se basaba en *DirectX*.

- **Accesibilidad**, proporciona clases para facilitar el uso de ordenadores y tecnología informática a disminuidos, como lupas de pantalla, y cosas así.

- **Arrastrar y Soltar** (*Drag and Drop*), son clases en las que se soporta *Glasgow*, una de las generaciones de *JavaBeans*.

- **Swing**, es la parte más importante. Ha sido creada en conjunción con **Netscape** y proporciona una serie de componentes muy bien descritos y especificados de forma que su presentación visual es independiente de la plataforma en que se ejecute el applet o la aplicación que utilice estas clases. **Swing** simplemente extiende el AWT añadiendo un conjunto de componentes, *JComponents*, y sus clases de soporte. Hay un conjunto de componentes de Swing que son análogos a los de AWT, y algunos de ellos participan de la arquitectura MVC (*Modelo-Vista-Controlador*), aunque **Swing** también proporciona otros widgets nuevos como árboles, pestañas, *tooltips*, *spins*, etc.

La estructura básica del AWT se basa en *Componentes y Contenedores*. Estos últimos contienen componentes posicionados a su respecto y son componentes a su vez, de forma que los eventos pueden tratarse tanto en contenedores como en componentes, siendo cuenta del programador el encaje de todas las piezas, así como la seguridad de tratamiento de los eventos adecuados. Con Swing se va un paso más allá, ya que todos los *JComponentes* son subclases de **Container**, lo que hace posible que widgets Swing puedan contener otros componentes, tanto de AWT como de Swing.

Como se ha indicado, AWT sobre entornos ***X*** está implementado utilizando *Xlib* y recibe el nombre de ***XAWT***. Utiliza el protocolo *XDnD* para poder ejecutar acciones de arrastrar y soltar entre Java y otras aplicaciones como *StarOffice* o *Mozilla*.

INTERFAZ DE USUARIO

La *interfaz de usuario* es la parte de una aplicación que permite a ésta interactuar con el usuario. Las interfaces de usuario pueden adoptar muchas formas, que van desde la simple línea de comandos hasta las interfaces gráficas que proporcionan las aplicaciones más modernas.

La interfaz de usuario es el aspecto más importante de cualquier aplicación. Una aplicación sin una interfaz fácil, impide que los usuarios obtengan el máximo rendimiento del programa. Java proporciona los elementos básicos para construir *decentes* interfaces de usuario a través del AWT, y opciones para mejorarlas mediante Swing, que sí permite la creación de interfaces de usuario de gran impacto y sin demasiados quebraderos de cabeza por parte del programador.

Al nivel más bajo, el sistema operativo transmite información desde el ratón y el teclado como dispositivos de entrada al programa. El AWT fue diseñado pensando en que el programador no tuviese que preocuparse de detalles como controlar el movimiento del ratón o leer el teclado, ni tampoco de detalles como la escritura en pantalla. El AWT constituye una librería de clases orientada a objeto para cubrir estos recursos y servicios de bajo nivel.

Debido a que el lenguaje de programación Java es independiente de la plataforma en que se ejecuten sus aplicaciones, el AWT también es independiente de la plataforma en que se ejecute. El AWT proporciona un conjunto de herramientas para la construcción de interfaces gráficas que tienen una apariencia y se comportan de forma semejante en todas las plataformas en que se ejecute. Los elementos de interfaz proporcionados por el AWT están implementados utilizando *toolkits* nativos de las plataformas, preservando una apariencia semejante a todas las aplicaciones que se creen para esa plataforma. Éste es un punto fuerte del AWT, pero también tiene la desventaja de que una interfaz gráfica diseñada para una plataforma puede no visualizarse correctamente en otra diferente. Estas carencias del AWT son subsanadas en parte por Swing, y en general por las JFC.

ESTRUCTURA DEL AWT

La estructura de la versión actual del AWT en la plataforma Java 2 se puede resumir en los puntos siguientes:

- Los Contenedores contienen componentes, que son los controles básicos.
- No se usan posiciones fijas de los componentes, sino que están situados a través de una disposición controlada (*layouts*).
- El común denominador de más bajo nivel se acerca al teclado, ratón y manejo de eventos.
- Alto nivel de abstracción respecto al entorno de ventanas en que se ejecute la aplicación (no hay áreas cliente, ni llamadas a *X*, ni `hWnds`, etc.).
- La arquitectura de la aplicación es dependiente del entorno de ventanas, en vez de tener un tamaño fijo.
- Es bastante dependiente de la máquina en que se ejecuta la aplicación (no se puede asumir que un diálogo tendrá el mismo tamaño en cada máquina).
- Carece de un formato de recursos. No se puede separar el código de lo que es propiamente interfaz.

COMPONENTES Y CONTENEDORES

Una interfaz gráfica está construida en base a elementos gráficos básicos, los *Componentes*. Típicos ejemplos de estos componentes son los botones, barras de desplazamiento, etiquetas, listas, cajas de selección o campos de texto. Los componentes permiten al usuario interactuar con la aplicación y proporcionar información desde el programa al usuario sobre el estado del programa. En el AWT, todos los componentes de la interfaz de usuario son instancias de la clase **Component** o uno de sus subtipos.

Los componentes no se encuentran aislados, sino agrupados dentro de *Contenedores*. Los contenedores contienen y organizan la situación de los componentes; además, los contenedores son en sí mismos componentes y como tales pueden ser situados dentro de otros contenedores. También contienen el código necesario para el control de eventos, cambiar la forma del cursor o modificar el icono de la aplicación. En el AWT todos los contenedores son instancias de la clase **Container** o uno de sus subtipos.

En la figura 13.1 se reproduce la ventana generada por el código de la aplicación del ejemplo `Java1301.java` que muestra todos los componentes que proporciona el AWT. Posteriormente se verán en detalle estos componentes, pero aquí se puede ya observar la estética que presentan en su conjunto. La ventana es necesaria porque el programa incluye un menú, y los menús solamente pueden utilizarse en ventanas. El código contiene un método *main()* para poder ejecutarlo como una aplicación independiente.



Figura 13.1

COMPONENTES

La clase **Component** es una clase abstracta que representa todo lo que tiene una posición, un tamaño, puede ser pintado en pantalla y puede recibir eventos.

No tiene constructores públicos, ni puede ser instanciada. Sin embargo, la clase **Component** puede ser extendida para proporcionar una nueva característica incorporada a Java, conocida como componentes ligeros o *Lightweight*.

Los objetos derivados de la clase **Component** que se incluyen en el *Abstract Window Toolkit* son los que aparecen a continuación:

Button
Canvas
Checkbox
Choice
Container
  Panel
  Window
    Dialog
    Frame
Label
List
Scrollbar
TextComponent
  TextArea
  TextField

Sobre estos componentes se podrían hacer más agrupaciones y quizá la más significativa fuese la que diferencie a los componentes según el tipo de entrada. Así habría componentes con entrada de tipo *no-textual* como los botones de pulsación (**Button**), las listas (**List**), botones de marcación (**Checkbox**), botones de selección (**Choice**) y botones de comprobación (**CheckboxGroup**); componentes de *entrada y salida textual* como los campos de texto (**TextField**), las áreas de texto (**TextArea**) y las etiquetas (**Label**); y, otros componentes *sin acomodo fijo* en ningún lado, en donde se encontrarían componentes como las barras de desplazamiento (**Scrollbar**), zonas de dibujo (**Canvas**) e incluso los contenedores (**Panel**, **Window**, **Dialog** y **Frame**), que también pueden considerarse como componentes.

Botones de pulsación

Los botones de pulsación (***Button***) se utilizan en la mayoría de los ejemplos de este Tutorial, aunque nunca se han considerado sus atributos específicamente.

La clase **Button** es aquella que produce un componente de tipo botón con un título. El constructor más utilizado es el que permite pasarle como parámetro una cadena, que será la que aparezca como título e identificador del botón en la interfaz de usuario. No dispone de campos o variables de instancia y pone al alcance del programador una serie de métodos entre los que destacan por su utilidad los siguientes:

`addActionListener()`, añade un receptor de eventos de tipo **Action** producidos por el botón.

`getLabel()`, devuelve la etiqueta o título del botón.

`removeActionListener()`, elimina el receptor de eventos para que el botón deje de realizar acción alguna.

`setLabel()`, fija el título o etiqueta visual del botón.

Además, dispone también de todos los métodos heredados de las clases **Component** y **Object**.

El evento **ActionListener** es un evento de tipo semántico. Un evento de tipo **Action** se produce cuando el usuario pulsa sobre un objeto **Button**. Además, un objeto **Button** puede generar eventos de bajo nivel de tipo **FocusListener**, **MouseListener** o **KeyListener**, porque hereda los métodos de la clase **Component** que permiten instanciar y registrar receptores de eventos de este tipo sobre objetos de tipo **Button**.

Botones de selección

Los botones de selección (***Choice***) permiten el rápido acceso a una lista de elementos, presentándose como título el ítem que se encuentre seleccionado.

La clase **Choice** extiende la clase **Component** e implementa la interfaz **ItemSelectable**, que es aquella que mantiene un conjunto de ítems en los que puede haber, o no, alguno seleccionado. Además, esta clase proporciona el método *addItemListener()*, que añade un registro de eventos de las opciones seleccionadas, muy importante a la hora de tratar los eventos que se producen sobre los objetos de tipo **Choice**.

La clase **Choice** también dispone de cerca de una veintena de métodos que permiten la manipulación de los ítems disponibles para la selección.



Figura 13.2

El ejemplo que se ha implementado, `Java1302.java`, para ilustrar el uso de los botones de selección, coloca un objeto de tipo **Choice** sobre un objeto **Frame** y añade tres objetos de tipo **String** al objeto **Choice**, fijando el tercero de ellos como preseleccionado a la hora de lanzar el programa. La presentación inicial de la aplicación al ejecutarla es la que reproduce la figura 13.2.

Se instancia y registra un objeto de tipo **ItemListener** sobre el objeto **Choice** para identificar y presentar el objeto **String** que se elige cuando el usuario utiliza una

selección. Cuando esto ocurre, se captura un evento en el método sobrescrito *itemStateChanged()* del objeto **ItemListener**. El código de este método utiliza una llamada a *getSelectedItem()* para el ítem que está marcado y presentar la cadena que le corresponde.

También se instancia y registra un objeto receptor de eventos *windowClosing()* sobre el **Frame** para concluir el programa cuando el usuario cierre la ventana.

Los trozos de código que merecen un repaso son los que se ven a continuación. Empezando por las sentencias siguientes, que son las típicas usadas en la instanciación del objeto **Choice** y la incorporación de objetos **String** a ese objeto **Choice**.

```
Choice miChoice = new Choice();
miChoice.add( "Primer Choice" );
. . .
```

La línea de código siguiente hace que el objeto *Tercer Choice* de tipo **String** sea el que se encuentre visible al lanzar la aplicación.

```
miChoice.select( "Tercera Opcion" );
```

La sentencia que se reproduce ahora es la que instancia y registra un objeto de tipo **ItemListener** sobre el objeto **Choice**.

```
miChoice.addItemListener( new MiItemListener( miChoice ) );
```

A esta sentencia le sigue el código que crea el objeto **Frame**, coloca el objeto **Choice** en él, etc. Ya se han visto varias veces sentencias de este tipo, así que no merece la pena volver sobre ellas. Solamente queda como código interesante en el programa el método *itemStateChanged()*, que está sobrescrito, del objeto **ItemListener**.

```
public void itemStateChanged( ItemEvent evt ) {
  System.out.println( oChoice.getSelectedItem() );
  }
```

Eventos de este tipo se producen siempre que el usuario realiza una selección (abre la lista de opciones y pulsa el botón del ratón con el cursor sobre una de ellas). Y, como se puede ver, el método *getSelectedItem()* es el encargado de obtener la representación en cadena del ítem que estaba seleccionado en ese momento.

Botones de comprobación

La clase **CheckBox** extiende la clase **Component** e implementa la interfaz **ItemSelectable**, que es aquella que contiene un conjunto de ítems entre los que puede haber o no alguno seleccionado.

Los botones de comprobación (***Checkbox***) se pueden agrupar para formar una interfaz de botón de radio (***CheckboxGroup***), que son agrupaciones de botones de

comprobación de exclusión múltiple, es decir, en las que siempre hay un único botón activo.

La programación de objetos **Checkbox** puede ser simple o complicada, dependiendo de lo que se intente conseguir. La forma más simple para procesar objetos **Checkbox** es colocarlos en un **CheckboxGroup**, ignorar todos los eventos que se generen cuando el usuario seleccione botones individualmente y, luego, procesar sólo el evento de tipo **Action** cuando el usuario fije su selección y pulse un botón de confirmación. Hay gran cantidad de programas, fundamentalmente para *Windows*, que están diseñados en base a este funcionamiento.

La otra forma, más compleja, para procesar información de objetos **Checkbox** es responder a los diferentes tipos de eventos que se generan cuando el usuario selecciona objetos **Checkbox** distintos. Actualmente, no es demasiado complicado responder a estos eventos, porque se pueden instanciar objetos **Listener** y registrarlos sobre los objetos **Checkbox**, y eso no es difícil. La parte compleja es la implementación de la lógica necesaria para dar sentido a las acciones del usuario, especialmente cuando ese usuario no tiene clara la selección que va a realizar y cambia continuamente de opinión.

El ejemplo `Java1303.java` utiliza la solución simple, permitiendo que el usuario cambie de opción cuantas veces quiera y tome una decisión final pulsando sobre el botón "*Aceptar*". Se colocan cuatro objetos **Checkbox**, definidos sobre un objeto **CheckboxGroup**, y un objeto **Button** sobre un objeto **Frame**. La apariencia de estos elementos en pantalla es la que muestra la figura 13.3.



Figura 13.3

Se instancia y registra un objeto de tipo **ActionListener** sobre el objeto **Button**. La acción de seleccionar y deseleccionar un objeto **Checkbox** es controlada automáticamente por el sistema, al formar parte de un **CheckboxGroup**. Cada vez que se pulse el botón, se generará un evento que al ser procesado por el método sobrescrito *actionPerformed()*, determina y presenta en pantalla la identificación del botón de comprobación que está seleccionado. En la ventana se muestra tanto la identificación asignada por el sistema a los botones, como la etiqueta que es asignada directamente por el programa, demostrando que tanto una como otra se pueden utilizar para identificar el botón seleccionado.

```
// Esta clase indica la caja de selección que está seleccionada
// cuando se pulsa el botón de Aceptar
class MiActionListener implements ActionListener {
  CheckboxGroup oCheckBoxGroup;
```

```
  MiActionListener( CheckboxGroup checkBGroup ) {
    oCheckBoxGroup = checkBGroup;
    }
  public void actionPerformed( ActionEvent evt ) {
    System.out.println(oCheckBoxGroup.getSelectedCheckbox().getName()+
      " " + oCheckBoxGroup.getSelectedCheckbox().getLabel() );
    }
  }
```

Hay cuatro trozos interesantes de código en este ejemplo que merecen atención especial. El primero es la creación del objeto **CheckboxGroup**, que posteriormente se utilizará para englobar los cuatro objetos **Checkbox** en un solo grupo lógico.

```
CheckboxGroup miCheckboxGroup = new CheckboxGroup();
```

Luego se crean el botón "*Aceptar*" y el objeto **ActionListener** que se va a registrar sobre él.

```
Button miBoton = new Button( "Aceptar" );
miBoton.addActionListener( new MiActionListener( miCheckboxGroup ) );
```

También está la instanciación de los cuatro objetos **Checkbox** como parte del grupo **CheckboxGroup** y su incorporación al **Frame**.

```
miFrame.add( new Checkbox( "A",true,miCheckboxGroup ) );
```

Y, finalmente, es interesante el fragmento de código en el que se sobrescribe el método *actionPerformed()* en el objeto **ActionListener** que extrae la identificación del objeto **Checkbox**, que se encuentra seleccionado. Aquí, el método *getSelectedCheckbox()* es un método de la clase **CheckboxGroup**, que devuelve un objeto de tipo **Checkbox**. El método *getLabel()* es miembro de la clase **Checkbox**, y el método *getName()* es miembro de la clase **Component**, que es una superclase de **Checkbox**.

```
public void actionPerformed( ActionEvent evt ) {
  System.out.println(oCheckBoxGroup.getSelectedCheckbox().getName()+
    " " + oCheckBoxGroup.getSelectedCheckbox().getLabel() );
  }
```

Listas

Las listas (***List***) aparecen en las interfaces de usuario para facilitar a los operadores la manipulación de muchos elementos. Se crean utilizando métodos similares a los implementados en los botones **Choice**. La lista es visible todo el tiempo, utilizándose una barra de desplazamiento para visualizar los elementos que no caben en el área de pantalla en que se está presentando la lista.

La clase **List** extiende la clase **Component** e implementa la interfaz **ItemSelectable**. Además, soporta el método *addActionListener()* que se utiliza para

capturar los eventos **ActionEvent** que se producen cuando el usuario pulsa dos veces con el ratón sobre un elemento de la lista.

El ejemplo `Java1304.java`, que ilustra el empleo de las listas, coloca un objeto **List** y un objeto **Button** sobre un objeto **Frame**. Se añaden quince objetos **String** a la lista y se deja el segundo como seleccionado inicialmente. La apariencia de la ventana así construida, al inicializar el programa, es la que se reproduce en la figura 13.4.



Figura 13.4

Sobre la lista se instancia y registra un objeto de tipo **ActionListener**, cuyo propósito es identificar y presentar en pantalla el objeto **String**, que el usuario ha seleccionado haciendo doble *clic* sobre él. Cuando selecciona y luego pulsa dos veces con el ratón sobre un elemento de la lista, el evento es capturado por el método sobrescrito *actionPerformed()* del objeto **ActionListener**. El código de este método utiliza la llamada a *getSelectedItem()* de la clase **List**, para identificar y presentar la cadena que corresponde al elemento seleccionado. Sin embargo, si el usuario realiza una doble pulsación con el ratón sobre un elemento, mientras otro se encuentra seleccionado, el método *getSelectedItem()* devuelve `null` y no aparecerá nada en pantalla.

Al objeto **Frame** también se le incorpora un botón para permitir realizar selección múltiple en la lista; de tal forma que cuando se pulsa, se captura el evento **ActionListener** que genera y presenta en pantalla los elementos que se encuentren seleccionados en ese momento, incluso aunque sólo haya uno de ellos.

```
class IHM {
  public IHM(){
    // Instancia un objeto List y coloca algunas cadenas sobre él,
    // para poder realizar selecciones
    List miLista = new List();
    for ( int i=0; i < 15; i++ )
      miLista.add( "Elemento "+i );
    // Activa la selección múltiple
    miLista.setMultipleMode( true );
    // Presenta el elemento 1 al inicio (la cuenta empieza en 0)
    miLista.select( 1 );

    // Instancia y registra un objeto ActionListener sobre el objeto
    // List. Se produce un evento de tipo Action cuando el usuario
    // pulsa dos veces sobre un elemento
    miLista.addActionListener( new MiListaActionListener( miLista ) );
    // Instancia un objeto Button para servicio de la selección
```

```
    // múltiple. También instancia y registra un objeto ActionListener
    // sobre el botón
    Button miBoton = new Button( "Selecciona Multiples Items" );
    miBoton.addActionListener( new miBotonActionListener( miLista ) );
    // Coloca el objeto List y el objeto Button en el objeto Frame
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );
    miFrame.setLayout( new FlowLayout() );
    miFrame.add( miLista );
    miFrame.add( miBoton );
    miFrame.setSize( 250,150 );
    miFrame.setVisible( true );
    // Instancia y registra un objeto receptor de eventos de ventana
    // para concluir la ejecución del programa cuando el Frame se
    // cierre por acción del usuario sobre él
    miFrame.addWindowListener( new Conclusion() );
    }
  }

// Clase para recibir eventos de tipo Action sobre el objeto List.
// Presenta el elemento seleccionado cuando el usuario pulsa dos veces
// sobre un ítem de la lista si la selección es individual.
// Si el usuario pulsa dos veces sobre una selección múltiple
// se produce un evento, pero el método getSelectedItem() de
// la clase List devuelve null y no se presenta nada en pantalla
class MiListaActionListener implements ActionListener {
  List oLista;
  MiListaActionListener( List lista ) {
    // Salva una referencia al objeto List
    oLista = lista;
    }

  // Sobrescribe el método actionPerformed() de la interfaz
  // ActionListener
  public void actionPerformed( ActionEvent evt ) {
    if ( oLista.getSelectedItem() != null ) {
      System.out.println( "Seleccion Simple de Elementos" );
      System.out.println( "  "+oLista.getSelectedItem() );
      }
    }
  }

// Clase para capturar los eventos Action que se produzcan sobre el
// objeto Button. Presenta los elementos que haya seleccionados
// cuando el usuario lo pulsa, incluso aunque solamente haya uno
// marcado. Si no hubiese ninguno, no se presentaría nada en la
// pantalla
class miBotonActionListener implements ActionListener {
  List oLista;
  miBotonActionListener( List lista ) {
    // Salva una referencia al objeto List
    oLista = lista;
    }

  // Sobrescribe el método actionPerformed() de la interfaz
  //  ActionListener
  public void actionPerformed( ActionEvent evt ) {
    String cadena[] = oLista.getSelectedItems();
    if ( cadena.length != 0 ) {
```

```
      System.out.println( "Seleccion Multiple de Elementos" );
      for ( int i=0; i < cadena.length; i++ )
        System.out.println( "  "+cadena[i] );
      }
    }
  }
```

Si se realiza una revisión del código del ejemplo, se encontrarán algunos fragmentos que merecen una reflexión. En el primero de ellos, reproducido en las líneas siguientes, se instancia un objeto **List** y se rellenan quince cadenas. La lista se define como una selección múltiple y se fija el segundo elemento como el inicialmente seleccionado en el momento del arranque del programa.

```
List miLista = new List();
for( int i=0; i < 15; i++ )
  miLista.add( "Elemento "+i );
miLista.setMultipleMode( true );
miLista.select( 1 );
```

La sentencia siguiente es la que instancia y registra un objeto de tipo **ActionListener** sobre la lista, para que responda a la doble pulsación del ratón sobre uno de los elementos de esa lista.

```
miLista.addActionListener( new MiListaActionListener( miLista ) );
```

Luego ya se encuentra el código que instancia un objeto **Button**, que instancia y registra un objeto **ActionListener** sobre ese botón, y que lo incorpora al **Frame**.

Otro fragmento interesante corresponde a la clase **MiListaActionListener**, que está diseñada para responder cuando el usuario pulse dos veces con el ratón sobre un ítem seleccionado de la lista. Si el usuario pulsa dos veces sobre una única selección de la lista, el método sobrescrito *actionPerformed()* identificará el elemento a través de una llamada al método *getSelectedItem()* y lo presentará en pantalla. Sin embargo, si se pulsa dos veces cuando hay más de un elemento seleccionado, el método *getSelectedItem()* devolverá `null` y el elemento será ignorado.

```
public void actionPerformed( ActionEvent evt ) {
  if( aLista.getSelectedItem() != null ) {
    System.out.println( "Seleccion Simple de Elementos" );
    System.out.println( "  "+oLista.getSelectedItem() );
    }
  }
```

Las líneas de código que se reproducen a continuación constituyen el código que responde a un evento **Action** producido sobre el objeto **Button**, y presenta en pantalla uno, o más, elementos seleccionados de la lista. En este caso, el método empleado es *getSelectedItem()*, para crear un array que mantenga todos los elementos que estén seleccionados en el momento de pulsar el botón y que luego se presenta en pantalla.

```
public void actionPerformed( ActionEvent evt ) {
  String cadena[] = oLista.getSelectedItems();
  if( cadena.length != 0 ) {
```

```
    System.out.println( "Seleccion Multiple de Elementos" );
    for( int i=0; i < cadena.length; i++ )
      System.out.println( "  "+cadena[i] );
      }
    }
  }
```

Campos de texto

Para la entrada directa de datos se suelen utilizar los campos de texto, que se presentan visualmente en pantalla como pequeñas cajas que permiten al usuario la entrada por teclado de una línea de caracteres.

Los campos de texto (***TextField***) son los encargados de realizar esta entrada, aunque también se pueden utilizar, activando su indicador de no-editable, para presentar texto en una sola línea con una apariencia en pantalla más llamativa, debido al borde simulando 3D que acompaña a este tipo de elementos de la interfaz gráfica.

La clase **TextField** extiende a la clase **TextComponent**, que extiende, a su vez, a la clase **Component**. Por ello, hay una gran cantidad de métodos que están accesibles desde los campos de texto. La clase **TextComponent** también es importante en las áreas de texto, en donde se permite la entrada de múltiples líneas de texto.

La clase **TextComponent** es un componente que permite la edición de texto. Tiene un campo y no dispone de constructores públicos, por lo que no es posible instanciar objetos de esta clase. Sin embargo, sí dispone de un amplio repertorio de métodos que son heredados por sus subclases que permiten la manipulación del texto. Entre esos métodos hay algunos muy interesantes, como son los que permiten la selección o recuperación del texto marcado desde el programa; la indicación de editabilidad de texto; la recuperación de los eventos producidos por ese componente, etcétera.

En el programa `Java1305.java`, que se implementa para ilustrar el uso de los campos de texto, se coloca un objeto **TextField** sobre un objeto **Frame**, inicializándolo con la cadena "Texto inicial", que aparecerá en el campo de texto, tal como aparece reproducido en la figura 13.5.



Figura 13.5

Sobre el objeto **TextField** se instancia y registra un objeto de tipo **ActionListener**. Cuando se produce un evento porque el usuario haya pulsado la tecla *Retorno* mientras el objeto **TextField** tiene el foco del teclado, éste es recibido por el receptor de eventos, que en este ejemplo concreto, extrae y presenta en pantalla, en el **TextField**, el texto en dos formas: presentando todo el texto y también, presentando solamente el trozo de texto que esté seleccionado.

Cuando se pulsa la tecla *Retorno* mientras el campo de texto tiene el foco, el evento es capturado por el método sobrescrito *actionPerformed()* del objeto **ActionListener**. El código de este método utiliza el método *getSelectedItem()* de la clase **TextField** para acceder y presentar el texto que haya seleccionado el usuario. El código de este método también invoca al método *getText()* de la clase **TextComponent**, para presentar el contenido completo del campo de texto.

```
class IHM {
  public IHM() {
    // Instancia un objeto TextField y coloca una cadena como Texto
    // para que aparezca en el momento de su creación
    TextField miCampoTexto = new TextField( "Texto inicial" );
    // Instancia y registra un receptor de eventos de tipo Action
    // sobre el campo de texto
    miCampoTexto.addActionListener(
      new MiActionListener( miCampoTexto ) );
    // Coloca el campo de texto sobre el objeto Frame
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );
    miFrame.setLayout( new FlowLayout() );
    miFrame.add( miCampoTexto );
    miFrame.setSize( 250,150 );
    miFrame.setVisible( true );
    // Instancia y registra un objeto receptor de eventos de ventana
    // para concluir la ejecución del programa cuando el Frame se
    // cierre por acción del usuario sobre él
    miFrame.addWindowListener( new Conclusion() );
    }
  }

// Clase para recibir los eventos de tipo Action que se produzcan
// sobre el objeto TextField sobre el cual se encuentra registrado
class MiActionListener implements ActionListener {
  TextField oCampoTexto;
  MiActionListener( TextField iCampoTexto ) {
    // Guarda una referencia al objeto TextField
    oCampoTexto = iCampoTexto;
    }

  // Se sobrescribe el método actionPerformed() de la interfaz
  // ActionListener para que indique en la consola el texto que
  // se introduce
  public void actionPerformed( ActionEvent evt ) {
    System.out.println( "Texto seleccionado: " +
      oCampoTexto.getSelectedText() );
    System.out.println( "Texto completo: " +
      oCampoTexto.getText() );
    }
```

```
}
```

Las sentencias que se vuelven a reproducir a continuación instancian un objeto **TextField**, inicializándolo con una cadena y, luego, un objeto **ActionListener** es instanciado y registrado sobre el objeto **TextField**.

```
TextField miCampoTexto = new TextField( "Texto inicial" );
miCampoTexto.addActionListener(new MiActionListener( miCampoTexto ) );
```

Hay más sentencias de código que son semejantes a las ya vistas en ejemplos anteriores, así que solamente es necesario recabar atención sobre el método *actionPerformed()*, en donde se invocan los métodos *getSelectedText()* y *getText()* para recuperar y presentar el texto.

```
public void actionPerformed( ActionEvent evt ) {
  System.out.println( "Texto seleccionado: " +
    oCampoTexto.getSelectedText() );
  System.out.println( "Texto completo: " +
    oCampoTexto.getText() );
  }
```

Áreas de texto

Un área de texto (***TextArea***) es una zona multilínea que permite la presentación de texto, que puede ser editable o de sólo lectura. Al igual que la clase **TextField**, esta clase extiende la clase **TextComponent** y dispone de cuatro campos, que son constantes simbólicas que pueden ser utilizadas para especificar la colocación de las barras de desplazamiento en algunos de los constructores de objetos **TextArea**. Estas constantes simbólicas son:

`SCROLLBARS_BOTH`	que crea y presenta barras de desplazamiento horizontal y vertical
`SCROLLBARS_NONE`	que no presenta barras de desplazamiento
`SCROLLBARS_HORIZONTAL_ONLY`	que crea y presenta solamente barras de desplazamiento horizontal
`SCROLLBARS_VERTICAL_ONLY`	que crea y presenta solamente barras de desplazamiento vertical

Esta clase **TextArea** contiene muchos métodos y, además, hay que tener en cuenta que hereda métodos definidos en las clases **TextComponent**, **Component** y **Object**, por lo que no queda más remedio que recurrir a la documentación del API que proporciona *Sun* para tener cumplida referencia de cada uno de ellos.

En el ejemplo `Java1306.java`, se coloca un objeto **TextArea** sobre un objeto **Frame**. Esta área de texto dispone de una barra de desplazamiento vertical y se instancia inicialmente con una cadena de diez líneas. La figura 13.6 reproduce la apariencia inicial de la ventana generada en el arranque del programa.



Figura 13.6

Sobre el objeto **TextArea** se instancia y registra un objeto **TextListener**, que capturará eventos de tipo **TextEvent**, que se producen siempre que haya un cambio en el valor que contiene en área de texto.

Este tipo de eventos se capturan en el método sobrescrito *textValueChanged()* del objeto **TextListener**, que utiliza el método *getText()* de la clase **TextComponent** para acceder y presentar en pantalla todo el texto del objeto **TextArea**.

En este programa, el procesado del texto se realiza a muy bajo nivel, generándose un evento cada vez que cambie un solo carácter.

```
class IHM {
  public IHM() {
    // Instancia un objeto TextArea, con una barra de desplazamiento
    // vertical y lo inicializa con diez líneas de texto
    TextArea miAreaTexto = new TextArea( "",5,20,
      TextArea.SCROLLBARS_VERTICAL_ONLY );
    for ( int i=0; i < 10; i++ )
      miAreaTexto.append( "linea "+i+"\n" );
    // Instancia y registra un receptor de eventos de tipo Text
    // sobre el área de texto
    miAreaTexto.addTextListener(new MiTextListener( miAreaTexto ) );
    // Coloca el área de texto sobre el objeto Frame
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );
    miFrame.setLayout( new FlowLayout() );
    miFrame.add( miAreaTexto );
    miFrame.setSize( 250,150 );
    miFrame.setVisible( true );
    // Instancia y registra un objeto receptor de eventos de ventana
    // para concluir la ejecución del programa cuando el Frame se
    // cierre por acción del usuario sobre él
    miFrame.addWindowListener( new Conclusion() );
    }
  }

// Clase para recibir los eventos de tipo Text que se produzcan
// sobre el objeto TextArea sobre el cual se encuentra registrado
class MiTextListener implements TextListener {
  TextArea oAreaTexto;
  MiTextListener( TextArea iAreaTexto ) {
    // Guarda una referencia al objeto TextArea
    oAreaTexto = iAreaTexto;
    }
  // Se sobrescribe el método textValueChanged() de la interfaz
```

```
  // TextListener para que indique en la consola el texto que
  // ocupa el área de texto cuando se cambie
  public void textValueChanged( TextEvent evt ) {
    System.out.println( oAreaTexto.getText() );
    }
  }
```

En el ejemplo anterior hay mucho código que ya se ha visto, y algunas líneas nuevas. En primer lugar se encuentra el código que instancia el objeto de la clase **TextArea** con una barra de desplazamiento vertical y diez líneas de texto en ese objeto. Para añadir el texto al área, se utiliza el método *append()* de la clase **TextArea**, aunque hay varios métodos más que también podrían haberse usado.

```
TextArea miAreaTexto = new TextArea( "",5,20,
  TextArea.SCROLLBARS_VERTICAL_ONLY );
for( int i=0; i < 10; i++ )
  miAreaTexto.append( "linea "+i+"\n" );
```

La línea siguiente instancia y registra un objeto **TextListener** sobre el objeto **TextArea**.

```
miAreaTexto.addTextListener( new MiTextListener( miAreaTexto ) );
```

El fragmento de código que se reproduce a continuación corresponde al método *textValueChanged()*, que utiliza el método *getText()* de la clase **TextComponent** para acceder y presentar todo el texto en el objeto **TextArea** siempre que haya un cambio en el valor del texto del objeto.

```
public void textValueChanged( TextEvent evt ) {
  System.out.println( oAreaTexto.getText() );
  }
```

Etiquetas

Una etiqueta (***Label***) proporciona una forma de colocar texto estático en un panel, para mostrar información fija, que no varía (normalmente), al usuario.

La clase **Label** extiende la clase **Component** y dispone de varias constantes que permiten especificar la alineación del texto sobre el objeto **Label**.

El ejemplo `Java1307.java`, cuya imagen inicial al ejecutarlo se reproduce en la figura 13.7, es muy simple y muestra el uso normal de los objetos **Label**.



Figura 13.7

El programa no proporciona ningún control de eventos, porque las etiquetas normalmente no poseen ningún tipo de eventos, aunque hay que recordar que cualquier receptor de eventos de bajo nivel que se pueda registrar sobre la clase **Component**, también se podrá registrar sobre la clase **Label**, al ser ésta una subclase de **Component**.

```
class IHM {
  public IHM(){
    // Instancia un objeto Label con una cadena para inicializarlo y
    // que aparezca como contenido en el momento de su creación
    Label miEtiqueta = new Label( "Texto inicial" );
    // Coloca la etiqueta sobre el objeto Frame
    Frame miFrame = new Frame( "Tutorial de Java, AWT" );
    miFrame.setLayout( new FlowLayout() );
    miFrame.add( miEtiqueta );
    miFrame.setSize( 250,150 );
    miFrame.setVisible( true );
    // Instancia y registra un objeto receptor de eventos de ventana
    // para concluir la ejecución del programa cuando el Frame se
    // cierre por acción del usuario sobre él
    miFrame.addWindowListener( new Conclusion() );
    }
  }
```

Canvas

Una zona de dibujo, o lienzo (***Canvas***), es una zona rectangular vacía de la pantalla sobre la cual una aplicación puede pintar, imitando el lienzo sobre el que un artista plasma su arte, o desde la cual una aplicación puede recuperar eventos producidos por acciones del usuario.

La clase **Canvas** existe para que se obtengan subclases a partir de ella. No hace nada por sí misma, solamente proporciona una forma de implementar componentes propios. Por ejemplo, un *canvas* es útil a la hora de presentar imágenes o gráficos en pantalla, independientemente de que se quiera saber si se producen eventos o no en la zona de presentación.

Cuando se implementa una subclase de la clase **Canvas**, hay que prestar atención en implementar los métodos *minimumSize()* y *preferredSize()* para reflejar adecuadamente el tamaño de *canvas*; porque, en caso contrario, dependiendo del

layout que utilice el contenedor del *canvas*, ese *canvas* puede llegar a ser demasiado pequeño, incluso invisible.

La clase **Canvas** es muy simple, consiste en un solo constructor sin argumentos y dos métodos, que son:

`addNotify()`, crea el observador del *canvas.*

`paint(Graphics)`, repinta el *canvas.*

El ejemplo `Java1308.java` muestra el uso de la clase **Canvas** y cómo se instancian objetos receptores de eventos que pueden manipular los objetos fuente sobre los que están registrados, sin necesidad de pasar referencias a esos objetos fuente a la hora de instanciar los objetos receptor. Por lo tanto, no se utilizan constructores parametrizados en la instanciación de los objetos receptores de eventos en el programa.

La figura 13.8 muestra la primera apariencia en pantalla cuando se ejecuta la aplicación y se pulsa con el ratón sobre una posición dentro del **canvas**. Aparecen cuatro botones, sin funcionalidad alguna, y un objeto **Canvas** verde, sobre el objeto **Frame**; las coordenadas aparecen por haber sido pulsado el botón del ratón. Los botones están ahí simplemente para que se vea que el **Frame** puede contener más cosas que el **Canvas**.

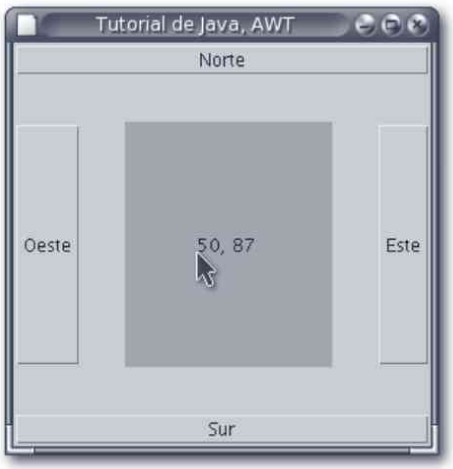


Figura 13.8

Los cuatro botones están colocados en los bordes del **Frame**, utilizando un **BorderLayout** como controlador de posicionamiento. El **Canvas** aparece situado en el centro del **Frame**. Cuando se pulsa el botón del ratón, con el cursor en el interior del *canvas*, aparecerán en pantalla las coordenadas en que se encuentra el ratón en el momento de la pulsación. El origen de los eventos del ratón es el objeto **Canvas**.

No se registran receptores de eventos sobre el **Frame** ni sobre los botones, por lo que si se pulsa sobre éstos, o sobre la zona de separación entre los componentes, no sucederá nada.

```
  public MiCanvas() {
    this.setBackground( Color.green );
    }

  // Se sobrescribe el método paint()
  public void paint( Graphics g ) {
    g.drawString( "" + posicionX + ", " + posicionY,
      posicionX,posicionY );
    }
  }

class Java1308 extends Frame {
  public static void main( String args[] ) {
    // Se instancia un objeto del tipo de la clase
    new Java1308();
    }
```

El primer trozo interesante de código de este ejemplo es el que se utiliza para conseguir una subclase de la clase **Canvas**, para poder sobrescribir el método *paint()* y también hacer que el objeto sea verde desde el primer momento en que es instanciado.

```
class MiCanvas extends Canvas {
  int posicionX;
  int posicionY;

  public MiCanvas() {
    this.setBackground( Color.green );
    }

  // Se sobrescribe el método paint()
  public void paint( Graphics g ) {
    g.drawString( "" + posicionX + ", " + posicionY,
      posicionX,posicionY );
    }
  }
```

El siguiente fragmento interesante de código es el usado para crear un **BorderLayout**, con separaciones (*gaps*) horizontal y vertical, para el objeto **Frame**. Aunque el controlador de posicionamiento por defecto para el **Frame** es precisamente el **BorderLayout**, la indicación de separación entre componentes no está incluida por defecto. Si se desea indicar esta separación, o *gap*, es necesario utilizar un controlador creado exprofeso, es decir, no se puede utilizar un objeto *anónimo*, sino uno con nombre para poder invocar el método que fija la separación entre componentes.

```
BorderLayout miLayout = new BorderLayout();
miLayout.setVgap( 30 );
miLayout.setHgap( 30 );
this.setLayout( miLayout );
```

Otro grupo de sentencias interesantes son las que permiten instanciar el objeto **Canvas** de la clase **MiCanvas**, que extiende a la clase **Canvas** y lo incorpora a un objeto **Frame** en su posición central. El objeto **MiCanvas** no se puede instanciar como objeto anónimo, porque sobre él se va a registrar posteriormente un receptor de eventos del ratón.

```
MiCanvas miObjCanvas = new MiCanvas();
this.add( miObjCanvas,"Center" );
```

Las sentencias anteriores están seguidas de código ya muy utilizado a la hora de incorporar botones a un objeto **Frame** y hacer que se visualicen en pantalla. También está el código que registra el receptor de eventos de la ventana utilizado para concluir la aplicación cuando se cierra el **Frame**.

Así que el siguiente trozo interesante es el fragmento de código que instancia y registra un objeto receptor de eventos que va a procesar los eventos del ratón, para determinar las coordenadas en que se encuentra el cursor en la pantalla, cuando el usuario pulsa el ratón. Este objeto **Listener** es instanciado anónimamente y no pasa referencia alguna del objeto **MiCanvas** al constructor del objeto **Listener**. Por lo tanto, el objeto receptor de eventos debe identificar el componente sobre el cual ha de presentar la información de las coordenadas desde su propio código.

```
miObjCanvas.addMouseListener( new ProcRaton() );
```

Para concluir la revisión del ejemplo, se encuentra el código que define la clase **Listener** que va a presentar las coordenadas del cursor sobre el mismo objeto sobre el cual se ha registrado. Esta versión utiliza el método *getComponent()* sobre el objeto **MouseEvent** que le llega para identificar el componente que ha originado en evento. Este método devuelve una referencia a un objeto de tipo **Component**, luego es necesario hacer un *moldeo* hacia el tipo **MiCanvas** antes de poder acceder a las variables de instancia que se han definido para la clase **MiCanvas**.

```
class ProcRaton extends MouseAdapter {
  // Se sobrescribe el método mousePressed() para que haga lo que
  // se ha indicado
  public void mousePressed( MouseEvent evt ) {
    // Obtiene las coordenadas x e y de la posición del cursor y las
    // almacena en variables de instancia del objeto MiCanvas.
    // Es necesario el casting para poder acceder a las variables
    // de instancia
    ((MiCanvas)evt.getComponent()).posicionX = evt.getX();
    ((MiCanvas)evt.getComponent()).posicionY = evt.getY();
    // Se presentan las coordenadas en pantalla
    evt.getComponent().repaint();
    }
  }
```

El resto del código del programa ya está muy visto y no merece la pena el volver sobre él.

Barra de desplazamiento

Las barras de desplazamiento (***Scrollbar***) se utilizan para permitir realizar ajustes de valores lineales en pantalla, proporcionan una forma de trabajar con rangos de valores o de áreas, como en el caso de un área de texto en donde se proporcionan las barras de desplazamiento de forma automática.



Figura 13.9

El ejemplo `Java1309.java` es muy simple y solamente presenta en pantalla tres barras de desplazamiento que podrían utilizarse, por ejemplo, como selector para fijar un color, en base a sus componentes básicos de rojo, verde y azul. La apariencia en pantalla es la que muestra la figura 13.9, siendo el código de este sencillo programa el que se reproduce en las siguientes líneas.

```
public Java1309() {
  Scrollbar rojo,verde,azul;
  rojo = new Scrollbar( Scrollbar.VERTICAL,0,1,0,255 );
  verde = new Scrollbar( Scrollbar.VERTICAL,0,1,0,255 );
  azul = new Scrollbar( Scrollbar.VERTICAL,0,1,0,255 );
  Frame miFrame = new Frame( "Tutorial de Java, AWT" );
  miFrame.setLayout( new FlowLayout() );
  // Se incorporan las tres barras de desplazamiento al objeto Frame
  miFrame.add( rojo );
  miFrame.add( verde );
  miFrame.add( azul );
  // Se fija el tamaño del Frame y se hace que aparezca todo
  // en pantalla
  miFrame.setSize( 250,100 );
  miFrame.setVisible( true );
  // Se instancia y registra un objeto receptor de eventos de la
  // ventana para poder concluir la aplicación cuando el usuario
  // cierre el Frame
  Conclusion conclusion = new Conclusion();
  miFrame.addWindowListener( conclusion );
  }
}
```

Este tipo de interfaz proporciona al usuario un punto de referencia visual de un rango y al mismo tiempo la forma de cambiar los valores. Por ello, las barras de desplazamiento son componentes un poco más complejos que los demás, reflejándose esta complejidad en sus constructores. Al crearlos hay que indicar su orientación, su valor inicial, los valores mínimo y máximo que puede alcanzar y el porcentaje de rango que estará visible.

Se podría utilizar, por ejemplo, una barra de desplazamiento para seleccionar un rango de valores de color, tal como se hace en el programa `Java1310.java`, en el cual se crea una barra de desplazamiento horizontal y en donde el ancho de esa barra será mayor con relación al **Scrollbar**. La figura 13.10 representa la captura de la ejecución inicial de la aplicación, modificada con las indicaciones explicativas de los valores.



Figura 13.10

Tanto en este ejemplo como en el anterior, no se controlan los eventos generados por la actuación del usuario sobre la barra. El código de este nuevo ejemplo simplemente cambia la declaración e instanciación de las tres barras de desplazamiento de la clase **Java1309** por una barra horizontal de la forma:

```
rango = new Scrollbar( Scrollbar.HORIZONTAL,0,64,0,255 );
```

En este caso, `maxValue` representa el valor máximo que va a alcanzar el lado izquierdo del indicador de la barra. Si se quieren representar 64 valores simultáneamente, es decir, de [0-63] a [192-255], `maxValue` debería ser 192.

El lector habrá observado que las barras de desplazamiento no proporcionan información textual al usuario sobre el valor que está seleccionado, o bien una zona donde poder mostrar directamente los valores asociados a los desplazamientos. Si se desea proporcionar esa información, se ha de proveer explícitamente una caja de texto u otro objeto similar donde presentar esa información, tal como se muestra en el ejemplo `Java1311.java`, cuya imagen en ejecución corresponde a la figura 13.11.



Figura 13.11

El código del ejemplo en este caso atrapa los eventos originados en la barra de desplazamiento. Cada vez que se produce un desplazamiento del indicador de la barra, se genera un evento de tipo ***Ajuste***, que es capturado por el receptor registrado sobre la barra, que a su vez se encarga de presentar el valor numérico correspondiente a la posición actual en el campo de texto utilizado como indicador auxiliar.

CONTENEDORES

La clase **Container** es una clase abstracta derivada de **Component**, que representa a cualquier componente que pueda contener otros componentes. Se trata, en esencia, de añadir a la clase **Component** la funcionalidad de adición, sustracción, recuperación, control y organización de otros componentes.

Al igual que la clase **Component**, no dispone de constructores públicos y, por lo tanto, no se pueden instanciar objetos de la clase **Container**. Sin embargo, sí se puede extender para implementar los componentes *ligeros* o ***Lightweight*** de la plataforma Java 2.

El AWT proporciona varias clases de Contenedores:

Panel
Applet
ScrollPane
Window
Dialog
FileDialog
Frame

Aunque los que se pueden considerar como verdaderos Contenedores son **Window**, **Frame**, **Dialog** y **Panel**, porque los demás son subtipos con algunas características determinadas y solamente útiles en circunstancias muy concretas.

Window

Es una superficie de pantalla de alto nivel (una ventana). Una instancia de la clase **Window** no puede estar enlazada o embebida en otro Contenedor.

El controlador de posicionamiento de componentes por defecto, sobre un objeto **Window**, es el **BorderLayout**.

Una instancia de esta clase no tiene ni título ni borde, así que es un poco difícil de justificar su uso para la construcción directa de una interfaz gráfica, porque es mucho más sencillo utilizar objetos de tipo **Frame** o **Dialog**. Dispone de varios métodos para alterar el tamaño y título de la ventana, o los cursores y barras de menús.

Frame

Es una superficie de pantalla de alto nivel (una ventana) con borde y título. Una instancia de la clase **Frame** puede tener una barra de menú. Una instancia de esta clase es mucho más aparente y más semejante a lo que se entiende por ventana.

Y a no ser que el lector haya comenzado la lectura por esta página, ya se habrá encontrado en varias ocasiones con la clase **Frame**, que es utilizada en gran parte de los ejemplos de este libro. Su uso se debe a la facilidad de su instanciación y, lo que tampoco deja de ser interesante, su facilidad de conclusión.

La clase **Frame** extiende a la clase **Window**, y su controlador de posicionamiento de componentes por defecto es el **BorderLayout**.

Los objetos de tipo **Frame** son capaces de generar varios tipos de eventos, de los cuales el más interesante es el evento de tipo **WindowClosing**, que se utiliza en este *Tutorial* de forma exhaustiva, y que se produce cuando el usuario pulsa sobre el botón de cerrar colocado en la esquina superior-derecha (normalmente) de la barra de título del objeto **Frame**.

En el ejemplo `Java1312.java`, se ilustra el uso de la clase **Frame** y algunos de sus métodos. El programa instancia un objeto **Frame** con tres botones que realizan la acción que se indica en su título. La figura 13.12 reproduce la ventana que genera la aplicación y su situación tras haber pulsado el botón que cambia el cursor de la flecha normal a forma de *mano*.



Figura 13.12

Es un ejemplo muy simple, aunque hay que advertir al lector que se hace uso en él de la sintaxis abreviada de las clases anidadas, creando objetos anónimos. Esta sintaxis se utiliza a la hora de instanciar y registrar receptores de eventos sobre los tres botones, más el de cerrar la ventana, colocados sobre el objeto **Frame**.

El siguiente trozo de código del ejemplo es el típico utilizado en la instanciación de un objeto **Frame**, la indicación del controlador de posicionamiento de componentes que se va a utilizar y, en este caso, la incorporación de los tres componentes de tipo **Button** al objeto **Frame**.

```
Button botonTitulo = new Button( "Imprime Titulo" );
Button botonCursorMano = new Button( "Cursor Mano" );
Button botonCursorFlecha = new Button( "Cursor Flecha" );

miFrame = new Frame( "Tutorial de Java, AWT" );
Image icono =
  Toolkit.getDefaultToolkit().getImage( "muneco.gif" );
miFrame.setIconImage( icono );
miFrame.setLayout( new FlowLayout() );
miFrame.add( botonTitulo );
miFrame.add( botonCursorMano );
miFrame.add( botonCursorFlecha );
```

La línea de código que permite incluir un icono específico para la aplicación

```
miFrame.setIconImage( icono );
```

permite personalizar la imagen que aparece en la esquina superior izquierda de la ventana, tal como se puede observar en la figura 14.12. Para obtener el icono por defecto asociado a una aplicación se puede utilizar la clase **Toolkit**, invocando al método:

```
Toolkit.getDefaultToolkit().getImage( "nombre_icono" );
```

que devuelve un objeto de tipo **Image**.

Para proporcionar un icono a la aplicación, se puede utilizar cualquiera de los siguientes tres métodos:

- Indicar el camino completo de un icono en la línea de comandos de invocación de la aplicación. No obstante, este método puede resultar incómodo al usuario final.

- También se puede indicar el icono como parámetro, de la forma

  ```
  -Dicono=nombre_icono.png
  ```

  y en el código de la aplicación, recuperar el valor del parámetro utilizando las propiedades del sistema, invocando al método *getProperty()* de la forma:

  ```
  System.getProperty( "icono" );
  ```

  Aunque este método también adolece del inconveniente del método anterior, ya que es necesario que el usuario especifique el icono.

- El método más adecuado es utilizar un archivo de propiedades. Esta solución es semejante a la anterior, aunque aquí los valores de las propiedades se proporcionan a través de un archivo de la forma:

  ```
  mi_aplicacion.cfg
  parametro1=valor1
  parametro2=valor2
  icono=nombre_icono.gif
  .  .  .
  parametroN=valorN
  ```

  Para utilizar este archivo es suficiente con crear una instancia del objeto **java.util.Property**, abrir el fichero mediante **java.io.FileInputStream** y cargarlo en el objeto **Property** invocando a su método *local()*. Para obtener el nombre del icono se llama al método *getProperty()* y se le pasa la cadena correspondiente al nombre el parámetro, en este caso "icono".

En los dos bloques de sentencias que se reproducen, se utilizan clases anidadas para instanciar y registrar objetos de tipo **ActionListener**. Por ejemplo, sobre el botón que permite obtener el título de la ventana, se hace tal como se indica.

```
botonTitulo.addActionListener( new ActionListener() {
  public void actionPerformed( ActionEvent evt ) {
```

```
    System.out.println( miFrame.getTitle() );
    }
  } );
```

A partir del JDK 1.4 se introduce, por fin, la posibilidad de crear objetos de tipo **Frame** sin ningún tipo de decoración, a través de la propiedad *undecorated*. Esta propiedad está fijada por defecto a `false` y no es posible alterar su valor una vez que el **Frame** se haya presentado en pantalla. Si se fija a `true` esta propiedad, tanto la barra de título como las demás decoraciones de la ventana desaparecerán, por lo que si es necesario soporte para desplazar la ventana o cerrarla, es responsabilidad del programador incorporar esos elementos.

Cualquier ventana se puede iconificar a través del método *setState()*. Desde el JDK 1.4 se añade el método *setExtendedState()*, que corresponde a una máscara de los cuatro estados posibles y un quinto estado, `MAXIMIZED_BOTH`, que corresponderá a la combinación de los estados de maximización. Los estados posibles son:

`NORMAL`	La ventana a su dimensión normal.
`ICONIFIED`	La ventana iconificada.
`MAXIMIZED_HORIZ`	La ventana maximizada horizontalmente (si el entorno de ventanas lo permite).
`MAXIMIZED_VERT`	La ventana maximizada verticalmente (si el entorno de ventanas lo permite).
`MAXIMIZED_BOTH`	La ventana maximizada, horizontal y verticalmente.

El ejemplo `Java1413.java` utiliza estas características, mostrando una ventana en la que se han eliminado las decoraciones e incorporado botones para ejecutar las acciones descritas en el párrafo anterior.

Dialog

Es una superficie de pantalla de alto nivel (una ventana) con borde y título, que permite entradas al usuario. La clase **Dialog** extiende la clase **Window**, que extiende la clase **Container**, que extiende a la clase **Component**; y el controlador de posicionamiento por defecto es el **BorderLayout**.

De los constructores proporcionados por esta clase, destaca el que permite que el diálogo sea o no *modal*. J2SE 6 proporciona cuatro tipos de ventanas modales, en base a las constantes **Dialog.ModalityType**.

Las ventanas "no modales", que no bloquean a ninguna ventana visible. Este es el modo por defecto, si no se especifica ningún otro tipo de modalidad.

Las ventanas "aplicación-modales", que bloquean a todas las ventanas de la aplicación, excepto a las hijas de la propia ventana. Es decir, solamente se aceptarán

entradas y se podrán crear diálogos desde la propia ventana, los otros diálogos de la aplicación estarán bloqueados.

Las ventanas "documento-modales", que bloquean a todas las ventanas padre que tengan un origen común, pero no bloquean a las ventanas hijas. Este es un modo interesante. Por ejemplo, suponga el lector una aplicación que presenta una ventana emergente de ayuda. Esta ventana es una ventana principal en el momento en que aparece, pero no forma parte de la jerarquía regular de la aplicación, por tanto, se puede hacer *modal*. Además, a partir de la ventana de ayuda se podrían abrir otras ventanas para proporcionar ayuda más detallada, y estas ventanas serían independientes también de la aplicación general. Este funcionamiento se puede implementar fácilmente, porque el tipo de modalidad `DOCUMENT_MODAL` permite que la aplicación principal y la ventana de ayuda tengan jerarquías diferentes.

Las ventanas "toolkit-modales", que bloquean a todas las ventanas del escritorio, aunque pertenezcan a otras aplicaciones. Aquí, la aplicación es el navegador Web. En este caso se puede hacer que un applet en un navegador sea modal, bloqueando a otros applets, impidiendo que acepten entradas. El applet debe tener habilitado el permiso **AWTPermission.toolkitModality** para que `TOOLKIT_MODAL` funcione.

La aplicación `Java1314.java` presenta dos ventanas que utilizan la modalidad de tipo `DOCUMENT_MODAL`. Cada frame tiene un botón que crea un panel que acepta entradas para modificar el rótulo del botón. El lector puede ejecutar el ejemplo para comprobar el funcionamiento de las modalidades.

El lector observará en la ejecución del ejemplo, que puede interaccionar con la ventana de más alto nivel, pero no con el diálogo que queda bajo ella, cuando dicha ventana está presente. Si se cambia el tipo de modalidad a `APPLICATION_MODAL`, para permitir la interacción con las dos ventanas al mismo tiempo, será necesario cerrar el diálogo, porque el cambio de tipo de modalidad en una ventana que está abierta no tiene efecto hasta que se cierra y se vuelve a hacer visible.

Todos los constructores requieren un parámetro **Frame** y, algunos de ellos, permiten la especificación de un parámetro booleano que indica si la ventana que abre el diálogo será modal o no. Si es modal, todas las entradas del usuario serán enviadas a esta ventana, bloqueando cualquier entrada que se pudiese producir sobre otros objetos presentes en la pantalla, pertenezcan o no a la aplicación, en función del tipo de ventana modal creada. Posteriormente, si no se ha especificado que el diálogo sea modal en el constructor, se puede hacer que adquiera esta característica invocando al método *setModal()*.



Figura 13.13

El ejemplo `Java1314.java`, figura 13.13, presenta el mínimo código necesario para conseguir que un objeto **Dialog** aparezca sobre la pantalla. Cuando se arranca el programa, en la pantalla se visualizará un objeto **Frame** y un objeto **Dialog**, que debería tener la mitad de tamaño del objeto **Frame** y contener un título y un botón de cierre no operativo.

El objeto **Dialog** puede ser movido y redimensionado, aunque no se puede ni minimizar ni maximizar. Se puede colocar en cualquier lugar de la pantalla, su posición no está restringida al interior del padre, el objeto **Frame**.

El objeto **Dialog** difiere significativamente en apariencia del objeto **Frame**, sobre todo por la presencia del borde en este último, circunstancia que llama mucho la atención.

```
// Constructor
public Java1314() {
  setTitle( "Tutorial de Java, AWT" );
  setSize( 250,150 );
  setVisible( true );
  Dialog miDialogo = new Dialog( this,"Dialogo" );
  miDialogo.setSize( 125,75 );
  // Hace que el diálogo aparezca en la pantalla
  miDialogo.setVisible( trae );
  }
}
```

La parte más interesante del ejemplo reside en las tres últimas sentencias. La primera instancia el objeto **Dialog** como hijo del objeto principal, **this**. La segunda sentencia establece el tamaño inicial del objeto **Dialog**. La tercera sentencia hace que el objeto **Dialog** aparezca en la pantalla.

El ejemplo `Java1315.java` está diseñado para producir dos objetos **Dialog**, uno *modal* y otro *no-modal*. Un objeto **Frame** sirve de padre a los dos objetos **Dialog**.

El **Dialog** *no-modal* se crea con un botón que sirve para cerrarlo. Sobre este botón se instancia y registra un objeto **ActionListener**. Este objeto **ActionListener** es instanciado desde una clase **ActionListener** compartida. El código del método sobrescrito *actionPerformed()* de la clase **ActionListener** cierra el objeto **Dialog**,

invocando al método *setVisible()* con el parámetro `false`, aunque también se podría haber utilizado *hide()* o *dispose()*.

El **Dialog** *modal* se crea de la misma forma, conteniendo un botón al que se asigna un cometido semejante.



Figura 13.14

Sobre el objeto **Frame** se crean dos botones adicionales, uno mostrará el diálogo *modal* y el otro mostrará el diálogo *no-modal*. Estos dos objetos **Button** comparten una clase **ActionListener** que está diseñada para mostrar un objeto **Dialog** de un tamaño predeterminado y en una posición parametrizada, controlada a través del parámetro `offset`, que se pasa al objeto **ActionListener** cuando se instancia.

La figura 13.14 muestra la ventana que genera la aplicación cuando se ejecuta por primera vez y se selecciona el *Diálogo No-Modal*, pulsando el botón correspondiente.

Si se compila y ejecuta el programa, aparecerán los dos botones en la pantalla, tal como muestra la figura 13.14. Uno de los botones puede ser utilizado para mostrar el objeto **Dialog** *no-modal* y el otro para visualizar el **Dialog** *modal*. Cuando el objeto **Dialog** *modal* no está visible, se podrá mostrar y cerrar el objeto **Dialog** *no-modal*, o pulsar en la caja de cierre del **Frame** para terminar la ejecución del programa. Sin embargo, cuando está visible el **Dialog** *modal*, no se podrá realizar ninguna otra acción dentro del programa; el modo de operación que hace que la aplicación se considere como *aplicación modal*.

Un objeto receptor de eventos *windowClosing()* es instanciado y registrado sobre el **Frame** para concluir la ejecución del programa cuando se cierre el **Frame**; sin embargo, el **Frame** no puede cerrarse cuando el objeto **Dialog** *modal* está visible.

A continuación, se comentan los trozos de código más interesantes del ejemplo. El primero de ellos es la sentencia que instancia el objeto **Frame**, que a pesar de ser semejante a muchas de las ya vistas, lo que la hace importante en este programa es el ser padre de los dos objetos **Dialog**, por lo cual es imprescindible que sea instanciado antes de los dos objetos **Dialog**.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
```

El siguiente fragmento interesante es el código utilizado para instanciar los objetos **Dialog**, colocar un objeto **Button** en el objeto **Dialog**, e instanciar y registrar un objeto **ActionListener** sobre el objeto **Button**.

```
Dialog dialogoNoModal = new Dialog( miFrame,"Dialogo No Modal" );
Button botonCerrarNoModal = new Button( "Cerrar" );
dialogoNoModal.add( botonCerrarNoModal );
botonCerrarNoModal.addActionListener(
  new closeDialogListener( dialogoNoModal ) );
```

Éste es el fragmento que crea el objeto **Dialog** *modal*. El que se usa para crear el objeto **Dialog** *no-modal* es esencialmente el mismo excepto en que no tiene el parámetro booleano `true` en la invocación del constructor.

Este código es seguido por el que instancia los objetos **Button** y registra objetos **ActionListener** sobre estos botones. Y, a este código le sigue el que finaliza la construcción del objeto **Frame**.

El trozo de código más interesante de todos es el fragmento en que la clase **ActionListener** es utilizada para instanciar objetos y mostrar un objeto **Dialog**. Es muy interesante por dos aspectos. Uno de ellos es el constructor parametrizado que se utiliza para guardar el desplazamiento, *offset*, que se le pasa como parámetro al objeto **ActionListener**, cuando se instancia. Este valor de *offset* es combinado con valores en el código para controlar el tamaño y la posición del objeto **Dialog** cuando aparece en la pantalla. El otro aspecto interesante es que el método *show()* de la clase **Dialog** es utilizado para hacer aparecer el objeto **Dialog** en la pantalla.

```
class showDialogListener implements ActionListener {
  Dialog oDialog;
  int oOffset;
  showDialogListener( Dialog dialogo,int offset ) {
    oDialog = dialogo;
    oOffset = offset;
    }

  public void actionPerformed( ActionEvent evt ) {
    // Seguir este orden es crítico para un diálogo modal
    oDialog.setBounds( oOffset,oOffset,150,100 );
    oDialog.show();
    }
  }
```

El orden de ejecución de las sentencias dentro del método *actionPerformed()* es crítico. Si el método *show()* se ejecuta antes del método *setBounds()* sobre el objeto **Dialog** *modal*, el método *setBounds()* utilizado para controlar el tamaño y posición del objeto **Dialog**, no tendría efecto alguno. Tamaño y posición del diálogo deben establecerse antes de hacerlo visible.

El último fragmento de código interesante es el método sobrescrito *actionPerformed()* utilizado para cerrar los objetos **Dialog**. En este ejemplo se usa la

llamada al método *setVisible()* con el parámetro `false`, aunque también se podría haber utilizado el método *hide()* o el método *dispose()* con el mismo cometido.

```
public void actionPerformed( ActionEvent evt ) {
  oDialog.setVisible( false );
  }
```

Ventanas modales

El nuevo modelo de gestión de las ventanas modales, proporciona una serie de características nuevas al AWT.

Una de las características es hacer una ventana *siempre visible*. Hace que la ventana de diálogo se encuentre sobre todas las ventanas de la pantalla. No obstante, pueden producirse contradicciones, como en el código siguiente:

```
JFrame jf = new JFrame(...);
jf.setAlwaysOnTop( true );
jf.setVisible( true );
JDialog jd = new JDialog( frame,"Dialogo",true );
jd.setVisible( true );
```

Por un lado, `jd` debe estar encima de `jf`, porque se trata de un diálogo modal y debería bloquear a `jf`. Pero, por otro lado, `jf` debe estar encima de `jd`, porque se le ha fijado la característica de *siempre visible* y a `jd` no. En estos casos, no se puede saber con exactitud qué sucederá, porque el valor relativo de la profundidad, correspondiente a la propiedad *Z-Order*, no está especificado y es dependiente de la plataforma en que se ejecuta la aplicación. Sin embargo, hay un modo de evitar este comportamiento: si `jd` se fija como *siempre visible* también, entonces el diálogo estará encima del *frame*, en cualquier situación.

Otras características corresponden al control de la posición de las ventanas bloqueadas en un diálogo modal, que se puede controlar mediante los métodos de traer al frente, *toFront()*, y enviar atrás, *toBack()*.

Hay algunas características más que el lector puede consultar en el API de J2SE, aunque la mayoría de ellas son dependientes de la plataforma en la que se ejecuta la aplicación, por lo que su comportamiento puede no ser el esperado, según de la plataforma de que se trate.

Panel

La clase **Panel** es un contenedor genérico de componentes. Una instancia de la clase **Panel** simplemente proporciona un contenedor al que ir añadiendo componentes.

El controlador de posicionamiento de componentes sobre un objeto **Panel**, por defecto, es el **FlowLayout**; aunque se puede especificar uno diferente en el

constructor a la hora de instanciar el objeto **Panel**, o bien aceptar el controlador de posicionamiento inicialmente, y después cambiarlo invocando al método *setLayout()*.

Panel dispone de un método *addNotify()*, que se utiliza para crear un observador general (*peer**) del **Panel**. Normalmente, un **Panel** no tiene manifestación visual alguna por sí mismo, aunque puede hacerse notar fijando su color de fondo por defecto a uno diferente del que utiliza normalmente.

El ejemplo `Java1316.java` ilustra la utilización de objetos **Panel** para configurar un objeto de tipo interfaz gráfica, o interfaz hombre-máquina, incorporando tres objetos **Panel** a un objeto **Frame**.

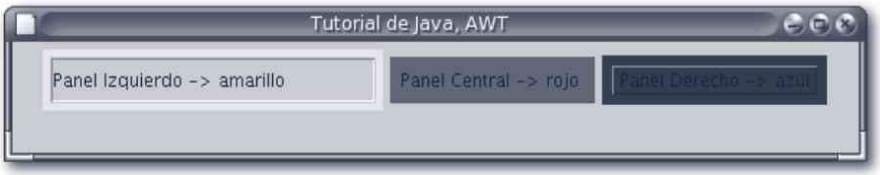


Figura 13.15

El controlador de posicionamiento de los componentes para el objeto **Frame** se especifica concretamente para que sea un **FlowLayout**, y se alteran los colores de fondo de los objetos **Panel** para que sean claramente visibles sobre el **Frame**. Sobre cada uno de los paneles se coloca un objeto, utilizando el método *add()*, de tal modo que se añade un objeto de tipo campo de texto sobre el **Panel** de fondo amarillo, un objeto de tipo etiqueta sobre el **Panel** de fondo rojo y un objeto de tipo botón sobre el **Panel** de fondo azul.

Ninguno de los componentes es activo, ya que no se instancian ni registran objetos receptores de eventos sobre ellos. Así, por ejemplo, el único efecto que se puede observar al pulsar el botón del panel azul, se limita al efecto visual de la pulsación.

Las sentencias de código más interesantes del ejemplo se limitan a la típica instanciación de los tres objetos **Panel**, al control del color de fondo y a la incorporación de otro componente al **Panel**, tal como se reproduce en las siguientes sentencias.

```
Panel panelIzqdo = new Panel();
panelIzqdo.setBackground( Color.yellow );
panelIzqdo.add( new TextField( "Panel Izquierdo -> amarillo" ) );
```

Las siguientes líneas de código instancian un objeto **Frame** y le añaden los tres objetos **Panel** construidos anteriormente.

---

* El autor no ha encontrado una traducción que explique decentemente lo que es un *peer*. Pero podría resumirse en que un *peer* es la manifestación plataforma-dependiente de un objeto plataforma-independiente.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.setLayout( new FlowLayout() );
miFrame.add( panelIzqdo );
miFrame.add( panelCentral );
miFrame.add( panelDrcho );
```

Con este trozo de código se genera el objeto de más alto nivel de la aplicación para la interfaz de usuario.

Añadir componentes a un contenedor

Para que una interfaz sea útil, no debe estar compuesta solamente por Contenedores, éstos deben tener componentes en su interior. Los componentes se añaden al Contenedor invocando al método *add()* del Contenedor. Este método tiene tres formas de llamada que dependen del manejador de composición o *layout manager* que se vaya a utilizar sobre el Contenedor.

En el programa `Java1317.java`, que implementa tanto un applet como una aplicación, se incorporan dos botones a un Contenedor de tipo **Frame**. La creación se realiza en el método *init()* porque éste siempre es llamado automáticamente al inicializarse el applet. De todos modos, al iniciarse la ejecución se crean los botones, ya que el método *init()* es llamado tanto por el navegador como por el método *main()*.

```
public class Java1317 extends java.applet.Applet {
  public void init() {
    add( new Button( "Uno" ) );
    add( new Button( "Dos" ) );
    }

  public static void main( String args[] ) {
    Frame f = new Frame( "Tutorial de Java" );
    Java1317 ejemplo = new Java1317();
    ejemplo.init();
    f.add( "Center",ejemplo );
    f.pack();
    f.setVisible( true );
    }
  }
```

El programa también muestra la forma en que el código puede ejecutarse tanto como aplicación, utilizando el intérprete de Java, como desde un navegador, funcionando como cualquiera de los applets que se han visto en este Tutorial. En ambos casos el resultado, en lo que al ejemplo se refiere, es el mismo: aparecerán dos botones en el campo delimitado por el Contenedor **Frame**.

Los componentes añadidos a un objeto **Container** entran en una lista cuyo orden define el orden en que se van a presentar los componentes sobre el Contenedor, de atrás hacia delante. Si no se especifica ningún índice de orden en el momento de incorporar un componente al Contenedor, ese componente se añadirá al final de la lista. Hay que tener esto muy en cuenta, sobre todo a la hora de construir interfaces de

usuario complejas, en las que pueda haber componentes que solapen a otros componentes o a parte de ellos, en cuyo caso sería necesario recurrir a indicar el orden de profundidad mediante el parámetro `Z-Order` y así indicar la visibilidad de unos componentes respecto a otros.

MENÚS

No hay ninguna regla para diseñar una *buena interfaz de usuario*, todo depende del programador. Los *Menús* son siempre el centro de la aplicación, porque son el medio para que el usuario interactúe con esa aplicación. La diferencia entre una aplicación útil y otra que es totalmente frustrante radica en la organización de los menús; pero lo dicho, las reglas del diseño de un buen árbol de menús no están claras. Hay un montón de libros acerca de la *ergonomía* y de *cómo* se debe implementar la interacción con el usuario. Lo cierto es que por cada uno que defienda una idea, seguro que hay otro que defiende la contraria. Todavía no hay un acuerdo para crear un estándar, con cada *Window Manager* se publica una guía de estilo diferente. Así que, aquí se explica lo básico, sin que se deba tomar como dogma de fe, para que luego cada uno haga lo que mejor le parezca.

En Java, la jerarquía de clases que intervienen en la construcción y manipulación de menús es la que se muestra en la lista siguiente:

```
Java.lang.Object
  MenuShorcut
  java.awt.MenuComponent
    java.awt.MenuBar
    java.awt.MenuItem
      java.awt.Menu
      java.awt.CheckboxMenuItem
      java.awt.PopMenu
```

A continuación, se exploran una a una las clases que se acaban de citar.

Clase MenuComponent

Es la superclase de todos los componentes relacionados con menús. Esta clase no contiene campos, solamente tiene un constructor y dispone de una docena de métodos que están accesibles a todas sus subclases.

Clase Menu

Es un componente de una barra de menú. Ésta es la clase que se utiliza para construir los menús que se manejan habitualmente, conocidos como *menús* desplegables, de persiana o *pull-down*. Dispone de varios constructores para poder, entre otras cosas, crear los menús con o sin etiqueta. No tiene campos y proporciona

varios métodos que se pueden utilizar para crear y mantener los menús en tiempo de ejecución. En la clase **Java1318**, se usan algunos de ellos.

Clase MenuItem

Esta clase se emplea para instanciar los objetos que constituirán los elementos seleccionables del menú, las opciones. No tiene campos y dispone de varios constructores, entre los que hay que citar a:

`MenuItem(String,MenuShortcut)`, que crea un elemento el menú con una combinación de teclas asociada para acceder directamente a él.

Esta clase proporciona una veintena de métodos, entre los que destacan:

`addActionListener(ActionListener)`, que añade el receptor específico que va a recibir eventos desde esa opción del menú.

`removeActionListener(ActionListener)`, contrario al anterior, por lo que ya no se recibirán eventos desde esa opción del menú.

`setEnabled(boolean)`, indica si esa opción del menú puede estar o no seleccionable.

`isEnabled()`, comprobación de si la opción del menú está habilitada.

El método *addActionListener()* ya debería resultar familiar al lector. Cuando se selecciona una opción de un menú, bien a través del ratón o por la combinación rápida de teclas, se genera un evento de tipo **ActionEvent**. Para que la selección de la opción en un menú ejecute una determinada acción, se ha de instanciar y registrar un objeto **ActionListener** que contenga el método *actionPerformed()* sobrescrito para producir la acción deseada. En el ejemplo `Java1418.java`, solamente se presenta en pantalla la identificación de la opción de menú que se ha seleccionado; en un programa realmente útil, la acción seguramente deberá realizar algo más interesante que eso.

Clase MenuShortcut

Esta clase se utiliza para instanciar un objeto que representa un acelerador de teclado, o bien una combinación de teclas rápidas, para un determinado **MenuItem**. No tiene campos y dispone de dos constructores.

Aparentemente, casi todas las teclas rápidas consisten en mantener pulsada la tecla *Control* a la vez que se pulsa cualquier otra tecla. Uno de los constructores de esta clase:

```
menuShortcut( int,boolean );
```

dispone de un segundo parámetro que indica si el usuario ha de mantener también pulsada la tecla de cambio a mayúsculas (*Shift*). El primer parámetro es el código de la

tecla, que es el mismo que se devuelve en el campo keyCode del evento **KeyEvent**, cuando se pulsa una tecla.

La clase **KeyEvent** define varias constantes simbólicas para estos códigos de teclas, como son: VK_8, VK_9, VK_A, VK_B.

Clase MenuBar

Encapsula el concepto de una barra de menú en un **Frame**. No tiene campos, sólo tiene un constructor público, y es la clase que representa el concepto que todo usuario tiene de la barra de menú superior que está presente en la mayoría de las aplicaciones gráficas basadas en ventanas.

El programa Java1318.java ilustra algunos de los aspectos que intervienen en los menús. Es una aplicación que coloca dos menús sobre un objeto Frame. Uno de los menús tiene dos opciones y el otro, tres. La primera opción del primer menú también tiene asignada una combinación de teclas rápidas: *Ctrl+Shift+K* (figura 13.16).

Cuando se selecciona un elemento del menú, éste genera un evento de tipo **ActionEvent**, que presenta una línea de texto en pantalla indicando cuál ha sido el elemento del menú que se ha seleccionado, por ejemplo:

```
% java Java1318
java.awt.MenuItem[menuitem0,label=Primer Elemento del Menu A,
shortcut=Ctrl+ Mayúsculas +K]
java.awt.MenuItem[menuitem1,label=Segundo Elemento del Menu A]
java.awt.MenuItem[menuitem0,label=Primer Elemento del Menu A,
shortcut=Ctrl+Mayúsculas+K]
java.awt.MenuItem[menuitem3,label=Segundo Elemento del Menu B]
```



Figura 13.16

A continuación, se revisan los fragmentos de código más interesantes del ejemplo, aunque en el programa se puede observar que hay gran cantidad de código repetitivo, ya que esencialmente se necesitan las mismas sentencias para crear cada opción del menú y registrar un objeto receptor de eventos sobre cada una de ellas.

El programa genera dos menús separados con el mismo código básicamente. Quizá lo más interesante no sea el código en sí mismo, sino el orden en que se realizan los diferentes pasos de la construcción del menú.

La primera sentencia que merece la pena es la que instancia un objeto **MenuShortcut**, que se utilizará posteriormente en la instanciación de un objeto **MenuItem**.

```
MenuShortcut miAcelerador = new MenuShortcut( KeyEvent.VK_K,true );
```

La lista de argumentos del constructor especifica la combinación de teclas que se van a utilizar y el parámetro `true`, indica que es necesaria la tecla del cambio a mayúsculas (*Shift*) pulsada. El primer parámetro es la constante simbólica que la clase **KeyEvent** define para la tecla **K** y, aparentemente, la tecla *Control* siempre debe estar pulsada en este tipo de combinaciones para ser utilizadas las teclas normales como aceleradores de teclado.

Cuando se utiliza este constructor, en el menú aparecerá la etiqueta asignada y a su derecha, la combinación de teclas alternativas para activarla directamente.

El siguiente fragmento de código que se puede ver es el típico de instanciación de objetos **MenuItem**, que posteriormente se añadirán al objeto **Menu** para crear el menú. Se muestran dos tipos de instrucciones, el primer estilo especifica un acelerador de teclado y el segundo no.

```
MenuItem primerElementoDeA = new MenuItem(
  "Primer Elemento del Menu A",miAcelerador );
MenuItem segundoElementoDeA = new MenuItem(
  "Segundo Elemento del Menu A" );
```

Ahora se encuentra el código que instancia y registra un objeto **ActionListener** sobre la opción del menú, tal como se ha visto en ejemplos anteriores. Aquí solamente se incluye para ilustrar el hecho de que la asociación de objetos **ActionListener** con opciones de un menú, no difiere en absoluto de la asociación de objetos **ActionListener** con objetos **Button**, o cualquier otro tipo de objeto capaz de generar eventos de tipo **ActionEvent**.

```
primerElementoDeA.addActionListener( new MiGestorDeMenu() );
```

Las sentencias que siguen son las ya vistas, empleadas para instanciar cada uno de los dos objetos **Menu** y añadirles las opciones existentes. Es la típica llamada al método *Objeto.add()* que se ha utilizado en programas anteriores.

```
Menu menuA = new Menu( "Menu A" );
menuA.add( primerElementoDeA );
```

El siguiente fragmento de código instancia un objeto **MenuBar** y le añade los dos menús que se han definido antes.

```
MenuBar menuBar = new MenuBar();
menuBar.add( menuA );
menuBar.add( menuB );
```

En este momento ya están creados los dos objetos **Menu** y colocados en un objeto **MenuBar**. Sin embargo, no se ha dicho nada sobre el ensamblado del conjunto. Esto se hace en el momento de asociar el objeto **MenuBar** con el objeto **Frame**. En este caso no se puede utilizar el método *add()*, sino que se tiene que invocar a un método especial de la clase **Frame** que tiene la siguiente declaración:

```
public synchronized void setMenuBar( MenuBar mb )
```

y en este caso concreto se hace en las sentencias que se reproducen seguidamente:

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.setMenuBar( menuBar );
```

Y hasta aquí lo más interesante del ejemplo, porque el código que resta es similar al que ya se ha visto y descrito en otros ejemplos del Tutorial, y que el lector se habrá encontrado si ha seguido la lectura secuencialmente.

Clase CheckboxMenuItem

Esta clase se utiliza para instanciar objetos que puedan utilizarse como opciones en un menú. Al contrario de las opciones de menú que se han descrito al hablar de objetos **MenuItem**, estas opciones tienen mucho más parentesco con las cajas de selección, tal como se podrá comprobar en el ejemplo `Java1319.java`.

Esta clase no tiene campos y proporciona tres constructores públicos, en donde se puede especificar el texto de la opción y el estado en que se encuentra. Si no se indica nada, la opción estará deseleccionada, aunque hay un constructor que permite indicar en un parámetro de tipo booleano que la opción se encuentra seleccionada, o marcada, indicando `true` en ese valor.

De los métodos que proporciona la clase, quizá el más interesante sea el método que tiene la siguiente declaración:

```
addItemListener( ItemListener )
```

Cuando se selecciona una opción del menú, se genera un evento de tipo **ItemEvent**. Para que se produzca la acción que se desea con esa selección, es necesario instanciar y registrar un objeto **ItemListener** que contenga el método *itemStateChanged()* sobrescrito con la acción que se quiere. Por ejemplo, en la clase **Java1319** la acción consistirá en presentar la identificación y estado de la opción de menú que se haya seleccionado.

Cuando se ejecuta el programa, aparece un menú sobre un objeto **Frame** (figura 13.17). El menú contiene tres opciones de tipo **CheckboxMenuItem**. Una opción de este tipo es semejante a cualquier otra, hasta que se selecciona. Cuando se seleccione, aparecerá una marca, o cualquier otra identificación visual, para saber que esa opción

está seleccionada. Estas acciones hacen que el estado de la opción cambie, y ese estado se puede conocer a través del método *getState()*.



Figura 13.17

Cuando se selecciona una opción, se genera un evento de tipo **ItemEvent**, que contiene información del nuevo estado, del texto de la opción y del nombre asignado a la opción. Estos datos pueden utilizarse para identificar cuál de las opciones ha cambiado y, también, para implementar la acción requerida que, en el caso del ejemplo siguiente, consiste en presentar una línea de texto en la pantalla con la información que contiene el objeto **ItemEvent.**

Hay algunos trozos de código en el programa que resultarán repetitivos al lector, debido al hecho de que se utiliza básicamente el mismo código para crear cada elemento del menú y registrar un objeto **Listener** sobre él.

El orden en que se instancian y asocian las opciones es importante. La primera sentencia de código interesante del ejemplo es la que se utiliza para instanciar varios objetos **CheckboxMenuItem** que serán añadidos al objeto **Menu** para producir un menú con varias opciones.

```
CheckboxMenuItem primerElementoMenu =
  new CheckboxMenuItem( "Primer Elemento" );
```

La sentencia que se reproduce ahora es interesante solamente por lo que tiene de novedad, porque hace uso de un tipo de clase **Listener** que no se ha visto antes. La sentencia, por otro lado, es la típica que se requiere para instanciar y registrar objetos **ItemListener** sobre cada **CheckboxMenuItem**.

```
primerElementoMenu.addItemListener( new ControladorCheckBox() );
```

A continuación, se encuentra el código que instancia un objeto **Menu** y le añade los objetos **CheckboxMenuItem**, que es semejante al utilizado en los menús normales. El siguiente código corresponde a la clase que implementa la interfaz **ItemListener**, que vuelve a resultar interesante por lo novedoso.

```
class ControladorCheckBox implements ItemListener {
  public void itemStateChanged( ItemEvent evt ) {
    System.out.println( evt.getSource() );
    }
  }
```

Y el resto del programa es similar al de ejemplos anteriores, así que no se vuelve más sobre él.

Clase PopupMenu

Esta clase se utiliza para instanciar objetos que funcionan como *menús emergentes* o *pop-up*. Una vez que el menú aparece en pantalla, el procesado de las opciones es el mismo que en el caso de los menús desplegables.

Esta clase no tiene campos y proporciona un par de constructores y un par de métodos, de los cuales el más interesante es el método *show()*, que permite mostrar el menú emergente en una posición relativa al componente origen. Este componente origen debe estar contenido dentro de la jerarquía de padres de la clase **PopupMenu**.

El programa `Java1320.java`, coloca un objeto **PopupMenu** sobre un objeto **Frame**. El menú contiene tres opciones de tipo **CheckboxMenuItem** (figura 13.18), y aparece cuando se pulsa dentro del **Frame**, posicionando su esquina superior-izquierda en la posición en que se encontraba el ratón al pulsar el botón.



Figura 13.18

El resto del funcionamiento es semejante al de los programas de ejemplo vistos en secciones anteriores.

El programa es muy similar al de la sección anterior en que se utilizaban objetos **CheckboxMenuItem** en un menú desplegable normal, así que solamente se verá el código que afecta al uso del objeto **PopupMenu**.

El primer fragmento de código interesante es el que instancia un objeto **PopupMenu** y le incorpora tres objetos **CheckboxMenuItem**.

```
PopupMenu miMenuPopup = new PopupMenu( "Menu Popup" );
miMenuPopup.add( primerElementoMenu );
miMenuPopup.add( segundoElementoMenu );
miMenuPopup.add( tercerElementoMenu );
```

El siguiente trozo de código es interesante porque el procedimiento en el caso de asociar un objeto **PopupMenu** con el objeto **Frame**, es diferente a cuando se utiliza un objeto **MenuBar**. En el caso de la barra de menú, se utiliza la llamada al método *setMenuBar()*, mientras que para asociar un menú emergente a un objeto **Frame**, se utiliza la típica llamada de la forma *Objeto.add()*, que es el método habitual de añadir muchos otros componentes a un Contenedor.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.addMouseListener( new ControladorRaton(miFrame,miMenuPopup) );
miFrame.add( miMenuPopup );
```

El siguiente código es el método sobrescrito *mousePressed()*, de la interfaz **MouseListener**, encapsulado aquí en la clase **ControladorRaton**. El propósito de esta clase es instanciar un objeto receptor que atrapará eventos de pulsación del ratón sobre el objeto **Frame**, y mostrará el objeto **PopupMenu** en la posición en que se encuentre el cursor del ratón en el momento de producirse el evento.

```
public void mousePressed( MouseEvent evt ) {
  if( evt.getY() > 0 )
    aMenuPopup.show( aFrame,evt.getX(),evt.getY() );
  }
```

Como se puede observar, es lo típico, excepto el uso del método *show()* de la clase **PopupMenu**. También se debe destacar la referencia al objeto **PopupMenu** y otra al objeto **Frame**, que se pasan cuando se instancia el objeto. Estas dos referencias son necesarias para la invocación del método *show()*, tal como se muestra en el código anterior. La referencia al objeto **Frame** se utiliza para establecer la posición en donde aparecerá el menú, y la referencia al objeto **PopupMenu** especifica el menú que se debe mostrar.

LAYOUTS

Los *layout managers* o *controladores de composición*, en traducción literal, ayudan a adaptar los diversos componentes que se desean incorporar a un **Panel**, es decir, especifican la apariencia que tendrán los componentes a la hora de colocarlos sobre un Contenedor. Java dispone de varios, en la actual versión, tal como se muestra en la figura 13.19.

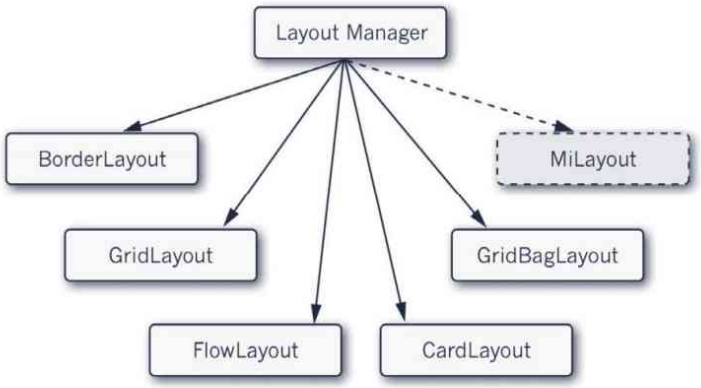


Figura 13.19

¿Por qué Java proporciona estos esquemas predefinidos de disposición de componentes? La razón es simple: imagínese el lector que desea agrupar objetos de distinto tamaño en celdas de una rejilla virtual: si confiado en su conocimiento de un sistema gráfico determinado, codificase a mano tal disposición, debería prever el redimensionamiento del applet, su repintado cuando sea cubierto por otra ventana, etc., además de todas las cuestiones relacionadas con un posible cambio de plataforma (nunca se sabe a dónde van a ir a parar los propios hijos, o los applets).

Imagínese ahora que un hábil equipo de desarrollo ha previsto las disposiciones gráficas más usadas y ha creado un gestor para cada una de tales configuraciones, que se ocupará, de forma transparente para el lector, de todas esas cuitas de formatos. Bien, pues estos gestores son instancias de las distintas clases derivadas de **LayoutManager** y se utilizan en el applet que genera la figura 13.20, donde se muestran los diferentes tipos de *layouts* que proporciona el AWT.



Figura 13.20

El ejemplo `Java1321.java` ilustra el uso de paneles, listas, barras de desplazamiento, botones, selectores, campos de texto, áreas de texto y varios tipos de *layouts*.

En el tratamiento de los **Layouts** se utiliza un *método de validación*, de forma que los componentes son marcados como *no válidos* cuando un cambio de estado afecta a la geometría o cuando el Contenedor tiene un hijo incorporado o eliminado. La validación se realiza automáticamente cuando se llama a *pack()* o *show()*. Los componentes visibles marcados como *no válidos* no se validan automáticamente.

FlowLayout

Es el más simple y el que se utiliza por defecto en todos los Paneles si no se fuerza el uso de alguno de los otros. Los componentes añadidos a un **Panel** con **FlowLayout** se encadenan en forma de lista. La cadena es horizontal, de izquierda a derecha, y se puede seleccionar el espaciado entre cada componente.

Si el Contenedor modifica su tamaño en tiempo de ejecución, las posiciones de los componentes se ajustarán automáticamente, para colocar el máximo número posible de componentes en la primera línea.

Los componentes se alinean según se indique en el constructor. Si no se indica nada, se considerará que los componentes que pueden estar en una misma línea estarán centrados, pero también se puede indicar que se alineen a izquierda o derecha en el Contenedor.

El ejemplo `Java1323.java` lo que hace es colocar cinco objetos **Button**, sin funcionalidad alguna, sobre un objeto **Frame**, utilizando como controlador de

posicionamiento un **FlowLayout**. Al compilar y ejecutar el programa, en pantalla aparecerá inicialmente una ventana como la que se muestra en la figura 13.21.



Figura 13.21

En el programa se añaden cinco botones a un **Frame** utilizando un objeto **FlowLayout** como manejador de posicionamiento de estos botones, fijando una separación de 3 píxeles entre los componentes, tanto en dirección horizontal como vertical.

Se instancia y registra un objeto receptor de eventos de tipo acción para capturar los eventos de los cinco botones. La acción del controlador de eventos es incrementar el espacio entre los componentes en 5 píxeles al pulsar cualquiera de los botones. Esto se consigue incrementando los atributos `Vgap` y `Hgap` del objeto **FlowLayout**, fijando como controlador de posicionamiento el *layout* modificado y validando el **Frame**. Este último paso es imprescindible para que los cambios tengan efecto y se hagan visibles.

Y el código que sigue ya es el programa controlador de eventos, que modifica el *layout* dinámicamente en tiempo de ejecución. Este código responde, cuando se pulsa cualquiera de los botones, utilizando los métodos *get()* y *set()* sobre los atributos de separación vertical, `Vgap`, y horizontal, `Hgap`, del **FlowLayout**; modificando el espaciado entre componentes. Luego, se utiliza el método *setLayout()* para hacer que el objeto **Layout** así modificado sea el controlador de posicionamiento del objeto **Frame**, y, por fin, se hace que los cambios sean efectivos y se visualicen en pantalla validando el objeto **Frame** a través de una llamada al método de validación, *validate()*.

```
public void actionPerformed( ActionEvent evt ){
  miObjLayout.setHgap( miObjLayout.getHgap()+5 );
  miObjLayout.setVgap( miObjLayout.getVgap()+5 );
  miObjFrame.setLayout( miObjLayout );
  miObjFrame.validate();
  }
```

BorderLayout

La composición **BorderLayout** (de borde) proporciona un esquema más complejo de colocación de los componentes en un panel. La composición utiliza cinco zonas para colocar los componentes sobre ellas: Norte, Sur, Este, Oeste y Centro. Es el *layout* o composición que utilizan por defecto **Frame** y **Dialog**.

El Norte ocupa la parte superior del panel, el Este ocupa el lado derecho, Sur la zona inferior y Oeste el lado izquierdo. Centro representa el resto que queda, una vez que se hayan rellenado las otras cuatro partes. Así, este controlador de posicionamiento resuelve los problemas de cambio de plataforma de ejecución de la aplicación, pero limita el número de componentes que pueden ser colocados en un contenedor a *cinco*; aunque, si se va a construir una interfaz gráfica compleja, algunos de estos cinco componentes podrán ser contenedores, con lo cual el número de componentes puede verse ampliado.

En los cuatro lados, los componentes se colocan y redimensionan de acuerdo a sus tamaños preferidos y a los valores de separación que se hayan fijado al contenedor. El tamaño prefijado y el tamaño mínimo son dos informaciones muy importantes en este caso, ya que un botón puede ser redimensionado a proporciones cualesquiera; sin embargo, el diseñador puede fijar un tamaño preferido para la mejor apariencia del botón. El controlador de posicionamiento puede utilizar este tamaño cuando no haya indicaciones de separación en el contenedor, o bien puede ignorarlo, dependiendo del esquema que utilice. Ahora bien, si se coloca una etiqueta en el botón, se podrá indicar un tamaño mínimo de ese botón para que siempre sea visible, al menos, el rótulo del botón. En este caso, el controlador de posicionamiento muestra un total respeto a este valor y garantiza que por lo menos ese espacio estará disponible para el botón.

El ejemplo `Java1324.java` crea una ventana a través de un objeto **Frame** y coloca cinco objetos **Button**, sin funcionalidad alguna, utilizando un **BorderLayout** como manejador de composición. A uno de los botones se le ha colocado un texto más largo, para que el controlador de posicionamiento reserve espacio de acuerdo al tamaño mínimo que se indique para ese botón.

Si se compila y ejecuta este programa, aparecerá en pantalla una imagen como la que se reproduce en la figura 13.22.



Figura 13.22

Se observará que se puede cambiar de tamaño el objeto **Frame**, y que dentro de los límites, los componentes se van modificando a la vez, para acomodarse al tamaño que va adoptando la ventana. Aunque todavía hay problemas que no se han eliminado, y se podrá redimensionar la ventana para conseguir que los botones desaparezcan, o se trunquen los rótulos. Sin embargo, es una forma muy flexible de posicionar componentes en una ventana y no tener que preocuparse de su redimensionamiento y colocación dentro de unos límites normales y sin buscarle las *cosquillas* al sistema.

El ejemplo `Java1325.java`, aunque tampoco hace algo interesante, sí tiene más sustancia que el anterior e ilustra algunos aspectos adicionales del **BorderLayout**. En el programa se añaden cinco botones a un **Frame** utilizando un objeto **BorderLayout** como manejador de posicionamiento de estos botones, fijando una separación de 3 píxeles entre los componentes, tanto en dirección horizontal como vertical.

Se instancia y registra un objeto receptor de eventos de tipo acción para capturar los eventos de los cinco botones. La acción del controlador de eventos es incrementar el espacio entre los componentes en 5 píxeles al pulsar cualquiera de los botones. Esto se consigue incrementando los atributos `Vgap` y `Hgap` del objeto **BorderLayout**, fijando como controlador de posicionamiento el *layout* modificado y validando el **Frame**. Este último paso es imprescindible para que los cambios tengan efecto y se hagan visibles.

Seguidamente se comentan algunas de las sentencias más interesantes que constituyen el código del programa. Por ejemplo, las líneas de código siguientes

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
BorderLayout miBorderLayout = new BorderLayout( 3,3 );
miFrame.setLayout( miBorderLayout );
```

además de instanciar un objeto **Frame**, también instancian un objeto **BorderLayout** con una separación de 3 píxeles entre componentes y establecen ese objeto **BorderLayout** como controlador de posicionamiento de componentes del **Frame**.

```
Button boton1 = new Button( "Sur" );
. . .
miFrame.add( boton1,"South" );
```

Las sentencias anteriores son las utilizadas habitualmente para la instanciación de los objetos **Button** y para añadir estos objetos al objeto **Frame**.

El código que sigue ahora, ya en el controlador de eventos *actionPerformed()* utiliza los métodos *getHgap()*, *getVgap()*, *setHgap()* y *setVgap()* para modificar los atributos de separación entre componentes en el **BorderLayout**. Este **BorderLayout** modificado es utilizado, en conjunción con *setLayout()*, para convertirlo en el controlador de posicionamiento del objeto **Frame**. Y, por último, se utiliza el método *validate()* para forzar al objeto **Frame** a reajustar el tamaño y posición de los componentes y presentar en pantalla la versión modificada de sí mismo.

```
public void actionPerformed( ActionEvent evt ) {
  miObjBorderLayout.setHgap( miObjBorderLayout.getHgap()+5 );
  miObjBorderLayout.setVgap( miObjBorderLayout.getVgap()+5 );
  miObjFrame.setLayout( miObjBorderLayout );
  miObjFrame.validate();
  }
```

GridLayout

La composición **GridLayout** proporciona gran flexibilidad para situar componentes. El controlador de posicionamiento se crea con un determinado número de filas y columnas y los componentes van dentro de las celdas de la tabla así definida.

Si el Contenedor es alterado en su tamaño en tiempo de ejecución, el sistema intentará mantener el mismo número de filas y columnas dentro de los márgenes de separación que se hayan indicado. En este caso, estos márgenes tienen prioridad sobre el tamaño mínimo que se haya indicado para los componentes, por lo que puede llegar a conseguirse que sean de un tamaño tan pequeño que sus etiquetas sean ilegibles.

En el ejemplo `Java1326.java` se muestra el uso de este controlador de posicionamiento de componentes. Además, el programa tiene un cierto incremento de complejidad respecto a los que se han visto hasta ahora para mostrar *layouts*, porque en este caso se crea un objeto para la interfaz de usuario que está compuesto a su vez de más objetos de nivel inferior. Y también se ilustra el proceso de modificar dinámicamente, en tiempo de ejecución, un *layout*.

La interfaz de usuario está constituida por un objeto **Frame** sobre el que se colocan objetos **Panel** utilizando el controlador de posicionamiento por defecto para el **Frame**, **BorderLayout**. Uno de los objetos **Panel** contiene seis objetos **Button** posicionados utilizando un **GridLayout**, y no tienen ninguna funcionalidad asignada, no se registra sobre ellos ningún receptor de eventos. Inicialmente, los botones se colocan en una tabla de dos filas y tres columnas.

El otro objeto **Panel** contiene dos botones con los rótulos *3x2* y *2x3*, que se colocan utilizando un **FlowLayout**. Estos botones sí son funcionales, ya que se registran sobre ellos objetos **ActionListener**. Cuando se pulsa sobre el botón *3x2*, los botones del otro **Panel** se reposicionan en tres filas y dos columnas. De forma semejante, cuando se pulsa sobre el botón *2x3*, los objetos **Button** del otro panel se recolocan en dos filas y tres columnas.

Si se compila y ejecuta la aplicación, inicialmente aparecerá en pantalla una ventana semejante a la que se reproduce en la figura 13.23.

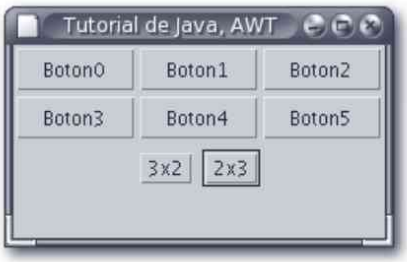


Figura 13.23

A continuación, se comentan algunas de las sentencias de código más interesantes del programa. Para comenzar, se instancian dos objetos **Button** que posteriormente serán los botones funcionales del programa. Estos objetos no pueden ser instanciados anónimamente, precisamente porque sobre ellos se va a registrar un objeto de tipo **ActionListener**, y es necesario que pueda tener una variable de referencia sobre cada objeto para permitir ese registro.

```
Button boton7 = new Button( "3x2" );
Button boton8 = new Button( "2x3" );
```

La siguiente sentencia instancia un objeto **GridLayout** que será utilizado como controlador de posicionamiento para uno de los objetos **Panel**. Tampoco este objeto puede ser anónimo, porque se necesitará acceso a él a la hora de modificar la colocación de los componentes sobre ese **Panel**.

```
GridLayout miGridLayout = new GridLayout( 2,3 );
```

En el siguiente trozo de código se instancia uno de los objetos **Panel** que se integrarán en el objeto **Frame** principal. Se usará el objeto **GridLayout**, instanciado antes, como controlador de posicionamiento en el panel. Y, a continuación, se usa un bucle para colocar los seis objetos **Button** en el **Panel**. En este caso, sí es posible hacer que los objetos **Button** sean anónimos, ya que no se va a registrar sobre ellos ningún tipo de receptor de eventos.

```
Panel panel1 = new Panel();
panel1.setLayout( miGridLayout );
for( int i=0; i < 6; i++)
  panel1.add( new Button( "Boton"+i ) );
```

Ahora se instancia el segundo objeto **Panel**, y se colocan en él los dos botones que se habían instanciado antes, que son los que van a tener funcionalidad. Esto se hace en las siguientes sentencias:

```
Panel panel2 = new Panel();
panel2.add( boton7 );
panel2.add( boton8 );
```

El próximo paso es instanciar un objeto **Frame** que servirá como interfaz. Una vez que se instancie, se colocarán en él los dos objetos **Panel** utilizando su controlador de posicionamiento por defecto, el **BorderLayout**.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.add( panel1,"North" );
miFrame.add( panel2,"South" );
```

En este instante, la creación física de la interfaz está concluida, así que el siguiente paso es instanciar y registrar un objeto anónimo receptor de eventos de tipo **Action**, para procesar los eventos de los dos objetos **Button** que van a permitir cambiar la apariencia de los botones.

```
boton7.addActionListener(
```

```
  new A3x2ActionListener( miGridLayout,miFrame ) );
boton8.addActionListener(
  new A2x3ActionListener( miGridLayout,miFrame ) );
```

El código del controlador de eventos, tal como se puede ver en las sentencias que se reproducen a continuación, hace uso de los métodos de la clase del *Layout* para ejecutar las acciones pertinentes a la recepción de un evento. En este caso, un evento provocado por uno de los objetos **Button** hace que los componentes se coloquen en una malla de 3 filas por 2 columnas; y un evento sobre el otro objeto **Button** hace que se recoloquen en dos filas y tres columnas. El código del otro controlador es muy parecido.

```
public void actionPerformed( ActionEvent evt ) {
  miObjGridLayout.setRows( 3 );
  miObjGridLayout.setColumns( 2 );
  miObjFrame.setLayout( miObjGridLayout );
  miObjFrame.validate();
  }
```

GridBagLayout

Es igual que la composición de **GridLayout**, con la diferencia que los componentes no necesitan tener el mismo tamaño. Es quizá el controlador de posicionamiento más sofisticado de los que actualmente soporta AWT.

A la hora de ponerse a trabajar con este controlador de posicionamiento, hay que asumir el papel de un auténtico aventurero. Parece que la filosofía de la gente de *Sun Microsystems* es que todo debe hacerse en el código. La verdad es que hasta que no haya en Java algo semejante a los *recursos* de X, el trabajo del programador, si quiere prescindir de herramientas de diseño, será un tanto prehistórico en la forma de hacer las cosas, al menos con AWT. En Swing, comprobará el lector que la creación de interfaces de usuario es bastante más sencilla y, además, los entornos integrados como *Eclipse*, *Netbeans* o *Sun Creador Studio*, ya incorporan herramientas de diseño de interfaces.

Si el lector acepta una recomendación, el uso del **GridBagLayout** es algo a evitar, porque tanta sofisticación lo único que acarrea son dolores de cabeza; y, siempre se puede recurrir a la técnica de combinar varios paneles utilizando otros controladores de posicionamiento dentro del mismo programa. Los applets no apreciarán esta diferencia, al menos no tanto como para justificar los problemas que conlleva el uso del **GridBagLayout**.

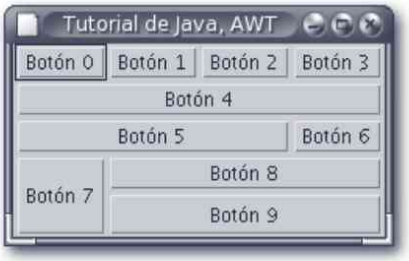


Figura 13.24

A pesar de los pesares, se ha implementado como muestra el ejemplo `Java1327.java`, un programa que presenta diez botones en pantalla, con la apariencia que muestra la figura 13.24.

Para aprovechar de verdad todas las posibilidades que ofrece este *layout*, hay que dibujar antes en papel cómo van a estar posicionados los componentes; utilizar `gridx`, `gridy`, `gridwidth` y `gridheight` en vez de `GridBagConstraints.RELATIVE`, porque en el proceso de validación del *layout* pueden quedar todos los componentes en posición indeseable. Además, se deberían crear métodos de conveniencia para hacer más fácil el posicionamiento de los componentes.

CardLayout

Éste es el tipo de composición que se utiliza cuando se necesita una zona de la ventana que permita colocar distintos componentes sobre ella. Este *layout* suele ir asociado con botones de selección (**Choice**), de tal modo que cada selección determina el panel (grupo de componentes) que se presentará.

En el ejemplo `Java1328.java`, con el que se ilustra el uso de este tipo de controlador de posicionamiento, se crea un objeto interfaz de usuario basado en un objeto **Frame**, que contiene a los dos objetos **Panel** que lo conforman. Uno de los objetos panel servirá como pantalla sobre la que presentar las diversas fichas (*cards*), utilizando el **CardLayout**. El otro **Panel** contiene varios objetos **Button** que se pueden utilizar para cambiar las fichas que se presentan en el **Panel** contrario.

El programa es un poco más complejo que en los ejemplos anteriores de los controladores de posición, debido a que el **CardLayout** también es más complicado, pero por el contrario, esta complejidad le proporciona un considerable poder y funcionalidad a la hora de su empleo.

Todos los botones de las fichas son pasivos, no tienen registrados controladores de eventos, excepto uno de ellos, que contiene un botón sobre el cual, al pulsarlo, se presenta la fecha y la hora del sistema. Esto se consigue a través de objetos de tipo **ActionListener** que funcionan a dos niveles. En un primer nivel, los objetos **ActionListener** se utilizan para seleccionar la ficha que se visualizará. Y en un segundo nivel, se registra un objeto **ActionListener** para uno de los botones de una de

las fichas, que hará que aparezca en pantalla la fecha y la hora, siempre que la ficha que contiene el botón esté visible y se pulse ese botón. La visualización de esta ficha en la ventana, durante la ejecución del ejemplo, será semejante a la reproducida en la figura 13.25.



Figura 13.25

En el programa, se crea un objeto **Frame** a un nivel superior de la interfaz, que contiene dos objetos **Panel**. Uno de los objetos **Panel** es el panel de visualización que utiliza el programa para presentar cada una de las siguientes fichas, que se añaden al **Panel** utilizando el **CardLayout** como controlador de posicionamiento:

- Una ficha con el botón "*La Primera ficha es un Boton*"
- Una ficha con la etiqueta "*La Segunda ficha es un Label*"
- Una ficha con la etiqueta "*Tercera ficha, tambien un Label*"
- Una ficha con la etiqueta "*Cuarta ficha, de nuevo un Label*"
- Una ficha, que es el "panel fecha", con el botón "*Fecha y Hora*" y la etiqueta donde presenta la fecha
- Una ficha con un campo de texto, inicializado con la cadena "*La Ultima ficha es un campo de texto*"

El otro **Panel**, que contiene el objeto **Frame** principal, es el panel de control, que contiene a su vez cinco botones, cuatro de desplazamiento y otro que va a permitir la visualización directa de la ficha en donde se presenta la fecha y la hora. Todos los botones son activos, ya que disponen de controladores de eventos registrados sobre ellos, y la acción que realizan es la que indica su etiqueta; por ejemplo, si se pulsa el botón "*siguiente*", aparecerá en la ventana la siguiente ficha a la que se esté visualizando en ese momento.

El listado completo del ejemplo lo puede encontrar el lector en el código que acompaña al libro, a continuación se discuten las partes más interesantes de ese código, que al ser más complejo que los vistos en secciones anteriores, también tiene más trozos de código que merecen un comentario.

El primer trozo de código interesante que se encuentra en la visión del programa es el usado para crear una cualquiera de las fichas, que contiene un objeto **Button** y un objeto **Label**, y que va a componer posteriormente el **Panel** que permitirá visualizar la fecha y la hora. Sobre el objeto **Button** se instancia y registra un objeto receptor de eventos de tipo **ActionListener**. El método sobrescrito *actionPerformed()* en el objeto

ActionListener hace que la fecha y la hora aparezcan en el objeto **Label** que está sobre la ficha cuando se pulsa sobre el botón.

```
Label labelFecha =
  new Label("                                                    ");
Button botonFecha = new Button( "Fecha y Hora" );
Panel panelFecha = new Panel();
panelFecha.add( botonFecha );
panelFecha.add( labelFecha );
botonFecha.addActionListener(
new ActionListernerFecha( labelFecha ) );
```

Las siguientes sentencias son las que permiten la creación de las fichas del panel de visualización. Primero se instancia un objeto de tipo **CardLayout** y se asigna a una variable de referencia, ya que no puede ser un objeto anónimo, porque se va a necesitar a la hora de procesar los eventos.

```
CardLayout miCardLayout = new CardLayout();
```

A continuación, se instancia el objeto **Panel** correspondiente al panel de visualización de las fichas, se especifica el controlador de posicionamiento de los componentes y se modifica su fondo para que sea de color amarillo y se pueda distinguir fácilmente del otro objeto **Panel** que se usa en la interfaz.

```
Panel panelPresentacion = new Panel();
panelPresentacion.setLayout( miCardLayout );
panelPresentacion.setBackground( Color.yellow );
```

Una vez que ya existe el objeto **Panel** para la visualización, el siguiente paso es añadir fichas al **Panel** utilizando el **CardLayout**. En las llamadas al método *add()* de la clase **Container** de las sentencias que se reproducen, el primer parámetro es el objeto que se añade, y el segundo parámetro es el nombre del objeto. Todos los objetos que se añaden son anónimos porque son pasivos, excepto el objeto `panelFecha`, que se ha construido anteriormente y que no puede ser anónimo al estar compuesto por un objeto **Panel**, un objeto **Button** y un objeto **Label**, y se necesita una variable de referencia del **Panel** para poder instanciarlo.

El nombre del objeto especificado como cadena en el segundo parámetro del método *add()*, se utilizará como parámetro del método *show()*, para indicar cuál de las fichas será la que se visualice.

```
panelPresentacion.add(
  new Button( "La Primera ficha es un Boton" ),"primero" );
panelPresentacion.add(
  new Label( "La Segunda ficha es un Label" ),"segundo" );
panelPresentacion.add(
  new Label( "Tercera ficha, tambien un Label" ),"tercero" );
panelPresentacion.add(
  new Label( "Cuarta ficha, de nuevo un label" ),"cuarto" );
panelPresentacion.add( panelFecha,"panel fecha" );
panelPresentacion.add(
  new TextField( "La Ultima ficha es un campo de texto" ),"sexto" );
```

La siguiente tarea es ya la construcción del panel de control. Las dos sentencias que siguen son conocidas, y sirven para instanciar los objetos **Button** del panel de control y registrar objetos **ActionListener** para que capturen los eventos que se produzcan sobre ellos.

```
Button botonSiguiente = new Button( "Siguiente" );
. . .
botonPrimero.addActionListener(
  new ListenerPrimero( miCardLayout,panelPresentacion ) );
```

Las dos líneas de código mostradas a continuación también resultarán conocidas, y aquí se utilizan para instanciar el objeto del panel de control y colocar los cinco botones sobre él.

```
Panel panelControl = new Panel();
panelControl.add( botonPrimero );
```

Ahora se colocan los dos paneles sobre el **Frame** que constituye la parte principal de la interfaz de usuario utilizando un **BorderLayout** y colocándolos en posiciones superior e inferior.

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.add( panelPresentacion,"North" );
miFrame.add( panelControl,"South" );
```

En este momento ya está creada la clase correspondiente a la interfaz de usuario y lo que resta es definir las clases que van a recibir los eventos, para lo cual se instancian y registran objetos receptores que contienen el método *actionPerformed()* sobrescrito. La primera clase **ActionListener** es la que se usa para dar servicio a los eventos provocados por la pulsación sobre el objeto **Button** colocado sobre la ficha que presenta la fecha y la hora. Esta clase contiene una variable de instancia, un constructor y el método *actionPerformed()* sobrescrito. La parte más interesante es el código del método *actionPerformed()*, tal como muestra el siguiente trozo de código:

```
public void actionPerformed( ActionEvent evt ) {
  miObjLabel.setText( new Date().toString() );
  }
```

Como se puede observar, este método instancia un nuevo objeto de la clase **Date**, que cada vez que es instanciado contiene información que puede utilizarse para obtener la fecha y hora en que fue instanciado. Esta información se extrae utilizando el método *toString()* y luego, se emplea el método *setText()* de la clase **Label** para depositar la fecha y hora en el objeto **Label**.

Esta clase es seguida de cinco clases **ActionListener** muy semejantes entre ellas, que se utilizan para cambiar la ficha que se presenta en pantalla cuando cualquiera de los botones del panel de control es pulsado. Las definiciones de las clases difieren únicamente en el método que se llama dentro del método *actionPerformed()*.

El resto del código no es de especial interés, y ya es suficiente (esperemos), con los comentarios situados en el código fuente del ejemplo para que el lector tome posición correcta ante el entendimiento de ese código.

Posicionamiento absoluto

Los componentes se pueden colocar en contenedores utilizando cualquiera de los controladores de posicionamiento, o bien utilizando posicionamiento absoluto para realizar esta función. La primera forma de colocar los componentes se considera más segura porque automáticamente serán compensadas las diferencias que se puedan encontrar entre resoluciones de pantalla de plataformas distintas.

La clase **Component** proporciona métodos para especificar la posición y tamaño de un componente en coordenadas absolutas indicadas en píxeles:

```
setBounds( int,int,int,int );
setBounds( Rectangle );
```

La posición y tamaño si se especifican en coordenadas absolutas puede hacer más difícil el conseguir que la apariencia de la interfaz de usuario sea uniforme en cualquier plataforma, según algunos autores; pero, a pesar de ello, es interesante saber cómo se hace.

La aplicación `Java1329.java` coloca un objeto **Button** y un objeto **Label** sobre un objeto **Frame** utilizando coordenadas absolutas en píxeles. El programa está diseñado para ser lo más simple posible y solamente contiene un controlador de eventos, para el botón de cerrar la ventana colocado sobre el **Frame**.

El objeto **Button** y el objeto **Label**, de color amarillo, se colocan sobre un objeto **Frame**, de tal forma que la posición y tamaño indicados para los componentes hacen que los dos se superpongan. Éste es uno de los potenciales problemas que se producen con el uso de coordenadas absolutas. Si se cambia el tamaño del **Frame**, el botón y la etiqueta permanecerán en el mismo tamaño y posición. El **Frame** puede ser cambiado de tamaño hasta el punto de no poder ver los dos componentes, y esto no es agradable para ningún usuario.

Ahora se entra un poco más a fondo en el programa, para aclarar las partes más interesantes. Por ejemplo, si el lector consulta el código completo del ejemplo, las siguientes dos sentencias

```
Button miBoton = new Button( "Boton" );
// Al rectángulo se le pasan los parámetros: x,y,ancho,alto
miBoton.setBounds( new Rectangle( 25,35,100,75 ) );
```

lo que hacen en realidad es crear un objeto **Button** con el rótulo "Boton", indicar que su posición en el eje X es 25 y en el eje Y es 35, medidas en píxeles desde la esquina superior izquierda del Contenedor del objeto, teniendo en cuenta que el eje Y crece

hacia abajo. El punto de referencia del botón es su esquina superior-izquierda; finalmente, especifica una anchura de 100 píxeles y una anchura de 75 píxeles para el botón.

En este punto del programa, una vez ejecutadas estas dos sentencias, todavía no se ha identificado el objeto que va a contener el Botón, por lo tanto, *setBounds()* proporciona información de las coordenadas absolutas donde se colocará el botón en cualquiera que sea el Contenedor que lo contenga. El método *setBounds()* es un método de la clase **Component** y por lo tanto, heredado en todas las subclases de **Component**, incluida la clase **Button**. El método indica que el componente tendrá un tamaño y estará posicionado de acuerdo a una caja o rectángulo, cuyos lados son paralelos a los ejes X e Y.

Hay dos versiones sobrecargadas de *setBounds()*. Una de ellas permite que el tamaño y posición se indiquen a través de cuatro enteros: ordenadas, abscisas, anchura y altura. La otra versión requiere que se utilice un objeto **Rectangle**, que se pasa como parámetro al método. Esta última versión es más flexible, porque la clase **Rectangle** tiene siete constructores diferentes que pueden ser utilizados para crear la caja que indicará la posición y tamaño del componente.

Las siguientes líneas de código crean un objeto **Label** con fondo amarillo y, de nuevo, se utiliza el método *setBounds()* para posicionarlo:

```
Label miEtiqueta = new Label( "Tutorial de Java" );
miEtiqueta.setBounds( new Rectangle( 100,80,100,75 ) );
miEtiqueta.setBackground( Color.yellow );
```

El método *setBackground()* también es un método de la clase **Component** heredado por todas sus subclases. Este método requiere un parámetro de tipo **Color**, que se puede crear de varias formas, pero la más simple es referirse a una de las constantes que están definidas en la clase **Color**, utilizando sintaxis del tipo:

```
public final static Color yellow;
```

Hay que recordar que a las variables declaradas como estáticas en una clase se puede acceder utilizando el nombre de la clase y el nombre de la variable. Haciendo las variables de tipo `final`, lo que se consigue es en realidad, *constantes*.

Hay unos treinta colores diferentes definidos de este modo, si se necesita un color distinto, no incluido en esta lista, se podrá utilizar uno de los constructores sobrecargados que permiten especificar un color en función de la cantidad de rojo, verde y azul que intervienen en su composición.

Las siguientes líneas de código

```
Frame miFrame = new Frame( "Tutorial de Java, AWT" );
miFrame.setLayout( null );
```

crean un objeto **Frame** con un título. Este objeto **Frame**, o marco, es el tipo de objeto que los usuarios consideran como una típica Ventana. Se puede cambiar de tamaño y tiene cuatro botones: un botón de control o de menú de ventana en la parte superior izquierda y, en la zona superior-derecha, dispone de tres botones para minimizar, maximizar y cerrar la ventana.

En este ejemplo no se acepta el *layout* de tipo **BorderLayout**, que por defecto proporciona el sistema, sino que se especifican posición y tamaño de los componentes en coordenadas absolutas. Por ello, será necesario sobrescribir la especificación del layout con `null`.

Este parámetro se le pasa al método *setLayout()* de la clase **Container** de la que **Frame** es una subclase, y que espera como parámetro un objeto de la clase **LayoutManager**, o `null`, para indicar que no se va a utilizar ningún manejador de composición, como en este caso.

```
miFrame.add( miBoton );
miFrame.add( miEtiqueta );
```

Las dos líneas anteriores hacen que los dos componentes que previamente se habían definido entren a formar parte de la composición visual utilizando para ello el método *add()* de la clase **Container**. Este método tiene varias versiones sobrecargadas, y aquí se utiliza la que requiere un objeto de tipo **Component** como parámetro; y como **Button** y **Label** son subclases de la clase **Component**, están perfectamente cualificados para poder ser parámetros del método *add()*.

Una vez que se ha llegado a este punto, ya solamente falta fijar el tamaño que va a tener el marco en la pantalla y hacerlo visible, cosa que hacen las dos líneas de código que se reproducen a continuación:

```
miFrame.setSize( 250,175 );
miFrame.setVisible( true );
```

BoxLayout

El controlador de posicionamiento **BoxLayout** es uno de los dos que incorpora Java a través de Swing. Permite colocar los componentes a lo largo del eje **X** o del eje **Y**, y también posibilita que los componentes ocupen diferente espacio a lo largo del eje principal.

En un controlador **BoxLayout** sobre el eje Y, los componentes se posicionan de arriba hacia abajo en el orden en que se han añadido. Al contrario en el caso del **GridLayout**, aquí se permite que los componentes sean de diferente tamaño a lo largo del eje **Y**, que es el eje principal del controlador de posicionamiento.

En el eje que no es principal, **BoxLayout** intenta que todos los componentes sean tan anchos como el más ancho, o tan altos como el más alto, dependiendo de cuál sea

el eje principal. Si un componente no puede incrementar su tamaño, el **BoxLayout** mirará las propiedades de alineamiento en **X** e **Y** para determinar dónde colocarlo.

OverlayLayout

El controlador de posicionamiento **OverlayLayout**, también se incorpora a Java con Swing, y es un poco diferente a todos los demás. Se dimensiona de forma que pueda contener el más grande de los componentes y superpone cada componente sobre los otros.

La clase **OverlayLayout** no tiene un constructor por defecto, así que hay que crearlo dinámicamente en tiempo de ejecución.

GroupLayout

El último controlador de posicionamiento incorporado a Java es el **GroupLayout** que, aunque ha sido desarrollado fundamentalmente para su utilización en herramientas destinadas al diseño interactivo de interfaces de usuario, también está accesible al programador, proporcionando la característica de independencia de la dimensión, que permite utilizar los posicionamientos horizontal y vertical de forma independiente.

Este comportamiento no es infrecuente, al contrario que en otros controladores, aunque en el **GroupLayout** no se utiliza un único objeto de tipo **Constraints** para especificar el comportamiento, sino que se cuando se define el posicionamiento horizontal, no es necesario saber nada del posicionamiento vertical, y viceversa. La contrapartida es que el posicionamiento necesita ser definido dos veces y, en caso de no hacerlo, se generará una excepción.

Profundidad, *Z-Order*

Una de las características nuevas que se integraron al AWT en el J2SE 5 es la definición de la profundidad de los componentes en la pantalla, o lo que es lo mismo, la posibilidad de determinar el *Z-Order* de cada uno de los componentes que se colocan en la interfaz de usuario.

La profundidad o *Z-Order*, controla la posición en que se colocarán los componentes uno encima de otro, considerando que cada uno se coloca en una capa, de modo que es posible determinar el solapamiento entre unos componentes con respecto a los otros.

El componente básico **Container** proporciona dos nuevos métodos: *setComponentZOrder()* y *getComponentZOrder()*, que son los que controlan la profundidad de los componentes a la hora del solapamiento. Es decir, estos métodos solamente tienen efecto cuando hay solapamiento de componentes.

Para mostrar esta característica, la aplicación `Java1330.java` crea cuatro botones y los presenta de arriba hacia abajo, respecto a la profundidad, solapándose unos a otros. La figura 13.26 muestra gráficamente lo que se acaba de explicar.



Figura 13.26

Como la propiedad *Z-Order* se asigna a la clase **Container**, funcionará tanto con los componentes AWT, como se acaba de ver en el ejemplo, pero también con los componentes ligeros de Swing, aunque es necesario tener en cuenta que no todas las plataformas soportan el cambio de profundidad de unos componentes con respecto a otros sin hacer una llamada a *removeNotify()*. No hay forma de saber si la plataforma soporta o no esta característica, así que no es posible hacer asunciones de ningún tipo al respecto.

ESCRITORIO

La aportación más importante realizada en el J2SE 6 a la plataforma Java, ha sido la integración con los escritorios gráficos de los distintos sistemas. Las características introducidas van desde el soporte para pantallas de presentación (*splash screen*), que permiten a las aplicaciones visualizar ventanas de bienvenida durante su inicialización, pasando por el soporte para la barra de tareas, que permite añadir iconos y menús emergentes en la barra de tareas, hasta una completa API de escritorio.

Este API de escritorio utiliza las asociaciones de ficheros del sistema operativo sobre el que se ejecuta Java, para lanzar aplicaciones asociadas a los distintos tipos de ficheros. Por ejemplo, la extensión `.doc` de un fichero, normalmente está asociada a Microsoft Word o a *OpenOffice Word*; a través del API de escritorio, una aplicación Java puede lanzar *Word* para escribir o imprimir ficheros con esta extensión, asumiendo que el sistema operativo sobre el que se está ejecutando la aplicación soporta la asociación de ficheros.

Este API también permite que las aplicaciones Java puedan lanzar el navegador por defecto establecido en el sistema para abrir una dirección URL específica, o lanzar el cliente de correo electrónico establecido por defecto o cualquier otra aplicación.

Antes de realizar cualquiera de las acciones anteriores, la aplicación Java debe determinar si el sistema operativo sobre el que se está ejecutando soporta el API de

escritorio. Esto se hace invocando al método *isDesktopSupported()* de la clase **Desktop**, que devuelve `true` si el API está soportado por el sistema operativo y `false` en caso contrario.

Tras invocar al método anterior, la aplicación debe recuperar un objeto de tipo **Desktop**, invocando al método *getDesktop()*, que lanzará una excepción de tipo **Headless** si el sistema operativo no soporta un teclado, una pantalla o el ratón, o una excepción de tipo **UnsupportedException** si el API de escritorio no está soportado.

Una vez que la aplicación dispone de una instancia del objeto **Desktop**, ya puede invocar a diferentes métodos, en función de las acciones a realizar, aunque en este caso también es conveniente invocar antes que a las propias acciones, al método *isSupported[acción]()*, donde *acción* es la operación que se quiere realizar. Devuelve `true` si la acción está soportada. Las acciones disponibles están determinadas por el enumerado **Desktop.Action**, que ofrece las siguientes posibilidades:

`BROWSE`, permite lanzar el navegador por defecto.

`MAIL`, permite lanzar el cliente de correo electrónico por defecto.

`OPEN`, permite abrir un tipo de fichero específico con la aplicación asociada a ese tipo de ficheros.

`EDIT`, igual que el anterior para editar un tipo de ficheros específico.

`PRINT`, igual que el anterior para imprimir ficheros de un determinado tipo.

En el caso de que se recuperen excepciones, no es necesario invocar al método *isSupported[acción]()* para determinar si una acción está soportada, porque la invocación directa de cualquiera de los métodos que realizan las acciones anteriores, lanzará una excepción de tipo **UnsupportedOperationException** en el caso de que la acción no esté soportada.

La aplicación `Java1335.java` muestra alguna de las características del API de escritorio. Consiste en una interfaz que presenta distintos botones para cada una de las acciones a realizar. Para ejecutar cada una de ellas, la ventana dispone de un campo de texto en el que se puede indicar un nombre de fichero o una dirección URL, según el tipo de acción seleccionada. La figura 13.27 reproduce la ventana de la aplicación una vez lanzada.



Figura 13.27

En la figura se observa que no hay editor definido por defecto y se han deshabilitado las impresoras del sistema, por lo que los botones correspondientes a dichas acciones se encuentran deshabilitados.

Pantalla de presentación

Las pantallas, o ventanas, de presentación durante el proceso de inicialización de las aplicaciones, son ya una parte inseparable de cualquier aplicación moderna que se ejecute en un entorno de ventanas. El principal propósito de este tipo de ventanas es proporcionar al usuario información de las acciones que está realizando la aplicación para arrancar, captando la atención y ganando la confianza de ese usuario. Además, permiten presentar información comercial, legal, información acerca de derechos, logotipos, etc.

Estas ventanas de inicialización pueden generarse a partir de las clases AWT o Swing, desde versiones anteriores del JDK a la actual. Sin embargo, la principal característica de este tipo de ventanas es proporcionar al usuario información inmediata del proceso de inicialización, por lo que el tiempo que transcurre entre el lanzamiento de la aplicación y el momento en que aparece la ventana debe ser mínimo. Antes de que aparezca cualquier ventana de la aplicación, se ha de cargar e inicializar la Máquina Virtual Java, AWT, quizás Swing y cualquier otra librería que utilice la aplicación. El resultado de todo ello conduce a un retraso de varios segundos en la aparición de ventanas, con lo que la tecnología Java no era adecuada para la creación de ventanas de presentación, aunque lo permitiese.

La plataforma J2SE 6, proporciona una solución al problema anterior, haciendo que las ventanas de inicialización se puedan visualizar mucho antes, incluso antes de que la Máquina Virtual Java se haya inicializado.

Para conseguirlo, Java recurre a una opción en el fichero de configuración *manifest*, que permite a cualquier aplicación empaquetada en un fichero JAR, presentar una ventana de inicialización. Otras aplicaciones pueden utilizar la línea de comandos para lanzarlas; en cuyo caso se puede utilizar un acceso directo en el

escritorio a un *script* que permita lanzar la aplicación con el parámetro correspondiente.

Por tanto, hay dos formas de mostrar una ventana de visualización. Una de ellas es a través de la línea de comandos o del acceso directo a un *script* que incluya la línea de comando, utilizando la opción `splash` para lanzar la aplicación:

```
java -splash:ventana.gif aplicacion
```

La otra forma es para la ocasión en que la aplicación se empaquete en un fichero JAR, en cuyo caso se puede utilizar la opción `SplashScreen-Image` en el fichero *manifest* para mostrar la ventana. La imagen se debe incluir en el fichero JAR y en la opción se ha de indicar el *path* en que se encuentra. Por ejemplo,

```
Manifest-Version: 1.0
Main-Class: aplicacion
SplashScreen-Image: ventana.gif
```

Las ventanas de inicialización pueden presentar cualquier tipo de imagen *GIF*, *PNG* y *JPEG*, con transparencia, traslucidez y animación.

Esta característica de Java es muy fácil de utilizar porque, en la mayoría de los casos, basta con crear una imagen y utilizarla mediante cualquiera de los dos métodos anteriores. La ventana desaparecerá por sí misma cuando se presente la primera ventana de AWT o Swing de la aplicación en pantalla.

Sin embargo, hay ocasiones en que es necesario proporcionar algún tipo de información dinámica en la ventana de inicialización. En estos casos, se utiliza la clase **SplashScreen** para manipular la imagen. Esta clase no se puede utilizar para crear la ventana, que solamente se puede hacer mediante cualquiera de los dos métodos que ya se han explicado.

Además, la clase **SplashScreen** no puede ser instanciada, solamente puede existir una única instancia, de la cual se puede obtener una referencia a través del método *getSplashScreen()*. Si la aplicación no ha creado la ventana de inicialización, la invocación a este método devolverá `null`.

Normalmente, el uso de esta clase es para mantener la imagen en pantalla y presentar algún tipo de información, como una barra de progreso por ejemplo, sobre esa imagen. La ventana de inicialización dispone de una zona superpuesta con un canal *alfa*, accesible a través de las interfaces **Graphics** y **Graphics2D**.

En el código siguiente se muestra el uso de esta clase. En primer lugar se obtiene la instancia de **SplashScreen** y el acceso al contexto gráfico invocando al método *getGraphics()*. Se obtiene también el tamaño de la ventana de inicialización y se elimina cualquier imagen que se hubiese colocado. Se fija el modo

AlphaComposite.Clear, se pinta un rectángulo que ocupa toda la ventana y se restaura el modo de dibujo para pintar ya cualquier otra cosa.

```
SplashScreen ss = SplashScreen.getSplashScreen();
Graphics2D g = ss.createGraphics();
Dimension size = ss.getDimension();
g.setComposite( AlphaComposite.Clear );
g.fillRect( 0,0,size.width,size.height );
g.setPaintMode();
```

Si se quiere cerrar la ventana de inicialización antes de que aparezca la primera ventana de AWT o Swing, o en el caso de que la aplicación no tenga presentación gráfica, se puede invocar al método *SplashScreen.close()*.

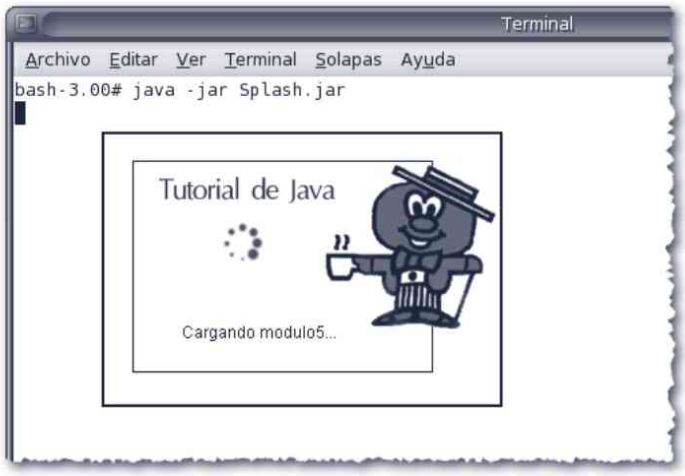


Figura 13.28

El ejemplo `Java1336.java` muestra cómo funciona esta característica de Java. La figura 13.28 corresponde a la ventana de inicialización en el arranque de la aplicación.

Si el lector consulta el código de la aplicación, observará que se utiliza el método *createtGraphics()* para crear el contexto gráfico que corresponde a la imagen superpuesta a la que se carga inicialmente. En lugar de pintar sobre la imagen inicial, se hace sobre esta imagen superpuesta utilizando el canal *alfa* que permite sobreimpresionar sobre la imagen original. También debe observar el lector el uso del método *update()* para actualizar el contenido de la ventana, porque el hecho de pintar sobre el contexto gráfico no realiza la actualización por sí solo.

Barra de tareas

El acceso a la barra de tareas es otra de las características aportadas por J2SE 6. Se realiza a través de dos clases: **SystemTray** y **TrayIcon**; que permiten añadir imágenes, menús y ventanas flotantes (*tooltips*) en la barra de tareas.

La barra de tareas es una zona especializada que, normalmente, se encuentra en la parte derecha de la barra inferior del escritorio, donde habitualmente se muestra la

hora y los usuarios pueden ver y acceder a las aplicaciones que se ejecutan de forma ininterrumpida. La imagen 13.29 muestra la barra de tareas en Fedora (Linux), en Windows Vista y en Sun Java Desktop 3, encontrándose en los tres casos en la parte inferior derecha de la pantalla.



Figura 13.29

Para acceder a la barra de tareas, hay que invocar al método *getSystemTray()* de la clase **SystemTray**, que es la que representa a la barra de tareas del escritorio. La invocación a este método ha de ir precedida de la llamada al método *isSupported()* o tratar la excepción **UnsupportedOperationException**, para asegurarse de que la plataforma soporta esta característica.

Cada aplicación solamente puede tener una única instancia de **SystemTray**, que no puede crearse, sino que debe obtenerse a través de la llamada al método *getSystemTray()*.

La instancia de **SystemTray** puede contener uno o varios objetos de tipo **TrayIcon**, que pueden ser añadidos o eliminados invocando a los métodos *add()* y *remove()*, respectivamente. El trozo de código que se reproduce a continuación, muestra la forma de acceder y personalizar la barra de tareas.

```
final TrayIcon tIcono;
if( SystemTray.isSupported() ) {
  SystemTray st = SystemTray.getSystemTray();
  Image img = Toolkit.getDefaultToolkit().getImage( "icono.gif" );
  MouseListener ml = new MouseListener() {
    public void mouseClicked( MouseEvent e ) {
      System.out.println( "Barra Tareas - Clic de ratón!");
      }
    };
  tIcono = new TrayIcon( img,"Demo",popup );
  tIcono.setImageAutoSize( true );
  tIcono.addMouseListener( ml );
  try {
    tray.add( tIcono );
  } catch( AWTException e ) {
    System.err.println( "TrayIcon no puede añadirse..");
  }
} else {
    //  System Tray no está soportado
}
```

Como el lector puede observar, la funcionalidad de **TrayIcon** va más allá de la simple presentación de un icono en la barra de tareas, porque además puede presentar

un *tooltip*, abrir un menú o permitir que le sean asociados receptores de eventos, aunque en el código anterior solamente se captura el evento de clic del ratón.

La imagen que presenta **TrayIcon** se puede actualizar en tiempo de ejecución utilizando el método *setImage()*. Y del mismo modo, invocando al método *setToolTip()* se puede actualizar el texto que aparece cuando se pasa el cursor sobre la imagen del objeto **TrayIcon**.

Una de las propiedades más importantes de **TrayIcon** es la de *autoajuste*. Esta propiedad determina el espacio disponible en la barra de tareas y redimensiona la imagen para que se adapte a dicho espacio. Por defecto, esta característica está deshabilitada, por lo que la imagen se presenta a su tamaño real, es decir, si la imagen es más grande que el espacio disponible, será recortada.

Otra de las propiedades que proporciona **TayIcon** es la posibilidad de presentar un mensaje utilizando el tooltip, mediante la invocación del método *displayMessage()*. Este método hará aparecer el mensaje cerca de la imagen de **TrayIcon** y desaparecerá después de un tiempo o cuando se haga clic sobre el mensaje. Esos mensajes se utilizan para indicar, por ejemplo, cambios en el estado de la aplicación, la recepción de mensajes, etc.

La aplicación `Java1337.java` muestra el uso de las clases **SystemTray** y **TrayIcon** para presentar una imagen en la barra de tareas con la que se puede interaccionar. La figura 13.30 muestra los estados de la aplicación. En la imagen izquierda aparece el *tooltip* cuando el cursor se pasa por encima de la imagen correspondiente al **TrayIcon** y en la imagen derecha aparece el menú *popup* que se abre al hacer clic sobre la imagen del **TrayIcon**.



Figura 13.30

IMPRIMIR CON EL AWT

La clase **Toolkit** es una clase que proporciona una interfaz independiente de plataforma para servicios específicos de las plataformas, como pueden ser: fuentes de caracteres, imágenes, impresión y parámetros de pantalla. El constructor de la clase es abstracto y, por lo tanto, no se puede instanciar ningún objeto de la clase **Toolkit**; sin embargo, sí que se puede obtener un objeto **Toolkit** mediante la invocación del método *getDefaultToolkit()*, que devolverá un objeto de este tipo, adecuado a la plataforma en que se esté ejecutando.

De entre los muchos métodos de la clase **Toolkit**, el que representa el máximo interés en este momento es el método *getPrintJob()*, que devuelve un objeto de tipo **PrintJob** para usarlo en la impresión desde Java.

En Java hay, al menos, dos formas de poder imprimir. Una es disponer de un objeto de tipo **Graphics**, que haga las veces del papel en la impresora y dibujar, o pintar, sobre ese objeto. La otra, consiste en preguntar a un componente, o a todos, si tienen algo que imprimir, y hacerlo a través del método *printAll()*.

Como el camino siempre se muestra andando, como decía el poeta, a continuación se emplea una de estas formas de imprimir. El propósito del ejemplo `Java1331.java` es comprobar la capacidad para imprimir de los componentes del AWT que se encuentran en un Contenedor, bien sea éste el Contenedor raíz o bien se encuentre incluido en otro Contenedor. La figura 13.31 reproduce la ventana que se presenta en pantalla al ejecutar el programa y pulsar el botón de selección del panel 0.



Figura 13.31

El programa coloca uno de dos objetos **Panel** seleccionables y cuatro objetos **Button**, sobre un objeto **Frame**. Uno de los botones tiene un receptor que hace que el **Panel** que se encuentre seleccionado y todos los componentes que se encuentren en ese **Panel** sean impresos. Otro de los botones tiene un receptor que hace que el **Frame** raíz y todos los componentes sean enviados a la impresora. En realidad, el **Frame** no se puede imprimir a sí mismo, sino que hace que todos sus componentes sean impresos, lo cual contradice un poco lo que aparece en la documentación de *Sun*, que dice, literalmente: "*Imprime este componente y todos sus subcomponentes*". Lo mismo, probablemente, le pase al **Panel**, pero este objeto **Panel** no tiene ninguna característica que permita saber si está siendo o no impreso.

Los dos botones anteriores comparten el mismo objeto receptor, pero la acción que realizan es la descrita. Los otros dos botones se utilizan para seleccionar los dos paneles. Es decir, el usuario puede seleccionar entre los dos paneles diferentes y hace que el que se esté visualizando en el **Frame** se envíe a la impresora.

Cuando el **Panel** seleccionado se está imprimiendo, los otros componentes del **Frame** son ignorados. Sin embargo, cuando el **Frame** se está imprimiendo, todos los componentes del **Frame**, incluido el **Panel** seleccionado, se enviarán a la impresora.

El contenido de los paneles es solamente como muestra, por ello uno contiene una etiqueta, un campo de texto y un botón no activo, y el otro contiene una etiqueta, un campo de texto y dos botones inactivos.

A continuación, se repasan los fragmentos de código que pueden resultar interesantes de todo el listado del ejemplo. El primer fragmento de código interesante es el que muestra la clase de control, con el método *main()* que instancia un objeto de la clase **IHM**, que es el que en realidad aparece en la pantalla.

```
public class Java1331 {
  public static void main( String args[] ) {
    // Se instancia un objeto de la clase Interfaz Gráfica
    IHM ihm = new IHM();
    }
  }
```

En el código siguiente se muestra el comienzo de la clase **IHM**, incluyendo la declaración de diferentes variables de referencia.

```
class IHM {
  // El contenedor miFrame y todos los componentes que contiene, serán
  // impresos o enviados a un fichero de impresora cuando se pulse el
  // botón con el rótulo "Imprimir Frame"
  Frame miFrame = new Frame( "Tutorial de Java, AWT" );
  // El contenedor panelAImprimir y todos los componentes que
  // contiene, serán impresos o enviados a un fichero de impresora
  // cuando se pulse el botón con el rótulo "Imprimir Panel"
  Panel panelAImprimir = null;
  // Referencias a los dos paneles seleccionables
  Panel panel0;
  Panel panel1;
```

El contenedor **miFrame** y todos los componentes que contiene serán enviados a la impresora, o a un fichero de impresión, cuando se pulse el botón rotulado "Imprimir Frame". El contenedor **panelAImprimir** y todos sus componentes serán impresos o enviados a un fichero de impresión al pulsar el botón "Imprimir Panel". Las variables de referencia de los objetos **Panel** son referencias a los dos paneles seleccionables.

Tanto el código correspondiente al constructor de la clase **IHM**, como el utilizado para construir los paneles, es muy semejante al que se ha visto en secciones anteriores, por lo que no merece la pena revisarlo, aunque no deja de tener un cierto interés.

Las líneas de código que aparecen a continuación sí que ya resultan interesantes para el objetivo de esta sección. Muestran la definición de una clase anidada de la clase **IHM**, que es utilizada por el objeto **Panel**, referenciado por **panelAImprimir**, o el objeto **Frame**, referenciado por **miFrame**, que van a ser los objetos que sean impresos.

```
class PrintActionListener implements ActionListener {
  public void actionPerformed( ActionEvent evt ) {
    PrintJob miPrintJob = miFrame.getToolkit().
```

```
      getPrintJob( miFrame,"Tutorial de Java, AWT",null );
```

Esta clase es un receptor de eventos de tipo **Action**, y el código del método *actionPerformed()* es el que hace que tenga lugar la impresión. Las líneas de código anteriores, correspondientes al comienzo de la clase anidada, son utilizadas para conseguir un objeto de tipo **PrintJob**. Esto hace que el diálogo estándar de selección de impresora aparezca en la pantalla. Si el usuario cierra este diálogo mediante el botón "Cancelar", es decir, sin activar la impresión, el método *getPrintJob()* devolverá `null`.

El siguiente código, de la misma clase anidada, comprueba que efectivamente, el usuario quiere imprimir para proceder con esa impresión.

```
if( miPrintJob != null ) {
  Graphics graficoImpresion = miPrintJob.getGraphics();
  if( graficoImpresion != null ) {
    if( evt.getActionCommand().equals( "Imprimir Panel" ) )
      panelAImprimir.printAll( graficoImpresion );
    else
      miFrame.printAll( graficoImpresion );
```

El primer paso, en caso afirmativo, es utilizar el método *getGraphics()* para obtener un objeto de tipo **Graphics**, que representará el papel de la impresora. Este objeto será el que sea requerido por el método *printAll()*, que se invocará seguidamente.

Una vez comprobada la validez del objeto **Graphics**, hay que determinar los objetos que se van a imprimir. El contenedor externo es un objeto **Frame**, y el contenedor interno es un objeto **Panel**. Esto se hace invocando el método *getActionCommand()* sobre el objeto que ha sido origen del evento **ActionEvent**. Luego, depende del origen del evento que el método *printAll()* sea invocado sobre el **Panel** o el **Frame**, pasando el objeto **Graphics** como parámetro.

Una vez concluida la impresión, es necesario hacer que el papel salga y, también, liberar todos los recursos que hayan sido utilizados por el objeto **Graphics**. Esto se consigue invocando al método *dispose()* sobre el objeto **Graphics**, tal como se muestra en el código que se reproduce a continuación.

```
    graficoImpresion.dispose();
    }
  else
    System.out.println( "No se puede imprimir el objeto" );
  miPrintJob.end();
  }
else
  System.out.println( "Impresion cancelada" );
```

En el código anterior también se llama al método *end()* sobre el objeto **PrintJob**, para hacer las cosas tal como indica *Sun Microsystems* a la hora de "*finalizar la impresión y realizar cualquier limpieza necesaria*" (sic).

Las líneas de código anteriores también contienen la parte del `else` correspondiente al `if` que comprueba la validez del objeto **Graphics**. Si no es válido, el proceso de impresión se cortará y aparecerán algunos mensajes en pantalla, aunque probablemente estaría más elegante el lanzar una excepción, pero como ejemplo es suficiente.

Las dos definiciones de las clases **ActionListener** que siguen en el código del ejemplo crean los objetos los utilizados para el control del proceso de selección del panel que se va a visualizar. El lector debería echarles un vistazo, aunque aquí no se reproduzcan por no ser el tema concreto de la sección.

El siguiente ejemplo, `Java1432.java`, muestra la capacidad para imprimir selectivamente componentes del AWT correspondientes a un Contenedor embebido en otro Contenedor.

La palabra *selectivamente* se usa para diferenciar el método de impresión del visto en el ejemplo anterior, porque en el programa se utilizaba el método *printAll()* para imprimir todos los componentes del Contenedor y, en este nuevo programa, se ha incorporado la capacidad de seleccionar los componentes que van a ser impresos y también, la posibilidad de seleccionar información de estos componentes para enviarla a imprimir.

De forma semejante a como se hacía en la clase **Java1331**, el programa coloca uno de los dos objetos **Panel** seleccionables y tres objetos **Button** sobre un objeto **Frame**. El objeto **Panel** sabe cómo imprimir sus componentes a través del método *paint()* que está sobrescrito.

Uno de los botones tiene un receptor que hace que el **Panel** seleccionado se imprima, lo cual requiere que cada **Panel** tenga un método *paint()* sobrescrito y para ello, cada **Panel** creado debe extender la clase **Panel**. El método *paint()* define la forma en que se va a imprimir el **Panel**.

Los otros dos botones se utilizan para seleccionar entre los dos paneles a la hora de presentarlos en pantalla e imprimirlos. En otras palabras, el usuario puede seleccionar entre los dos paneles y hacer que el que esté presente en el **Frame** se imprima. Cuando este **Panel** seleccionado se está imprimiendo, los otros componentes del **Frame** son ignorados.

En este caso, el formato de impresión definido en el método *paint()* hace que el texto situado en el objeto **Label** y en el objeto **TextField** se impriman y también los títulos de los objetos **Button**. Esto es solamente como ejemplo, porque utilizando la misma técnica, el programador tiene completa libertad a la hora de asociar la información que será impresa con cada uno de los componentes del programa.

Si se vuelven a repasar los fragmentos de código más interesantes del ejemplo, se pueden reproducir en primer lugar las líneas que crean la clase **IHM**, que muestran las referencias a los dos paneles seleccionables que se instalan sobre el objeto **Frame**.

```
class IHM {
  Frame miFrame = new Frame( "Tutorial de Java, AWT" );
  Panel areaAImprimir = null;
  MiPanel0 panel0;
  MiPanel1 panel1;
```

La diferencia con el programa anterior estriba fundamentalmente en el tipo del **Panel**, **MiPanelX**. En este caso no pueden ser de tipo **Panel**, como en el programa anterior, porque va a ser necesario sobrescribir su método *paint()*, con lo cual no queda más remedio que extender la clase **Panel**. Además, han de ser de tipos distintos porque su apariencia y comportamiento a la hora de la impresión van a ser diferentes. Por ello, el constructor difiere del ejemplo anterior en lo indicado, en el resto es muy semejante, tal como se muestra en el código que sigue.

```
public IHM() {
...
  panel0 = new MiPanel0();
  panel1 = new MiPanel1();
...
  }
```

Ahora le toca el turno a la clase anidada de la clase **IHM** que se utiliza para que el objeto **Panel** referenciado por la variable `areaAImprimir` se imprima. Esto se consigue solicitando un contexto para imprimir y pasándoselo al método sobrescrito *paint()* del panel referenciado por la variable anterior. La mayor parte del código es similar al ejemplo anterior, excepto por la línea que se ha dejado huérfana en el siguiente código.

```
public void actionPerformed( ActionEvent evt ) {
...
  areaAImprimir.paint( graficoImpresion );
...
  }
```

El fragmento de código se encuentra dentro del método *actionPerformed()* de la clase **PrintActionListener**, donde se invoca al método *paint()* propio, lo cual hace que el objeto se imprima. Todo esto viene desde la clase en la cual se instanciaron los paneles, ya que los objetos de esta clase saben cómo imprimirse a través del método *paint()* sobrescrito. En este método *paint()* es donde hay que definir el formato de impresión que se desea.

Un hecho a resaltar es que la clase extiende a la clase **Panel**, para poder sobrescribir su método *paint()*. Dentro del método es necesario separar el pintado en pantalla del pintado en impresora, porque de no hacerlo pueden aparecer elementos en pantalla que en realidad están destinados a la impresora.

Se hace una comprobación inicial para ejecutar el código del método *paint()* solamente si el objeto **Graphics** es de tipo **PrintGraphics**; si no, se invoca al método *paint()* de la superclase para seguir preservando la posibilidad de pintar en pantalla. El método *paint()* sobrescrito imprime una línea de cabecera y extrae datos de los componentes del panel, imprimiéndolos en sucesivas líneas. Aquí es donde se puede colocar el código que formatee la salida impresa al gusto.

Hay que tener en cuenta que la impresión directa no tiene una fuente de caracteres por defecto, así que hay que proporcionarle una, porque si no, el sistema puede quejarse. El siguiente código reproduce el método *paint()*.

```
public void paint( Graphics g ) {
  if( g instanceof PrintGraphics ) {
    int margenIzqdo = 10; // Posición X de cada línea
    int margenSup = 20;   // Posición Y de la primera línea
    int pasoLinea = 13;   // Incremento o salto entre líneas

    g.setFont( new Font( "Serif",Font.BOLD,18 ) );
    g.drawString( "Hola desde el Panel 0 del TUTORIAL",
      margenIzqdo,margenSup += pasoLinea );
    g.setFont( new Font( "Serif",Font.PLAIN,10 ) );
    g.drawString( "Texto de la Etiqueta: "+labPanel0.getText(),
      margenIzqdo,margenSup += pasoLinea );
    g.drawString( "Texto del Campo: "+textoPanel0.getText(),
      margenIzqdo,margenSup += pasoLinea );
    g.drawString( "Rotulo del Boton: "+botonPanel0.getLabel(),
      margenIzqdo,margenSup += pasoLinea );
    }
  else
    super.paint( g );
```

Como se puede comprobar, la impresión consiste simplemente en invocar al método *drawString()* sobre el objeto **PrintGraphics** del mismo modo que se hace en cualquier otro programa o applet, incluso en el básico *HolaMundo*. Lo interesante, si el lector observa, es que se está pintando sobre el papel; es decir, que no hay que resignarse a imprimir solamente texto, sino que la imaginación es la que impone el límite.

La parte final del código anterior, correspondiente a la parte del `else`, es una invocación al método *paint()* de la superclase, **Panel** en este caso, cuando el método *paint()* sobrescrito es invocado, pero no se le pasa como parámetro un objeto de tipo **PrintGraphics**. Esto es necesario porque hay que seguir preservando la posibilidad de pintar sobre la pantalla.

La clase anterior está seguida por otra semejante para el otro panel, así que no se va a insistir sobre ello, dejando aquí el tema.

CAPÍTULO 14

SWING

Cuando se empieza a utilizar *Swing*, se observa que *Sun Microsystems* ha dado un gran paso adelante respecto al AWT. Ahora los componentes de la interfaz gráfica son *Beans* y utilizan el nuevo modelo de Delegación de Eventos de Java. *Swing* proporciona un conjunto completo de componentes, todos ellos *lightweight*, es decir, ya no se usan componentes "*peer*" dependientes del sistema operativo, y además, *Swing* está totalmente escrito en Java. Todo ello redunda en una mayor funcionalidad en manos del programador, así como en la posibilidad de mejorar en gran medida la *cosmética* de las interfaces gráficas de usuario.

Son muchas las ventajas que ofrece el uso de *Swing*. Por ejemplo, la navegación con el teclado es automática, cualquier aplicación Swing se puede utilizar sin ratón, sin tener que escribir ni una línea de código adicional. Las etiquetas de información, o "*tool tips*", se pueden crear con una sola línea de código. Además, Swing aprovecha la circunstancia de que sus componentes no están renderizados sobre la pantalla por el sistema operativo para soportar lo que se llama "*pluggable look and feel*", es decir, que la apariencia de la aplicación se adapta dinámicamente al sistema operativo y plataforma en que esté corriendo.

Los componentes Swing no soportan el modelo de Eventos de Propagación, sino solamente el modelo de Delegación incluido desde el JDK 1.1; por lo tanto, si se van a utilizar componentes Swing, se debe programar exclusivamente en el modelo de Delegación. Swing utiliza el paradigma *Modelo-Vista-Controlador*, introducido hace ya bastante tiempo por el lenguaje *Smalltalk*, que permite reducir el esfuerzo de programación necesario para la implementación de sistemas múltiples y sincronizados de los mismos datos. La figura 14.1 compara las diferentes arquitecturas que se utilizan en AWT, muy dependiente de la máquina, y Swing, mucho más flexible.

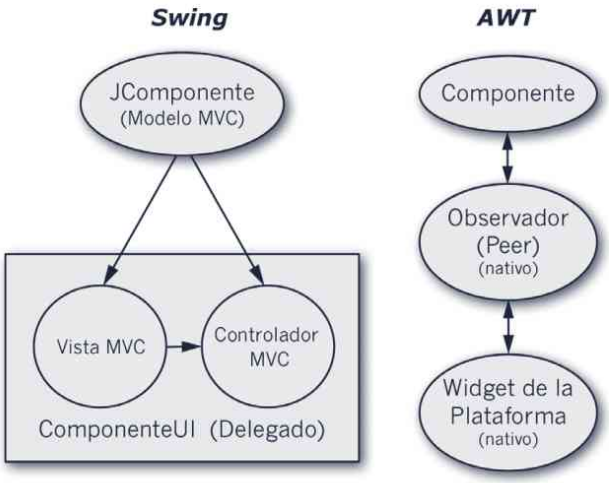


Figura 14.1

El paso de AWT a Swing es muy sencillo y no hay que descartar nada de lo que se haya hecho con el AWT. Afortunadamente, los programadores de Swing han tenido compasión y, en la mayoría de los casos es suficiente con añadir una "J" al componente AWT para que se convierta en un componente Swing.

Es muy importante entender y asimilar el hecho de que Swing es una extensión del AWT, y no un sustituto encaminado a reemplazarlo. Aunque esto sea verdad en algunos casos en que los componentes de Swing se corresponden a componentes del AWT; por ejemplo, el **JButton** de Swing puede considerarse como un sustituto del **Button** del AWT, y una vez que se usen los botones de Swing se puede tomar la decisión de no volver a utilizar jamás un botón de AWT, pero, la funcionalidad básica de Swing descansa sobre el AWT.

Para iniciar la entrada en Swing, qué mejor que implementar de nuevo otra versión del saludo inicial, pero con componentes Swing, así que la versión del "*Hola Mundo!*", se convierte ahora en `JHolaMundo.java`, cuyo código es el que sigue.

```
import javax.swing.*;

public class JHolaMundo extends JFrame {
  public static void main( String argv[] ) {
    new JHolaMundo();
    }

  JHolaMundo() {
    JLabel hola = new JLabel( "Hola Mundo!" );
    getContentPane().add( hola,"Center" );
    setSize( 200,100);
    setVisible( true );
    }
  }
```

El ejemplo `Java1401.java`, aunque ya un poco más en serio, también es muy sencillo y en él se pueden observar mejor los cambios que introduce Swing, que son casi exclusivamente de nomenclatura.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Java1401 extends JPanel {
  JButton boton1 = new JButton( "JButton 1" );
  JButton boton2 = new JButton( "JButton 2" );
  JTextField texto = new JTextField( 20 );

  public Java1401() {

    ActionListener al = new ActionListener() {
      public void actionPerformed( ActionEvent evt ) {
        String nombre = ( (JButton)evt.getSource()).getText();
        texto.setText( nombre+" Pulsado" );
        }
      };

    boton1.addActionListener( al );
    boton1.setToolTipText( "Soy el JBoton 1" );
    add( boton1 );

    boton2.addActionListener( al );
    boton2.setToolTipText( "Soy el JBoton 2" );
    add( boton2 );

    texto.setToolTipText( "Soy el JCampoDeTexto" );
    add( texto );
    }

  public static void main( String args[] ) {
    JFrame ventana = new JFrame( "Tutorial de Java, Swing" );

    ventana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    ventana.getContentPane().add(
      new Java1401(),BorderLayout.CENTER );

    ventana.setSize( 300,100 );
    ventana.setVisible( true );
    }
  }
```

Si se exceptúa la nueva sentencia `import`, el resto del código parece de AWT con una **J** delante de cada nombre de componente. Además, no se puede añadir algo con *add()* a un **JFrame**, sino que hay que tener antes su contenido, tal como muestra tanto este ejemplo, como el anterior del saludo. En resumen, con una simple conversión se tiene toda la potencia de Swing.

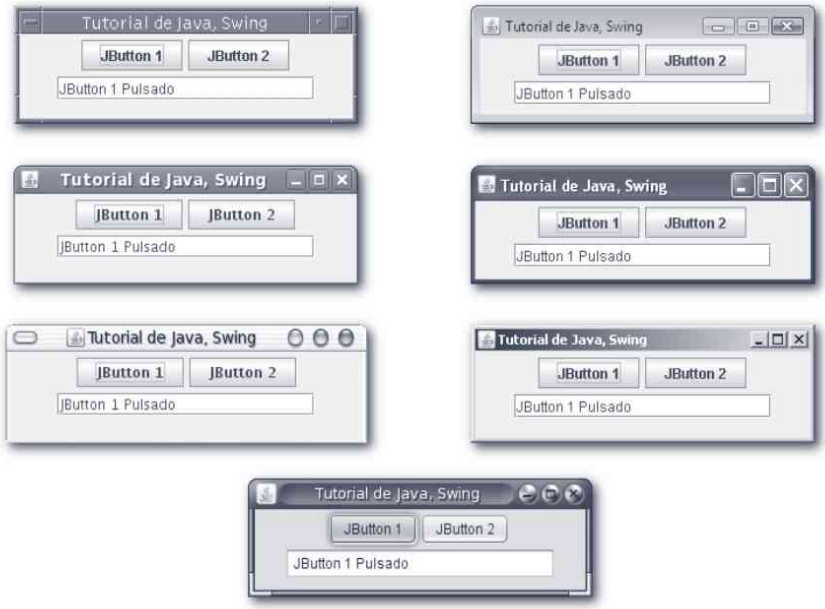


Figura 14.2

En la figura 14.2, las imágenes corresponden a la captura de la ventana que aparece en pantalla tras la ejecución del ejemplo en Solaris CDE, Linux Fedora y Mac OS-X en la parte izquierda y en Windows Vista, Windows XP y Windows 2000 en la derecha, donde se puede observar que el parecido con el AWT se va alterando un poco, a mejor por supuesto, debido a que se está utilizando un *look*, o apariencia gráfica, diferente, en este caso el que *Sun Microsystems* llama **Ocean** , incorporado al arsenal de Swing a partir del J2SE 5, para sustituir al *look* **Metal** utilizado por defecto en las primeras versiones de Swing. En las versiones venideras del JDK, Sun incorporará el *look* **Nimbus** por defecto (que corresponde a la ventana inferior en la figura anterior, sobre Sun Java Desktop 3), que proporciona una apariencia más profesional y actual que los dos anteriores. No obstante, se puede también seleccionar el tipo de botones, cajas, textos, etc. con la apariencia de *Windows*, *Motif* o *Mac*.

ARQUITECTURA DE SWING

La arquitectura de una aplicación Swing es la que se muestra en la figura 14.3.

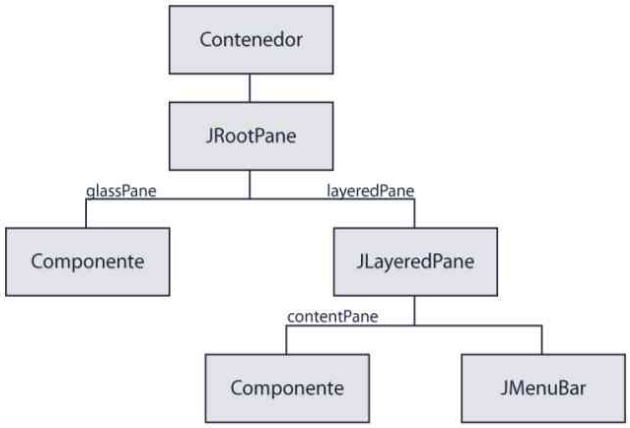


Figura 14.3

Los componentes que aparecen en la figura y las responsabilidades que les corresponden son las siguientes:

1. *Contenedor*. Es el que soporta la ventana principal de la aplicación, contiene el panel raíz sobre el que se realizan todas las acciones.
2. *Panel Raíz* (**JRootPane**). Es el componente principal de la jerarquía.
3. *Panel Contenedor*. Es el que actúa como padre de todos los componentes que se añaden al contenedor. Contiene el gestor de posicionamiento de los componentes.
4. *Panel de Capas* (**JLayeredPane**). Contiene la barra de menú y proporciona profundidad al contenedor permitiendo que los componentes puedan solaparse unos con otros.
5. *Barra de Menú* (**JMenuBar**). Presenta la barra de menú superior tradicional de todas las aplicaciones y los menús desplegables cuando se selecciona alguna de sus opciones.
6. *Panel Transparente*. Es un componente que se coloca por encima de todos los demás en el panel raíz y da soporte a la característica de *arrastrar y soltar*.

Para completar la figura anterior, es necesario tratar por separado el panel de capas, **JLayeredPane**, que se encuentra dividido en varias capas, cada una con un propósito distinto, tal como se muestra en la figura 14.4.

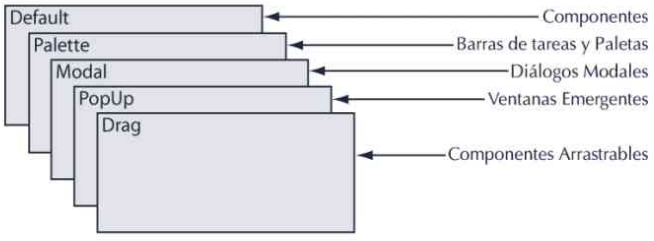


Figura 14.4

BORDES

La clase **JComponent** también contiene un método llamado *setBorder()*, que permite colocar diferentes bordes a un componente visible. El ejemplo `Java1402.java` genera los diferentes tipos de borde que están disponibles y que se reproducen en la figura 14.5.

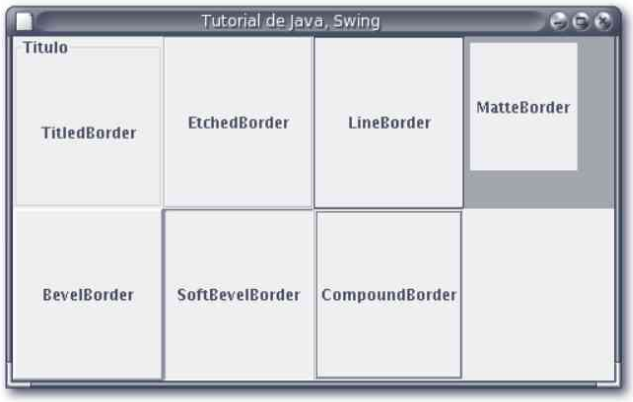


Figura 14.5

Para generar la ventana anterior, el código utiliza el método *creaBorde()* que crea un **JPanel** y le coloca un borde diferente en cada caso y, además, coloca el nombre del borde en medio del panel.

```
static JPanel creaBorde( Border b ) {
  JPanel panel = new JPanel();
  String str = b.getClass().toString();
  str = str.substring( str.lastIndexOf('.') + 1 );
  panel.setLayout( new BorderLayout() );
  panel.add(new JLabel( str,JLabel.CENTER ),BorderLayout.CENTER );
  panel.setBorder( b );
  return( panel );
  }

public Java1502() {
  setLayout( new GridLayout( 2,4 ) );
  add( creaBorde( new TitledBorder("Titulo") ) );
  add( creaBorde( new EtchedBorder() ) );
  add( creaBorde( new LineBorder(Color.blue) ) );
  add( creaBorde( new MatteBorder(5,5,30,30,Color.green) ) );
  add( creaBorde( new BevelBorder(BevelBorder.RAISED) ) );
  add( creaBorde( new SoftBevelBorder(BevelBorder.LOWERED) ) );
  add(creaBorde(new CompoundBorder(
    new EtchedBorder(),new LineBorder(Color.red) ) ) );
  }
```

Muchos de los ejemplos que se desarrollarán en este capítulo van a usar el **TitledBorder**, pero el resto de los bordes son igual de fáciles de utilizar. También es posible crear bordes propios y colocarlos dentro de botones, etiquetas, etc.; virtualmente en cualquier elemento que derive de **JComponent**.

ETIQUETAS

Las etiquetas, junto con los botones y las cajas de selección, son uno de los componentes más básicos de toda interfaz de usuario, independientemente de que se hayan visto otros hasta ahora, o el lector difiera de esta clasificación. Y el más simple de todos ellos es la etiqueta, que se limita a presentar textos en pantalla. Swing introduce la clase **JLabel** para presentar estos textos en pantalla, siendo mucho más versátil que la clase correspondiente del AWT. En Swing, al derivar de **JComponent**, la clase **JLabel** implementa todas las características inherentes a los componentes Swing, como pueden ser los aceleradores de teclado, bordes y demás.

Si se echa un vistazo a los constructores de la clase **JLabel**, se puede observar claramente que Swing ofrece un API mucho más rico, presentando constructores con habilidades no disponibles en el AWT. El ejemplo `Java1403.java`, crea varias instancias de la clase **JLabel** utilizando algunas de las características que permite Swing, como por ejemplo la utilización de gráficos, que resulta extremadamente simple. Al igual que en los ejemplos que siguen, no se reproduce el código completo del ejemplo, que el lector puede consultar en el código fuente de los ejemplos que acompañan al libro.

```
public Java1403() {
  setLayout( new GridLayout( 2,2 ) );
  JLabel etiq1 = new JLabel();
  etiq1.setText( "Etiqueta1" );
  add( etiq1 );
  JLabel etiq2 = new JLabel( "Etiqueta2" );
  etiq2.setFont( new Font( "Helvetica",Font.BOLD,18 ) );
  add( etiq2 );
  Icon imagen = new ImageIcon( "imagenes/star0.gif" );
  JLabel etiq3 = new JLabel( "Etiqueta3",
    imagen,SwingConstants.CENTER );
  etiq3.setVerticalTextPosition( SwingConstants.TOP );
  add( etiq3 );
  JLabel etiq4 = new JLabel( "Etiqueta4",SwingConstants.RIGHT );
  add( etiq4 );
  }
```

La figura 14.6 es el resultado de la ejecución del programa anterior, donde se puede observar que en la `Etiqueta2` se muestra el cambio de fuente de caracteres y en la `Etiqueta3` se incluye una imagen. En este último caso, el tamaño de la imagen es el que determina el tamaño mínimo de la etiqueta.



Figura 14.6

El tipo de letra con que se presenta el texto de la etiqueta se puede cambiar fácilmente, basta con crear una nueva fuente de caracteres e invocar al método *setFont()* de la clase **JLabel** para que el cambio surta efecto. La fuente de caracteres puede ser cualquiera de las estándar, dentro de los tamaños soportados por la Máquina Virtual Java de la plataforma. El control de la fuente de caracteres utilizada para la etiqueta se puede realizar con una línea de código semejante a la siguiente:

```
label.setFont( new Font( "Dialog",Font.PLAIN,12 ) );
```

Los colores del texto y del fondo de la etiqueta también se pueden cambiar de forma muy sencilla, mediante la invocación de los métodos *setForeground()* y *setBackground()*.

BOTONES

Swing añade varios tipos de botones y cambia la organización de la selección de componentes: todos los botones, cajas de selección, botones de selección y cualquier opción de un menú deben derivar de **AbstractButton**. El ejemplo `Java1404.java`, muestra los diferentes tipos de botones que están disponibles ante el programador a través de Swing.

```
public Java1404() {
  add( new JButton( "JButton" ) );
  add( new JToggleButton( "JToggleButton") );
  add( new JCheckBox( "JCheckBox" ) );
  add( new JRadioButton( "JRadioButton" ) );
  }
```

La figura 14.7 corresponde a la captura de la ejecución del programa anterior, y reproduce la apariencia de estos tipos de botones implementados por Swing.



Figura 14.7

El componente **JButton** parece igual que el botón que hay en el AWT, pero se pueden hacer muchas más cosas con él. Todos los botones, tienen ahora la posibilidad de incorporar imágenes a través del objeto **Icon**, que se puede asignar a cualquier tipo de botón. E incluso se pueden asignar varios iconos a un mismo botón para visualizar los diferentes estados en que pueda encontrarse dicho botón, tal como muestra la figura 14.8 que corresponde a la captura de la ejecución del ejemplo `Java1405.java`.



Figura 14.8

```
public class Java1405 extends JPanel {
  public Java1405() {
    ImageIcon izq = new ImageIcon( "imagenes/left.gif" );
    ImageIcon izqRollover =
      new ImageIcon( "imagenes/leftRollover.gif" );
    ImageIcon izqDown = new ImageIcon( "imagenes/leftDown.gif" );
    // Se le indica el texto que va a presentar y cuál es la
    // imagen que debe usar en estado de reposo
    JButton boton = new JButton( "Boton",izq );
    // Ahora se asignan las otras imágenes a los diferentes estados,
    // una para cuando se pulsa o hunde el botón y otra para
    // cuando el ratón pasa por encima del botón
    boton.setPressedIcon( izqDown );
    boton.setRolloverIcon( izqRollover );
    boton.setRolloverEnabled( true );
    boton.setToolTipText( "Boton con imagenes asociadas" );
    add( boton );
    }
```

Tanto en el caso de **AbstractButton** como en el de **JLabel** se ha incorporado la posibilidad de que se pueda incluir un carácter *mnemotécnico*, que realice la acción asociada al botón cuando se pulse la tecla correspondiente a ese carácter en el teclado. Este carácter se puede establecer a través de los métodos *setMnemonic()* y *setDisplayedMnemonic()*.

GRUPOS DE BOTONES

Si se quieren botones de selección única, los conocidos como *botones radio*, aquellos que tienen la particularidad de que solamente uno puede estar seleccionado a la vez, hay que crearse un grupo de botones, añadiendo botones a ese grupo uno a uno. Pero, Swing permite que cualquier **AbstractButton** pueda ser añadido a un **ButtonGroup**.

La figura 14.9 muestra los grupos de botones con la apariencia correspondiente a Swing, según se obtiene la ventana tras la ejecución del ejemplo `Java1406.java`, que crea varios paneles de botones.



Figura 14.9

Para evitar la repetición del código correspondiente a la incorporación de cada uno de los botones, en el ejemplo se utiliza la *reflexión* para generar los grupos de diferentes tipos de botones. Esto se puede ver en *creaPanelBotones()*, que crea un grupo de botones y un **JPanel**, y para cada **String** en el array que figura como segundo argumento de *creaPanelBotones()*, añade un objeto de la clase indicada por el primero de los argumentos.

```
public class Java1406 extends JPanel {
  static String ids[] = {
    "Mortadelo","Filemon","Carpanta",
    "Rompetechos","Pepe Gotera","Otilio",
    };

  static JPanel creaPanelBotones( Class bClass,String ids[] ) {
    ButtonGroup botones = new ButtonGroup();
    JPanel panel = new JPanel();
    String titulo = bClass.getName();
    titulo = titulo.substring( titulo.lastIndexOf('.')+1 );
    panel.setBorder( new TitledBorder( titulo ) );
    for ( int i=0; i < ids.length; i++ ) {
      AbstractButton botonAbs = new JButton( "fallo" );
      try {
        // Se utiliza el constructor dinámico al que se pasa una
        // cadena como argumento
        Constructor ctor = bClass.getConstructor( new Class[] {
          String.class
          } );
        // Se crea un nuevo objeto del tipo del botón
        botonAbs = ( AbstractButton )ctor.newInstance( new Object[]{
          ids[i]
          } );
      } catch( Exception e ) {
        System.out.println( "No puedo crear " + bClass );
        }
      botones.add( botonAbs );
      panel.add( botonAbs );
      }

    return( panel );
    }
```

```
public Java1406() {
  add( creaPanelBotones( JButton.class,ids ) );
  add( creaPanelBotones( JToggleButton.class,ids ) );
  add( creaPanelBotones( JCheckBox.class,ids ) );
  add( creaPanelBotones( JRadioButton.class,ids ) );
  }
```

El título se toma del nombre de la clase, eliminándole toda la información referente al camino en que se encuentra en disco. El objeto **AbstractButton** es inicializado a un **JButton** que tiene la etiqueta "*Fallo*", para que si se ignora el mensaje de la excepción, todavía se pueda ver en la pantalla que hay problemas. El método *getConstructor()* genera un objeto **Constructor** que obtiene el array de argumentos de tipos en el array **Class** pasado a *getConstructor()*. Y ya, todo lo que hay que hacer es llamar a *newInstance()*, pasarle un array de **Object** conteniendo los argumentos actuales; en este caso concreto, la cadena obtenida desde el array de `ids`.

Ahora se añade un poco de complejidad a lo que antes era un proceso muy simple, para conseguir que funcione el grupo con un único botón seleccionado. Hay que crear un grupo de botones e indicarle a cada uno de los botones de ese grupo cuál es el comportamiento que se desea de él. Cuando se ejecute el programa, se observará que todos los botones, excepto el **JButton**, disponen de ese comportamiento de exclusividad, marcándose y desmarcándose automáticamente, en función de la pulsación del ratón sobre el botón.

LISTAS Y CAJAS COMBINADAS

Las listas y cajas "*combo*" en Swing funcionan del mismo modo que lo hacían en el AWT, aunque tienen incrementada la funcionalidad a través de algunas funciones de conveniencia añadidas. Por ejemplo, **JList** tiene un constructor al que se puede pasar un array de objetos **String** para que los presente. El ejemplo `Java1407.java`, muestra el uso básico de estos dos componentes.

```
public Java1407() {
  setLayout( new GridLayout( 2,1 ) );
  JList lista = new JList( Java1406.ids );
  add( new JScrollPane( lista ) );
  JComboBox combo = new JComboBox();
  for ( int i=0; i < 100; i++ )
    combo.addItem( Integer.toString( i ) );
  add( combo );
  }
```



Figura 14.10

La figura 14.10 es el resultado que se obtiene en pantalla tras la ejecución del ejemplo, y después de haber seleccionado uno de los elementos en la caja *combo* y abrirla para proseguir la selección de otra de las opciones o elementos que se permite elegir.

Lo que se ha mantenido con respecto a las listas de antes es que los objetos **JList** no proporcionan automáticamente la posibilidad de *scroll* o desplazamiento del contenido, que es algo que se espera que haga automáticamente, y resulta un poco chocante. No obstante, el añadir *scroll* a listas es sumamente sencillo, ya que es suficiente con incluir la **JList** en un **JScrollPane**, y todos los detalles del desplazamiento del contenido serán ya manejados bajo la responsabilidad del sistema.

Aunque lo más frecuente es que cada elemento seleccionable de una lista sea una etiqueta, Swing proporciona al componente **JList** también la posibilidad de presentar gráficos, con o sin texto asociado, y también proporciona un control adicional de eventos para adecuarse a los que se producen en la manipulación de este tipo de componentes. Es decir, que en una lista también se pueden incluir instancias de clases como **JButton**, **JTextField**, **JCheckBox**, e incluso **JTextArea**, que es un componente multilínea. En el ejemplo `Java1408.java` se muestra precisamente una lista en la que se utilizan instancias de la clase **JTextArea**, para mostrar los valores que se cargan en la lista.

Cada valor incluye el carácter de nueva línea, `\n`, para forzar el salto de línea en medio del texto. El código del ejemplo se reproduce a continuación.

```
class Java1408 extends JPanel {
  private JList lista;

  public Java1408() {
    // Se crea un panel para poder contener todos los componentes
    setLayout( new BorderLayout() );
    // Se crean algunos elementos para poder seleccionar de la lista
    String datosLista[] = {
      "Capítulo 4\nLenguaje Java",
```

```
      "Capítulo 5\nProgramas Básicos en Java",
      "Capítulo 6\nConceptos Básicos de Java",
      "Capítulo 7\nClases Java",
      "Capítulo 8\nAlmacenamiento de Datos",
      "Capítulo 9\nFicheros en Java"
      };
    // Creamos una lista con un controlador propio que va a permitir
    // la personalización en la presentación en pantalla de cada uno
    // de los ítems que se pueden seleccionar de esa lista
    lista = new JList( datosLista );
    lista.setCellRenderer( new MiRendererDeLista() );
    add( lista, BorderLayout.CENTER );
    }

  // Clase anidada que implementa el visualizador especial para las
  // celdas que componen la lista
  class MiRendererDeLista extends JTextArea
    implements ListCellRenderer {
    public Component getListCellRendererComponent(
      JList lista,Object valor,int indice,
      boolean seleccionado, boolean conFoco ) {
      setBorder( new BevelBorder( BevelBorder.RAISED ) );
      // Presenta el text correspondiente al ítem
      setText( valor.toString() );
      // Pinta en los colores indicados, con la fuente seleccionada...
      if ( seleccionado ) {
        // ... en el caso de un ítem marcado (rojo/blanco)
        setBackground( Color.red );
        setForeground( Color.white );
        }
      else {
        // ... en el caso de un ítem no marcado (gris/negro)
        setBackground( Color.lightGray );
        setForeground( Color.black );
        }
      return( this );
      }
    }
```

El código es bastante simple. La parte más interesante es la correspondiente a la clase anidada **MiRendererDeLista**, que implementa un controlador diferente para cada celda de la lista, extendiendo en este caso a **JTextArea** en lugar de **JLabel**, como suele ser habitual. El resultado de la ejecución del ejemplo lo muestra la figura 14.11.



Figura 14.11

Aunque el componente **JTextArea** también tiene sus limitaciones, como son el no permitir el uso de gráficos, se puede observar que las posibilidades que se presentan al programador se han visto tremendamente incrementadas con la inclusión de este tipo de componente y con las nuevas características que proporciona Swing. En este caso, incluso es posible hacer que la altura de la celda esté recortada, en el caso de utilizar fuentes de caracteres muy grandes, debido a que la altura de cada uno de los elementos seleccionables de la lista se determina en el momento en que son construidos.

JComboBox y **JList** a la hora de renderizar celdas consumen bastante tiempo, lo cual hace que el uso de listas muy largas sea complicado, ya que cada una de las celdas se comprueba individualmente. No obstante, es posible utilizar prototipos, de forma que un prototipo solamente se compruebe una única vez y se aplique a todas las celdas. Para el control de estos prototipos, el desarrollador dispone de los métodos *setPrototypeDisplayValue()* en el caso de **JComboBox** y *setPrototypeCellValue()* en el caso de **JList**. El siguiente trozo de código muestra un ejemplo de uso de este último método.

```
String contenido;
Vector v = new Vector();
for( int i=0; i < 10000; i++ ) {
  contenido = "Número: " + Integer.toString( i );
  v.addElement( contenido );
  }
JList lista = new JList( v );
String patron = "Número: 9999";
lista.setPrototypeCellValue( patron );
```

Una modificación del control de **JComboBox** es **JSpinner**, que permite diseñar controles de selección en los cuales el usuario puede ir variando los elementos a seleccionar uno a uno. La diferencia con **JComboBox** es que no se presenta una lista desplegable con los elementos seleccionables, sino que van apareciendo en la zona de visualización en orden ascendente o descendente, según el usuario utilice los botones de selección arriba o abajo, respectivamente. El ejemplo `Java1409.java` muestra el uso de los distintos modelos de **JSpinner** que implementa Swing.

La figura 14.12 muestra la ventana generada tras la ejecución del ejemplo, en donde el lector puede comprobar la apariencia de los distintos modelos de este componente.



Figura 14.12

Swing proporciona clases adicionales para la creación de los modelos de datos que se pueden integrar en este componente, dependiendo del tipo de entrada que se desee presentar al usuario. La clase **SpinnerDateModel** acepta la entrada o selección de fechas, presentando en la zona de visualización del componente fechas que se pueden indicar a través de las constantes definidas en la clase **Calendar** , por ejemplo, `Calendar.WEEK_OF_MONTH` cambiará la fecha presentada de semana en semana, en períodos de 7 días. La clase **SpinnerListModel** permite la selección de elementos dentro de una lista predefinida. La clase **SpinnerNumberModel** acepta como entrada la selección de números entre un rango determinado y con un intervalo prefijado.

El código del ejemplo `Java1409.java` se reproduce a continuación, en donde el lector puede comprobar el uso de las clases anteriores para permitir al usuario la selección de los diferentes tipos de valores que se declaran en cada componente **JSpinner**.

```
public class Java1409 extends JPanel {
  JSpinner spinFecha1;
  JSpinner spinFecha2;
  JSpinner spinNumero;
  JSpinner spinLista;

  public Java1409() {
    setLayout( null );
    // Creamos un selector que va a permitir ir avanzando día a día,
    // es el que utiliza el modelo de Fecha por defecto.
    JLabel labFecha1 = new JLabel(  "Selecciona un día:" );
    labFecha1.setBounds( 5,10,120,24 );
    add( labFecha1 );
    SpinnerDateModel modeloFecha1 = new SpinnerDateModel();
    modeloFecha1.setCalendarField( Calendar.DAY_OF_WEEK );
    spinFecha1 = new JSpinner( modeloFecha1 );
    spinFecha1.setBounds( 160,10,120,24 );
    add( spinFecha1 );
    // Creamos un selector de fecha que va a permitir ir avanzando de
    // semana en semana, ya que en el campo correspondiente se indica
    // que se debe mostrar la semana del mes
    JLabel labFecha2 = new JLabel( "Selecciona una fecha:" );
    labFecha2.setBounds( 5,40,150,24 );
    add( labFecha2 );
    SpinnerDateModel modeloFecha2 = new SpinnerDateModel();
    modeloFecha2.setCalendarField( Calendar.WEEK_OF_MONTH );
    spinFecha2 = new JSpinner( modeloFecha2 );
    // Fijamos el formato con que se va a presentar la fecha en el
    // campo de selección
    JSpinner.DateEditor editor = new JSpinner.DateEditor(
      spinFecha2, "dd MMMMM yyyy");
    spinFecha2.setEditor( editor );
    spinFecha2.setBounds( 160,40,120,24 );
    add( spinFecha2 );
    // Se crea un elemento de selección que permite ir seleccionando
    // de forma ascendente o descendente, números con un intervalo
    // fijado en 0,5
    JLabel labNumero = new JLabel( "Selecciona un número:" );
    labNumero.setBounds( 5,70,150,24 );
```

```
add( labNumero );
SpinnerNumberModel modeloNumero =
  new SpinnerNumberModel( 1,0.0,1000.0,0.5 );
spinNumero = new JSpinner( modeloNumero );
spinNumero.setBounds( 160,70,120,24 );
add( spinNumero );
// Se crea un elemento de selección que permite elegir entre los
// distintos colores que integran la lista
String[] colores = { "Amarillo","Marrón","Azul","Rojo","Naranja",
  "Negro","Blanco","Morado" };
JLabel labLista = new JLabel( "Selecciona un color:" );
labLista.setBounds( 5,100,120,24 );
add( labLista );
SpinnerListModel modeloLista = new SpinnerListModel( colores );
spinLista = new JSpinner( modeloLista );
spinLista.setValue( "Azul" );
spinLista.setBounds( 160,100,120,24 );
add( spinLista );
}
```

TEXTO

Swing también introduce nuevos componentes dentro de la manipulación de textos. Así la clase **JTextArea** actúa como sustituto de la clase **TextArea** del AWT, la clase **JTextField** sustituye **TextField** del AWT y se incorporan las clases **JPasswordField** que viene a ser equivalente al uso de **JTextField** junto con el método *setEchoChar()*, **JFormattedTextField** que permite la entrada de texto con formato, de manera que resulte simple la introducción de números de teléfono o números de tarjeta de crédito, por ejemplo; y la clase **JTextPane** que permite que se presente el texto con diferentes fuentes de caracteres, colores, tamaños, etc.

El cambio de las características del texto que se presenta se realiza a través de estilos. Un estilo es un conjunto de características como son **FontSize**, **ForegroundColor**, **isBold**, **isItalic**, etc. y se manipula a través de constantes, como muestra la línea siguiente:

```
StyleConstants.setForeground( estilo,Color.red )
```

El componente **JTextPane**, que es el que permite estos cambios, siempre tiene un estilo por defecto como modelo, **DefaultStyleDocument**. En la figura 14.13, la primera línea se ha escrito nada más arrancar el programa, es decir, con el estilo por defecto de la aplicación, y las siguientes se introdujeron cambiando al estilo correspondiente el color del texto.

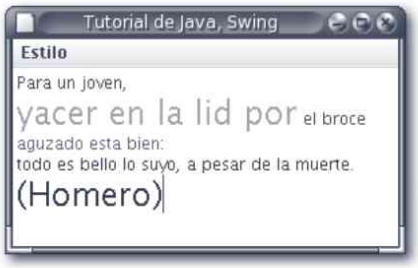


Figura 14.13

Los estilos se pueden aplicar a todo un documento o solamente a parte de él. El estilo por defecto, **DefaultStyleDocument**, tiene un **StyleContext** que es el que puede manejar estilos diferentes. El ejemplo `Java1410.java`, cuya imagen se mostraba antes, fija varios estilos y permite que se seleccione cualquiera de ellos para la introducción del texto.

```
public class Java1410 extends JPanel implements ActionListener {
  private Style estiloRojo,estiloVerde,estiloAzul;
  private JTextPane texto;

  public Java1410() {
    setLayout( new BorderLayout() );
    add( creaMenu(),BorderLayout.NORTH );
    JTextPane texto = creaEditor();
    add( texto,BorderLayout.CENTER );
    }

  private JMenuBar creaMenu() {
    JMenuBar menu = new JMenuBar();
    JMenu estilo = new JMenu( "Estilo" );
    menu.add( estilo );
    JMenuItem mi = new JMenuItem( "Rojo" );
    estilo.add( mi );
    mi.addActionListener( this );
    mi = new JMenuItem( "Verde" );
    estilo.add( mi );
    mi.addActionListener( this );
    mi = new JMenuItem( "Azul" );
    estilo.add( mi );
    mi.addActionListener( this );
    return( menu );
    }

  public void actionPerformed( ActionEvent evt ) {
    Style estilo = null;
    String color = (String)evt.getActionCommand();
    if ( color.equals( "Rojo" ) ) {
      estilo = estiloRojo;
      }
    else if ( color.equals( "Azul" ) ) {
      estilo = estiloAzul;
      }
    else if ( color.equals( "Verde" ) ) {
      estilo = estiloVerde;
      }
```

```
    texto.setCharacterAttributes( estilo,false );
    }

  private JTextPane creaEditor() {
    StyleContext sc = creaEstilos();
    DefaultStyledDocument doc = new DefaultStyledDocument( sc );
    return( texto = new JTextPane( doc ) );
    }

  private StyleContext creaEstilos() {
    StyleContext sc = new StyleContext();
    estiloRojo = sc.addStyle( null,null );
    StyleConstants.setForeground( estiloRojo,Color.red );
    estiloVerde = sc.addStyle( null,null );
    StyleConstants.setForeground( estiloVerde,Color.green );
    StyleConstants.setFontSize( estiloVerde,24 );
    estiloAzul = sc.addStyle( null,null );
    StyleConstants.setForeground( estiloAzul,Color.blue );
    return( sc );
    }
```

Swing también incorpora receptores de eventos de conveniencia, es decir, que se pueden utilizar para que hagan acciones solos, por ejemplo **StyledEditorKit** dispone de un **ActionListener** que realiza todas las acciones que debe realizar el método *actionPerformed()*, sin necesidad de intervención del programador. Estos receptores se pueden añadir a cualquier componente, por ejemplo, a un **MenuItem**, de la siguiente forma:

```
ActionListener a = new StyledEditorKit.ForegroundAction(
  "set-foreground-red",Color.red );
mi.addActionListener( a );
```

Otro receptor de eventos que incorpora Swing corresponde al cursor de texto, **CaretListener**. El ejemplo `Java1411.java` muestra cómo se puede controlar perfectamente la posición en la que se encuentra el cursor, tanto respecto a la fila como a la columna del texto por el que se desplaza. La ventana que genera la aplicación permite la inserción de texto y el desplazamiento del cursor de texto, mostrando en la parte superior e inferior de la zona de texto la columna y fila en la que se encuentra situado el cursor.

TOOL TIPS

Casi todas las clases que se usan para crear las interfaces de usuario se derivan de **JComponent**. Esta clase contiene un método llamado *setToolTipText(String)*. Así que se puede indicar un texto de información virtualmente en cualquier lugar y lo único que se necesita es invocar a este método, indicando el texto correspondiente al mensaje. Por ejemplo, al primer programa del capítulo tras la aplicación de saludo, se incorporaron dos mensajes de información de este tipo, uno a cada uno de los botones y otro al campo de texto:

```
. . .
```

```
boton1.addActionListener( al );
boton1.setToolTipText( "Soy el JBoton 1" );
add( boton1 );
boton2.addActionListener( al );
boton2.setToolTipText( "Soy el JBoton 2" );
add( boton2 );
texto.setToolTipText( "Soy el JCampoDeTexto" );
add( texto );
. . .
```

y cuando el ratón permanezca sobre cada **JButton** o sobre el **JTextField**, en este caso, pero sobre un objeto **JComponent** en el caso más general, durante un período de tiempo predeterminado, una pequeña cajita conteniendo el texto aparecerá al lado del puntero del ratón, también en este caso, indicando en botón sobre el cual se encuentra situado el cursor del ratón o que se encuentra sobre el campo de texto.

Hay ocasiones en que es necesario obtener más flexibilidad de la que ofrece el uso de este tipo de mensajes de información, por ejemplo, en el caso de un componente **JTree**, **JList** o **JTable**, puede resultar interesante que el texto de ese mensaje esté basado en el contenido del nodo del árbol, del elemento de la lista o de la celda de la tabla.

Para conseguir crear mensajes de información personalizados en el caso de estos componentes, es posible sobrescribir el método *getToolTipText()*, o también personalizar el control sobre los mensajes del propio componente.

En el primero de los casos, el método *getToolTipText()* necesita como parámetro un objeto de tipo **MouseEvent**, el cual contiene las coordenadas del cursor, siendo responsabilidad del programador determinar sobre qué elemento se encuentra situado el cursor. Luego, es necesario cambiar el texto del mensaje y devolverlo. El componente debe registrarse manualmente con el **ToolTipManager** invocando al método *registerComponent()*, en caso de no haber utilizado la llamada al método *setToolTipText()*, el cual se encarga de realizar ese registro automáticamente.

El ejemplo `Java1412.java` utiliza este mecanismo, obteniendo los valores de las propiedades del sistema para presentarlos como mensaje de texto sobre cada uno de los elementos de la lista creada con esas propiedades.

```
public class Java1412 extends JList {
  DefaultListModel modelo;
  Properties propiedadesTip;

  public Java1412( Properties propiedades ) {
    modelo = new DefaultListModel();
    setModel( modelo );
    ToolTipManager.sharedInstance().registerComponent( this );
    propiedadesTip = propiedades;
    Enumeration enu = propiedades.propertyNames();
    while( enu.hasMoreElements() )
      modelo.addElement( enu.nextElement() );
    }
```

```
  public String getToolTipText( MouseEvent evt ) {
    Point p = evt.getPoint();
    int posicion = locationToIndex( p );
    String clave = (String)modelo.getElementAt( posicion );
    String mensajeTip = propiedadesTip.getProperty( clave );
    return( mensajeTip );
    }
```

Utilizando la segunda posibilidad, para personalizar el controlador del elemento será necesario un poco más de preparación en un principio, pero no será necesario determinar qué elemento se encuentra bajo el cursor en tiempo de ejecución. No obstante, al ser el propio controlador el que genera el texto, puede que se produzca una sobrecarga de trabajo inconveniente, porque el controlador de cada elemento se estaría invocando con demasiada frecuencia. En este caso sería más útil el uso del primer método. El ejemplo `ToolTipArbolRenderer.java` muestra una implementación del segundo método de personalización de mensajes de información, en donde se reutiliza un **DefaultTreeCellRenderer** como controlador, que será el encargado de fijar el texto.

```
public class ToolTipArbolRenderer implements TreeCellRenderer {
  DefaultTreeCellRenderer render =
    new DefaultTreeCellRenderer();
  Dictionary tablaTip;

  public ToolTipArbolRenderer( Dictionary tablaTip ) {
    this.tablaTip = tablaTip;
    }

  public Component getTreeCellRendererComponent(
    JTree tree,Object value,boolean selected,boolean expanded,
    boolean leaf,int row,boolean hasFocus ) {
    render.getTreeCellRendererComponent( tree,value,selected,
      expanded,leaf,row,hasFocus );
    if( value != null ) {
      Object claveTip;
      if( value instanceof DefaultMutableTreeNode ) {
        claveTip = ((DefaultMutableTreeNode)value).getUserObject();
        }
      else {
        claveTip = tree.convertValueToText( value,selected,
          expanded,leaf,row,hasFocus );
        }
      Object mensaje = tablaTip.get( claveTip );
      if( mensaje != null )
        render.setToolTipText( mensaje.toString() );
      else
        render.setToolTipText( null );
      }
    return( render );
    }
  }
```

El programa de ejemplo `Java1413.java` registra el árbol con el **ToolTipManager** y registra el controlador. Aunque se utiliza un árbol, también se

emplean las propiedades del sistema como elementos, por lo que el árbol solamente tendrá un nivel de profundidad.

```
public class Java1413 {
  public static void main( String args[] ) {
    JFrame miFrame = new JFrame( "Tutorial de Java, Swing" );
    miFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    Properties props = System.getProperties();
    JTree arbol = new JTree( props );
    ToolTipManager.sharedInstance().registerComponent( arbol );
    TreeCellRenderer render = new ToolTipArbolRenderer( props );
    arbol.setCellRenderer( render );
    JScrollPane panel = new JScrollPane( arbol );
    miFrame.getContentPane().add( panel );
    miFrame.setSize( 300,150 );
    miFrame.setVisible( true );
    }
  }
```

La figura 14.14 muestra la ventana generada por la ejecución del ejemplo, en donde el lector puede observar el uso del mensaje de información personalizado.



Figura 14.14

ICONOS

Se puede utilizar un objeto **Icon** dentro de un objeto **JLabel**, o cualquier objeto que derive de **AbstractButton**; incluyendo **JButton**, **JCheckBox**, **JRadioButton** y los diferentes tipos de **JMenuItem**. El uso de iconos con etiquetas Swing es muy simple, tal como ya se mostraba en la clase **Java1403** y ahora en el ejemplo `Java1414.java`, en donde se exploran todas las formas en las que se pueden emplear los iconos con los botones y sus descendientes. La figura 14.15 reproduce la captura de la ejecución de este último programa, cuando el ratón llevaba varios segundos sobre la imagen correspondiente al botón, con lo cual el mensaje incorporado como **ToolTip** al botón, aparece en pantalla.



Figura 14.15

Se puede utilizar cualquier fichero gráfico soportado: *gif*, *png*, *jpg*. Para abrir ese fichero y convertirlo en imagen, simplemente se crea un objeto **ImageIcon** indicándole el nombre del fichero, y ya se puede utilizar en el código el objeto **Icon** resultante.

```
public class Java1414 extends JPanel {
  static Icon imgs[] = {
    new ImageIcon( "star0.gif" ),
    new ImageIcon( "star1.gif" ),
    new ImageIcon( "star2.gif" ),
    new ImageIcon( "star3.gif" ),
    new ImageIcon( "star4.gif" ),
    };
  JButton boton = new JButton( "JButton",imgs[3] );
  JButton boton2 = new JButton( "Deshabilita" );
  boolean mad = false;

  public Java1414() {
    boton.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent evt ){
        if( mad ) {
          boton.setIcon( imgs[3] );
          mad = false;
          }
        else {
          boton.setIcon( imgs[0] );
          mad = true;
          }
        boton.setVerticalAlignment( JButton.TOP );
        boton.setHorizontalAlignment( JButton.LEFT );
        }
      } );
    boton.setRolloverEnabled( true );
    boton.setRolloverIcon( imgs[1] );
    boton.setPressedIcon( imgs[2] );
    boton.setDisabledIcon( imgs[4] );
    boton.setToolTipText( "AleHop!" );
    add( boton );
    boton2.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent evt ){
        if( boton.isEnabled() ) {
          boton.setEnabled( false );
          boton2.setText( "Habilita" );
          }
        else {
          boton.setEnabled( true );
          boton2.setText( "Deshabilita" );
          }
        }
      } );
    add( boton2 );
    }
```

Un objeto **Icon** puede ser utilizado en muchos constructores, pero también se puede utilizar el método *setIcon()* para añadir o cambiar un icono. El ejemplo muestra cómo un **JButton**, o cualquier **AbstractButton**, puede manejar diferentes iconos que

aparecen cuando se realizan acciones sobre ese botón: se pulsa, se deshabilita o se pasa el cursor por encima (sin pulsar ningún botón del ratón físico). Con esto se consigue proporcionar al botón la apariencia de encontrarse animado.

MENÚS

Los menús están mucho mejor desarrollados y son más flexibles en Swing que los que se encuentran habitualmente en otras herramientas, incluyendo paneles y applets. La sintaxis para utilizarlos es la misma que en el AWT, y se conserva el mismo problema que también presentan los menús en el AWT, es decir, que hay que codificar los menús directamente y no hay soporte para, por ejemplo, recursos; que, entre otras cosas, facilitaría la conversión entre lenguajes de los textos de los menús. Así que, hay veces en que el código se hace demasiado largo y feo. En el ejemplo `Java1415.java`, se intenta dar un paso hacia la resolución de este problema colocando toda la información de cada menú en un array de elementos de tipo **Object** de dos dimensiones, para facilitar la introducción de cualquier cosa en el array. Este array está organizado de tal forma que la primera fila representa el nombre del menú y las siguientes filas representan los elementos u opciones del menú y sus características. Las filas del array no tienen por qué ser uniformes, porque el código sabe perfectamente dónde se encuentra, así que no importa que las filas sean completamente diferentes.

```
public class Java1415 extends JPanel {
  static final Boolean bT = new Boolean( true );
  static final Boolean bF = new Boolean( false );
  static ButtonGroup grupoBotones;

  // Clase que se utiliza para crear los distintos tipos de menús
  // que se van a presentar en la ventana
  static class TipoMenu {
    TipoMenu( int i ) {}
    };

  // Menú con elementos normales
  static final TipoMenu mi = new TipoMenu( 1 );
  // Menú con cajas de selección
  static final TipoMenu cb = new TipoMenu( 2 );
  // Menú con botones de radio
  static final TipoMenu rb = new TipoMenu( 3 );
  JTextField txt = new JTextField( 10 );
  JLabel lbl = new JLabel( "Icono Seleccionado",Java1414.imgs[0],
    JLabel.CENTER );
  ActionListener al1 = new ActionListener() {
    public void actionPerformed( ActionEvent evt ) {
      txt.setText( ((JMenuItem)evt.getSource() ).getText() );
      }
    };
  ActionListener al2 = new ActionListener() {
    public void actionPerformed( ActionEvent evt ) {
      JMenuItem mi = (JMenuItem)evt.getSource();
      lbl.setText( mi.getText() );
      lbl.setIcon( mi.getIcon() );
```

```
      }
    };

  // En estas estructuras se almacenan los datos de los menús como
  // si se tratara de los típicos recursos de X
  public Object menuArchivo[][] = {
    // Nombre del menú y tecla rápida asociada
    { "Archivo",new Character('A')},
    // Nombre, tipo, tecla rápida, receptor asociado, habilitado o no
    // para cada uno de los elementos del menú
    { "Nuevo",mi,new Character('N'),al1,bT},
    { "Abrir",mi,new Character('b'),al1,bT},
    { "Guardar",mi,new Character('G'),al1,bF},
    { "Guardar como...",mi,new Character('c'),al1,bF},
    { null}, // Separador
    { "Salir",mi,new Character('S'),al1,bT},
    };

  public Object menuEdicion[][] = {
    // Nombre del menú y tecla rápida asociada
    { "Edicion",new Character('E')},
    // Nombre, tipo, tecla rápida, receptor asociado, habilitado o no
    { "Cortar",mi,new Character('t'),al1,bT},
    { "Copiar",mi,new Character('C'),al1,bT},
    { "Pegar",mi,new Character('P'),al1,bT},
    { null}, // Separator
    { "Seleccionar Todo",mi,new Character('S'),al1,bT},
    };

. . .

  public Object barraMenu[] = {
    menuArchivo,menuEdicion,menuIconos,menuOpciones,menuAyuda,
    };

  static public JMenuBar creaMenuBarra( Object barraMenuDato[] ) {
    JMenuBar barraMenu = new JMenuBar();
    for ( int i=0; i < barraMenuDato.length; i++ )
      barraMenu.add( creaMenu((Object[][])barraMenuDato[i]) );
    return( barraMenu );
    }

  static public JMenu creaMenu( Object[][] menuDato ) {
    JMenu menu = new JMenu();
    menu.setText( (String)menuDato[0][0] );
    menu.setMnemonic( ((Character)menuDato[0][1]).charValue() );
    grupoBotones = new ButtonGroup();
    for ( int i=1; i < menuDato.length; i++ ) {
      if ( menuDato[i][0] == null )
        menu.add( new JSeparator() );
      else
        menu.add( creaMenuItem( menuDato[i] ) );
      }
    return( menu );
    }

  static public JMenuItem creaMenuItem( Object[] dato ) {
    JMenuItem m = null;
```

```
    TipoMenu tipo = (TipoMenu)dato[1];
    if ( tipo == mi )
      m = new JMenuItem();
    else if ( tipo == cb )
      m = new JCheckBoxMenuItem();
    else if ( tipo == rb ) {
      m = new JRadioButtonMenuItem();
      grupoBotones.add( m );
      }
    m.setText( (String)dato[0] );
    m.setMnemonic( ((Character)dato[2]).charValue() );
    m.addActionListener( (ActionListener)dato[3] );
    m.setEnabled( ((Boolean)dato[4]).booleanValue() );
    // Y ahora el caso opcional de los iconos
    if ( dato.length == 6 )
      m.setIcon( (Icon)dato[5] );
    return( m );
    }

  Java1415() {
    setLayout( new BorderLayout() );
    add( creaMenuBarra( barraMenu ),BorderLayout.NORTH );
    JPanel p = new JPanel();
    p.setLayout( new BorderLayout() );
    p.add( txt,BorderLayout.NORTH );
    p.add( lbl,BorderLayout.CENTER );
    add( p,BorderLayout.CENTER );
    }
```

La intención del ejemplo anterior es permitir al programador el uso de tablas simples que representen a cada menú, en vez de tener que teclear líneas enteras para la construcción de esos menús. Cada tabla genera un menú, figurando en la primera fila el nombre del menú y el acelerador de teclado y en las siguientes los datos correspondientes a cada opción del menú, que son: la cadena de caracteres que se colocará como opción, el tipo de opción de que se trata, la tecla aceleradora, el receptor de eventos de tipo **Action** que se disparará cuando la opción sea seleccionada y, finalmente, un indicador para saber si la opción está habilitada o no. Si una fila empieza por un carácter nulo, se tratará como un separador.

Para evitar la tediosa creación de múltiples objetos booleanos y banderas, se han creado los valores estáticos al comienzo de la clase, `bT` y `bF`, para representar los booleanos y diferentes objetos de la clase **TipoMenu** para describir las opciones o elementos normales del menú (`mi`), elementos con cajas de selección (`cb`) y elementos de selección única o botones de radio (`rb`). Hay que tener en cuenta que un array de **Object** puede contener solamente objetos y no valores primitivos, por lo que no se puede usar `int` sino **Integer**, etc.



Figura 14.16

El ejemplo, cuya imagen en ejecución reproduce la figura 14.16, también muestra cómo **JLabel** y **JMenuItem**, y sus descendientes, pueden manejar iconos. Un objeto **Icon** es colocado en **JLabel** a través de su constructor y modificado cuando el correspondiente elemento del menú está seleccionado.

El array `barraMenu` está ordenado de la misma forma en que se quiere que aparezcan los elementos en la barra. Se puede pasar este array al método *creaMenuBarra()*, que ya lo divide en los arrays individuales de datos del menú, pasando cada uno de estos últimos al método *creaMenu()*. Este método, por su parte, toma la primera línea del menú y crea un objeto **JMenu** a partir de ella, luego invoca al método *creaMenuItem()* para cada una de las siguientes líneas del array. Finalmente, el método *creaMenuItem()* chequea toda la línea y determina el tipo de menú y sus atributos, creando el elemento o ítem que corresponda. Al final, como se puede comprobar en el constructor del ejemplo, *Java1415()*, para crear un menú desde estas tablas, es suficiente con la invocación del constructor de la forma:

```
creaMenuBarra( barraMenu );
```

y ya todo se maneja de forma recursiva.

En el ejemplo no se tienen en cuenta los menús en cascada, pero el lector ya estará en condiciones de añadir esta característica al ejemplo en caso de necesitarla.

MENÚS POPUP

En la implementación de **JPopupMenu** hay que llamar al método *enableEvents()* y seleccionar los eventos del ratón, en vez de utilizar un receptor de eventos como sería de esperar. En versiones anteriores del JDK, la implementación de **JPopupMenu** era muy extraña, fundamentalmente debido a que no era posible obtener el foco. Actualmente, con la arquitectura de implementación del control del foco, Swing puede solicitar un cambio temporal del foco y pasárselo al objeto **JPopupMenu** que ya puede soportar el control de las teclas de desplazamiento o de la tecla de retorno. Además las clases **Popup** y **PopupFactory** son públicas para que el desarrollador pueda personalizar la apariencia de los menús *popup*.

Antes del J2SE 5 era necesario asignar los correspondientes receptores de eventos a cada opción del menú *popup* para detectar cuándo el usuario solicita la presencia del menú *popup*. Ahora ya no es necesario asignar estos receptores, sino solamente utilizar la sentencia:

```
componente.setComponentPopupMenu( menuPopup );
```

También se ha incorporado un método para permitir que los componentes hereden el menú popup de su componente padre. En el ejemplo siguiente, el campo de texto hereda el menú *popup* al utilizar la sentencia:

```
txt.setInheritsPopupMenu( true );
```

En caso contrario, al pulsar con el botón derecho sobre el campo de texto, no aparecería el menú *popup*. El valor de esta propiedad depende del componente de que se trate, en este caso, sobre el campo de texto aparecerá un menú de edición, tal como muestra la imagen derecha de la figura 14.17.

El ejemplo `Java1416.java` presenta el menú en la pantalla, figura 14.17, ante la llegada del evento correspondiente a la activación del menú. En la imagen izquierda se muestra el menú que aparece el pulsar sobre cualquier zona del interior de la ventana y la imagen derecha muestra el menú *popup* asignado al campo de texto.



Figura 14.17

El lector observará que es necesario controlar la pulsación del botón derecho del ratón para presentar el menú en pantalla.

```
public class Java1416 extends JPanel {
  private JPopupMenu popup = new JPopupMenu();
  private JTextField txt = new JTextField( 10 );

  public Java1416() {
    // Creamos el menú popup que aparecerá sobre el campo de texto
    JPopupMenu popEdicion = new JPopupMenu();
    popEdicion.add( new JMenuItem("Copiar") );
    popEdicion.add( new JMenuItem("Cortar") );
    popEdicion.addSeparator();
    popEdicion.add( new JMenuItem("Pegar") );
    txt.setComponentPopupMenu( popEdicion );
    // Dejamos que se pueda activar el menú popup desde el componente
    txt.setInheritsPopupMenu( true );
    // Añadimos el campo de texto al panel
    add( txt );
    // Creamos el menú popup que aparecerá sobre la ventana
```

```
    JMenuItem elemento = new JMenuItem( "Carpanta" );
    popup.add( elemento );
    elemento = new JMenuItem( "Rompetechos" );
    popup.add( elemento );
    elemento = new JMenuItem( "Otilio" );
    popup.add( elemento );
    popup.addSeparator();
    elemento = new JMenuItem( "Mortadelo" );
    popup.add( elemento );
    // Habilitamos los eventos de ratón sobre la ventana
    enableEvents( AWTEvent.MOUSE_EVENT_MASK );
    }

  protected void processMouseEvent( MouseEvent evt ){
    if ( evt.isPopupTrigger() )
      popup.show( evt.getComponent(),evt.getX(),evt.getY() );
    else
      super.processMouseEvent( evt );
    }
```

El mismo **ActionListener** es el que se añade a cada **JMenuItem**, que toma el texto de la etiqueta del menú y la inserta en el **JTextField**.

Una cuestión importante a la hora de implementar menús *popup*, y menús desplegables en general, es que se debe tener muy presente el componente sobre el que se implementan dichos menús, ya que si se trata de un componente ligero (**JPanel**, por ejemplo) e intersecciona con la ventana principal de la interfaz de usuario implementada en base a un componente pesado (**Window**, por ejemplo), el componente pesado se presentará encima del ligero. Esto es cierto en general, aunque en el caso de los menús *popup* se puede evitar utilizando el método *setLightWeightPopupEnabled()* de la clase **JPopupMenu** para desactivar las ventanas ligeras.

ESCALAS Y BARRAS DE PROGRESO

Una escala permite al usuario introducir datos desplazando el marcador de la escala hacia un lado o hacia otro, lo cual resulta altamente intuitivo en algunos casos; como pueden ser, por ejemplo, controles de volumen. Una barra de progreso presenta al usuario información en forma de una barra parcialmente llena, o parcialmente vacía, para que pueda tener una perspectiva visual de los datos. El ejemplo `Java1417.java`, muestra los dos objetos en consonancia, de tal forma que desplazando la escala hacia un lado o hacia otro, la barra de progreso sigue el camino marcado por esos desplazamientos.



Figura 14.18

La figura 14.18 corresponde a la ejecución del programa, en donde se observa que la barra se ha desplazado al unísono que la escala, correspondiéndose en espacio.

```
public class Java1417 extends JPanel {
  JProgressBar barraProg = new JProgressBar();
  JSlider barraSlid = new JSlider( JSlider.HORIZONTAL,0,100,60 );

  public Java1417() {
    setLayout( new GridLayout(2,1) );
    add( barraProg );
    barraSlid.setValue( 0 );
    barraSlid.setPaintTicks( true );
    barraSlid.setMajorTickSpacing( 20 );
    barraSlid.setMinorTickSpacing( 5 );
    barraSlid.setBorder( new TitledBorder("Desplázame") );
    barraSlid.addChangeListener( new ChangeListener() {
      public void stateChanged( ChangeEvent evt ) {
        barraProg.setValue(barraSlid.getValue() );
        }
      } );
    add( barraSlid );
    }
```

El control de la barra de progreso es muy simple, aunque el objeto **JSlider** tiene una tremenda cantidad de opciones, que van desde la orientación hasta que las marcas sean más o menos grandes.

El ejemplo `Java1418.java` es autónomo y lo que hace es crear una tarea, cuando se pulsa el botón *Arrancar*, que va presentando en el campo de texto un número de 0 a 100, y va rellenando la barra de progreso, para indicar la carga, o el tanto por ciento de completitud que se lleva. La figura 14.19 muestra el ejemplo en pleno funcionamiento.



Figura 14.19

En el código solamente destacar la presencia de la clase anidada, que es la encargada de la actualización de la información, tanto de forma numérica en el campo de texto como de forma gráfica en la barra de progreso.

```
public class Java1418 extends JPanel {
  Thread hilo;
  Object objeto = new Object();
  boolean pideParar = false;
  JTextField texto;
  JProgressBar barra;

  public Java1418() {
    setLayout( new BorderLayout() );
    texto = new JTextField();
    add( texto,BorderLayout.NORTH );
    JPanel panelInferior = new JPanel();
    barra = new JProgressBar();
    panelInferior.setLayout( new GridLayout(0,1) );
    panelInferior.add( barra );
    panelInferior.add( new JLabel( "Cargando..." ) );
    JPanel panelBotones = new JPanel();
    JButton botonArranque = new JButton( "Arrancar" );
    botonArranque.setBackground( SystemColor.control );
    panelBotones.add( botonArranque );
    botonArranque.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent evt ) {
        iniciaCuenta();
        }
      } );

    JButton botonParar = new JButton( "Parar" );
    botonParar.setBackground( SystemColor.control );
    panelBotones.add( botonParar );
    botonParar.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent evt ) {
        detieneCuenta();
        }
      } );
    panelInferior.add( panelBotones );
    add( panelInferior,BorderLayout.SOUTH );
    }

  public void iniciaCuenta() {
    if( hilo == null ) {
      hilo = new ThreadCarga();
      pideParar = false;
      hilo.start();
      }
    }

  public void detieneCuenta() {
    synchronized( objeto ) {
      pideParar = true;
      objeto.notify();
      }
    }
```

```
class ThreadCarga extends Thread {
  public void run() {
    int min = 0;
    int max = 100;
    barra.setValue( min );
    barra.setMinimum( min );
    barra.setMaximum( max );
    for( int i=min; i <= max; i++ ) {
      barra.setValue( i );
      texto.setText( ""+i );
      synchronized( objeto ) {
        if( pideParar )
          break;
        try {
          objeto.wait( 100 );
        } catch( InterruptedException e ) {
          // Se ignoran las excepciones
          }
        }
      }
    hilo = null;
    }
  }
```

Un tipo de barra de progreso también disponible en Swing, es la barra de progreso *indeterminada*. Se trata de una barra animada que se mueve continuamente. No tiene ningún interés especial, pero sí puede resultar útil a la hora de querer indicar que se están realizando acciones en la aplicación, y que el usuario se sienta menos frustrado al ver que hay movimiento, por ejemplo a la hora de instalar una aplicación o de realizar la descarga de un archivo.

El método que permite indicar que la barra de progreso es de este tipo es *setIndeterminate()* con un valor `true`. En esta barra es posible indicar la velocidad de la animación. El ejemplo `Java1419.java` muestra el uso de esta barra. Si el lector lo ejecuta y pulsa sobre el botón *Arrancar*, se iniciará la animación de la barra, que se detiene al pulsar el botón *Parar*.

ÁRBOLES

Utilizar en Swing un árbol, como los que se despliegan en muchas de las ventanas de los sistemas operativos al uso, es tan simple como escribir:

```
add( new JTree( new Object[]{ "este","ese","aquel" } ) );
```

Esto crea un arbolito muy primitivo; sin embargo, el API de Swing para árboles es inmenso, quizá sea uno de los más grandes. En principio parece que se puede hacer cualquier cosa con árboles, pero lo cierto es que si se van a realizar tareas con un cierto grado de complejidad, es necesario un poco de investigación y experimentación antes de lograr los resultados deseados.

Afortunadamente, como en el término medio está la virtud, en Swing, *Sun Microsystems* proporciona árboles "*por defecto*", que son los que generalmente necesita el programador la mayoría de las veces, así que no hay demasiadas ocasiones en que haya que entrar en las profundidades de los árboles para que se deba tener un conocimiento exhaustivo del funcionamiento de estos objetos.

Los árboles en Swing se utilizan para representar estructuras de datos jerárquicas empleando para ello la representación típica de los exploradores de ficheros. Cada dato de la estructura jerárquica es un *nodo*. La figura 14.20 muestra la representación abstracta de nodos en la parte izquierda y, en la derecha, la estructura típica de árbol que el lector reconocerá con facilidad.

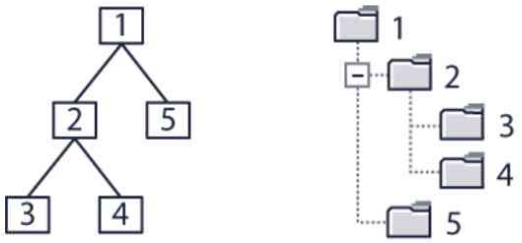


Figura 14.20

El nodo superior es el nodo *raíz* (en este caso 1), los nodos finales se conocen como *hojas* (en este caso 3, 4 y 5) y los nodos intermedios reciben el nombre de *ramas* (en este caso 2). También se puede hablar de nodos *padre* y entonces los nodos que cuelgan de él se conocen como nodos *hijo*.

La clase JTree

La clase **JTree** es la punta del iceberg que constituye la implementación de árboles en Swing. Se trata de un componente que puede ser incorporado a cualquier contenedor Swing, siendo lo más habitual añadirlo a un contenedor **JScrollPane** para proporcionar desplazamiento vertical al árbol cuando se expandan sus nodos. La clase **JTree** es la que tiene asignados los roles de Vista y Controlador del patrón MVC utilizado por Swing.

Un objeto **JTree** necesita de un objeto de tipo **TreeModel** a partir del cual poder obtener información. Este objeto **TreeModel** se puede crear explícita o implícitamente a partir de otras fuentes como vectores o tablas *hash*. **TreeModel** es pues una interfaz que proporciona los datos que se van a presentar a través de **JTree**, constituyendo la parte del Modelo del patrón MVC. Esta interfaz describe métodos para consultar aspectos de la topología del árbol, incluyendo el nodo raíz. La clase receptora de eventos **TreeModelListener** se utiliza cuando es importante conocer cambios en la estructura del **TreeModel**. Swing proporciona una implementación por defecto de la interfaz **TreeModel**, la clase **DefaultTreeModel**.

La clase **TreeNode** es quizás la más importante dentro de la generación de árboles Swing. Proporciona métodos para nombrar a los hijos, determinar cuándo se pueden añadir hijos a un nodo padre, determinar cuándo un nodo es una hoja y métodos para localizar el nodo padre de un nodo cualquiera. Resulta también importante la interfaz **MutableTreeNode** que extiende la interfaz **TreeNode**, incorporándole métodos para insertar y eliminar nodos hijos, cambiar el padre de un nodo y almacenar un objeto cualquiera en un nodo. Sin embargo, esta interfaz no proporciona ningún método para recuperar un objeto almacenado en un nodo, por lo que el desarrollador no está obligado a permitir que los objetos almacenados en un árbol puedan ser recuperados. Afortunadamente, la implementación por defecto de la interfaz, la clase **DefaultMutableTreeNode**, sí proporciona un método para esta acción.

La clase **DefaultMutableTreeNode** dispone de un constructor por defecto que crea un nodo, sin ningún objeto almacenado, al cual se pueden añadir hijos. También dispone de otros constructores que permiten incorporar objetos al nodo e indicar si ese nodo puede o no tener hijos.

En resumen, la clase **JTree** puede construirse de varias formas. La primera de ellas consiste en utilizar el constructor por defecto que creará un objeto **TreeModel** automáticamente. Otra forma es mediante un array de objetos de tipo **Object**, **Vector** o **Hash**, en cuyo caso el modelo se creará en base a los datos que proporcionen estos objetos. Y la tercera forma de construir una clase **JTree** es mediante un objeto **TreeModel** o **TreeNode** propios, siendo ésta la más flexible de las tres y la que se utiliza en el siguiente ejemplo.

En la aplicación `Java1420.java` se utilizan instancias de la clase **DefaultMutableTreeNode** para crear un modelo para **JTree** y establecer las relaciones jerárquicas entre los componentes del árbol. El código que se muestra reproduce la creación de estos objetos.

```
// Creamos los nodos padre
DefaultMutableTreeNode arbol =
  new DefaultMutableTreeNode( "Árbol" );
DefaultMutableTreeNode color =
  new DefaultMutableTreeNode( "Colores" );
DefaultMutableTreeNode sabor =
  new DefaultMutableTreeNode( "Sabores" );
DefaultMutableTreeNode medida =
  new DefaultMutableTreeNode( "Medidas" );
// Utilizamos el primer nodo como raiz y añadimos los demás
// como hijos
arbol.add( color );
arbol.add( sabor );
arbol.add( medida );
// A cada uno de los nodos padre le incorporamos algunos
// elementos finales
color.add( new DefaultMutableTreeNode( "Rojo" ) );
color.add( new DefaultMutableTreeNode( "Verde" ) );
color.add( new DefaultMutableTreeNode( "Azul" ) );
```

```
sabor.add( new DefaultMutableTreeNode( "Salado" ) );
sabor.add( new DefaultMutableTreeNode( "Dulce" ) );
sabor.add( new DefaultMutableTreeNode( "Amargo" ) );
sabor.add( new DefaultMutableTreeNode( "Agrio" ) );
medida.add( new DefaultMutableTreeNode( "Corta" ) );
medida.add( new DefaultMutableTreeNode( "Media" ) );
medida.add( new DefaultMutableTreeNode( "Larga" ) );
// Creamos el árbol en base el modelo
JTree miArbol = new JTree( arbol );
```

En la aplicación se crea el nodo raíz `arbol` al cual se añaden los hijos `color`, `sabor` y `medida`, a quienes se añaden también otros hijos para generar en la última línea de código el objeto **JTree**, que internamente crea un **TreeModel** en base al nodo raíz, incorporándole un **TreeModelListener** para ese **TreeModel**, que será el encargado de sincronizar la presentación con los cambios que se vayan produciendo en el **TreeModel** o en cualquiera de los **TreeNode** asociados, todo ello de forma transparente al programador. La figura 14.21 muestra el árbol creado en la ejecución del ejemplo.

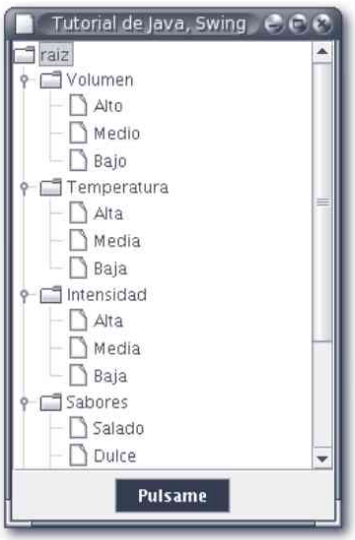


Figura 14.21

En la figura se observa cómo los nodos que contienen elementos colgando se presentan de forma diferente a los nodos finales. Ello se debe a que **JTree** utiliza el método *isLeaf()* para determinar el icono que debe mostrar y en la implementación por defecto de **DefaultMutableTreeNode** se devuelve `true` cuando un nodo no tiene ningún hijo. No obstante, aunque éste es el funcionamiento normal, puede que en ocasiones no sea el adecuado, por lo que será necesario extender la clase **DefaultMutableTreeNode** para que se adapte a circunstancias particulares, si fuera necesario. Por ejemplo, si se está utilizando un árbol para la presentación del contenido de una unidad de disco, la representación que se muestre estaría basada en *directorios* y *archivos*, en donde es más importante un directorio, que representa la capacidad de contener archivos, que la existencia de esos archivos. Es decir, que si se utiliza la implementación por defecto de la clase **DefaultMutableTreeNode**, un directorio vacío aparecerá representado mediante el icono correspondiente a un

archivo, ya que no contiene hijos; por ello, ésta es una de las ocasiones en las que será necesario proporcionar características nuevas a la clase para que presente correctamente los directorios vacíos.

Aplicación: Árbol personalizado

La aplicación `Java1421.java` es un poco más compleja que las demás presentadas en este capítulo, de forma que permita mostrar al lector la forma de personalizar el funcionamiento y apariencia de un objeto **JTree** en Swing. La aplicación construye un nuevo objeto **JVolumenTree** que es capaz de mostrar visualmente el tamaño de cada uno de los nodos del árbol, además de su nombre y estructura, que son los elementos habituales de **JTree**. Por ejemplo, si se utiliza esta nueva clase para presentar la estructura de archivos de una unidad de almacenamiento de datos, será posible observar el tamaño relativo de todos los archivos y directorios de esa unidad.

En la construcción de la aplicación se utiliza una implementación propia de la clase **TreeNode** en conjunción con la clase proporcionada por Java, **DefaultTreeModel**. La implementación de **TreeNode** permitirá almacenar la información referente al tamaño de cada uno de los nodos, ya que la obtención de este dato es una operación costosa y es recomendable realizarla una sola vez, almacenado luego el resultado. Además, a la hora de visualizar el contenido del árbol, será necesario presentar la información referente al tamaño de cada nodo, por lo que es imprescindible disponer de métodos para acceder a esa información. Sin embargo, el funcionamiento de despliegue y presentación del conjunto del árbol reaccionará del modo habitual, por lo que es suficiente el uso de la clase **DefaultMutableTreeNode** para manipular los objetos **TreeNode** personalizados que se hayan creado en la ejecución de la aplicación.

Ni la interfaz **TreeNode**, ni la interfaz **MutableTreeNode**, que extiende a la anterior, definen el tamaño del nodo. Por ello, la primera acción consiste en crear una nueva interfaz que extienda a **MutableTreeNode** y proporcione los métodos necesarios para poder recuperar esa información a la hora de la presentación del nodo en pantalla. El código correspondiente a la interfaz **VolumenNodo** se presenta a continuación.

```
public interface VolumenNodo extends MutableTreeNode {
  // Devuelve el tamaño del nodo excluyendo a sus hijos
  public long getTamano();
  // Devuelve el tamaño total del nodo y sus descendientes
  public long getTamanoTotal();
  // Devuelve el número total de descendientes del nodo
  public int getLeafCount();
  // Devuelve el nodo raíz de la jerarquía en que se encuentra
  // englobado el nodo
  public TreeNode getRoot();
  // Devuelve true si el tamaño total ya ha sido calculado, o false
  // en caso contrario
```

```
public boolean estaCalculado();
}
```

El lector puede observar que no se hacen indicaciones de la forma en que se va a obtener el tamaño a través del método *getTamano()*. Por ejemplo, si el árbol se utiliza para visualizar un sistema de ficheros, el tamaño de cada fichero se podrá obtener mediante el método *File.length()*, mientras que si la jerarquía representa cualquier otro tipo de objetos, será necesario proporcionar otro mecanismo de obtención del tamaño del nodo.

Tampoco se hacen indicaciones de la forma en que el tamaño de los nodos va a ser almacenado. Esta tarea puede ser muy pesada y nada trivial, por lo que se proporciona el método *estaCalculado()* para indicar cuándo un nodo ya conoce su propio tamaño.

El siguiente paso en la construcción del árbol de esta aplicación consiste en proporcionar una implementación por defecto para la interfaz **VolumenNodo**, de forma que cada uno de los métodos de la interfaz disponga de su propia implementación y sea el programador quien decida qué métodos debe sobrescribir. En este caso, la clase **DefaultMutableTreeNode** ya proporciona la implementación por defecto para algunos de los métodos, como *getRoot()* y *getLeafCount()*; no obstante, será necesario sobrescribir este último para adecuarlo a su cometido en la aplicación.

Por lo tanto, la clase **DefaultVolumenNodo** será la que implemente la interfaz **VolumenNodo** y extienda la clase **DefaultMutableTreeNode**, proporcionando miembros para almacenar el tamaño del nodo, el tamaño del nodo con sus descendientes y el número de descendientes del nodo. También proporciona un indicador, inicializado a `false`, que permitirá conocer si los tamaños de los nodos hijos han sido calculados. El código que se muestra a continuación reproduce la declaración de los miembros de la clase y el constructor.

```
public class DefaultVolumenNodo extends DefaultMutableTreeNode
  implements VolumenNodo {
  // La instancia del helper para este nodo
  private final VolumenNodoHelper helper;
  // Tamaño del nodo excluyendo los hijos
  private final long tamano;
  // Tamaño total del nodo más el de todos sus descendientes
  private long totalTamano;
  // Almacena el número total de hijos que son descendientes
  // de este nodo. Debe mantenerse en una caché, para
  // proporcionar mejor rendimiento a la implementación del
  // método getLeafCount() de la implementación por
  // defecto DefaultMutableTreeNode
  private int totalNodos;
  // Indica si se ha concluido el cálculo de todos los hijos
  // del nodo
  private boolean calculado;
  // Indica si el árbol de descendientes del nodo se
  // ha construido
  private boolean explorado = false;
```

```
  // Construye un nodo a partir del objeto que se proporciona como
  // parámetro y la clase de ayuda que también debe indicarse
  public DefaultVolumenNodo( final Object obj,
    final VolumenNodoHelper _helper ) {
    // Guarda los parámetros de la invocación
    super( obj );
    helper = _helper;
    // Obtiene el tamaño del objeto a través de la clase auxiliar
    tamano = helper.getTamano( obj );
    // Inicializa el tamaño total del nodo al tamaño del nodo,
    // que posteriormente será aumentado en el tamaño de los
    // nodos hijos
    totalTamano = tamano;
    // Inicializa el número de nodos a 0, o a 1 si estamos ante
    // una hoja, para que se tenga en cuenta a la hora de calcular
    // el tamaño completo del nodo padre
    totalNodos = helper.esContenedor(obj) ? 0 : 1;
    // Si el nodo no permite tener hijos, se da por concluido el
    // cálculo del tamaño de ese nodo
    calculado = !getAllowsChildren();
    }
  }
```

Para que la representación del árbol pueda llevarse a cabo, será necesario proporcionar los datos que necesita esta clase correspondientes al texto que se presentará en el árbol como identificador de cada uno de los nodos, a la indicación de si el nodo es un contenedor o no, para poder determinar su icono, y finalmente, el tamaño del nodo. Esa información se puede proporcionar de tres formas distintas:

- La clase **DefaultVolumenNodo** puede disponer de métodos abstractos para devolver la información, de forma que se puedan especializar esos métodos para cada tipo de nodo que se vaya a representar.
- La clase **DefaultVolumenNodo** puede recibir esa información a través del constructor o mediante métodos *set()* que permitan fijar esos valores.
- La provisión de esa información se puede dejar en manos de una clase auxiliar de ayuda, en un objeto de tipo **Helper**.

En este ejemplo se utiliza la tercera opción, porque es la más ilustrativa a efectos de aprendizaje de las características de Java. Esta decisión hace necesaria la definición de una nueva interfaz, **VolumenNodoHelper**, para proporcionar la información necesaria citada anteriormente. La clase **DefaultMutableTreeNode** proporciona la noción de *objeto de usuario* asociado a cada nodo, que se va a utilizar en este caso como clave para solicitar del objeto **Helper** la información acerca del nodo. La interfaz define varios métodos, tal como se muestra en el siguiente código.

```
public interface VolumenNodoHelper {
  // Devuelve el conjunto de objetos de los nodos hijos del nodo que
  // se pasa como parámetro
  public Object[] getHijos( Object obj );
  // Indica si el nodo representado por el objeto que se pasa como
  // parámetro es un contenedor
```

```
public boolean esContenedor( Object obj );
// Devuelve el tamaño del nodo representado por el objeto que se
// pasa como parámetro, excluyendo a cualquiera de los hijos
public long getTamano( Object obj );
// Devuelve la representación del nodo en forma de cadena
public String toString( Object obj );
}
```

El objeto **Helper** se proporciona a la clase en el constructor y ésta lo almacena para su uso posterior. Se puede utilizar el mismo objeto en el constructor de todos los nodos de la jerarquía del árbol. Si el lector observa el constructor de la clase **DefaultVolumenNodo**, advertirá que el miembro que representa el tamaño del nodo se inicializa con el tamaño del propio nodo, al que se irá sumando el tamaño de todos los nodos hijos de que disponga en la jerarquía del árbol.

Ya se ha indicado antes que la obtención del tamaño de un nodo puede ser una tarea complicada y sobre todo, que consuma mucho tiempo, por lo que debería realizarse en una única ocasión. Sin embargo, es importante que el cálculo se realice una vez conocido el tamaño de todos los nodos hijos. Un nodo no puede saber si el cálculo del tamaño de sus hijos está concluido, por lo que es necesario proporcionar un método para indicar a ese nodo que ya puede iniciar el cálculo de su propio tamaño, una vez conocido el de sus hijos.

El método *calcular()* se añade a cada nodo como encargado de ejecutar la tarea anterior. Primero invocará a los métodos *calcular()* de todos los hijos, para luego realizar el cálculo de su propio tamaño. Por lo tanto, lo único necesario es invocar este método sobre el nodo raíz, una vez que la jerarquía de nodos del árbol esté completa, y será el propio método *calcular()* de ese nodo raíz el que recorra toda la jerarquía, calculando el tamaño de cada uno de los nodos, realizando los cálculos necesarios y sin ningún tipo de redundancia. La implementación del método es la que se muestra a continuación y observará el lector que es más sencilla que la explicación anterior.

```
public void calcular( final DefaultTreeModel modelo ) {
  // Nos aseguramos de que hemos construido el árbol completo
  // del nodo
  explorar( modelo );
  // Para cada uno de los nodos hijos, hacemos lo siguiente...
  final Enumeration hijos = hijos();
  while( hijos.hasMoreElements() ) {
    final DefaultVolumenNodo node =
      (DefaultVolumenNodo)hijos.nextElement();
    // Si el hijo es un contenedor, seguimos recursivamente
    // a través de sus hijos y almacenamos los resultados
    if( node.getAllowsChildren() ) {
      node.calcular( modelo );
      totalTamano += node.getTamanoTotal();
      totalNodos += node.getLeafCount();
      }
    // Si el hijo es una hoja, acumulamos los valores
    else {
      totalTamano += node.getTamano();
      totalNodos++;
```

```
      }
    }
  // Indicamos que se ha acabado el cálculo del tamaño total
  calculado = true;
  // La llamada se engloba en un try-catch para evitar la
  // excepción de Swing de finalización de la exploración
  try {
    // Indicamos al modelo que hemos cambiado el nodo
    modelo.nodeChanged( this );
  } catch( NullPointerException e ) {
    e.printStackTrace();
    }
  }
```

Cuando el cálculo del tamaño de un nodo está completo, se actualiza el valor de la variable miembro que almacena esa información y se asigna el valor `true` al indicador de que el tamaño de ese nodo es conocido, de modo que el método *estaCalculado()* devolverá `true` a partir de ese instante.

Una mejora más que todavía se puede incorporar es automatizar la construcción de la jerarquía de nodos del árbol, de forma que si ya existe previamente, solamente se recorra, sin necesidad de volver a crear nodo alguno ni realizar cálculos. Siguiendo el ejemplo del sistema de ficheros, se puede enumerar la jerarquía de cualquier directorio tomando ese directorio como nodo raíz y recorriendo todos sus subdirectorios y ficheros que pueda contener, obteniendo de este modo todos sus hijos. Entonces, se puede incorporar un nuevo método a la clase **Helper**, *getHijos()*, al cual se pasará como parámetro el nodo a partir del cual se quiere obtener la jerarquía del árbol.

Para recorrer la jerarquía de nodos sin realizar cálculos se añade el método *explorar()*, de forma que se creen automáticamente los nodos hijos. Es importante asegurar que *explorar()* solamente se invoque una vez, de forma que cada uno de los nodos se añada en una única ocasión a la jerarquía. Para ello se incorpora una variable booleana como miembro que indicará si el método *explorar()* sobre un nodo determinado ya ha sido invocado. La implementación de este método es la siguiente.

```
public synchronized void explorar( final DefaultTreeModel modelo ) {
  // Controlamos que no se haya explorado ya
  if( !explorado ) {
    // Recuperamos el conjunto de hijos, si los hay
    final Object[] hijos = helper.getHijos( getUserObject() );
    for( int hijo=0; hijo < hijos.length; hijo++ ) {
      // Para cada uno de ellos se construye un nuevo nodo,
      // utilizando la misma clase de ayuda y añadiéndolo
      add( new DefaultVolumenNodo(hijos[hijo],helper) );
     }
    // Indicamos que esta rama ya está explorada
    explorado = true;
    // La llamada se engloba en un try-catch para evitar la
    // excepción de Swing de finalización de la exploración
    try {
      // Notificamos al modelo que la estructura ha cambiado
      modelo.nodeStructureChanged( this );
    } catch( NullPointerException e ) {
```

```
            e.printStackTrace();
            }
        }
    }
```

Invocando a este método al inicio del método *calcular()* se asegura que todos los hijos son creados con su tamaño adecuado. Además, la expansión de cada nodo en sus nodos hijos se realizará cuando se solicite, y no antes, de forma que no se realice ningún trabajo en vano, es decir, si un nodo nunca va a ser necesario, nunca será creado; sin embargo, la naturaleza recursiva del método *calcular()* hace que una sola llamada a este método sobre el nodo raíz del árbol, cree la jerarquía de nodos completa y el tamaño de cada uno de ellos sea calculado.

Este método se encuentra protegido contra llamadas repetidas a través de la variable `explorado`, que indica si la jerarquía del nodo ya ha sido revisada o no, de forma que esa jerarquía solamente se construya la primera vez que se llama al nodo. Luego es posible aprovechar esa circunstancia para que en el caso de que el usuario decida abrir una rama no creada se invoque a este método sobre ese nodo en ese mismo instante y, sino, ya será invocado por *calcular()* posteriormente, pero el resultado será el mismo.

LA CLASE VOLUMENNODORENDERER

La interfaz **TreeCellRenderer** define el método *getTreeCellRendererComponent()*, que devuelve un objeto **Component** que puede ser personalizado de acuerdo a los parámetros del método. Éste es el método que utiliza la clase **JTree** a la hora de visualizar un nodo.

Los parámetros del método describen el estado del nodo que va a ser presentado, como por ejemplo si está seleccionado, si está expandido, si es un nodo final o si tiene el foco. Son muchos parámetros e imprescindibles, porque hay información que solamente posee el objeto **JTree**, no la clase **TreeModel** o cualquiera de las instancias de **TreeNode**. Por ejemplo, el modelo no sabe nada acerca de si un nodo está expandido o no, porque según el patrón MVC puede haber dos o más vistas activadas sobre un mismo árbol, presentando cada una de ellas un estado diferente.

Para atender a todas estas diferencias, que resultan muy significativas a la hora de visualizar un árbol, Swing proporciona la clase **DefaultTreeCellRenderer**, que implementa la interfaz **TreeCellRenderer**. Esta clase, además de implementar la interfaz, también extiende la clase **JLabel** de modo que puede presentar un icono con un texto adyacente.

En la aplicación que atañe a este capítulo es necesario presentar también información gráfica acerca del tamaño del nodo. Como no se pretende ser preciso, sino solamente mostrar al lector las técnicas de uso de **JTree** y, como una imagen es mucho más representativa que un texto, se va a incorporar a la identificación de cada nodo una imagen con un diagrama de tarta que represente el tamaño relativo del nodo.

```
            e.printStackTrace();
            }
        }
    }
```

Invocando a este método al inicio del método *calcular()* se asegura que todos los hijos son creados con su tamaño adecuado. Además, la expansión de cada nodo en sus nodos hijos se realizará cuando se solicite, y no antes, de forma que no se realice ningún trabajo en vano, es decir, si un nodo nunca va a ser necesario, nunca será creado; sin embargo, la naturaleza recursiva del método *calcular()* hace que una sola llamada a este método sobre el nodo raíz del árbol, cree la jerarquía de nodos completa y el tamaño de cada uno de ellos sea calculado.

Este método se encuentra protegido contra llamadas repetidas a través de la variable `explorado`, que indica si la jerarquía del nodo ya ha sido revisada o no, de forma que esa jerarquía solamente se construya la primera vez que se llama al nodo. Luego es posible aprovechar esa circunstancia para que en el caso de que el usuario decida abrir una rama no creada se invoque a este método sobre ese nodo en ese mismo instante y, sino, ya será invocado por *calcular()* posteriormente, pero el resultado será el mismo.

LA CLASE VOLUMENNODORENDERER

La interfaz **TreeCellRenderer** define el método *getTreeCellRendererComponent()*, que devuelve un objeto **Component** que puede ser personalizado de acuerdo a los parámetros del método. Éste es el método que utiliza la clase **JTree** a la hora de visualizar un nodo.

Los parámetros del método describen el estado del nodo que va a ser presentado, como por ejemplo si está seleccionado, si está expandido, si es un nodo final o si tiene el foco. Son muchos parámetros e imprescindibles, porque hay información que solamente posee el objeto **JTree**, no la clase **TreeModel** o cualquiera de las instancias de **TreeNode**. Por ejemplo, el modelo no sabe nada acerca de si un nodo está expandido o no, porque según el patrón MVC puede haber dos o más vistas activadas sobre un mismo árbol, presentando cada una de ellas un estado diferente.

Para atender a todas estas diferencias, que resultan muy significativas a la hora de visualizar un árbol, Swing proporciona la clase **DefaultTreeCellRenderer**, que implementa la interfaz **TreeCellRenderer**. Esta clase, además de implementar la interfaz, también extiende la clase **JLabel** de modo que puede presentar un icono con un texto adyacente.

En la aplicación que atañe a este capítulo es necesario presentar también información gráfica acerca del tamaño del nodo. Como no se pretende ser preciso, sino solamente mostrar al lector las técnicas de uso de **JTree** y, como una imagen es mucho más representativa que un texto, se va a incorporar a la identificación de cada nodo una imagen con un diagrama de tarta que represente el tamaño relativo del nodo.

Es decir, además del icono y texto estándares de cada nodo, se añadirá una imagen más, y la clase **VolumenNodoRenderer** será la encargada de esa misión.

El método para añadir esa imagen consiste en aumentar en el **JLabel** de identificación del nodo, la separación entre el icono y el texto, colocando en medio la imagen que indica el tamaño del nodo. A la clase **VolumenNodoRenderer** se pasará un array de imágenes representando el tanto por ciento del tamaño relativo de cada nodo respecto a su padre; así que dependerá del número de imágenes proporcionadas el que cada gráfico sea más o menos preciso. Un detalle importante es la distinción entre un nodo vacío y otro que está casi vacío, pero que tiene un tamaño, aunque sea mínimo; para ello, en el constructor de la clase **VolumenNodoRenderer**, que se reproduce a continuación, se ha añadido un parámetro para indicar la imagen que se presentará cuando se produzca esta circunstancia.

```
public VolumenNodoRenderer( final Image[] _imagenes,
  final Image _imgVacio,final Image _imgCalculando ) {
  // Almacenamos los parámetros en las variables miembro de la
  // clase
  imagenes = _imagenes;
  imgVacio = _imgVacio;
  imgCalculando = _imgCalculando;
  umbralVacio = (1.0 / (_imagenes.length - 1)) / 2.0;
  // Fijamos el modo de presentación inicial
  setModoPresentacion( JVolumenTree.TAM_RELATIVO_RAIZ );
  // Preparamos el controlador de las imágenes, indicándole su
  // tamaño para que controle su carga
  imgTrack = new CargaImagenes( imagenes,imgVacio,imgCalculando );
  }
```

Cuando se invoque al método *getTreeCellRendererComponent()*, se seleccionará la imagen adecuada al tamaño del nodo, asumiendo que ese tamaño ya está calculado. El método *selImagen()* será el encargado de elegir la imagen correcta entre el array de imágenes que se haya proporcionado al constructor de la clase.

```
private Image selImagen( final double proporcion,
  final VolumenNodo nodo,final VolumenNodo nodoCompara ) {
  Image imagen;
  // Si todavía no tenemos imagen seleccionada y el nodo o el que
  // sirve como referencia no ha sido añadido todavía, mostramos
  // la imagen de que estamos calculando
  if( (imgCalculando != null) && ( !nodo.estaCalculado() ||
    ( (nodoCompara != null) && !nodoCompara.estaCalculado()) ) )
    imagen = imgCalculando;
  // Si el tamaño es mayor que cero, pero está por debajo del
  // umbral del tamaño que sirve de límite para indicar que un
  // nodo está casi vacío, peresentamos esa imagen
  else if( (imgVacio != null) && (proporcion > 0.0) &&
    (proporcion < umbralVacio) )
    imagen = imgVacio;
  // En cualquier otro caso, presentamos la imagen que corresponda
  // a la proporción del tamaño del nodo
  else {
    final int index = (int)Math.round(
```

```
        (imagenes.length-1) * Math.min(1.0,proporcion) );
      imagen = imagenes[index];
      }
    return( imagen );
    }
```

SISTEMA DE FICHEROS

En las secciones anteriores se han descrito las interfaces y clases que permiten crear un **JTree** personalizado que ahora se utilizará para crear la clase **JVolumenTree**, que mostrará la jerarquía de un sistema de ficheros indicando el tamaño de cada uno de los directorios y archivos que lo componen.

Para recuperar información acerca de los directorios y archivos de un sistema de ficheros, Java proporciona la clase **File** en el paquete **java.io**, cuyos métodos pueden recuperar información acerca del nombre del archivo o directorio, *getName()*; distinción de si se trata de archivo o directorio, *isDirectory()*; tamaño del archivo o directorio, *length()*; o la lista de archivos que contiene un directorio, *listFiles()*.

En el caso del sistema de ficheros, es necesario proporcionar una implementación adecuada de la interfaz **VolumenNodoHelper** que aunque simple, porque todo el trabajo reside en cada nodo, sí debe obtener la información de la forma adecuada. La implementación a través de la clase **FileVolumenNodoHelper** se reproduce a continuación.

```
public class FileVolumenNodoHelper implements VolumenNodoHelper {
  // Devuelve el nombre del fichero que se pasa, o el directorio si
  // el nombre está en blanco
  public String toString( final Object object ) {
    final File file = (File)object;
    final String name = file.getName();
    return( (name.length() > 0) ? name : file.getPath() );
    }

  // Devuelve el array de ficheros que cuelgan del que se pasa
  public Object[] getHijos( final Object object ) {
    File[] files;
    if( (files = ((File)object).listFiles()) == null )
      files = new File[0];
    return( files );
    }

  // Indica si el fichero que se pasa es un contenedor, directorio
  // en este caso
  public boolean esContenedor( final Object object ) {
    return( ((File)object).isDirectory() );
    }

  // Devuelve el tamaño del fichero que se pasa como parámetro
  public long getTamano( final Object object ) {
    return( ((File)object).length() );
    }
  }
```

Para hacer más sencillo el uso de todas las características propias que se han descrito en las secciones anteriores, se introduce ahora la clase **JVolumenTree**, que extiende a la clase **JTree** de Swing.

La característica principal de esta nueva clase es su constructor, en donde el nodo raíz es creado a través del objeto **Helper**, se aplica el visualizador y se invoca el método *calcular()* sobre el nodo raíz, que será el encargado de crear la jerarquía completa del árbol y calcular el tamaño de cada uno de los nodos.

```
public JVolumenTree( final Object raiz,final VolumenNodoHelper helper,
  final Image[] imagenes,final Image imgVacia,
  final Image imgCalculando ) {
  // Crea el nodo raíz, crea el modelo del árbol con ese nodo y
  // construye el objeto Swing JTree con ese modelo
  super(new DefaultTreeModel(new DefaultVolumenNodo(raiz,helper)) );
  // Utilizamos el visualizador con las imágenes para mostrar el
  // tamaño con respecto al padre
  renderCelda = new VolumenNodoRenderer( imagenes,
    imgVacia,imgCalculando );
  renderCelda.setModoPresentacion( TAM_RELATIVO_PADRE );
  setCellRenderer( renderCelda );
  // Nos aseguramos de que el árbol permita Tooltips, que es donde
  // presentaremos la información textual del tamaño de los nodos
  ToolTipManager.sharedInstance().registerComponent( this );
  // Incorporamos un receptor de eventos para controlar la expansión
  // del árbol
  addTreeExpansionListener( new TreeExpansionListener() {
    public void treeCollapsed( TreeExpansionEvent evt ) {}
    public void treeExpanded( TreeExpansionEvent evt ) {
    // Comprobamos que el nodo que está siendo expandido ya ha
    // sido explorado. Para ello lo localizamos y nos aseguramos
    // de que se ha explorado
      final DefaultVolumenNodo nodo =
        (DefaultVolumenNodo)evt.getPath().getLastPathComponent();
      nodo.explorar( (DefaultTreeModel)getModel() );
      }
    } );
  // Iniciamos la tarea en segundo plano que calcula los tamaños
  inicioCalculo();
  }
```

La clase **Java1421** es una muestra del uso que se puede dar a la clase definida anteriormente. Su código es muy simple y se basa en la clase **JVolumenFrame** que crea la ventana y el contenedor sobre el que se presenta el árbol jerárquico del sistema de ficheros que se indique. La figura 14.22 muestra el resultado aplicado a uno de los directorios de instalación del J2SE 6 en Sun Java Desktop.

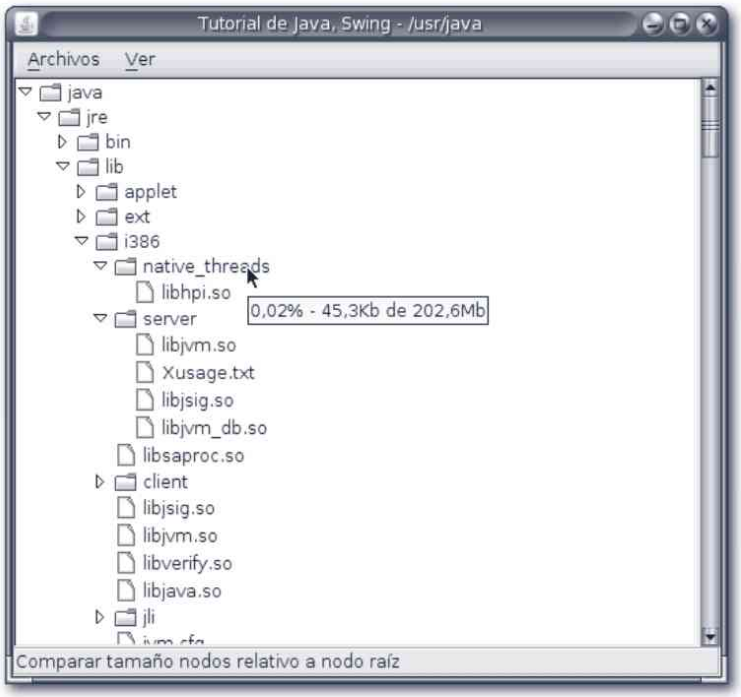


Figura 14.22

La última línea del constructor de la clase invoca al método *inicioCalculo()* que lanza una tarea en segundo plano para realizar los cálculos de tamaño de directorios y ficheros, de forma que el usuario siga teniendo el control de la aplicación. El código del método es el que se reproduce en las siguientes líneas.

```
private void inicioCalculo() {
  // Ejecutamos la acción como una tarea en segundo plano
  final Thread t = new Thread() {
    public void run() {
      // Localizamos el nodo raíz e invocamos al método calcular()
      // sobre él
      final DefaultTreeModel modelo =
        (DefaultTreeModel)getModel();
      final DefaultVolumenNodo raiz =
        (DefaultVolumenNodo)modelo.getRoot();
      raiz.calcular( modelo );
      // Nos aseguramos de que el árbol actualiza la información
      repaint();
      }
    };
  // Fijamos la prioridad mínima a la tarea
  t.setPriority( Thread.MIN_PRIORITY );
  // Y la convertimos en demonio
  t.setDaemon( true );
  t.start();
  }
```

Las tres últimas líneas de código son especialmente interesantes. En la primera se asigna a la tarea una prioridad muy baja, de forma que aunque Java no ofrece garantía alguna acerca del *scheduling* de tareas, esto asegurará que cualquier otra tarea con una prioridad superior tendrá preferencia a la tarea del ejemplo. Esta circunstancia es muy

importante, ya que así se evita la interferencia con tareas que necesitan una rápida ejecución, como por ejemplo la respuesta a la interfaz de usuario. La segunda sentencia convierte a la tarea en un *demonio*, es decir, que si la máquina virtual Java debe cerrarse, lo hará aunque esa tarea esté en ejecución, ya que lo contrario no tendría sentido. La máquina virtual Java concluye su ejecución cuando todas las tareas *no-demonio* hayan concluido, o cuando se invoque al método *System.exit()*, lo que ocurra en primer lugar. La tercera línea es la encargada de arrancar la ejecución de la tarea.

En el funcionamiento del ejemplo hay que tener en cuenta que el árbol puede aparecer antes que se haya concluido la jerarquía completa, con lo que el usuario debe esperar para poder interactuar con el árbol. Es muy importante, entonces, asegurar que las vistas del árbol se actualizan conforme se vayan añadiendo nodos a la jerarquía durante las llamadas al método *calcular()*.

En el patrón MVC se especifica que el Modelo debe notificar a cualquier Vista registrada todos los cambios que se produzcan. En Swing, esto se consigue añadiendo receptores de eventos al Modelo. El objeto **JTree** asegurará que se ha añadido un receptor **TreeModelListener** al modelo del árbol, de forma que se realice un redibujado del árbol cada vez que se reciba una notificación de cambio.

Para que el árbol vaya cambiando a través de la jerarquía de nodos, el modelo debe lanzar una notificación de tipo *nodeStructureChanged()*. Como la clase **VolumenNodo** no conoce el modelo asociado al árbol, es necesario pasarlo como parámetro al método *explorar()*, que es llamado directamente por el método *calcular()*, luego este último también debe recibir el modelo como parámetro.

Cuando la ejecución de *calcular()* llegue a su fin, ya se habrán realizado todas las acciones necesarias para conocer el tamaño de los nodos de la jerarquía en ese punto, por lo que se puede enviar la notificación de actualización del árbol invocando al método *nodeChanged()* de la clase **DefaultTreeModel**.

En el constructor se utiliza el método *addTreeExpansionListener()* para detectar cuándo el usuario desea expandir una rama del árbol. La incorporación de este receptor de eventos implica que el usuario pueda intentar expandir una rama en el mismo instante en que *calcular()* accede a esa misma rama, en cuyo caso es posible que las dos tareas se ejecuten sobre ese mismo nodo y se obtenga el doble de hijos de los reales. Colocando la asignación del valor `true` a la variable de control del método *explorar()* al principio del método se disminuye el riesgo de que se produzca el conflicto, aunque no se elimine completamente.

En esta aplicación se ha intentado mostrar al lector el manejo de árboles en Swing, presentando algunas de las técnicas que permiten su adecuación a fines concretos y también, el modo de evitar algunos de los efectos colaterales que siempre lleva aparejado el uso de componentes de este tipo y que el lector debe tener siempre presentes a la hora de realizar sus propias aplicaciones.

TABLAS

Al igual que los árboles, las tablas en Swing son importantes y poderosas. En principio, se crearon para constituir una interfaz ligada a bases de datos a través del *Java Database Connectivity* (JDBC), y así evitar la complejidad inherente al manejo de los datos, proporcionando mucha flexibilidad al programador. Hay suficientes características como para montar desde una hoja de cálculo básica hasta información para escribir un libro completo. Sin embargo, también es posible crear una **JTable** relativamente simple si entiende correctamente el funcionamiento.

La **JTable** controla cómo se presentan los datos, siendo el **TableModel** quien controla los datos en sí mismos. Para crear una **JTable** habrá que crear pues, previamente, un **TableModel**. Se puede implementar, para ello, la interfaz **TableModel**, pero es mucho más simple heredar de la clase ayuda **AbstractTableModel**. El ejemplo `Java1422.java` muestra esta circunstancia.

```
// El Modelo de la Tabla es el que controla todos los datos que se
// colocan en ella
class ModeloDatos extends AbstractTableModel {
  Object datos[][] = {
    {"uno","dos","tres","cuatro"},
    {"cinco","seis","siete","ocho"},
    {"nueve","diez","once","doce"},
    };

  // Esta clase imprime los datos en la consola cada vez que se
  // produce un cambio en cualquiera de las casillas de la tabla
  class TablaListener implements TableModelListener {
    public void tableChanged( TableModelEvent evt ) {
      for ( int i=0; i < datos.length; i++ ) {
        for ( int j=0; j < datos[0].length; j++ )
          System.out.print( datos[i][j] + " " );
        System.out.println();
        }
      }
    }

  // Constructor
  ModeloDatos() {
    addTableModelListener( new TablaListener() );
    }
  // Devuelve el número de columnas de la tabla
  public int getColumnCount() {
    return( datos[0].length );
    }
  // Devuelve el número de filas de la tabla
  public int getRowCount() {
    return( datos.length );
    }
  // Devuelve el valor de una determinada casilla de la tabla
  // identificada mediante fila y columna
  public Object getValueAt( int fila,int col ) {
    return( datos[fila][col] );
    }
```

```
  // Cambia el valor que contiene una determinada casilla de la tabla
  public void setValueAt( Object valor,int fila,int col ) {
    datos[fila][col] = valor;
    // Indica que se ha cambiado
    fireTableDataChanged();
    }
  // Indica si la casilla identificada por fila y columna es editable
  public boolean isCellEditable( int fila,int col ) {
    return( true );
    }
  }

public class Java1422 extends JPanel {
  public Java1422() {
    setLayout( new BorderLayout() );
    JTable tabla = new JTable( new ModeloDatos() );
    // La tabla se añade a un ScrollPane para que sea éste el que
    // controle automáticamente el tamaño de la tabla, presentando una
    // barra de desplazamiento cuando sea necesario
    JScrollPane panel = new JScrollPane( tabla );
    add( panel,BorderLayout.CENTER );
    }
```

En el ejemplo, para no complicar la cosa, **DataModel** es quien contiene el conjunto de datos, pero también se podrían haber obtenido a partir de una base de datos. El constructor añade un receptor de eventos de tipo **TableModelListener** que va a imprimir el contenido de la tabla cada vez que se produzca un cambio. El resto de los métodos utilizan la notación de los *Beans* y son utilizados por el objeto **JTable** cuando se quiere presentar la información del **DataModel**. **AbstractTableModel** proporciona por defecto los métodos *setValueAt()* y *isCellEditable()* que se utilizan para cambiar los datos, por lo que si se quiere realizar esta operación, no queda más remedio que sobrescribir los dos métodos.

La figura 14.23 muestra la tabla obtenida tras la ejecución del programa en donde se está editando la celda correspondiente a la segunda fila y columna A.

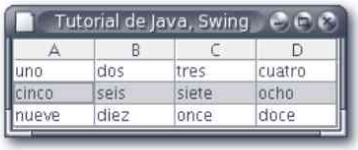


Figura 14.23

Una vez que se tiene un **TableModel**, sólo resta colocarlo en el constructor de **JTable**. Todos los detalles de presentación, edición y actualización están ocultos al programador. En este ejemplo, se coloca la **JTable** en un **JScrollPane**, por lo que es necesario un método especial en **JTable**.

Las tablas pueden soportar un comportamiento más complejo. En el ejemplo `Java1423.java` se utiliza un método para cargar un array de ocho columnas por un centenar de filas y, posteriormente, la tabla es configurada para que muestre solamente

las líneas verticales y permita la que se marquen todas las celdas de la fila en que se encuentre la celda seleccionada.

```
class Java1423 extends JPanel {
  private JTable tabla;
  private JScrollPane panelScroll;
  private String titColumna[];
  private String datoColumna[][];

  public Java1423() {
    setLayout( new BorderLayout() );
    // Creamos las columnas y las cargamos con los datos que van a
    // aparecer en la pantalla
    CreaColumnas();
    CargaDatos();
    // Creamos una instancia del componente Swing
    tabla = new JTable( datoColumna,titColumna );
    // Aquí se configuran algunos de los parámetros que permiten
    // variar la JTable
    tabla.setShowHorizontalLines( false );
    tabla.setRowSelectionAllowed( true );
    tabla.setColumnSelectionAllowed( false );
    // Cambiamos el color de la zona seleccionada (rojo/blanco)
    tabla.setSelectionForeground( Color.white );
    tabla.setSelectionBackground( Color.red );
    // Incorporamos la tabla a un panel que incorpora ya una barra
    // de desplazamiento, para que la visibilidad de la tabla sea
    // automática
    panelScroll = new JScrollPane( tabla );
    add( panelScroll, BorderLayout.CENTER );
    }

  // Creamos las etiquetas que sirven de título a cada una de
  // las columnas de la tabla
  public void CreaColumnas() {
    titColumna = new String[8];
    for ( int i=0; i < 8; i++ ) {
      titColumna[i] = "Col: "+i;
      }
    }

  // Creamos los datos para cada uno de los elementos de la tabla
  public void CargaDatos() {
    datoColumna = new String[100][8];
    for ( int iY=0; iY < 100; iY++ ) {
      for ( int iX=0; iX < 8; iX++ ) {
        datoColumna[iY][iX] = "" + iX + "," + iY;
        }
      }
    }
```

La figura 14.24 muestra el resultado de la ejecución del ejemplo. En este ejemplo los colores de fondo y texto para la región seleccionada se han alterado, que es otra de las facilidades que soporta el componente **JTable**, el cambio de color para un área de selección, en este caso para la selección de una fila.

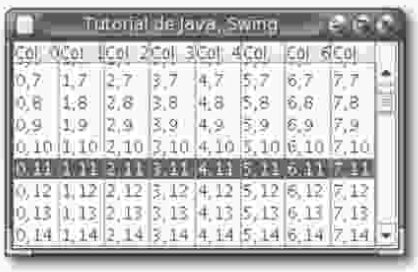


Figura 14.24

Aunque el ejemplo contiene un array relativamente grande de datos, la clase **JTable** no manipula demasiado bien grandes cantidades de información, resultando un rendimiento bastante pobre cuando se sobrepasan los 2.000 elementos.

Otra cuestión importante a la hora de tratar las tablas en Swing es la impresión de su contenido. La clase **JTable** dispone de métodos para acceder a la impresión, sin necesidad de que haya que recurrir al conocimiento de las APIs de impresión. El método *print()* imprime de forma directa y sencilla un objeto **JTable**, mientras que *getPrintable()* está disponible para permitir manipulaciones del formato de impresión.

La aplicación `Java1424.java` presenta de forma sencilla un ejemplo del poder y flexibilidad que proporcionan los métodos que permiten imprimir un objeto de tipo **JTable**, en concreto el método *print()* y los distintos argumentos que se le pueden pasar. La declaración larga del método *print()* es la siguiente:

```
public bolean print( PrintMode printMode,
  MessageFormat headerFormat,
  MessageFormat footerFormat,
  boolean showPrintDialog,
  PrintRequestAttributeSet attr,
  boolean interactive ) throws PrintException
```

La clase **JTable** dispone de este mismo método con menos parámetros para utilización de forma más simple. En el caso anterior, el parámetro `printMode` permite indicar si la impresión volcará en papel el tamaño real de la tabla, `NORMAL`; o si por el contrario, ajustará la anchura de la tabla a la anchura de la página que utilice el papel, `FIT_WITH`. Los parámetros `headerFormat` y `footerFormat` establecen el texto a utilizar en cabecera y pie de página, respectivamente. El parámetro `showPrintDialog` indica si el diálogo de selección de impresora se presentará al usuario para que pueda configurar los parámetros de impresión correspondientes a la impresora que utilice, como número de copias, posición del papel normal o apaisada, etc. El parámetro `attr` sirve para indicar atributos de impresión. Y el parámetro `interactive` indica si se presentarán al usuario diálogos de estado en donde se indica cómo evoluciona la impresión, por ejemplo, indicando el número de página que se está enviando a la impresora, también permite abortar la impresión, porque en caso de que este parámetro sea `false`, el objeto **JTable** será impreso sin opción al usuario para que detenga esa impresión. En caso de que ocurra algún error de impresión, se lanzará una excepción de tipo **PrinterException**.

Cuando se ejecuta la aplicación, en pantalla aparecerá una ventana en donde se muestran los ficheros que contiene el directorio actual de ejecución y un diálogo en el que se pueden cambiar algunos de los parámetros del método *print()*. La figura 14.25 representa la ventana en ejecución.



Figura 14.25

Los controles de la parte inferior de la ventana permiten modificar la salida a impresión de la tabla, por ejemplo, cambiando la altura de las líneas seleccionadas, o indicando el texto que aparecerá en la cabecera y pie de página. La página impresa contendrá la tabla centrada sobre esa página, el texto de cabecera de cada columna etiquetando a cada una de ellas, un borde negro rodeando a la tabla y los textos indicados de cabecera y pie de página centrados por encima y por debajo de la tabla, respectivamente. La figura 14.26 muestra la salida por impresora generada por la ejecución de la aplicación, con los parámetros fijados en la figura 14.25.

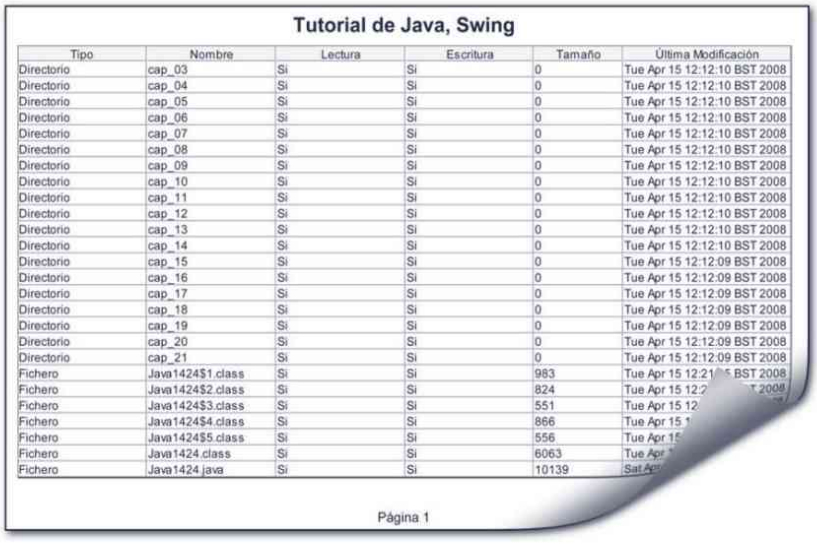


Figura 14.26

El lector puede cambiar otros parámetros en la ejecución del ejemplo y observar cómo se comporta la salida. Los parámetros corresponden a los argumentos asignados en la llamada al método *print()* de la instancia de **JTable** que genera la aplicación.

PESTAÑAS

En el capítulo correspondiente al AWT se introduce el controlador de posicionamiento de componentes **CardLayout**, y se explica y muestra cómo puede manejar diferentes fichas o tarjetas. No es un mal diseño, pero Swing introduce algo mucho más interesante y flexible, el **JTabbedPane**, que puede manejar directamente todas las fichas, a base de pestañas, y permite cambiar a cualquiera de ellas. El contraste entre la apariencia de **CardLayout** y **JTabbedPane** es impresionante.

El ejemplo `Java1425.java` se aprovecha de algunos de los ejemplos anteriores para poder implementar los diferentes paneles. Todos se construyen como descendientes de **JPanel**, así que en el programa se colocarán cada uno de los ejemplos anteriores en su propio panel dentro del **JTabbedPane**.

```
public class Java1425 extends JPanel {
  static Object objetos[][] = {
    { "AWT-Clon",Java1401.class},
    { "Bordes",Java1402.class},
    { "Botones",Java1404.class},
    { "Grupo de Botones",Java1406.class},
    { "Listas y Combo",Java1407.class},
    { "Barras",Java1417.class},
    { "Arbol",Java1420.class},
    { "Tabla",Java1423.class},
    };

  static JPanel creaPanel( Class clase ) {
    String titulo = clase.getName();
    titulo = titulo.substring( titulo.lastIndexOf('.') + 1 );
    JPanel panel = null;
    try {
      panel = (JPanel)clase.newInstance();
    } catch ( Exception e ) {
      System.out.println( e );
      }
    panel.setBorder( new TitledBorder( titulo ) );
    return( panel );
    }

  public Java1425() {
    setLayout( new BorderLayout() );
    JTabbedPane pestana = new JTabbedPane();
    for ( int i=0; i < objetos.length; i++ ) {
      pestana.addTab( (String)objetos[i][0],
        creaPanel( (Class)objetos[i][1] ) );
      }
    add( pestana,BorderLayout.CENTER );
    pestana.setSelectedIndex( objetos.length/2 );
    }
```

Se ha utilizado un array para la configuración: el primer elemento es el objeto **String** que será colocado en la pestaña y el segundo corresponde al objeto de la clase **JPanel** que será presentado dentro del correspondiente panel. En el constructor del ejemplo se puede observar que se utilizan los dos métodos más importantes del **JTabbedPane**: *addTab()* que crea una nueva pestaña y *setSelectedIndex()* que indica el panel que se va a presentar en primer lugar al arrancar el programa; en este caso se selecciona uno de en medio para mostrar que no es necesario comenzar siempre por el primero. La figura 14.27 reproduce esta ventana inicial, en la que aparecen algunos de los botones utilizados en los ejemplos anteriores de este capítulo.



Figura 14.27

En la llamada al método *addTab()* se le proporciona la cadena que debe colocar en la pestaña y cualquier componente del AWT. No es necesario que sea un componente Swing de tipo **JComponent**, sino cualquier componente del AWT, entre los cuales se encuentra, evidentemente, el **JComponent**, que deriva del **Component** del AWT. El componente se presentará sobre el panel y ya no será necesario ningún control posterior, el **JTabbedPane** se encarga de todo.

El trozo más interesante del código del ejemplo es el correspondiente al método *creaPanel()* que toma el objeto **Class** de la clase que se quiere crear y utiliza el método *newInstance()* para crear uno, moldeándolo a **JPanel**; desde luego, esto asume que la clase que se quiere añadir debe heredar de **JPanel**, sino no serviría de nada. Añade un **TitledBorder** que contiene el nombre de la clase y devuelve el resultado como un **JPanel** para que sea utilizado en *addTab()*.

```
static JPanel creaPanel( Class clase ) {
  String titulo = clase.getName();
  titulo = titulo.substring( titulo.lastIndexOf('.') + 1 );
  JPanel panel = null;
  try {
    panel = (JPanel)clase.newInstance();
  } catch( Exception e ) {
    System.out.println( e );
```

```
  }
  panel.setBorder( new TitledBorder( titulo ) );
  return( panel );
}
```

El ejemplo `Java1426.java` utiliza algunas de las características adicionales que se pueden emplear con **JTabbedPane**. En esta aplicación, las pestañas se muestran en la parte inferior y se ha añadido un mensaje de tipo *tooltip* a la segunda pestaña. También se ha incorporado un receptor de eventos que indicará la pestaña seleccionada. El código completo del ejemplo es el siguiente.

```
public class Java1526 {
  public static void main( String args[] ) {
    // Creamos dos paneles
    JPanel panel1 = new JPanel();
    JPanel panel2 = new JPanel();
    // Colocamos un botón en cada uno
    panel1.add( new JButton("Panel 1 - Tab 1") );
    panel2.add( new JButton("Panel 2 - Tab 2") );
    // Creamos el panel de pestañas y fijamos el receptor de
    // eventos de cambios de estado
    final JTabbedPane tabPanel = new JTabbedPane(
      SwingConstants.BOTTOM );
    tabPanel. addChangeListener( new ChangeListener() {
      public void stateChanged( ChangeEvent evt ) {
        int indice = tabPanel.getSelectedIndex();
        String titulo = tabPanel.getTitleAt( indice );
        System.out.println( "Tab = "+ titulo );
        }
      } );
    // Añadimos las pestañas, incluyendo tooltips, colores...
    tabPanel.add( "Pestaña 1",panel1 );
    tabPanel.addTab( "Pestaña 2",null,panel2,"Esta es la pestaña 2" );
    tabPanel.setForegroundAt( 0,Color.green );
    tabPanel.setBackgroundAt( 1,Color.blue );
    // Creamos la ventana
    JFrame f = new JFrame( "Tutorial de Java, Swing" );
    f.getContentPane().add( tabPanel,BorderLayout.CENTER );
    f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    f.setBounds( 200,200,220,100 );
    f.setVisible( true );
    }
  }
```

También es posible utilizar el teclado para seleccionar una pestaña; las teclas del cursor permiten cambiar de pestaña y el *Tabulador* permite desplazarse al componente interno del panel correspondiente a la pestaña seleccionada.

Otra característica incorporada a **JTabbedPane** es la posibilidad de que todas las pestañas puedan ser presentadas en una sola línea, aunque su dimensión sea superior; es decir, que la ventana se quede pequeña para el número de pestañas que se desea colocar en el panel. Anteriormente, Java sólo permitía el uso de múltiples líneas, lo cual era una verdadera pesadilla. Ahora, el componente **JTabbedPane** dispone de la propiedad **tabLayoutPolicy**, de forma que si se cambia la asignación inicial de

WRAP_TAB_LAYOUT por SCROLL_TAB_LAYOUT, las pestañas no ocuparán varias líneas, sino que se presentarán en una sola línea, apareciendo flechas de desplazamiento en la ventana para permitir acceder a las pestañas que no se visualizan en pantalla. La figura 14.28, capturada de la ejecución del ejemplo Java1427.java, muestra esta circunstancia.



Figura 14.28

El código completo de la aplicación que genera la imagen anterior es el que se muestra a continuación.

```
public class Java1427 {
  public static void main( String args[] ) {
    // Nombres de los meses para el botón
    String[] txtMeses = new DateFormatSymbols().getMonths();
    // Iniciales de los meses para las pestañas
    String[] meses = new DateFormatSymbols().getShortMonths();
    JTabbedPane tabPanel = new JTabbedPane();
    // Aseguramos que solamente haya una línea de pestañas
    tabPanel.setTabLayoutPolicy( JTabbedPane.SCROLL_TAB_LAYOUT );
    // Creamos los 12 paneles que corresponden a los meses, que
    // contendrán solamente un botón y la pestaña correspondiente
    for( int i=0; i < 12; i++ ) {
      JPanel panel = new JPanel( new BorderLayout() );
      // Creamos un botón en el que presentamos el nombre del
      // mes correspondiente a la pestaña seleccionada
      JButton boton = new JButton( txtMeses[i] );
      panel.add( boton );
      tabPanel.add( meses[i],panel );
      }
    // Creamos la ventana
    JFrame f = new JFrame( "Tutorial de Java, Swing" );
    f.getContentPane().add( tabPanel,BorderLayout.CENTER );
    f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    f.setBounds( 300,200,310,100 );
    f.setVisible( true );
    }
  }
```

En este código las líneas más importantes son las que se utilizan para cambiar la política de presentación de la pestañas del **JTabbedPane**.

```
    JTabbedPane tabPanel = new JTabbedPane();
    tabPanel.setTabLayoutPolicy( JTabbedPane.SCROLL_TAB_LAYOUT );
```

SELECTOR DE FICHEROS

Swing también proporciona nuevos modelos de diálogos predefinidos del sistema, como son el diálogo de selección de colores, el de selección de ficheros, los diálogos

de aviso, error y confirmación, y algunos más. La apariencia es muy distinta a la que se presentaba en el AWT en algunos casos, por ejemplo en la selección de ficheros.

La ventana que permite la selección de ficheros suele ser una ventana *modal*, ya que los cambios que se produzcan en ella, o la selección que se haga, repercutirán en el funcionamiento de la aplicación general. Normalmente, la ventana de selección de ficheros se utiliza para presentar una lista de ficheros y permitir al usuario seleccionar cuál de ellos debe abrir la aplicación; o, por el contrario, permitir al usuario la introducción de un nombre o selección del fichero en donde se quieren salvar datos. El objeto **JFileChooser** no realiza ninguna de estas acciones, es decir, no abre ni salva nada, sino que se limita a seleccionar el nombre del fichero con el que se desea realizar la acción; es responsabilidad del programa el llevar a cabo la apertura del fichero o la grabación de datos.

Tal como se muestra en la figura 14.29, generada tras la ejecución del ejemplo `Java1439.java` y realizado un montaje de las dos selecciones obtenidas tras la pulsación de cada botón presente en la ventana, se puede observar que la mejora que se ha incorporado a Swing es sustancial en comparación con su opción en AWT.

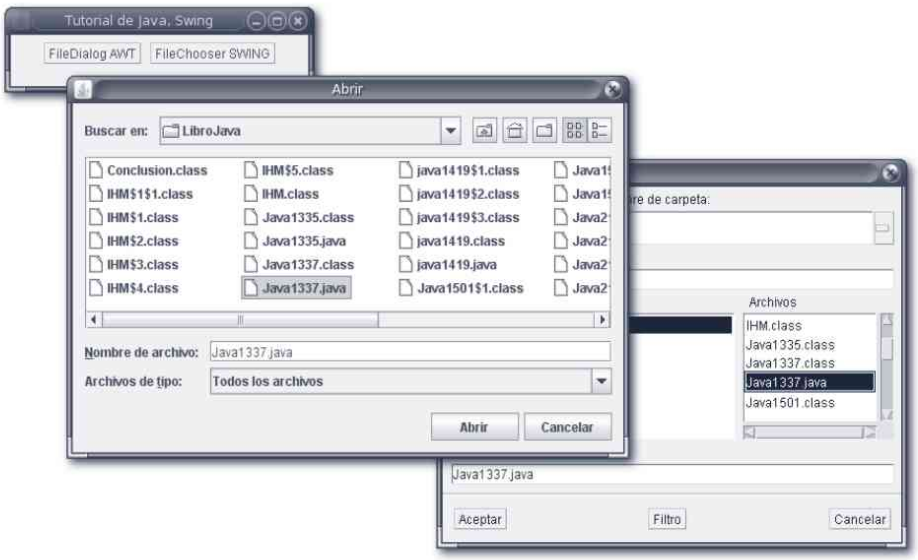


Figura 14.29

DIÁLOGOS PREDEFINIDOS

En el ejemplo `Java1428.java` se genera un diálogo de confirmación, utilizando el que por defecto proporciona Swing, del mismo modo que en los ejemplos siguientes se usan algunos de los demás diálogos predefinidos.

```
public class Java1428 extends JFrame implements ActionListener {
  public Java1428() {
    JButton boton = new JButton( "Muestra Ventana" );
    getContentPane().add( boton,"Center" );
    pack();
    boton.addActionListener( this );
```

```
  }
public void actionPerformed( ActionEvent evt ) {
  int res = JOptionPane.showConfirmDialog( this,"Responda Sí o No",
    "Tutorial de Java, Swing",JOptionPane.YES_NO_OPTION );
  String respuesta = null;
  if ( res == JOptionPane.YES_OPTION )
    respuesta = "Si";
  else
    respuesta = "No";
  System.out.println( "Respuesta: "+respuesta );
  }
public static void main( String args[] ) {
  new Java1428().setVisible( true );
  }
}
```

Si se ejecuta el programa anterior, la imagen que aparece en la pantalla (figura 14.30) corresponderá a la ventana de diálogo que va a presentar un mensaje y dos botones, uno para confirmación, en cuyo caso el **Panel** devuelve como respuesta la constante `YES_OPTION`, y otro para la desestimación.

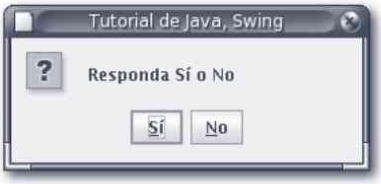


Figura 14.30

Así el ejemplo `Java1429.java` utiliza el diálogo de aviso y en el ejemplo `Java1430.java` se emplea el diálogo predefinido que permite introducir datos, y que resulta extremadamente útil cuando no se desea implementar ninguna interfaz de entrada, por ejemplo a la hora de pedir contraseñas de acceso, en que no merece la pena el desarrollo de ventanas específicas para tal acción. La figura 14.31 muestra ejemplos de estos dos diálogos predefinidos en Swing.



Figura 14.31

TECLADO

Algunas de las clases Java proporcionan métodos de conveniencia para permitir realizar acciones a través del teclado. Por ejemplo, la clase **AbstractButton** dispone del método *setMnemonic()*, que permite especificar el carácter, que en combinación con una de las teclas de modificación (dependiendo del *Look-and-Feel* que se esté utilizando), hace que se ejecuten las acciones asociadas a los botones. Esta característica es muy útil en los menús, tal como ya se vio en la clase **Java1415**.

El ejemplo `Java1431.java` es muy sencillo, pero muestra la forma en que se utilizan estos métodos. En la ventana aparecerán nueve botones.



Figura 14.32

Si se pulsan las teclas correspondientes a los cursores, la `X` que aparece sobre los botones se desplazará en la dirección correspondiente al botón que se haya pulsado.

```
public class Java1431 extends JFrame implements ActionListener {
  protected JButton botones[] = new JButton[9];

  public Java1431() {
    super( "Tutorial de Java, Swing" );
    Container pane = getContentPane();
    pane.setLayout( new GridLayout( 3,3 ) );
    Border borde = BorderFactory.createLineBorder( Color.black );
    KeyStroke arriba = KeyStroke.getKeyStroke( KeyEvent.VK_UP,0 );
    KeyStroke abajo = KeyStroke.getKeyStroke( KeyEvent.VK_DOWN,0 );
    KeyStroke izqda = KeyStroke.getKeyStroke( KeyEvent.VK_LEFT,0 );
    KeyStroke drcha = KeyStroke.getKeyStroke( KeyEvent.VK_RIGHT,0 );
    JRootPane rootPane = getRootPane();
    rootPane.registerKeyboardAction( this,"arriba",arriba,
      JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT );
    rootPane.registerKeyboardAction( this,"abajo",abajo,
      JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT );
    rootPane.registerKeyboardAction( this,"drcha",drcha,
      JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT );
    rootPane.registerKeyboardAction( this,"izqda",izqda,
      JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT );
    for( int i=0; i < 9; i++ ) {
      JButton boton;
      boton = new JButton();
      boton.setBorder( borde );
      boton.setName( new Integer(i).toString() );
```

```
      pane.add( boton );
      botones[i] = boton;
      }
    setSize( 200,200 );
    }

  public void actionPerformed( ActionEvent evt ) {
    Component foco = getFocusOwner();
    String nombre = foco.getName();
    int indice = Integer.parseInt( nombre );
    botones[indice].setText( "" );
    String accion = evt.getActionCommand();
    if( accion.equals( "arriba" ) ) {
      indice = (indice < 3) ? indice + 6 : indice - 3;
      }
    else if( accion.equals( "abajo" ) ) {
      indice = (indice > 5) ? indice - 6 : indice + 3;
      }
    else if( accion.equals( "izqda" ) ) {
      indice = (indice == 0) ? indice = 8 : indice - 1;
      }
    else { // asume drcha
      indice = (indice == 8) ? indice = 0 : indice + 1;
      }
    botones[indice].setText( "X" );
    botones[indice].requestFocus();
    }

  static public void main( String argv[] ) {
    new Java1431().setVisible( true );
    }
}
```

PANELES DESPLAZABLES

En anteriores programas se ha utilizado la clase **JScrollPane** para proporcionar una forma automática de desplazar el contenido de una ventana por parte del sistema. Esta clase permite dejar las manos libres al programador para centrarse en el código de su aplicación sin tener que estar pendiente de tener que controlar la visibilidad de todo o parte del contenido de la ventana. Ninguno de los ejemplos que se han visto se centraban en la clase **JScrollPane**, por lo que son muy simples, pero visto que es una clase muy usada y útil, el ejemplo `Java1432.java` sirve para presentar una aplicación un poco más avanzada y mostrar cómo interactúa en un programa. Además, el ejemplo utiliza la clase **JSplitPane**, que permite dividir la ventana en subventanas independientes.

El programa implementa un **JFrame** que contiene un panel dividido en dos zonas, cada una conteniendo una instancia de un componente Swing. La zona superior de la ventana contiene un campo de texto y la inferior un gráfico, para ilustrar el hecho de que el contenido de los paneles de desplazamiento puede ser cualquiera que se elija.

El código del ejemplo se muestra a continuación, y debería resultar sencillo de entender al lector, aunque hay algunos trozos sobre los que merece la pena detenerse y se discutirán a continuación.

```
class Java1432 extends JPanel implements ChangeListener {
  private JSplitPane panelVert;
  private JScrollPane panelScro1;
  private JScrollPane panelScro2;

  public Java1432() {
    setLayout( new BorderLayout() );
    // Se crean las zonas para mostrar el fichero y el gráfico
    creaPanelSup();
    creaPanelInf();
    // Se crea un panel dividido verticalmente
    panelVert = new JSplitPane( JSplitPane.VERTICAL_SPLIT );
    add( panelVert,BorderLayout.CENTER );
    // Se incorporan las dos zonas que se habían creado a las dos
    // partes en que se ha dividido el panel principal
    panelVert.setLeftComponent( panelScro1 );
    panelVert.setRightComponent( panelScro2 );
    }

  public void stateChanged( ChangeEvent evt ) {
    // Si el evento proviene del panel principal, seguimos...
    if ( evt.getSource() == panelScro1.getViewport() ) {
      // Miramos la posición actual dentro de la vista correspondiente
      // al panel principal
      Point point = panelScro1.getViewport().getViewPosition();
      // Ahora determinamos la escala correcta para las vistas, para
      // las dos zonas del panel principal
      Dimension dim1 = panelScro1.getViewport().getViewSize();
      Dimension dim2 = panelScro2.getViewport().getViewSize();
      float escalaX = 1;
      float escalaY = 1;
      if ( dim1.width > dim2.width ) {
        escalaX = (float)dim1.width / (float)dim2.width;
        escalaY = (float)dim1.height / (float)dim2.height;
        // Escalamos en función del movimiento
        point.x /= escalaX;
        point.y /= escalaY;
        }
      else {
        escalaX = (float)dim2.width / (float)dim1.width;
        escalaY = (float)dim2.height / (float)dim1.height;
        point.x *= escalaX;
        point.y *= escalaY;
        }

  private void creaPanelSup() {
    // Creamos el panel de la zona de texto
    JTextArea areaTexto = new JTextArea();
    // Se carga el fichero en el área de texto, procurando capturar
    // todas las excepciones que se puedan producir
    try {
      FileReader fileStream = new FileReader( "Java1432.java" );
      areaTexto.read( fileStream, "Java1432.java" );
```

```
    } catch( FileNotFoundException e ) {
      System.out.println( "Fichero no encontrado" );
    } catch( IOException e ) {
      System.out.println( "Error por IOException" );
      }
    // Creamos el panel desplazable para el área de texto
    panelScro1 = new JScrollPane();
    panelScro1.getViewport().add( areaTexto );
    panelScro1.getViewport().addChangeListener( this );
    }

  private void creaPanelInf() {
    // Cargamos el gráfico, o imagen, en la pantalla
    Icon imagenP2 = new ImageIcon( "imagenes/aguila.jpg" );
    JLabel etiqP2 = new JLabel( imagenP2 );
    // Creamos el panel para el gráfico
    panelScro2 = new JScrollPane();
    panelScro2.setVerticalScrollBarPolicy(
      ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER );
    panelScro2.setHorizontalScrollBarPolicy(
      ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
    panelScro2.getViewport().add( etiqP2 );
    panelScro2.getViewport().addChangeListener( this );
    }
```

Los métodos que se usan para crear el contenido de los subpaneles no son nada complicados. El método *creaPanelSup()* se limita a utilizar un componente **JTextArea** sobre el que carga el fichero que contiene el código fuente del ejemplo, y el método *creaPanelInf()* presenta una imagen que se desplazará en consonancia con el movimiento del panel superior.

Sin embargo, sí que hay un trozo de código interesante en el método *stateChanged()*, que controla los cambios producidos por acción del usuario, y es el cálculo de la escala que diferencia a los dos paneles de desplazamiento. En el programa, el cálculo de la escala no es demasiado preciso, debido a que no tiene en cuenta del todo el tamaño de los dos paneles. Si se mueve la barra horizontal en el panel superior, se podrá observar esta circunstancia. Con un poco más de matemáticas por parte del lector, seguro que podrá compensar el error, y a su ejercicio se deja esa corrección, teniendo en cuenta que hay que experimentar con los métodos *getViewPosition()* y *getExtentSize()* de la clase **JViewport** para conseguir que los cálculos sean correctos.

Cuando se ejecuta el programa y se amplía la ventana, se puede ver una imagen semejante a la que reproduce la figura 14.33.

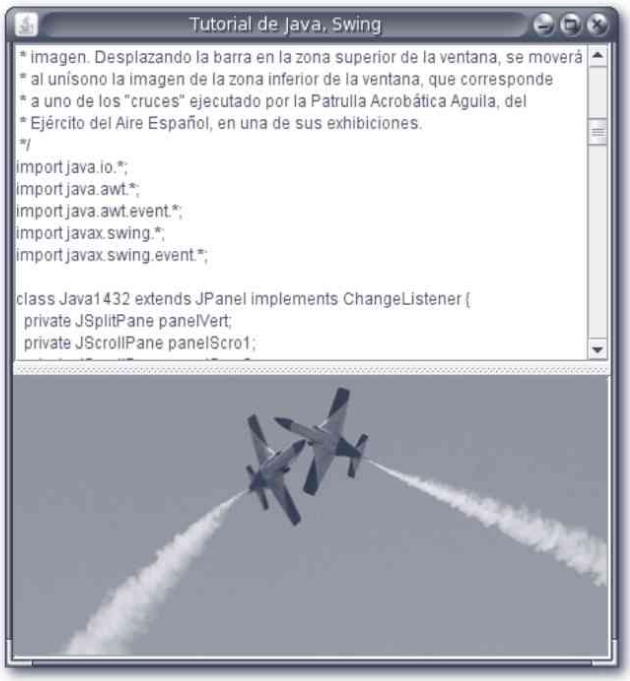


Figura 14.33

Ahora, los movimientos sobre la barra de desplazamiento vertical en el panel superior hacen que la imagen que está en el panel inferior se desplace al mismo tiempo. Quizá esto no le parezca al lector demasiado práctico, pero muestra cómo se puede conseguir el desplazamiento de un panel con respecto a otro; y este tipo de funcionalidad sí que es de uso frecuente en muchas aplicaciones, y aquí se muestra una forma muy sencilla de poder implementarla.

Otro detalle importante es la ausencia de barras de desplazamiento en el panel inferior, a pesar de que la imagen es mucho más grande que el tamaño de la subventana. Las líneas de código siguientes

```
panelScro2.setVerticalScrollBarPolicy(
  ScrollPaneConstants.VERTICAL_SCROLLBAR_NEVER );
panelScro2.setHorizontalScrollBarPolicy(
  ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER );
```

son las que permiten fijar las características de las barras de desplazamiento, y en este caso se indica que no deben estar presentes.

ARRASTRAR Y SOLTAR

La característica de Arrastrar y Soltar (*drag-and-drop*) es una de las más difíciles de implementar en el desarrollo de interfaces de usuario. Proporciona un alto nivel de usabilidad e intuición, pero suele intimidar un poco a los programadores. En esta sección se muestra al lector el uso de componentes que utilizan esta característica y cómo se incorporan a una aplicación Java.

Arrastrar

Dar soporte a esta acción, significa permitir a los usuarios hacer clic con el cursor sobre un objeto de la interfaz gráfica, mantener el objeto seleccionado y arrastrarlo hasta otra posición dentro de la misma aplicación, de otra aplicación Java o de otra aplicación nativa que se esté ejecutando en ese instante. Todas estas acciones se realizan a través de la clase **DragSource**, mediante los siguientes pasos:

1. Crear un objeto de tipo **DragSource**.
```
ds = new DragSource();
```
2. Consultar el objeto **DragSource** para obtener las características de un determinado componente; por ejemplo, un objeto de tipo **JList** que se extiende para añadirle el soporte *drag-and-drop*.
```
ds.createDefaultDragGestureRecognizer( this,
  DnDConstants.ACTION_MOVE,this );
```
3. Registrar una clase con el objeto **DataSource** para recibir notificaciones, a través de la interfaz **DragGestureListener**, cuando la característica *arrastrar* sea reconocida.
4. Crear una instancia de un objeto de tipo **Transferable** para colocar en él los datos que se van a permitir arrastrar. Se crea, por ejemplo, un objeto de tipo **StringSelection**, que implementa a **Transferable** y que va a permitir llevar el texto (en este caso) hacia el objeto sobre el que se soltará.
```
StringSelection ss = new StringSelection( sel.toString() );
```
5. Indicar a **DragSource** que comience la acción de arrastrar, invocando el método *startDrag()*.
```
ds.startDrag( dge,DragSource.DefaultMoveDrop,ss,this );
```
6. Recibir los eventos que se generan cuando un objeto de tipo **Transferable** es arrastrado sobre un objeto que acepte que se suelten otros objetos sobre él.
```
public void dragEnter( DragSourceDragEvent dsde ) {}
public void dragOver( DragSourceDragEvent dsde ) {}
public void dropActionChanged( DragSourceDragEvent dsde ) {}
public void dragExit( DragSourceEvent dse ) {}
```
7. Responder a la acción de soltar.
```
public void dragDropEnd( DragSourceDropEvent dsde ) {
  if( dsde.getDropSuccess() ) {
    System.out.println( "Acción de soltar correcta.." );
    } else {
    System.out.println( "Accion de soltar NO completada.." );
    }
  }
```

En la aplicación ejemplo `ListaArrastrar.java` se llevan a cabo todas las acciones correspondientes a los pasos anteriores, para convertir un objeto de tipo **JList**, en un objeto del mismo tipo pero con soporte para la característica de arrastrar sus elementos.

Soltar

La acción de soltar es la opuesta a la anterior. Cuando un objeto tiene la capacidad de que se puedan soltar otros objetos sobre él, necesita indicar al sistema operativo que dispone de dicha capacidad. La evaluación de los objetos arrastrados y el control de ellos, se realiza a través de la clase **DropTarget**, siguiendo los pasos que se indican a continuación, en donde como ejemplo, se añade soporte de la característica *soltar* a un objeto de tipo **JList**. El fichero `ListaSoltar.java` contiene el código fuente completo de la clase.

1. Crear un objeto de tipo **DropTarget**.
   ```
   dt = new DropTarget( this,this );
   ```
2. Indicar al objeto **DropTarget** que compruebe los objetos con la característica de *arrastrar*.
3. Notificar eventos de esta clase a través de la interfaz **DropTargetListener**.
4. Manejar las notificaciones cuando un objeto es arrastrado sobre el componente.
   ```
   public void dragExit( DropTargetEvent dte ) {}
   public void dragEnter( DropTargetDragEvent dtde ) {}
   public void dragOver( DropTargetDragEvent dtde ) {}
   public void dropActionChanged( DropTargetDragEvent dtde ) {}
   ```
5. Controlar la acción soltar, decidiendo si se admite o no el objeto arrastrado.
   ```
   dtde.acceptDrop( DnDConstants.ACTION_MOVE )
   ```
6. Extraer el objeto **Transferable** del objeto que se ha soltado y utilizarlo.
   ```
   String s = (String)t.getTransferData( DataFlavor.stringFlavor );
   ```
7. Completar la acción soltar, notificándolo al objeto **DragSource**.
   ```
   dtde.getDropTargetContext().dropComplete( true );
   ```

La aplicación `Java1440.java` engloba las dos clases definidas previamente, permitiendo la acción de arrastrar elementos desde el componente origen y la acción de soltarlos sobre el componente destino.

Las clases anteriores aplicadas a un objeto de tipo **JList**, ilustran perfectamente la implementación de la característica de arrastrar y soltar, aunque hay ocasiones en que se desea incorporar dicha característica a controles más avanzados, como pueden ser tablas o árboles. La gran ventaja de Swing consiste en que una vez captado un concepto, ese mismo concepto se puede aplicar siempre.

La aplicación `Java1441.java` implementa un objeto de tipo **JTree** con soporte de la característica de arrastrar y soltar. Incorpora las clases y receptores de eventos para dar soporte a esta característica y luego construye una cadena XML que es la que se pasa al objeto al cual se arrastre. La diferencia entre el árbol y la tabla o la lista, es la obtención del objeto que está siendo arrastrado y la identificación del objeto sobre el cual se va a soltar.

Si el lector consulta el código del ejemplo, observará que los distintos constructores que proporciona la clase invocan al método *init()*.

```
protected void init() {
  ds = new DragSource();
  ds.createDefaultDragGestureRecognizer( this,
    DnDConstants.ACTION_COPY,this );
  dt = new DropTarget( this,this );
  }
```

Primero se crea una instancia de **DragSource** para controlar el proceso de arrastre. Luego consulta a esa instancia si se trata del árbol de la aplicación (el primer **this**) y se registra para recibir notificaciones de eventos de tipo **DragGestureListener** (el segundo **this**). Finalmente, crea una instancia de **DropTarget** para controlar el árbol (el primer **this**) y recibir eventos de tipo **DropTargetListener** cuando el objeto que se arrastra entra, sale o se suelta sobre el árbol (el segundo **this**).

La construcción del objeto transferible con todos los objetos seleccionados se realiza mediante la invocación del método *getSelectionPaths()*. Se crea un objeto XML conteniendo la lista de *paths* que se convierte a un objeto de tipo **String**, que es moldeado a un objeto arrastrable de tipo **StringSelection**.

Cuando un objeto se suelta sobre el árbol, el método *drop()* será invocado. En este ejemplo se llama al método *getLocation()* de la clase **DropTargetDropEvent** para obtener el punto donde el objeto se ha soltado. Luego se convierte a un objeto de tipo **TreePath** llamando al método *getPathForLocation()*. A partir del objeto **TreePath** se llama al método *getLastPathComponent()* para obtener el nodo sobre el que se puede trabajar.

El bucle se completa con la llamada a *dropComplete()*, que invoca al método *dragDropEnd()* de la clase **DragSource**.

La aplicación también ilustra cómo determinar si la acción de *soltar* ha tenido éxito y cómo diferenciar entre operaciones de copiar o mover. El lector puede utilizarla como plantilla para sus propias aplicaciones.

LOOK AND FEEL

Cada ejecutable Java tiene un objeto **UIManager** que determina *el Look-and-Feel*, es decir, la *apariencia* en pantalla y el *funcionamiento* que van a tener los componentes Swing de ese ejecutable. El *Look-and-Feel* es la apariencia que se proporciona a los diferentes componentes: botones, cajas de texto, cajas de selección, listas, etc.

Java utiliza la interfaz gráfica de la plataforma sobre la que se está ejecutando para presentar los componentes del AWT con el aspecto asociado a esa plataforma, de este modo los programas que se ejecuten en *Windows* tendrán esa apariencia y los que se ejecuten en Unix tendrán apariencia *Motif*. Pero Swing permite la selección de esta apariencia gráfica, independientemente de la plataforma en que se esté ejecutando; tanto es así que la apariencia por defecto de los componentes Swing se denomina

Metal, y es propia de Java, J2SE 5 introdujo un *tema* más actual llamado *Ocean*, basado en *Metal* y J2SE 6 implementa también el tema *Nimbus*, desarrollado para el *Sun Java Desktop*. Teniendo siempre en cuenta las restricciones impuestas por el control de seguridad, se puede seleccionar la apariencia, o *Look-and-Feel* de los componentes Swing invocando al método *setLookAndFeel()* del objeto **UIManager** correspondiente al ejecutable. La forma en que se consigue esto es porque cada objeto **JComponent** tiene un objeto **ComponentUI** correspondiente que realiza todas las tareas de dibujo, manejo de eventos, control de tamaño, etc. para ese **JComponent**.

Una de las ventajas que representa las capacidades de *Look&Feel* incorporadas a Swing para las empresas, es el poder crear una interfaz gráfica estándar y corporativa. Con el crecimiento de las intranets se están soportando muchas aplicaciones propias que deben ejecutarse en varias plataformas, ya que lo más normal es que en una empresa haya diferentes plataformas. Swing permite ahora que las aplicaciones propias diseñadas para uso interno de la empresa tengan una apariencia exactamente igual, independientemente de la plataforma en que se estén ejecutando. Y es que hay pequeñas cosas que vuelven loco al usuario al cambiar de entorno, por ejemplo:

¿Ratón de dos o tres botones?
¿Clic simple o doble para activar aplicaciones?
¿La ventana activa sigue al foco o no lo sigue?
Diferentes esquemas de color
Diferentes aproximaciones para proporcionar accesibilidad
Diferente conjunto de iconos, o widgets
Diferente nomenclatura, geografía, efectos...

La verdad es que la frase hecha famosa por **Sun**, en uno de sus anuncios de Java "*Escribir una vez, ejecutar en cualquier lugar*", no se ha hecho realidad hasta que han aparecido las *Java Foundation Classes*, incorporando Swing. Generalmente, el código Java, con AWT, se puede desarrollar y probar completamente sobre una plataforma, pero debido a las diferencias entre las máquinas virtuales de las distintas plataformas o sistemas operativos, se pueden presentar desde pequeños problemas de presentación gráfica hasta grandes inconsistencias, que pueden llegar a que la aplicación se interrumpa durante la ejecución o, sencillamente, que no arranque. El problema en realidad no es Java, sino el AWT; más concretamente, los elementos que los diseñadores del AWT han tenido que dejar de lado para hacer portable la interfaz de usuario.

Con Swing estos problemas quedan relegados a la historia y se contarán como anécdotas. Utilizando las librerías de *Look&Feel*, las aplicaciones se pueden escribir y probar utilizando una única interfaz. Cuando esas aplicaciones sean portadas a otras plataformas, la interfaz permanecerá invariable.

No obstante, las librerías que vienen con Swing pueden resultar indeseables en algunas ocasiones; algunas se ejecutan sobre una sola plataforma, otras no satisfacen

el gusto de los usuarios. Así que también se puede crear un *Look&Feel* propio que satisfaga a todos y simplifique la transferencia de código de una plataforma a otra.

En general, para la implementación de una apariencia nueva, hay que tener en cuenta que aunque en principio pueda parecer muy interesante, y se piense que todas las ideas son buenas para mejorar el diseño de una interfaz gráfica; los usuarios que ya están utilizando el ordenador están acostumbrados a lo que les ofrece la plataforma con la que trabajan, y la interfaz que se diseñe ha de resultarles familiar; en caso contrario la curva de aprendizaje de la nueva interfaz puede resultar tan abrupta que no se tome en cuenta el nuevo diseño. Por lo tanto, se debe utilizar con cautela el poder que pone en manos de los programadores/diseñadores la posibilidad de desarrollar un *Look&Feel* propio.

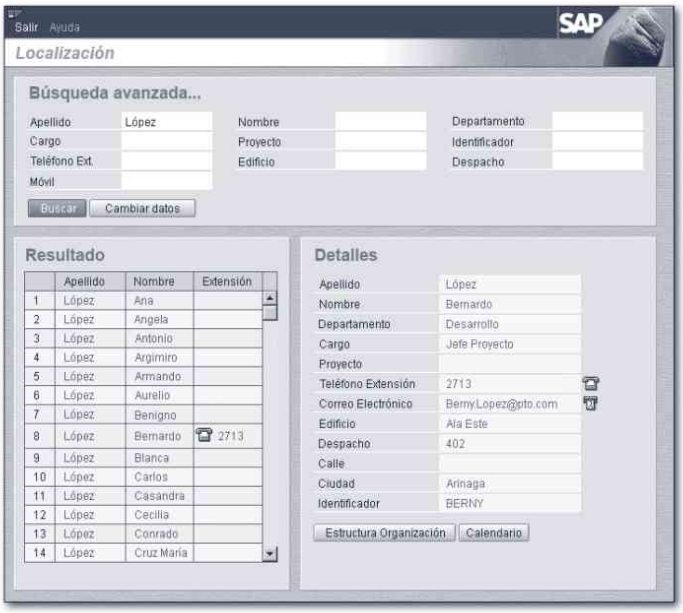


Figura 14.34

No obstante, y como el lector podrá observar en los ejemplos de interfaces que se muestran, en el caso de crear una nueva interfaz, hay que tener presente infinidad de pequeños detalles, por ejemplo, que la imagen es mucho más importante que el borde, es decir, que no tiene por qué haber un borde separando todos los elementos. O también que el color gris no es el único color neutro que se puede utilizar, y el entorno debe ser consistente con los que ya se conocen. Precisamente la figura 14.35 reproduce ventanas creadas con algunos de los widgets más comunes en los entornos gráficos utilizados actualmente.

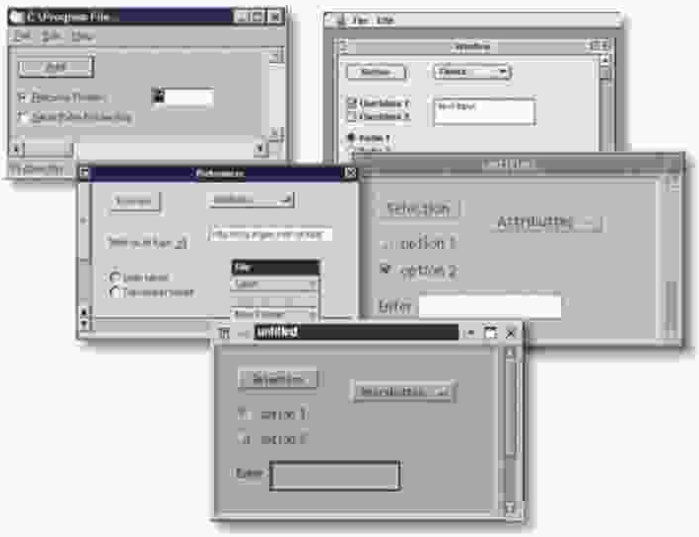


Figura 14.35

Java incorpora su propio *Look&Feel* por defecto, cuyo nombre es *Ocean*, aunque en las últimas actualizaciones se incorpora *Nimbus*; como ya se ha indicado, ambos son una evolución del *Look&Feel Metal*, utilizado por Swing hasta el J2SE 5. En realidad, tanto *Ocean* como *Nimbus* son temas o *pieles* para *Metal*. Su apariencia es la que ha podido observar el lector en muchos de los ejemplos anteriores.

El *Look&Feel Ocean*, que se ha utilizado en casi todos los ejemplos de este capítulo, se basa en unos principios muy sencillos, para así poder adaptarse mucho mejor a los gustos del usuario. Por ejemplo, su esquema de colores es muy simple, utilizando solamente 8 colores y pudiendo seleccionarse cualesquiera, lo que hace que ya haya muchos usuarios que se han dedicado a crear *Temas*, o esquemas de color, para satisfacer cualquier gusto por parte del usuario que se siente delante de la pantalla. Además de los colores, se permite cambiar los iconos, las fuentes de caracteres y los bordes.

Como se ha citado en párrafos anteriores, *Ocean* es un tema para el *Look&Feel Metal*, que fue el utilizado anteriormente por Swing y con el cual es plenamente compatible. La apariencia original del *Look&Feel Metal* todavía puede utilizarse empleando el parámetro *steel* en la propiedad `swing.metalTheme`.

La arquitectura *Look&Feel* permite que una aplicación pueda presentarse bajo distintas apariencias utilizando los mismos componentes del API. Esta característica se consigue con la separación de las clases encargadas de pintar los componentes de las clases que se encargan de la gestión de eventos. Así, es posible escribir componentes del API que se ejecuten bajo cualquier *Look&Feel*, sin tener que hacer cambio alguno en las aplicaciones. Todas estas clases derivan de la clase **ComponentUI**, por ejemplo, **JButton** delega todas las acciones de pintado a una subclase de **ComponentUI**, que es **ButtonUI**, responsable de manejar la apariencia del botón en pantalla e instalar los receptores de eventos del ratón que responderán al funcionamiento que se espera de un botón.

El siguiente código correspondería a una aplicación que se encarga de visualizar un botón en una ventana:

```
JButton boton = new JButton( "Botón" );
JFrame f = new JFrame( "Botón" );
f.add( boton );
f.pack();
f.setVisible( true );
```

Si al botón simple creado mediante el código anterior se le quiere añadir un fondo degradado cuando el ratón pasa por encima, es necesario crear una clase **ButtonUI** con dicha característica. Para hacerlo, se extiende una implementación ya hecha de **ButtonUI**, como puede ser la que proporciona el *Look&Feel* de Java, que en este caso corresponde a *MetalButtonUI*. El código que generaría el efecto descrito se reproduce a continuación:

```
ButtonUI degradadoUI = new MetalButtonUI() {
  public void paint( Graphics g,JComponent c ) {
    // Realmente, aquí habría que utilizar primero instanceOf(), para
    // controlar el caso en que "g" no pertenezca a Graphics2D, pero
    // esto solamente ocurre en ciertas ocasiones en que se trata de
    // imprimir objetos, así que no afecta en este caso
    Graphics2D g2 = (Graphics2D)g;
    // Configuramos el gradiente para que sea un degradado desde el
    // color que se pasa en primer lugar al color correspondiente
    // al que se indica en último lugar y de arriba-abajo
    g2.setPaint( new GradientPaint(
      0,0,claro,
      0,c.getHeight(),oscuro ) );
    g.fillRect( 0,0,c.getWidth(),c.getHeight() );
    g2.setPaint( null );
    super.paint( g,c );
    }
  };
```

Para hacer que el botón anterior contenga esta característica, basta con indicárselo mediante la sentencia:

```
boton.setUI( degradadoUI );
```

Si se presentase el botón anterior en pantalla, al pulsar sobre él, el degradado se perdería porque el botón cambia de estado y la clase **MetalButtonUI** rellena el color de fondo por defecto. Para evitar también esa circunstancia es necesario sobrescribir el método *paintButtonPressed()*, que es el responsable de fijar el color de fondo del botón cuando se pulsa. El código siguiente muestra el método ya sobrescrito, que se encargará de invertir el degradado cuando se pulse el botón:

```
  protected void paintButtonPressed( Graphics g,AbstractButton b ) {
    // El mismo comentario de antes acerca de instanceof()
    Graphics2D g2 = (Graphics2D)g;
    // Configuramos el gradiente para que sea un degradado invertido
    // de los colores utilizados en el método paint.
    g2.setPaint( new GradientPaint(
      0,0,oscuro,
```

```
  0,b.getHeight(),claro ) );
g.fillRect( 0,0,b.getWidth(),b.getHeight() );
g2.setPaint( null );
}
```

Si el lector ejecuta la aplicación `Java1433.java`, observará el botón en funcionamiento, con la apariencia especificada en los trozos de código presentados, tal como reproduce la figura 14.36 correspondiente a la imagen del botón en estado pulsado.



Figura 14.36

Para personalizar la apariencia de otros componentes visuales de Swing se pueden crear elementos de tipo **ComponentUI** de modo semejante, sobrescribiendo los métodos de pintado que sean necesarios. Este proceso requiere un profundo conocimiento de cómo está implementada cada una de las clases **ComponentUI**, y hay más de una treintena, con lo que el trabajo de crear un *Look&Feel* no es precisamente sencillo; a pesar de que el lector tenga esa impresión a través de lo comentado en párrafos anteriores para cambiar la apariencia de un botón, porque hay ciertos componentes que requieren sobrescribir un gran número de métodos y reescribir una gran cantidad de código para controlar su funcionamiento.

En ayuda de los desarrolladores llega ***Synth***, que ha sido diseñado con la pretensión de facilitar y simplificar la creación de apariencias, *Look&Feel*, personalizados. En el caso de Synth solamente hay que concentrarse en las clases encargadas de pintar y no en todo lo que concierne a las clases **ComponentUI**, de hecho, todas las clases **ComponentUI** que forma parte de Synth son privadas.

Synth es un *Look&Feel* personalizable a base de *skins*, donde cada *skin*, *piel* o tema, es decir, la interfaz de usuario, se configura mediante un fichero XML. Esta circunstancia posibilita la carga de un fichero XML con la definición de los componentes, sin necesidad de tener que personalizar el *Look&Feel* proporcionando propiedades por defecto para el **UIManager** en una tabla de propiedades. Esto significa que se puede personalizar la apariencia de la aplicación, sin escribir una sola línea de código.

Para poner en marcha un *Look&Feel* basado en Synth, es necesario un paso adicional respecto a cualquier otro *Look&Feel*, porque se debe cargar el fichero XML que contiene la configuración de los componentes. Por tanto, para especificar un *Look&Feel* Synth hay que hacer lo siguiente:

1. Crear un objeto de tipo **SynthLookAndFeel**.
   ```
   SynthLookAndFeel laf = new SynthLookAndFeel();
   ```

2. Cargar el fichero de configuración mediante el método *load()*. Este método tiene dos parámetros, el primero corresponde a la URL del fichero a cargar y el segundo a la clase utilizada para resolver los *paths* de la dirección URL. Por ejemplo, si se pasa la clase **com.paquete.Clase** como segundo parámetro del método *load()*, todos los *path* serán realativos al paquete **com.paquete**.
```
laf.load( Java1434.class.getResourceAsStream(fichero),
  Java1434.class );
```

3. Activar el *Look&Feel* y fijarlo como activo.
```
UIManager.setLookAndFeel( laf );
```

En el ejemplo `Java1434.java` se pueden comprobar estos tres pasos con la carga del fichero *java1434.1.xml*, que define la apariencia que debe presentar Synth, cuyo contenido es el siguiente.

```
<synth>
  <!-- Creamos un estilo que afectará a todos los widgets -->
  <style id="tutorialStyle">
    <!-- Hacemos que todos los widgets usen este skin opaco -->
    <opaque value="TRUE"/>
    <!-- Fijamos la fuente de caracteres y su tamaño -->
    <font name="Dialog" size="48"/>
    <state>
      <!-- Fijamos los colores de fondo y del primer plano -->
      <color value="WHITE" type="BACKGROUND"/>
      <color value="BLACK" type="FOREGROUND"/>
    </state>
  </style>
  <!-- Asociamos este estilo a todos los widgets -->
  <bind style="tutorialStyle" type="region" key=".*"/>
</synth>
```

Este código muestra un conjunto de los componentes básicos de Synth. El elemento `style` define un objeto de tipo **SynthStyle** consistente en una serie de propiedades seleccionadas con un estilo como: fuentes, opacidad, márgenes, etc. Cada componente está asociado al menos a un **SynthStyle**. Los objetos **ComponenteUI** de Synth utilizan objetos **SynthStyle** para obtener toda la información relacionada con el estilo. Por ejemplo, cada objeto **ComponentUI** es Synth ejecuta el siguiente código para instalar una fuente de caracteres:

```
Font fuente = componente.getFont();
if( fuente == null || (fuente instanceOf UIResource) ) {
  componente.setFont( estilo.getFont( contexto ) );
  }
```

El fichero XML que se mostraba anteriormente crea un nuevo estilo con el identificador `tutorialStyle`, fija la fuente de caracteres a utilizar y su tamaño, los colores de fondo y para el texto en primer plano e indica que todos los componentes deben utilizar ese estilo. Si se ejecuta la aplicación `Java1434.java` que carga este estilo, en pantalla aparecerá una imagen semejante a la que reproduce la figura 14.37.



Figura 14.37

En el fichero XML, el elemento `bind` se utiliza para asociar un **SynthStyle** a un conjunto de componentes. Se puede asociar un estilo a tipos de componentes indicando *region* como valor del atributo `type`, o directamente a componentes individuales indicando *name* como valor de `type`. En el ejemplo se asocia el estilo **SynthStyle** con todos los componentes. Si se quiere asociar a un tipo determinado, se utiliza la constante definida en la clase **Region** para ese componente, sin los guiones bajos. Por ejemplo, para asociar el estilo anterior a componentes de tipo campo de texto, se especificaría *region* como valor del atributo `type` y *textfield* como valor del atributo `key`.

Cada objeto Synth tiene un objeto **SynthPainter** asociado, que es utilizado para renderizar las distintas partes en que se divide cada componente Swing. Synth permite así personalizar la forma en que se presentarán imágenes, dividiéndolas en nueve zonas, tal como se muestra en la figura 14.38. Las esquinas siempre permanecen fijas, mientras que las zonas intermedias son ampliadas, reducidas o reproducidas como mosaico, dependiendo de la forma de la imagen que se ha de asociar al componente.

Continuando con el ejemplo anterior, se puede asociar una imagen al botón que define la aplicación, el cual será dividido en las nueve zonas descritas. La figura 14.38 muestra la imagen del botón ampliada varias veces para presentar las nueve zonas que considera Synth, antes de asociarlo al botón Swing.

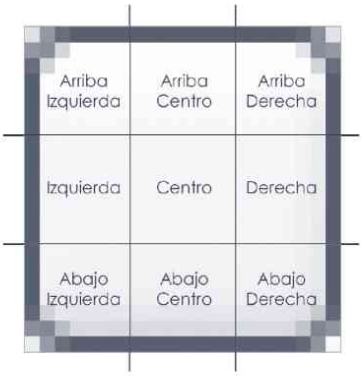


Figura 14.38

Para ver el ejemplo en funcionamiento, basta ejecutar la aplicación `Java1434.java` pasando como último parámetro en la línea de comando el fichero XML *java1434.2.xml*.

Ejecutando el ejemplo, cuando se pulsa el botón no se produce cambio visual alguno. Para proporcionar los elementos visuales correspondientes a los distintos

estados del botón, es necesario especificar un objeto **imagePainter** para cada uno de esos estados. El fichero *java1434.3.xml* contempla los estados del botón y, si el lector ejecuta la aplicación `Java1434.java` con ese fichero XML como parámetro, podrá ver cómo ahora el botón sí responde visualmente a los diferentes eventos generados por el ratón sobre él. El estado correspondiente al botón queda pues como sigue:

```
<style id="botonStyle">
  <!-- Indicamos que cuando se pase a Pulsado, el texto del botón debe
    desplazarse 1 pixel -->
  <property key="Button.textShiftOffset" type="integer" value="1"/>
  <!-- Establecemos un margen alrededor del contenido interno del
    botón -->
  <insets top="6" left="10" right="10" bottom="6"/>
  <state>
    <!-- Asignamos una imagen al fondo del botón, de forma que se
      visualice el fondo de botón con borde -->
    <imagePainter method="buttonBackground" path="imagenes/boton.png"
      sourceInsets="10 10 10 10"/>
  </state>
  <state value="MOUSE_OVER">
    <!-- Asignamos una imagen al botón para indicar visualmente el
      estado MOUSE_OVER, el cursor está sobre el botón -->
    <imagePainter method="buttonBackground" path="imagenes/botonm.png"
      sourceInsets="10 10 10 10"/>
  </state>
  <state value="PRESSED">
    <!-- Asignamos una imagen al botón para indicar visualmente el
      estado PULSADO -->
    <imagePainter method="buttonBackground" path="imagenes/botonp.png"
      sourceInsets="10 10 10 10"/>
  </state>
</style>
<!-- Asociamos este estilo a todos los objetos JButton -->
<bind style="botonStyle" type="region" key="button"/>
```

Como los elementos `state` añadidos para indicar el estado pulsado del botón y cuando el cursor pasa sobre él, son hijos de cada uno de esos estados, solamente estarán en funcionamiento cuando estos eventos de ratón tengan lugar.

También se incorpora la etiqueta `property`, que se utiliza para indicar propiedades específicas de determinados componentes. En este caso, la propiedad `Button.textShiftOffset` indica que cuando se pulse el botón para lanzar el evento *PRESSED*, el texto debe ser desplazado.

Hasta ahora se ha mostrado cómo Synth actúa sobre un componente utilizando un botón como ejemplo. Pero Synth puede actuar en base a regiones y no necesariamente sobre componentes, como ya se ha citado en párrafos anteriores. Hay componentes, que reciben el nombre de componentes compuestos, formados por múltiples regiones y, por tanto, pueden tener asociados múltiples estilos. Por ejemplo, una barra de desplazamiento, implementada mediante un objeto de tipo **JScrollBar**, está constituida por varias regiones: barra de desplazamiento, zona de desplazamiento, marcador y botones de dirección. La región *barra de desplazamiento* se refiere al

componente en su totalidad, incluyendo las demás regiones. La figura 14.39 muestra las regiones que componen la barra de desplazamiento.

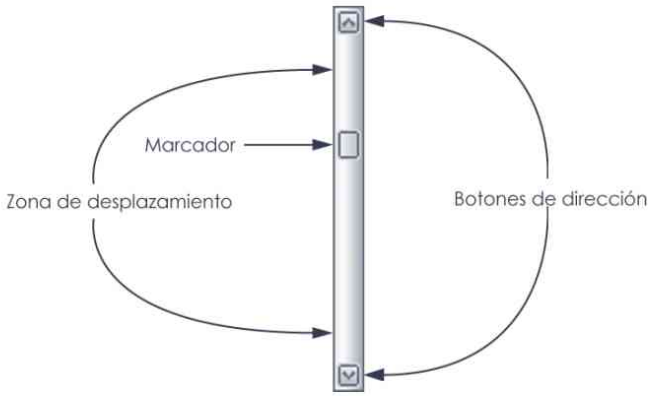


Figura 14.39

La aplicación `Java1435.java` muestra un ejemplo más complejo, con los componentes anteriores más un panel con pestañas, un menú superior, menús desplegables y un campo de texto. El fichero *java1435.1.xml* es el que contiene la descripción del formato de los componentes.

Una posibilidad que ofrece Syth es la admisión de objetos **SynthPainter** propios. Se utilizan cuando los objetos **imagePainter**, como los utilizados en los ejemplos anteriores, resultan demasiado limitados. Por ejemplo, cuando se quiere incorporar una animación o un degradado. Synth controla la forma de pintar los elementos gráficos a través de la clase **SynthPainter**. Esta clase define un método para pintar el borde y el fondo de cada región; por ejemplo, **SynthPainter** dispone de los métodos *paintButtonBorder()* y *paintButtonBackgroud()* para realizar las acciones anteriores sobre un objeto de tipo **JButton**. En los ejemplos anteriores se ha empleado XML para crear indirectamente los objetos **SynthPainter**, pero también se puede crear una clase propia derivada de **SynthPainter** e incluirla en un fichero Synth utilizando la persistencia de *JavaBeans*. Para incluir este tipo de objetos se usa el formato siguiente:

```
<synth>
  <object class="MiClase" id="miClaseId" />
</synth>
```

Para crear un objeto **SynthPainter** propio; por ejemplo, para colocar una línea degradada bajo el texto correspondiente a la pestaña que se encuentre seleccionada, en el ejemplo anterior. Para conseguir esta presentación hay que extender la clase **SynthPainter**, en este caso creando la clase **TabPainter**, que se reproduce a continuación:

```
import java.awt.*;
import javax.swing.plaf.synth.*;

public class TabPainter extends SynthPainter {
  // Colores que definen el degradado
  private static final Color COLOR_INICIAL = new Color( 0x4F6B81 );
```

```
  private static final Color COLOR_FINAL = new Color( 0xB7BFC0 );

  public void paintTabbedPaneTabBorder( SynthContext context,
    Graphics g,int x,int y,int w,int h,int tabIndex ) {
    Graphics2D g2 = (Graphics2D)g;
    // Nos colocamos en la mitad de la línea
    g2.translate( x+w/2, y+h-5);
    // Hacemos el degradado hacia un lado
    int tamLinea = w / 2 - 4;
    g2.setPaint( new GradientPaint(
      0,0,COLOR_INICIAL,tamLinea,0,COLOR_FINAL) );
    g2.fillRect( 0, 0, tamLinea, 2);
    // hacemos el degradado hacia el otro lado
    g2.setPaint( new GradientPaint(
      -tamLinea,0,COLOR_FINAL,0,0,COLOR_INICIAL) );
    g2.fillRect( -tamLinea,0,tamLinea,2 );
    g2.setPaint( null );
    g2.translate( -(x+w/2),-(y+h-5) );
    }
  }
```

En el fichero *java1436.1.xml* se incluye en la definición de la pestaña y ejecutando la aplicación `Java1436.java` pasando ese fichero como parámetro, aparecerá la pestaña en la pantalla. Sin embargo, en el fichero *java1436.2.xml* se incluye la definición de la pestaña que emplea la línea degradada creada por la clase **TabPainter** para indicar la pestaña que se encuentra seleccionada. La definición de la pestaña en el fichero XML quedaría de la siguiente forma:

```
<style id="tabPaneTabStyle">
  <opaque value="TRUE"/>
  <insets top="5" left="5" right="5" bottom="4"/>
  <imagePainter method="tabbedPaneTabBackground"
    path="imagenes/tab.png"
    sourceInsets="3 4 2 5" paintCenter="true"/>
  <object class="TabPainter" id="tabPainter"/>
  <state value="SELECTED">
    <painter method="tabbedPaneTabBorder"
                idref="tabPainter"/>
  </state>
</style>
<bind style="tabPaneTabStyle" type="region" key="tabbedpanetab"/>
```

Si se compila el fichero fuente `TabPainter.java` y se ejecuta la aplicación `Java1436.java` con el comando:

```
% java Java1436 java1436.2.xml
```

en la pantalla aparecerá una ventana semejante a la que se reproduce a continuación en la figura 14.40.

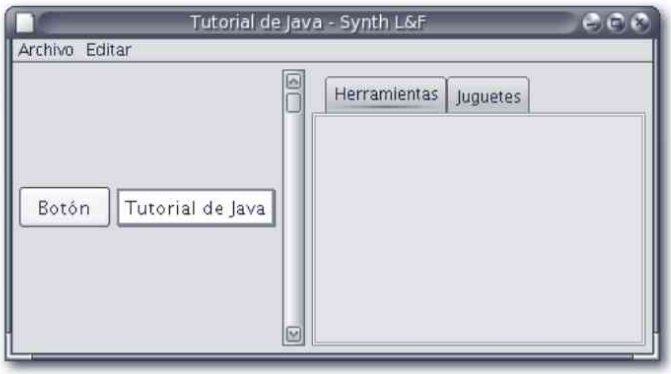


Figura 14.40

Synth también permite su extensión mediante código Java, sin tener que recurrir a un fichero de configuración, para aquellos casos en que sea más simple esta solución; por ejemplo, hay apariencias en las que sólo se utilizan líneas, en cuyo caso es más rápido crear un **SynthLookAndFeel** propio.

Como se ha indicado anteriormente, cada región de un componente tiene su propio estilo. La clase **ComponentUI** en Synth obtiene los estilos a utilizar desde **SynthStyleFactory**, una clase abstracta con un único método *getStyle()*.

Cuando se utiliza el fichero de configuración XML para Synth, indirectamente se está creando un objeto **SynthStyleFactory** que devuelve estilos personalizados basados en el contenido de ese fichero. Si en lugar de utilizar el fichero se desea crear una apariencia Synth con una factoría **SynthStyleFactory** propia, hay que incluir las siguientes líneas de código:

```
SynthLookAndFeel laf = new SynthLookAndFeel();
UIManager.setLookAndFeel( laf );
SynthLookAndFeel.setFileFactory( new TutorialEstiloFactory() );
```

La clave está en la última línea, en la cual se indica la factoría encargada de proporcionar los estilos de tipo **SynthStyle** para las distintas regiones. La clase **SynthStyle** es abstracta y contiene dos métodos que todas las subclases deben implementar: *getColorForStyle()* y *getFontForStyle()*, que devuelven objetos de tipo **Color** y **Font**, respectivamente.

A la hora de crear una factoría de estilos propia, se necesita crear una implementación concreta de **SynthStyle**. Por ejemplo, para crear un estilo Synth que pinte un campo de texto al estilo clásico de los controladores de ventanas, en primer lugar hay que crear un estilo que utilicen todos los componentes en general. El código siguiente de la clase **EstiloPorDefecto** muestra un ejemplo.

```
private static class EstiloPorDefecto extends SynthStyle {
  private static Font FONT_POR_DEFECTO = new FontUIResource(
    "Helvetica",Font.PLAIN,14 );
  private static Color COLOR_PRIMER_PLANO = new ColorUIResource(
    Color.BLACK );
  private static Color COLOR_FONDO = new ColorUIResource(
```

```
    Color.WHITE );

  // Este método se invoca en dos ocasiones: cuando se quiere fijar
  // la fuente de caracteres o los colores de fondo o primer plano
  // del componente, y cuando se cambia a un estado que no sea
  // ENABLED.
  // Los valores de retorno para el estado habilitado se instalan
  // sobre el componente, por lo que es importante devolver objetos
  // de tipo UIResource.
  protected Color getColorForState( SynthContext context,
    ColorType type ) {
    if( type == ColorType.FOREGROUND ||
      type == ColorType.TEXT_FOREGROUND ) {
      return( COLOR_PRIMER_PLANO );
      }
    return( COLOR_FONDO );
    }

  protected Font getFontForState( SynthContext context ) {
    return( FONT_POR_DEFECTO );
    }
  }
```

Observe el lector que no se ha proporcionado un **Painter**, por lo que no habrá ni bordes ni fondos. El estilo para los campos de texto debe disponer de un **Painter** propio y se deben personalizar los *insets*, tal como se muestra a continuación:

```
private static class EstiloCampoDeTexto extends EstiloPorDefecto {
  private static final SynthPainter PAINTER_CAMPO_TEXTO =
    new PainterCampoDeTexto();
  public Insets getInsets( SynthContext context,Insets insets ) {
    if( insets == null )
      insets = new Insets( 10,10,10,10 );
    else
      insets.set( 10,10,10,10 );
    return( insets );
    }
  public SynthPainter getPainter( SynthContext context ) {
    return( PAINTER_CAMPO_TEXTO );
    }
  }
```

El objeto **Painter** propio para el campo de texto sería una instancia de la clase que se reproduce en el siguiente código:

```
private static class PainterCampoDeTexto extends SynthPainter {
  private static final Color SOMBRA = new ColorUIResource(0x808080);
  private static final Color SOMBRA_OSCURA =
    new ColorUIResource(0x000000);
  private static final Color LUZ = new ColorUIResource(0xC0C0C0);
  private static final Color LUZ_CLARA =
    new ColorUIResource(0xFFFFFF);
  public void paintTextFieldBorder( SynthContext context,
    Graphics g,int x,int y,int w,int h ) {
    BasicGraphicsUtils.drawEtchedRect( g,x,y,w,h,
      SOMBRA,SOMBRA_OSCURA,LUZ,LUZ_CLARA );
    }
```

```
  }
```

Una vez que se han definido los estilos, es necesario crear la factoría encargada de devolver estos estilos, acción que se desarrolla en las siguientes líneas de código:

```
public SynthStyle getStyle( JComponent c,Region id ) {
  if( id == Region.TEXT_FIELD ) {
    return( ESTILO_CAMPO_TEXTO );
    }
  return( ESTILO_POR_DEFECTO );
  }
```

El código de las clases anteriores forma parte de la clase **TutorialEstiloFactory** que se utiliza en `Java1437.java` para mostrar en la pantalla tres campos de texto.

El ejemplo es muy sencillo, pero se puede hacer más complejo, por ejemplo añadiendo animación al borde cuando el campo de texto reciba el foco. Para poder conseguir este efecto visual es necesario añadir un receptor de eventos de tipo **FocusListener** a cada objeto **JTextField**. Cuando un campo de texto tenga el foco, se activa un **Timer** para repintar el borde de ese campo de texto.

Cuando se asocia un objeto **SynthStyle** a una región, se invoca el método *installDefaults()* sobre ese objeto. Este método fija el color de fondo, color del primer plano, fuente de caracteres y borde. De igual modo, cuando un estilo deja de estar asociado a una región, se invoca al método *uninstallDefaults()* sobre el objeto **SynthStyle**. Estos dos métodos permiten que se pueda registrar una funcionalidad específica sobre cada componente. Por ejemplo, en el ejemplo es necesario sobrescribir el método *installDefaults()* en la clase **EstiloCampoDeTexto** y luego registrar un receptor de eventos **FocusListener** sobre el objeto **JTextField**. El código siguiente muestra estas acciones.

```
public void installDefaults( SynthContext context ) {
  // Invocamos a la superclase para instalar nuestros defaults
  super.installDefaults( context );
  // Añadimos el receptor de eventos de foco
  JTextField tf = (JTextField)context.getComponent();
  tf.addFocusListener( this );
  }
```

Cuando el foco pasa al campo de texto, se arranca el **Timer** que activa la animación del borde invocando al método *repaint()* del componente. El código que controla el evento es el que reproducen las siguientes líneas.

```
public void focusGained( FocusEvent e ) {
  start( (JTextField)e.getSource() );
  }
public void focusLost( FocusEvent e ) {
  stop( (JTextField)e.getSource() );
  }
```

El código encargado del cambio de color del borde del campo de texto que tiene el foco es muy simple, corresponde al método *paintTextFieldBorder()* de la clase anidada **PainterCampoDeTexto**, incluida en la clase **TutorialEstiloFactory2** que se utiliza en la aplicación `Java1438.java` para mostrar en la ventana campos de texto con el efecto de borde cambiante.

Synth dispone de muchas otras características, pero con las vistas en este apartado, el lector ya habrá tomado conciencia del uso y potencia que ofrece. Debe referirse a la documentación del API que proporciona *Sun Microsystems* para describir toda la potencia que Synth proporciona y que se escapa al alcance de este Tutorial.

CAPÍTULO 15

GRÁFICOS

Toda la parte gráfica de Java se basa en la clase **Graphics**, por lo que este capítulo se dedicará casi exclusivamente al uso de esta clase para manejar formas, fuentes de caracteres e imágenes sobre la pantalla. Y, aunque para el uso normal de gráficos no sea necesario, por lo que el lector puede obviar esa parte, se ha incluido un amplio repaso al *sistema de repintado* de Java, de forma que si el lector tiene curiosidad por saber exactamente cómo funciona el sistema de *repintado* y poder crear sus propios mecanismos, tenga suficiente información como para acometer esa tarea, no sin esfuerzo, pero al menos, sí con los conocimientos suficientes.

EL SISTEMA DE COORDENADAS

Cada uno de los componentes de Java tiene su propio sistema de coordenadas, que va desde la posición (0,0) hasta la posición determinada por su anchura total y altura total, menos una unidad; la unidad de medida son *píxeles* de pantalla. Como se puede apreciar en la figura 15.1, la esquina superior izquierda del componente es la posición que coincide con las coordenadas (0,0). La coordenada en el eje de abscisas se incrementa hacia la derecha y en las ordenadas hacia abajo.

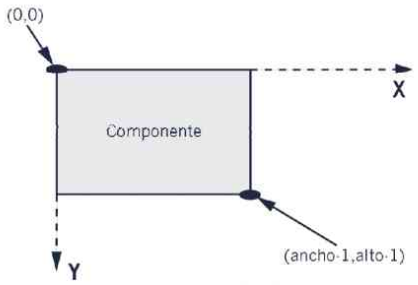


Figura 15.1

A la hora de pintar un componente, se debe tener en cuenta además del tamaño de ese componente, el tamaño del borde del componente, si lo tuviera. Por ejemplo, un borde que ocupa un píxel alrededor de un componente, haría que la coordenada de la esquina superior izquierda pasase de ser (0,0) a ser (1,1), y reduciendo además la anchura y altura totales del componente en dos píxeles, uno por cada lado.

Las dimensiones de un componente se pueden conocer a través de sus métodos *getWidth()* y *getHeight()*. El método *getInsets()*, permite conocer el tamaño del borde. El siguiente trozo de código muestra la forma de conocer exactamente la anchura y altura de la zona disponible en un componente para poder pintar en su interior.

```
public void paintComponent( Graphics g ) {
  ...
  Insets borde = getInsets();
  int anchura = getWidth() - borde.left - borde.right;
  int altura = getHeight() - borde.top - borde.bottom;
  ...
  /* La primera posición disponible para pintar sería (x,y), donde
     x es al menos borde.left, e y es al menos borde.height
  */
  }
```

El ejemplo `Java1501.java` permitirá al lector familiarizarse con el sistema de coordenadas. La figura 15.2 representa la ventana que aparece en pantalla cuando se ejecuta el ejemplo como aplicación independiente. Cuando se pulsa el botón del ratón con el cursor dentro de la zona enmarcada, aparecerá un punto en esa posición, y en la parte inferior de la ventana se indicarán las coordenadas correspondientes a la posición en que apareció el punto. Si se pulsa exactamente sobre el marco de la zona, las coordenadas sí aparecen, pero el punto no se visualizará debido a que el borde del componente se redibuja posteriormente a las acciones de repintado generadas por el usuario. Si se quiere eliminar este efecto, será suficiente con colocar el componente en otro **JPanel** que lo contenga.



Figura 15.2

El programa es muy sencillo, y semejante a la clase **Java1203** presentada al hablar de los eventos en Java. Se implementan dos componentes, el mayor es la zona rectangular con borde que es el origen de los eventos de ratón que se van a capturar y donde se muestra gráficamente a través de un punto y sus coordenadas el sitio exacto en que se encontraba el cursor del ratón en el momento de pulsar el botón, y el otro

componente es la etiqueta que se presenta en la parte inferior donde se vuelve a mostrar la coordenada en que se ha pulsado el ratón.

PINTADO EN AWT

Antes de entrar en más profundidades, tanto en la revisión de gráficos en *AWT*, en *Swing* o en el *API 2D*, es necesario entender exactamente cómo funciona el mecanismo de repintado que utiliza Java para poder comprender cómo se producen y tratan los eventos que redibujan el contenido de los componentes. En AWT hay dos mecanismos por los que se producen las operaciones de repintado, dependiendo de quién sea el ordenante de ese repintado, el *sistema* o la *aplicación*.

En el caso de que sea el *sistema* el que ordena el repintado, él es quien indica a un componente que debe regenerar su contenido, y las razones más normales por las que lo hace son:

- El componente se hace visible por primera vez en la pantalla.
- El componente ha cambiado de tamaño.
- El componente se ha deteriorado y necesita ser regenerado; por ejemplo, había un solapamiento de componentes y estaba medio cubierto por otro componente que ahora se ha movido, con lo cual hay una zona del componente que estaba oculta y ahora debe mostrarse.

En el caso de que el repintado sea ordenado por la *aplicación*, es el propio componente el que decide la necesidad de la actualización; normalmente, debido a algún cambio en su estado interno, por ejemplo, un botón detecta que el ratón ha sido pulsado sobre él y determina que tiene que cambiar su imagen de botón normal a botón *pulsado*.

El método *paint()*

Independientemente de dónde proceda el origen de la petición de repintado, el AWT utiliza un mecanismo de *callback* para ese repintado, que es igual tanto para los componentes normales como para los componentes ligeros, *lightweight*. Esto significa que un programa debe colocar su código de repintado dentro de un método sobrescrito, y será Java quien se encargará de invocarlo en el momento del repintado. Este método se encuentra en **java.awt.Component**.

```
public void paint( Graphics g )
```

Cuando el AWT llama a este método, el objeto **Graphics** que se pasa como parámetro está preconfigurado con el estado correspondiente al pintado en ese componente concreto:

- El *color* del objeto **Graphics** se fija a la propiedad **foreground** del componente.
- La *fuente de caracteres* se fija a la propiedad **font** del componente.
- La *traslación* también se determina, teniendo en cuenta que la coordenada (0,0) representa la esquina superior-izquierda del componente.
- El *rectángulo de recorte*, o *clipping*, determina el área del componente que es necesario repintar.

El programa debe utilizar este objeto **Graphics**, o uno derivado de él, para redibujar la salida. Es el encargado de cambiar el estado del objeto **Graphics** como sea necesario. Por ejemplo, el código siguiente muestra la forma de crear un círculo relleno dentro de los límites del componente.

```
public void paint( Graphics g ) {
  // Se calcula el tamaño de la zona
  Dimension tam = getSize();
  // Ahora el diámetro
  int d = Math.min( tam.width,tam.height );
  int x = (tam.width - d) / 2;
  int y = (tam.height - d) / 2;
  // Se pinta el círculo, fijando el color al de frente
  g.fillOval( x,y,d,d );
  // Se pinta la circunferencia de borde, en color negro
  g.setColor( Color.black );
  g.drawOval( x,y,d,d );
  }
```

El ejemplo `Java1502.java` es un poco más entretenido que el código anterior, pero en esencia es igual, y proporciona al lector un buen ejemplo de cómo se utiliza el método *paint()* en un programa AWT. La imagen que genera el ejemplo es la que reproduce la figura 15.3.

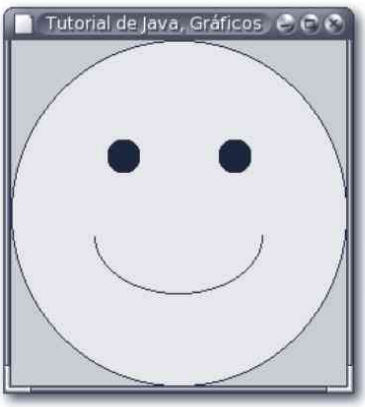


Figura 15.3

En general, se debe evitar en todos los programas escribir código que dibuje algo fuera del ámbito del método *paint()*. El porqué se debe a que ese código puede ser invocado a veces en un momento inadecuado; por ejemplo, antes de hacer visible al

componente o tener acceso a un objeto **Graphics** válido. No es nada recomendable que los programas invoquen directamente al método *paint()*.

Para habilitar el pintado desde las aplicaciones, AWT proporciona los siguientes métodos, que pueden ser llamados en cualquier momento para solicitar un repintado del componente afectado; son métodos de **Component**, por supuesto.

```
public void repaint()
public void repaint( int x,int y,int width,int height )
public void repaint( long tm )
public void repaint( long tm,int x,int y,int width,int height )
```

En las líneas de código siguientes se muestra un ejemplo muy simple de un receptor de eventos de ratón que utiliza el método *repaint()* para actualizar un componente de tipo botón en el momento en que el ratón sea pulsado y soltado.

```
MouseListener procesoRaton = new MouseAdapter() {
  public void mousePressed( MouseEvent evt ) {
    MiBoton b = (MiBoton)evt.getSource();
    b.setSelected( true );
    b.repaint();
    }
  public void mouseReleased( MouseEvent evt ) {
    MiBoton b = (MiBoton)evt.getSource();
    b.setSelected( false );
    b.repaint();
    }
  };
```

Los componentes que realicen operaciones complejas deberían invocar al método *repaint()* con argumentos definiendo solamente la región que necesita actualización. Un error muy común es llamar a *repaint()* sin ningún parámetro, lo que hace que se repinte el componente completo, provocando que se realicen repintados que no son necesarios, sin lugar a dudas.

El lector se estará preguntando el porqué de la distinción entre el pintado desde el sistema y el pintado desde la aplicación. En los componentes normales, o *heavyweight*, los dos tipos de origen del repintado se producen de dos formas distintas, dependiendo de que la operación sea ordenada por el sistema o por la aplicación.

En el caso de que sea el sistema el que ordena el repintado:

- El AWT determina si el componente necesita ser repintado completamente o solamente parte de él.
- El AWT lanza el evento para invocar al método *paint()* sobre el componente.

Si quien ordena el repintado es la aplicación, lo que sucede es lo siguiente:

- El programa determina si parte o todo el componente debe ser repintado, en respuesta a cambios en algún estado interno.

- El programa invoca al método *repaint()* sobre el componente, el cual lanza una petición al AWT indicándole que ese componente necesita ser repintado.
- El AWT lanza el evento para invocar al método *update()* sobre el componente. En el caso se que se produzcan múltiples llamadas al método *repaint()* antes de que se inicie el repintado, todas estas llamadas se resumen en una; el algoritmo que determina cuándo hay múltiples llamadas que pueden ser comprimidas en una sola es dependiente de la implementación. En el caso de que el colapso de todas estas llamadas se produzca, el rectángulo resultante que se actualizará será la unión de los rectángulos indicados en cada una de las peticiones de repintado.
- Si el componente no sobrecarga el método *update()*, la implementación por defecto de *update()* limpiará el fondo del componente (si no se trata de un componente *lightweight*) y luego hace una llamada a *paint()*.

El problema está en que al tratarse del mismo resultado final, una llamada al método *paint()*, muchos programadores no entienden el propósito de tener un método *update()* separado. Aunque esto es verdad en la implementación por defecto de *update()*, porque se limita a dar un rodeo para llamar a *paint()*; aquí hay una forma de que un programa pueda controlar los repintados provocados por la aplicación de forma distinta, si se quiere. Un programa debe asumir que una llamada a *paint()* implica que el área definida por el rectángulo que se va a repintar está dañada y debe ser repintada completamente; sin embargo, una llamada a *update()* no implica esto, sino que posibilita al programa a realizar un repintado incremental.

Este repintado incremental es útil, por ejemplo, si un programa desea actualizar una capa por encima de las ya existentes en el componente. El ejemplo `Java1503.java`, es un programa que se beneficia del uso de *update()* para realizar este repintado incremental. Al ejecutarlo, aparecerán dos ventanas en las cuales se van a ir pintando líneas que unirán cada uno de los puntos en que se pulse el ratón. El lector podrá comprobar la diferencia de realizar acciones en una y en otra.

A pesar de toda la discusión anterior, la mayoría de los componentes de una *Interfaz Gráfica* no necesitan este tipo de repintado incremental, por lo que muchos programas pueden ignorar completamente el método *update()* y sobrecargar directamente el método *paint()* para repintar el componente a su estado actual. Es decir, en muchas implementaciones de componentes, tanto el repintado originado por la aplicación como el originado por el sistema serán, esencialmente, equivalentes.

Componentes Lightweight

Desde el punto de vista de la aplicación, el API de pintado es básicamente el mismo para los componentes ligeros, *lightweight*, que para los componentes normales, *heavyweight*; esto es, se puede sobrescribir el método *paint()* e invocar al método *repaint()* para provocar actualizaciones. Sin embargo, debido a que todo el control de

los componentes ligeros está escrito en código Java, hay leves diferencias en la implementación del mecanismo para estos componentes.

Para que exista un componente ligero, es imprescindible que exista un componente normal superior en su árbol jerárquico. Cuando este componente normal decide un repintado, debe trasladar esa petición a todos sus componentes ligeros descendientes. Esto es controlado por el método *paint()* de la clase **Container**, el cual llama al método *paint()* de todo lo que haya visible, de forma que los hijos que interseccionan con el rectángulo de repintado serán refrescados. Por lo tanto, es crítico para todas las subclases de **Container**, ligeras o normales, el sobrescribir el método *paint()* para que haga lo siguiente

```
public class MiContenedor extends Container {
  public void paint( Graphics g ) {
    // repinto mi contenido primero...
    // luego, aseguramos que se repintan los hijos ligeros
    super.paint( g );
    }
  }
```

Si la llamada a *super.paint()* no se coloca, entonces los descendientes ligeros del contenedor no se mostrarán. Hay que destacar que la implementación por defecto de *Container.update()* no utiliza recursión para invocar a *update()* o *paint()* sobre descendientes ligeros; es decir, que si cualquier subclase normal de **Container** utiliza *update()* para hacer repintado incremental, debe asegurarse de que sus descendientes ligeros son llamados recursivamente para su repintado, en caso necesario. Afortunadamente, hay pocos contenedores normales que necesiten este tipo de repintado incremental, por lo que la mayoría de los programas no se verán afectados.

El código de los componentes ligeros que implementa el comportamiento de las ventanas (mostrarlas, ocultarlas, moverlas, cambiarles el tamaño, etc.) está completamente escrito en Java. Evidentemente, dentro de la implementación Java de estas funciones, el AWT debe indicar explícitamente a varios componentes ligeros que se pinten; esencialmente por repintados ordenados por el sistema. Sin embargo, en el control del comportamiento de los componentes ligeros se utiliza *repaint()* para indicar a esos componentes que se redibujen, lo cual como se ha indicado antes, hace una llamada a *update()* en lugar de llamar a *paint()* directamente. Por lo tanto, para los componentes ligeros, el redibujado ordenado por el sistema puede seguir dos caminos:

1. El redibujado solicitado por el sistema nativo; por ejemplo, ordenado por el componente normal antecesor de los componentes ligeros, provocará una llamada directa a *paint()*.
2. El redibujado originado por el control de los componentes ligeros; por ejemplo, el cambio de tamaño de un componente ligero, realiza una llamada a *update()*, la cual será reenviada por defecto a *paint()*.

Esto significa que en el caso de los componentes ligeros no hay una distinción real entre *update()* y *paint()*, lo que se traduce en que la técnica del redibujado incremental no debe utilizarse en los componentes ligeros.

Como los componentes ligeros reflejan en pantalla el estado real de un antecesor normal, también soportan la *transparencia*. Esto funciona porque los componentes ligeros son dibujados de atrás hacia delante, por lo que si un componente ligero deja alguno o todos los píxeles asociados a él sin pintar, el componente normal que se encuentra por encima en el árbol jerárquico se encargará de *mostrarlo*. Ésta es la razón por la que la implementación por defecto de *update()* no limpia el fondo de un componente, si se trata de un componente ligero, o *lightweight*. En el ejemplo `Java1504.java` se muestra esta característica de la transparencia, en lo que a los componentes ligeros se refiere.

Mientras que el AWT intenta hacer que el proceso de pintado, o *renderizado*, de los componentes sea lo más eficiente posible, una implementación de *paint()* para un componente específico puede tener un fuerte impacto sobre el rendimiento global. Hay dos áreas clave que pueden verse afectadas en este proceso.

- El uso de una región de repintado estrecha y ceñida a lo que hay que repintar.
- El uso del conocimiento interno que tiene el controlador de posicionamiento de los componentes (*layout*), para repintar solamente la zona que ocupan los componentes que se van a repintar (sólo en el caso de componentes ligeros).

Si se trata de un componente simple, por ejemplo, un botón, entonces no es necesario realizar ningún esfuerzo para pintar solamente la porción de intersección con el rectángulo de *clipping*; es preferible pintar el botón entero y no modificar el rectángulo de *clipping*. Sin embargo, si se trata de un gráfico complejo, como por ejemplo un campo de texto, aquí sí que es crítico que el código utilice la información de *clipping* para reducir lo más posible la zona a repintar.

Además, si se está escribiendo un contenedor complejo de componentes ligeros, que maneje muchos componentes, donde el componente y/o su controlador de posicionamiento tenga información acerca de la posición de esos componentes; en este caso, es más simple utilizar el conocimiento del *layout*, que ponerse a determinar cuáles de los hijos deben repintarse. La implementación por defecto de *Container.paint()* simplemente navega a través de sus hijos secuencialmente para comprobar su visibilidad e intersección, una operación que puede resultar innecesariamente ineficiente con ciertos *layouts*. Por ejemplo, si un contenedor coloca sus componentes en una malla de 100x100, la información de esa malla puede ser utilizada para determinar más rápidamente cuáles de los 10.000 componentes interseccionan con el rectángulo de repintado y necesitan volver a pintarse.

PINTADO EN SWING

Swing sigue el modelo básico de pintado del AWT y lo extiende para maximizar su rendimiento y mejorar su extensibilidad. Al igual que el AWT, Swing soporta las llamadas de pintado y el uso de *repaint()* para provocar actualizaciones. Adicionalmente, *Swing* proporciona soporte para el uso del *doble-buffer*, así como cambios para soportar algunas estructuras adicionales de Swing; como los bordes, por ejemplo. Finalmente, Swing proporciona el API de **RepaintManager** para aquellos programas que quieren desarrollar su propio mecanismo de pintado.

Soporte de Doble Buffer

Una de las aportaciones más notables de Swing es que ha incluido el soporte para operaciones de *doble-buffering* dentro del *toolkit*, que se activa a través de la propiedad `doubleBuffered` de **javax.swing.JComponent**.

```
public boolean isDoubleBuffered()
public void setDoubleBuffered( boolean o )
```

El mecanismo de *doble-buffer* de Swing utiliza un solo buffer oculto de pantalla por cada jerarquía de contenedores (normalmente, por cada ventana *top-level*) en la que se haya activado el *doble-buffering*. Aunque esta propiedad puede ser habilitada a nivel del componente, el resultado de fijarla para un determinado contenedor hace que todos los componentes ligeros que contenga ese contenedor sean pintados en el buffer oculto, independientemente del valor de su propiedad `doubleBuffered` individual.

Por defecto, esta propiedad está fijada a `true` para todos los componentes Swing; pero en realidad, lo que se hace es fijarla sobre **JRootPane**, lo cual hace que se active el doble buffer para todo aquello que esté por debajo de la ventana *top-level*. La mayoría de los programas no necesitan hacer cosas especiales con el doble buffer, nada que no vaya más allá de su activación o desactivación. Swing asegura que el objeto **Graphics** adecuado (el objeto **Graphics** oculto en el caso de doble buffer, o el objeto **Graphics** normal en cualquier otro), sea pasado al método *paint()* del componente; así que todo lo que tienen que hacer los componentes es pintar con él.

Propiedades adicionales de pintado

Swing introduce un conjunto de propiedades nuevas en **JComponent** para aumentar la eficiencia de los algoritmos internos de repintado. Estas propiedades fueron introducidas para ser aplicadas sobre las dos operaciones que pueden hacer que el repintado de los componentes ligeros sea una operación muy costosa en recursos.

- *Transparencia*. Si un componente ligero es repintado, es posible que ese componente no pinte todos los bits asociados a él si es parcial o totalmente transparente; esto significa que siempre que sea repintado, cualquier cosa que haya debajo de él ha de repintarse en primer lugar. Esto requiere que el

sistema navegue a través de la jerarquía de componentes para encontrar el antecesor normal más profundo desde el cual comenzar las operaciones de repintado de atrás hacia delante.

- *Componentes solapados*. Si un componente ligero es pintado, es posible que haya algún otro componente ligero que lo solape parcialmente; esto significa que siempre que el componente ligero original sea repintado, cualquier componente que solape a ese componente (donde el rectángulo de *clipping* interseccione con la zona de solape), el componente que está sobre el componente original también debe ser parcialmente repintado. Esto requiere que el sistema navegue a través de la jerarquía de componentes, comprobando los componentes solapados en cada operación de repintado.

Opacidad

Para mejorar el rendimiento en el caso de componentes opacos, Swing añade la propiedad de opacidad a **javax.swing.JComponent**.

```
public boolean isOpaque()
public void setOpaque( boolean o )
```

Los valores que puede tomar esta propiedad son:

- `true`: El componente repintará todos los bits contenidos dentro de sus límites rectangulares.
- `false`: El componente no garantiza que se vayan a repintar todos los bits que se encuentren dentro de sus límites rectangulares.

Esta propiedad permite al sistema de pintado de Swing detectar cuándo una petición de repintado sobre un componente necesitará repintados adicionales de sus antecesores. El valor por defecto para cada componente estándar de Swing es fijado por el objeto actual *Look&Feel*; suele ser `true` para la mayoría de los componentes.

Uno de los errores más comunes en la implementación de componentes es hacer que su propiedad de opacidad esté a `true`, ya que no van a repintar completamente la zona que ocupan, de forma que ocasionalmente la pantalla se llenará de porquería en algunos sitios. Cuando se diseña un componente, hay que poner especial atención al manejo de esta propiedad, tanto para asegurar que la transparencia se utilice prudentemente, porque es una operación costosa, como para ser respetuosos con el sistema de pintado.

El significado de esta propiedad, a menudo es malinterpretado. A veces, se toma su significado como, "*hacer transparente el fondo del componente*"; sin embargo, ésta no es la interpretación estricta que Swing hace de la opacidad. Algunos componentes, como los botones de pulsación, por ejemplo, pueden fijar esta propiedad a `false` para proporcionar al componente una imagen no-rectangular, o dejar sitio alrededor del

componente para otras indicaciones, como puede ser el indicador de foco. En estos casos, el componente no es opaco, aunque la mayor parte del fondo esté rellena.

Si un componente utiliza la característica de opacidad para definir cómo se aplica la transparencia a la parte visual del componente, entonces hay que documentar perfectamente esta circunstancia; puede resultar preferible en algunos componentes, para definir propiedades adicionales que controlan visualmente cómo se está aplicando la transparencia. Por ejemplo, **javax.swing.AbstractButton** proporciona la propiedad **ContentAreaFilled** para este propósito.

Otra circunstancia que hay que tener en cuenta es cómo se relaciona la opacidad con la propiedad de asignación de borde a un componente Swing. El área pintada por un objeto **Border** sobre un componente se considera como parte de la geometría del componente. Esto significa que si un componente es opaco, sigue siendo el responsable de rellenar la zona ocupada por el borde. El borde se debe situar, por tanto, en las capas más superiores del componente opaco.

Si se quiere tener un componente que permita mostrar a los componentes que tiene debajo su borde; es decir, si el borde soporta transparencia, en cuyo caso el método *isBorderOpaque()* devuelve `false`, entonces el componente debe definirse a sí mismo como no-opaco y asegurarse de que deja el área ocupada por el borde sin pintar.

Redibujo "optimizado"

La superposición de los componentes es algo difícil de controlar. Incluso aunque ninguno de los componentes hermanos inmediatos solape al componente, siempre es posible que otro componente (puede ser un primo o un tío) pueda solaparlo. En el caso de repintar un componente individual dentro de una jerarquía compleja, puede ser necesario un poco de navegación a través del árbol jerárquico para asegurar que el repintado se produce en su orden correcto. Para reducir la navegación al mínimo, Swing añade la propiedad de sólo lectura **isOptimizedDrawingEnabled** a **javax.Swing.JComponent**.

Los valores que puede tomar son:

- `true`: El componente indica que ninguno de sus hijos inmediatos lo solapan.
- `false`: El componente no garantiza nada sobre si sus hijos inmediatos lo solapan o no.

Comprobando el valor de esta propiedad, Swing puede reducir rápidamente su búsqueda de componentes que estén solapando al componente afectado en el momento del repintado. Como esta propiedad es de sólo lectura, la única forma de que los componentes puedan cambiar su valor es creando una clase derivada y sobrescribiendo el método que devuelve ese valor. Todos los componentes estándar de Swing

devuelven esta propiedad a `true`, excepto **JLayeredPane**, **JDesktopPane** y **JViewPort**.

Los métodos *paint()*

Las reglas que se aplican a los componentes ligeros del AWT también se aplican a los componentes Swing; por ejemplo, *paint()* se llama cuando es necesario el repintado, excepto que en el caso de Swing, la llamada a *paint()* se convierte en la llamada a tres métodos separados, que siempre se invocan en el mismo orden:

```
protected void paintComponent( Graphics g )
protected void paintBorder( Graphics g )
protected void paintChildren( Graphics g )
```

Los programas que utilicen Swing deberían sobrescribir el método *paintComponent()*, en lugar de sobrescribir el método *paint()*. Aunque el API lo permita, no hay razón habitualmente, para sobrescribir los métodos *paintBorder()* o *paintChildren()*, y si el lector lo hace, debe asegurarse de saber lo que está haciendo.

Todo esto hace que sea más fácil para los programas sobrescribir solamente la porción de pintado que necesitan para extenderse. Por ejemplo, así se resuelve el problema mencionado al hablar del AWT donde el fallo de no invocar a *super.paint()* hacía que no apareciesen los componentes ligeros hijos. En el ejemplo `Java1505.java`, se muestra el uso del método *paintComponent()* de Swing; la salida gráfica del programa presenta una diana.

Repintado e interfaz de usuario

Muchos de los componentes estándar de Swing tienen su apariencia, o *look-and-feel*, implementado a través de objetos *look-and-feel* separados (llamados **UI delegates**) para soportar la posibilidad de cambio de apariencia de Swing. Esto significa que muchos, o todos los procesos de repintado de los componentes estándar se delegan en la interfaz de usuario, y esto se produce de la siguiente forma:

1. *paint()* invoca a *paintComponent()*.
2. Si la propiedad **ui** no es nula, *paintComponent()* invoca a *ui.update()*.
3. Si la propiedad de opacidad del componente es `true`, *ui.update()* rellena el fondo del componente con el color de fondo e invoca a *ui.paint()*.
4. *ui.paint()* repinta el contenido del componente.

Es decir, que las clases derivadas de componentes Swing que tengan un **UI delegate**, en contraposición a las clases derivadas directamente de **JComponent**, deberían invocar a *super.paintComponent()* dentro de su método sobrescrito *paintComponent()*:

```
public class MiPanel extends JPanel {
  protected void paintComponent( Graphics g ) {
    // Se deja primero trabajar a UI delegate
    // (incluyendo el relleno del fondo, si somos opacos)
    super.paintComponent(g);
    // pintamos el resto de nuestro componente
    }
  }
```

Si por alguna razón la extensión del componente no quiere permitir que **UI delegate** repinte; por ejemplo, porque se esté reemplazando completamente la parte visual de un componente, se puede saltar la llamada a *super.paintComponent()*, pero no puede evitar la responsabilidad de rellenar su propio fondo si la propiedad de opacidad está a `true`, tal como ya se ha visto.

Proceso de repintado

El proceso de repintado en Swing es ligeramente diferente a la forma en que se hace en el AWT, aunque el resultado final para el programador de aplicaciones sea esencialmente el mismo: hay que invocar a *paint()*. Swing proporciona el API **RepaintManager** para optimizar el rendimiento del repintado. En Swing, el proceso de repintado se realiza a través de dos caminos.

A. La petición de pintado tiene su origen en el primer antecesor normal, habitualmente **JFrame**, **JDialog**, **JWindow** o **JApplet**:

1. La tarea encargada de despachar eventos invoca el método *paint()* de un ascendiente.
2. La implementación por defecto del método *Container.paint()* llama recursivamente a *paint()* sobre todos los descendientes ligeros.
3. Cuando se alcanza el primer componente Swing, la implementación por defecto del método *JComponent.paint()* hace lo siguiente:
   a. Si la propiedad **doubleBuffered** del componente está a `true` y el uso del doble buffer está habilitado sobre el **RepaintManager** del componente, el objeto **Graphics** se convertirá en el gráfico correspondiente (en el buffer oculto).
   b. Se invoca a *paintComponent()* (en el gráfico oculto si es con doble buffer).
   c. Se invoca a *paintBorder()* (en el gráfico oculto si es con doble buffer).
   d. Se invoca a *paintChildren()* (en el gráfico oculto si es con doble buffer), que utiliza el rectángulo de dibujo, las propiedades **opaque** y **optimizedDrawingEnabled** para determinar exactamente qué descendientes deben invocar recursivamente al método *paint()*.
   e. Si la propiedad **doubleBuffered** del componente está a `true` y el uso del doble buffer está habilitado sobre el **RepaintManager** del componente, se

copiará la imagen del buffer oculto a la original utilizando el objeto **Graphics** que se está mostrando en pantalla.

Los pasos #a y #e en el caso de *JComponent.paint()* se saltan en las llamadas recursivas a *paint()*; a partir de *paintChildren()*, descrito en el paso #d, porque todos los componentes ligeros dentro de la jerarquía de una ventana Swing comparten la misma imagen oculta para el uso del doble buffer.

B. La petición de pintado tiene su origen en una llamada a *repaint()* sobre una extensión de **javax.swing.JComponent**:

1. El método *JComponent.repaint()* registra una petición asíncrona de repintado sobre el **RepaintManager** del componente, el cual utiliza el método *invokeLater()* para encolar un objeto **Runnable** y procesar posteriormente esa petición sobre la tarea (*thread*) encargada de despachar los eventos.
2. El objeto **Runnable** ejecuta esa petición y hace que el **RepaintManager** del componente invoque al método *paintInmediately()* sobre del componente, que hace lo siguiente:
   a. Utiliza el rectángulo de dibujo y las propiedades **opaque** y **optimizeDrawingEnabled** para determinar cuál es el componente *raíz* desde el que hay que comenzar la operación de pintado (para tener en cuenta la transparencia y los potenciales componentes que se solapen).
   b. Si la propiedad **doubleBuffered** del componente raíz es `true`, y el uso del doble buffer está habilitado sobre el **RepaintManager** de ese componente, el objeto **Graphics** se convertirá en el gráfico correspondiente en el buffer oculto.
   c. Se invoca el método *paint()* sobre el componente raíz (que a su vez ejecuta los pasos #b a #d del caso #A), haciendo que todo lo que se encuentre en la intersección con el rectángulo ocupado por el componente raíz sea repintado.
   d. Si la propiedad **doubleBuffered** del componente raíz es `true` y el uso del doble buffer está habilitado sobre el **RepaintManager** de ese componente raíz, se copia la imagen del buffer oculto a la original utilizando el objeto **Graphics** que se está mostrando en pantalla.

Si se producen múltiples llamadas a *repaint()* sobre un componente o sobre cualquiera de sus antecesores Swing antes de que la petición de repintado sea procesada, esas múltiples peticiones se agruparán en una sola llamada al método *paintInmediately()* sobre el componente Swing más alto del árbol jerárquico de los que se haya invocado a su método *repaint()*. Por ejemplo, si un **JTabbedPane** contiene un objeto **JTable** y ambos realizan una llamada a *repaint()* antes de que cualquier otra petición de repintado sea procesada, el resultado será una sola llamada al método *paintInmediately()* sobre del objeto **JTabbedPane**, que hará que se llame a *paint()* sobre los dos componentes.

Esto significa que en el caso de los componentes Swing, el método *update()* no se invoca nunca. Aunque *repaint()* desemboque en una llamada al método *paintInmediately()*, esto no se considera como una llamada de tipo *callback*, y el código del cliente no se debería colocar dentro del ámbito *de paintInmediately()*. De hecho, no hay ninguna razón para sobrescribir el método *paintInmediately()*, al menos, ninguna razón normal.

Repintado síncrono

Como se ha visto antes, el método *paintInmediately()* actúa como punto de entrada para indicar a un componente Swing que se repinte a sí mismo, asegurando que todo el proceso de repintado se produce de forma adecuada. Este método también se podría utilizar para hacer peticiones síncronas de pintado, en el caso de que sea necesario, para aquellos componentes que necesiten mantener su apariencia actualizada en tiempo real; por ejemplo, un objeto **JScrollPane** durante una operación de desplazamiento.

Las aplicaciones no deberían invocar directamente a este método, a no ser que haya buenas razones para necesitar este pintado en tiempo real; porque el pintado asíncrono a través del método *repaint()* realizará múltiples peticiones de solape que se agruparán de forma muy eficiente en una sola, mientras que la llamada directa a *paintInmediately()* no lo hará. Además, la norma para la invocación de este método es que debe ser invocado desde la tarea (*thread*) que despacha los eventos; es decir, no es un API diseñado para realizar el repintado a través de varias tareas.

La clase RepaintManager

El propósito de la clase **RepaintManager** de Swing es maximizar la eficiencia del proceso de repintado sobre la jerarquía de componentes Swing y, también, implementar el mecanismo de *revalidación*. Implementa el proceso de pintado interceptando todas las peticiones de repintado sobre componentes Swing (porque ya no van a ser procesados por el AWT) y manteniendo su propio estado sobre sus necesidades de actualización (conocido como *regiones sucias*, "dirty regions"). Finalmente, llama al método *invokeLater()* para procesar las peticiones pendientes de repintado de la tarea que despacha los eventos, tal como se ha descrito en el caso **B** del Proceso de Repintado.

En la mayoría de los programas, el objeto **RepaintManager** puede considerarse como una parte interna de Swing y ser totalmente ignorado. Sin embargo, su API proporciona a los programas la posibilidad de obtener el control de ciertos aspectos del repintado.

EL OBJETO REPAINTMANAGER ACTUAL

El objeto **RepaintManager** está diseñado para conectarse dinámicamente, aunque por defecto solamente hay una instancia de él. Los siguientes métodos estáticos permiten a los programas fijar y obtener la instancia de **RepaintManager** actual.

```
public static RepaintManager currentManager( Component c )
public static RepaintManager currentManager( JComponent c )
public static void setCurrentManager( RepaintManager aRepaintManager )
```

CAMBIO DEL OBJETO REPAINTMANAGER ACTUAL

Un programa que extendiese y reemplazase el objeto **RepaintManager** tendría que hacer una llamada de la forma siguiente:

```
RepaintManager.setCurrentManager( new miRepaintManager() );
```

En el ejemplo `Java1506.java` se muestra cómo hacerlo, en este caso de forma muy simple, ya que lo único que se hace es imprimir información acerca de lo que se está repintando.

Una razón más interesante para cambiar el objeto **RepaintManager** sería alterar la forma en que se procesan las peticiones de pintado. Normalmente, el estado interno utilizado por la implementación que maneja las *regiones sucias* es un paquete privado y no está accesible a las subclases. Sin embargo, las aplicaciones pueden implementar sus propios mecanismos de tratamiento de las *regiones sucias* y agrupar esas peticiones sobrescribiendo los siguientes métodos:

```
public synchronized void addDirtyRegion( JComponent c,
  int x,int y,int w, int h)
public Rectangle getDirtyRegion( JComponent aComponent )
public void markCompletelyDirty( JComponent aComponent )
public void markCompletelyClean( JComponent aComponent )
```

El método *addDirtyRegion()* es uno de los que se invocan cuando se llama a *repaint()* sobre un componente Swing, de forma que se puedan ir agrupando todas las peticiones de repintado. Si se sobrescribe este método en un programa (y no se llama a *super.addDirtyRegion()*), entonces es responsabilidad del programador la utilización de *invokeLater()* para colocar un objeto **Runnable** en la cola de eventos, que invocará a *paintInmediately()* sobre el componente. Es recomendable no experimentar demasiado con esto, so pena de sufrir un ataque de ansiedad, porque... a saber lo que funciona y lo que no.

Control global sobre el doble buffer

El objeto **RepaintManager** proporciona un API para poder habilitar y deshabilitar globalmente el uso del doble buffer.

```
public void setDoubleBufferingEnabled( boolean aFlag )
public boolean isDoubleBufferingEnabled()
```

Esta propiedad es comprobada dentro de cada **JComponent** durante el proceso de la operación de pintado, para determinar si se utiliza el buffer oculto o no. Por defecto, esta propiedad está fijada a `true`, pero las aplicaciones pueden deshabilitar globalmente el uso del doble buffer para todos los componentes Swing haciendo lo siguiente:

```
RepaintManager.currentManager( miComponente ).
  setDoubleBufferingEnabled( false );
```

Aunque como la implementación por defecto de Swing instancia un solo objeto **RepaintManager**, el argumento `miComponente` es irrelevante.

LA CLASE GRAPHICS

La clase **Graphics** es la clase base abstracta que proporciona toda, o al menos la mayoría, de la funcionalidad para poder pintar, tanto sobre componentes como sobre imágenes fuera de pantalla.

Un objeto **Graphics** encapsula la siguiente información que será necesaria a la hora de las operaciones básicas de pintado.

- El objeto de tipo **Component** sobre el que se pinta.
- Un origen de traslación para coordenadas de pintado y *clipping*.
- La región actual ocupada por el componente.
- El color actual.
- La fuente de caracteres actual.
- La operación lógica actual para utilizar con píxeles (XOR o Paint).
- La actual alteración de color XOR.

Por lo tanto, lo que sucede cuando se utiliza uno de los métodos de **Graphics** para dibujar una línea o una figura sobre un objeto **Component**, permite asegurar algunas afirmaciones, tales como las siguientes:

- Las coordenadas son infinitamente finas y se corresponden con los píxeles del dispositivo de salida.
- Las operaciones con dibujo lineal de una figura operan pintando un camino infinitamente pequeño entre los píxeles, con un pincel del tamaño fijado, desde abajo y a la derecha del punto inicial del camino.
- Las operaciones con relleno operan rellenando el interior del camino infinitamente pequeño anterior.
- Las operaciones con texto horizontal pintan la porción ascendente del carácter completamente sobre las coordenadas de la línea base.

El hecho de que el pincel gráfico se sitúe *abajo y a la derecha* tiene algunas implicaciones que resultan interesantes.

Si se pinta una figura que cubra un rectángulo dado, esa figura ocupará una fila extra de píxeles por la derecha y por abajo, si se compara con una figura rellena con los límites de ese rectángulo.

Otra implicación es que si se pinta una línea horizontal a lo largo de la misma coordenada **Y** como línea base de un texto, esa línea se pintará completamente por debajo del texto, excepto en donde los caracteres tengan parte *descendente*.

Cuando se pasan coordenadas a los métodos de un objeto **Graphics**, estas coordenadas están consideradas relativas al origen de traslación del objeto **Graphics** que se haya hecho antes de la invocación al método.

Un objeto **Graphics** describe un contexto gráfico. Un contexto gráfico define una zona de recorte, una zona a la que va a afectar; cualquier operación gráfica que se realice modificará solamente los píxeles que se encuentren dentro de los límites de la zona de recorte actual y el componente que fue utilizado para crear el objeto **Graphics**.

Cuando se pinta o escribe, ese dibujo o escritura se realiza en el color actual, utilizando el modo de dibujo actual y la fuente de caracteres actual.

Hay muchas otras clases, como la clase **Rectangle** o la clase **Polygon**, que utilizan como soporte las operaciones que se pueden realizar con la clase **Graphics**.

Para poder revisar esta clase, quizá una de las mejores formas sea a través de sus múltiples métodos, intentando agruparlos por funcionalidad, que es lo que se ha procurado en los siguientes párrafos.

Hay que empezar hablando del constructor de la clase **Graphics**, que no tiene argumentos; aunque **Graphics** es una clase abstracta, por lo que las aplicaciones no pueden llamar a este constructor directamente. Se puede obtener un objeto de tipo **Graphics** a partir de otro objeto **Graphics** llamando al método *getGraphics()* sobre un componente. También se puede recibir un objeto **Graphics** como parámetro cuando se van a sobrescribir los métodos *paint()* o *update()*.

Obtener un contexto gráfico

La verdad es que se han utilizado varias veces las palabras *contexto gráfico*, y no se ha proporcionado al lector una explicación concreta de lo que significan estos términos. Hay varias definiciones, para unos autores significa que la aplicación ha conseguido la habilidad para pintar o colocar imágenes sobre un componente que tiene la característica de soportar el pintado o visualización de imágenes. Otros autores

prefieren decir que cada objeto **Graphics** representa una determinada superficie de dibujo, luego ese objeto **Graphics** define un contexto gráfico para poder manipular todas las actividades gráficas sobre esa superficie. Y otros autores indican que un objeto **Graphics** es la superficie última sobre la que se pueden colocar líneas, figuras y texto, por lo cual puede recibir también el nombre de contexto gráfico al aunar información sobre la zona de dibujo, más la fuente de caracteres, color y cualquier otro factor.

Ahora que ya se sabe lo que es un contexto gráfico, hay que ver cómo se consigue crear uno. Para empezar, esto no puede hacerse instanciando directamente un objeto de tipo **Graphics**, ya que la clase **Graphics** es abstracta y no puede ser instanciada, así que hay que recurrir a formas indirectas para conseguir el contexto gráfico.

Uno de estos caminos indirectos para obtener un contexto gráfico es invocar el método *getGraphics()* sobre otro objeto, como ya se ha indicado antes. Sin embargo, este método devuelve un contexto gráfico de una imagen, es decir, que solamente funciona para objetos de tipo **Image** creados en memoria a través del método *createImage()*, de la clase **Component**. Ésta es una técnica utilizada normalmente cuando se están usando imágenes que se crean en memoria y luego se transfieren a la pantalla, es decir, se está pintando en el doble buffer.

Hay otros dos caminos para obtener un contexto gráfico y, son sorprendentemente simples, porque se hace automáticamente, y es cuando se sobrescriben los métodos *paint()* y *update()*, en los cuales Java pasa como parámetro el contexto gráfico del objeto al que pertenece el método.

Normalmente, el método *paint()* se sobrescribe cuando se quiere colocar algún tipo de material gráfico sobre la pantalla, y el método *update()* se sobrescribe en circunstancias especiales, como puede ser el caso de una animación o que se vaya a utilizar doble buffer. Lo normal es, pues, la presentación de información gráfica colocando el código encargado de ello en el método sobrescrito *paint()* y luego invocar al método *repaint()* para indicar al sistema que presente ese material en pantalla; aunque el método *paint()* también puede ser invocado por causas externas, sin control alguno por parte de la aplicación, como puede ser el redimensionamiento de la ventana en la que se está presentando la información gráfica.

Hay que tener en cuenta que el método *repaint()* pide al sistema que redibuje el componente tan pronto como sea posible, pero esto lo hará el método *update()* que se llame a continuación. No hay una relación uno a uno entre las llamadas a *repaint()* y *update()*, por lo que es posible que múltiples llamadas a *repaint()* puedan agruparse en una sola llamada a *update()*.

El método *update()* es invocado automáticamente al repintar un componente. Si el componente no es ligero, la implementación por defecto de *update()* borra el contexto gráfico rellenando el fondo con el color que se haya asignado como color de fondo, fijando de nuevo el color al color del primer plano y llamando a *paint()*. Si no se

sobrescribe *update()* para hacer una animación, se verá siempre un parpadeo en el refresco del componente a causa del borrado del fondo.

El método *paint()* es el que ofrece el sistema para poder pintar lo que se quiera sobre un determinado componente. En la clase base **Component**, este método no hace absolutamente nada. Normalmente, en el caso de applets, se sobrescribe para hacer presentar un rectángulo relleno con el color de fondo.

El ejemplo `Java1508.java` muestra la forma de utilizar el contexto gráfico de forma simple. En el programa, se invoca al método *drawString()* sobre el contexto gráfico de un objeto **Frame** para presentar las palabras "Tutorial de Java". Si se compila y ejecuta la aplicación, aparecerá el objeto **Frame** en la pantalla, y en su área cliente se presentará el texto, véase figura 15.4. Cuando se pulse el botón para cerrar la ventana, desaparecerá esa ventana y el control se devolverá al sistema operativo.



Figura 15.4

La única parte interesante del código de este ejemplo, desde el punto de vista gráfico, es el método sobrescrito *paint()* que permite pintar las palabras sobre el contexto gráfico que se le pasa como parámetro, tal como se reproduce en las siguientes líneas.

```
public void paint( Graphics g ) {
  g.drawString( "Tutorial de Java",100,40 );
  }
```

El resto del código del ejemplo no merece la pena revisarlo, porque es semejante al que aparece en otros muchos programas de ejemplo de este Tutorial.

Crear un contexto gráfico

El método *create()* crea un nuevo objeto **Graphics** que es una copia del objeto **Graphics** sobre el cual es invocado. Quizá sea más adecuado decir que se crea una segunda referencia al objeto **Graphics** sobre el cual es invocado el método, porque cualquier cosa que se pinte utilizando la nueva referencia, aparecerá sobre el contexto gráfico original cuando se repinte.

Aunque en esta sección de lo que se trata es de presentar el uso del método *create()*, lo cierto es que todo se encuentra implicado. A través de los ejemplos, el lector comprobará que se van introduciendo otra serie de conceptos, como en este caso el uso de `insets` a la hora de pintar un objeto que tiene borde, como es un objeto de tipo **Frame**, el uso de *clipping*, el uso de *setColor()* para cambiar el color con que se pinta o el uso del método *dispose()* para devolver los recursos utilizados por el contexto gráfico al sistema operativo. Por ello, el lector no debe abrumarse, ya que

aunque se introduzcan muchos conceptos, siempre se intentará dar a cada uno de ellos su auténtica relevancia, aunque sea en secciones diferentes.

El ejemplo que se presenta a continuación, `Java1509.java`, resuelve el problema asociado con `insets`, añadiendo valores para compensarlo a las coordenadas que se pasan como parámetros en las invocaciones de los métodos. El ejemplo `Java1510.java`, lo resuelve de forma distinta, superponiendo sobre el objeto **Frame** un objeto **Canvas**, que no tiene `insets`. Posteriormente, el lector podrá comprobar que el problema de `insets` también se eliminará con el uso del método *translate()*.

Pero antes de seguir, un comentario sobre el uso del método *dispose()*, porque hay autores que ofrecen explicaciones diferentes sobre la necesidad de eliminar el contexto gráfico que se crea. Quizá lo mejor sea aceptar la documentación de *Sun Microsystems*, que dice que el método *dispose()* se encarga de eliminar el contexto gráfico y devolver al sistema todos los recursos que estuviese utilizando, de forma que un objeto **Graphics** nunca puede ser llamado después de haber invocado a su método *dispose()*.

Cuando se ejecuta un programa Java, un gran número de objetos de tipo **Graphics** son creados con muy poco intervalo de tiempo; aunque al finalizar el proceso, el recolector de basura, *garbage collector*, es capaz de liberar esos mismos recursos, es preferible liberar manualmente los recursos asociados a los objetos **Graphics** llamando a *dispose()* en lugar de esperar a la finalización del proceso, que puede estar muy dilatada en el tiempo.

Los recursos asociados a los objetos **Graphics** que se pasan como argumentos a los métodos *paint()* y *update()* de los componentes, son automáticamente liberados por el sistema cuando finaliza la ejecución del método. Por eficiencia, los programadores deberían incluir una llamada a *dispose()* cuando no van a utilizar un objeto **Graphics**, solamente si ese objeto fue creado directamente desde un componente o a partir de otro objeto **Graphics**.

En el ejemplo `Java1509.java`, se ilustra el uso de *clipping*, color, rectángulos, líneas, *insets* y *dispose()*, realizando todas las acciones en el método *paint()*.

Cuando se compila y ejecuta este programa, en pantalla aparece un objeto **Frame**, con un rectángulo rojo en su interior, colocado a partir de la esquina superior-izquierda del **Frame** (ver figura 15.5). Atravesando a este rectángulo hay una línea azul, que a su vez está atravesada por una línea verde, que cruza el rectángulo en sentido contrario, de la cual solamente se visualiza la parte central, ya que es el resultado de la acción de recorte (*clipping*) que se ha aplicado a la segunda referencia del objeto.



Figura 15.5

El listado que se muestra a continuación, correspondiente al ejemplo, muestra en sus comentarios todas las acciones que se van llevando a cabo para la generación del gráfico.

```
// Se sobrecarga el método paint()
public void paint( Graphics g ) {
  // Obtenemos el tamaño de los insets para tenerlos en cuenta
  int arr = this.getInsets().top;
  int izq = this.getInsets().left;
  // Fijamos el color rojo para pintar
  g.setColor( Color.red );
  // Creamos una segunda referencia del contexto gráfico que se
  // utiliza en el método
  Graphics clpArea = g.create();
  // Usamos esta referencia para pintar un rectángulo rojo
  clpArea.drawRect( 0+izq,0+arr,250,100 );
  // Se reduce el rectángulo de clipping, de forma que solamente se
  // quede sin recortar un rectángulo más pequeño
  clpArea.clipRect( 25+izq,25+arr,200,50 );
  // Se usa el contexto para pintar una línea diagonal verde a lo
  // largo del rectángulo original, aunque por la acción de la zona
  // de clipping que se ha fijado antes, solamente estará visible la
  // parte central
  clpArea.setColor( Color.green );
  clpArea.drawLine( 0+izq,0+arr,250+izq,100+arr );
  // Liberamos los recursos usados por el contexto gráfico
  clpArea.dispose();
  // Y lo seleccionamos como candidato a la basura
  clpArea = null;
  // Ahora usamos el contexto gráfico original para pintar una
  // línea azul que atraviesa todo el rectángulo, y que ya no
  // se ve afectada por el rectángulo de recorte que se había fijado
  // en la segunda referencia del contexto que habíamos creado antes
  g.setColor( Color.blue );
  g.drawLine( 0+izq,100+arr,250+izq,0+arr );
  }
```

El ejemplo anterior presentaba el problema ocasionado por *insets*. En el ejemplo `Java1510.java`, se utiliza un objeto de tipo **Canvas** como superficie de dibujo, para evitar este problema, ya que un objeto **Canvas** no tiene bordes.

El problema que ocasiona *insets*, se debe al hecho de que algunos contenedores, como lo es un objeto **Frame**, tienen bordes, y el área ocupada por estos bordes es considerada como parte de la zona de dibujo, de modo que las coordenadas del punto

0,0 corresponden a la esquina superior-izquierda del contenedor, oculta por los bordes, en caso de que ese contenedor tenga bordes.

El método *getInsets()* proporciona la anchura en píxeles de los cuatro bordes, lo cual hace posible que se compense matemáticamente la influencia de esos bordes cuando se trabaje con valores de coordenadas.

Para poder utilizar un objeto **Canvas** como superficie de dibujo, es necesario extender la clase **Canvas** y que sea posible sobrescribir su método *paint()*. En este programa, la clase **Canvas** se extiende en la clase **MiClase**, que es donde se sobrescribe el método *paint()* para poder realizar operaciones gráficas. Se instancia un objeto de tipo **MiCanvas**, que se añade al objeto **Frame** y actúa como zona de dibujo superpuesta a este objeto **Frame**. Como el objeto **MiCanvas** no tiene bordes, el problema *insets* está eliminado.

```
// Clase para crear la zona de dibujo, que extiende a la clase Canvas
// para que sea posible sobrecargar el método paint()
class MiCanvas extends Canvas {
  // Se sobrecarga el método paint()
  public void paint( Graphics g ) {
    // Fijamos el color rojo para pintar
    g.setColor( Color.red );
    // Creamos una segunda referencia del contexto gráfico que se
    // utiliza en el método
    Graphics clpArea = g.create();
    // Usamos esta referencia para pintar un rectángulo rojo
    clpArea.drawRect( 0,0,250,100 );
    // Se reduce el rectángulo de clipping, de forma que solamente se
    // quede sin recortar un rectángulo más pequeño
    clpArea.clipRect( 25,25,200,50 );
    // Se usa el contexto para pintar una línea diagonal verde a lo
    // largo del rectángulo original, aunque por la acción de la zona
    // de clipping que se ha fijado antes, solamente estará visible la
    // parte central
    clpArea.setColor( Color.green );
    clpArea.drawLine( 0,0,250,100 );
    // Liberamos los recursos usados por el contexto gráfico
    clpArea.dispose();
    // Y lo seleccionamos como candidato a la basura
    clpArea = null;
    // Ahora usamos el contexto gráfico original para pintar una
    // línea azul que atraviesa todo el rectángulo, y que ya no
    // se ve afectada por el rectángulo de recorte que se había fijado
    // en la segunda referencia del contexto que habíamos creado antes
    g.setColor( Color.blue );
    g.drawLine( 0,100,250,0 );
    }
  }

// Clase de control del ejemplo
class Java1510 extends Frame {
  // Constructor de la clase
  public Java1510() {
    // Se crea una superficie amarilla y se utiliza para cubrir el
```

```
    // área cliente del objeto Frame
    MiCanvas zonaDibujo = new MiCanvas();
    zonaDibujo.setBackground( Color.yellow );
    this.add( zonaDibujo );
    this.setVisible( true );
    }
}
```

El listado anterior no necesita ningún comentario adicional a los que ya se encuentran insertados en el código, que el lector puede seguir para ver las acciones que se llevan a cabo.

Copiar y Borrar

Los dos métodos involucrados en estas acciones son *copyArea()* y *clearRect()*. El segundo es el más sencillo y su función es borrar el rectángulo determinado por los cuatro parámetros que se le pasan, rellenándolo con el color de fondo de la superficie de dibujo.

Los parámetros que admite son: `x`, `y`, `anchura`, `altura`. Siendo `x` e `y`, como en casi todos los métodos gráficos en Java, las coordenadas referidas a la esquina superior-izquierda de la zona rectangular que se quiere borrar, y `anchura` y `altura`, las dimensiones de esa misma zona rectangular.

El método *copyArea()* copia una zona rectangular de la superficie actual de dibujo a otra zona que se encuentra separada de la primera por una distancia especificada en los parámetros adicionales `dx` y `dy`. La zona se copia desplazando hacia abajo y a la derecha, si se quiere copiar hacia arriba o hacia la izquierda hay que proporcionar valores negativos en `dx` o en `dy`. Cualquier parte del rectángulo origen que se salga de la superficie de dibujo no se copiará.

En el ejemplo `Java1511.java` se ilustra el uso de estos métodos. El programa pinta la cadena de texto "Tutorial de Java" en la esquina superior-izquierda de un objeto **Frame**. Luego, utiliza el método *copyArea()* para hacer dos copias adicionales de esa zona, copiando un área rectangular a partir de la esquina superior-izquierda en otras dos áreas. Y después utiliza el método *clearRect()* para borrar la mayor parte de la letra "T" de la segunda copia, borrando una zona rectangular de la pantalla que contiene a la letra "T".



Figura 15.6

La figura 15.6 muestra el resultado que se observa en pantalla cuando se ejecuta el ejemplo, en donde se pueden observar las dos copias del texto, con la letra "T" casi

desaparecida en la segunda de ellas. El listado completo del ejemplo es el que se reproduce a continuación.

```
// Clase de control del ejemplo
class Java1511 extends Frame {
  // Se sobrecarga el método paint(), para presentar una cadena de
  // texto en la ventana que se abre en pantalla, sobre el contexto
  // gráfico del objeto Frame
  public void paint( Graphics g ) {
    g.drawString( "Tutorial de Java",10,40 );
    // Se copia la zona ocupada por la cadena
    g.copyArea( 0,0,100,100,100,0 );
    // Se hace otra copia
    g.copyArea( 0,0,100,100,100,40 );
    // Se borra parte de esta segunda copia, y pintamos un rectángulo
    // para indicar la zona borrada
    g.clearRect( 100,40,15,40 );
    g.drawRect( 100,40,15,40 );
    }
  }
```

Traslación del origen

El método *translate()* se utiliza para trasladar el origen del contexto gráfico al punto de coordenada que se pasa como parámetro al método. El método modifica el contexto gráfico de forma que el origen del sistema de coordenadas que se utilice en ese contexto gráfico corresponda al punto especificado. Todas las coordenadas utilizadas en posteriores operaciones gráficas sobre ese contexto gráfico, serán relativas al nuevo origen.

El uso de este método es otra forma de eliminar el problema originado por la anchura de los bordes colocados sobre la zona de dibujo, *insets*. En particular, el método se puede utilizar para trasladar el origen del contexto gráfico a la esquina superior-izquierda de la zona cliente del objeto **Frame**, por la parte interna de los bordes.

El ejemplo `Java1512.java` replica la funcionalidad del ejemplo `Java1509.java`, pero elimina el problema de *insets* invocando al método *translate()* sobre el contexto gráfico original para desplazar el origen de coordenadas a la parte interna de los bordes.

Los cambios a realizar al ejemplo `Java1509.java` son mínimos, limitándose casi exclusivamente a la incorporación de la llamada al método *translate()*, para implementar esta nueva solución al problema de *insets*.

```
// Trasladamos el origen de coordenadas que se sitúa en la
// esquina superior izquierda, para evitar el problema que se
// produce con insets. De este modo el origen de coordenadas sí
// que lo dejamos situado en la zona cliente del objeto Frame
// que es la que se utiliza para pintar
g.translate( this.getInsets().left,this.getInsets().top );
```

Métodos para pintar figuras

Aquí se refieren los métodos destinados al dibujo de líneas, polilíneas, figuras geométricas, etc. tanto en las versiones lineales como en su correspondiente rellena. Algunos de estos métodos son los que se citan a continuación.

`drawLine(int,int,int,int)`, pinta una línea, utilizando el color actual, ente dos puntos del sistema de coordenadas del contexto gráfico.

`drawPolyline(int[],int[],int)`, pinta una secuencia de líneas conectadas, definidas por conjuntos de coordenadas `x` e `y`. La figura resultante no estará cerrada si el primer punto es diferente al último.

`drawRect(int,int,int,int)`, pinta una línea delimitando el rectángulo especificado por las esquinas indicadas en las coordenadas que se pasan utilizando el color actual del contexto gráfico.

`fillRect(int,int,int,int)`, rellena el rectángulo especificado con el color actual. Tenga en cuenta el lector que los lados izquierdo y derecho del rectángulo son `x` y `x+ancho-1`, y que los lados superior e inferior son `y` e `y+alto-1`; esto es así en todos los métodos en donde se rellenen figuras.

`drawArc(int,int,int,int,int,int)`, pinta una línea de un arco circular o elíptico.

`fillArc(int,int,int,int,int,int)`, rellena un arco circular o elíptico delimitado por el rectángulo que se indica.

`drawPolygon(Polygon)`, pinta un polígono definido por el objeto **Polygon** que se pasa como parámetro. La clase **Polygon** encapsula la descripción de una zona, dentro del espacio de coordenadas, delimitada por un número arbitrario de segmentos lineales, cada uno de los cuales corresponde a uno de los lados del polígono. Internamente, un polígono consta de una lista de pares de coordenadas `(x,y)`, donde cada uno de esos pares define un vértice del polígono y cada dos pares consecutivos definen un lado del polígono. Los vértices inicial y final se unen automáticamente formando otro de los lados del polígono.

De todos los métodos, quizás el más complejo sea *drawArc()*, por la forma en que hay que pasar los parámetros y el significado de éstos; el resto de los métodos no reviste dificultad alguna en su uso. Cada uno de los métodos que permiten representar en pantalla elipses y arcos, requieren como parámetros las coordenadas del punto central del óvalo o arco y la anchura y altura, en valor positivo, del rectángulo que circunscribe a ese óvalo o arco. Para pintar arcos, es necesario proporcionar dos parámetros adicionales, un ángulo de inicio y un ángulo para el arco; de este modo se especifica el inicio del arco y el tamaño del arco en grados (no en radianes). En la figura 15.7 se indica cómo se tiene en cuenta el ángulo al especificar los ángulos en los valores de los parámetros a pasar a los métodos *drawArc()* y *fillArc()*.

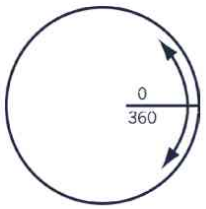


Figura 15.7

El ejemplo `Java1513.java` muestra ejemplos de todas las figuras que se pueden generar con los métodos que se acaban de enumerar. Si el lector compila y ejecuta el programa, verá una imagen semejante a la que presenta la figura 15.8.

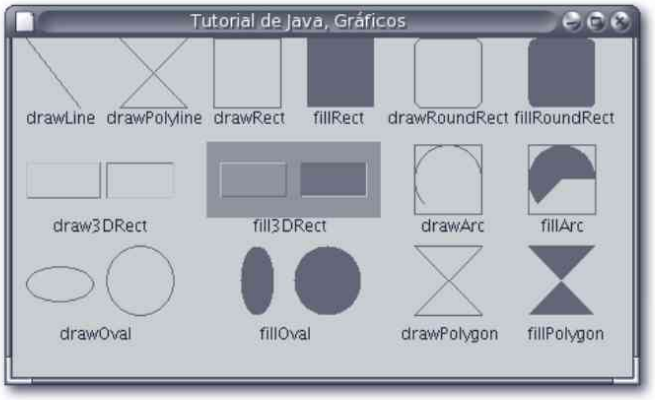


Figura 15.8

Parte del código del ejemplo es el que se reproduce a continuación, en el cual el lector puede comprobar los parámetros que se pasan a cada uno de los métodos para generar las figuras que se muestran en la imagen anterior.

```
    // Se sobrecarga el método paint()
  public void paint( Graphics g ){
    g.setColor( Color.red );
    // Trasladamos el origen de coordenadas que se sitúa en la
    // esquina superior izquierda, para evitar el problema que se
    // produce con insets. De este modo el origen de coordenadas sí
    // que lo dejamos situado en la zona cliente del objeto Frame
    // que es la que se utiliza para pintar
    g.translate( this.getInsets().left,this.getInsets().top );
    // Línea simple
    g.drawLine( 10,0,50,50 );
    g.setColor( Color.black );
    g.drawString( "drawLine",10,62 );
    g.setColor( Color.red );
    // Se crean dos arrays de coordenadas para pintar una
    // polilínea
    int x1Datos[] = {80,130,80,130};
    int y1Datos[] = {0,50,50,0};
    g.drawPolyline( x1Datos,y1Datos,4 );
    g.setColor( Color.black );
    g.drawString( "drawPolyline",70,62 );
    g.setColor( Color.red );
    // Rectángulo
```

```
g.drawRect( 150,0,50,50 );
g.setColor( Color.black );
g.drawString( "drawRect",150,62 );
g.setColor( Color.red );
// Rectángulo relleno
g.fillRect( 220,0,50,50 );
g.setColor( Color.black );
g.drawString( "fillRect",225,62 );
g.setColor( Color.red );
// Rectángulo redondeado
g.drawRoundRect( 300,0,50,50,10,10 );
g.setColor( Color.black );
g.drawString( "drawRoundRect",280,62 );
g.setColor( Color.red );
// Rectángulo redondeado relleno
g.fillRoundRect( 385,0,50,50,10,10 );
g.setColor( Color.black );
g.drawString( "fillRoundRect",375,62 );
// Pinta un rectángulo 3D, sobresaliendo de la pantalla
// No parece demasiado 3D
g.setColor( Color.gray );
g.draw3DRect( 10,90,55,25,true );
// Rectángulo 3D, pulsado
g.draw3DRect( 70,90,50,25,false );
g.setColor( Color.black );
g.drawString( "draw3DRect",30,140 );
// Rectángulo 3D relleno. Se ha puesto un fondo gris,
// con lo cual se puede apreciar mucho mejor el efecto
// de tres dimensiones
// Fondo gris
g.setColor( Color.gray );
g.fillRect( 145,75,130,55 );
g.fill3DRect( 155,90,50,25,true );
// Rectángulo 3D relleno, pulsado
g.fill3DRect( 215,90,50,25,false );
g.setColor( Color.red );
// De todos modos, la apariencia de tres dimensiones
// con 3DRect no es demasiado buena, porque es necesario
// seleccionar muy bien la paleta de colores para que
// se genere la ilusión de 3D
g.setColor( Color.black );
g.drawString( "fill3DRect",180,140 );
g.setColor( Color.red );
// Pinta un ángulo de 255 grados inscrito en un rectángulo
g.drawRect( 300,77,50,50 );
g.drawArc( 300,77,50,50,0,225 );
g.setColor( Color.black );
g.drawString( "drawArc",305,140 );
g.setColor( Color.red );
// Ángulo relleno de 255 grados inscrito en un rectángulo
g.drawRect( 385,77,50,50 );
g.fillArc( 385,77,50,50,0,225 );
g.setColor( Color.black );
g.drawString( "fillArc",395,140 );
g.setColor( Color.red );
// Elipse, con el eje grande horizontal
g.drawOval( 10,165,50,25 );
// Círculo
```

```
        g.drawOval( 70,150,50,50 );
        g.setColor( Color.black );
        g.drawString( "drawOval",35,218 );
        g.setColor( Color.red );
        // Elipse rellena, con el eje grande vertical
        g.fillOval( 170,150,25,50 );
        // Círculo relleno
        g.fillOval( 210,150,50,50 );
        g.setColor( Color.black );
        g.drawString( "fillOval",185,218 );
        g.setColor( Color.red );
        // Polígono
        int x2Datos[] = {300,350,300,350};
        int y2Datos[] = {150,200,200,150};
        g.drawPolygon( x2Datos,y2Datos,4 );
        g.setColor( Color.black );
        g.drawString( "drawPolygon",290,218 );
        g.setColor( Color.red );
        // Polígono relleno
        int x3Datos[] = {385,435,385,435};
        int y3Datos[] = {150,200,200,150};
        g.fillPolygon( x3Datos,y3Datos,4 );
        g.setColor( Color.black );
        g.drawString( "fillPolygon",385,218 );
    }
```

Quizás merezca la pena un repaso de los métodos *draw3Drect()* y *fill3Drect()*, que son los encargados de pintar rectángulos que aparecen en pantalla dando sensación de que están flotando sobre ella o hundidos. Este efecto visual se consigue mostrando dos de los lados en un color más claro que los otros dos, ofreciendo al ojo humano la ilusión de que los bordes más claros están iluminados y los más oscuros son producto de las sombras. Esto puede ser útil en ocasiones, pero hay otras en que en modo alguno se consigue el efecto. Si el color de fondo del contenedor sobre el que se va a colocar el rectángulo 3D, tiene un color cercano al color de ese rectángulo, entonces el efecto está conseguido, pero si se intenta colocar un rectángulo relleno con color rojo sobre un fondo blanco, el efecto pasa desapercibido.

Métodos para pintar texto

La clase **Graphics** dispone de métodos para pintar texto y métodos para devolver información sobre el texto y sobre la fuente de caracteres que se está utilizando para presentarlo en pantalla. Un resumen de estos métodos es lo que se presenta a continuación.

drawString(String,int,int), pinta el texto indicado en la cadena, en la posición indicada en los parámetros que corresponden a la coordenada donde se colocará el primer carácter de la cadena, utilizando el color y la fuente de caracteres actuales.

drawChars(char[],int,int,int), pinta el texto indicado en el array de caracteres, utilizando el color y fuente de caracteres actuales del contexto gráfico. Hay otra

versión que permite pasar un array de bytes para representar los caracteres que se van a pintar.

`getFont()`, obtiene la fuente de caracteres actual y devuelve un objeto de tipo **Font** que describe esa fuente de caracteres.

`getFontMetrics()`, obtiene los parámetros de la fuente actual y devuelve un objeto de tipo **FontMetrics**. Los métodos de la clase **FontMetrics** permiten obtener información sobre los parámetros de la fuente de caracteres (tamaño, estilo, etc.) a la que se aplica el método *getFontMetrics()*.

`getFontMetrics(Font)`, devuelve la información de la fuente indicada en el parámetro de llamada al método.

`setFont(Font)`, fija la fuente actual del contexto gráfico a la que se indica en el parámetro de llamada.

Al igual que ocurre en cualquiera de los apartados de la programación Java, la lista anterior solamente proporciona los métodos que permiten el control, pero el poder real se encuentra examinando detenidamente los métodos y las clases de los objetos que devuelven esos métodos, porque a la hora de pintar un texto, hay al menos tres clases implicadas. La primera es la clase **String**, de la que se trata profundamente en otro capítulo, y las demás clases son **FontMetrics** y **Font**, de las cuales se proporciona una pequeña descripción en los párrafos siguientes. Tenga en cuenta el lector, que ésta es una descripción muy somera y que las clases son realmente potentes, proporcionando una flexibilidad casi absoluta a la hora de pintar texto en pantalla.

Las coordenadas que se indican para posicionar el texto, corresponden al punto de la esquina inferior izquierda del texto, aunque no exactamente. En realidad, la coordenada **y** indica dónde está la línea base del texto, que no contempla el desplazamiento por debajo de esa línea de caracteres con imagen descendente, como la letra "g". Por ello, hay que asegurarse de especificar en la coordenada **y** un valor suficiente para permitir el pintado del carácter completo, porque en caso contrario, la parte descendente de los caracteres con esta característica no se visualizaría.

LA CLASE FONT

Una fuente de caracteres es un conjunto de caracteres de un tipo y un tamaño determinados. La fuente de caracteres define la apariencia, tamaño, estilo (itálica, negrilla, normal) de la cadena de texto que se va a pintar con esa fuente de caracteres.

Una *familia* de fuentes de caracteres no consiste solamente en un conjunto de fuentes con una apariencia similar, sino también de diferentes tamaños y estilos; por ejemplo, la fuente de caracteres *Helvetica negrilla* de *10 puntos* y la fuente *Helvetica itálica* de *12 puntos*, son fuentes de caracteres de la misma familia.

La clase **Font** permite crear objetos de tipo **Font** y, además, proporciona constantes simbólicas que permiten establecer las características del objeto **Font**

instanciado. No obstante, antes de instanciar un objeto **Font**, es necesario saber las fuentes de caracteres de que dispone el sistema y sus nombres. Las fuentes de caracteres tienen un nombre lógico, un nombre de familia y un nombre de fuente. El nombre lógico es el nombre que corresponde a una de las fuentes específicas del sistema y se puede obtener llamando al método *getName()*. El nombre de familia es el que corresponde al diseño tipográfico y se sabe invocando al método *getFamily()*. El nombre de fuente determina una fuente de caracteres dentro de una familia concreta, por ejemplo, *Helvetica Bold*; se sabe llamando al método *getFontName()*. Para determinar todas las fuentes disponibles en el sistema hay que llamar al método *getAllFonts()* de la clase **GraphicsEnvironment**.

La forma más sencilla de crear un objeto de tipo **Font** es indicar un nombre de fuente, tamaño en puntos y estilo. Una vez que se ha instanciado un objeto **Font**, se puede derivar de él cualquier número de nuevos objetos de tipo **Font** llamando al método *deriveFont()* sobre el objeto **Font** existente y especificando un nuevo tamaño de puntos, estilo, transformación (posición, escala, rotación, inclinación) o mapa de atributos, por ejemplo:

```
Font fuenteBold = new Font( "Helvetica",Font.BOLD,12 );
Font fDerivItalica = fuenteBold.deriveFont( Font.ITALIC,12 );
Font fDerivPlain = fuenteBold.deriveFont( Font.PLAIN,14 );
```

Una vez que se tiene una fuente, ya se puede utilizar para pintar cualquier texto. La plataforma Java 2 incluye un conjunto de fuentes *TrueType*, a las que denomina ***fuentes físicas***, para distinguirlas de las fuentes lógicas, como la fuente *Monospaced*, que se encuentran mapeadas en el fichero *font.properties* a las fuentes de la plataforma correspondiente.

La incorporación de las fuentes físicas permite que las aplicaciones tengan la misma apariencia en cualquiera de las plataformas, al utilizar el mismo conjunto de fuentes. Además, no es necesario cambiar de fuente para poder utilizar alfabetos diferentes, por ejemplo, en un campo de texto que utilice la fuente *Lucida Sans Regular*, se pueden presentar caracteres en hebreo, árabe, alemán e inglés.

Actualmente, todas las fuentes físicas de la plataforma Java 2 se engloban dentro de la familia de fuentes *Lucida*. Esta familia contiene 3 grupos: *Sans*, *Bright* y *Typewriter*, con 4 fuentes en cada grupo: *Regular*, *Bold*, *Oblique* y *Bold Oblique*. No obstante, se puede acceder a todas las fuentes disponibles en el sistema, o al menos a todas aquellas que se encuentren en el directorio especificado en la variable de entorno `JAVA_FONTS`.

LA CLASE FONTMETRICS

Un objeto de tipo **FontMetrics** proporciona toda la información necesaria sobre una determinada fuente de caracteres, lo que puede resultar imprescindible a la hora de posicionar un texto en la pantalla. La información que **FontMetrics** proporciona cuando se instancia sobre una determinada fuente incluye la línea de base, la parte

ascendente, la parte descendente, la anchura de los caracteres y el espacio que hay que dejar entre un carácter y el siguiente. Incluso es posible obtener información sobre el tamaño total de una cadena de texto cuando se pinta con una determinada fuente.

La figura 15.9 muestra gráficamente a qué corresponden algunos de los datos que un objeto de tipo **FontMetrics** puede proporcionar sobre una determinada fuente de caracteres.

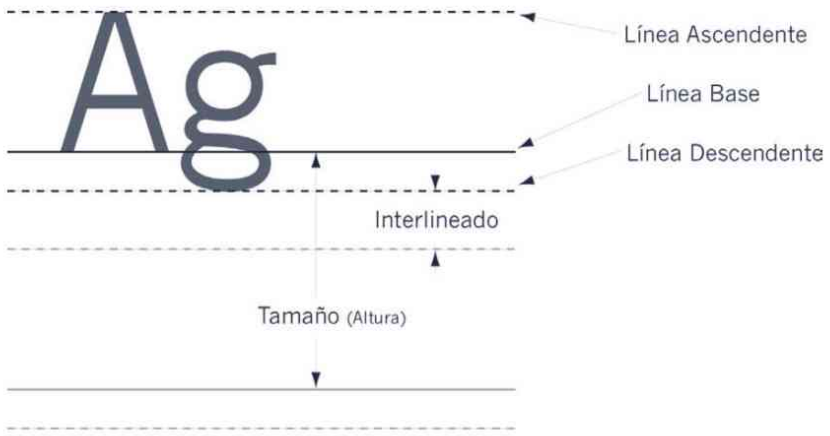


Figura 15.9

Hay que hacer notar al lector, no obstante, que el tamaño de la fuente devuelto por el método *getSize()* de la clase **Font**, es una medida abstracta; teóricamente, corresponde a la suma de la parte ascendente y la parte descendente; pero en la práctica es el diseñador de la fuente el que decide exactamente a qué se refiere una fuente de "12 puntos". Por ejemplo, la fuente *Times* de 12 puntos es ligeramente más pequeña que la fuente *Helvética* de 12 puntos, siendo un punto aproximadamente 1/72 de una pulgada.

El ejemplo `Java1514.java` muestra cómo se utilizan algunos de los métodos de las clases anteriores, teniendo en cuenta que el autor ha capturado la ventana ejecutando el ejemplo sobre *Sun Java Desktop 3*, por lo que la lista de fuentes disponibles se adecua a las que se encontraban instaladas, que puede no coincidir en absoluto con las fuentes de que disponga en lector en su plataforma específica.

```
// Se sobrecarga el método paint()
public void paint( Graphics g ) {
  // Trasladamos el origen de coordenadas que se sitúa en la esquina
  // superior izquierda, para evitar el problema que se produce con
  // insets
  g.translate( this.getInsets().left,this.getInsets().top );
  // Obtenemos la lista completa de fuentes del sistema
  String[] fuentes =
    GraphicsEnvironment.getLocalGraphicsEnvironment().
    getAvailableFontFamilyNames();
  // Fijamos una fuente cualquiera, a la quinta por ejemplo, le
  // cambiamos el estilo a Negrita y su tamaño a 14 puntos
  g.setFont( new Font( fuentes[4],Font.PLAIN,14 ) );
  // Obtenemos la altura de la fuente seleccionada
  int alto = g.getFontMetrics().getHeight();
```

```
    // Fijamos la Y utilizando la altura de la fuente como valor
    int y = alto;
    // Presentamos un rótulo general para la presentación
    String rotulo = "Algunas de las fuentes disponibles son:";
    g.drawString( rotulo,5,y );
    // Cambiamos a color rojo
    g.setColor( Color.red );
    // Presentamos una pequeña lista de todas las fuentes del sistema
    // indicando el nombre que tiene almacenado el sistema para ellas,
    // le cambiamos el estilo a Negrita y Oblicua
    for( int i=12; i < 18; i++ ) {
      g.setFont( new Font( fuentes[i],Font.BOLD | Font.ITALIC,18) );
      alto = g.getFontMetrics().getHeight();
      g.drawString( fuentes[i],
        160-(g.getFontMetrics().stringWidth(fuentes[i]))/2,y+=alto );
      }
    // Cambiamos a color azul
    g.setColor( Color.blue );
    // Volvemos a tomar la fuente del rótulo y cambiamos el estilo a
    // fuente Normal de 15 puntos
    g.setFont( new Font( fuentes[4],Font.PLAIN,15 ) );
    alto = g.getFontMetrics().getHeight();
    // Presentamos otro rótulo
    y += alto;
    g.drawString( "Información sobre la fuente de este texto: ",5,y );
    // Presentamos información sobre la fuente actual
    y += alto;
    g.drawString( "" + g.getFont(),5,y );
    // Presentamos otro rótulo
    y += 2*alto;
    g.drawString( "Aquí está parte de este Tutorial: ",5,y );
    // Creamos un array de letras para presentarlo
    char aTexto[] =
  { 'T','u','t','o','r','i','a','l',' ','d','e',' ','J','a','v','a' };
    y += alto;
    // Presentamos en pantalla el array de letras, saltándonos el
    // primero y no presentado los últimos
    g.drawChars( aTexto,1,13,5,y );
    }
```

En el programa se genera un array de cadenas conteniendo los nombres de cada una de las fuentes de caracteres disponibles en el sistema, presentando a continuación algunas de ellas indicando su propio nombre. El paso de línea se va incrementando en cada una de las líneas de texto que se presentan en pantalla. La figura 15.10 muestra la ventana resultado de la ejecución del ejemplo anterior.

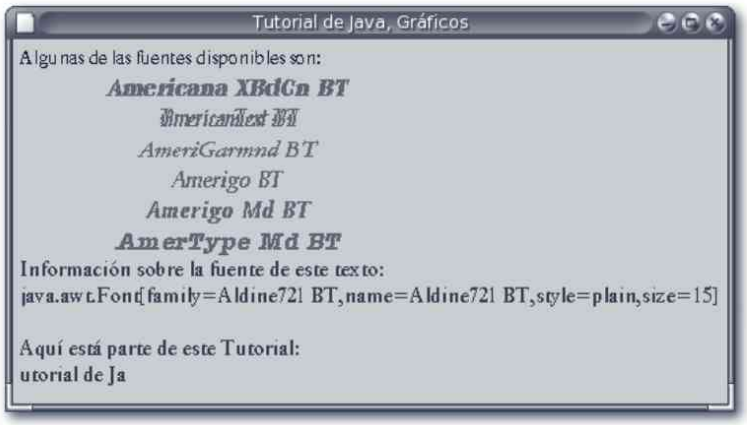


Figura 15.10

Recuerde el lector que la plataforma de ejecución era *Sun Java Desktop 3*.

El ejemplo `Java1515.java` es una aplicación que presenta al lector la información obtenida a través de la clase **FontMetrics** de la fuente de caracteres con que se presenta el texto en la ventana que genera el programa. La fuente elegida es *SansSerif*, pero puede modificarla el lector y recompilar el ejemplo para comprobar la información que Java proporciona acerca de cualquier otra fuente soportada por la plataforma en la cual se ejecute el programa.

La figura 15.11 muestra la captura de la ventana generada al ejecutar la aplicación `Java1516.java`. Esta aplicación intenta mostrar al usuario el uso de otras características especiales del uso de fuentes de caracteres en Java, de forma que es posible utilizar caracteres *Unicode* para presentar mensajes en alfabetos diferentes, teniendo en cuenta el mapeo que se hace de las *fuentes físicas* instaladas en la plataforma en donde se ejecuta la aplicación, y las *fuentes lógicas* que utiliza Java; mapeo que se realiza a través del archivo de configuración `font.properties`.



Figura 15.11

Aunque se puede utilizar cualquier fuente de caracteres en una ventana, no es posible cambiar el título de ninguna ventana, o contenedor de alto nivel como pueden ser **Frame**, **Dialog**, **JFrame** o **JDialog**, ni usando solamente AWT ni usando Swing, ya que el control del título de las ventanas corresponde al sistema operativo, por lo que es posible que la fuente de caracteres que se utiliza para la presentación de los títulos de las ventanas no contenga todo el conjunto de caracteres admitido.

Métodos de Clipping

La palabra ***Clipping*** está muy arraigada entre los programadores gráficos, sin tener una contrapartida en español adecuada, ya que quizá la que más se asemeje sea ***recorte***, aunque tampoco es demasiado precisa; por lo que se va a seguir utilizando el término sajón. ***Clipping*** es el proceso por el que se define la zona del contexto gráfico en la que se van a realizar las modificaciones de los subsiguientes procesos de dibujo. Cualquier píxel que se encuentre fuera del área de clipping permanecerá inmune a toda modificación.

Los métodos que están involucrados son varios, unos que permiten fijar el rectángulo de *clipping* y otros que permiten obtener la zona de *clipping* actual.

`clipRect(int,int,int,int)`, realiza la intersección del rectángulo de *clipping* actual con el formado por los parámetros que se pasan al método. Este método solamente se puede utilizar para reducir el tamaño de la zona de *clipping*, no para aumentarla.

`getClip()`, devuelve la zona de *clipping* actual como un objeto de tipo **Shape**. No obstante, **Shape** es una interfaz, por lo que hay que tener cuidado en su uso.

`getClipBounds()`, devuelve el rectángulo que delimita el borde de la zona de *clipping* actual.

`setClip(int,int,int,int)`, fija el rectángulo de *clipping* actual al indicado en las cuatro coordenadas que se pasan como parámetro al método.

Métodos para manejo de imágenes

Hay varias formas distintas para el método *drawImage()*, que difieren fundamentalmente en las posibilidades de manipulación del método y los parámetros que hay que pasar para realizar esas manipulaciones. La siguiente llamada es solamente representativa del grupo.

`drawImage(Image,int,int,int,int,int,int,int,int,Color,ImageObserver)`, pinta sobre la zona indicada la imagen que se pasa como parámetro, escalándola sobre la marcha para que rellene completamente el área de destino.

No obstante, antes de entrar en el desarrollo de los diferentes métodos *drawImage()*, hay que introducir la clase **Color**, que es relativamente simple de entender y utilizar.

LA CLASE COLOR

Esta clase es simple, encapsula los colores utilizando el formato *RGB*. En este formato, los colores se definen por sus componentes Rojo, Verde y Azul, representado cada uno de ellos por un número entero en el rango 0-255. En el caso del API Java2D, el espacio de color que se usa es el *sRGB*, un estándar del consorcio W3.

Hay otro modelo de color llamado *HSB* (matiz, saturación y brillo). La clase **Color** proporciona métodos de conveniencia para poder realizar la conversión entre un modelo y otro sin dificultad.

La clase **Color** proporciona variables estáticas finales que permiten el uso de cualquiera de los treinta colores simplemente especificando su nombre. Para usar estas variables sólo es necesario referenciar el color por su nombre de variable, por ejemplo:

```
objeto.setBackground( Color.red );
```

Los valores predefinidos para los colores en esta clase son los que se muestran en la siguiente lista, todos ellos en el espacio *sRGB*:

black, negro
blue, azul
cyan, azul celeste
darkGray, gris oscuro
gray, gris
green, verde
lightGray, gris claro
magenta, violeta
orange, naranja
pink, rosa
red, rojo
white, blanco
yellow, amarillo

La especificación del lenguaje Java indica que todos los nombres de constantes deben indicarse en minúsculas, por ello, todos los nombres han estado definidos en minúsculas, tal como se ha indicado, desde el inicio del desarrollo de Java. A partir del JDK 1.4 se han incorporado también las definiciones en mayúsculas de los colores básicos.

Los constructores de esta clase son tres, el primero de los cuales permite instanciar un nuevo objeto de tipo **Color** indicando las cantidades de rojo, verde y azul que entran en su composición, mediante valores enteros en el rango 0 a 255:

```
Color( int,int,int )
```

Otro de los constructores permite especificar la contribución en porcentaje de rojo, verde y azul al color final, admitiendo valores flotantes en el rango 0.0 a 1.0.

```
Color( float,float,float )
```

Y el tercer constructor admite solamente un número entero en el cual se especifica el valor RGB, de forma que la cantidad de color rojo que interviene en el color final está especificada en los bits 16-23 del argumento, la cantidad de verde en los bits 8-15 y la cantidad de azul en los bits 0-7.

```
Color( int )
```

Para el uso con los colores, la plataforma Java 2 dispone de una serie de métodos que aquí se intentan englobar en base a su funcionalidad.

Métodos que devuelven un entero representando el valor RGB para un determinado objeto **Color**.

`getRed()`, devuelve el componente rojo del color como un entero en el rango 0 a 255.

`getGreen()`, devuelve la cantidad de verde que entra en la composición del objeto **Color**, en el rango 0 a 255.

`getBlue()`, devuelve el componente azul del color con un entero en el rango 0 a 255.

`getRGB()`, devuelve un entero representando el color RGB, utilizando los bits para indicar la cantidad de cada uno de los componentes rojo, verde y azul que entran en su composición. Los bits 24 a 31 del entero que devuelve el método son `0xff`, los bits 16 a 23 son el valor rojo, los bits 8 a 15 son el valor verde y los bits 0 a 7 indican el valor del color azul. Siempre en el rango 0 a 255.

Los siguientes métodos devuelven un objeto de tipo **Color**, de forma que se pueden utilizar para crear objetos de este tipo.

`brighter()`, crea una versión más brillante del color.

`darker()`, crea una versión más oscura del color.

`decode(String)`, convierte una cadena a un entero y devuelve el color correspondiente.

Los métodos siguientes también devuelven un objeto de tipo **Color**, pero están especializados para trabajar con el sistema a través de la clase **Properties**.

`getColor(String)`, busca un color entre las propiedades del sistema. El objeto **String** se utiliza como el valor clave en el esquema clave/valor utilizado para describir las propiedades en Java. El valor es entonces utilizado para devolver un objeto **Color**.

`getColor(String,Color)`, busca un color entre las propiedades del sistema. El segundo parámetro es el que se devuelve en el caso de que falle la búsqueda de la clave indicada en el objeto **String**.

`getColor(String,int)`, busca un color entre las propiedades del sistema. El segundo parámetro es utilizado para instanciar y devolver un objeto **Color** en el caso de que falle la búsqueda de la clave indicada en el objeto **String**.

Los métodos que se indican a continuación se utilizan para realizar la conversión entre el modelo de color *RGB* y el modelo *HSB*.

`getHSBColor(float,float,float)`, crea un objeto de tipo **Color** basado en los valores proporcionados por el modelo HSB.

`HSBtoRGB(float,float,float)`, convierte los componentes de un color, tal como se especifican en el modelo HSB, a su conjunto equivalente de valores en el modelo RGB.

`RGBtoHSB(int,int,int,float[])`, convierte los componentes de un color, tal como se especifican en el modelo RGB, a los tres componentes del modelo HSB.

Y ya, los métodos que quedan son métodos de utilidad general, sin ninguna aplicación específica en lo que al **Color** respecta.

`equals(Object)`, determina si otro objeto es igual a un color.

`hashCode()`, calcula el código *hash* para un color.

`toString()`, crea una cadena representando al color, indicando el valor de sus componentes RGB.

El método *drawImage()*

Una vez presentada la clase **Color**, se puede seguir con el método *drawImage()* de la clase **Graphics**, que es el que se utiliza para presentar una imagen en pantalla. Aunque hay métodos de otras clases que son utilizados para la manipulación de imágenes en Java, lo primero que se necesita es saber cómo se visualiza una imagen, y eso es lo que hace el método *drawImage()*.

Hay varias versiones sobrecargadas de este método, algunas de las cuales se muestran a continuación:

`drawImage(Image,int,int,Color,ImageObserver)`, pinta la imagen que se indica situando su esquina superior izquierda en la coordenada que se pasa, en el contexto gráfico actual. Los píxeles transparentes se pintan en el color de fondo que se indica. Esta operación es equivalente a rellenar un rectángulo de la anchura y altura de la imagen dada con un color y luego pintar la imagen sobre él, aunque probablemente más eficiente.

`drawImage(Image,int,int,ImageObserver)`, pinta la imagen que se indica situando su esquina superior izquierda en la coordenada que se pasa, en el contexto gráfico actual. Los píxeles transparentes de la imagen no se ven afectados. Este método

retorna inmediatamente en todos los casos, incluso aunque la imagen completa no se haya terminado de cargar y no se haya presentado completamente en el dispositivo de salida.

drawImage(Image,int,int,int,int,Color,ImageObserver), pinta la imagen que se pasa dentro del rectángulo que se indica en los parámetros, escalando esa imagen si es necesario. El método retorna inmediatamente, aunque la imagen no se haya cargado completamente. Si la representación de la imagen en el dispositivo de salida no se ha completado, entonces *drawImage()* devolverá el valor `false`.

drawImage(Image,int,int,int,int,ImageObserver), pinta la imagen que se pasa dentro del rectángulo que se indica en los parámetros, escalando esa imagen si es necesario. El método retorna inmediatamente, aunque la imagen no se haya cargado completamente. Si la representación de la imagen en el dispositivo de salida no se ha completado, entonces *drawImage()* devolverá el valor `false`. El proceso que pinta las imágenes notifica al observador a través de su método *imageUpdate()*.

drawImage(Image,int,int,int,int,int,int,int,int,Color,ImageObserver).

drawImage(Image,int,int,int,int,int,int,int,int,ImageObserver).

Como puede comprobar el lector, se permite mucha flexibilidad a la hora de presentar imágenes, aunque quizá necesite una descripción más detallada de la llamada a cada método, por ejemplo, la descripción que encontrará en el API para una de las versiones del método es:

```
public abstract boolean drawImage( Image img,
                                   int x,
                                   int y,
                                   int width,
                                   int height,
                                   ImageObserver observer )
```

Los parámetros del método son:

- `img`, la imagen que se quiere pintar.
- `x` e `y`, coordenadas para posicionar la esquina superior izquierda de la imagen.
- `width` y `height`, dimensiones del rectángulo sobre el que se posicionará la imagen.
- `observer`, objeto que ha de ser notificado tan pronto como la imagen se haya convertido y situado en su posición.

En esta versión del método, la imagen se pintará dentro del rectángulo que se pasa en los parámetros y se escalará si es necesario. El método retorna inmediatamente, incluso aunque la imagen no se haya terminado de escalar o convertir al formato que admita el dispositivo de salida. Si esta conversión no se ha completado, *drawImage()* devuelve `false`, y como hay parte de imagen que no está procesada, el proceso que

pinta la imagen notifica al observador a través de su método *imageUpdate()* y esa imagen continúa pintándose por trozos.

El método *imageUpdate()* pertenece a la clase **Component**, cuando es llamado, encola una llamada a *repaint()*, permitiendo que se pinte un trozo más de imagen sin que se interfiera en el proceso de la tarea principal. Hay también una propiedad, **awt.image.incrementaldraw**, que determina si la imagen puede ser pintada en piezas, tal como se ha indicado antes. El valor por defecto para esta propiedad es `true`, en cuyo caso el sistema va pintando trozos de imagen según van llegando. Si el valor es `false`, el sistema esperará a que la imagen esté completamente cargada antes de pintarla.

Hay una segunda propiedad, **awt.image.redrawrate** que determina el período mínimo, en milisegundos, que debe haber entre llamadas a *repaint()* para imágenes. El valor por defecto es 100, y solamente se aplica cuando la propiedad anterior es `true`. Tanto esta propiedad como la anterior se pueden modificar, pero solamente sirve esa modificación para la ejecución actual del programa.

En el ejemplo `Java1517.java` se muestra el efecto de pintar una imagen en trozos, según van estando disponibles, además del escalado de esa imagen. La salida visual que se genera no es demasiado satisfactoria porque tiende a producir un cierto parpadeo al realizar varias llamadas al método *imageUpdate()* durante el proceso de pintado de la imagen.

La versión escalada de una imagen no necesariamente estará disponible inmediatamente, porque antes debe construirse una versión en dimensiones normales para el dispositivo de salida, sin embargo, como podrá comprobar el lector en el ejemplo `Java1517.java`, este efecto aparentemente no se produce.

De los parámetros que se pasan al método *drawImage()*, todos resultan obvios, excepto quizá, el último de ellos, `observer`, que se refiere al objeto que debe ser notificado cuando la imagen esté disponible. Cuando se realiza una llamada a *drawImage()*, se lanza una tarea que carga la imagen solicitada. Hay un observador que monitoriza el proceso de carga; la tarea que está cargando la imagen notifica a ese observador cada vez que llegan nuevos datos. Para este parámetro se puede utilizar perfectamente `this` como observador en la llamada a *drawImage()*; es más, se puede decir que cualquier componente puede servir como observador para imágenes que se pintan sobre él. Quizás sea éste el momento de entrar un poco más en detalle en la clase **ImageObserver**, e incluso antes en la clase **Image**, porque son los dos parámetros diferentes del típico entero que aparecen en la llamada a *drawImage()*.

La clase Image

La clase **Image**, como cualquier otra, está formada por una serie de constantes o variables, constructores y métodos. Uno de estos métodos es *getScaledInstance()*, que devuelve una versión escalada de una imagen. Uno de los parámetros de este método

es un valor entero que especifica el algoritmo a utilizar para la realización del escalado. Se puede utilizar cualquiera de ellos, la lista siguiente muestra su nombre y una pequeña descripción.

`SCALE_AREA_AVERAGING`, utiliza el algoritmo *Area Averaging*.

`SCALE_FAST`, utiliza el algoritmo que proporcione mayor rapidez en el escalado, a costa de la suavidad de realización de ese escalado.

`SCALE_SMOOTH`, al contrario que el anterior, sacrifica la velocidad por la suavidad de realización del escalado.

`SCALE_REPLICATE`, utiliza un algoritmo interno de la clase **ReplicateScaleFilter**.

`SCALE_DEFAULT`, utiliza el algoritmo de defecto.

`UndefinedProperty`, un objeto de este tipo debe ser devuelto siempre que una propiedad no se encuentre definida para una determinada imagen. Esta propiedad no se utiliza para el escalado, sino que es utilizada como valor de retorno del método *getProperties()* para indicar que la propiedad solicitada no está disponible.

Aunque la clase **Image** dispone de un constructor, es una clase abstracta, por lo que no se puede instanciar ningún objeto llamando a este constructor. Se puede conseguir un objeto **Image** indirectamente por la invocación del método *getImage()* de las clases **Applet** o **Toolkit**. El método *getImage()* utiliza una tarea separada para cargar la imagen. El resultado práctico de la invocación a *getImage()*, es la asociación entre una referencia de tipo **Image** y un fichero localizado en algún lugar que contiene la imagen que interesa; en este Tutorial se utilizan imágenes que están en el disco local, pero nada impide que puedan encontrarse en cualquier servidor de la Red.

Otra forma de obtener un objeto **Image** es invocar al método *createImage()*, de las clases **Component** y **Toolkit**. La lista siguiente muestra una pequeña descripción de los métodos de la clase **Image**:

`flush()`, libera todos los recursos que utiliza el objeto **Image**.

`getGraphics()`, crea un contexto gráfico para pintar una imagen en segundo plano.

`getHeight(ImageObserver)`, determina la altura de la imagen.

`getWidth(ImageObserver)`, determina la anchura de la imagen.

`getProperty(String,ImageObserver)`, obtiene una propiedad de la imagen por su nombre.

`getScaledInstance(int,int,int)`, crea una versión escalada de la imagen.

`getSource()`, devuelve el objeto que produce los píxeles de la imagen.

Algunos de estos métodos se utilizarán en los ejemplos que se presentan. No obstante, el interés no debe centrarse solamente en la clase **Image**, sino también en las otras clases que se necesitan para instanciar los parámetros de llamada a estos

métodos, e incluso, algunas otras clases de soporte general. Por ello, aun a costa de resultar un poco ladrillo, se presenta a continuación una pequeña revisión de las clases más interesantes, **MediaTracker** e **ImageProducer**, ya que sobre las clases **String**, **ImageObserver**, **Graphics** y **Object**, que son las que intervienen en los métodos de la clase **Image**, ya debería tener el lector alguna referencia.

La interfaz ImageProducer

Se trata de una interfaz para objetos que pueden producir imágenes para la clase **Image**. Cada imagen contiene un **ImageProducer** que se utiliza en la reconstrucción de esa imagen cuando es necesario; por ejemplo, cuando se escala la imagen, o cuando se cambia la imagen de tamaño. Esta interfaz declara varios métodos, algunos de los cuales se verán un poco más adelante.

La clase MediaTracker

Ésta es una clase de utilidad general diseñada para controlar el estado de los objetos de tipo *media*, que en teoría pueden ser tanto *clips* de sonido como cualquier otro objeto *media*, como es el caso de una imagen. Por ahora, en la plataforma Java 2 solamente se soporta el control de imágenes.

Un objeto **MediaTracker** se utiliza a través de una instancia de la clase **MediaTracker** sobre el objeto **Component** a monitorizar, e invocando al método *addImage()* para cada una de las imágenes que se quiere controlar. A cada una de estas imágenes se puede asignar un identificador único, o también se puede asignar el mismo identificador a un grupo de imágenes. Este identificador controla el orden de prioridad en que se cargarán, de forma que imágenes con un identificador bajo tienen preferencia sobre otras con identificador más alto. El identificador también se puede utilizar para identificar un conjunto único de imágenes. En otras palabras, asignando el mismo identificador a varias imágenes, se las puede manejar a la vez.

Se puede determinar el estado de una imagen (o grupo de imágenes) invocando alguno de los distintos métodos sobre el objeto **MediaTracker**, pasándole como parámetro el identificador de la imagen (o grupo de imágenes).

El objeto **MediaTracker** también se puede utilizar para bloquear la ejecución, a la espera de la carga completa de una imagen (o grupo de imágenes), tal como se verá en alguno de los ejemplos, en que se espera a que la carga de las imágenes se realice completamente antes de presentarlas en pantalla.

La clase **MediaTracker** proporciona cuatro constantes, que se utilizan como valores de retorno de algunos de los métodos de la clase y, tal como su nombre sugiere, indican el estado en que se encuentra una imagen dada. Estas constantes:

`ABORTED`, la carga de la imagen (o cualquier *media*) ha sido abortada.

`COMPLETE`, realizada satisfactoriamente la carga completa.

`ERRORED`, se ha encontrado algún tipo de error en la carga.

`LOADING`, se está cargando la imagen (o cualquier *media*).

Para esta clase solamente hay un constructor que puede ser invocado a la hora de instanciar objetos que controlen el estado de algunas, o todas, las imágenes sobre un determinado componente.

```
MediaTracker( Component )
```

Sin embargo, el número de métodos disponibles para los distintos propósitos, sí que es bastante extenso. Se ha intentado agruparlos en categorías, pero se recuerda al lector que lo que aquí se expone es una pequeña descripción de cada uno de ellos para poder tener una visión global del funcionamiento, en este caso de la manipulación de imágenes, y que sea sencilla de entender su función; pero si el lector va a utilizar estos métodos en sus propios programas, debe consultar la documentación del API de la plataforma Java que esté utilizando.

Un objeto **MediaTracker** tiene la posibilidad de controlar el estado de algunas, o todas, las imágenes que están siendo cargadas para el objeto **Component** que se ha pasado como parámetro a la hora de instanciar ese objeto **MediaTracker**. Los métodos siguientes se utilizan para crear y mantener esta lista de imágenes. Cuando se añade una imagen a la lista, hay que proporcionar un identificador numérico para esa imagen, que será luego el que sea utilizado por otros métodos para poder indicar el estado en que se encuentra una imagen determinada.

`addImage(Image,int)`, incorpora una nueva imagen a la lista de imágenes que está siendo controlada por el objeto **MediaTracker**.

`addImage(Image,int,int,int)`, incorpora una imagen escalada a la lista.

`removeImage(Image)`, elimina la imagen indicada de la lista.

`removeImage(Image,int)`, elimina la imagen que corresponde al identificador que se pasa como parámetro de la lista.

`removeImage(Image,int,int,int)`, elimina la imagen que corresponde al ancho, alto e identificador, de la lista de imágenes.

El objeto **MediaTracker** se puede utilizar para hacer que la tarea, o hilo de ejecución, se bloquee hasta que una o más imágenes de su lista hayan completado su carga. Esto se consigue con los siguientes métodos:

`waitForAll()`, inicia la carga de todas la imágenes controladas por el objeto **MediaTracker**, devuelve `void`.

`waitForAll(long)`, inicia la carga de todas la imágenes, devolviendo `true` si todas las imágenes se han cargado y `false` en cualquier otro caso.

waitForID(int), inicia la carga de las imágenes que corresponden al identificador que se pasa, devuelve `void`.

waitForID(int,long), inicia la carga de las imágenes que corresponden al identificador que se pasa; devuelve `true` si todas las imágenes se han cargado y `false` en cualquier otro caso.

Por supuesto, es posible utilizar métodos que no bloquean la tarea que carga las imágenes para comprobar el estado de una o más imágenes de la lista. Esto permite continuar haciendo el trabajo mientras las imágenes siguen cargándose. Estos métodos devuelven `true` o `false` para indicar si la carga es completa. Observará el lector que hay dos versiones sobrecargadas de cada uno de los métodos. La versión con el parámetro booleano comenzará la carga de cualquier imagen que no se esté cargando, en caso de que ese parámetro booleano sea `true`. La otra versión no iniciará la carga de ninguna imagen. Esta interpretación también es válida para el parámetro booleano de otros métodos de la clase que disponen de él.

checkAll() y *checkAll(boolean)*, comprueban si todas las imágenes que están siendo controladas por el objeto **MediaTracker** han finalizado la carga.

checkID(int) y *checkID(int,boolean)*, comprueban si todas las imágenes que corresponden al identificador especificado han concluido su carga.

El hecho de que los métodos anteriores indiquen que la carga de las imágenes ha sido completa, no garantiza que esa carga esté libre de errores. Los siguiente métodos se utilizan para determinar si se ha producido algún problema durante la carga de las imágenes.

getErrorsAny(), devuelve una lista de todas las imágenes (o cualquier *media*) en las que se ha producido un error en la carga.

getErrorsAny(int), devuelve una lista de todas las imágenes correspondientes a un determinado identificador en las que se ha producido un error en la carga.

isErrorAny(), comprueba el estado de error de todas las imágenes.

isErrorID(int), comprueba el estado de error de todas las imágenes correspondientes a un determinado identificador.

Los dos siguientes métodos devuelven un valor entero formado por la operación OR entre los valores de estado de todas las imágenes que se requieren; en el primero de todas las que controla el objeto **MediaTracker**, y en el segundo de las que corresponden al identificador que se especifica.

statusAll(boolean).

statusID(int,boolean).

El ejemplo `Java1517.java` muestra la carga y pintado de una imagen sin utilizar la clase **MediaTracker**, que sí se utilizará en la clase **Java1518**. El ejemplo muestra

el uso de la clase **Toolkit** y el método *getImage()* para asociar un fichero de imagen que se encuentra en el disco duro local con el nombre de una variable de referencia de tipo **Image**. También muestra el uso del método *drawImage()* para pintar la imagen cargada sobre un objeto **Frame**; el uso de *getWidth()* y *getHeight()* para determinar el tamaño de la imagen; el uso de *translate()* para eliminar el efecto de *offset* generado por los *insets* del objeto **Frame**; y algunos otros métodos que ya deben ser archiconocidos para el lector. El código del ejemplo es el que se reproduce a continuación.

```
// Clase de control del ejemplo
class Java1517 extends Frame {
  // Referencia a la imagen
  Image imagen;
  // Constructor de la clase
  public Java1517() {
    // Obtenemos en la variable "imagen" el fichero de imagen que se
    // indica, y que se supone situado en el mismo directorio y disco
    // que la clase del ejemplo
    imagen = Toolkit.getDefaultToolkit().getImage(
      "imágenes/muneco.gif" );
    // Se hace visible el Frame, que en la pantalla da origen a la
    // ventana, aunque la primera imagen no es visible en el mismo
    // momento en que aparece la ventana en pantalla, porque hasta que
    // se invoque por primera vez el método paint(), no se colocará
    // una imagen en el contenedor
    this.setVisible( true );
    }

  // Se sobrecarga el método para pintar la imagen
  public void paint( Graphics g ) {
    // Se traslada el origen para evitar el efecto del borde
    g.translate( this.getInsets().left,this.getInsets().top );
    // Ahora se pinta la imagen a la mitad de su tamaño
    g.drawImage( imagen,0,0,
      imagen.getWidth(this)/2,imagen.getHeight(this)/2,this );
    }
  }
```

La figura 15.12 reproduce la ventana generada en la ejecución del programa, que se repetirá en los demás ejemplos, en donde se utilizan distintas formas de presentar lo mismo, por lo que sirva de referencia al lector esta imagen para éste y subsiguientes ejemplos.



Figura 15.12

Como el lector también supondrá, lo que sigue ahora es la revisión de algunos de los trozos de código del ejemplo que resultan más interesantes. Aunque precisamente en este ejemplo, no sea muy interesante el detenerse porque es una simple presentación y toma de contacto con la clase **Image**, pero sí se pueden revisar aquellas partes que van a ser comunes en todos los ejemplos.

Así que lo primero donde detenerse es la declaración de la variable de instancia que es una variable de referencia de tipo **Image** y que será la que posteriormente se utilice para manipular esa imagen.

```
class Java1517 extends Frame {
  Image imagen;  // Referencia a un objeto Image
```

El siguiente trozo de código que aquí debe resaltarse es la sentencia, en el constructor del contenedor, que utiliza el método *getImage()* de la clase **Toolkit** para asociar el fichero *muneco.gif* con la variable de referencia imagen.

```
imagen = Toolkit.getDefaultToolkit().getImage("imagenes/muneco.gif");
```

Es importante hacer notar al lector que esta sentencia no hace que la imagen aparezca en pantalla en el momento en que el contenedor se haga visible.

Las líneas de código que de verdad puedan interesar del ejemplo hay que buscarlas ya en el método *paint()*, en donde se invoca una de las versiones sobrecargadas del método *drawImage()*, al cual se le indica en la lista de parámetros que se le pasa; en primer lugar, el nombre de la variable de referencia que apunta al fichero en disco que contiene la imagen, luego las coordenadas donde se debe colocar la esquina superior izquierda de la imagen, después la anchura y altura que debe tener la imagen, y por fin, `this` como objeto observador.

```
g.drawImage( imagen,0,0,
  imagen.getWidth(this)/2,imagen.getHeight(this)/2,this );
```

Los métodos *getWidth()* y *getHeight()* de la clase **Image** se han utilizado para determinar el tamaño original de la imagen y hacer que se presente en pantalla reducida a la mitad para que el ejemplo no sea muy soso. Observe el lector que el

parámetro que hay que pasar a estos métodos es el observador, para que puedan indicarle las dimensiones de la imagen.

La clase **Java1517** no proporcionaba un efecto visual demasiado bueno mientras se carga la imagen para presentarla en pantalla. Esto se corrige al usar un objeto **MediaTracker** para bloquear la tarea hasta que la imagen esté totalmente cargada, con lo que se elimina el parpadeo que se produce al ir presentándose en pantalla partes de esa imagen que no está totalmente cargada, como se muestra en el ejemplo `Java1518.java`.

El tiempo máximo de carga permitido es un segundo; si la imagen no se carga satisfactoriamente en un segundo, el programa presentará un mensaje indicando la circunstancia y sale. Una imagen grande, o una imagen que se cargue directamente desde Internet (que es una fuente muy lenta), puede necesitar más tiempo. El código correspondiente al método *paint()* es el que se reproduce a continuación, y se revisa después.

```
// Se sobrecarga el método para pintar la imagen
public void paint( Graphics g ) {
  // Se traslada el origen para evitar el efecto del borde
  g.translate( this.getInsets().left,this.getInsets().top );
  // Se utiliza un objeto MediaTracker para bloquear la tarea hasta
  // que la imagen esté cargada completamente, antes de intentar
  // pintarla en pantalla. Si la carga de la imagen falla en el
  // intervalo de un segundo, el programa se termina. Sin el uso de
  // un objeto de tipo MediaTracker, si la imagen es pequeña, apenas
  // se notará el uso de MediaTracker, pero sí que es importante a
  // la hora de cargar imágenes muy grandes o gran cantidad de ellas
  MediaTracker tracker = new MediaTracker( this );
  // Se añade la imagen a la lista del tracker
  tracker.addImage( imagen,1 );
  try {
    // Se bloquea la tarea durante un segundo mientras se intenta
    // la carga de la imagen. En caso de que la imagen sea mayor que
    // la que se utiliza en este ejemplo, puede ser necesario
    // aumentar este tiempo
    if( !tracker.waitForID(1,1000) ) {
      System.out.println( "Fallo en la carga de la imagen" );
      System.exit( 0 );
      }
  } catch( InterruptedException e ) {
    System.out.println( e );
    }
  // Ahora se pinta la imagen a la mitad de su tamaño normal
  g.drawImage( imagen,0,0,
    imagen.getWidth(this)/2,imagen.getHeight(this)/2,this );
  // Y se presenta en pantalla
  this.repaint();
  }
```

Para revisar el código del ejemplo, se prescindirá de la parte que está duplicada con respecto a ejemplos anteriores, para ver solamente el código nuevo. Comenzando esta revisión, lo primero en que hay que detenerse es en la sentencia que instancia un

objeto de tipo **MediaTracker**. El parámetro del constructor es el objeto **Component** a quien pertenecen las imágenes que se van a controlar. En este caso es:

```
MediaTracker tracker = new MediaTracker( this );
```

El siguiente código interesante es la sentencia que incorpora una nueva imagen a la lista de imágenes a controlar. En este caso los parámetros a pasar son una referencia al fichero de la imagen y un identificador que se utilizará posteriormente para saber el estado en que se encuentra la carga de la imagen; para este identificador se puede utilizar cualquier valor entero, teniendo en cuenta que este valor representa la prioridad de carga, siendo los valores más bajos los que tienen más alta prioridad.

```
tracker.addImage( imagen,1 );
```

Ahora hay que detenerse en la invocación del método *waitForId()* sobre el objeto controlador. En este caso, el identificador de la imagen que se está controlando es el parámetro a pasar. También se pasa un valor de 1.000 milisegundos como parámetro para hacer que el programa cese el bloqueo y termine si la carga de la imagen no se ha completado dentro de ese intervalo de tiempo. Este método devuelve `true` si la imagen se ha cargado con éxito y `false` en cualquier otro caso. Hay algunos otros métodos que se pueden utilizar para este mismo propósito y que devuelven `void`, por lo que necesitan otras referencias para saber si la imagen se ha cargado bien.

En este caso, el valor de retorno del método *waitForID()* es utilizado para terminar el programa si la carga de la imagen falla (note el lector el operador `"!"` delante de la palabra `tracker`). Este método también lanza una excepción de tipo **InterruptedException** si la carga no se puede completar en el intervalo de tiempo determinado; por ello, la llamada al método debe ir encerrada en un bloque `try-catch`.

```
try {
  if( !tracker.waitForID(1,1000) ) {
    System.out.println( "Fallo en la carga de la imagen" );
    System.exit( 0 );
    }
} catch( InterruptedException e ) {
  System.out.println( e );
  }
```

Finalmente, se utiliza el método *drawImage()* para pintar la imagen sobre el contenedor, una vez cargada. Cuando el lector ejecute el programa verá que la sensación visual es mucho mejor que la del ejemplo anterior porque solamente se realiza una operación de dibujo para pintar la imagen completa.

```
g.drawImage( imagen,0,0,
  imagen.getWidth(this)/2,imagen.getHeight(this)/2,this );
```

ANIMACIÓN

En párrafos anteriores se han presentado muchos de los aspectos principales del funcionamiento de las imágenes en Java; sin embargo, el problema clásico en todo el tratamiento de gráficos es siempre el mismo, la *animación*. Para mejorar en lo posible la ilusión de que las imágenes que aparecen en pantalla están animadas, se va a manejar ampliamente el *doble buffer*.

La *animación* se puede concretar en la presentación de imágenes en sucesión, cada una con ligeras diferencias respecto a la anterior. Si las imágenes se presentan con suficiente rapidez y los cambios incrementales no son muchos, el ojo humano creerá a pies juntillas que esas imágenes se están viendo en movimiento.

En caso de que lo anterior no se haga correctamente, la imagen parpadeará y destruirá la ilusión de movimiento. Los ejemplos que se presentan aquí se pueden ejecutar con o sin el uso del doble buffer, con lo cual el lector podrá comprobar perfectamente la diferencia tan abrumadora que hay entre el uso del doble buffer y prescindir de él.

El primer ejemplo, `Java1519.java`, muestra lo comentado en el párrafo anterior, es decir, la diferencia que existe entre el uso o no del doble buffer. La imagen que se utilice en el ejemplo ha de ser lo suficientemente grande para poder observar detalles en su tamaño normal, en principio sirve cualquier imagen para ver el efecto. Por simplicidad, el nombre del fichero que contiene la imagen está incluido en el código del programa, por lo que si el lector quiere utilizar otra imagen, deberá cambiar ese nombre o incluir código adicional en el método *main()* para solicitar la introducción de un nombre de fichero.

Si el programa no es capaz de cargar la imagen en diez segundos, cancelará su ejecución con un mensaje de error. Si el fichero que contiene la imagen es muy grande, o se está cargando desde un servidor Web, puede ser necesario modificar el código del ejemplo e incrementar el intervalo de tiempo que controla la carga de la imagen. Para ejecutar el ejemplo utilizando doble buffer, es suficiente con introducir en la línea de comandos

```
java Java1519
```

y si se quiere ejecutar el programa con el doble buffer deshabilitado, se incorpora un segundo parámetro `x`, de forma que la ejecución se realizaría introduciendo en la línea de comandos:

```
java Java1519 x
```

La animación se realiza sobre un objeto **Frame** que actúa de contenedor, y que se ajusta automáticamente al tamaño de la imagen. Una vez arrancado el programa, la aplicación continuará animando la imagen hasta que se pulse el botón de cerrar la ventana situado en la esquina superior derecha de la ventana. La imagen se escalará

hasta su tamaño original, y luego al revés, como si fuese un yoyó; arrancando desde la esquina superior-izquierda del **Frame** al lanzar el programa. Evidentemente, para que la animación sea lo más realista posible, no se utilizan imágenes preescaladas, sino que el escalado de cada una de las imágenes se realiza en el buffer oculto y en el mismo instante previo al que se va a presentar en pantalla, dentro del método *paint()*.

El método *paint()* se invoca unas 20 veces por minuto, lo que es suficientemente rápido para que se produzca la ilusión de animación que se pretende proporcionar; no obstante, cuando se llega a las zonas en que la imagen es más grande, parece que la animación se ralentiza debido al tiempo que es necesario para escalar las imágenes de mayor tamaño. Probablemente, una mejor aproximación sería utilizar un reloj independiente del tiempo que se tarde en realizar la operación de escalado de la imagen, porque en este ejemplo, el método *paint()* se invoca aproximadamente 50 milisegundos después de que ha terminado, pero el tiempo que tarda en escalar las imágenes grandes es mayor que en el caso de que las imágenes tengan un menor tamaño. No obstante, el ejemplo solamente pretende introducir al lector en los aspectos básicos de la animación, no en la perfección del algoritmo.

La imagen de la pantalla se modifica cada vez que se invoca a *paint()*. Cuando se está utilizando el doble buffer, el contenido del buffer es transferido a la pantalla cuando se entra en el método *paint()*; luego se genera la siguiente versión de la imagen en el buffer oculto, que será el que se envíe a la pantalla en la siguiente entrada de *paint()*, y concluye el trabajo del método. Si no se está utilizando doble buffer, la imagen se va pintando directamente en pantalla al entrar en el método y cuando se termina de dibujar, el método concluye.

En Java, cuando se invoca al método *repaint()* para que se produzca un refresco de la pantalla, una de las cosas que ocurren es que se llama al método *update()* que a su vez llama a *paint()*. El método *update()* normalmente borra toda la ventana cada vez que *repaint()* es invocado, llamando a continuación a *paint()*. No se suele sobrecargar el método *update()* excepto en circunstancias muy especiales, como ocurre en este caso. Aquí el método *update()* se sobrecarga para eliminar el borrado innecesario de la ventana, que es el que provoca el parpadeo de la imagen. Sin embargo, como ahora ya no está *update()* para borrar la ventana, hay que hacer esto en el método *paint()* (cuando no esté en modo doble buffer), pero ya más controlado, de forma que solamente se borre la zona donde se encontraba la imagen antes de colocar la nueva, eliminando los dibujos residuales.

En el caso del doble buffer no es necesario borrar la ventana, porque es el contenido completo de esa ventana el que será reemplazado por la nueva que se está generando en el buffer oculto; aunque sí es necesario borrar la zona de la imagen del doble buffer antes de pintar la nueva.

Cuando se ejecuta la aplicación sin doble buffer hay un cierto parpadeo, sobre todo en el escalado de las imágenes grandes, y la sensación de animación no es muy buena. Aparentemente, este parpadeo se debe a que se pinta la imagen completa cada

vez; es decir, que aunque las operaciones gráficas se realizan muy rápidamente, todavía se puede observar un leve rastro de lo que van haciendo, con lo cual, entre el borrado y el pintado de la nueva imagen, provocan ese parpadeo. Algo así como lo que pasa en las películas antiguas de cine, que el lector seguramente habrá observado alguna vez.

Al utilizar el doble buffer, el parpadeo se elimina completamente y la ilusión de movimiento se consigue de forma mucho más eficaz. En este caso, la generación y pintado de la nueva imagen tienen lugar en el buffer oculto, que no está visible al observador, y cuando está finalizado el proceso, se produce un intercambio de buffers, entre el que se está viendo actualmente y el que se ha terminado de generar en la parte oculta. Este intercambio se produce muy rápidamente, de forma que el observador no es capaz de apreciarlo, solamente se puede ver el resultado final.

Eso sí, tenga en cuenta el lector lo que se está utilizando en el ejemplo como animación, que es el escalado. Y ésta no es una operación perfecta en ninguna circunstancia, bien por el detalle de las imágenes o por la representación de píxeles; esto es especialmente visible en la presentación de caracteres. Por ello, si a la ejecución del ejemplo no se ve muy perfecto, se debe a esta circunstancia, no a que el proceso se realice de forma anómala.

El listado completo del ejemplo lo puede encontrar el lector en el soporte digital que acompaña al libro, a continuación se discuten las partes más interesantes, que en este caso sí hay bastantes.

El primer lugar donde hay que detenerse es en el conjunto de variables de referencia que se utilizan para implementar la funcionalidad del programa. Especialmente interesante es la variable de tipo **Image** llamada `imagenFuente` que apunta a la imagen que se encuentra en disco. También son interesantes las variables `imgDobleBuffer`, que constituyen una referencia a un objeto de tipo **Image** que se utiliza para contener la versión que se crea del dibujo, oculta a la vista durante el proceso de utilización del doble buffer; y la variable `contextoDobleBuffer` de tipo **Graphics**, que es el contexto gráfico para ese objeto. Recuerde el lector que no se puede pintar sobre un objeto **Image**; hay que obtener un contexto gráfico para ese objeto y pintar sobre el contexto.

```
Image imagenFuente;              // Imagen cargada de disco
Image imgDobleBuffer;
Graphics contextoDobleBuffer;
```

En la aplicación habrá numerosas referencias a estas variables de instancia. El siguiente código interesante es la parte que comprueba la línea de argumentos que ha introducido el usuario, para saber si se va a operar utilizando el modo doble buffer o no, por defecto se utiliza el modo doble buffer.

```
if( args.length == 0 ) {
  obj.usarDobleBuffer = true;
  System.out.println( "Doble Buffer Activado" );
```

```
  }
else {
  obj.usarDobleBuffer = false;
  System.out.println( "Doble Buffer Desactivado" );
  }
```

Posteriormente, en el código del método *paint()* se comprobará el estado de la variable `usarDobleBuffer` para determinar si el doble buffer va a ser utilizado o no.

La parte que sigue en el código ya es la implementación de bucle que realiza la animación, controlada en el método *main()*, en donde el bucle se ejecuta con un intervalo de retraso de 50 milisegundos. En cada iteración se realizan los cálculos necesarios para establecer el tamaño de la siguiente imagen a presentar y se invoca al método *repaint()*.

Cuando se invoca a *repaint()*, se provocan llamadas en cascada a otros métodos, de forma que: *repaint()* llama a *update()*, el cual llama a *clear()* y luego llama a *paint()*. De hecho, el método *update()* hace algo como lo que se muestra en las siguientes líneas de código:

```
public void update( Graphics g ) {
  g.setColor( getBackground() );
  g.fillRect( 0,0,ancho,alto );
  g.setColor( getForeground() );
  paint( g );
  }
```

El efecto de *update()*, por lo que refiere a su uso en este ejemplo, es el redibujo de la imagen sobre un fondo de color (sobre el contexto gráfico) y luego la llamada al método *paint()* para presentar ese contexto gráfico en la pantalla. Posteriormente, se sobrecargará el método *update()* para eliminar las tres sentencias que solamente hacen que *repaint()* invoque a *paint()*.

Volviendo al código del ejemplo, las líneas siguientes más interesantes corresponden al bucle sin fin donde se realizan los cálculos para la generación de la imagen siguiente a presentar; se llama al método *repaint()* y finalmente se coloca una pausa de 50 milisegundos antes de entrar en la nueva iteración.

Los cálculos y comprobaciones que se realizan son muy simples. El código solamente decide si la imagen está aumentando o disminuyendo su tamaño, comprobando los límites de cambio de una acción a otra, y luego fija el factor de escala para calcular el nuevo tamaño de la imagen siguiente a generar. Después de llamar al método *repaint()*, la tarea se duerme durante 50 milisegundos, al cabo de ese tiempo se despierta y realiza una nueva iteración.

Ahora le toca el turno al constructor. Lo primero que aparece en su código es la sentencia que obtiene la referencia a la imagen que se encuentra en disco que se va a manipular, que ya se ha visto en múltiples ejemplos.

```
imagenFuente = Toolkit.getDefaultToolkit().getImage(
  "imagenes/muneco.gif" );
```

Ahora hay que controlar que la imagen esté completamente cargada para empezar a obtener versiones escaladas, para ello se utiliza un objeto de tipo **MediaTracker** como control de la carga de la imagen.

```
MediaTracker tracker = new MediaTracker( this );
tracker.addImage( imagenFuente,1 );
try {
  if( !tracker.waitForID( 1,10000 ) ) {
    System.out.println( "Error en la carga de la imagen" );
    System.exit( 1 );
  }
} catch( InterruptedException e ) {
  System.out.println( e );
  }
```

Una vez que se ha instanciado el objeto **MediaTracker**, se invoca a su método *addImage()* para incorporar la imagen que se quiere cargar a la lista de imágenes que controla ese objeto. En este caso se asigna el identificador "1" para la imagen, que posteriormente puede ser utilizado para obtener información sobre la imagen.

El objeto **MediaTracker** también puede bloquear la tarea hasta que la imagen está totalmente cargada, o bien durante un período de tiempo determinado. En este caso, se utiliza el método *waitForID()* que bloquea la tarea hasta que la imagen, cuyo identificador se ha pasado, se haya cargado completamente, o bien hasta que hayan pasado diez segundos, lo que suceda primero será lo que desbloquee la tarea. El método *waitForID()* devuelve `true` si la imagen se ha cargado correctamente dentro del intervalo de los diez segundos y `false` si no es así, en cuyo caso se presenta un mensaje y se termina la ejecución del programa.

El método *waitForID()* lanza una excepción de tipo **InterruptedException**, que tiene que ser comprobada, por lo que hay que declarar esta circunstancia en la firma inicial del método, o bien encerrar a *waitForID()* en un bloque `try-catch` que capture esa excepción.

Las dos líneas siguientes son las que determinan el tamaño de la ventana que va a servir como contenedor a la imagen, y que se adapta automáticamente al tamaño de la imagen que se acaba de cargar. Por ello, las dimensiones de la imagen se pasan a las variables de instancia declaradas al principio de la definición de la clase, que serán las que estarán disponibles posteriormente en el proceso de ajuste del tamaño del **Frame** a las dimensiones de la imagen que va a contener.

```
iniAncho = imagenFuente.getWidth( this );
iniAlto = imagenFuente.getHeight( this );
```

La información de *insets* también es necesaria para que el objeto **Frame** se adapte a la imagen. El siguiente paso es, pues, hacer visible el objeto **Frame** y capturar la información de *insets* que se asigna a dos variables de instancia. Tenga en cuenta el

lector que la información de *insets* no está disponible hasta que el objeto **Frame** se asocia a un objeto de control del sistema; y el método *setVisible()* es una forma de conseguirlo.

```
this.setVisible( true );
insetArriba = this.getInsets().top;
insetIzqda = this.getInsets().left;
```

Ahora se puede fijar el tamaño del **Frame** teniendo en cuenta todas las dimensiones obtenidas.

```
this.setSize( insetIzqda+iniAncho,insetArriba+iniAlto );
```

El siguiente código es totalmente nuevo. En él se utiliza el método *createImage()* para generar una imagen de un tamaño determinado, que puede utilizarse para crear la imagen en el buffer oculto. Recuerde el lector, de nuevo, que no se puede pintar directamente sobre un objeto **Image**, sino que hay que obtener un contexto gráfico para esa imagen, por ello se realiza una llamada al método *getGraphics()*.

```
imgDobleBuffer = this.createImage( iniAncho,iniAlto );
contextoDobleBuffer = imgDobleBuffer.getGraphics();
```

Con esto concluye la revisión del constructor, el siguiente código corresponde al método *update()* sobrecargado, en donde se ha eliminado la parte de borrado de la pantalla antes de la llamada al método *paint()*.

```
public void update( Graphics g ) {
  paint( g );
  }
```

Ahora le toca el turno al método sobrecargado *paint()*, que es donde se realiza todo el trabajo de pintado. En este método se distinguen dos partes, la parte que pinta en modo doble buffer y la parte que pinta sin la utilización del doble buffer. Este método recibe como parámetro el contexto gráfico para el objeto **Frame**, la clase para la que el método está siendo sobrecargado. El contexto gráfico se conoce en el método como `g`, y el método también tiene acceso al contexto gráfico de la imagen oculta, que el método conoce como `contextoDobleBuffer`.

Cada vez que se invoca a este método en modo doble buffer, se pinta una nueva imagen en el contexto gráfico oculto y se deja ahí. Se utiliza el método *drawImage()* del contexto gráfico para hacer esto, pasándole el `contextoDobleBuffer` como parámetro. Esto hace que `contextoDobleBuffer` sea pintado sobre la pantalla en la posición especificada por los parámetros segundo y tercero. Observe el lector que como ésta es una versión que no escala del método *drawImage()*, se ejecuta muy rápidamente.

```
g.drawImage( imgDobleBuffer,insetIzqda,insetArriba,this );
contextoDobleBuffer.clearRect( 0,0,iniAncho,iniAlto );
contextoDobleBuffer.drawImage( imagenFuente,
  0,0,finAncho,finAlto,this );
```

Ahora es necesario pintar una nueva imagen en el buffer oculto, basada en los valores `finAncho` y `finAlto`, comenzando a partir de la esquina superior izquierda. El proceso de escalado se ejecuta con mucha más lentitud que la versión de *drawImage()* que no realiza el escalado. Sin embargo, esto no reviste problema porque el escalado se está realizando fuera de la vista del usuario. Por supuesto, si se quieren realizar operaciones complejas que ralenticen mucho el proceso, se puede resentir de forma significativa la velocidad que se desea para la animación.

Y ya se sale del método *paint()*, dejando en el buffer oculto la imagen lista para que sea mostrada en la siguiente invocación que se haga al método.

La otra posibilidad de invocación del método *paint()* es sin usar el doble buffer, en donde se ejecuta el código que se reproduce a continuación.

```
g.clearRect( insetIzqda,insetArriba,iniAncho,iniAlto );
g.drawImage( imagenFuente,
  insetIzqda,insetArriba,finAncho,finAlto,this);
```

En primer lugar se invoca al método *clearRect()* del contexto gráfico de la pantalla para borrar la zona que está ocupando la imagen que se está viendo actualmente. Luego se utiliza la versión con escalado del método *drawImage()* para pintar la nueva imagen, escalándola durante el proceso de presentación. Todo esto se realiza a la vista del usuario, lo cual provoca parpadeo y en algunos casos el efecto es totalmente desastroso.

En los dos casos, se use o no el doble buffer, se utiliza el método *drawImage()* para pintar la imagen en la pantalla. La principal diferencia es que en el modo sin doble buffer se utiliza una versión de este método que realiza el escalado de la imagen. Esto hace que sea más lento que la otra versión que simplemente transfiere la imagen escalada previamente a la pantalla, sin ningún retraso significativo en el proceso.

USO AVANZADO DE IMÁGENES

Las capacidades de procesado de imágenes de Java se encuentran englobadas en el paquete **java.awt.image**. Observe el lector que ésta no es la clase **Image** del paquete **java.awt**. Este paquete contiene ocho interfaces y una treintena de clases. No se verán todas aquí, sino solamente las más interesantes.

Interfaces

Ésta es una pequeña descripción de las tres interfaces más interesantes que son: **ImageProducer**, **ImageConsumer** e **ImageObserver**.

Las clases que implementan la interfaz **ImageProducer** sirven como fuentes de píxeles. Los métodos de estas clases pueden generar los píxeles a partir de la pantalla, o bien puede interpretar cualquier otra fuente de datos, como cualquier fichero gráfico

en un formato soportado por Java. No importa cómo genere los datos, el principal propósito de un productor de imágenes es proporcionar píxeles a una clase **ImageConsumer**.

Los productores de imágenes operan como fuentes de imágenes, y el modelo *productor/consumidor* que se sigue en el tratamiento de imágenes es el mismo que se utiliza en el modelo *fuente/receptor* de eventos.

En particular, los métodos declarados en la interfaz **ImageProducer** hacen posible que uno o más objetos **ImageConsumer** se registren para mostrar su interés en una imagen. El productor de la imagen invocará a los métodos declarados en la interfaz **ImageConsumer** para enviar píxeles a los consumidores de imágenes.

Un productor de imágenes puede registrar muchos consumidores de sus píxeles, de la misma forma que una fuente de eventos puede registrar múltiples receptores de sus eventos. En la interfaz **ImageProducer** hay varios métodos para manipular la lista de consumidores, por ejemplo para añadir nuevos consumidores interesados en los píxeles del productor, o bien para eliminar consumidores de la lista.

La interfaz **ImageConsumer** declara métodos que deben ser implementados por las clases que reciben datos desde un productor de imágenes. El principal de estos métodos es *setPixels()*.

```
void setPixels( int,int,int,int,ColorModel,byte[],int,int )
```

La interfaz **ImageObserver** ya ha sido discutida en otro párrafo anterior, por lo que no se insiste más.

La clase ColorModel

Las imágenes sobre la pantalla están formadas a partir de puntos individuales. Cada uno de esos puntos se llama ***píxel***, que es una abreviatura del término anglosajón *Picture Element*.

Eso ya es conocido por el lector, pero dependiendo de cómo se quiera la representación de la imagen en pantalla, un solo píxel puede contener mucha información, o casi ninguna. Por ejemplo, si se quiere dibujar una serie de líneas del mismo color sobre un fondo de un color diferente, la única información que debe contener cada píxel es si debe estar encendido o apagado, es decir, que solamente sería necesario un bit de información para representar al píxel.

En el otro extremo de la información que puede encontrar un píxel que muestre los gráficos complejos de los juegos de ordenador, que requieren 32 bits para representar un píxel individual. Estos 32 bits se dividen en grupos de cuatro octetos de bits, o bytes. Tres de estos grupos representan la cantidad de colores fundamentales: *rojo*, *verde* y *azul*, que forman parte del color del píxel; y el cuarto byte, normalmente

conocido con byte *alpha*, se utiliza para representar la transparencia, en un rango de 0 a 255, siendo 0 la transparencia total y 255 la opacidad completa.

En teoría, este formato permite la utilización de 16 millones de colores en cada píxel individual, aunque puede ser que el monitor o la tarjeta gráfica no sean capaces de visualizar electrónicamente tal cantidad de colores. Cada uno de los extremos se puede considerar como un *modelo de color*. En Java un modelo de color determina cuántos colores se van a representar dentro del AWT. La clase **ColorModel** es una clase abstracta que se puede extender para poder especificar representaciones de color propias.

Ahora bien, el lector puede plantearse el porqué de la necesidad de un modelo de color. Esto es necesario porque hay muchos métodos que reciben un array de bytes y convierten esos bytes en píxeles para su presentación en pantalla, es decir, convierten los bytes de datos en píxeles visuales sobre la pantalla. Por ejemplo, con un modelo de color directo simple, cada grupo de cuatro bytes se podría interpretar como la representación de un valor del color de un píxel individual. En un modelo de color indexado simple, cada byte en el array se podría interpretar como un índice a una tabla de enteros de 32 bits donde cada uno de esos enteros representa el valor del color adscrito al píxel.

El AWT proporciona tres subclases de **ColorModel**: **IndexColorModel**, **ComponentColorModel** y **PackedColorModel**. Aunque se puede definir cualquier subclase que el lector tenga en mente.

La clase IndexColorModel

Cuando se utilizan imágenes de alta resolución, se consume gran cantidad de memoria; por ejemplo, una imagen de dimensiones 1024x768, contiene 786.432 píxeles individuales. Si se quieren representar estos píxeles con sus cuatro bytes en memoria, se necesitarán 3.145.728 bytes de memoria para esta imagen.

Para evitar este rápido descenso de memoria, se han incorporado varios esquemas de forma que las imágenes se puedan representar de forma razonable sin comprometer demasiado el uso de memoria. Un esquema muy común consiste en localizar un pequeño número de bits para cada píxel (8 bits por ejemplo) y luego utilizar los valores de los píxeles como un índice a una tabla de enteros de 32 bits que contenga el subconjunto de todos los colores que se utilizan en la imagen. Por ejemplo, si se utilizan ocho bits para representar un píxel, el valor del píxel se puede utilizar como índice a una tabla de contenga hasta 256 colores que se utilicen en la imagen. Estos 256 colores se pueden seleccionar entre los millones disponibles, conociéndose a este subconjunto como la *paleta de colores* de la imagen.

Volviendo al ejemplo de la imagen de dimensiones 1024x768, empleando este último esquema solamente serían necesario 1.024 bytes para representar la paleta de

256 colores y 786.432 bytes para representar la imagen. Si el lector es habitual de entornos X, este modelo es semejante al modelo *PseudoColor* de X11.

La clase DirectColorModel

Esta clase extiende a la clase **PackedColorModel** e implementa el formato completo de color de 32-bit, en el que cada píxel está formado por los cuatro bytes, representando el canal *alfa* y las cantidades de rojo, verde y azul que componen cada píxel. En entornos X, este modelo es semejante al modelo *TrueColor* de X11.

La clase FilteredImageSource

Esta clase es una implementación de la interfaz **ImageProducer** que toma una imagen y un objeto de tipo **ImageFilter** para generar una nueva imagen que es una versión filtrada de la imagen original. Las operaciones de filtrado que se pueden realizar permiten una gran variedad de acciones, como son el desplazamiento y sustitución de colores, la rotación de imágenes, etc.

La clase ImageFilter

Esta clase es una implementación de la interfaz **ImageConsumer**. Además de los métodos declarados en la interfaz, hay métodos para implementar un filtro *nulo*, que no realiza modificación alguna sobre los datos que se le pasan.

Hay varias subclases que extienden esta clase base para permitir la manipulación de la imagen, como son **RGBImageFilter**, **CropImageFilter**, **ReplicateScaleFilter** y **AreaAveragingScaleFilter**.

La clase MemoryImageSource

Esta clase es una implementación de la interfaz **ImageProducer** que utiliza un array de datos para generar los valores de los píxeles de una imagen. Dispone de varios constructores, cuyos parámetros son (todos o en parte) los siguientes.

- Anchura y altura en píxeles de la imagen que va a ser creada.
- Modelo de color que se empleará en la conversión, si el constructor no requiere este parámetro utilizará el modelo de color RGB por defecto.
- Un array de bytes o enteros conteniendo los valores a ser convertidos en píxel según el modelo de color especificado.
- Un offset para indicar el primer píxel dentro del array.
- El número de píxeles por línea en el array.
- Un objeto de tipo **Hashtable** conteniendo las propiedades asociadas de la imagen en caso de que la haya.

En todos los casos, el constructor genera un objeto **ImageProducer** que se utiliza como un array de bytes o enteros para generar los datos de un objeto **Image**. Esta clase puede pasar múltiples imágenes a consumidores interesados en ellas, recordando a la funcionalidad *multiframe* que ofrece el formato gráfico *GIF89a* para producir animaciones.

La clase PixelGrabber

Ésta es una clase de utilidad para convertir una imagen en un array de valores que corresponden a sus píxeles. Implementa la interfaz **ImageConsumer**, que puede ser acoplada a un objeto **Image** o **ImageProducer** para realizar cualquier manipulación de esa imagen.

Ejemplo

Como se ha indicado antes, las capacidades de procesado de imágenes de Java son muy amplias, así que serían necesarios varios ejemplos para poder presentar cada una de ellas. Por eso, solamente se tratarán algunas, dejando al lector que investigue las demás por sí mismo.

El ejemplo `Java1520.java` muestra la forma de manipular imágenes, eliminando el componente rojo de todos los píxeles de una imagen y haciéndola parcialmente transparente. Hay gran parte del código del ejemplo que ya se ha visto anteriormente, y no merece la pena insistir más en él, así que solamente se revisará aquello que proporciona nuevo al ejemplo. La figura 15.13 muestra el resultado de la ejecución.

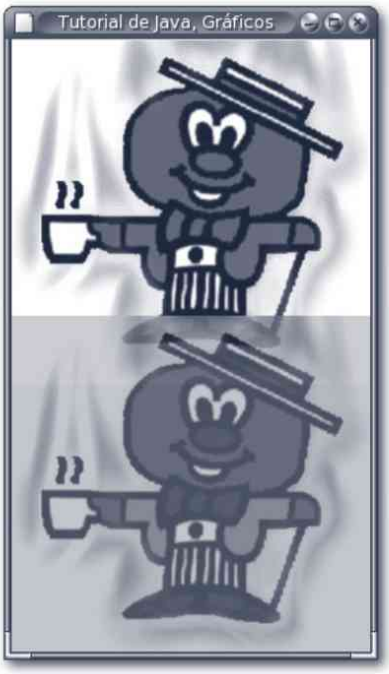


Figura 15.13

El programa comienza leyendo una imagen de disco y salvándola en memoria bajo el nombre de `imagenFuente`.

```
imagenFuente =
  Toolkit.getDefaultToolkit().getImage( "imagenes/muneco.gif" );
```

Luego declara un array de enteros de suficiente tamaño para almacenar el valor de cada uno de los píxeles de la imagen, llamado `pix`.

```
int[] pix = new int[iniAncho * iniAlto];
```

Se instancia un objeto de tipo **PixelGrabber** que asocia la `imagenFuente` con el array de enteros `pix`.

```
PixelGrabber pgObj = new PixelGrabber( imagenFuente,
  0,0,iniAncho,iniAlto,pix,0,iniAncho );
```

Una vez que se ha conseguido el objeto **PixelGrabber**, se invoca el método *grabPixels()* sobre ese objeto para realizar la conversión de la imagen a su forma numérica, y también se invoca el método *getStatus()* para comprobar el estado de la operación.

```
if( pgObj.grabPixels() &&
  ( (pgObj.getStatus() & ImageObserver.ALLBITS ) != 0 ) ) {
```

Según *Sun Microsystems*, el método *getStatus()* necesita un observador que indique qué información se desea obtener, en este caso se hace uso de **ALLBITS**, pero se pueden usar cualquiera de los que están disponibles, que son los siguientes:

- **ABORT**, una imagen que estaba siendo controlada asíncronamente fue abortada antes de que se completase su proceso.
- **ALLBITS**, una imagen estática está completa y puede ser pintada de nuevo en su forma final.
- **ERROR**, en una imagen que estaba siendo controlada asíncronamente se ha detectado un error.
- **FRAMEBITS**, otra de las imágenes de una imagen compuesta por múltiples *frames* está lista para pintarse de nuevo.
- **HEIGHT**, la altura de la imagen base está ya calculada y disponible.
- **PROPERTIES**, están disponibles las propiedades de la imagen.
- **SOMEBITS**, están disponibles más píxeles de la versión escalada de la imagen.
- **WIDTH**, la anchura de la imagen está ya calculada y disponible.

Si la sentencia anterior devuelve `true`, el siguiente paso es procesar los valores numéricos de los píxeles para generar las modificaciones deseadas en la imagen.

Una vez que ya están los datos de los píxeles en formato numérico (con sus bytes correspondientes al canal *alfa* y componentes rojo, verde y azul), se utiliza un bucle y una operación AND para enmascarar el byte correspondiente al componente *rojo* y para modificar el byte *alfa*.

```
for( int i=0; i < (iniAncho*iniAlto); i++ ) {
  pix[i] = pix[i] & 0xC000FFFF;
  }
```

El siguiente paso es el uso del método *createImage()* de la clase **Component** para crear una imagen a partir del array numérico. Para conseguirlo, es necesario instanciar un objeto de tipo **MemoryImageSource** describiendo la forma en que se quiere convertir el array numérico en imagen. Esto se reproduce en el siguiente código.

```
imagenNueva = this.createImage( new MemoryImageSource(
  iniAncho,iniAlto,pix,0,iniAncho ) );
```

Y, por último, la revisión finaliza en el método sobrecargado *paint()*, que es el que presenta la imagen original y la imagen alterada, para poder compararlas.

```
g.drawImage( imagenFuente,insetIzqda,insetArriba,this );
// Colocamos la imagen nueva debajo de la original, superponiéndola
// ligeramente para poder comprobar el efecto de la modificación
// del canal alfa de esta nueva imagen, que dejará ver la parte
// de la imagen original sobre la que se encuentre
g.drawImage( imagenNueva,insetIzqda,insetArriba+iniAlto-50,this );
```

En pantalla aparecen las dos imágenes tal como indica el comentario incluido en las líneas anteriores.

La clase AlphaComposite

Esta clase permite combinar dos imágenes en base a una docena de reglas conocidas como *reglas Porter-Duff*. La posibilidad de combinar imágenes es imprescindible en aplicaciones destinada al campo lúdico, en las cuales es habitual incluir múltiples imágenes, tanto para utilizarlas como imágenes de fondo, imágenes de jugadores, imágenes de herramientas, etc. Siempre resulta sencillo, en un juego de plataformas, por ejemplo, presentar al jugador sobre el fondo; sin embargo, si el jugador pasa por detrás de un objeto, será necesario que la imagen del jugador se vaya *fundiendo* con la imagen del objeto, y es en este momento cuando la composición de imágenes resulta útil.

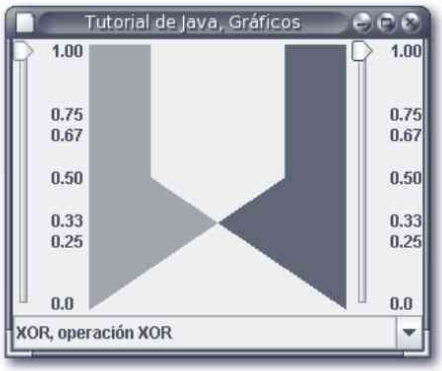


Figura 15.14

El ejemplo `Java1521.java`, cuya ejecución inicial reproduce la figura 15.14, permite ver en acción los doce tipos de constantes que define la clase **AlphaComposite**, una para cada una de las reglas *Porter-Duff*, y la modificación de las escalas de valores laterales, correspondientes a la imágenes origen y destino, permitirá al lector comprobar el efecto de los distintos niveles de transparencia en todas ellas.

CAPÍTULO 16

MÉTODOS NATIVOS

La plataforma Java 2 incorpora la interfaz de programación **Java Native Interface** (*JNI*), para permitir que se puedan escribir programas en otros lenguajes diferentes a Java y mantener la portabilidad entre todas las plataformas, entendiendo por métodos o aplicaciones *nativas* aquellas escritas en un lenguaje diferente a Java. Además, JNI permite que el código Java que se ejecute en una *Máquina Virtual Java* pueda interactuar con aplicaciones y librerías escritas en otros lenguajes, como C, C++, Fortran, Cobol o ensamblador. Además, la interfaz de programación *Invocation API* permite que se puedan llamar a código de la Máquina Virtual Java desde aplicaciones nativas.

El objetivo de utilizar JNI para escribir métodos nativos es el de permitir que los programadores puedan manejar aquellas situaciones en las que una aplicación no puede ser escrita enteramente en Java. Por ejemplo, algunas de las situaciones en las que puede ser necesario recurrir a métodos nativos son las siguientes.

- La librería de clases estándar de Java no soporta las características dependientes de plataforma necesarias para la ejecución de la aplicación.
- Ya hay una aplicación o librería escrita en otro lenguaje y se quiere que sea accesible a los programas Java.
- Puede necesitarse escribir una pequeña porción de código crítica en cuanto a su tiempo de ejecución, en un lenguaje de bajo nivel, ensamblador por ejemplo, y hacer que los programas Java llamen a esas funciones.

La programación a través del entorno que proporciona JNI para implementar métodos nativos involucra varias operaciones, y proporciona a estos métodos nativos una gran flexibilidad, de forma que un método nativo puede utilizar los objetos Java del mismo modo que el propio Java los usa. Un método nativo puede crear objetos

Java, incluyendo arrays y objetos de tipo **String**, y luego inspeccionarlos y utilizarlos para sus propias operaciones. Un método nativo también puede inspeccionar y utilizar objetos creados por la parte de código Java de la aplicación. Un método nativo puede actualizar objetos Java ya creados para que luego sean usados por la parte de código Java de la aplicación. En resumen, las partes Java y nativa de la aplicación pueden crear, actualizar, acceder y compartir objetos Java.

Los métodos nativos pueden llamar a métodos Java con suma facilidad. El lector terminará por construir su propia librería de métodos Java, así que no es necesario que *reinvente la rueda* para implementar una funcionalidad ya incorporada a métodos Java existentes. JNI permite utilizar las ventajas que ofrece el lenguaje de programación Java desde métodos nativos; en concreto, permite capturar y lanzar excepciones desde métodos nativos y que estas excepciones sean manejadas en la aplicación Java. Finalmente, los métodos nativos pueden utilizar JNI para realizar la comprobación de tipos estricta que proporciona Java en tiempo de ejecución.

La figura 16.1 resume todas las características anteriores, mostrando cómo JNI sirve de enlace entre Java y aplicaciones nativas, por ejemplo, escritas en lenguaje C.

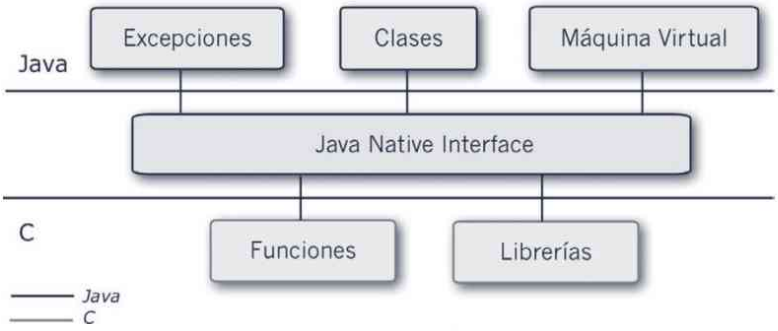


Figura 16.1

A continuación, se muestran los pasos necesarios para mezclar código nativo C y programas Java. Se recurre al archiconocido *saludo*; en este caso, el programa `HolaMundoN` tiene dos clases Java: la primera implementa el método *main()* y la segunda, **HolaMundoN**, tiene un método nativo que presenta el mensaje de saludo. La implementación de este segundo método se realiza en C, y la clase se llama **HolaMundoN**, para indicar que es la versión del saludo que contiene código nativo en su interior.

Las acciones que se deben realizar para conseguir que la nueva versión del saludo funcione, serán las que se desarrollen en las páginas siguientes.

ESCRIBIR CÓDIGO JAVA

En primer lugar, hay que crear una clase Java, **HolaMundoN**, que declare un método nativo. También hay crear el método principal que cree el objeto **HolaMundoN** y llame al método nativo.

Las siguientes líneas de código definen la clase **HolaMundoN**, que consta de un método y un segmento estático de código:

```
class HolaMundoN {
  public native void presentaSaludo();
  static {
    System.loadLibrary( "hola" );
    }
```

Se puede decir que la implementación del método *presentaSaludo()* de la clase **HolaMundoN** está escrito en otro lenguaje, porque la palabra reservada `native` aparece como parte de la definición del método. Esta definición proporciona solamente la definición para *presentaSaludo()* y ninguna implementación para él. La implementación se le proporcionará desde un fichero fuente separado, escrito en lenguaje C.

La definición para *presentaSaludo()* también indica que el método es público, no acepta argumentos y no devuelve ningún valor. Al igual que cualquier otro método, los métodos nativos deben estar definidos dentro de una clase Java.

El código C que implementa el método *presentaSaludo()* debe ser compilado en una librería dinámica o compartida y cargado en la clase Java que lo necesite. Esta carga mapea la implementación del método nativo sobre su definición.

El siguiente bloque de código carga la librería compartida, en este caso *hola*. El sistema Java ejecutará un bloque de código estático de la clase cuando la cargue.

Todo el código anterior forma parte del fichero `HolaMundoN.java`, que contiene la clase **HolaMundoN**. Solamente falta incorporarle el método principal que llame al método nativo *presentaSaludo()*.

```
public static void main( String args[] ) {
  new HolaMundoN().presentaSaludo();
  }
```

Como el lector puede observar, se llama al método nativo del mismo modo que a cualquier otro método Java; se añade el nombre del método al final del nombre del objeto con un punto (`"."`). El conjunto de paréntesis que sigue al nombre del método encierra los argumentos que se le pasen. En este caso, el método *presentaSaludo()* no recibe ningún tipo de argumento.

En resumen, en el código Java para que se puedan incorporar métodos nativos hay que hacer dos cosas:

- Escribir la declaración de cada método nativo que se vaya a utilizar. Esto sería igual que la declaración de un método normal de una interfaz Java, incorporándole la palabra clave `native`.

  ```
  public native void presentaSaludo();
  ```

- Colocar la carga de la librería que contiene el código nativo a través de un bloque de la clase estática.

```
static {
  System.loadLibrary( "hola" );
```

Y ya se puede incorporar en el método principal la llamada que permita invocar al método nativo:

```
public static void main( String args[] ) {
  new HolaMundoN().presentaSaludo();
  }
```

COMPILAR EL CÓDIGO JAVA

Se utiliza ahora el compilador *javac* para compilar el código Java que se acaba de desarrollar, con el siguiente comando:

```
% javac HolaMundoN.java
```

CREAR EL FICHERO DE CABECERA

El siguiente paso consiste en utilizar la aplicación *javah* para conseguir el fichero de cabecera `.h`. El fichero de cabecera define una estructura que representa la clase **HolaMundoN** sobre código C y proporciona la definición de una función C para la implementación del método nativo *presentaSaludo()* definido en esa clase.

Se ejecuta *javah* sobre la clase **HolaMundoN**, con el siguiente comando:

```
% javah HolaMundoN
```

Por defecto, *javah* creará el nuevo fichero `.h` en el mismo directorio en que se encuentra el fichero `.class`, obtenido al compilar con *javac* el código fuente Java correspondiente a la clase. El fichero que creará será un fichero de cabecera del mismo nombre que la clase y con extensión `.h`. Por ejemplo, el comando anterior habrá creado el fichero `HolaMundoN.h`, cuyo contenido será el siguiente:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HolaMundoN */

#ifndef _Included_HolaMundoN
#define _Included_HolaMundoN

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Class:     HolaMundoN
 * Method:    presentaSaludo
```

```
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HolaMundoN_presentaSaludo
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

En este fichero de cabecera se puede observar que la llamada de la función C está declarada como:

```
Java_HolaMundoN_presentaSaludo()
```

Ésta es la definición de la función C que se deberá escribir para implementar el método nativo *presentaSaludo()* de la clase **HolaMundoN**. Se debe utilizar esa definición a la hora de su implementación. Si **HolaMundoN** llamase a otros métodos nativos, las definiciones de las funciones también aparecerían aquí.

El nombre de la función C que implementa el método nativo está derivado del nombre del paquete, el nombre de la clase y el nombre del método nativo. Así, el método nativo *presentaSaludo()* dentro de la clase **HolaMundoN** se denomina *HolaMundoN_presentaSaludo()*. En este ejemplo, no hay nombre de paquete porque **HolaMundoN** se considera englobada dentro del paquete por defecto, que no tiene nombre.

La función C acepta dos parámetros, aunque el método nativo definido en la clase Java no acepte ninguno. El primer parámetro es un puntero a la interfaz **JNIEnv**, a través del cual el código nativo accede a los parámetros y objetos que se le pasan desde la aplicación Java. El segundo parámetro es una referencia a la instancia actual del objeto, se puede considerar como si fuese la variable `this`. En este caso concreto, el ejemplo es tan sencillo que se ignoran los dos parámetros.

ESCRIBIR LA FUNCIÓN C

Se escribirá ahora la función C para el método nativo en un fichero fuente de código C. La implementación será una función habitual C, que luego se integrará con la clase Java. La definición de la función C debe ser la misma que la que se ha generado con *javah* en el fichero `HolaMundoN.h`.

La implementación que se propone se salva en un fichero C. Por convenio, el código nativo se coloca en un fichero con el mismo nombre de la clase Java, añadiéndole la partícula `"Imp"`. En este caso, el fichero será `HolaMundoNImp.c` y contendrá las siguientes líneas de código:

```
#include <jni.h>
#include "HolaMundoN.h"
```

```
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HolaMundoN_presentaSaludo( JNIEnv *env,jobject obj ) {
  printf( "Hola Mundo, desde C y el Tutorial de Java\n" );
  return;
  }
```

Como se puede observar, la implementación no puede ser más sencilla: hace una llamada a la función *printf()* para presentar el saludo y sale.

En el código se incluyen tres ficheros de cabecera:

- `jni.h`
  Proporciona la información para que el código C pueda interactuar con el sistema Java. Cuando se escriben métodos nativos, siempre habrá que incluir este fichero en el código fuente C.

- `HolaMundoN.h`
  Es el fichero de cabecera que se ha generado para la clase. Contiene la estructura C que representa la clase Java para la que se escribe el método nativo y la definición de la función para ese método nativo.

- `stdio.h`
  Es necesario incluirlo porque se utiliza la función *printf()* de la librería estándar de C, cuya declaración se encuentra en este fichero de cabecera.

CREAR LA LIBRERÍA DINÁMICA

Se pasa a utilizar ahora el compilador C para compilar el fichero `.h` y el fichero fuente `.c`; para crear una librería dinámica o compartida. A la hora de generarla, se utilizará el compilador C del sistema en el que se esté desarrollando, haciendo que el fichero `HolaMundoNImp.c` genere una librería dinámica de nombre ***hola***, que será la que el sistema Java cargue cuando ejecute la aplicación que se está construyendo, en este caso concreto.

A continuación, se indica la forma de generar esta librería dinámica o compartida en sistemas Windows y Unix.

Unix

Se teclea el siguiente comando:

```
% cc -G HolaMundoNImp.c -o libhola.so
```

En caso de que aparezca un error porque encuentre el compilador los ficheros de cabecera, se puede utilizar el flag `-I` para indicarle el camino de búsqueda, por ejemplo:

```
% cc -G -I$JAVA_HOME/include HolaMundoNImp.c -o libhola.so
```

donde `$JAVA_HOME` es el directorio donde se ha instalado la plataforma Java 2, o la versión que se esté utilizando del JDK.

En caso de estar ejecutando la aplicación sobre ***Linux***, o cualquier Unix que disponga del compilador de GNU, la librería se crea con `gcc`. Primero hay que compilar indicándole al compilador dónde se encuentran los ficheros fuente y, además, hay que indicarle (y éste es el *quid* de la cuestión), que genere código PIC (*Position Independent Code*). La línea de comandos sería:

```
% gcc -I/usr/local/java/include -I/usr/local/java/include/genunix
  -f PIC -c HolaMundoNImp.c
```

suponiendo que el JDK está instalado en */usr/local/java*. A continuación, se crea la librería compartida con los ficheros objeto obtenidos en la compilación, mediante el comando:

```
% gcc -shared -Wl,-soname, libhola.so.1 -o libhola.so.1.0
  HolaMundoNImp.o
```

Se copia la librería al fichero estándar con nombre corto, utilizando el siguiente comando:

```
% cp libhola.so.1.0 libhola.so
```

y, por fin, se le indica al enlazador dinámico de librerías dónde se encuentra esta nueva librería, que también debe buscar. En el *shell* `bash` habitual de Linux, el comando sería:

```
% export LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH
```

Windows

El comando a utilizar en este caso es el siguiente:

```
c:\>cl HolaMundoNImp.c /Fehola.dll -LD
```

Este comando funciona con el compilador C/C++ que forma parte de *Microsoft Visual Studio 6.0*, con otros compiladores puede ser diferente; el lector debe consultar el manual del compilador. Si se quiere indicar al compilador dónde se encuentran los ficheros de cabecera y las librerías, hay que fijar dos variables de entorno:

```
c:\>SET INCLUDE=%JAVA_HOME%\include;%INCLUDE%
c:\>SET LIB=%JAVA_HOME%\lib;%LIB%
```

donde `%JAVA_HOME%` es el directorio donde se ha instalado la plataforma Java 2, o la versión que sea del JDK.

EJECUTAR EL PROGRAMA

Y, por fin, se utilizará el intérprete de Java, *java*, para ejecutar el programa que se acaba de construir siguiendo todos los pasos anteriormente descritos. Al teclear el comando

```
> java HolaMundoN
```

en pantalla aparecerá el mensaje tan ansiosamente *esperado* de:

```
Hola Mundo, desde C y el Tutorial de Java
```

Si no aparece este mensaje de saludo y lo que aparece en pantalla es un mensaje procedente de la excepción **UnsatisfiedLinkError**, es porque no está correctamente fijado el camino de la librería dinámica que se ha generado. Este camino es la lista de directorios que el sistema Java utilizará para buscar las librerías que debe cargar. Hay que asegurarse de que el directorio donde se encuentra la librería *hola* recién creada, figura entre ellos.

Si se indica el camino correcto y se intenta ejecutar de nuevo el programa, ahora sí se obtendrá el mensaje de saludo que era de esperar.

USAR LIBRERÍAS DEL SISTEMA

Entre las razones fundamentales de utilizar JNI, como se ha indicado al comienzo del capítulo, es la existencia de aplicaciones escritas en otros lenguajes en las que se ha invertido gran cantidad de esfuerzo y dinero. La modernización de ese código para proporcionar una interfaz más sofisticada, dotarlo de acceso a Internet, etc., casi siempre requiere la completa reescritura de la aplicación en un nuevo lenguaje, circunstancia que suele ser prohibitiva en la mayoría de los casos. Además, muchas de las entidades o empresas que disponen de estas aplicaciones, no están dispuestas a migrar el código, porque son aplicaciones muy robustas, que funcionan sin problemas y que han sido perfiladas hasta en sus mínimos aspectos durante muchos años.

Utilizando JNI, además de extender las capacidades de Java como en el saludo anterior, también se puede acceder a código propietario. Por ejemplo, la aplicación `Java1601.java` utiliza la clase **NombreHost** para obtener el nombre del ordenador en que se está ejecutando la Máquina Virtual Java, accediendo para ello a la librería *kernel32.dll* del sistema operativo Windows, en este caso.

```
class NombreHost {
  public static boolean inicializar() {
    try {
      System.loadLibrary( "win32" );
      return( true );
    }catch( UnsatisfiedLinkError e ) {
      return( false );
      }
```

```
    }
  public static native String getNombreHost();
  }
```

Ésta es la parte Java de la aplicación, en donde se carga la librería y se declara como `native` el método que devuelve el nombre del ordenador. En este caso se utiliza el método *inicializar()*, en el que se controla la excepción **UnsatisfiedLinkError**, que indicaría problemas en la carga de la librería, con lo cual no se podría acceder a la función de esa librería para retornar el nombre del ordenador.

A partir del fichero fuente `win32.cpp` se crea la librería nativa propia *win32.dll* para acceder a la librería del sistema *kernel32.dll*, que dispone del método *GetComputerName()* que devuelve el nombre del ordenador; en caso de que el lector desee soportar otros sistemas como Solaris, Linux, etc., solamente necesitará indicar la librería equivalente a *kernel32.dll* que contiene la función y crear el fichero fuente en el lenguaje de la librería para poder acceder a ella.

Como observará el lector, aunque la librería que se carga es *win32.dll*, al igual que en el ejemplo anterior era *hola.dll*, la extensión `.dll` no se indica, porque el método *loadLibrary()* no admite la extensión del fichero a la hora de indicarle la librería que debe cargar. La clase **Java1601** es la que muestra el nombre del ordenador, simplemente invoca al método *inicializar()* y si consigue cargar la librería, invoca al método de la clase **NombreHost** que devuelve el nombre de la máquina.

```
class Java1601 {
  public static void main( String[] args ) {
    if( !NombreHost.inicializar() ) {
      System.out.println(
        "No se puede inicializar la extensión.." );
      System.exit( 1 );
      }
    System.out.printf( "Nombre del ordenador: %s.",
      NombreHost.getNombreHost() );
    }
  }
```

Si se compila y ejecuta en este instante la clase anterior, aparecerá en pantalla el mensaje indicando que la inicialización no ha tenido éxito. Esto sucede porque la librería *win32.dll* no existe y por ello se lanza la excepción **UnsatisfiedLinkError**.

Para crear dicha librería, se siguen los pasos indicados en el primer ejemplo del capítulo. En primer lugar se utiliza la herramienta *javah* para extraer la información de la clase y preparar un fichero de cabecera compatible con los compiladores C/C++. Se ejecuta con el comando:

```
javah NombreHost
```

En el fichero *NombreHost.h* generado se incluye el fichero *jni.h*, que es el que identifica los tipos compatibles C/C++ con sus equivalentes Java, y también el prototipo de la función *Java_Extension_getNombreHost()*, que es la función que será

llamada por la Máquina Virtual Java cada vez que se realice una llamada al método *getNombreHost()* de la clase **NombreHost**.

A continuación se crea el fichero C/C++ que se empleará para crear la librería, *win32.cpp*, el cual mantiene el código fuente de la función cuyo prototipo ha creado *javah*.

```
#include <windows.h>
#include "NombreHost.h"

JNIEXPORT jstring JNICALL Java_NombreHost_getNombreHost
  ( JNIEnv *env, jclass clase ) {

  char szName[256+1]; // Añadimos una posición por seguridad
  DWORD size = 256;

  return( (GetComputerName((LPTSTR)szName,(LPDWORD)&size) == 0) ? 0 :
    env->NewStringUTF(szName) );
  }
```

Como se puede observar, el fichero solamente contiene un método, que en este caso invoca al método correspondiente de la librería *kernel32.dll* del API de Windows, para recuperar el nombre del ordenador.

Ahora se compila del mismo modo que en el caso del *saludo* utilizando el compilador C++, generando la librería *win32.dll*, indicando los directorios correctos para *include* y *lib*, que debe colocarse en el mismo directorio que los ficheros `.class` de las clases que componen la aplicación **NombreHost** y **Java1601**.

Si ahora se ejecuta la aplicación de ejemplo con el comando:

```
java Java1601
```

se obtendrá una respuesta semejante a la que se reproduce a continuación, en donde estará, evidentemente, el nombre del ordenador en el que el lector esté ejecutando la aplicación:

```
% java Java1601
Nombre del ordenador = LEGOEIRO.
```

CAPÍTULO 17

COMUNICACIONES EN RED

Para los propósitos que persigue el Tutorial, que no pretende profundizar en cómo funcionan las redes o los puertos de un ordenador, una *red de ordenadores* es un conjunto de máquinas o dispositivos que están interconectados a través de algún medio que les permite intercambiar datos.

Cada uno de los dispositivos de la red puede considerarse como un *nodo*, y cada uno de estos nodos está identificado mediante una *dirección única*. La forma en que se asignan estas direcciones varían de un tipo de red a otro, pero en todos los casos la dirección de cada dispositivo debe ser *única*, para permitir distinguir a ese dispositivo de cualquier otro. Las direcciones son *cifras* con las cuales los ordenadores trabajan con facilidad, pero son muy difíciles de recordar para los humanos; por lo tanto, las redes también proporcionan *nombres*, que son más fáciles de retener.

Las redes actuales utilizan para la transferencia de datos un concepto conocido como *packet switching*. Los datos son encapsulados en paquetes que son transferidos desde el origen hasta el destino, en donde los datos se van extrayendo de uno o más paquetes para reconstruir el mensaje original.

Aunque el lector deba conocer todos estos parámetros para entender el funcionamiento de una red, lo cierto es que el concepto de *red* no es nada difícil, ya que se trata solamente de pasar la información de una máquina a otra, o viceversa; es decir, es semejante a la lectura y escritura de ficheros, excepto en que estos ficheros están situados en una máquina remota, y esa máquina remota puede decidir qué es lo que quiere hacer con la información que le llega. Una de las principales características de Java es precisamente su tratamiento de la red. Tanto como es posible, Java abstrae todos los detalles de manejo a bajo nivel de la red, dejándole ese trabajo a la Máquina Virtual Java. El modelo de programación que se utiliza es el de ficheros; de hecho, se

puede comparar una conexión de red (un *socket*) con un canal. Además, la capacidad de manejo de múltiples tareas que proporciona Java es muy cómoda a la hora de poder manejar múltiples conexiones a la vez.

SOCKETS

Java proporciona varias formas distintas de atacar la programación de comunicaciones a través de red, al menos en lo que a la comunicación web concierne. Por un lado están las clases **Socket**, **DatagramSocket**, **MulticastSocket** y **ServerSocket**, y por otro lado están las clases **URL**, **URLEncoder** y **URLConnection**.

Los *sockets* son puntos finales de enlaces de comunicaciones entre procesos. Los procesos los tratan como descriptores de ficheros, de forma que se pueden intercambiar datos con otros procesos transmitiendo y recibiendo a través de sockets. El tipo de socket describe la forma en que se transfieren datos a través de ese socket.

Sockets Stream (TCP)

Son un servicio orientado a conexión, donde los datos se transfieren sin encuadrarlos en registros o bloques. Si se rompe la conexión entre los procesos, éstos serán informados de tal suceso para que tomen las medidas oportunas.

El protocolo de comunicaciones con *streams* es un protocolo orientado a conexión, ya que para establecer una comunicación utilizando el protocolo TCP, hay que establecer en primer lugar una conexión entre un par de *sockets*. Mientras uno de los *sockets* atiende peticiones de conexión (*servidor*), el otro solicita una conexión (*cliente*). Una vez que los dos *sockets* estén conectados, se pueden utilizar para transmitir datos en ambas direcciones. Equivale al servicio telefónico.

Sockets Datagrama (UDP)

Son un servicio de transporte sin conexión. Son más eficientes que TCP, pero en su utilización no está garantizada la fiabilidad. Los datos se envían y reciben en paquetes, cuya entrega no está garantizada. Los paquetes pueden estar duplicados, perderse o llegar en un orden diferente al que se envió.

El protocolo de comunicaciones con *datagramas* es un protocolo sin conexión, es decir, cada vez que se envíen datagramas es necesario enviar el descriptor del *socket* local y la dirección del *socket* que debe recibir el datagrama. Como se puede observar, hay que enviar datos adicionales cada vez que se realice una comunicación, aunque tiene la ventaja de que se pueden indicar direcciones globales y el mismo mensaje llegará a muchas máquinas a la vez. Se asemeja al servicio de correos.

Sockets Raw

Son *sockets* que dan acceso directo a la capa de software de red subyacente o a protocolos de más bajo nivel. Se utilizan sobre todo para la depuración del código de los protocolos.

Programación de Sockets

La programación utilizando *sockets* involucra a dos clases principalmente: **Socket** y **DatagramSocket**, a la que se incorpora una tercera no tan empleada, **ServerSocket**, que solamente se utiliza para implementar *servidores*, mientras que las dos primeras se pueden usar para crear tanto *clientes* como *servidores*, representando comunicaciones *TCP* la primera y comunicaciones *UDP* la segunda. La comunicación *Multicast* se tratará de forma específica.

La programación con *sockets* es una aproximación de bastante bajo nivel para la comunicación entre dos ordenadores que van a intercambiar datos. Uno de ellos será el *cliente* y el otro el *servidor*, como ya se ha repetido en varias ocasiones.

Aunque la distinción entre cliente y servidor se va haciendo menos clara cada día, en Java hay una clara diferencia que es inherente al lenguaje. El cliente siempre inicia conversaciones con servidores y éstos siempre están esperando a que un cliente quiera establecer una conversación. Ya después, a nivel de aplicación, se determinará lo que sucede una vez que se establezca la conexión y se inicie la conversación.

El hecho de que dos ordenadores puedan conectarse no significa que puedan comunicarse, es decir, que además de establecerse la conexión, las dos máquinas deben utilizar un protocolo aceptado por ambas para poder entenderse. Por ejemplo, el hecho de que se pueda marcar un número de teléfono de China desde España, no da por sentado que se pueda establecer una comunicación con la persona que descuelgue al otro lado, si no se conoce el lenguaje chino; o la otra persona habla español, o los dos chapurrean una lengua común; porque sino, muy poca comunicación va a poder tener lugar aunque la conexión sea perfecta y la voz de ambos clara y sin distorsiones.

La programación de *sockets* en el mundo Unix viene desde muy antiguo, Java simplemente encapsula una gran parte de la complejidad de su uso en clases, permitiendo un acercamiento a esa programación mucho más orientado a objetos de lo que podía hacerse antes. Básicamente, la programación de *sockets* hace posible que el flujo de datos se establezca en las dos direcciones entre cliente y servidor, por ello lo de comentar que la diferencia entre cliente y servidor, que una vez establecida la conexión, se diluye. El flujo de datos que se intercambian cliente y servidor se puede considerar de la misma forma que cuando se guardan y recuperan datos de un disco: como un conjunto de bytes a través de un canal. Y como en todo proceso en el que intervienen datos, el sistema es responsable de llevar esos datos desde su origen a su destino, y es responsabilidad del programador el asignar significado a esos datos.

Y *asignar significado* tiene una especial relevancia en el caso de la utilización de *sockets*. En particular, como se ha dicho, entre las responsabilidades del programador está la implementación de un protocolo de comunicaciones que sea mutuamente aceptable entre las dos máquinas a nivel de aplicación, para hacer que los datos fluyan de forma ordenada. Un *protocolo a nivel de aplicación* es un conjunto de reglas a través de las cuales los programas que se ejecutan en los dos ordenadores pueden establecer una conversación e intercambiar datos. Por ejemplo, el lector puede escribir un programa utilizando el protocolo de correo SMTP para enviar un mensaje de correo electrónico a alguien. Aquí también se desarrollarán ejemplos que implementen de forma muy básica el protocolo HTTP para poder descargar páginas Web desde un servidor de Internet y presentarlas en pantalla, o bien un servidor que soporte el protocolo *echo* tanto para TCP como para UDP. Cada uno de estos programas involucra la aceptación y uso de un protocolo para poder entender la información que proviene o se envía a la otra máquina.

Programación de URL

La programación de **URL** se produce a un nivel más alto que la programación de *sockets* y, al menos en teoría, resulta una idea muy poderosa. Esa teoría dice que, utilizando la clase **URL**, se puede establecer una conexión con cualquier recurso que se encuentre en Internet, especificando un objeto **URL** y simplemente invocando el método *getContent()* sobre ese objeto **URL**. El contenido del recurso será descargado y aparecerá en la máquina cliente, incluso aunque requiera un protocolo que no exista cuando el programa fue escrito y ese contenido no fuese entendido en el momento de la implementación de la aplicación.

Esta descripción puede resultar un poco oscura, pero es concretamente lo que se proclama a los cuatro vientos que se puede hacer. La idea es excelente, pero habrá que esperar un poco más al futuro para recoger sus frutos. Si estuviese completamente implementada en los navegadores, esta idea significaría poder colocar un material nuevo y desconocido en un sitio web junto con los manejadores de contenido y protocolos. Cuando un navegador bajase este contenido desde el sitio web, se bajaría el manejador del contenido y podría interpretarlo sin necesidad de instalar ningún software nuevo.

La cosa varía si de lo que se trata es de una intranet, ya que si se quiere dotar a los clientes de la capacidad de manejar nuevos contenidos, será necesario proporcionar el adecuado protocolo y manejador de esos contenidos; y probablemente será necesario en más de una ocasión que los clientes ejecuten aplicaciones escritas en Java en vez de usar los navegadores estándar para acceder a los datos.

La clase **URL** también proporciona una forma alternativa de conectar un ordenador con otro y compartir datos, basándose en *streams*. En algún ejemplo se verá esta técnica, que es un tanto redundante con la programación de *sockets*.

LA CLASE INETADDRESS

Hasta ahora se ha visto la fundamentación teórica en que se basa la programación de comunicaciones a través de red, así que ha llegado la hora de presentar código Java que utilice este tipo de comunicación. El ejemplo `Java1701.java` presenta el uso de la clase **InetAddress** y cómo se emplean varios de sus métodos. Tanto en éste, como en la mayoría de los ejemplos de este capítulo, será necesario estar conectado a Internet a la hora de ejecutarlos para que funcionen correctamente; en caso contrario, se obtendrá una excepción de tipo **UnknownHostException**.

La salida que se obtiene al ejecutar el programa es la que se reproduce a continuación. Observe el lector, que al estar situado el ordenador en que se ejecuta la aplicación en una red privada, la dirección corresponde a esa red.

```
% java Java1701
-> Direccion IP de una URL, por nombre
 java.sun.com/72.5.124.55
 -> Nombre a partir de la direccion
 /72.5.124.55
 -> Direccion IP actual de LocalHost
 Breogan/192.168.1.100
 -> Nombre de LocalHost a partir de la direccion
 /192.168.1.100
 -> Nombre actual de LocalHost
 Breogan
 -> Direccion IP actual de LocalHost
 192 168 1 100
```

Y el código completo del ejemplo es el que se reproduce a continuación, que posteriormente se comentará en detalle.

```
class Java1701 {
  public static void main( String[] args ) {
    try {
      System.out.println(
        "-> Direccion IP de una URL, por nombre" );
      InetAddress address =
        InetAddress.getByName( "java.sun.com" );
      System.out.println( address );
      // Extrae la dirección IP a partir de la cadena que se encuentra
      // a la derecha de la barra /, luego proporciona esta dirección
      // IP como argumento de llamada al método getByName()
      System.out.println(
        "-> Nombre a partir de la direccion" );
      int temp = address.toString().indexOf( '/' );
      address = InetAddress.getByName(
        address.toString().substring(temp+1) );
      System.out.println( address );
      System.out.println(
        "-> Direccion IP actual de LocalHost" );
      address = InetAddress.getLocalHost();
      System.out.println( address );
      System.out.println(
        "-> Nombre de LocalHost a partir de la direccion" );
```

```
      temp = address.toString().indexOf( '/' );
      address = InetAddress.getByName(
        address.toString().substring(temp+1) );
      System.out.println( address );
      System.out.println(
        "-> Nombre actual de LocalHost" );
      System.out.println( address.getHostName() );
      System.out.println(
        "-> Direccion IP actual de LocalHost" );
      // Obtiene la dirección IP como un array de bytes
      byte[] bytes = address.getAddress();
      // Convierte los bytes de la dirección IP a valores sin signo y
      // los presenta separados por espacios
      for( int cnt=0; cnt < bytes.length; cnt++ ) {
        int uByte = bytes[cnt] < 0 ? bytes[cnt]+256 : bytes[cnt];
        System.out.print( uByte+" " );
      }
      System.out.println();
    } catch( UnknownHostException e ) {
      System.out.println( e );
      System.out.println(
        "Debes estar conectado para que esto funcione bien." );
      }
    }
  }
```

Todo el código del programa está en el método principal *main()*. La clase **InetAddress** proporciona objetos que se pueden utilizar para manipular tanto direcciones IP como nombres de dominio; sin embargo, no se pueden instanciar estos objetos directamente, representa el número actual, no el nombre o dirección IP de un ordenador. La clase proporciona varios métodos estáticos que devuelven un objeto de tipo **InetAddress**.

El método estático *getByName()* devuelve un objeto **InetAddress** representando el *host* que se le pasa como parámetro. Este método se puede utilizar para determinar la dirección IP de un *host*, conociendo su nombre; entendiendo por nombre del *host* el nombre de la máquina, como "java.sun.com", o la representación como cadena de su dirección IP, como "72.5.124.55". El método *getAllByName()* devuelve un array de objetos **InetAddress**, y se puede utilizar para determinar todas las direcciones IP asignadas a un *host*. El método *getLocalHost()* devuelve un objeto **InetAddress** representando el ordenador local sobre el que se ejecuta la aplicación.

El primer trozo de código interesante es el que obtiene un objeto **InetAddress** representando un determinado servidor y presenta esta dirección utilizando el método sobrecargado *toString()* de la clase **InetAddress**.

```
InetAddress address = InetAddress.getByName( "java.sun.com" );
System.out.println( address );
```

El siguiente trozo de código es la acción contraria al anterior, en que se proporciona la dirección IP para presentar el nombre del *host*. Como ya se ha indicado, el método *getByName()* puede aceptar como parámetro de entrada tanto el

nombre del *host*, como su dirección IP en forma de cadena. El código utiliza el resultado de la llamada al método anterior para construir una cadena con la parte numérica del resultado, que es pasada al método *getByName()*.

```
int temp = address.toString().indexOf( '/' );
address =
  InetAddress.getByName( address.toString().substring(temp+1) );
System.out.println( address );
```

El término **localhost** se utiliza para describir el ordenador local, la máquina en la que se está ejecutando la aplicación. Cuando se conecta a una red IP, el ordenador local debe tener una dirección IP, que puede conseguir de diferentes formas; no obstante, la siguiente explicación se basa en que el ordenador sobre el cual se ejecuta la aplicación se conecta a Internet a través de un proveedor de servicios.

El proveedor de Internet tiene reservadas una serie de direcciones IP, que puede compartir entre todos sus clientes. Cuando alguien se conecta al proveedor, automáticamente se le asigna una dirección a esa conexión, válida durante todo el tiempo que dure la sesión. Si se produce una desconexión y luego se vuelve a conectar, lo más seguro es que la dirección IP no sea la misma que se había asignado a la primera conexión.

Aunque ésa sea la situación más habitual del lector, en otras ocasiones puede ser diferente. Por ejemplo, si el ordenador se encuentra en la red interna de ordenadores de una empresa, esa empresa puede tener un bloque de direcciones IP reservadas y asignar *permanentemente* las direcciones a los ordenadores; en cuyo caso, cada vez que se ejecute el programa, la dirección IP será siempre la misma. También es posible que el lector disponga de su propia dirección *permanente* IP y nombre de dominio.

En cualquier caso, el método *getLocalHost()* se puede utilizar para obtener un objeto de tipo **InetAddress** que represente al ordenador en el cual se está ejecutando la aplicación. Eso es justamente lo que muestra el código que aparece a continuación.

```
address = InetAddress.getLocalHost();
System.out.println( address );
```

En este caso, la parte numérica que aparece al ejecutar el programa, corresponde a la dirección que el proveedor de Internet ha asignado a la conexión sobre la cual se estaba ejecutando el programa.

Las líneas de código siguientes realizan la operación contraria a las anteriores; se utiliza el método *getByName()* para determinar el nombre por el cual el servidor de nombres de dominio reconoce a esa dirección numérica.

```
temp = address.toString().indexOf( '/' );
address =
  InetAddress.getByName( address.toString().substring(temp+1) );
System.out.println( address );
```

Una vez que se ha obtenido el objeto **InetAddress**, hay otra serie de métodos de la clase **InetAddress** que se pueden utilizar para invocar a este objeto; por ejemplo, las siguientes líneas de código muestran la invocación del método *getHostName()*, para obtener el nombre de la máquina.

```
System.out.println( address.getHostName() );
```

De forma semejante, el siguiente fragmento de código utiliza la invocación del método *getAddress()* para obtener un array de bytes conteniendo la dirección IP de la máquina.

```
byte[] bytes = address.getAddress();
// Convierte los bytes de la dirección IP a valores sin
// signo y los presenta separados por espacios
for( int cnt=0; cnt < bytes.length; cnt++ ) {
  int uByte = bytes[cnt] < 0 ? bytes[cnt]+256 : bytes[cnt];
  System.out.print( uByte+" " );
  }
```

Como no existe nada parecido a un byte sin signo en Java, la conversión del array de bytes en algo que se pueda presentar en pantalla requiere una cierta manipulación de los bytes, tal como se hace en el bucle `for` del código anterior.

LA CLASE NETWORKINTERFACE

Las versiones anteriores de la plataforma Java 2 a la J2SE 6, disponían de soporte limitado para obtener información relacionada con múltiples conexiones de red, restringida a los datos proporcionados por la clase anterior **InetAddress**. Ahora, la plataforma Java dispone de la clase **NetworkInterface**, que proporciona nuevos métodos para recuperar datos correspondientes a las interfaces de red.

La aplicación `Java1718.java` utiliza la clase **NetworkInterface** para acceder a las redes y presentar información relativa a cada una de ellas. Ejecutando la aplicación en una máquina Linux típica que monte *Fedora Core*, el resultado sería el siguiente:

```
>java Java1718
Interfaz de Red: eth0
Nombre: eth0
Dirección Inet: /fe80:0:0:0:203:ffff:f3d4:ee3b%2
Dirección Inet: /192.168.2.242
Padre: null
Levantada: true
Bucle: false
Punto-a-punto: false
Soporte Multicast: false
Virtual: false
Dirección hardware: [0, 3, -1, -44, -18, 59]
MTU: 1500
Dirección Interfaz: /fe80:0:0:0:203:ffff:f3d4:ee3b%2
Dirección Interfaz: /192.168.2.242

Interfaz de Red: lo
```

```
Nombre: lo
Dirección Inet: /0:0:0:0:0:0:0:1%1
Dirección Inet: /127.0.0.1
Padre: null
Levantada: true
Bucle: true
Punto-a-punto: false
Soporte Multicast: false
Virtual: false
Dirección hardware: [0, 3, -1, -44, -18, 59]
MTU: 16436
Dirección Interfaz: /0:0:0:0:0:0:0:1%1
Dirección Interfaz: /127.0.0.1
```

Aunque, evidentemente, los nombres y las direcciones serán distintos en función de la máquina en que se ejecute el ejemplo y la configuración de que disponga. Por ejemplo, las máquinas *Windows* probablemente presenten nombres semejantes aunque con distintos nombres descriptivos y posiblemente direcciones diferentes.

Los métodos que proporciona la clase **NetworkInterface** permiten obtener información básica, por ejemplo, si la red está *levantada* con *isUp()*, si se trata de una interfaz punto-a-punto con *isPointToPoint()* o si se trata de una interfaz virtual con *isVirtual()*.

También dispone de métodos que proporcionan información de otros parámetros de la interfaz de red, como puede ser la dirección física hardware o el tamaño máximo de los paquetes que puede admitir.

La clase **Java1718** presenta información más completa que **InetAddress**. El resultado dependerá de la configuración del sistema en que se ejecute y habrá información que no estará accesible por razones de seguridad, si el usuario con que se lance la aplicación no dispone de privilegios para acceder a dicha información.

LA CLASE URL

La clase **URL** contiene constructores y métodos para la manipulación de URL (*Universal Resource Locator*): un objeto o servicio en Internet. El protocolo TCP, como ya se ha indicado, necesita dos tipos de información: la dirección IP y el número de puerto. A continuación, se expone la forma en que se recibe la página Web principal de uno de los buscadores al teclear:

```
http://www.yahoo.com
```

En primer lugar, *Yahoo* tiene registrado su nombre, permitiendo que se use *yahoo.com* como su dirección IP, o lo que es lo mismo, cuando se indica *yahoo.com* es como si se hubiese especificado 205.216.146.71, su dirección IP real.

La verdad es que la cosa es un poco más complicada que eso, al intervenir el servicio DNS (*Domain Name Service*), que traslada *www.yahoo.com* a

205.216.146.71, que es realidad lo que permite teclear *www.yahoo.com*, en lugar de tener que recordar su dirección IP.

Como se ha visto al tratar la clase **InetAddress**, si se quiere obtener la dirección IP real de la red en la cual se está ejecutando una aplicación, se pueden realizar llamadas a los métodos *getLocalHost()* y *getAddress()*. Primero, *getLocalHost()* devuelve un objeto **InetAddress**, que si se usa con *getAddress()* generará un array con la dirección IP.

Una cosa interesante en este punto es que *una* red puede mapear *muchas* direcciones IP. Esto puede ser necesario para un servidor Web, como *Yahoo*, que tiene que soportar grandes cantidades de tráfico y necesita más de una dirección IP para poder atender a todo ese tráfico. El nombre interno para la dirección 205.216.146.71, por ejemplo, es *www7.yahoo.com*. El DNS puede trasladar una lista de direcciones IP asignadas a *Yahoo* en *www.yahoo.com*.

Se conoce pues la dirección IP, ahora falta conocer el número del puerto. Si no se indica nada, se utilizará el que se haya definido por defecto en el fichero de configuración de los servicios del sistema. En Unix se indican en el fichero `/etc/services`, en WindowsNT en el fichero `services` y en otros sistemas puede ser diferente.

El puerto habitual de los servicios Web es el 80, así que, si no se indica un puerto específico, la entrada en el servidor de *Yahoo* se realiza por el puerto 80. Si se teclea la URL siguiente en un navegador

```
http://www.yahoo.com:80
```

también se recibirá la página principal de *Yahoo*. No hay nada que impida cambiar el puerto en el que residirá el servidor Web; sin embargo, el uso del puerto 80 es casi estándar, porque elimina pulsaciones en el teclado y, además, las direcciones URL son lo suficientemente difíciles de recordar como para añadirle encima el número del puerto.

Si se necesita otro protocolo, como

```
ftp://ftp.sun.com
```

el puerto a utilizar se derivará de ese protocolo. Así el puerto FTP de *ftp* es el 21, según el fichero `services`. La primera parte, antes de los dos puntos, de la URL, indica el protocolo que se quiere utilizar en la conexión con el servidor. El protocolo HTTP (*HyperText Transmission Protocol*), es el utilizado para manipular documentos Web. Y si no se especifica ningún documento, muchos servidores están configurados para devolver un documento de nombre *index.html*.

Con todo esto, Java permite los siguientes cuatro constructores para la clase **URL**:

```
public URL( String spec )
public URL( String protocol,String host,int port,String file )
public URL( String protocol,String host,String file )
  throws MalformedURLException;
public URL( URL context,String spec )
```

Así se podrían especificar todos los componentes de una dirección URL como en:

```
URL( "http","www.yahoo.com","80","index.html" );
```

o dejar que los sistemas utilicen los valores por defecto definidos, como en:

```
URL( "http://www.yahoo.com" );
```

y en los dos casos se obtendría la descarga de la página principal de *Yahoo* en la aplicación desde la cual se haya invocado.

El ejemplo `Java1702.java` muestra cómo se utilizan los cuatro constructores de la clase **URL** y los métodos que proporciona dicha clase. En el programa también se muestra la utilización de la clase **URLEncoder** para convertir una cadena que contenga espacios en otra que se encuentre en el formato *x-www-form-urlencoded*.

```
class Java1702 {
  public static void main( String[] args ) {
    Java1702 obj = new Java1702();
    try {
      System.out.println(
        "Constructor simple para URL principal" );
      obj.display( new URL( "http://java.sun.com" ) );
      System.out.println(
        "Constructor de cadena para URL + directorio" );
      obj.display( new URL( "http://java.sun.com/docs" ) );
      System.out.println(
        "Constructor con protocolo, host y directorio" );
      obj.display( new URL(
        "http","java.sun.com","/docs" ) );
      System.out.println(
        "Constructor com protocolo, host, puerto y directorio" );
      obj.display( new URL(
        "http","java.sun.com",80,"/docs" ) );
      System.out.println(
        "Construye una direccion absoluta a partir de la \n"+
        "direccion del Host y una URL relativa" );
      URL baseURL = new URL(
        "http://java.sun.com/docs/index.html");
      obj.display( new URL( baseURL,
        "/docs/books/tutorial/index.html" ) );
      System.out.println(
        "Uso de URLEncoder para crear una cadena x-www-form-url" );
      System.out.println( URLEncoder.encode(
        "http://espacio .tilde~.mas+.com","ISO-8859-1" ) );
    } catch( Exception e ) {
      System.out.println( e );
      }
    }
```

```
  // Método para poder presentar en la pantalla partes de una
  // dirección URL
  void display( URL url ) {
    System.out.print( url.getProtocol()+" " );
    System.out.print( url.getHost()+" " );
    System.out.print( url.getPort()+" " );
    System.out.print( url.getFile()+" " );
    System.out.println( url.getRef() );
    // Presentamos la dirección completa como una cadena
    System.out.println( url.toString() );
    System.out.println();
    }
  }
```

El último método que aparece en el código del ejemplo, *display()*, es aquel que ilustra el uso de algunos de los métodos de la clase **URL**, y también sirve como forma práctica de ver la información que contiene un objeto **URL**. Este método recibe un objeto **URL** como parámetro y presenta todos sus componentes separados por un espacio en blanco; finalmente, utiliza el método sobrescrito *toString()* de la clase **URL** para presentar el contenido del objeto **URL** como un único objeto **String**.

Como se puede observar, hay un método para obtener cada una de las partes que componen una dirección URL, exceptuando el método *getFile()*, que devuelve el directorio y el nombre del fichero combinados.

El resto del código de la clase se encuentra en el método *main()*. El primer fragmento de código interesante es el que ilustra la instanciación de un objeto de tipo **URL** utilizando la versión del constructor que espera recibir la dirección URL como una cadena. En este caso se ha eliminado el código de manejo de excepciones para no liar el ejemplo, pero el lector podrá observarlo en posteriores programas.

```
Java1702 obj = new Java1702();
try {
  System.out.println(
    "Constructor simple para URL principal" );
  obj.display( new URL( "http://java.sun.com" ) );
. . .
```

La primera línea de código es la instanciación de un objeto de control de la clase para poder acceder al método *display()*. Luego se instancia un nuevo objeto **URL** utilizando la versión que admite como parámetro una cadena, y se le pasa al método *display()*, que va a presentar todos sus componentes en pantalla. La salida de este fragmento de código en pantalla es la siguiente:

```
http java.sun.com -1 / null
http://java.sun.com/
```

El `-1` que aparece es para indicar que no se ha especificado el puerto, y `null` es para indicar que no se ha indicado ninguna página o fichero en el constructor del objeto **URL**.

El programa presenta a continuación otras formas de construir el objeto **URL**, que no tienen mayor trascendencia, excepto uno de ellos, que resulta muy interesante, y es el constructor que necesita dos parámetros: una dirección URL y una cadena; y su interés radica en que permite construir tanto una dirección URL *absoluta* como *relativa*. Asúmase que el interés se centra en escribir un método para presentar ficheros HTML para que sean interpretados por un navegador, en vez de presentarlos como simples ficheros de texto. Es normal que estos ficheros contengan enlaces a direcciones *URL relativas*, es decir, el enlace que se proporciona toma como referencia la dirección URL base en donde reside el fichero HTML.

El siguiente fragmento de código utiliza este constructor para combinar la dirección base URL y la dirección relativa del fichero que se desea para generar una nueva dirección que apunta directamente a ese fichero.

```
URL baseURL = new URL( "http://java.sun.com/docs/index.html");
obj.display( new URL( baseURL,"/docs/books/tutorial/index.html" ) );
```

La salida que generan esas líneas de código se muestra a continuación, en donde se puede comprobar la construcción de la dirección relativa que se ha indicado para convertirla en la dirección absoluta de la página, es perfecta.

```
http java.sun.com -1 /docs/books/tutorial/index.html null
http://java.sun.com/docs/books/tutorial/index.html
```

Y las últimas líneas de código interesantes son las finales, en que se utiliza la clase **URLEncoder**. Esta clase intenta ayudar al programador en todos los problemas que causan los *espacios*, *caracteres especiales*, *caracteres no-alfanuméricos*, etc., que utilizan algunos sistemas operativos y que pueden hacer que los nombres de ficheros no lleguen a convertirse en una dirección URL válida.

Si se desea crear un objeto **URL** utilizando una cadena URL que tiene los problemas que se han indicado, primero es necesario utilizar el método *encode()* de la clase **URLEncoder**, para convertirla a una cadena URL ya entendible. Lo que hace este método estático *encode()* es convertir la cadena que se le pasa a un formato llamado *x-www-from-urlencoded* y la devuelve como un objeto **String**.

Para realizar la conversión, todos los caracteres se van examinando uno a uno, de izquierda a derecha. Los caracteres ASCII de la 'a' a la 'z', de la 'A' a la 'Z' y del '0' al '9', permanecen igual. Los espacios en blanco ' ' se convierten a un signo más, '+', El resto de los caracteres se convierten a una cadena de 3 caracteres de la forma "%*xy*", donde *xy* son los dos dígitos en hexadecimal que representan los 8 bits más significativos del carácter.

El siguiente fragmento de código muestra cómo se realiza esta conversión:

```
System.out.println(
  URLEncoder.encode( "http://espacio .tilde~.mas+.com" ) );
```

En la pantalla, la cadena en el formato que se ha descrito aparecerá como:

```
http%3A%2F%2Fespacio+.tilde%7E.mas%2B.com
```

Como bien puede observar el lector, los caracteres especiales han sido sustituidos por su valor hexadecimal precedido del signo del tanto por ciento, excepto el espacio que se ha sustituido por el signo más y el carácter *más*, que se ha sustituido por su valor hexadecimal, `%2B`.

Un buen ejercicio para el lector, si decide aceptarlo, es crear una clase **URLDecoder**, que no presenta demasiadas dificultades, y que en su creación puede ayudar mucho a la comprensión de la forma en que se utilizan las direcciones URL y las consecuencias que se derivan de la inclusión de los caracteres especiales de idiomas diferentes al inglés.

LA CLASE URLCONNECTION

Es una clase *abstracta* que puede ser *extendida*, con un constructor protegido que admite un objeto **URL** como parámetro. Tiene unas ocho variables que contienen información muy útil sobre la conexión que se haya establecido y cerca de cuarenta métodos que se pueden utilizar para examinar y manipular el objeto que se crea con la instanciación de la clase.

Si el lector pretende utilizar la clase **URL** por las capacidades de alto nivel que proporciona, y su intención es escribir manejadores de protocolos o cosas así, es muy probable que tenga que conocer en profundidad esta clase. Pero el objetivo que se pretende aquí no es llegar a ese extremo, sino simplemente que el lector tenga conocimiento de la existencia de esta clase y de alguno de sus métodos.

El ejemplo `Java1703.java` muestra cómo se realiza la conexión a una dirección URL y se crea un objeto de tipo **URLConnection**. El programa usa entonces ese objeto para obtener y presentar en pantalla algunos de los aspectos de alto nivel de la dirección URL, como son el tipo de contenido, la fecha de la última modificación o la propia dirección URL. Por ejemplo, la última ejecución del programa realizada por el autor generó la siguiente salida por pantalla:

```
% java Java1703
http://java.sun.com/docs/index.html
Wed Apr 17 18:26:22 GMT 2008
text/html
. . . .  seguido del código html de la página
```

El código completo del ejemplo es el que se reproduce a continuación.

```
class Java1703 {
  public static void main( String[] args ) {
    String cadena;
    try {
      // Creamos un objeto de tipo URL
```

```
      URL url = new URL(
         "http://java.sun.com/docs/index.html" );
      // Se abre una conexión hacia la dirección recogiéndola en un
      // objeto de tipo URLConnection
      URLConnection conexion = url.openConnection();
      // Se utiliza la conexión para leer y presentar la dirección
      System.out.println( conexion.getURL() );
      // Se utiliza la conexión para leer y presentar la fecha de
      // última modificación
      Date fecha = new Date( conexion.getLastModified() );
      System.out.println( fecha );
      // Se utiliza la conexión para leer y presentar el tipo de
      // contenido
      System.out.println( conexion.getContentType() );
      // Se utiliza la conexión para conseguir un objeto de tipo
      // InputStream, que luego se emplea para instanciar un
      // objeto DataInputStream
      BufferedReader paginaHtml =
        new BufferedReader( new InputStreamReader(url.openStream()) );
      // Se utiliza el objeto DataInputStream para leer y presentar el
      // fichero línea a línea
      while( (cadena = paginaHtml.readLine()) != null ) {
        System.out.println( cadena );
      }
    } catch( UnknownHostException e ) {
      e.printStackTrace();
      System.out.println(
        "Debes estar conectado para que esto funcione bien." );
    } catch( MalformedURLException e ) {
      e.printStackTrace();
    } catch( IOException e ) {
      e.printStackTrace();
      }
    }
  }
```

La clase **URLConnection**, como se ha indicado, es abstracta, por lo que no se puede instanciar directamente, pero sí se puede extender. Una forma habitual de conseguir un objeto de tipo **URLConnection** es invocar un método sobre un objeto **URL** que devuelva un objeto de una subclase de la clase **URLConnection**.

```
URL url = new URL( "http://java.sun.com/docs/index.html" );
```

El primer trozo de código que merece la pena, representado en la línea anterior, es la instanciación de un objeto **URL**. La línea de código que se muestra ahora crea un objeto de tipo **URLConnection** invocando al método *openConnection()* del objeto que se ha instanciado antes.

```
URLConnection conexion = url.openConnection();
```

Sólo queda como interesante el fragmento de código que se usa para invocar a tres de los métodos de la clase **URLConnection**, sobre el objeto instanciado, para obtener la información de alto nivel que muestra la dirección URL, la última fecha en que se ha modificado y el tipo de contenido del fichero al que apunta esa dirección URL.

```
System.out.println( conexion.getURL() );
Date fecha = new Date( conexion.getLastModified() );
System.out.println( fecha );
System.out.println( conexion.getContentType() );
```

La verdad es que la información puede no ser demasiado fiable a veces, porque la impresión que da es que, por ejemplo, el tipo de contenido está basado en la extensión del fichero, y la fecha de última modificación corresponde a la capacidad que pueda tener el sistema operativo para mostrar la fecha a la hora de requerir el contenido de un directorio. No obstante, este tipo de información puede ser útil para mantener una utilización correcta del fichero apuntado por la dirección URL. De nuevo, se recuerda al lector que la clase **URLConnection** dispone de más métodos para proporcionar información relevante a la conexión que se haya establecido.

LA CLASE SOCKET

La programación de *sockets* proporciona un mecanismo de muy bajo nivel para la comunicación e intercambio de datos entre dos ordenadores, uno considerado como *cliente*, que es el que inicia la conexión con el otro, *servidor*, que está a la espera de conexiones. Un objeto **Socket** es la representación Java de una conexión TCP.

El protocolo de comunicación entre ambos determinará lo que suceda tras el establecimiento de la conexión. Para que las dos máquinas puedan entenderse, ambas deben implementar un protocolo conocido por la dos. En la programación de *sockets*, la comunicación es *full-duplex*, en ambos sentidos a la vez, entre cliente y servidor; siendo responsabilidad del sistema el llevar los datos de una máquina a otra, dejando al programador el proporcionar significado a esos datos. Parte de la información que fluye entre las dos máquinas es, pues, para implementar el protocolo, y el resto son los propios datos que se quieren transferir.

Es muy sencilla la utilización de sockets para establecer la comunicación entre cliente y servidor; en realidad, no es más complicada que lo que pueda serlo el escribir datos en un fichero. Enviar y recuperar los datos que se intercambian es la parte fácil del asunto; porque más allá de esto ya se encuentra el protocolo de comunicación que debe ser entendido por cliente y servidor, que en caso de ser necesario implementarlo, se convierte en algo verdaderamente complicado.

Cliente Eco

El primer ejemplo de programación de sockets que se presenta, `Java1704.java`, implementa un cliente que realiza una prueba de eco con un servidor, enviándole una línea de texto por el puerto 7 del servidor, destinado a estos menesteres.

Para poder ejecutar este programa es necesario estar conectado en una red, o a Internet, aunque es posible que también pueda hacerse sobre el mismo ordenador si el lector usa un sistema operativo que lo soporte, Linux por ejemplo; en cualquier otro

caso, el resultado cuando se lleve a cabo la ejecución del programa será una triste **UnknownHostException**.

El programa comienza instanciando un objeto de tipo **String** que contiene el nombre del servidor que se va a utilizar, seguido por la declaración e inicialización de la variable que guarda el número del puerto, que en el caso del servidor estándar de *echo* es el 7. A continuación, se abren los canales de entrada y salida desde el socket que se mapean en las clases **Reader** y **Writer**.

Una vez que se ha establecido la conexión y los canales de entrada y salida están listos para ser usados, el programa envía una línea de texto al puerto de eco del servidor. Esto hace que el servidor reenvíe esa misma línea de vuelta al cliente, y el programa la leerá y presentará en pantalla.

El código completo del ejemplo es el que se muestra a continuación.

```
class Java1704 {
  public static void main( String[] args ) {
    String servidor = "www.rediris.es";
    int puerto = 7;    // puerto echo
    try {
      // Abrimos un socket conectado al servidor y al puerto estándar
      // de echo
      Socket socket = new Socket( servidor,puerto );
      // Conseguimos el canal de entrada
      BufferedReader entrada =
        new BufferedReader( new InputStreamReader(
        socket.getInputStream() ) );
      // Conseguimos el canal de salida
      PrintWriter salida =
        new PrintWriter( new OutputStreamWriter(
        socket.getOutputStream() ),true );
      // Enviamos una línea de texto al servidor
      salida.println( "Prueba de Eco" );
      // Recuperamos la línea devuelta por el servidor y la presentamos
      // en pantalla
      System.out.println( entrada.readLine() );
      // Cerramos el socket
      socket.close();
    } catch( UnknownHostException e ) {
      e.printStackTrace();
      System.out.println(
        "Debes estar conectado para que esto funcione bien." );
    } catch( IOException e ) {
      e.printStackTrace();
      }
    }
  }
```

En caso de que se seleccione un servidor que no tenga soporte para el protocolo *echo* en el puerto 7, la salida que generará el programa será diferente de la que cabría esperar. Por ejemplo, si el lector intenta ejecutar este programa contra el servidor codificado, la respuesta que obtendrá será un rechazo de la conexión, algo así como

"*connection refused*" o incluso un mensaje de exceso de tiempo para el establecimiento de la conexión "*connection timed out*", generados a través de la excepción **ConnectException**, porque el puerto 7 no está abierto en ese servidor. Es difícil encontrar servidores en Internet con puertos abiertos que no sean los estrictamente necesarios (el puerto 7, *echo*, evidentemente no lo es), por lo que será necesario utilizar un servidor de la propia red local o conectarse contra el servidor de Eco que se implementará en un ejemplo posterior.

Como se puede ver en el listado, el método *main()* es el que contiene todo el código de la clase, es muy sencillo y no merece la pena desglosarlo. Las dos primeras líneas que hay que ver del ejemplo son las que contienen las variables a las que se asigna el nombre del servidor al que se va a conectar y el número del puerto de ese servidor que se va a atacar.

```
String servidor = "www.rediris.es";
int puerto = 7;    // puerto echo
```

El resto del programa se encuentra englobado en un bloque `try-catch` para tratar las excepciones. La línea siguiente es la clave del programa, en ella se establece una conexión con el puerto indicado del servidor que se ha designado para instanciar un nuevo objeto de tipo **Socket**

```
Socket socket = new Socket( server,port );
```

Una vez que este objeto existe, es posible realizar la comunicación con el servidor utilizando el protocolo que tenga predefinido para el servicio que proporciona a través de ese puerto. Lo siguiente a destacar en el código del ejemplo es el uso de las clases **Reader** y **Writer**.

```
// Conseguimos el canal de entrada
BufferedReader entrada = new BufferedReader( new InputStreamReader(
  socket.getInputStream() ) );
// Conseguimos el canal de salida
PrintWriter salida = new PrintWriter( new OutputStreamWriter(
  socket.getOutputStream() ),true );
```

El parámetro `true` de la última línea hace que el canal de salida realice una descarga o limpieza de su contenido automáticamente. La realización de esta descarga o vaciado automático de los canales, es algo de vital importancia a la hora de la programación con sockets.

Lo que resta del código es solamente el uso del canal de salida para enviar la línea de texto al servidor de eco, la captura de su respuesta a través del canal de entrada y el cierre del socket; una vez presentada la cadena devuelta por el servidor.

```
salida.println( "Prueba de Eco" );
System.out.println( entrada.readLine() );
socket.close();
```

Y esto es todo lo que hay sobre sockets desde el punto de vista de la parte cliente, no piense el lector que la complejidad es mucho mayor, se haga lo que se haga, tal como podrá comprobar en los ejemplos siguientes. Ahora bien, sí que la programación de *sockets* se puede complicar, pero todos los problemas van a venir asociados a la necesidad de implementar un protocolo a nivel de aplicación para comunicarse correctamente con el servidor, no del uso en sí de los sockets.

Cliente Fecha

El ejemplo `Java1705.java` también implementa un cliente, pero en este caso para atacar el puerto que proporciona el día y la hora sobre un servidor que dé soporte a este protocolo. El ejemplo es incluso más sencillo que el ejemplo `Java1704.java`, ya que no es necesario enviar nada al servidor para obtener respuesta, todo lo que hay que hacer es establecer la conexión. El código por tanto es bien simple, tal como se muestra en el listado completo del programa.

```
class Java1705 {
  public static void main( String[] args ) {
    String servidor = "time-a.nist.gov";
    int puerto = 13;             // puerto de daytime
    try {
      // Abrimos un socket conectado al servidor y al
      // puerto estándar de echo
      Socket socket = new Socket( servidor,puerto );
      System.out.println( "Socket Abierto." );
      // Conseguimos el canal de entrada
      BufferedReader entrada = new BufferedReader(
        new InputStreamReader( socket.getInputStream() ) );
      entrada.readLine(); // Saltamos la línea en blanco
      System.out.println( "Hora actual en Maryland (USA):" );
      System.out.println( "\t"+entrada.readLine() );
      System.out.println( "Hora Actual en Canarias, Spain:" );
      System.out.println( "\t"+new Date() );
      // Cerramos el socket
      socket.close();
    } catch( UnknownHostException e ) {
      System.out.println( e );
      System.out.println(
        "Debes estar conectado para que esto funcione bien." );
    } catch( IOException e ) {
      System.out.println( e );
      }
    }
  }
```

El programa obtiene y presenta la hora que envía el servidor de tiempo del *Observatorio Naval de la Marina de Estados Unidos* en Maryland, y luego presenta la fecha y hora donde reside el autor (o donde el lector esté ejecutando el programa), por motivos de comparación solamente. La salida que se obtiene, por ejemplo, puede ser:

```
% java Java1705
Socket Abierto.
```

```
Hora actual en Maryland (USA):
        54571 08-04-15 16:20:03 50 0 0 571.5 UTC(NIST) *
Hora Actual Local:
        Tue Apr 15 17:20:18 BST 2008
```

Como se puede comprobar, hay diferencias en la fecha debido a las zonas en que se está obteniendo esa fecha. También se puede ver que hay una diferencia de varios segundos entre una hora y otra, el autor sincronizó el reloj de su ordenador con el servidor de tiempo de **bernina.ethz.ch**, y con el convencimiento de que la hora del servidor del Observatorio naval es igual de correcta, esa diferencia solamente puede ser atribuible a la duración de la transmisión entre el servidor y el cliente.

La filosofía del programa es muy similar a la del ejemplo del cliente de Eco. Se comienza con la instanciación de un objeto **String** para contener el nombre del servidor que va a servir la fecha, y la declaración de inicialización de la variable que va a contener el número del puerto de ese servidor que se va a atacar, en este caso el puerto 13, que es el puerto estándar para el servicio *daytime*.

Luego el programa se agencia un canal de entrada a través de la conexión *socket* que ha establecido previamente y ya está listo, porque en este caso, la sola conexión es suficiente para que el servidor envíe la fecha y la hora. Así que todo lo que hay que hacer es leer la línea de entrada que contiene esa fecha y hora enviada por el servidor. El programa la presenta y también presenta la fecha y hora actuales de la máquina en que se está ejecutando el programa, para establecer una comparación con la que se ha llegado a través del *socket*.

Como no hay diferencias significativas con el ejemplo `Java1704.java` que vayan más allá del uso del método *System.out.println()*, no merece la pena detenerse a contar nada del programa, es suficiente con que el lector lea los comentarios del código.

Cliente HTTP

El ejemplo que se va a ver ahora, `Java1706.java` es un navegador extremadamente simple; o más correctamente, el programa es un cliente HTTP muy simple, implementado mediante *sockets*. El programa implementa el suficiente protocolo HTTP para poder obtener un fichero desde cualquier servidor Web. Si el lector desea realizar un navegador más útil, también tendrá que invertir mucho más tiempo, pero para ver el funcionamiento, es suficiente con lo que se muestra en este ejemplo, cuyo código fuente se muestra a continuación.

```
class Java1706 {
  public static void main( String[] args ) {
    String servidor = "java.sun.com";          // servidor
    int puerto = 80;                           // puerto
    try {
      // Crea un socket conectado al servidor y puerto que se indica
      Socket socket = new Socket( servidor,puerto );
      // Crea el canal de entrada desde el socket
      BufferedReader entrada = new BufferedReader(
```

```
        new InputStreamReader( socket.getInputStream() ) );
      // Crea el canal de salida
      PrintWriter salida = new PrintWriter(
        new OutputStreamWriter( socket.getOutputStream() ),true );
      // Envía un comando GET al servidor
      salida.println( "GET /docs/index.html" );
      // En esta cadena se va a ir leyendo el fichero
      String linea = null;
      // Bucle para leer y presentar líneas hasta que se reciba null
      while( (linea = entrada.readLine()) != null )
        System.out.println( linea );
      // Se cierra el socket
      socket.close();
    } catch( UnknownHostException e ) {
      e.printStackTrace();
      System.out.println(
        "Debes estar conectado para que esto funcione bien." );
    } catch( IOException e ) {
      e.printStackTrace();
      }
    }
  }
```

El programa se inicia definiendo el servidor y el puerto con el que se va a establecer la conexión, abriendo a continuación un *socket* sobre ese puerto. Al igual que en los ejemplos anteriores, el programa crea canales de entrada y salida para la transferencia de información entre la parte del cliente y la parte del servidor.

El programa envía un comando **GET**, actuando como cliente, al servidor, indicándole el fichero que desea recibir. Este comando forma parte del protocolo HTTP, que provoca la búsqueda en el servidor del fichero indicado y su envío al cliente. Si el servidor está montado sobre el puerto con el que se ha establecido la conexión, siempre enviará algo; si encuentra el fichero, enviará su contenido, y si no lo encuentra o hay algún otro problema, siempre envía un mensaje de error indicando la circunstancia por la cual el fichero no ha podido ser transferido al cliente.

El programa lee el texto que recibe por el canal de entrada y lo presenta en el dispositivo estándar de salida, por lo que se verá el código HTML que forma la página, es decir, se verá como texto normal.

Si el lector revisa el código del programa Java, recordará de ejemplos anteriores toda la parte inicial de declaración de variables para determinar el servidor y puerto al que conectarse y la apertura de *sockets* para los canales de entrada y salida. Así que la primera parte de código que realmente merece la pena revisar es el envío de comandos HTTP al servidor.

```
salida.println( "GET /docs/books/tutorial/index.html" );
```

Observe el lector que el árbol de directorios que se indica al servidor para que consiga localizar el fichero es relativo a su *pseudoraíz*, es decir, relativo al nivel inicial

en que se encuentran los documentos HTML. La petición se realiza simplemente escribiendo el comando en el canal de salida.

Si el comando **GET** ha sido enviado correctamente, hay que esperar siempre una respuesta del servidor, aunque se trate de un mensaje de error. El siguiente fragmento de código va leyendo texto del canal de entrada, presentándolo a continuación en la pantalla, cerrando el *socket* cuando ya no hay más datos que leer.

```
String linea = null;
while( (linea = entrada.readLine()) != null )
  System.out.println( linea );
socket.close();
```

El resto del código es ya la parte conocida de control de las excepciones. Si el lector desea incorporar más características a este ejemplo, a fin de convertirlo en un navegador útil; se le sugiere que revise las especificaciones HTML y escriba un método de presentación al estilo que normalmente se asocia a las páginas Web. E incluso puede englobar todo el ejemplo en AWT, proporcionando una interfaz gráfica mediante un área de texto, **TextArea**, para la parte de presentación de la página y un campo de texto, **TextField**, para especificar la dirección del fichero del servidor que se desea visualizar.

La clase ServerSocket

La clase **ServerSocket** es la que se utiliza a la hora de crear servidores, al igual que, como ya se ha visto, la clase **Socket** se utilizaba para crear clientes. Un objeto **ServerSocket** representa una conexión TCP a la escucha.

El ejemplo `Java1707.java` utiliza *sockets* para implementar dos servidores diferentes sobre una red IP. Desde luego, el programa es solamente ilustrativo, por lo que si el lector decide utilizarlo en alguno de sus propósitos, el lector asume el riesgo de su uso. El porqué de esta advertencia reside en el uso del agente de seguridad que se implementa en éste, y algunos más, de los ejemplos que se van a presentar, que no es nada restrictivo, y puede permitir el acceso no deseado a la máquina en que se ejecuten estos programas de ejemplo. El control realizado se limita a no permitir la navegación por encima del directorio actual; pero esto es más como ejemplo para que el lector pueda probar el envío de mensaje de error al cliente, que un verdadero método de seguridad.

Este programa implementa dos servidores operando sobre dos puertos diferentes del ordenador que está ejecutando el programa. Uno de los servidores es un servidor "*echo*" que está implementado a través de una tarea que monitoriza el puerto 7777, aunque el puerto por defecto para el servicio *echo* es el 7. Este servidor devuelve la cadena que reciba por el puerto 7777 al mismo cliente que la ha enviado. El otro servidor es un servidor HTTP abreviado, que se ha implementado a través de una tarea que monitoriza el puerto 8090, aunque el puerto estándar para el protocolo *http* es el 80. En este servidor se ha implementado la respuesta al comando **GET**, que devuelve

al cliente el contenido del fichero que haya solicitado, o un mensaje de error en caso de no encontrarlo.

Si el lector ejecuta los ejemplos en entorno *Windows*, es suficiente con que arranque los ejemplos como cualquier otro programa Java, antes de lanzar la parte cliente, o al menos antes de que la parte cliente solicite algo al servidor. Sin embargo, si la ejecución se realiza en entornos *Solaris*, *Linux*, etc. probablemente se encuentre con que no hay permiso para utilizar los puertos que asigne si son números muy bajos. Esto es debido a que los puertos por debajo del 1024 están reservados para uso del sistema; no obstante, el lector puede cambiar el número de puerto tanto en el servidor como en el cliente y ejecutarlos.

Los ejemplos `Java1708.java` y `Java1709.java` son clientes implementados para probar los dos servidores que se crean en este ejemplo. La parte del servidor de *echo* se puede probar utilizando el primero de los ejemplos, `Java1708.java`, y el otro ejemplo permite probar la parte HTTP. También puede el lector comprobar el funcionamiento del agente de seguridad, solicitando ficheros no válidos al servidor y, por supuesto, puede utilizar cualquier navegador desde el propio ordenador para realizar peticiones al servidor en la forma: *http://localhost:8090/fichero*.

Si el lector consulta el código del ejemplo, observará que la parte principal de la aplicación se encuentra en el método *main()*, conteniendo las dos sentencias que permiten el uso de las clases que implementan el objetivo concreto de la creación de los dos servidores.

```
public static void main( String[] args ) {
  ServidorHttp servidorHttp = new ServidorHttp();
  ServidorEco servidorEcho = new ServidorEco();
  }
```

La primera sentencia instancia un objeto servidor que monitoriza el puerto 8090; este servidor implementa parte del protocolo HTTP y permite la descarga de ficheros a navegadores. Y la segunda sentencia instancia un objeto servidor que monitoriza el puerto 7777; este servidor devuelve al cliente TCP/IP la misma cadena que haya enviado al servidor. Estos dos servidores se implementan en tareas diferentes para que puedan funcionar en paralelo de forma asíncrona.

Agente de seguridad

A la hora de implementar el servidor HTTP, es importante el control de seguridad que se establezca, de modo que no se permita el acceso a cualquier fichero de la plataforma en que se ejecute, sino solamente a aquellos que se desee estén expuestos. Por ejemplo, un usuario puede indicar la URL siguiente:

```
http://servidor:1234/../prohibido.html
```

donde ".." referencia al directorio superior del que se localiza el servidor. Esta circunstancia es indeseable, por lo que Java implementa un sistema de control de acceso a recursos desde aplicaciones Java que involucra a la clase **SecurityManager**, ficheros de normas o políticas de acceso, permisos y otras herramientas. El tratamiento de la seguridad en Java es un tema que se escapa al alcance de este libro, por lo que solamente se introducirán las nociones básicas para entender la seguridad en el servidor HTTP que implementa el ejemplo.

Para restringir el acceso a los recursos del sistema se puede cargar un agente de seguridad mediante un objeto de tipo **SecurityManager** que controle el acceso a esos recursos. Normalmente cuando se ejecuta una aplicación Java, no hay ningún agente de seguridad, o en caso de que lo haya, no funciona de la forma restrictiva en que se hace en el ejemplo. Sin embargo, cuando se establece un agente de seguridad para una aplicación, mediante la llamada al método *setSecurityManager()*, lo contrario es cierto. Es decir, cuando se coloca un agente de seguridad en una aplicación, a ese código se le prohíbe hacer cualquier cosa; la aplicación deja de tener privilegios, y es el programador, como diseñador del sistema de seguridad, el que decide qué privilegios va a tener disponibles y qué cosas van a estar prohibidas a esa aplicación.

Esto es un proceso tedioso y complicado. Aunque no es difícil hacer que un privilegio esté disponible una vez que se haya decidido que va a estarlo, el llegar a esa conclusión requiere un gran conocimiento de cómo se relacionan los privilegios con las operaciones que se pueden realizar; porque hay ocasiones en que esta relación entre privilegio y operación no es tan obvia a simple vista. Para crear un agente de seguridad propio, hay que extender la clase **SecurityManager**, que contiene alrededor de una treintena de métodos que comienzan con la palabra *check* (comprobar), como *checkAccept()*. Esos métodos son los que controlan el acceso a los privilegios; cada uno de los métodos controla el acceso a un privilegio que generalmente coincide con lo que indica la segunda parte del nombre.

Sin embargo, a partir del JDK 1.2 ya no es necesario montar la parafernalia indicada en los párrafos anteriores, que requiere mucho código propio para el control de acceso en cada método sobrescrito, sino que se puede emplear el sistema de permisos basados en un fichero de normas o políticas de seguridad, que es comprobado por el agente de seguridad cada vez que va a realizar una acción, para ver si está permitida. Si una acción no se encuentra en el fichero indicando sus permisos, está prohibida.

En el capítulo 9 se introdujo el concepto de políticas o normas de seguridad a la hora de tratar el acceso a ficheros, y se volverá sobre el tema a la hora de las conexiones RMI en un capítulo posterior. En el capítulo 9, se utilizaba el fichero de políticas de seguridad para permitir la lectura y escritura de ficheros. También se puede utilizar para permitir el borrado de ficheros; por ejemplo, la siguiente entrada en el fichero de políticas de seguridad:

```
grant codebase "file:/home/afq/tutorial/" {
```

```
  permission java.io.FilePermission "*.tmp", "delete";
  };
```

permitiría borrar del directorio */home/afq/tutorial*, aquellos ficheros con extensión *.tmp* y no otros.

También se indicaba en el capítulo 9 cómo invocar el controlador de seguridad y hacer que se leyese el fichero de políticas de seguridad propio, mediante el comando:

```
java -Djava.security.manager -Djava.security.policy=politicas clase
```

Como puede suponer el lector, esta aproximación es mucho más flexible, sencilla y clara, que tener que extender la clase **SecurityManager**, personalizar cada uno de los métodos de comprobación y recompilar después de cada modificación. En la implementación de los servidores del ejemplo **Java1707** se utiliza un fichero de políticas para definir los permisos de acceso a ficheros, de forma que a través del servidor HTTP un cliente no puede pedir cualquier fichero del sistema. También se fijan los permisos de acceso a los sockets que utilizan los servidores. El contenido del fichero de políticas de seguridad sería el siguiente:

```
grant {
  permission java.io.FilePermission "/home/afq/tutorial/cap_17/-","read";
  permission java.net.SocketPermission "localhost:1234-",
    "accept,connect,listen";
  };
```

Los servidores que implementa el ejemplo deben acceder a recursos externos específicos, concretamente, el servidor Eco necesita tener acceso al socket que se especifique en la aplicación. Por tanto, es imprescindible conceder los permisos necesarios en el fichero de políticas de seguridad para que la aplicación pueda aceptar conexiones, establecer conexiones y quedarse a la escucha en los puertos de acceso de cada servidor. En el fichero de políticas se fijan estos permisos para cualquier socket por encima del 1234, lo cual se indica mediante la expresión *1234-*.

La expresión */home/afq/tutorial/cap_17/-*, significa que se permite la lectura de los ficheros situados en el directorio indicado y en todos sus subdirectorios, pero no en los directorios que se encuentren por encima.

La seguridad en Java no involucra solamente al agente de seguridad sino también temas como la criptografía, claves públicas y privadas, certificados, etc. Sin embargo, los conceptos básicos sobre el establecimiento de permisos son sencillos. Además de las clases ya vistas como **SocketPermission** y **FilePermission**, hay otra serie de clases que representan varios tipos de acceso a sistemas externos o áreas restringidas del sistema que está bajo el control del agente de seguridad. Entre estas clases están:

```
java.security.AllPermission
java.security.SecurityPermission
java.awt.AWTPermission
java.io.SerializablePermission
java.lang.reflect.ReflectPermission
```

```
java.lang.RuntimePermission
java.net.NetPermission
java.net.SSLPermission
java.sql.SQLPermission
java.util.PropertyPermission
java.util.logging.LoggingPermission
javax.sound.sampled.AudioPermission
```

y algunas otras más, todas ellas subclases de **java.security.Permission**. El lector deberá consultar el API de Java para obtener información completa de estas clases.

El formato básico para permitir el acceso a los recursos, debe tener los parámetros que indican las sentencias siguientes:

```
grant codeBase "URL" {
  permission clave_permiso_1 "destino","acción[,acción..]";
  permission clave_permiso_2 "destino","acción[,acción..]";
  . . .
  };
```

En donde `codeBase` indica la localización del recurso al cual se quiere asignar un permiso. Si se carga una clase desde una localización distinta, el permiso no se aplicaría y si se deja vacío, entonces los permisos se aplicarán a cualquier localización. Hay permisos que requieren indicar un objeto `destino`, al cual se aplicarán los permisos, y una lista de *acciones* que se permiten realizar sobre él.

Hay ocasiones en que los programas necesitan comprobar si una acción está permitida antes de intentarla. Por ejemplo, el lector puede desarrollar una clase que vaya a ser utilizada en otras aplicaciones sobre las que no tiene control, y desea evitar un tiempo de ejecución en acciones cuyo paso final no está permitido. En estos casos, para comprobar si una acción está permitida, se crea una instancia del permiso de que se trate y se pasa como argumento al método *checkPermission()* que proporciona la clase **SecurityManager**. Si se lanza una excepción de tipo **SecurityException**, indicará que la acción no está permitida. El código siguiente, por ejemplo, comprueba el permiso de acceso a ficheros:

```
Permission permiso = new java.io.FilePermission("/-","read" );
SecurityManager agente = System.getSecurityManager();
if( agente != null ) {
  try {
    agente.checkPermission( permiso );
  }catch( SecurityExcepcion e ) {
    System.out.println( "Acceso a ficheros no permitido" );
    }
  }
```

PolicyTool

En el capítulo 9 se citó la herramienta *PolicyTool* como alternativa a la generación del fichero de normas o políticas de seguridad, en lugar de tener que crearlo mediante un editor de texto normal.

PolicyTool es una herramienta gráfica que permite indicar todos los detalles que necesita el fichero de políticas de seguridad como el directorio donde estará el fichero de políticas, la información de la base de códigos, los permisos que contendrá, el objeto destino de los permisos y las acciones que se permitirán realizar sobre él.

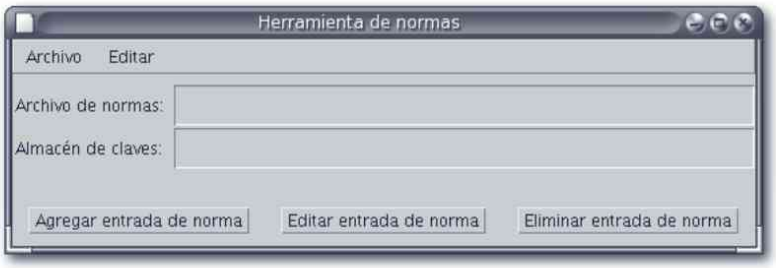


Figura 17.1

Para crear un fichero de normas o políticas de seguridad, se selecciona el botón *Agregar entrada de norma*, que presentará el diálogo que permite introducir la base de códigos, `codeBase`, y los permisos que se van a conceder.

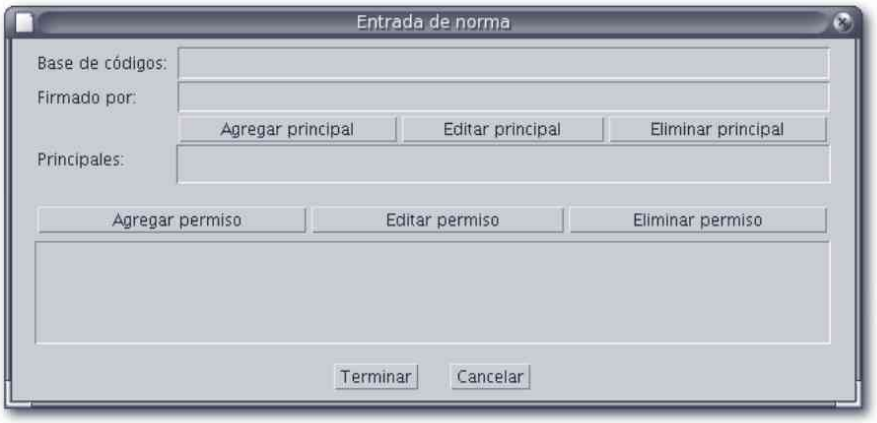


Figura 17.2

La generación del fichero de políticas de seguridad `java.politicas` correspondiente a este capítulo se puede indicar en el campo *Base de códigos* el directorio en donde se está ejecutando el ejemplo **Java1707**, o directamente a la hora de indicar el *Nombre de destino*. Para incorporar nuevos permisos al fichero, se selecciona el botón *Agregar permiso*, que presentará un nuevo diálogo para permitir la introducción de permisos.



Figura 17.3

Seleccionando los desplegables se van eligiendo los permisos e indicando en los campos de texto de la parte derecha correspondientes, los datos adecuados. También se puede editar el contenido, sin seleccionar el desplegable, por ejemplo para introducir `localhost:1234-` en el campo *Nombre de destino*, ya que el menú desplegable no ofrece tal opción.

A continuación se cierran todos los diálogos y en la ventana principal, desde el menú *Archivo*, se salva el fichero de políticas de seguridad en el directorio deseado.

Servidor Eco

La siguiente parte interesante del código del programa es la clase que implementa el servidor del protocolo *echo*, que es el más simple de los dos que se muestran en el ejemplo. En general, los dos servidores funcionan del mismo modo, corriendo cada uno de ellos en su propia tarea, y escuchando el puerto que corresponde a su protocolo, en espera de conexiones de clientes. Mientras un servidor se encuentra en este estado de espera de conexiones, estará bloqueado para consumir el mínimo de recursos del sistema.

Cuando un cliente requiere una conexión en cualquiera de los dos puertos, el servidor lanza otra tarea para manejar las necesidades del cliente y luego vuelve para seguir escuchando el puerto en su propia tarea. Las tareas para dar soporte a los clientes se lanzan con prioridad mínima, de tal modo que estas tareas no interfieran la capacidad asignada a los servidores de reconocer y responder a cualquier otro cliente que esté solicitando una conexión. Si hay muchos clientes solicitando conexiones, el servidor que atiende al puerto lanzará una tarea por cada una de esas conexiones (dentro de las capacidades del sistema); de forma que puede haber muchas tareas ejecutándose simultáneamente, atendiendo cada una de ellas a las necesidades de un cliente específico.

La primera sentencia interesante del servidor de eco es el constructor, que simplemente invoca a su propio método *start()* encargado de iniciar el arranque de su propia tarea.

```
start();
```

La clave del funcionamiento de un servidor en Java se encuentra en el método *accept()* de la clase **ServerSocket**. Antes, hay que construir un objeto de esa clase y, como se puede ver en el siguiente trozo de código, el constructor de **ServerSocket** tiene solamente un parámetro: el número del puerto que va a ser monitorizado por el servidor que construye.

```
ServerSocket socket = new ServerSocket( 7777 );
System.out.println( "Servidor escuchando el 7777" );
```

Y acto seguido se puede realizar la llamada al método *accept()* para recuperar las conexiones que establezcan los clientes con el servidor a través del puerto que está monitorizando ese servidor. Las líneas siguientes muestran cómo se hace esto en el ejemplo.

```
while( true )
  new ConexionEcho( socket.accept() );
```

Cuando se instancia un objeto de tipo **ServerSocket** y se invoca el método *accept()* sobre ese objeto, este método bloquea el servidor y se queda a la espera hasta que se produce una conexión de un cliente en el puerto que controla. Cuando esto sucede, se instancia de forma automática un objeto **Socket** que el que devuelve el método *accept()*.

El lector comprobará que hay muchos conceptos que se están repitiendo, y que si va leyendo con detenimiento, se vuelve sobre ellos una y otra vez. Esto se hace de forma deliberada, porque hay una serie de conceptos y métodos muy simples en los que se basa todo el funcionamiento de las comunicaciones en red con Java, y que son los que verdaderamente tienen que quedar arraigados en el conocimiento del lector. Por ello, aun a pesar de resultar un poco ladrillo, es preferible recaer una vez sobre otra en los conceptos considerados fundamentales, para que una vez concluida la lectura del capítulo, al menos esos conceptos hayan quedado suficientemente claros.

El objeto **Socket** devuelto por el método *accept()* se conecta automáticamente con el objeto **Socket** utilizado por el cliente que ha establecido la conexión, y se puede utilizar para establecer la comunicación con el cliente. Esto es muy importante, así que hay que detenerse un poco en ello.

No hay ninguna transferencia de datos por el hecho de disponer de un objeto de tipo **ServerSocket**. Es en el momento en que el cliente solicita una conexión cuando se instancia automáticamente un objeto **Socket** que se conecta con el **Socket** del cliente que ha pedido la conexión. Este nuevo socket es el devuelto por el método *accept()* para poder establecer la comunicación con el socket del cliente.

En el trozo de código anterior, se observa que cuando se recibe una conexión, se lanza una tarea del tipo **ConexionEcho** que atiende al cliente que ha solicitado esa conexión. Al constructor de la clase que maneja este nueva tarea, se le pasa como

parámetro el objeto **Socket** que el método *accept()* devuelve, que ya está conectado con el cliente, y a través del cual el protocolo de comunicaciones para el servicio, *echo* en este caso, puede comunicarse con el cliente.

Observe el lector que la tarea en la cual está corriendo el objeto **ServidorEco** se encuentra en medio de un bucle infinito. Luego una vez que la tarea **ConexionEcho** haya sido lanzada con éxito, el objeto **ServidorEco** volverá a su función inicial de seguir esperando a la conexión de otro cliente a través del puerto de eco.

El siguiente trozo de código en que merece la pena detenerse es el constructor de la clase **ConexionEcho**. Los objetos de esta clase son creados a raíz de la petición de conexión de algún cliente al servidor. No obstante, estos objetos no saben nada de clientes y servidores, lo único que conocen es que hay un socket TCP/IP conectado con otro socket en otra máquina, o con otro proceso de la misma máquina.

```
ConexionEcho( Socket socket ) {
  System.out.println( "Recibida una llamada en el puerto 7777" );
  this.socket = socket;
  setPriority( NORM_PRIORITY-1 );
  start();
  }
```

Como puede observar el lector, tras guardar el parámetro en una variable de instancia, el constructor fija la prioridad de la tarea. Esto es para que las tareas que están monitorizando los puertos no se vean interrumpidas por las tareas que están atendiendo a las conexiones ya establecidas. Una vez ajustada la prioridad, solamente resta invocar al método *start()* para que la tarea se levante y empiece a correr. Como es habitual, el método *start()* invocará al método *run()* de la tarea.

El método *run()* de esta tarea es similar al código que ya se ha presentado en ejemplos anteriores al programar la parte cliente. Por referenciar algo, se reproducen las líneas de código en donde se lee la cadena recibida del cliente y se devuelve a ese cliente, exactamente igual que se ha recibido. Una vez hecho esto, se cierra el socket de conexión con el cliente y la tarea se muere.

```
String cadena = entrada.readLine();
salida.println( cadena );
System.out.println( "Recibido: "+cadena );
socket.close();
System.out.println( "Socket cerrado" );
```

Con esto se concluye el repaso al código que implementa el servidor Eco en el programa y se pasa a discutir la clase que atiende a las conexiones con el puerto estándar del protocolo HTTP.

Servidor HTTP

Este servidor HTTP es realmente simple, y solamente responde al comando **GET** del protocolo HTTP, cualquier otro comando será ignorado y el cliente obtendrá como respuesta un mensaje de error.

En el método *main()* se instancia un objeto de la clase **ServidorHttp**, que monta una tarea que instancia un objeto de tipo **ServerSocket** para atender al puerto que utiliza para el protocolo HTTP, el 8090.

```
ServerSocket socket = new ServerSocket( 8090 );
System.out.println( "Servidor escuchando el puerto 8090" );
```

El método *accept()*, tal como muestran las líneas de código siguientes, se invoca sobre este objeto dentro de un bucle infinito para bloquear y monitorizar el puerto 8090, esperando conexiones de clientes. Cuando esto ocurre, el método *accept()* instancia y devuelve un objeto **Socket**, que estará conectado con la máquina cliente.

```
while( true )
  new ConexionHttp( socket.accept() );
```

Este objeto **Socket** se pasa como parámetro al constructor de un nuevo objeto de tipo **ConexionHttp**, que lanzará una tarea específica para atender a la conexión que se ha establecido con el cliente, a través de la versión abreviada del protocolo HTTP que se ha implementado en esta clase.

El constructor es similar al visto para el servidor de Eco; así que, en aras de la brevedad no se incluye aquí. Si el lector sigue el código del ejemplo, no encontrará nada nuevo hasta llegar a la transmisión de información al cliente, que en este caso se hace enviando un array de bytes. El siguiente fragmento de código muestra la creación del canal de salida que va a permitir este tipo de comunicación.

```
OutputStream os = socket.getOutputStream();
```

Esta sentencia se encuentra dentro del método *run()*, corazón de la tarea que está atendiendo a la conexión. A continuación se moldea el canal de salida para enviar texto a través de él, mediante el objeto **OutputStreamWriter**, al cual se indica la codificación de caracteres ASCII.

```
OutputStreamWriter osw = new OutputStreamWriter( os,"8859_1" );
```

Y finalmente, se utiliza un objeto de tipo **PrintWriter** para tener acceso a los métodos de alto nivel que proporciona para enviar a través de él. Se abre en modo *autoflush*, de forma que el vaciado del canal se produce de forma automática cuando se invoca al método *println()*.

```
salida = new PrintWriter( osw,true );
```

A continuación, lo que se espera es la conexión del cliente, para lo que se utiliza el método *readLine()* sobre el canal de entrada que lee las peticiones y almacena esa petición en un objeto **String**.

```
String peticion = entrada.readLine();
```

Lo siguiente que hay que hacer es el análisis de la petición que ha realizado el cliente para comprobar si es posible atenderla o no, teniendo en cuenta que solamente se responde al comando **GET**. Para realizar este análisis se utiliza un objeto **StringTokenizer**, que indicará si hay o no un comando **GET** en la petición realizada por el cliente.

```
StringTokenizer st = new StringTokenizer( peticion );
if( ( st.countTokens() >= 2 )
  && st.nextToken().equals("GET") ) {
```

En caso de que no sea una petición **GET**, se creará una página dentro del código para devolver el mensaje de error al cliente. Si la petición sí corresponde a un comando **GET**, lo que se intenta es comprobar el nombre del fichero que ha solicitado el cliente y enviárselo; y en caso de que no solicite ninguno, completar el camino que haya indicado con el fichero `index.html`, que es el fichero estándar que utilizan casi todos los navegadores como fichero de defecto en caso de no especificar una página determinada en el acceso a un sitio Web. Las siguiente línea de código elimina la posibilidad de que la petición contenga barras "/" extra.

```
if( (peticion = st.nextToken()).startsWith("/") ) {
```

Si la petición termina con una barra "/" o es una cadena vacía, se supone que el cliente quiere descargar el fichero `index.html`; así que, en este caso, se añade este nombre de fichero a la cadena que contiene la petición realizada por el cliente.

```
if( peticion.endsWith("/") || peticion.equals("") ) {
  System.out.println( "Peticion terminada en / o blanco, "+
    "se le incorpora: index.html" );
  peticion = peticion + "index.html";
  System.out.println( "Peticion modificada: "+peticion );
  }
```

Llegados a este punto, ya se sabe cuál es el fichero que hay que enviarle al cliente como respuesta a su petición, así que se intenta abrir un objeto de tipo **FileInputStream** con ese nombre de fichero y en el camino que se indique. Si no se puede conseguir, se lanzará una excepción que será procesada en el bloque `try-catch` del final del programa. Si el fichero sí se puede leer, se creará un array de bytes igual al contenido del fichero y se leerá ahí ese contenido.

```
FileInputStream fichero = new FileInputStream( peticion );
// Se instancia un array de bytes igual al número de bytes que
// se pueden leer del canal de entrada sin bloquearlo.
// Y luego se rellena con los datos
byte[] datos = new byte[fichero.available()];
fichero.read( datos );
```

Una vez que se ha identificado, localizado y leído el fichero en un array en memoria, el siguiente paso consiste en el uso del canal de salida que se había creado anteriormente para transmitir el contenido del array al cliente; concluyendo con la liberación forzada de todo el contenido del array, tal como se muestra en las siguientes líneas de código.

```
pagina.write( datos );
pagina.flush();
```

El siguiente fragmento de código se ejecuta en el caso de que el cliente no envíe una petición **GET**, en cuyo caso se crea un documento sobre la marcha conteniendo el mensaje de error que indica tal circunstancia y se le envía como respuesta al cliente.

```
} else
  salida.println( "<HTML><BODY><P>400 Petici&oactute;n "+
    "Err&oacute;nea<P></BODY></HTML>" );
socket.close();
```

Esta última sentencia cierra el socket y permite que la tarea termine normalmente, siempre asumiendo que no se ha lanzado ninguna excepción mientras; porque en el caso de que se hubiese generado alguna excepción, hay varios controladores para tratar estas excepciones, e incluso algunos, como el lector podrá observar en el código del ejemplo, generan páginas sobre la marcha que se envían al cliente, indicando la circunstancia que ha provocado el lanzamiento de la excepción que controlan.

Quizá merezca la pena detenerse en el controlador de la última excepción, la de tipo **IOExcepction**, que se muestra en el siguiente trozo de código.

```
}catch( IOException e ) {
  e.printStackTrace();
  try {
    socket.close();
    System.out.println( "Socket cerrado" );
  }catch( IOException evt ) {
    System.out.println( evt );
  }
```

Es especial porque es necesario cerrar el *socket* dentro del controlador de la excepción. La verdad es que no es previsible que se pueda forzar una excepción de tipo **IOException**, dentro del manejador de una excepción de tipo **IOException** lanzada anteriormente.

En la actualidad, muchas empresas están sacando partido a Internet utilizando esta tecnología como base de sus productos; sin embargo, la parte Web sigue siendo la más visible de Internet y el lenguaje HTML sigue siendo el más importante para almacenar contenido. Aunque ahora existen multitud de siglas como XML, XHTML, CSS, DOM, XSL, RDF, RSS y otras más que pertenecen a estándares que el programador Web debe tener en cuenta a la hora de sus diseños; lo cierto es que al final, se hablarán con los navegadores a través de un servidor Web y utilizarán el protocolo HTTP.

Java no es ajeno a esto y el escribir aplicaciones cliente/servidor, tal como se ha dejado entrever en párrafos anteriores, se resume en desarrollar la parte HTTP en el servidor y dejar que cualquier navegador Web sea el cliente. Actualmente hay dispositivos que son configurables a través de navegadores como impresoras, servidores, routers, cortafuegos y, en realidad, cualquier dispositivo que disponga de una dirección IP. El protocolo HTTP hace esto posible.

La creación de un servidor HTTP/1.1 no es una tarea sencilla y crear una implementación eficiente, menos todavía. Sin embargo, implementar un servidor HTTP que se pueda incluir en una aplicación Java no es difícil, es más, incluso resulta sencillo, y el lector lo ha podido comprobar en el ejemplo anterior.

LA CLASE DATAGRAMPACKET

La clase **DatagramPacket**, junto con la clase **DatagramSocket**, es la que se utiliza para la implementación del protocolo **UDP** (*User Datagram Protocol*).

En este protocolo, a diferencia de lo que ocurría en el protocolo TCP, en el cual si un paquete se dañaba durante la transmisión se reenviaba ese paquete, para asegurar una comunicación segura entre cliente y servidor; con UDP no hay garantía alguna de que los paquetes lleguen en el orden correcto a su destino y, ni tan siquiera hay seguridad de que lleguen todos los paquetes que se hayan enviado. Sin embargo, los paquetes que consiguen llegar, lo hacen mucho más rápidamente que con TCP y, en algunos casos, la velocidad de transmisión es mucho más importante que el que lleguen todos los paquetes; por ejemplo, si lo que se están transmitiendo son señales de sensores en tiempo real para la presentación en pantalla de las medidas que obtienen, la velocidad es más importante que la integridad, porque si un paquete no llega o no puede recomponerse, en el instante siguiente llegará otro.

La programación para uso del protocolo UDP se diferencia de la programación del protocolo TCP en que no existe el concepto de **ServerSocket** para los datagramas y que es el programador quien debe construirse los paquetes a enviar por UDP.

Para enviar datos a través de UDP, hay que construir un objeto de tipo **DatagramPacket** y enviarlo a través de un objeto **DatagramSocket**, y al revés para recibirlos, es decir, a través de un objeto **DatagramSocket** se captura el objeto **DatagramPacket**. Toda la información respecto a dirección, puerto y datos está contenida en el paquete.

Para enviar un paquete, primero se construye ese paquete con la información que se desea transmitir, luego se almacena en un objeto **DatagramSocket** y, finalmente se invoca el método *send()* sobre ese objeto. Para recibir un paquete, primero se construye un paquete vacío y luego se le presenta a un objeto **DatagramSocket** para que almacene allí el resultado de la ejecución del método *receive()* sobre ese objeto.

Hay que tener en cuenta que la tarea encargada de todo esto estará bloqueada en el método *receive()* hasta que un paquete físico de datos se reciba a través de la red; este paquete físico será el que se utilice para rellenar el paquete vacío que se había creado.

También hay que tener cuidado cuando se pone a escuchar a un objeto **DatagramSocket** en un puerto determinado, porque va a recibir los datagramas enviados por cualquier cliente. Es decir, que si los mensajes enviados por los clientes están formados por múltiples paquetes; en la recepción pueden llegar paquetes entremezclados de varios clientes y es responsabilidad de la aplicación el ordenarlos.

Para la clase **DatagramPacket** se dispone de dos constructores, uno utilizado cuando se quieren enviar paquetes y el otro se usa cuando se quieren recibir paquetes. Ambos requieren que se les proporcione un array de bytes y la longitud que tiene. En el caso de la recepción de datos, no es necesario nada más, los datos que se reciban se depositarán en el array; aunque en el caso de que se reciban más datos físicos de los que soporta el array, el exceso de información se perderá y se lanzará una excepción de tipo **IllegalArgumentException**, que a pesar de que no sea necesaria su captura, siempre es bueno capturarla.

Cuando se construye el paquete a enviar, es necesario colocar los datos en el array antes de llamar al método *send()*; además de eso, hay que incluir la longitud de ese array, y también se debe proporcionar un objeto de tipo **InetAddress** indicando la dirección de destino del paquete y el número del puerto de ese destino en el cual estará escuchando el receptor del mensaje. Es decir, que la dirección de destino y el puerto de escucha deben ir en el paquete, al contrario de lo que pasaba en el caso de TCP que se indicaba en el momento de construir el objeto **Socket**.

El tamaño físico máximo de un datagrama es 65.535 bytes, y teniendo en cuenta que hay que incluir datos de cabecera, esa longitud nunca está disponible para datos de usuario, sino que siempre es algo menor.

La clase **DatagramPacket** proporciona varios métodos para poder extraer los datos que llegan en el paquete recibido. La información que se obtiene con cada método coincide con el propio nombre del método, aunque hay algunos casos en que es necesario saber interpretar la información que proporciona ese mismo método.

El método *getAddress()* devuelve un objeto de tipo **InetAddress** que contiene la dirección del *host* remoto. El saber cuál es el ordenador de origen del envío depende de la forma en que se haya obtenido el datagrama. Si ese datagrama ha sido recibido a través de Internet, la dirección representará al ordenador que ha enviado el datagrama (el origen del datagrama); pero si el datagrama se ha construido localmente, la dirección representará al ordenador al cual se intenta enviar el datagrama (el destino del datagrama).

De igual modo, el método *getPort()* devuelve el puerto desde el cual ha sido enviado el datagrama, o el puerto a través del cual se enviará, dependiendo de la forma en que se haya obtenido el datagrama.

El método *getData()* devuelve un array de bytes que contiene la parte de datos del datagrama, ya eliminada la cabecera con la información de encaminamiento de ese datagrama. La forma de interpretar ese array depende del tipo de datos que contenga. Los ejemplos que se ven en este Tutorial utilizan exclusivamente datos de tipo **String**, pero esto no es un requisito, y se pueden utilizar datagramas para intercambiar cualquier tipo de datos, siempre que se puedan colocar en un array de bytes en un ordenador y extraerlos de ese array en la parte contraria. Es decir, que la responsabilidad del sistema se limita a desplazar el array de bytes de un ordenador a otro, y es responsabilidad del programador asignar significado a esos bytes.

El método *getLength()* devuelve el número de bytes que contiene la parte de datos del datagrama, y el método *getOffset()* devuelve la posición en la cual empieza el array de bytes dentro del datagrama completo.

LA CLASE DATAGRAMSOCKET

Un objeto de la clase **DatagramSocket** puede utilizarse tanto para enviar como para recibir un datagrama. Representa un socket no orientado a conexión.

La clase tiene tres constructores. Uno de ellos se conecta al primer puerto libre de la máquina local; el otro permite especificar el puerto a través del cual operará el socket; y el tercero permite especificar un puerto y una dirección para identificar a una máquina concreta.

Independientemente del constructor que se utilice, el puerto desde el cual se envía el datagrama siempre se incluirá en la cabecera del paquete. Normalmente, la parte del servidor utilizará el constructor que permite indicar el puerto concreto a usar, ya que si no, la parte cliente no tendría forma de conocer el puerto por el cual le van a llegar los datagramas.

La parte cliente puede utilizar cualquier constructor, pero por flexibilidad, lo mejor es utilizar el constructor que deja que el sistema seleccione uno de los puertos disponibles. El servidor debería entonces comprobar cuál es el puerto que se está utilizando para el envío de datagramas y enviar la respuesta por ese puerto.

A diferencia de esta posibilidad de especificar el puerto, o no hacerlo, no hay ninguna otra distinción entre los *sockets* datagrama utilizados por cliente y servidor. Si el lector se encuentra un poco perdido, no desespere, porque en los ejemplos todo esto que es bastante difícil de explicar con palabras, se ve mucho más claramente al intuir el funcionamiento físico de la comunicación entre cliente y servidor.

Para enviar un datagrama hay que invocar al método *send()* sobre un socket datagrama existente, pasándole el objeto paquete como parámetro. Cuando el paquete es enviado, la dirección y número de puerto del ordenador origen se coloca automáticamente en la porción de cabecera del paquete, de forma que esa información pueda ser recuperada en el ordenador destino del paquete.

Para recibir datagramas, hay que instanciar un objeto de tipo **DatagramSocket**, conectarse a un puerto determinado e invocar al método *receive()* sobre ese socket. Este método bloquea la tarea hasta que se recibe un datagrama, por lo que si es necesario hacer alguna acción durante la espera, hay que invocar al método *receive()* en la tarea que realice dicha acción.

Si se trata de un servidor, hay que conectarse con un puerto específico. Si se trata de un cliente que está esperando respuestas de un servidor, hay que escuchar en el mismo puerto que fue utilizado para enviar el datagrama inicial. Si se envía un datagrama a un puerto anónimo, se puede mantener el *socket* abierto que fue utilizado en el envío del primer datagrama y utilizar ese mismo *socket* para esperar la respuesta. También se puede invocar al método *getLocalPort()* sobre el *socket* antes de cerrarlo, de forma que se pueda saber y guardar el número del puerto que se ha empleado; de este modo se puede cerrar el *socket* original y abrir otro en el mismo puerto en el momento en que se necesite.

Para responder a un datagrama, hay que obtener la dirección del origen y el número de puerto a través del cual fue enviado el datagrama, de la cabecera del paquete y luego, colocar esta información en el nuevo paquete que se construya con la información a enviar como respuesta. Una vez pasada esta información a la parte de datos del paquete, se invoca al método *send()* sobre el objeto **DatagramSocket** existente, pasándole el objeto paquete como parámetro.

Es importante tener en cuenta que los números de puerto TCP y UDP no están relacionados. Se puede utilizar el mismo número de puerto en dos procesos si uno se comunica a través de protocolo TCP y el otro lo hace a través de protocolo UDP. Es muy común que los servidores utilicen el mismo puerto para proporcionar servicios similares a través de los dos protocolos en algunos servicios estándar, como puede ser el servicio *echo*.

Cliente Eco

Esta aplicación no es más que una actualización del ejemplo `Java1704.java`, al que se le incluye además el protocolo UDP. El ejemplo `Java1710.java` realiza dos pruebas del servicio *echo* contra el mismo servidor, enviando una línea de texto al puerto estándar de este servicio, el puerto 7. En la primera prueba utiliza el protocolo TCP/IP y en la segunda emplea un datagrama UDP.

En el programa se instancian dos objetos de tipo **String**, para utilizar uno diferente en cada protocolo. Luego está la parte correspondiente a la prueba de Eco a través de TCP, sobre la que no se insiste más. Una vez cerrado el socket TCP, comienza la prueba de eco a través de UDP.

En primer lugar se convierte el mensaje que se ha de enviar por UDP a un array de bytes. Hay que instanciar un objeto de tipo **InetAddress** que contenga la dirección del servidor con el que se va a realizar la conexión y el envío del datagrama con el array de bytes recién creado.

Se crea un objeto de tipo **DatagramPacket** que contenga el array de bytes de la cadena a enviar, junto con la dirección del servidor y el puerto al que hay que conectarse. Se crea ahora un objeto de tipo **DatagramSocket** que será utilizado para enviar el paquete al servidor. Sin embargo, hay que recordar que el protocolo UDP no garantiza que el paquete llegue íntegro al servidor, o que tan siquiera llegue.

Solamente resta invocar al método *send()* sobre el objeto **DatagramSocket**, pasándole el objeto **DatagramPacket** como parámetro. Esto hace que la dirección local y el número de puerto se incorporen en el paquete y éste sea enviado a la dirección y número de puerto que se ha encapsulado en el objeto **DatagramPacket** en el momento de su creación.

Los mismos objetos **DatagramSocket** y **DatagramPacket** serán los utilizados para recibir el paquete de respuesta del servidor (siempre que la suerte acompañe). Se usa un bucle para sobrescribir los datos en el paquete con una letra para poder comprobar que una vez recibido el paquete de respuesta del servidor de eco, los datos son nuevos y no simplemente el residuo del mensaje que se había colocado originalmente en el paquete.

Luego se invoca el método *receive()* sobre el objeto **DatagramSocket**, pasándole el objeto **DatagramPacket** como parámetro. Esto hace que la tarea se bloquee en el mismo puerto por el que se ha enviado el paquete, hasta que llegue una respuesta. En el momento en que un paquete físico llegue desde el servidor, se extraen los datos y se colocan en el objeto **DatagramPacket** que se le ha proporcionado como parámetro. Una vez hecho esto, la tarea se desbloquea y el flujo de control de la aplicación sigue por la sentencia que muestra el contenido del paquete.

Finalmente, se cierra el *socket* y el programa termina.

Como el ejemplo es una simple actualización de otro de los ejemplos del Tutorial, gran parte del código ya está más que visto; y es exactamente igual al del ejemplo `Java1704.java`. Así que solamente se van a repasar a continuación algunas de las líneas de código más interesantes de la parte que se aporta nueva en este ejemplo, que corresponderá, evidentemente, a la implementación del intercambio de mensajes con el servidor a través del puerto del servicio *Eco*, pero con protocolo UDP.

El primer fragmento en que hay que detenerse es justo en la declaración del método *main()*, en donde se declaran e inicializan algunas variables importantes.

```
public static void main( String[] args ) {
String servidor = "www.rediris.es";        // servidor
int puerto = 7;                            // puerto eco
String cadTcp = "Prueba de Eco TCP";
String cadUdp = "Prueba de Eco UDP";
```

Si el lector obtiene el mensaje de conexión rechazada cuando ejecuta el programa, se debe a que el servidor no permite el acceso a ese puerto. En ese caso puede probar con otro servidor, o indicar como servidor *localhost* y uno de los puertos en los que escuchan los distintos servidores que se implementan en este mismo capítulo. Por ejemplo, si se indica `localhost` como servidor y `7777` como puerto y el servidor que implementa la aplicación `Java1711.java` está en ejecución, la ejecución del ejemplo `Java1710.java` debería ser totalmente correcta, sin mensaje de advertencia alguno. Esta circunstancia se muestra en la clase **Java1712**.

Luego está todo el código referente al protocolo TCP, que en este caso no resulta interesante, y ya se alcanza el punto del programa en el que se convierte el mensaje en un array de bytes, y se instancia un objeto que identifique al servidor. Se utiliza el método *getBytes()* de la clase **String** para convertir el mensaje a un array de bytes.

```
byte[] mensajeUdp = cadUdp.getBytes();
// Obtenemos la dirección IP del servidor
InetAddress dirIp = InetAddress.getByName( servidor );
```

La siguiente línea de código importante es la que instancia un objeto de tipo **DatagramPacket** para llevar toda la información del mensaje y del servidor, como es el array de bytes creado antes, su longitud, y la dirección y puerto del servidor.

```
DatagramPacket paquete =
  new DatagramPacket( mensajeUdp,mensajeUdp.length,dirIp,puerto );
```

A continuación, se instancia un objeto **DatagramSocket** anónimo. Esto de *anónimo* puede resultar confuso al lector. El objeto es *anónimo* porque el número de puerto no está especificado. En algún sitio anteriormente se ha indicado que los *objetos anónimos* son aquellos que no tienen una variable de referencia asignada; que no es el caso que se produce aquí. Este objeto se usa para enviar el datagrama al servidor invocando al método *send()* sobre el objeto **DatagramSocket**.

```
DatagramSocket socketDgrama = new DatagramSocket();
socketDgrama.send( paquete );
```

Las siguientes líneas de código son las que sobrescriben los datos del mensaje con una letra, para el propósito que se ha indicado antes de comprobar que el mensaje que se recibe no es en realidad el resto del que se ha mandado. Este fragmento de código también muestra al lector cómo acceder a la parte de datos de un objeto **DatagramPacket** en caso de que lo necesite para cualquier otro propósito.

```
byte[] arrayDatos = paquete.getData();
for( int cnt=0; cnt < paquete.getLength(); cnt++ )
  arrayDatos[cnt] = (byte)'x';
```

A partir de este punto, se asume que habrá una respuesta del servidor, así que se invoca el método *receive()* sobre el mismo objeto **DatagramSocket** que se ha utilizado para enviar el mensaje original al servidor. Esto bloquea la tarea, quedando a la espera de respuesta. Aunque no se ha indicado en ningún sitio, es posible utilizar el método *setTimeout()* para indicar la cantidad de tiempo que el método *receive()* estará a la espera y bloqueando, lo cual permite colocar una protección al programa para que no se quede colgado esperando una respuesta que no llegue jamás.

```
socketDgrama.receive( paquete );
```

Y el resto del código del ejemplo es la presentación en pantalla del mensaje recibido del servidor, que debe coincidir con el enviado, y el código de manejo de excepciones.

Servidor UDP

En el ejemplo `Java1711.java` se utiliza como base el ejemplo `Java1707.java`, para actualizarlo y hacer que soporte el protocolo UDP. Además, esta aplicación se completa con el ejemplo `Java1712.java`, que es un programa que permite comprobar su funcionamiento, constituyendo la parte cliente del servidor que se implementa.

Como el programa es una actualización del ejemplo `Java1707.java`, el lector deberá tener en cuenta las mismas advertencias que se realizaban al respecto de ese ejemplo, en lo que se refiere al agente de seguridad y a la configuración mediante políticas de seguridad, en caso de que decida utilizar el ejemplo para sus propios propósitos. Aquí se utiliza el mismo fichero de normas o políticas de seguridad para el control de acceso a puertos y ficheros.

En el ejemplo se implementan tres servidores. Uno de ellos es un servidor *Eco UDP* implementado a través de una tarea que monitoriza un **DatagramSocket** sobre el puerto 7777. Este servidor devuelve el array de bytes que llegue en cada datagrama que reciba, enviando esos datos de regreso al cliente que haya originado el mensaje.

El segundo servidor es un servidor *Eco TCP* implementado a través de una tarea que monitoriza un **ServerSocket** sobre el puerto 7777. Este servidor también devuelve los datos que recibe al cliente que haya realizado la conexión.

El tercer servidor es un servidor *HTTP* muy simple implementado a través de una tarea TCP que monitoriza el puerto 8090. Este servidor solamente responde al comando **GET** que se envíe desde un navegador, devolviendo un fichero como un *stream* de bytes. La inclusión de estos tres tipos de servidores es para mostrar al lector la forma en que se pueden utilizar las tareas para dar servicio a múltiples puertos, haciendo además una mezcla de protocolos TCP, UDP y HTTP.

La parte del servidor HTTP se puede comprobar a través de un navegador indicando `localhost` como nombre del servidor, y los otros servidores se pueden chequear mediante el ejemplo `Java1712.java`, pensado específicamente para probar los servidores Eco instalados en la propia máquina en que se ejecuten cliente y servidor.

Ahora llega el turno a la revisión de las partes más interesantes del ejemplo, que en este caso se limitan a las líneas de código que se han incorporado nuevas a la clase **Java1707**, para implementar el servidor *Eco* a través de UDP. Cada uno de los servidores se implementa mediante tareas, así que los tres servidores actúan concurrente y asíncronamente en diferentes tareas. Además, siempre que un objeto servidor necesite proporcionar un servicio a un cliente, lanzará otra tarea a una prioridad más baja, para dar servicio a ese cliente, siguiendo a la escucha de otros posibles clientes que requieran su atención.

Dejando a un lado la parte ya vista en el ejemplo base, el primer trozo de código en que hay que detenerse es la implementación del servidor UDP.

```
ServidorEcoUdp servidorEchoUdp = new ServidorEcoUdp();
```

Lo siguiente es el propio constructor de esa clase que se utiliza para instanciar el objeto que va a proporcionar los servicios UDP de *Eco* a través del puerto 7777. Este constructor, como se muestra, se limita a invocar a su propio método *start()* para levantarse y empezar a correr. El método *start()* invoca al método *run()*.

```
ServidorEcoUdp() {
  start();
  }
```

La siguiente línea de código muestra la instanciación del objeto **DatagramSocket** sobre el puerto 7777, que se encuentra dentro de la acción del método *run()*, es decir, es el comienzo de la tarea.

```
DatagramSocket socketDgrama = new DatagramSocket( 7777 );
```

Solamente queda el corazón de la tarea, que está formado por el bucle infinito que instancia en primer lugar un objeto **DatagramPacket** vacío, para invocar al método *receive()* sobre el **DatagramSocket**, pasándole el objeto **DatagramPacket** como parámetro.

Observe el lector que se ha limitado a 1.024 bytes los datos de entrada, por lo que no va a poder recibir mensajes de longitud mayor que ésa. Si es necesaria una longitud mayor, será suficiente con indicar el valor al constructor.

```
while( true )
  DatagramPacket paquete = new DatagramPacket( new byte[1024],1024 );
  socketDgrama.receive( paquete );
  new ConexionEcoUdp( paquete );
}
```

El método *receive()* bloquea la tarea y se queda a la espera de que llegue un paquete datagrama. Cuando esto suceda, se rellenará el objeto **DatagramPacket** vacío y se instanciará una nueva tarea de tipo **ConexionEcoUdp** para atender la petición del cliente, pasándole también como parámetro el objeto **DatagramPacket**, pero en este caso relleno con los datos recibidos en el paquete enviado por el cliente.

Una vez satisfecho el requerimiento del cliente, la tarea vuelve al inicio del bucle, instanciando un nuevo objeto **DatagramPacket** vacío y bloqueando la tarea a la espera de la llegada del siguiente paquete datagrama.

Ese objeto es guardado por el constructor para usarlo más tarde, a continuación fija la prioridad por debajo del nivel de las tareas que están monitorizando los puertos, de forma que la actividad de la tarea correspondiente a **ConexionEchoUdp** no interfiera con otras tareas que puedan lanzarse para atender a las peticiones recibidas por esos puertos. Y, por fin, se invoca el método *start()*, que a su vez invoca al método *run()* y la tarea se pone en marcha. Todo esto es lo que se hace en el código que se muestra.

```
ConexionEchoUdp( DatagramPacket paquete ) {
  System.out.println( "Recibida una llamada en el puerto 7777" );
  this.paquete = paquete;
  // Trabajamos por debajo de la prioridad de los otros puertos
  setPriority( NORM_PRIORITY-1 );
  // Se arranca el hilo y se pone a correr
  start();
  }
```

La misión encargada a esta tarea se limita al envío de una copia de los datos que le llegan en el paquete recibido, de vuelta al cliente que se lo ha enviado. La dirección y puerto de este cliente están incluidos en el paquete, donde el **DatagramSocket** del cliente los colocó antes de enviarlo.

El objeto **DatagramPacket** es casi directamente enviable de vuelta al cliente, pero para mostrar el uso de algunos de los métodos de la clase **DatagramPacket**, se construye un nuevo objeto para enviar de regreso al cliente, lo cual suele suceder más a menudo en servidores que realicen tareas más complejas. El código siguiente es el que permite extraer la información necesaria para generar un nuevo objeto. Como ejercicio al lector, podría intentar incluir la fecha en la parte de datos del objeto, para que la información devuelta al cliente sea la misma que él envió, más la fecha en que se recibió en el servidor; pero esto queda como sugerencia al lector.

```
DatagramPacket paqueteEnvio = new DatagramPacket(
  paquete.getData(),paquete.getLength(),
  paquete.getAddress(),paquete.getPort() );
```

El último código en que merece la pena detenerse en este ejemplo es la instanciación del nuevo objeto **DatagramSocket** que se va a utilizar para enviar el nuevo objeto **DatagramPacket** creado, invocando al método *send()* del socket, de

regreso al cliente. El resto del código es el cierre del socket y el tratamiento de las excepciones.

```
socketDgrama = new DatagramSocket();
// Se utiliza el nuevo socket datagrama para enviar el mensaje
// y cerrar el socket
socketDgrama.send( paqueteEnvio );
socketDgrama.close();
```

COMUNICACIONES SEGURAS

El lector habrá comprobado a lo largo de este capítulo que uno de los puntos fuertes de Java es que permite escribir potentes aplicaciones de comunicaciones en red con muy poco código. Desde el J2SE 1.4 se incorpora la posibilidad de utilizar protocolos *Secure Socket Layer* (SSL) y *Transport Layer Security* (TLS), con lo cual las aplicaciones de red implementadas con Java pueden ser, además de sencillas en su desarrollo, seguras.

El protocolo SSL está montado sobre una conexión TCP/IP normal. Proporciona un mecanismo de autenticación, asegura la privacidad y mantiene la integridad de los mensajes. Se basa en un mecanismo de clave pública para encriptar los datos, que es lento, pero resulta una de las mejores opciones a la hora de asegurar la privacidad de los datos. Además, **Sun** incluye soporte para criptografía *Rivest-Shamir-Adleman* (RSA) , el estándar *de facto* para seguridad en Internet.

El protocolo SSL está soportado a través del paquete **javax.net.ssl**, que es altamente configurable y, por lo tanto, complejo en su utilización. Sin embargo, los usos más normales de SSL son muy fáciles de implementar porque se basan en las clases **SSLSocket** y **SSLServerSocket** que funcionan de forma muy similar a las ya familiares **Socket** y **ServerSocket**.

La aplicación que se desarrolla seguidamente es una variación del cliente *echo* ya presentado en secciones anteriores, aunque en este caso se incluye la seguridad en la comunicación entre servidor y clientes. El servidor, `Java1713.java`, crea una conexión sobre un *socket servidor seguro* que atenderá conexiones desde clientes que se identifiquen con un certificado válido. El código del servidor es el que se reproduce en las siguientes líneas:

```
public class Java1713 {
  public static void main( String[] args ) throws IOException {
    // Obtenemos el objeto de tipo Factory para crear sockets SSL
    SSLServerSocketFactory fact =
      (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
    // Utilizamos el objeto para crear un socket servidor seguro
    SSLServerSocket socketServidorSsl =
      (SSLServerSocket)fact.createServerSocket( 9999 );
    SSLSocket socketSsl = (SSLSocket)socketServidorSsl.accept();
    // Creamos un canal de entrada sobre el socket seguro que
    // hemos abierto
```

```
    BufferedReader entrada = new BufferedReader(
      new InputStreamReader(socketSsl.getInputStream()));
    String linea = null;
    System.out.println( "Esperando..." );
    // Presentamos todas las líneas que vayan entrando en
    // el canal a través del socket
    while( (linea = entrada.readLine()) != null ) {
      System.out.println( linea );
      System.out.flush();
      }
    }
  }
```

Para ejecutar el servidor es necesario indicar el certificado que se utilizará. Si el lector utiliza el proporcionado junto al código fuente de esta aplicación, *claveSSL.crt*, el comando a utilizar para la ejecución del servidor será:

```
java -Djavax.net.ssl.keyStore=StoreSSL \
  -Djavax.net.ssl.keyStorePassword=cualquiera Java1713
```

La aplicación que implementa el cliente utiliza el mismo certificado para no complicar las cosas, pero podría utilizarse un archivo con una lista de certificados. El código del cliente `Java1714.java` es el que se muestra a continuación:

```
public class Java1714 {
  public static void main( String[] args ) throws IOException {
    // Obtenemos el objeto de tipo Factory para crear sockets SSL
    SSLSocketFactory fact =
      (SSLSocketFactory)SSLSocketFactory.getDefault();
    // Utilizamos el objeto para crear un socket seguro
    SSLSocket socketSsl =
      (SSLSocket)fact.createSocket( "localhost",9999 );
    // Consola desde la que leemos la entrada del usuario
    BufferedReader entrada = new BufferedReader(
      new InputStreamReader(System.in));
    // Canal de comunicación con el servidor de eco
    BufferedWriter salida = new BufferedWriter(
      new OutputStreamWriter(socketSsl.getOutputStream()));
    String linea = null;
    System.out.println( "Listo..." );
    // Vamos enviando las líneas al servidor
    while( (linea = entrada.readLine()) != null ) {
      salida.write( linea+'\n' );
      salida.flush();
      }
    }
  }
```

El comando para invocar al cliente es el siguiente:

```
java -Djavax.net.ssl.trustStore=TrustSSL \
  -Djavax.net.ssl.trustStorePassword=cualquiera Java1714
```

Para crear un certificado diferente es necesario utilizar la herramienta **KeyTool** proporcionada por la plataforma Java 2. A continuación, se reproducen los comandos

utilizados por el autor para generar el archivo de certificado *claveSSL.crt*, y se remite al lector al capítulo de applets en donde se describe el uso de esta herramienta para crear un certificado útil en la firma de applet.

El siguiente comando crea un nuevo certificado con las correspondientes claves pública y privada:

```
keytool -genkey -alias claveSsl -keyalg RSA -keystore StoreSSL
```

Este comando exporta el certificado a un archivo:

```
keytool -export -alias claveSsl -keystore StoreSSL -rfc \
  -file claveSSL.crt
```

Finalmente, es necesario incorporar el certificado al nuevo almacenamiento para permitir realizar la validación; lo cual se hace con el comando siguiente:

```
keytool -import -alias claveTSsl -file claveSSL.crt -keystore TrustSSL
```

Observando el código de las aplicaciones servidor y cliente, el lector comprobará que la arquitectura de seguridad en Java utiliza el patrón de diseño **Factory** de modo que lo primero que hay que hacer siempre es obtener un objeto de este tipo. En este caso se obtiene a través de los métodos estáticos que proporcionan los objetos por defecto **ServerSocketFactory**, en el caso del servidor, y **SocketFactory**, en el caso del cliente.

Dos métodos importantes del API de seguridad de sockets en que se incluyen estos objetos son *getDefaultCipherSuites()* y *getSupportedCipherSuites()*, que proporcionan una lista de la combinación de algoritmos criptográficos definidos para un determinado nivel de seguridad en una conexión segura basada en SSL. Es decir, define si la conexión es encriptada, si la integridad es verificada o cómo se realiza la autenticación.

Otro método importante en el caso del servidor es *getNeedClientAuth()*, que será el que determine si el socket requerirá autenticación al cliente, porque la especificación SSL indica que la autenticación del cliente es opcional.

El API de la extensión de seguridad para sockets define más métodos para el control de la conexión, todos ellos concentrados en las clases **SSLSocketFactory**, **SSLServerSocketFactory**, **SSLSocket** y **SSLServerSocket**, que permiten crear y utilizar sockets seguros.

La conexión segura utilizando el protocolo HTTPS, que es el protocolo HTTP sobre SSL, se maneja de forma muy sencilla. Es suficiente con indicar el acceso a direcciones que comiencen por *https* a través de la siguiente propiedad:

```
System.setProperty( "java.protocol.handler.pkgs",
  "com.sun.nel.ssl.internal.www.protocol" );
```

Estableciendo esta propiedad es posible acceder a una URL segura de la forma siguiente:

```
URL url = new URL( "https://digitalid.verisign.com" );
InputStream is = (InputStream)url.getContent();
```

El ejemplo Java1715.java realiza una conexión con un servidor seguro de Internet y presenta información del certificado que envía ese servidor para validar la conexión segura. Si el usuario confía en el certificado, se realizará la petición de la página inicial del servidor. Si el lector ejecuta la aplicación, el resultado que obtendrá será semejante al que reproducen en las líneas siguientes:

```
% java Java1715
digitalid.verisign.com presenta un certificado perteneciente a:
  [CN=digitalid.verisign.com,
  OU=Terms of use at www.verisign.com/rpa (c)06,
  OU=Production Security Services,
  O="VeriSign, Inc.", STREET=487 East Middlefield Road,
  L=Mountain View, ST=California, OID.2.5.4.17=94043,
  C=US, OID.1.3.6.1.4.1.311.60.2.1.1=Wilmington,
  OID.1.3.6.1.4.1.311.60.2.1.2=Delaware,
  OID.1.3.6.1.4.1.311.60.2.1.3=US,
  SERIALNUMBER=2497886]
El certificado contiene una firma validada por:
  [CN=VeriSign Class 3 Extended Validation SSL SGC CA,
  OU=Terms of use at https://www.verisign.com/rpa (c)06,
  OU=VeriSign Trust Network, O="VeriSign, Inc.", C=US]
+Confía en este certificado (s/n)? s
HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Tue, 15 Apr 2008 16:43:23 GMT
Content-length: 22539
Content-type: text/html
Last-modified: Tue, 05 Dec 2006 16:59:21 GMT
Accept-ranges: bytes
Connection: close
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>SSL Certificate Authority and Digital IDs from VeriSign,
Inc</title>
... sigue el contenido de la página
```

MULTICAST

Multicast viene a ser como la versión *broadcast* de comunicación a través de redes aplicada a Internet; es decir, de emisión de información a todo el mundo, independientemente de quién esté a la escucha. Equivale a las emisiones de televisión o radio; la señal se origina en un punto, pero alcanza a cualquiera que se encuentre en el radio de alcance de la señal y no la recibirá quien no quiera o quien no disponga del equipo necesario para recuperarla.

Las comunicaciones de este tipo son adecuadas para aquellas aplicaciones que requieren que un número de máquinas reciban los mismos datos; por ejemplo, una conferencia, un grupo de noticias, una lista de correo, etc.

La mayoría de los protocolos de red de alto nivel solamente proporcionan servicio de transmisión *unicast*; lo que significa que los nodos de la red solamente tienen capacidad de enviar la información a un nodo individual cada vez. Se trata de un servicio punto a punto. Si un nodo quiere enviar la misma información a varios destinatarios utilizando un servicio *unicast*, debe realizar tantas conexiones como destinatarios, enviando la información a cada uno de ellos separadamente.

Una forma más optimizada de enviar información desde un punto a varios destinos es proporcionar un servicio de transporte *multicast*, en el cual un único nodo puede enviar datos a múltiples destinatarios haciendo una sola llamada al servicio de transporte. Cuando el servicio *multicast* está implementado sobre una red, el rendimiento mejora notablemente, porque si el hardware soporta multicast, un paquete que sea enviado a varios recipientes puede ser lanzado a la red como un único paquete.

Grupo

En el concepto de *multicast* es fundamental la noción de **grupo**. Por definición, un mensaje multicast se envía desde un origen a un grupo de destino. En el caso de *multicasting* sobre el protocolo IP, los grupos multicast tienen un identificador de grupo que especifica el destino cada vez que se envía un mensaje multicast. Este identificador de grupo es un conjunto de direcciones IP de *clase D*, de modo que si una máquina quiere recibir un mensaje multicast enviado a un determinado grupo necesita, en cierto modo, escuchar todos los mensajes dirigidos a ese grupo.

Tipos de direcciones

En la versión 4 del protocolo IP, IPv4, hay tres tipos de direcciones: unicast, broadcast y multicast.

Las direcciones *unicast* son utilizadas para transmitir un mensaje a un único y determinado nodo destino.

Las direcciones *broadcast* son utilizadas cuando se supone que un mensaje debe ser enviado a todos los nodos de una subred.

Las direcciones *multicast* se usan cuando se quiere enviar un mensaje a un grupo de nodos que no estén necesariamente en la misma subred.

Las direcciones IP de clases A, B y C son utilizadas por los mensajes unicast. Las direcciones de clase D, en el rango 224.0.0.1 a 239.255.255.255, inclusive, junto con un puerto estándar UDP son utilizadas por los mensajes multicast.

Tiempo de vida

Otro concepto importante en lo referente a la comunicación *multicast* es el **tiempo de vida** de los paquetes. Es necesario que los paquetes tengan un tiempo de vida limitado para evitar que estén circulando por la red indefinidamente. Cada paquete tiene asegurado un tiempo de vida, un valor que se decrementa cada vez que el paquete pasa por un *hub*, un *router*, de la red. Es decir, cada paquete multicast lleva en su interior una bomba de tiempo que lo autodestruirá.

Normalmente, un paquete multicast con un tiempo de vida grande, quizás 200, tiene la garantía de alcanzar cualquier rincón del mundo.

Implementación Java

Java incluye en su paquete de comunicaciones en red la clase **MulticastSocket** para satisfacer las necesidades de comunicaciones multicast. Este tipo de socket se define en el lado cliente para escuchar los paquetes que el servidor lanza a los múltiples posibles clientes.

Esta clase, junto la clase **DatagramPacket**, vista en secciones anteriores, permiten la implementación de comunicaciones multicast de una forma bastante simple, desde el punto de vista del desarrollador Java.

Un socket de tipo **MulticastSocket** no es más que un socket UDP de tipo **DatagramSocket** con características adicionales que le permiten incorporarse a grupos de otros *hosts multicast* en la red. Cuando se necesita la incorporación a uno de estos grupos, primero hay que crear un socket **MulticastSocket** en el puerto deseado, para luego invocar al método *joinGroup()*. Para abandonar el grupo basta con hacer una llamada al método *leaveGroup()*.

Otros métodos importantes son los que permiten el control del tiempo de vida del paquete (*time-to-live*), que ofrecen la posibilidad de especificar, o conocer, a cuántos *hubs*, o *routers*, será enviado el paquete antes de que expire. Estos métodos son *setTimeToLive()* y *getTimeToLive()*.

El ejemplo `Java1716.java` implementa un servidor multicast que envía la fecha del sistema cada segundo al grupo en el cual se integra. El código del servidor es el siguiente:

```
public class Java1716 {
  static final int puerto = 1200;
  public static void main( String[] args ) throws Exception {
    DatagramPacket paquete;
    // Creamos el socket para el envío Multicast
    MulticastSocket socket = new MulticastSocket();
    // Obtenemos la dirección que identifica al ordenador
    InetAddress direccion = InetAddress.getByName( args[0] );
```

```
    System.out.println( "Dirección: "+direccion );
    // Nos incorporamos a un grupo Multicast y enviamos
    // información a ese grupo
    socket.joinGroup( direccion );
    byte[] datos = null;
    // Nos quedamos enviando la fecha y hora continuamente
    while( true ) {
      // Mandamos la información cada segundo
      Thread.sleep( 1000 );
      // Convertimos la fecha en un array de bytes para el envío
      String fecha = new Date().toString();
      datos = fecha.getBytes();
      paquete = new DatagramPacket(
        datos,datos.length,direccion,puerto );
      // Enviamos el paquete con la fecha y hora
      socket.send( paquete );
      System.out.println( "Enviados... "+datos.length+" bytes" );
      }
    }
  }
```

El código del cliente multicast de este pequeño ejemplo se implementa en el programa `Java1717.java`, que crea un socket multicast para incorporarse al grupo del servidor anterior y recibir los paquetes que éste envíe. Imprime la fecha en la ventana en que se esté ejecutando.

```
public class Java1717 {
  static final int puerto = 1200;
  public static void main( String[] args ) throws IOException {
    DatagramPacket paquete;
    // Creamos el socket para el envío Multicast
    MulticastSocket socket = new MulticastSocket( puerto );
    // Obtenemos la dirección que identifica al ordenador
    InetAddress direccion = InetAddress.getByName( args[0] );
    // Nos incorporamos a un grupo Multicast y enviamos
    // información a ese grupo
    socket.joinGroup( direccion );
    byte[] datos = new byte[40];
    // Creamos el paquete que obtendrá la información
    paquete = new DatagramPacket( datos,datos.length );
    // Nos quedamos enviando la fecha y hora continuamente
    while( true ) {
      // Quedamos a la espera de recibir información del servidor
      socket.receive( paquete );
      // Reconvertimos la trama de bytes recibidos a una cadena de
      // texto legible que presentamos en pantalla
      String fecha = new String( paquete.getData() );
      System.out.println( "Servidor: "+ paquete.getAddress()
        +" Fecha: "+ fecha );
      }
    }
  }
```

Para ejecutar las aplicaciones anteriores es necesario que el lector disponga de una configuración multicast, lo cual requiere modificar los parámetros del ordenador, lo cual requiere del lector los conocimientos necesarios. En el caso de Linux es muy

simple, ya que basta con añadir la dirección de red multicast en el fichero de configuración e incorporar la ruta para las direcciones multicast sobre uno de los dispositivos de red. Por ejemplo, si se ejecuta el comando:

```
route add -net 224.0.0.0 netmask 224.0.0.0 dev lo
```

y se añade la siguiente dirección al final del fichero de configuración */etc/hosts*:

```
224.1.1.1       EjMulticast
```

será posible ejecutar el servidor multicast con el comando:

```
% java Java1716 EjMulticast
```

apareciendo en la ventana el mensaje de envío de la información al grupo multicast.

El cliente se puede ejecutar en un terminal diferente del mismo ordenador mediante el comando:

```
% java Java1717 EjMulticast
```

y en ese terminal aparecerá la fecha del sistema en donde se ejecuta el servidor, a intervalos de un segundo. La figura 17.4 reproduce el terminal sobre el cual se ejecuta el cliente que recibe los mensajes multicast originados en el servidor.

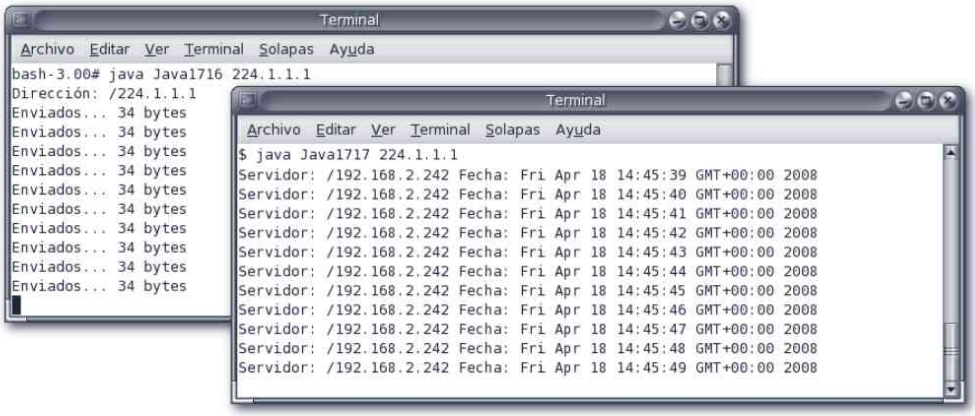


Figura 17.4

CAPÍTULO 18

SERVLETS

Los applets Java son una gran solución que ayuda a la creación de páginas dinámicas, a la hora de plantear el diseño de un sitio Web. También hay otras formas de conseguirlo, como puede ser la utilización de *JavaScript*, d*Shockwave*, *ActiveX*, *Flash*, etc., que escapan del alcance de este Tutorial.

Una forma también muy extendida, sobre todo a la hora de solicitar datos, son los *formularios*, de los que no carece casi ningún sitio Web. Los formularios son una combinación directa de código *html* y *scripts*, o programas que se ejecutan en el servidor.

Visto desde el lado del sitio Web, un servidor HTTP se limita a enviar los ficheros que solicitan los navegadores. Por ejemplo, cuando un navegador solicita el fichero `index.html`, el servidor envía ese fichero a través de la red. En el caso más complejo, como el representado por los formularios, el servidor hace llamadas a programas específicos, que se conocen como *scripts* porque, históricamente, la mayoría de ellos estaban escritos en lenguajes de este tipo, como *Perl*. Los *scripts* recuperan los datos de los formularios y construyen las páginas dinámicamente para responder a la petición. Estas páginas no existen en el disco duro en el que se encuentra el sitio Web, por eso se llaman *páginas dinámicas*.

Los *scripts* en los servidores son populares, por varias razones:

- Son totalmente independientes del navegador, al encontrarse en el servidor.
- Respuestas con cierta complejidad, se ejecutarán más rápido en el servidor.
- La seguridad es mayor, porque los programas corren bajo el control directo del administrador del servidor.

Los *scripts* estándar que corren en los servidores son los **CGI**, *o Common Gateway Interface*, que son muy simples y están ampliamente soportados. Desafortunadamente, no son muy eficientes porque el servidor lanza una copia del *script* por cada respuesta que debe generar, lo cual si el sitio Web recibe muchas visitas, significa una gran sobrecarga para el servidor. Los fabricantes han desarrollado alternativas propietarias como **ISAPI** (Microsoft) , **NSAPI** (Netscape) o **ColdFusion** (Macromedia) para paliar esta pérdida de rendimiento. *Sun Microsystems*, por su parte, ofrece una alternativa basada en Java a los CGI con la introducción de los ***servlets***.

SERVLETS Y CGI

Como se ha indicado, la forma tradicional de añadir funcionalidad a un servidor Web era a través de *scripts CGI*, en los que cada respuesta que deba generar el servidor es lanzada en un proceso independiente por una nueva instancia del programa o script CGI. Los servlets presentan varias ventajas respecto a esta aproximación:

- Un servlet no se ejecuta en un proceso separado, lo que evita tener que lanzar una nueva instancia cada vez que se solicita su intervención.
- Una vez que se ha invocado, se queda en memoria, sin consumir más recursos del servidor, aunque se llame insistentemente. Un programa CGI necesita ser cargado y descargado en cada invocación.
- Solamente hay una instancia del servlet que responde a todas las peticiones concurrentemente, lo cual, además de suponer un considerable ahorro de memoria, permite utilizar y manejar de forma muy sencilla datos persistentes.
- Un servlet puede ser ejecutado por un *motor Servlet* en una caja restringida (*Sandbox*), de la misma forma que un applet es ejecutado por un navegador en una caja restringida; lo cual permite a los proveedores de servicios de Internet admitir que sus usuarios coloquen servlets en sus páginas, protegiendo al servidor ante el uso de servlets no fiables, bien por funcionar incorrectamente o por contener código malicioso.
- Los servlets están escritos en Java, el lenguaje que el lector está aprendiendo, o sabe ya; es decir, tienen un futuro asegurado si los hados de Java siguen por donde se espera. Como consecuencia, también tienen acceso a todos los paquetes de Java y todos los JavaBeans del mercado.
- Los servlets son portables entre plataformas, siguiendo la premisa en la que se basa Java, "*escribir una vez, ejecutar en cualquier lugar*". *Netscape*, *Apache* e *IIS*, probablemente los tres servidores Web más populares, están soportados, y muchas otras compañías han anunciado el propósito de soportar el API servlet en sus productos.
- Los servlets, al estar escritos en Java, disfrutan de las características de seguridad inherentes a Java, como puede ser el manejo de memoria.
- Los servlets son mucho más eficientes que los CGI.

API SERVLET

Los servlets son clases normales Java que se crean cuando se necesitan y se destruyen cuando ya no se van a usar más. Los servlets se ejecutan en una caja restringida, mediante un motor especial, que es el encargado de la creación y destrucción de cada uno de los servlets de la aplicación Web, mediante los métodos *init()* y *destroy()*, respectivamente. Este motor de servlets es la implementación de la especificación Servlet y es el responsable de mantener el ciclo de vida del servlet, y se puede usar solo o en combinación con un servidor Web. La implementación oficial de *Sun* de la especificación Servlet y, por tanto, el motor de servlets oficial, es el contenedor **Jakarta-Tomcat**, desarrollado por la *Apache Software Foundation*.

Los servlets no son específicos para cada servidor Web, el API es flexible y soporta muchos servidores Internet; por ejemplo, ***Marimba*** puede reemplazar su *plugin* propietario por servlets. Según *Sun Microsystems*, la mayoría de los programadores que están interesados en los servlets, lo están en el **HttpServlet** concretamente, que será tratado detenidamente a continuación.

Al crear un servlet extendiendo la clase **HttpServlet**, la clase debe colocarse en el directorio */WEB-INF/classes* de la aplicación Web. La instancia del servlet será generada por el motor en dos casos:

1. Cuando se indica al motor específicamente que cargue el servlet al iniciarse la aplicación. Para ello, en el descriptor de la aplicación, *web.xml*, se ha de colocar un valor distinto de cero en el atributo `load-on-startup`.
2. Si el servlet no está cargado en la zona restringida de ejecución previamente, el motor lo cargará cuando reciba la primera petición destinada a ese servlet.

El API Servlet es claro y simple. Un servlet es una clase Java que implementa la interfaz **Servlet**, que define cinco métodos:

`service()`, es el corazón de los servlets. El servidor invoca al método *service()* para ejecutar respuestas. El método *service()* acepta como parámetros objetos **ServletRequest**, que encapsulan la petición del cliente, y **ServletResponse**, que disponen de métodos para devolver información al cliente.

`init()`, es el lugar donde se inicializa el servlet. Es un método que acepta como parámetro un objeto de tipo **ServletConfig**, que contiene la configuración del servidor, y se garantiza que solamente se llamará una vez durante la vida del servlet.

`destroy()`, libera el servlet, se llama cada vez que el servlet debe ser descargado; por ejemplo, debido a que hay que cargar una nueva versión del servlet. Todos los recursos del sistema bloqueados por *init()* son liberados al invocar este método y se garantiza que solamente se llamará una vez durante la vida del servlet.

`getServletConfig()`, devuelve una referencia al objeto **ServletConfig** que se pasa como parámetro al método *init()*, proporcionando información de la configuración del servlet.

`log()`, escribe mensajes en los ficheros de *log* del motor de servlets.

`getServletContext()`, devuelve una referencia al objeto **ServletContext** que se utiliza para interaccionar con otros componentes de la aplicación Web: páginas JavaServer Pages (JSP), JavaBeans y otros servlets. **ServletContext** representa a la aplicación Web completa.

Para asegurar un óptimo rendimiento, el servidor solamente carga una instancia de cada servlet. Una ver cargado, el servlet permanece en memoria, estando disponible en cualquier instante para procesar cualquier petición. Por lo tanto, varias tareas pueden llamar simultáneamente al método *service()*, así que la sincronización dentro de *service()* debe ser una premisa a no olvidar jamás. La figura 18.1 representa el ciclo de vida de un servlet.

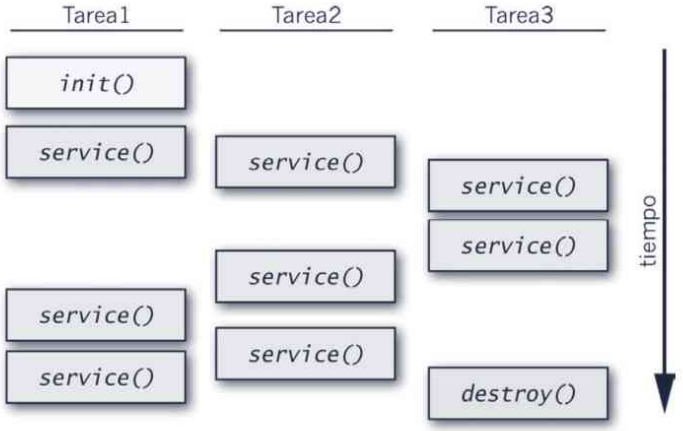


Figura 18.1

LA CLASE HTTPSERVLET

La clase **HttpServlet** es una clase que implementa la interfaz **Servlet** incorporando además métodos específicos para servidores Web. Un uso típico de **HttpServlet** es el procesamiento de formularios *html*. Pero antes de poder escribir el primer servlet, es necesario tener unas nociones básicas sobre el protocolo **HTTP**, *HyperText Transfer Protocol*, que es el protocolo de comunicaciones que se utiliza para que un cliente, un navegador, por ejemplo, envíe peticiones a un servidor Web.

HTTP es un protocolo orientado a petición-respuesta. Una petición HTTP está formada por unos campos de cabecera y un cuerpo, que puede estar vacío. Una respuesta HTTP contiene un código de resultado y de nuevo una cabecera y un cuerpo.

El método *service()* de la clase **HttpServlet** lanza diferentes peticiones a distintos métodos Java para sistemas de petición diferentes. Reconoce los métodos estándar en

formato *HTTP/1.1* y no es conveniente sobrecargarlo en subclases, a no ser que se necesiten implementar métodos adicionales. Los sistemas o métodos de petición que reconoce son GET, HEAD, PUT, POST, DELETE, OPTIONS y TRACE; cualquier otro método obtendrá como respuesta un error HTTP de tipo *Bad Request*. Para cada uno de los métodos anteriores, Java lanza un método de tipo *doXxx()*, por ejemplo, para GET lanza *doGet()*. Todos los métodos esperan dos parámetros:

```
HttpServletRequest peticion,
HttpServletResponse respuesta
```

Los métodos *doOptions()* y *doTrace()* disponen de una implementación por defecto y no está permitido sobrecargarlos. El método HEAD, que se supone que devuelve las mismas líneas de cabecera que podría devolver el método GET, sin incluir el cuerpo. Así que los métodos que se pueden utilizar van a ser *doGet()*, *doPut()*, *doPost()* y *doDelete()*, cuyas implementaciones por defecto en la clase **HttpServlet** devuelven un error HTTP de *tipo Bad Request*. Cualquier subclase de **HttpServlet** debe sobrecargar uno o más de estos métodos para proporcionar la adecuada implementación a las acciones que desea realizar.

Los datos de la petición se pasan a todos los métodos como primer argumento de tipo **HttpServletRequest**, que es una subclase de la clase más general **ServletRequest**. Las respuestas que pueden crear los distintos métodos se devuelven en el segundo argumento de tipo **HttpServletResponse**, que es una subclase de **ServletResponse**.

Cuando se solicita una página desde un navegador, se está utilizando el método GET. La petición GET no tiene cuerpo, es decir, el cuerpo de la petición está vacío. La respuesta debería contener un cuerpo con los datos de la página solicitada y campos de cabecera que describan la información que contiene el cuerpo; de estos campos de cabecera, son especialmente útiles los campos *Content-Type* y *Content-Encoding*.

A la hora de enviar datos, de un formulario, por ejemplo, se puede utilizar tanto el método GET como POST. La diferencia entre uno y otro es que en una petición GET los parámetros se codifican en la *URL*, o dirección de la página, mientras que en una petición POST se transmiten en el cuerpo. Los editores HTML y otras herramientas de descarga utilizan peticiones PUT para descargar recursos en un servidor Web y luego peticiones DELETE para eliminar esos recursos.

HolaMundoServlet

Para empezar con un servlet sencillo, se propone al lector un servlet que envíe como respuesta a toda petición un mensaje de saludo, y para ser originales, podría tratarse de "Hola Mundo!". El código es muy sencillo, tal como se muestra a continuación, que corresponde al fichero fuente HolaMundoServlet.java.

```
import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;

public class HolaMundoServlet extends HttpServlet {
  protected void doGet(
    HttpServletRequest req,HttpServletResponse res )
    throws ServletException,IOException {
    res.setContentType( "text/html" );
    PrintWriter out = res.getWriter();
    out.println(
      "<HTML><HEAD><TITLE>Hola Mundo!</TITLE>" +
      "</HEAD><BODY>Hola Mundo!</BODY></HTML>" );
    out.close();
    }

  public String getServletInfo() {
    return "HolaMundoServlet, Tutorial de Java (C)A.Froufe";
    }
  }
```

Antes de entrar a ver detenidamente las partes interesantes de ese código, en el que al tratarse del primer ejemplo, cada línea resulta interesante; es conveniente explicar al lector la forma de ejecutar el servlet, que solamente puede verse cuando es proporcionado a través de un servidor Web. A continuación, se indica la forma de arrancarlo utilizando el motor **Jakarta-Tomcat**, que es la implementación oficial de la especificación Servlet.

En la especificación Servlet se introduce el concepto de ***aplicación Web***, que consiste en una colección de servlets, páginas JSP, clases Java, documentos estáticos: HTML, XHTML, imágenes, etc. y otros recursos que pueden ser empaquetados y ejecutados en distintos servidores de proveedores de servicios diferentes. El contenedor que alberga una aplicación Web consiste en una estructura de directorios para la ejecución de la aplicación Web.

En el caso de utilizar la implementación **Jakarta-Tomcat**, el directorio a partir del cual se instala cualquier aplicación Web debe ser *CATALINA_HOME/webapps*, en donde `CATALINA_HOME` apunta al directorio de instalación de **Jakarta-Tomcat**. Los directorios más importantes de la aplicación, aparte del de la propia aplicación, son el directorio *WEB-INF*, en donde se coloca el archivo *web.xml* que contiene la descripción de la configuración de la aplicación, y el directorio *classes*, en donde se colocan los servlets y clases complementarias necesarias para la ejecución de la aplicación.

Los pasos a seguir para compilar y ejecutar el servlet son los que se describen en los siguientes puntos, partiendo de la premisa de que el motor de servlets **Jakarta-Tomcat** está correctamente instalado:

- Crear el directorio *tutorial*, colgando del directorio de instalación de **Tomcat**: *CATALINA_HOME/webapps*. Colgando del directorio *holamundo*, crear el directorio *WEB-INF*, y colgando de este último, crear el directorio *classes*.

- Compilar el archivo `HolaMundoServlet.java` y colocar el archivo *.class* obtenido en el directorio *WEB-INF/classes*. Probablemente se generen errores porque el JDK no incluye la librería de servlets, que sí la proporciona la implementación de **Tomcat**. Para poder compilar o se incluye la ruta de la librería de Tomcat, ***servlet-api.jar***, en el `CLASSPATH` de compilación, o se copia ese fichero JAR de la implementación servlet en el directorio *jre/lib/ext* de la instalación del JDK. Una vez realizada cualquiera de las dos acciones anteriores, el archivo se compilará sin errores.
- Crear el descriptor de la aplicación, *web.xml*, y colocarlo en el directorio *WEB-INF*. El contenido del archivo debe ser el que se reproduce a continuación.

```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">

  <display-name>Tutorial de Java, Servlets</display-name>
  <description>
    Ejemplo básico de servlet: HolaMundo.
  </description>

  <!-- Se define el servlet dentro de la aplicación web -->
  <servlet>
    <servlet-name>HolaMundo Servlet</servlet-name>
    <servlet-class>HolaMundoServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HolaMundo Servlet</servlet-name>
    <url-pattern>/holamundo</url-pattern>
  </servlet-mapping>
</web-app>
```

- Arrancar el motor **Jakarta-Tomcat**. Para lanzar y parar este servidor, los comandos que se utilizan en *Windows*, desde una ventana de *Símbolo del Sistema*, son los siguientes, respectivamente:

```
CATALINA_HOME\bin\startup.bat
CATALINA_HOME\bin\shutdown.bat
```

- Una vez arrancado el servidor con `startup.bat`, ya es posible acceder al servlet a través de un navegador, introduciendo la dirección:

```
http://localhost:8080/tutorial/holamundo
```

- El resultado de la ejecución del servlet, y como resumen de todos los pasos anteriores, se muestra en la figura 18.2, que corresponde al navegador *Firefox*, donde se muestra la resolución de la ejecución del servlet.

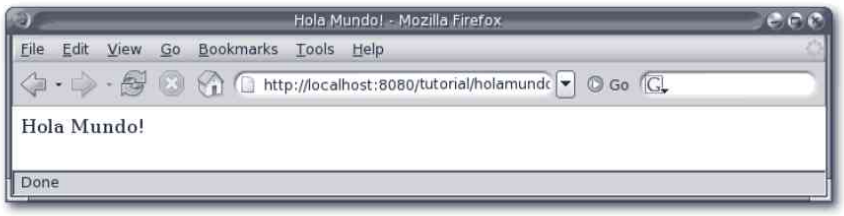


Figura 18.2

En el fichero `readme` de **Tomcat**, hay la suficiente información del uso de este motor servlet y servidor Web, en caso de que el lector se encuentre con problemas a la hora de poder ejecutar el ejemplo, o sus propios servlets.

En el soporte digital de este Tutorial, se adjunta un fichero empaquetado *ZIP* con todas las aplicaciones Web de este capítulo, listas para ser desplegadas en el directorio de instalación de *Jakarta-Tomcat*, para que el lector pueda tomar como referencia la estructura que sigue.

Las partes del código interesantes, como se ha dicho, son todas, así que a continuación se comentan cada una de las líneas de código del ejemplo. Las primeras líneas importan algunos paquetes que contienen las clases utilizadas por el servlet; la mayoría de los servlets necesitarán clases de estos paquetes.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

La siguiente línea de código es la declaración de la clase, que en este caso extiende la clase base estándar de los servlets HTTP.

```
public class HolaMundoServlet extends HttpServlet {
```

Ahora se encuentra la sobrecarga del método *doGet()* de la clase **HttpServlet**, que obliga a capturar las excepciones de tipo **ServletException** e **IOException**.

```
protected void doGet(
  HttpServletRequest req,HttpServletResponse res )
  throws ServletException,IOException {
```

Dentro del método *doGet()*, la siguiente línea de código muestra el uso de uno de los métodos del objeto **HttpServletResponse** para fijar el tipo de contenido que se va a devolver como respuesta. Todas las cabeceras que se vayan a enviar como respuesta a una petición realizada al servlet, han de colocarse antes de que se obtenga un objeto de tipo **PrintWriter** o **ServletOutputStream** para escribir el cuerpo de la respuesta.

```
res.setContentType( "text/html" );
```

La siguiente línea crea un objeto de tipo **PrintWriter** para colocar el texto del mensaje de respuesta a la petición.

```
PrintWriter out = res.getWriter();
```

En el siguiente trozo de código se muestra el uso del objeto **PrintWriter** para escribir el texto de tipo *text/html*, que constituirá la respuesta devuelta por este servlet.

```
mensaje.println(
  "<HTML><HEAD><TITLE>Hola Mundo!</TITLE>" +
  "</HEAD><BODY>Hola Mundo!</BODY></HTML>" );
```

Lo último que se hace en el método *doGet()* es cerrar el objeto **PrintWriter** cuando ya se ha terminado de escribir el mensaje.

```
mensaje.close();
```

La verdad es que este cierre explícito se incluye por coherencia, porque el servidor Web cierra el objeto **PrintWriter** o **ServletOutputStream** automáticamente cuando la ejecución del servicio le devuelve el control. La llamada directa al método *close()* es útil cuando se quiere hacer algún post-procesado una vez la respuesta que se va a devolver al cliente ya esté construida. Esta llamada a *close()* indica al servidor Web que la respuesta está creada y que la conexión con el cliente se puede cortar.

Las líneas siguientes, que corresponden al método sobrecargado *getServletInfo()*, se supone que devuelven información sobre el servlet. La verdad es que en este servlet de ejemplo esta información no aporta nada de valor al usuario del servlet, pero su uso sí puede resultar interesante para las herramientas de administración del servidor Web.

```
public String getServletInfo() {
  return "HolaMundoServlet, Tutorial de Java (C)A.Froufe";
  }
```

Eventos de aplicación

Los eventos de aplicación proporcionan control sobre la aplicación Web. Los más importantes son los eventos de inicio y final de la aplicación y los que permiten crear y posteriormente invalidar la sesión. Las interfaces **ServletContextListener** y **HttpSessionListener** son las que se deben implementar para la creación de este tipo de eventos.

La interfaz **ServletContextListener** contiene los métodos *contextInitialized()*, que es invocado durante la inicialización de la aplicación, y *contextDestroyed()*, que es invocado durante el proceso de finalización de la aplicación.

El ejemplo `AplicacionEvt.java` implementa esta interfaz, almacenando en una variable de la clase el momento en que se arranca la aplicación. Su código se reproduce seguidamente:

```
public class AplicacionEvt implements ServletContextListener {
  private static long inicio = 0L;
  // Evento de lanzamiento de la aplicación
  public void contextInitialized( ServletContextEvent evt ) {
    inicio = System.currentTimeMillis();
```

```
    }
  // Evento de cierre de la aplicación
  public void contextDestroyed( ServletContextEvent evt ) {
    }

  // Devuelve el instante en que se arrancó la aplicación
  public long getInicializacion() {
    return( inicio );
    }
  }
```

La interfaz **HttpSessionListener** contiene los métodos *sessionCreated()*, invocado durante la creación de la sesión con el servidor, y *sessionDestroyed()*, que es invocado durante el proceso de invalidación o destrucción de la sesión.

El ejemplo `SesionEvt.java` implementa esta interfaz y mantiene un contador que indica el número de sesiones activas establecidas con el servidor. El código de este programa es el siguiente:

```
public class SesionEvt implements HttpSessionListener {
  private static int sesionesActivas = 0;

  // Evento de creación de sesiones
  public void sessionCreated( HttpSessionEvent evt ) {
    sesionesActivas++;
    }

  // Evento de invalidación de sesiones
  public void sessionDestroyed( HttpSessionEvent evt ) {
    if( sesionesActivas > 0 )
      sesionesActivas--;
    }

  // Devuelve el número de sesiones activas
  public int getSesionesActivas() {
    return( sesionesActivas );
    }
  }
```

La aplicación `Java1801.java` es la que se encarga de extender la clase **HttpServlet** y generar la respuesta al usuario en base a la información proporcionada por las dos clases anteriores. Para que las clases que implementan los receptores de eventos estén a disposición del servidor de aplicaciones, es necesario colocarlas bajo la etiqueta `<listener>` en el archivo de configuración de la aplicación, *web.xml*, que en este caso concreto quedaría de la forma que se muestra seguidamente:

```
<web-app>
. . .
  <!-- Se define el receptor de eventos de la sesión -->
  <listener>
    <listener-class>SesionEvt</listener-class>
  </listener>
  <!-- Se define el receptor de eventos de la aplicación -->
  <listener>
```

```
    <listener-class>AplicacionEvt</listener-class>
  </listener>

  <!-- Se definen el servlet dentro de la aplicación web -->
  <servlet>
    <servlet-name>Eventos Servlet</servlet-name>
    <servlet-class>Java1801</servlet-class>
  </servlet>

  <!-- Se definen las URL a las que atiende el servlet -->
  <servlet-mapping>
    <servlet-name>Eventos Servlet</servlet-name>
    <url-pattern>/java1801</url-pattern>
  </servlet-mapping>
. . .
</web-app>
```

Cada vez que el servlet se arranque o se destruya, o una sesión sea creada o destruida, el servidor de aplicaciones invocará a los métodos correspondientes al evento producido. Para la ejecución del servlet y ver el resultado en un navegador, el texto anterior se incluye en el fichero web.xml del directorio tutorial creado anteriormente y los ficheros .class se colocan en el directorio WEB-INF/classes, que servirá para el resto de los ejemplos del capítulo. Y ya es suficiente con introducir la URL que se indica a continuación en el navegador para ver el servlet en ejecución:

```
http://localhost:8080/tutorial/java1801
```

Diccionario

El ejemplo `Java1802.java` también muestra el uso de **HttpServlet**, aunque en este caso realiza una tarea un poco más complicada, consistente en una búsqueda en un diccionario, devolviendo la definición a través de una página Web dinámica.

Los servlets procesan cada petición en tres pasos:

1. Analizan la petición; por ejemplo, desde un formulario, se captarían los valores de varios campos.
2. Ejecutan la petición; por ejemplo, buscan en la base de datos la información que se requiere.
3. Crean la respuesta; por ejemplo, formatean el resultado de la consulta para presentarlo en una página Web.

A continuación, se desarrollan estos tres puntos en base al ejemplo propuesto, utilizando un objeto **ServletOutputStream**, en lugar de un objeto **PrintWriter**, para que el lector pueda comprobar su uso. Cuando se analiza la petición, el servlet trabaja con el objeto **ServletRequest**; aunque en este caso, como en el anterior, sea con la versión especializada del objeto anterior **HttpServletRequest**.

El primer paso consiste en determinar cómo ha enviado el navegador los parámetros, los campos del formulario y otros datos. El navegador puede hacerlo a través de dos métodos: POST y GET.

La clase **HttpServlet** proporciona un método *service()* por defecto, que analiza la petición y la envía al método adecuado según la petición: *doPost()* o *doGet()*. Ésta es una de las ventajas de utilizar el **HttpServlet** en lugar del servlet genérico; lo cual en la práctica se va a traducir en que lo que hay que hacer es sobrescribir los métodos *doGet()* o *doPost()*, en vez del método *service()*.

POST y GET difieren en la forma de pasar los datos del navegador al servlet. GET codifica los parámetros en la URL, o dirección de la página, mientras que POST los pasa por separado. Los servlets pueden soportar cualquiera de los dos métodos, o los dos, es algo que debe decidir el programador, aunque en el ejemplo se soporten los dos, hay muchas ocasiones en que no tiene sentido el que se proporcione soporte para alguno de ellos. La decisión de usar GET o POST viene condicionada por el código *html*, ya que una vez que el dato alcanza al servidor, no hay nada que el servlet pueda hacer para devolver un error en caso de que la llamada se realice a través de un método no soportado.

Con hiperenlaces, la única solución es utilizar GET. Los formularios pueden soportar cualquiera de los dos métodos, pero siempre teniendo bien presente que si los parámetros son parte de un URL, GET siempre es una mejor solución que enviar los datos en pequeños paquetes; aunque cuando se hace un uso inadecuado de GET, se pueden provocar problemas de seguridad para el propio ordenador o el servidor; los navegadores mantienen una historia exhaustiva de los URL, así que no debe pasarse información confidencial, como contraseñas o palabras de paso, en las URL.

POST

POST solamente está disponible para el tratamiento de formularios. El código *html* de un formulario sería:

```
<FORM ACTION="http: //localhost:8080/tutorial/java1802" METHOD="POST">
<INPUT NAME="pregunta">
<INPUT TYPE="submit">
</FORM>
```

La marca <FORM> necesita dos parámetros: ACTION, que es el URL del servlet y METHOD, que fuerza el método a POST, ya que por defecto es GET. Le siguen las marcas <INPUT>, de las cuales la primera es una entrada de texto y la segunda un botón.

En el servlet, el dato del formulario es enviado al método *doPost()*, que se implementa como reproducen las siguientes líneas de código:

```
protected void doPost( HttpServletRequest req,
  HttpServletResponse res ) {
  String clave = req.getParameter( "pregunta" );
```

```
  String definicion = getDefinicion( clave );
  creaRespuesta( clave,definicion,res );
  }
```

Como se puede ver, el servlet recupera el campo del formulario a través del método *getParameter()*, utilizando el nombre de la marca `<INPUT>` para identificar los parámetros.

GET

El método `GET` también funciona con formularios, pero es mucho más útil en el caso de los hiperenlaces. La siguiente línea es un ejemplo de un hiperenlace que salta a una página dinámica:

```
<A HREF="http://localhost:8080/tutorial/java1802?java>
Definición de Java</A>
```

en esta línea, la parte del principio, antes del interrogante -?-, apunta al servlet. El parámetro aparece después del interrogante.

En el servlet, la petición es enviada al método *doGet()*:

```
protected void doGet( HttpServletRequest req,
  HttpServletResponse res ) {
  String clave = req.getQueryString();
  String definicion = getDefinicion( clave );
  creaRespuesta( clave,definicion,res );
  }
```

De nuevo el servlet recupera los parámetros del objeto **HttpServletRequest**. Aquí se utiliza el método *getQueryString()*, que trabaja muy bien si solamente hay un parámetro; en caso de que haya varios parámetros, esto también funciona, pero es necesario utilizar parámetros con nombre, al igual que lo hace `POST`. Una dirección URL con parámetros con nombre es de la forma:

```
http://localhost:8080/tutorial/java1802?detalle=total&frame=no
```

Los nombres de los parámetros y sus valores se agrupan en parejas unidas por un *ampersand* -&-. Este hiperenlace tiene dos parámetros: `detalle` y `frame`, y sus valores son *total* y *no*, respectivamente.

Después de analizar la petición, el servlet debe preparar la respuesta. Normalmente el servlet se conecta a una base de datos o lee ficheros en el servidor. Otra opción muy popular es crear un mensaje de correo electrónico, pero el límite sólo está impuesto por la imaginación del creador del sitio Web a que se aplica. En el ejemplo que se está tratando, el servlet comprueba la palabra que se pasa contra su diccionario:

```
protected String getDefinicion( String palabra ) {
```

Después de ejecutar la petición, el servlet debe devolver una respuesta al cliente. Para comunicarse con él, el servlet utiliza el objeto **HttpServletResponse**, que es una versión especializada del objeto genérico **ServletResponse**.

Hay dos respuestas posibles: indicar que se ha producido un error, o devolver algún resultado. Si se produce un error, el servlet podrá lanzar una excepción o devolver el error a través del método *sendError()*.

Si no se produce ningún error, el servlet creará dinámicamente una página Web. Para devolverla al cliente, el servlet ha de indicar en primer lugar el tipo de contenido *MIME* de la respuesta con *setContextType()*. Las páginas Web son de tipo *text/html*, pero los servlets pueden devolver datos en cualquier formato, incluyendo imágenes `gif` (*image/gif*), `png` (*image/png*) o `jpeg` (*image/jpeg*).

Adicionalmente, un servlet puede indicar la longitud del contenido con la llamada al método *setContextLength()*. La indicación de la longitud es opcional y sólo debe utilizarse si no resulta muy costoso. La indicación de una longitud errónea es peligrosa, porque el servidor puede generar una excepción. Finalmente, el servlet escribe su respuesta en el canal de salida devuelto por el método *getOutputStream()*.

En el ejemplo, la página Web se crea en las líneas de código siguientes:

```
protected void creaRespuesta( String clave,String definicion,
  HttpServletResponse res ) throws IOException {
  // Primero prepara la respuesta como una página Web, usando
  // un StringBuffer para almacenarla, a fin de poder calcular
  // la longitud de la página
  StringBuffer buffer = new StringBuffer();
    buffer.append( "<HTML>\n" );
    buffer.append( "<HEAD>\n" );
    buffer.append( "<TITLE>Tutorial de Java, Servlets</TITLE>\n" );
    buffer.append( "</HEAD>\n" );
    buffer.append( "<BODY BGCOLOR=white TEXT=black LINK=blue>\n");
    buffer.append( "<H2>Servlets, Diccionario</H2>\n" );
    buffer.append( "<P>La respuesta encontrada para <EM>" );
    buffer.append( clave );
    buffer.append( "</EM> es la siguiente:\n<P><BLOCKQUOTE>" );
    buffer.append( definicion );
    buffer.append( "</BLOCKQUOTE>\n" );
    buffer.append( "<table width=70% cellspacing=0 "+
      "cellpadding=0 border=0>\n");
    buffer.append( "<tr bgcolor=#0000ff><td align=center "+
      "height=16>\n");
    buffer.append( "<font face=Arial,Helvetica size=1\n");
    buffer.append( "color=#FFFFFF>&copy; 2008, Agustin Froufe\n");
    buffer.append( "</font></td></tr></table>\n");
    buffer.append( "</BODY>\n" );
    buffer.append( "</HTML>" );
  // Ahora ya se pasa la respuesta al navegador
  res.setContentType( "text/html" );
  res.setContentLength( buffer.length() );
  res.getOutputStream().print( buffer.toString() );
```

```
}
```

El servlet de este ejemplo construye la página Web en memoria, para poder calcular la longitud, fija el tipo de contenido, la longitud de la página y envía la página al navegador.

EJECUCIÓN DEL SERVLET

Para probar el funcionamiento del servlet, es imprescindible un servidor Web que implemente la interfaz **Servlet**. En caso de que el lector utilice uno con soporte para servlets, debe seguir las instrucciones del manual para instalar el servlet implementado por la clase **Java1802**, pero de nuevo, la solución más sencilla para probar el funcionamiento del servlet es utilizar el ya conocido ***Jakarta-Tomcat*** que es la implementación oficial de la especificación API Servlet.

El servlet solamente necesita un parámetro: `dbpath`, que es el camino completo donde se encuentra el fichero diccionario al que accede para buscar las claves que se le pasan desde el navegador. Deben seguirse escrupulosamente las instrucciones del servidor Web para la declaración de variables y declarar `dbpath`. Si el lector utiliza un servidor Web con soporte para servlets, asegúrese de entender bien la forma de declarar variables, porque los errores en su declaración son muy frecuentes e impiden el correcto funcionamiento del servlet. En el caso de **Tomcat**, es suficiente con colocar el parámetro `dbpath` en el descriptor de la aplicación, *web.xml*.

El diccionario utilizado en este ejemplo es un fichero muy simple, *db.txt*, construido en base a líneas que contienen parejas de claves y definiciones, del tipo

```
clave1=definicion1
clave2=definicion2
clave3=definicion3
```

asegurándose de escribir siempre las claves en minúsculas. Este fichero debe estar colocado en el directorio *tutorial*.

Para probar el funcionamiento del servlet, deben introducirse las siguientes líneas en el archivo *web.xml* del directorio *WEB-INF* del directorio *tutorial*, que define la configuración de la aplicación:

```
<servlet>
  <servlet-name>Diccionario Servlet</servlet-name>
  <servlet-class>Java1802</servlet-class>
  <init-param>
    <param-name>dbpath</param-name>
    <param-value>db.txt</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>Diccionario Servlet</servlet-name>
  <url-pattern>/java1802</url-pattern>
```

```
  </servlet-mapping>
C
```

Seguidamente ya es posible arrancar el servidor e introducir en cualquier navegador la dirección en donde contesta el servlet, indicando la clave de la cual se quiere obtener su definición:

```
http://localhost:8080/tutorial/java1802?java
```

aunque esto debe adaptarse al nombre del servidor que utilice el lector.

Seguimiento de sesiones

El *seguimiento de sesiones* consiste básicamente en reconocer la petición de un mismo usuario en distintas ocasiones. Es decir, un usuario realiza una petición de una página a un servidor Web y se identifica; mientras no cierre la sesión, el servidor reconocerá que las peticiones corresponden a ese usuario concreto a través del mecanismo de seguimiento de sesiones. Esta característica es muy importante a la hora de la implementación de aplicaciones en las que se trate de fidelización de usuarios, carritos de compra, servidores de información, etc.

Las técnicas para identificar servicios y asignar datos a ellos se basan en *Cookies*, en la reescritura de URL o en el uso de campos HTML ocultos. Todas ellas necesitan programación adicional para su control. Sin embargo, es posible utilizar la clase **HttpSession** del API de los servlets para eliminar la necesidad de programación alguna, ya que **HttpSession** permite ver y manejar la información correspondiente a una sesión y asegurar que esa sesión persiste a través de múltiples conexiones del mismo usuario.

El uso de *Cookies* es quizás la forma más conocida de realizar el seguimiento de sesiones. Los Cookies almacenan información acerca de la sesión en la máquina cliente, de forma que sesiones posteriores puedan acceder a esos Cookies para recuperar la información. El servidor asocia un identificador de sesión a partir de la información del Cookie. El problema se presenta cuando hay múltiples Cookies involucrados, o cuando hay que decidir que un Cookie expire o cuando hay que generar identificadores únicos de sesión. Además, los Cookies tienen una serie de restricciones en su uso, por ejemplo, un Cookie no puede crecer más de de 4 kbytes y no puede haber más de 20 Cookies por dominio. Y, por encima de todo, está el problema de la desconfianza de los usuarios, que al almacenar información sensible en un Cookie, como puede ser el número de una tarjeta de crédito, puede decidir que no se utilicen Cookies y deshabilitarlos. En resumen, no es buena práctica limitarse al uso exclusivo de Cookies para el seguimiento de sesiones.

La *reescritura de URL* consiste en incorporar un identificador de sesión a cada URL, de forma que el servidor pueda asociar el identificador con los datos correspondientes a una sesión. La URL se construye utilizando el método `GET` de HTTP y puede incluir cualquier número de parejas de tipo parámetro-valor. Esta

solución, aunque suele generar URL muy largas, es una buena opción cuando los Cookies están deshabilitados, aunque hay que tener ciertas precauciones, como asegurarse de que se añade el identificador de la sesión a cada URL y tener cuidado con la pérdida del identificador cuando el usuario deja la sesión y regresa utilizando un enlace de tipo *favorito* (*bookmark*), en cuyo caso el identificador se pierde.

Los *campos ocultos* de un formulario también se pueden utilizar para guardar los datos de la sesión, que posteriormente se pueden recuperar utilizando el objeto **HTTPServletRequest**. Cuando un formulario es enviado, los datos de los campos ocultos se incluyen tanto si el modo de envío es `GET` como `POST`. Sin embargo, el problema es que los formularios solamente se pueden utilizar en páginas generadas dinámicamente, por lo que su uso es limitado y, además, tienen el inconveniente de que el receptor puede ver los datos almacenados y ver el código fuente de la página HTML.

La clase HttpSession

Independientemente de la técnica utilizada, lo que está claro es que hay que almacenar los datos de la sesión en algún sitio. Una buena opción es el objeto **HttpSession** que puede almacenar los datos de la sesión en el servidor. Para utilizar este objeto es necesario obtener un objeto de la sesión, leerlo y escribirlo, y eliminarlo cuando se cierre la sesión o después de un cierto tiempo si la sesión no se ha cerrado expresamente.

La persistencia del objeto es válida en el contexto de la aplicación Web, por lo que puede ser compartida por varios servlets. Un servlet puede acceder a objetos almacenados en otro servlet. Estos objetos, también llamados *atributos*, pueden estar disponibles para otros servlets dentro del ámbito de la petición, sesión o aplicación.

El objeto **HttpSession** almacena y accede a la información de la sesión que controla el servlet, por lo que no es necesario que esta información sea manipulada en el código. Los métodos más interesantes de esta clase son los que se describen a continuación.

`getId()`, devuelve una cadena conteniendo el identificador único asignado a la sesión. Es el utilizado en la reescritura de URL para identificar a esa sesión.

`isNew()`, devuelve `true` si el cliente no ha establecido ya una sesión. Si el cliente tiene deshabilitados los Cookies, la sesión será *nueva* en cada petición.

`setAttribute()`, asigna un objeto a la sesión, utilizando un nombre determinado.

`getAttribute()`, devuelve el objeto asignado a la sesión, identificado por el nombre que se indique.

`setMaxInactiveInterval()`, especifica el tiempo máximo que ha de pasar entre peticiones consecutivas del cliente para que la sesión se dé como no válida. Un

número negativo como argumento de este método hará que nunca se invalide la sesión.

`invalidate()`, elimina la sesión actual y libera todos los objetos que estuviesen asociados a ella.

Páginas JSP

La tecnología *JavaServer Pages* (JSP) ha sido introducida para ayudar a los diseñadores y desarrolladores de aplicaciones Web a separar la lógica de la presentación, de la lógica de la aplicación; es decir, para permitir separar el contenido dinámico del contenido estático de una página HTML. Utilizando esta tecnología es posible cambiar el diseño de la página sin necesidad de que haya que tocar el código que genera el contenido de esa página.

La tecnología JSP es una tecnología de servidor, es decir, que todo su proceso se realiza en el servidor, lo cual permite que otros componentes como JavaBeans o *Enterprise JavaBeans* (EJB) puedan interacturar con las páginas JSP; de hecho, los servlets utilizan beans como intermediarios para acceder a las páginas JSP. La figura 18.3 muestra gráficamente los pasos fundamentales en la comunicación entre un servlet y una página JSP para proporcionar información al cliente.



Figura 18.3

La descripción de estos pasos es la siguiente:

- La petición del usuario llega al servlet, que la procesa.
- El resultado se almacena en un bean, el objeto Sesión.
- El servlet envía la respuesta a la página JSP.
- La página JSP obtiene los datos del bean y los formatea para su presentación en el navegador.

El uso de beans hace posible que otras páginas JSP puedan acceder a la misma información; no obstante, también es posible utilizar el objeto **Session**, tal como se hace en el ejemplo que se presentará, en aras de que la aplicación sea lo más simple posible. Si el lector se encontrase en el mundo real, debería utilizar beans y almacenar

la información en una base de datos a través de JDBC, ya que las páginas JSP pueden acceder directamente, a través de JDBC, a la mayoría de las bases de datos comerciales, y los componentes JavaBeans son el elemento ideal para aislar el código JDBC que debe ser ejecutado repetidas veces para acceder a la base de datos.

La aplicación controlada por el servlet **Java1803** trata de organizar los *enlaces favoritos* de varios usuarios, de forma que conjuntamente puedan añadir, quitar y actualizar los enlaces a páginas Web. Estos enlaces se guardarán en un fichero y se accederá a él a través de páginas JSP que lo manipularán mediante servlets, los cuales también se encargarán del seguimiento de la sesión de cada uno de los usuarios.

La figura 18.4 muestra el flujo de información de la aplicación y las funciones en que se divide el control de la aplicación, para crear de este modo las clases y elementos encargados de llevarlas a cabo.

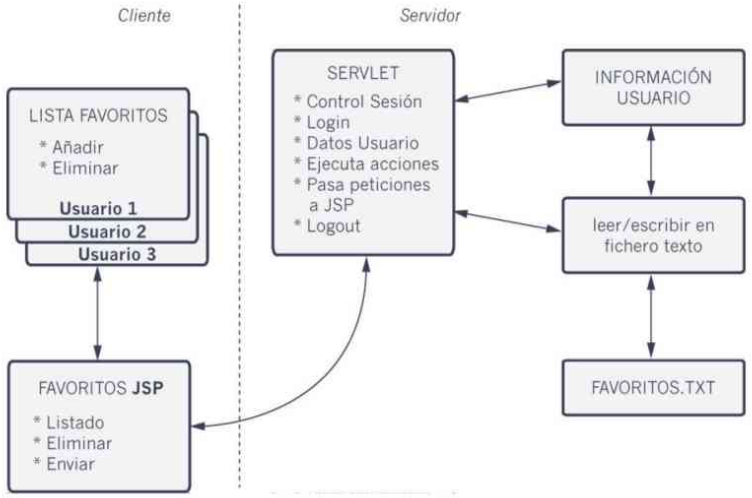


Figura 18.4

A continuación, se describen las clases y las partes más importantes del código que las componen, empezando por el servlet que controla las peticiones y ejecuta las acciones, **FavoritosServlet**, que además se encarga de la identificación de usuarios. Para la ejecución de esta aplicación web se crea una nueva estructura de directorio de aplicación Web, *favoritos*.

El servlet implementado en el archivo fuente `Java1803.java` extiende la clase **HttpServlet**, proporciona acceso a otros paquetes y especifica los directorios. En el método *init()* se invoca al mismo método de la clase padre para registrarlo, en previsión de un uso posterior. Ésta siempre es una técnica recomendable.

También en el método *init()* se inicializa la clase **FavoritosBD**, que es la encargada de mantener el control sobre los usuarios y los enlaces favoritos, leyendo y almacenando la información de cada uno de ellos en el archivo `favoritos.txt`. Aquí es donde el lector, si utiliza JDBC, debería realizar la conexión a la base de datos.

```
public void init( ServletConfig conf ) throws ServletException {
```

```
super.init( conf );
ServletContext sc = conf.getServletContext();
UsuariosDB.abrirBD( sc.getRealPath("favoritos.txt") );
}
```

El método *doPost()* es el que procesa la petición del usuario. Lo primero que hace es asignar la sesión al cliente, para luego comprobar qué acción es la que se ha solicitado a través del método *getParameter()*, que recupera la petición incluida en la llamada al servlet.

```
public void doPost( HttpServletRequest req,
  HttpServletResponse resp ) {
  // Recuperamos la sesión asociada al cliente. Si no existe ninguna
  // se le asigna una nueva, por ello pasamos el parámetro TRUE
  HttpSession sesion = req.getSession( true );
  // Determinamos la acción que se va a realizar
  String accion = req.getParameter( "accion" );
  try {
    // Si la entrada no la entendemos, redirigimos al usuario a la
    // página de registro en la aplicación
    if( accion == null ) {
      resp.sendRedirect( pathHtml+"registrar.html" );
      return;
      }
```

El control de las distintas acciones que permite realizar el servlet se realiza a través de sentencias condicionales simples, en las cuales se comprueba el parámetro de invocación al servlet.

Una de las acciones es la autenticación ante el sistema, que se realiza a través de la indicación del identificador y contraseña de la clase **UsuariosBD**. Si el usuario es validado, el servlet almacenará la información de ese usuario en el objeto **Session** que se le ha asignado, a través de la llamada al método *setAttribute()*, para pasar luego el control a la página JSP que proporciona la interfaz gráfica al usuario para que pueda seguir interaccionando con el sistema.

```
if( accion.equals("login") ) {
  // Si el usuario está entrando, recuperamos los datos que
  // corresponden a su identificación
  FavoritosDB nombre = UsuariosDB.getUsuario(
    req.getParameter("usuario"),
    req.getParameter("password") );
  // Si los datos introducidos son correctos, guardamos la
  // información del usuario en el objeto Sesión utilizando el
  // método setAttribute() y pasamos el control a la página
  // JSP que permitirá la interacción al usuario
  if( nombre != null ) {
    sesion.setAttribute( "usuario",nombre );
    passItOn( req,resp,pathJsp+"Favoritos.jsp" );
    }
  // Si no es correcta la información, mandamos al usuario a la
  // página de registro en la aplicación
else {
  resp.sendRedirect( pathHtml+"registrar.html" );
```

```
    }
}
```

La figura 18.5 muestra la ventana que permite introducir los datos de identificación del usuario ante esta aplicación.



Figura 18.5

Otra de las acciones posibles consiste en el abandono de la aplicación, en cuyo caso se almacena la información en el archivo de usuarios, se hace expirar la sesión y se redirecciona al usuario a la página de salida. Si el lector está utilizando JDBC, aquí es donde deberá cerrar la conexión a la base de datos.

```
else if( accion.equals("salir") ) {
  // Guardamos los datos del usuario, por si acaso
  UsuariosDB.actualizaBD();
  // Concluimos la sesión asignada al usuario
  sesion.invalidate();
  // Presentamos la página de salida
  resp.sendRedirect( pathHtml+"salir.html" );
  }
```

También hay acciones relacionadas con el control de enlaces favoritos y de usuarios. El código siguiente muestra el control de usuarios, que consiste en la actualización del objeto **FavoritosBD** y la información en **UsuariosBD**.

```
else if( accion.equals("registrar") ) {
  // Insertamos los datos del usuario en la base de datos
  UsuariosDB.nuevoUsuario( req.getParameter("usuario"),
    req.getParameter("password") );
  FavoritosDB nombre = UsuariosDB.getUsuario(
    req.getParameter("usuario"),
    req.getParameter("password") );
  // Ahora guardamos esa misma información en el objeto Sesión
  // utilizando el método setAttribute(), pasando el control a
  // la página JSP
  if( nombre != null ) {
    sesion.setAttribute( "usuario",nombre );
    passItOn( req,resp,pathJsp+"favoritos.jsp" );
    UsuariosDB.actualizaBD();
    }
  else {
    resp.sendRedirect( pathHtml+"error.html" );
    }
```

```
    }
```

El resto de acciones corresponden al borrado o incorporación de nuevos enlaces favoritos. Antes de la realización efectiva de estas acciones, se comprueba si el usuario es conocido o no y si la acción se encuentra englobada en una misma sesión. De este modo, si se invoca directamente a esta página, el control se devuelve a la página de autenticación, para que el usuario proceda a su identificación ante la aplicación.

```
else if( accion.equals("nueva") ) {
  // Comprobamos si estamos ante la primera petición, porque si es
  // así reenviamos al usuario al login, indicando que no se puede
  // entrar directamente en esta página
  if( sesion.isNew() ) {
    resp.sendRedirect( pathHtml+"login.html" );
    }
  else {
    // Recuperamos la información del usuario del objeto Sesión, le
    // añadimos el enlace favorito, volvemos a guardar la nueva
    // información en el objeto Sesión y pasamos la petición a la
    // página JSP
    FavoritosDB nombre =
      (FavoritosDB)sesion.getAttribute( "usuario" );
    nombre.nuevoFavorito( req.getParameter("nombre"),
      req.getParameter("favorito") );
    sesion.setAttribute( "usuario",nombre );
    passItOn( req,resp,pathJsp+"favoritos.jsp" );
    }
  }
else if( accion.equals("eliminar") ) {
  // Comprobamos si estamos ante la primera petición, porque si es
  // así reenviamos al usuario al login, indicando que no se puede
  // entrar directamente en esta página
  if( sesion.isNew() ) {
    resp.sendRedirect( pathHtml+"login.html" );
    }
  else {
    // En este caso recuperamos la información del usuario del
    // objeto Sesión, eliminamos el enlace, volvemos a guardar los
    // datos en el objeto Sesión y pasamos el control a la página
    // JSP que controla las acciones del usuario
    FavoritosDB nombre =
      (FavoritosDB)sesion.getAttribute( "usuario" );
    nombre.eliminaFavorito( req.getParameter("favorito") );
    sesion.setAttribute( "usuario",nombre );
    passItOn( req,resp,pathJsp+"favoritos.jsp" );
    }
  }
```

Si el usuario está identificado correctamente, el servlet utiliza la clase **FavoritosBD** para realizar la acción solicitada acerca del enlace favorito. La última acción, en cualquiera de los casos, es el paso del control a la página JSP para presentar la información al usuario. Para ello se utiliza el método *forward()* del objeto **RequestDispatcher**, que devuelve el control al servlet cuando termina.

```
private void passItOn( HttpServletRequest req,
  HttpServletResponse resp, String pagina )
  throws ServletException, IOException {
  RequestDispatcher disp =
    getServletContext().getRequestDispatcher( pagina );
  disp.forward( req,resp );
  }
```

La clase **FavoritosBD**, que pertenece al paquete **db**, es la encargada de manejar la información correspondiente a los enlaces de cada uno de los usuarios. En el constructor de la clase se inicializa esta información analizando la cadena que se pasa como parámetro. Para las acciones de borrado y adición a la tabla se utilizan los métodos *remove()* y *put()* de la tabla *hash* que se emplea como soporte para guardar la información. Esta tabla contendrá elementos de tipos **String**, lo cual se indica mediante la especificación de tipos genéricos. De nuevo, si el lector utiliza JDBC, estos métodos deberían contener la invocación de las acciones correspondientes a la eliminación y adición de registros en la base de datos.

```
public FavoritosDB( String _usuario,String _pass,String _cadena ) {
  this.usuario = _usuario;
  this.password = _pass;
  this.favoritos = new Hashtable<String,String>();
  if( _cadena != null ) {
    this.leeDBString( _cadena );
    }
  }

// Añade un favorito a la cadena que corresponde al usuario que
// se indica
public void nuevoFavorito( String _usuario,String _favorito ) {
  favoritos.put( _usuario,_favorito );
  }

// Elimina un favorito de la cadena que corresponde al usuario
// que se indica
public void eliminaFavorito( String _usuario ) {
  favoritos.remove( _usuario );
  }
```

Los métodos *getBDString()* y *leeBDString()* son los encargados de convertir la información del usuario al formato utilizado por el archivo de almacenamiento, y de recuperar esa información y reconstruirla.

```
public String getDBString() {
  String cadena = "";
  boolean primero = true;
  try {
    cadena += URLEncoder.encode( this.usuario,"UTF-8" ) + " "+
      URLEncoder.encode( this.password,"UTF-8" ) + " ";
    for( Enumeration enu = favoritos.keys();
      enu.hasMoreElements(); ) {
      String favId = (String)enu.nextElement();
      String favUrl = (String)favoritos.get(favId);
      if( primero ) {
        cadena += URLEncoder.encode( favId,"UTF-8" ) + "!"+
```

```
            URLEncoder.encode( favUrl,"UTF-8" );
          primero = false;
          }
        else {
          cadena += "|"+URLEncoder.encode( favId,"UTF-8" )+ "!"+
            URLEncoder.encode( favUrl,"UTF-8" );
          }
        }
    } catch( UnsupportedEncodingException  e ) {
      e.printStackTrace();
      }
    return( cadena );
    }

  public void leeDBString( String cadena ) {
    StringTokenizer token = new StringTokenizer( cadena );
    try {
      this.usuario = URLDecoder.decode( token.nextToken(),"UTF-8" );
      this.password = URLDecoder.decode( token.nextToken(),"UTF-8" );
      StringTokenizer ftoken =
        new StringTokenizer( token.nextToken(),"|" );
      while( ftoken.hasMoreTokens() ) {
        StringTokenizer htoken =
          new StringTokenizer( ftoken.nextToken(),"!" );
        favoritos.put( URLDecoder.decode(htoken.nextToken(),"UTF-8"),
          URLDecoder.decode(htoken.nextToken(),"UTF-8") );
        }
    } catch( UnsupportedEncodingException e ) {
      e.printStackTrace();
      }
    }
```

La base de datos del ejemplo está constituida por el fichero `favoritos.txt` y la clase **UsuariosBD**, que pertenece al paquete **db** y se encarga de abrir el archivo, almacenar datos en él y recuperarlos. El formato que se utiliza en el archivo es el siguiente:

```
usuario contraseña id1!url1|id2!url2|. . . idn!urln
```

También se puede indicar un archivo distinto para almacenar la información, en cuyo caso se debe pasar como argumento al método *abrirBD()*, que está sobrecargado para que admita esta posibilidad. Los datos de los usuarios se recuperan en un **Vector**, al cual hay que indicar el tipo de elementos que contendrá mediante el uso de tipos genéricos, en este caso de tipo **FavoritosDB**.

```
  public static void abrirBD() {
    usuarios = new Vector<FavoritosDB>();
    leerFavFichero( fichero );
    }

  public static void abrirBD( String _fichero ) {
    usuarios = new Vector<FavoritosDB>();
    fichero = _fichero;
    leerFavFichero( _fichero );
    }
```

La información de cada usuario se almacena en una línea independiente. Los métodos más interesantes son: el encargado de leer el archivo para generar un vector con la información del usuario; el que se encarga de escribir de nuevo la información del usuario, que se encuentra sincronizado con el anterior para que no se produzcan operaciones de lectura y escritura simultáneas; el método que se encarga de recuperar los datos de identificación de un usuario y, por fin, el que se encarga de añadirlo al fichero si no existe.

```
public static synchronized void leerFavFichero( String _fichero ) {
  try {
    BufferedReader buff =
      new BufferedReader( new FileReader(_fichero) );
    String linea;
    while( (linea=buff.readLine()) != null ) {
      usuarios.add( new FavoritosDB(null,null,linea.trim()) );
      }
    buff.close();
  } catch( Exception e ){
    e.printStackTrace();
    }
  }

public static synchronized void escribirFavFichero(
  String _fichero ) {
  try {
    PrintWriter salida = new PrintWriter( new BufferedWriter(
      new FileWriter(_fichero) ) );
    for( Enumeration enum = usuarios.elements();
      enum.hasMoreElements() ; ) {
      salida.println( ( (FavoritosDB)enum.nextElement()
        ).getDBString() );
      }
    salida.close();
  } catch( Exception e ) {
    e.printStackTrace();
    }
  }

public static FavoritosDB getUsuario( String _usuario,
  String _pass ) {
  FavoritosDB nombre = null;
  for( Enumeration enum = usuarios.elements();
    enum.hasMoreElements() ; ) {
    nombre = (FavoritosDB)enum.nextElement();
    if( nombre.usuario.equals(_usuario) &&
      nombre.password.equals(_pass) ) {
      break;
      }
    else {
      nombre = null;
      }
    }
  return( nombre );
  }

public static void nuevoUsuario( String _usuario,String _pass ) {
```

```
    usuarios.add( new FavoritosDB(_usuario,_pass,null) );
    }
```

La interacción del usuario con la aplicación se realiza a través de la página `favoritos.jsp` que presenta la información de la sesión y permite que el usuario realice acciones de incorporación y borrado de nuevos enlaces. No es objetivo de este Tutorial el desarrollo de páginas JSP, para lo cual puede el lector consultar la bibliografía en donde se encontrarán recursos excelentes que tratan el tema; por ello, solamente se presenta al lector el código correspondiente a la página, con los comentarios adecuados acerca de las acciones que corresponden a cada parte de esa página.

```
<html>
<head>
<title>Tutorial de Java, Servlets</title>
</head>
<%--
  Importamos las clases necesarias e indicamos al servidor que
  debe mantener la sesión.
--%>
<%@ page import="java.util.*" %>
<%@ page import="java.net.*" %>
<%@ page import="db.FavoritosDB" %>
<%@ page session="true" %>
<%! private FavoritosDB nombre; %>
<%! private static final String servletUrl =
   "http://localhost:8080/favoritos/java1803"; %>
<%
// Recuperamos el identificador del usuario en esta sesión
nombre = (FavoritosDB)session.getValue("usuario");
%>
<body>
<%-- Se utiliza una expresión JSP para saludar al usuario --%>
<h1 align="center">Bienvenido, <%= nombre.usuario %></h1>
<table width="100%">
  <tr>
    <td>
    <%-- Codificamos la acción de salir de la aplicación --%>
    <b><a href="<%=response.encodeURL(servletUrl+"?accion=salir")%>">
      SALIR</a></b></td>
    <td><b>Favorito</b></td>
    <td><b>URL Favorito</b></td>
  </tr>
  <tr><td colspan="3"><hr></td>
  </tr>
<%
// Imprimimos cada uno de los enlaces en una línea separada
for( Enumeration enu=nombre.favoritos.keys();
  enu.hasMoreElements(); ) {
  String favNombre = (String)enu.nextElement();
  String favUrl = (String)nombre.favoritos.get(favNombre);
  // Codificamos la URL para el seguimiento de la sesión mediante
  // el método response.encodeURL()
  // La URL codifica la información pasada al servidor mediante
  // el método URLEncoder.encode()
  // Esto es necesario hacerlo, porque el parámetro que se pase puede
```

```
  // tener espacios o caracteres que no puedan ser entendibles por
  // el servidor como una URL
  String eliminarUrl = response.encodeURL(servletUrl +
    "?accion=eliminar&favorito=" + URLEncoder.encode(favNombre));
%>
<tr>
  <td><a href="<%= eliminarUrl %>">Eliminar</a></td>
  <td><a href="<%= favUrl %>" target="_blank"><%= favNombre %>
  </a></td>
  <td><a href="<%= favUrl %>" target="_blank"><%= favUrl %></a></td>
</tr>
<%
}
%>
</table>
<hr>
<%-- Opción de añadir un nuevo enlace favorito --%>
<form action="<%= response.encodeURL(servletUrl) %>" method="post">
<input type="hidden" name="accion" value="nueva">
Nuevo Enlace<input type="text" size="30" name="nombre"><br>
  URL Enlace <input type="text" size="30" name="favorito">
<input type="submit" name="nuevoFavorito" value=" Añadir ">
</form>
</body>
</html>
```

Correo electrónico

El ejemplo `Java1806.java` consiste en un servlet capaz de enviar un mensaje de correo electrónico desde una página Web.

El ejemplo está compuesto por unos cuantos métodos, de los cuales unos están encargados de servir a la página, como son el método *service()*, que es invocado tanto en peticiones `GET` como `POST`; recupera los parámetros de la petición HTTP, envía el correo y luego la página de respuesta al usuario. El método *obtieneParametro()* se utiliza para recuperar el valor de los parámetros enviados en el objeto **HttpRequest**. Y el método *enviaRespuesta()* es el que envía la página HTML de respuesta al usuario.

Para el envío del correo se utilizan otros métodos. El método *enviaCorreo()* es utilizado para enviar el correo electrónico a través del protocolo SMTP. El método *enviaComando()* se emplea para enviar comandos al servidor de correo; escribe una cadena en el canal de salida y guarda el comando para poder seguir la pista a lo que hace. El método *leeRespuesta()* se utiliza para leer respuestas del servidor de correo; lee una cadena del canal de entrada y también guarda esa respuesta en la lista de análisis.

En este caso no se han utilizado variables miembro y se pasan todas desde el método *service()* a los demás métodos, porque el servidor Web creará una instancia del servlet **Java1806**, y después este objeto deberá realizar múltiples servicios, probablemente simultáneos, para atender a las peticiones de las páginas.

Para probar el servlet, las cosas no son tan sencillas como en ejemplos anteriores, ya que depende de si se tiene acceso o no a un servidor Web, en cuyo caso se facilita el asunto. Para realizar las pruebas con **Jakarta-Tomcat**, el camino a seguir será el que se describe a continuación.

Lo primero es crear la estructura de directorio para la aplicación Web *correo* y editar el fichero `correo.html` con cualquier editor y cambiar "*localhost*" por la dirección IP de la máquina en que se esté ejecutando el servlet, si fuese necesario, que coincidirá probablemente con la misma que se use para editar el fichero. Luego renombrarlo a *index.html*, para que el navegador lo reconozca como la página por defecto a presentar cuando se invoque la aplicación Web.

Lo siguiente es crear el descriptor de la aplicación *web.xml*, en donde se establece la configuración del servlet y arrancar el servidor **Tomcat** solicitando desde un navegador la página:

```
http://localhost:8080/correo
```



Figura 18.6

Se lanza el navegador, en este caso *Firefox*, que además de ser *Open Source*, dispone de versiones para casi cualquier plataforma, y se carga la página modificada anteriormente, que se visualizará tal como muestra la figura 18.6.

A continuación, se cambian los datos que aparecen por defecto en el formulario, para introducir los datos correspondientes a la conexión de que disponga el lector, y a quien va a enviar su mensaje de prueba. Y para finalizar la prueba pulsar el botón *Enviar*. Si todo ha ido bien, y el correo electrónico ha funcionado correctamente, debería aparecer en el navegador una página con el historial de la sesión, semejante a la que reproduce la figura 18.7.

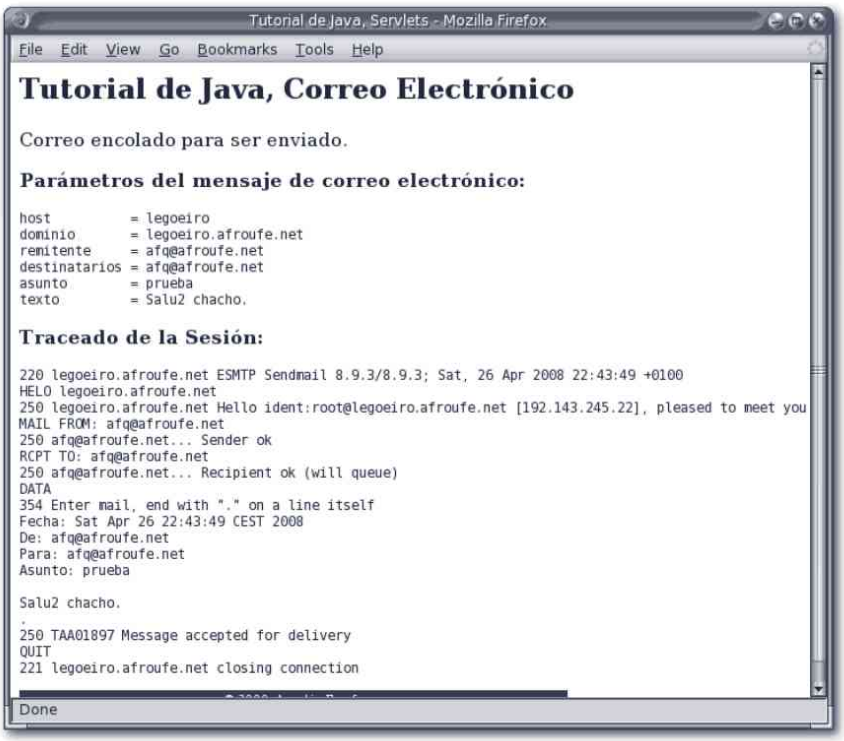


Figura 18.7

Autenticación de usuarios

Otra de las aplicaciones más habituales de los servlets es la autenticación de usuarios para el acceso a páginas restringidas de un sitio Web. El ejemplo `Java1807.java` presenta un modelo básico de autenticación que, aunque codifica en su interior los usuarios con acceso permitido, no es complicado modificar para su acceso a una base de datos en donde se vayan almacenando los usuarios y claves con acceso permitido a las páginas restringidas.



Figura 18.8

En este caso, la autorización de usuarios se realiza a través del mecanismo básico de HTTP, que depende del navegador que se esté utilizando. Es el propio navegador el que presenta la ventana de entrada del identificador y contraseña del usuario. La figura 18.8 muestra la ventana de entrada presentada por *Firefox*.

El método *controlAcceso()* es el encargado de la comprobación de la identificación del usuario y de permitir o no el acceso a zonas restringidas, comparando la información proporcionada por el navegador en la cabecera *Authorization* del protocolo WWW contra la lista que el ejemplo mantiene en la tabla **usuarios**.

En esta ocasión ya no es necesario indicar al lector los pasos para ejecutar el servlet. La opción más simple es incorporarlo a la aplicación *tutorial*, incluyendo la declaración del servlet en el fichero *web.xml* y copiando el archivo *.class* en el directorio *WEB-INF/classes*.

Filtros

Los filtros forman parte de la especificación Servlet y se utilizan cuando una aplicación Web requiere una respuesta estándar a cualquier petición HTTP. Los filtros pueden utilizarse para interceptar cualquier petición que se haga al servidor Web y para el control *pre* y *post* de la respuesta del servidor.

Un filtro proporciona servicios a cualquier recurso de la aplicación. Es un trozo de código que intercepta una petición solicitada por un recurso determinado al servidor Web. Cuando un filtro intercepta una petición, tienes tres opciones: procesar la petición y devolver una respuesta a quien ha realizado la petición; pasar la petición al servidor sin cambio alguno; o procesar la petición antes de pasarla al servidor. Los filtros, además, pueden estar encadenados, de modo que en los dos últimos casos, la petición es enviada al siguiente filtro de la cadena o al recurso solicitado, si se trata del último filtro de la cadena.

La figura 18.9 muestra gráficamente la petición del cliente realizada al servlet que reside en el servidor, y cómo pasa en ambos sentidos a través de filtros.

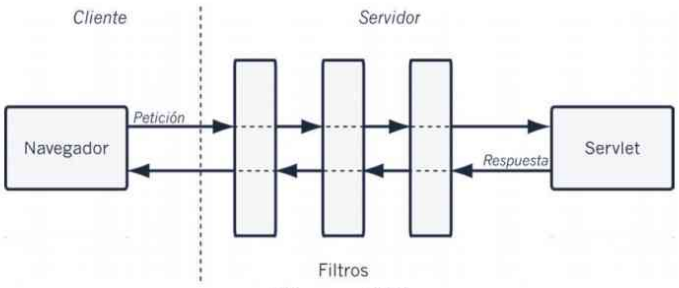


Figura 18.9

Los filtros tienen muchos usos y pueden ser de varios tipos: filtros de autenticación, de auditoría, de conversión de imágenes, de compresión de datos, de encriptación de información, de transformación de contenido XML, y muchos más.

Todos los filtros implementan la interfaz **Filter**, que define el ciclo de vida del filtro a través de los métodos *init()*, *doFilter()* y *destroy()*.

El método *init()* se invoca una única vez cuando el filtro está completamente cargado. Se le pasa el objeto **FilterConfig**, que proporciona al filtro acceso a cualquier parámetro de configuración definido en el descriptor de la aplicación, *web.xml*. También proporciona acceso al **ServletContext** en el cual el filtro está configurado.

El método más importante del filtro es *doFilter()*, en donde se realiza la mayor parte de las acciones encomendadas al filtro. Este método moldea los parámetros `request` y `response` a sus tipos correctos, realiza las acciones encomendadas e invoca al método *doFilter()* del objeto **FilterChain**, que es el que contiene la lista de filtros que deben ser procesados. Esta última llamada es opcional y el filtro puede ejecutar cualquier otro recurso utilizando el objeto **RequestDispatcher**.

El código siguiente muestra el contenido del método *doFilter()* de una forma completamente genérica:

```
public void doFilter( ServletRequest request,
  ServletResponse response,FilterChain chain )
  throws IOException,ServletException {
  // Aseguramos los parámetros de entrada a sus tipos correctos
  HttpServletRequest req = (HttpServletRequest)request;
  HttpServletResponse resp = (HttpServletResponse)response;
  // Realizamos las acciones PRE filtro
  // Opcionalmente invocamos a otros filtros encadenados
  chain.doFilter( request,resp );
  // Realizamos las acciones POST filtro
}
```

El ejemplo `Java1808.java` es muy simple y solamente trata de mostrar al lector la forma en que varios filtros pueden aplicarse al mismo patrón, tal como mostraba la figura 18.9. El orden de ejecución de los filtros vendrá dado por su orden de colocación en el descriptor de la aplicación, *web.xml*. En el caso de este ejemplo, los filtros solamente presentan mensajes en la consola.

Para poder ejecutar un filtro, es necesario asociarlo a un recurso, de forma que cuando ese recurso sea solicitado al servidor, el motor de servlets ejecutará el filtro como parte de la generación de la respuesta. La asociación de filtros a recursos se realiza también en el archivo descriptor de la aplicación, *web.xml*. En el ejemplo, en el que se ejecutan dos filtros, la definición de los filtros el archivo descriptor sería de la siguiente forma:

```
<!-- Se definen los filtros dentro de la aplicación Web -->
<filter>
  <filter-name>Primer Filtro</filter-name>
  <filter-class>Filtro1</filter-class>
</filter>
<filter>
  <filter-name>Segundo Filtro</filter-name>
  <filter-class>Filtro2</filter-class>
</filter>
<!-- Se define la URL que interceptarán los filtros -->
<filter-mapping>
```

```
    <filter-name>Primer Filtro</filter-name>
    <url-pattern>/java1808</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>Segundo Filtro</filter-name>
    <url-pattern>/java1808</url-pattern>
  </filter-mapping>
```

Para invocar el ejemplo, una vez introducida su definición en el archivo general de **Tomcat**, por ejemplo, en la aplicación Web *tutorial*, bastará con solicitar en un navegador la siguiente URL:

```
http://localhost:8080/tutorial/java1808
```

INFORMACIÓN ADICIONAL

En los siguientes párrafos se presenta información adicional y se muestra el uso de algunas características y técnicas que no se han explicado en los programas anteriores. Solamente se muestran ejemplos de código, para ilustrar otros aspectos de los servlets y permitir al lector la investigación propia de dichas características.

Concurrencia

Si el método *service()* de la clase **Servlet**, o cualquiera de los métodos *doXxx()* de la subclase **HttpServlet**, no puede ser implementado de forma que sea seguro ante la concurrencia de varias tareas, el servlet puede declararse implementando la interfaz **javax.servlet.SingleThreadModel** que garantiza que el método *service()* nunca va a ser llamado de forma concurrente. Esta interfaz no contiene ningún método, solamente actúa como filtro para indicar que el servlet no es seguro ante el funcionamiento en multitarea.

```
public class ServletNoConcurrente extends HttpServlet
  implements SingleThreadModel {
  protected void doGet(HttpServletRequest req,
    HttpServletResponse res ) throws ServletException,IOException {
    // Este método es llamado desde el método service() de
    // la clase HttpServlet. Y declarada esta clase como
    // SingleThreadModel, se asegura que nunca será llamado
    // de forma concurrente
    }
  }
```

Si un servidor lanza peticiones concurrentes a un servlet, por ejemplo al grabar datos, o al crear múltiples instancias del servlet para formar un *Pool*, la implementación de la interfaz **SingleThreadModel** asegura que no van a ejecutarse de forma concurrente. Esto también se puede conseguir utilizando la palabra reservada `synchronized`, que permite conseguir el mismo resultado.

```
public class ServletNoConcurrente extends HttpServlet {
  protected synchronized void doGet(HttpServletRequest req,
```

```
    HttpServletResponse res ) throws ServletException,IOException {
    // Este método es llamado desde el método service() de
    // la clase HttpServlet. Y declarada esta clase como
    // SingleThreadModel, se asegura que nunca será llamado
    // de forma concurrente
    }
  }
```

Cookies en el cliente

En un ejemplo anterior ya se han utilizado *Cookies* indirectamente para manejar *sesiones*, que son datos almacenados en el servidor, y al cliente solamente se transmite un índice a esos datos, el identificador de la sesión. Los Cookies también se pueden utilizar para almacenar datos directamente sobre el cliente. El API Servlet proporciona la clase **javax.servlet.http.Cookie** para poder representar de forma orientada a objetos los Cookies, de modo que no sea necesario que el programador componga y descomponga las cabeceras `Cookie` y `Set-Cookie` del protocolo HTTP por sí mismo.

Incluso aunque no se quieran guardar datos en el cliente, hay veces en que es más ventajoso manejar los Cookies a través de esta clase **Cookie**, para tener más control sobre parámetros como `Domain` o `Path`, por ejemplo, para compartir Cookies entre distintos servlets, o incluso entre distintos servidores.

Un ejemplo de uso podría ser un servlet de autenticación que reciba un identificador de usuario y una contraseña a través de una entrada de acceso vía `POST` y verifique esa información contra una base de datos central. En este caso puede controlar perfectamente la cadena de autenticación; por ejemplo, la combinación `"user:password"` codificada en *Base-64*, tal como se usa en la autenticación básica del protocolo HTTP. Esta cadena se puede colocar ahora en un Cookie y ser enviada de regreso al cliente.

```
Cookie cokClave = new Cookie( "xyz-Auth",credenciales );
cokClave.setVersion( 1 );
cokClave.setDomain( ".xyz.com" );
res.addCookie( cokClave );
```

El dominio del Cookie se fija a "*.xyz.com*", de forma que será enviado a todos los ordenadores que se encuentren en el dominio "*xyz.com*", como pueden ser: *a.xyz.com*, *perico.xyz.com*, *timba.xyz.com* (pero no *b.c.xyz.com*). Tenga el lector en cuenta que el atributo *domain* está soportado por la *versión 1* de Cookies, que siguen la norma *RFC2109*, pero no por los Cookies de estilo Netscape, *versión 0*, que es el modelo por defecto para la creación de nuevos objetos de tipo **Cookie**.

```
boolean verificado = false;
Cookie[] cookies = req.getCookies();
for( int i=0; i < cookies.length; i++ ) {
  String n = cookies[i].getName();
  String d = cookies[i].getDomain();
  if( n != null && n.equals("xyz-Auth") &&
     d != null && d.equals(".xyz.com") ) {
```

```
    String credenciales = cookies[i].getValue();
    verificado = verifyCredentials( credenciales );
    break;
    }
  }
if( !verificado ) {
  res.sendRedirect(...);
  return;
  }
```

Los credenciales son recuperados del Cookie y verificados por la base de datos de autenticación. Si esos credenciales no son válidos o no figuran, el cliente es redirigido a la página de entrada del servidor de autenticación; y si todo es correcto, se envía al cliente la información restringida.

Comunicación entre servlets

Los servlets no están solos en un servidor Web, sino que tienen acceso a todos los servlets que estén dentro de su contexto, normalmente el directorio "*servlet*", representado por un objeto de tipo **ServletContext**. Todos los servlets que pertenecen a un mismo contexto son los que figuran en el fichero de propiedades `servlet.properties`. El objeto **ServletContext** siempre está disponible, y se puede acceder a él a través del método *getServletContext()* de la clase **ServletConfig**.

Un servlet puede obtener la lista de todos los servlets que se encuentran en su mismo contexto llamando al método *getServletNames()* del objeto **ServletContext**. El método *getServlet()* devuelve el servlet correspondiente a un nombre conocido, obtenido mediante *getServletNames()* por ejemplo; aunque hay que tener cuidado con este método porque puede lanzar una excepción de tipo **ServletException**, ya que es posible que necesite cargar e inicializar el servlet requerido si no estaba funcionando.

Una vez obtenida la referencia a otro servlet, los métodos de ese servlet quedan accesibles, teniendo en cuenta que todos los métodos que no estén declarados en la clase **Servlet**, sino en subclases, deben ser moldeados para que devuelvan objetos del tipo requerido.

El lector debe tener también en cuenta que en Java una clase no está definida solamente por su nombre de clase, sino que en su identificación se incluye el cargador de clases, **ClassLoader**, que la ha cargado. Los servidores Web, normalmente, cargan cada servlet con un cargador de clases distinto, para poder recargar los servlets en tiempo de ejecución, porque las clases simples no se pueden reemplazar en una Máquina Virtual Java que esté corriendo; solamente un objeto **ClassLoader** con todas las clases puede ser reemplazado.

Es decir, que las clases que son cargadas por el cargador de clases de un servlet no se pueden utilizar para la comunicación entre servlets. El nombre de una clase llamada **AbServlet**, utilizado en un moldeo de la forma:

```
AbServlet clase = (AbServlet)contexto.getServlet( "AbServlet" )
```

por la clase **CdServlet**, es diferente del nombre ***AbServlet*** de la clase que se utiliza en la propia clase **AbServlet**.

Una forma simple de evitar este problema es el uso de superclases o, de forma más elegante, utilizar una interfaz para el cargador del sistema y así compartir todos los servlets. En un servidor Web escrito en Java todas estas clases se localizarían normalmente en el camino que define la variable de entorno CLASSPATH.

Por ejemplo, el servlet **AbServlet** quiere llamar al método público *chorra()* del servlet **CdServlet**. Entonces se define una interfaz **CdInterface** que define el método que se quiere llamar, que es el que va a ser cargado por el cargador del sistema. **CdServlet** implementa la interfaz, los servlets se colocan en el mismo directorio y la interfaz se coloca en uno de los directorios a los que apunte la variable de entorno CLASSPATH.

```
public class AbServlet extends HttpServlet {
  protected void doGet( HttpServletRequest req,
    HttpServletResponse res ) throws ServletException,IOException {
    ...
    ServletContext context = getServletConfig().getServletContext();
    CdInterface cd = (CdInterface)context.getServlet( "CdServlet" );
    cd.chorra();
    ...
    }
  ...
  }

public interface CdInterface {
   public void chorra();
   }

public class CdServlet extends HttpServlet implements CdInterface {
  public void chorra() {
    System.err.println( ""chorra() llamado"");
    }
  ...
  }
```

CAPÍTULO 19

JDBC

JDBC es más que un acrónimo de *Java DataBase Connectivity*, es un API de Java que permite al programador ejecutar instrucciones en el lenguaje estándar de acceso a Bases de Datos, **SQL** (*Structured Query Language*, lenguaje estructurado de consultas), un lenguaje de muy alto nivel para crear, examinar, manipular y gestionar Bases de Datos relacionales. Para que una aplicación pueda hacer operaciones en una Base de Datos, ha de tener una conexión con ella, que se establece a través de un *driver*, que convierte el lenguaje de alto nivel a sentencias de Base de Datos. Es decir, las tres acciones principales que realizará JDBC son las de establecer la conexión a una base de datos, ya sea remota o no; enviar sentencias SQL a esa base de datos y, en tercer lugar, procesar los resultados obtenidos de la base de datos.

BASES DE DATOS

Una base de datos es una serie de tablas que contienen información ordenada en alguna estructura que facilita el acceso a esas tablas, así como ordenarlas y seleccionar filas de las tablas según criterios específicos. Las bases de datos generalmente tienen *índices* asociados a alguna de sus columnas, de forma que el acceso a las filas sea lo más rápido posible.

Las Bases de Datos son, sin lugar a dudas, las estructuras más utilizadas en ordenadores, ya que son el corazón de sistemas tan complejos como el censo de una nación, la nómina de empleados de una empresa, el sistema de facturación de una multinacional, o el medio por el que una agencia de viajes expide el billete para las próximas vacaciones.

En el caso, por ejemplo, del registro de trabajadores de una empresa, el lector puede imaginar una tabla con los nombres de los empleados y direcciones, sueldos,

retenciones y beneficios. Para organizar esta información, se puede empezar con una tabla que contenga los nombres de los empleados, su dirección y su número de teléfono. También se podría incluir la información relativa a su sueldo, categoría, última subida de salario, etc.

¿Podría todo esto colocarse en una sola tabla? Casi seguro que no. Los rangos de salario para diferentes empleados probablemente sean iguales, por lo que se podría optimizar la tabla almacenando solamente el tipo de salario en la tabla de *empleados* y los rangos de salario (en euros) en otra tabla, creando un índice (indexada) a través del tipo de salario. Por ejemplo:

Key	Nombre	Tipo de Salario
1	Pérez	2
2	García	1
3	Cabrera	2
4	López	3
5	Gómez	1

Tipo de Salario	Mínimo	Máximo
1	1100	1200
2	1200	1500
3	1500	1800

Los datos de la columna *Tipo de Salario* están referidos a la segunda tabla. Se pueden imaginar muchas categorías para estas tablas secundarias, como por ejemplo la provincia de residencia y los tipos de retención de Hacienda, o si tiene seguro de vida, vivienda propia, coche, apartamento en la playa, etc. Cada tabla tiene una primera columna que sirve de clave para las demás columnas, que son las que contienen datos relevantes. La construcción de tablas en las bases de datos es tanto un arte como una ciencia, y su estructura está referida por su *forma normal*. Las tablas se dice que están en primera, segunda o tercera forma normal, o de modo abreviado como 1NF, 2NF o 3NF.

1. Cada celda de la tabla debe tener solamente un valor (nunca un conjunto de valores). (**1NF**)
2. 1NF y cada columna que no es clave, depende completamente de la columna clave. Esto significa que hay una relación uno a uno entre la clave primaria y las restantes celdas de la fila. (**2NF**)
3. 2NF y todas las columnas que no son clave son mutuamente independientes. Esto significa que no hay columnas de datos que contengan datos calculados a partir de los datos de otras columnas. (**3NF**)

Actualmente todas las bases de datos se construyen de forma que todas sus tablas están en la *Tercera Forma Normal* (3NF); es decir, que las bases de datos están constituidas por un número bastante alto de tablas, cada una de ellas con relativamente pocas columnas de información.

A la hora de extraer datos de las tablas, se realizan consultas contra ella. Por ejemplo, si se quiere generar una tabla de empleados y sus rangos de salario para

algún tipo de plan especial de la empresa, esa tabla no existe directamente en la base de datos, así que debe construirse haciendo una consulta a la base de datos, y se obtendría una tabla que contendría la siguiente información:

Nombre	Mínimo	Máximo
Pérez	1200	1500
García	1100	1200
Cabrera	1200	1500
López	1500	1800
Gómez	1100	1200

O quizá ordenada por el incremento de salario:

Nombre	Mínimo	Máximo
García	1100	1200
Gómez	1100	1200
Pérez	1200	1500
Cabrera	1200	1500
López	1500	1800

Para generar la tabla anterior se habría de realizar una consulta a la base de datos que tendría la siguiente forma:

```
SELECT DISTINCTROW Empleados.Nombre, TipoDeSalario.Minimo,
TipoDeSalario.Maximo FROM Empleados INNER JOIN TipoDeSalario ON
Empleados.SalarioKey = TipoDeSalario.SalarioKey
ORDER BY TipoDeSalario.Minimo;
```

El lenguaje en que se ha escrito la consulta es SQL, actualmente soportado por casi todas las bases de datos. Los estándares de SQL han sido varios a lo largo de los años y muchas de las bases de datos soportan alguno de esos tipos. El estándar **SQL'92** es el que se considera origen de todas las actualizaciones. Hay que tener en cuenta que hay posteriores versiones de SQL, perfeccionadas y extendidas para explotar características únicas de bases de datos particulares; así que no es conveniente separarse del estándar básico si se pretende hacer una aplicación que pueda atacar a cualquier tipo de base de datos.

Desde que los PC se han convertido en una herramienta presente en la mayor parte de las oficinas, se han desarrollado un gran número de bases de datos para ejecutarse en ese tipo de plataformas; desde bases de datos muy elementales como *Microsoft Works*, hasta otras bastante sofisticadas como *dBase*, *Paradox*, *Access*. O incluso *JavaDB*, que *Sun Microsystems* proporciona junto con el JDK.

Otra categoría más seria de bases de datos para PC son aquellas que usan la plataforma PC como cliente para acceder a un servidor. Estas bases de datos son *IBM DB/2*, *Microsoft SQL Server*, *Oracle*, *Sybase*, *SQLBase*, *Informix*, *MySQL* o

PostgreSQL. Todas estas bases de datos soportan varios dialectos similares de SQL, y todas parecen, a primera vista, intercambiables. La razón de que no sean intercambiables, evidentemente, es que cada una está diseñada con unas características de rendimiento distintas, con una interfaz de usuario y programación diferente. Aunque todas ellas soportan SQL y la programación es similar, cada base de datos tiene su propia forma de recibir las consultas SQL y su propio modo de devolver los resultados. Aquí es donde aparece el siguiente nivel de estandarización, de la mano de **ODBC** (*Open DataBase Connectivity*).

La idea es conseguir que se pueda escribir código independientemente de quién sea el propietario de la base de datos a la que se quiera acceder, de forma que se puedan extraer resultados similares de diferentes tipos de bases de datos sin necesidad de tocar el código del programa. Si se consiguiese escribir alguna forma de trazadores para estas bases de datos que tuviesen una interfaz similar, el objetivo no sería difícil de alcanzar.

Microsoft hizo su primer intento en 1992, con la especificación que llamaron *Object Database Connectivity*; y que se suponía iba a ser la respuesta a la conexión a cualquier tipo de base de datos desde *Windows*. Como en toda aplicación informática, ésta no fue la única versión, sino que hubo varias hasta llegar a la de 1994, que era más rápida y más estable. Además de ser la primera de las versiones de 32 bits, comenzó a desplazarse a otras plataformas diferentes de *Windows*, acaparando además de los PC también el mundo de las estaciones de trabajo. Tanto es así, que ahora casi todos los fabricantes de bases de datos proporcionan un *driver* ODBC para acceder a su base de datos.

Sin embargo, ODBC dista mucho de ser la panacea que en un principio se podía pensar. Muchos fabricantes de bases de datos soportan ODBC como una *interfaz alternativa* a la suya estándar, y la programación en ODBC no es nada, pero que nada trivial; incluyendo toda la parafernalia de la programación para Windows con *handles*, punteros y opciones que son difíciles de asimilar. Finalmente, ODBC no es un estándar libre, sino que ha sido desarrollado y es propiedad de *Microsoft*, lo cual, dados los vientos que soplan en este competitivo mundo de las compañías de software, hace su futuro difícil de predecir.

CONECTIVIDAD JDBC

JDBC está diseñado teniendo en mente la comunicación con bases de datos. JDBC especifica una serie de clases y métodos para permitir a cualquier programa Java una forma homogénea de acceso a sistemas de bases de datos. Este acceso se realiza a través de *drivers*, que son los que implementan la funcionalidad especificada en JDBC. Esto es semejante a lo que ofrece ODBC.

La necesidad de JDBC, a pesar de la existencia de ODBC, viene dada porque ODBC es una interfaz escrita en lenguaje C, que al no ser un lenguaje portable, haría

que las aplicaciones Java también perdiesen la portabilidad. Y además, ODBC tiene el inconveniente de que se ha de instalar manualmente en cada máquina; al contrario que los drivers JDBC, que al estar escritos en Java son automáticamente instalables, portables y seguros.

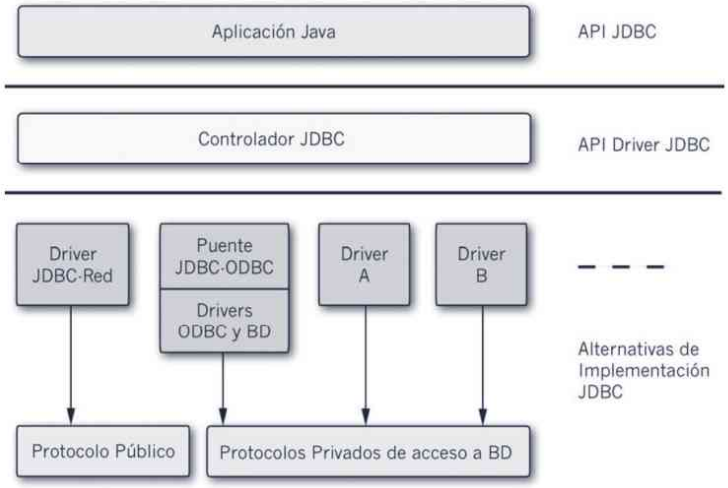


Figura 19.1

La conectividad de bases de datos se basa en sentencias SQL, que a través de JDBC, permiten realizar la conexión a la base de datos, realizar consultas y recibir los resultados. Los programadores que utilizan JDBC suelen crear sus propias sentencias SQL, lo que les obliga a realizar más trabajo que si utilizasen herramientas visuales para generar las consultas, además de necesitar poseer un buen conocimiento de SQL.

Sin embargo, JDBC permite ciertas facilidades, como realizar la actualización de múltiples registros con un solo comando o acceder a múltiples servidores de bases de datos dentro de una transacción simple. Además, permite reutilizar las conexiones a la base de datos, proceso que se conoce como "*connection pooling*", de modo que no se necesita realizar una nueva conexión a la base de datos para cada comando JDBC.

JDBC también permite escribir aplicaciones que accedan a datos a través de sistemas de bases de datos incompatibles ejecutándose en plataformas distintas, partiendo de la base de que Java se puede ejecutar sobre plataformas hardware y sistemas operativos diferentes.

Acceso de JDBC a Bases de Datos

El API JDBC soporta dos modelos diferentes de acceso a Bases de Datos, los modelos de dos y tres capas.

MODELO DE DOS CAPAS

Este modelo se basa en que la conexión establecida entre la aplicación Java o el applet que se ejecuta en el navegador, se realiza directamente con la base de datos, véase la figura 19.2.

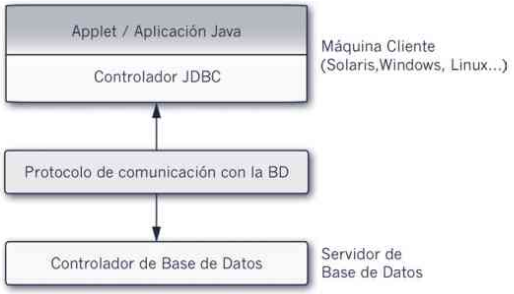


Figura 19.2

Esto significa que el driver JDBC específico para conectarse con la base de datos, debe residir en el sistema local. La base de datos puede estar en cualquier otra máquina y se accede a ella mediante la red. Ésta es la configuración de típica *Cliente/Servidor*: el programa cliente envía instrucciones SQL a la base de datos, ésta las procesa y envía los resultados de vuelta a la aplicación.

MODELO DE TRES CAPAS

En este modelo de acceso a las bases de datos, las instrucciones son enviadas a una capa intermedia entre Cliente y Servidor, que es la que se encarga de enviar las sentencias SQL a la base de datos y recuperar el resultado desde la base de datos, figura 19.3. En este caso el usuario no tiene contacto directo, ni siquiera a través de la red, con la máquina donde reside la base de datos.

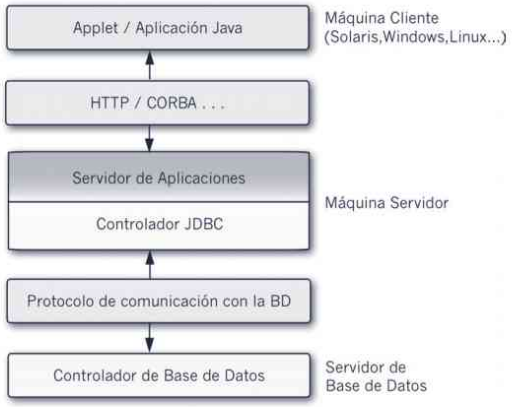


Figura 19.3

Este modelo presenta la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan contra la base de datos, y además, está la ventaja adicional de que los drivers JDBC no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de driver.

Tipos de drivers

Para conectarse a bases de datos individuales, JDBC necesita un driver específico para cada una de ellas. Hay cuatro tipos de drivers, que se describen a continuación; los drivers de los dos primeros tipos están destinados a los programadores de aplicaciones, mientras que los dos últimos son los que se utilizan comercialmente en la explotación de bases de datos.

PUENTE JDBC-ODBC

Éste es el driver que *Sun Microsystems* incluye en la plataforma Java 2. Proporciona acceso a bases de datos desde JDBC a través de uno o más drivers ODBC, representando una forma de aprovechar todo lo que ya exista, sin necesidad de realizar configuraciones adicionales.

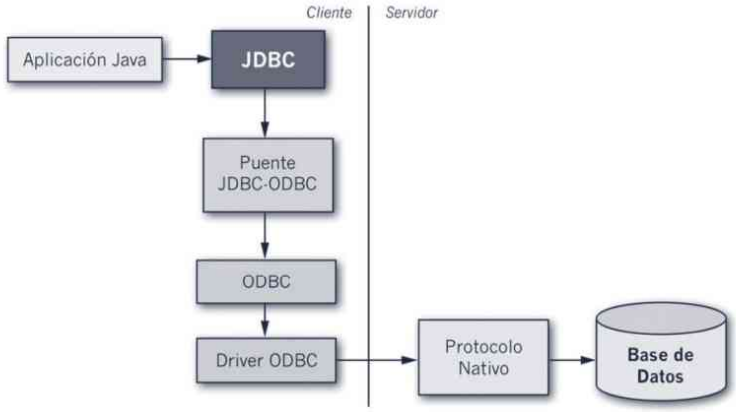


Figura 19.4

Este driver proporciona una forma excelente de acercarse a JDBC, y puede resultar útil cuando ya hay drivers ODBC instalados en las máquinas cliente, como suele ser el caso de las máquinas que están ejecutando aplicaciones *Windows*.

Sin embargo, este driver es inadecuado cuando se trata de aplicaciones de cierta entidad, porque el rendimiento se resiente enormemente al ser necesario realizar la conversión de las transacciones de JDBC a ODBC. Además, este driver no soporta todas las características de Java y el programador se ve limitado por la funcionalidad que ofrezca el driver ODBC que esté utilizando.

Normalmente, el driver ODBC se carga de forma local a través de una librería *DLL*, lo cual impide el acceso a datos a través de la red. Por ello, los drivers JDBC de este tipo son adecuados solamente en tiempo de desarrollo, ya que para utilizarlos en un entorno de red hay que recurrir a RMI, que replica una conexión local en una base de datos remota, incluyendo una nueva capa que hace resentirse aún más el rendimiento de la aplicación.

JAVA/BINARIO

Este driver es una variación del anterior en la que se realiza una comunicación directa con la librería que proporciona el fabricante. El rendimiento es mejor que el del puente JDBC-ODBC porque contiene la parte de código independiente del sistema ya compilada para el acceso a la base de datos de que se trate, cuyo driver debe estar cargado en cada máquina cliente.

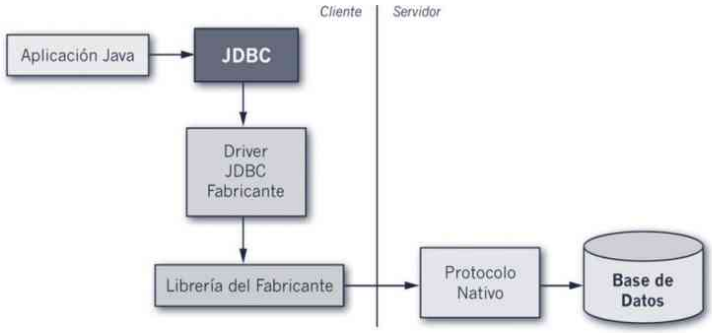


Figura 19.5

Actualmente, sólo *IBM* y *Oracle* proporcionan este tipo de driver para acceso a sus bases de datos, aunque *WebLogic* ofrece drivers de este tipo para acceder a las bases de datos más populares, como Microsoft SQL, Oracle y Sybase.

Este tipo de drivers elimina la parte ODBC de los drivers de tipo JDBC-ODBC, y proporcionan rendimientos aceptables en entornos intranet.

100% JAVA/PROTOCOLO NATIVO

Éste es sin duda el driver más utilizado y se le conoce también como *driver de protocolo de red*, ya que está realizado totalmente en Java y convierte directamente las peticiones JDBC del cliente en peticiones de red contra el servidor. Esta conversión se realiza en el cliente.

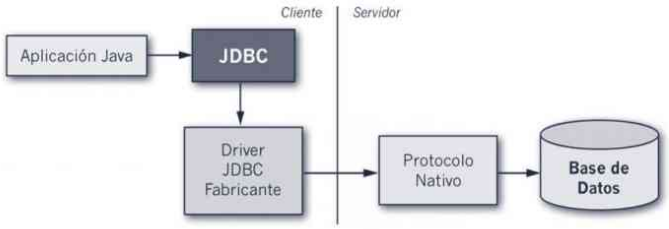


Figura 19.6

Este driver puede manejar múltiples clientes Java conectados a múltiples bases de datos. Además, como puede conectarse a diferentes direcciones de red, permite desarrollar aplicaciones cliente/servidor que sigan el modelo de tres capas comentado anteriormente. El driver se ejecuta como un servicio (o demonio) sobre el servidor.

100% JAVA/PROTOCOLO INDEPENDIENTE

Éste es el más flexible y se le conoce también como *driver de protocolo nativo*. El driver convierte las llamadas JDBC directamente a llamadas nativas hacia el sistema DBMS de que se trate. Esta conversión se realiza en el servidor, el cliente puede utilizar un driver JDBC genérico.

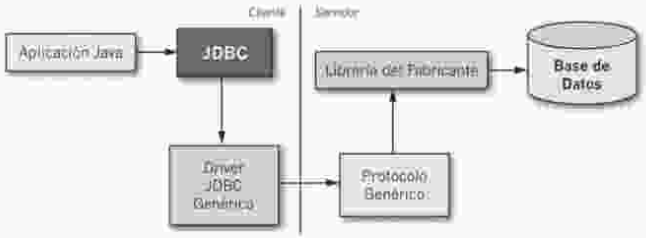


Figura 19.7

La ventaja de este tipo de drivers es que no necesita nada especial en el cliente, sino que todo el control se ejerce desde el servidor, lo cual hace muy simple el cambio de un servidor a otro. Sin embargo, estos drivers, al estar desarrollados por los fabricantes de DBMS, están optimizados para proporcionar toda la funcionalidad que cada uno de estos fabricantes ofrece, pero no tienen en cuenta las ventajas o características que pueda ofrecer el sistema operativo sobre el que se ejecutan.

Los ejemplos del Tutorial están referidos al driver que se proporciona con ***MySQL***, una base de datos de distribución gratuita que dispone de interfaces de administración y explotación muy cómodas. Si el lector utiliza el puente JDBC-ODBC y bases de datos ***Access***, será necesario que tenga en cuenta la restricción de que la base de datos debe residir en el mismo ordenador en el que se ejecute la aplicación que accede a sus datos. También puede el lector utilizar ***PostgreSQL***, un sistema de base de datos relacional que une las estructuras hasta ahora utilizadas en la programación clásica con los conceptos de programación orientada a objetos y también con distribución gratuita. Además, PostgreSQL, comparada con otros DBMS existentes, ofrece una funcionalidad, potencia, rendimiento y compatibilidad con ANSI SQL, al menos igual y permite realizar las tareas de administración y explotación con bastante comodidad, si bien su interfaz no es demasiado cómoda, como ocurre en la mayoría de los productos Linux. Y, la opción que proporciona el JDK es utilizar ***JavaDB***, que se distribuye conjuntamente con él y proporciona una potencia similar a cualquiera de las anteriores.

Primera aproximación a JDBC

JDBC define varias interfaces que permiten realizar operaciones con bases de datos; a partir de ellas se derivan las clases correspondientes. La lista siguiente recopila las clases e interfaces junto con una breve descripción.

Driver, permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.

DriverManager, permite gestionar todos los drivers instalados en el sistema.

DriverPropertyInfo, proporciona diversa información acerca de un driver.

Connection, representa una conexión con una base de datos. Una aplicación puede tener más de una conexión a más de una base de datos.

DatabaseMetadata, proporciona información acerca de una base de datos, como las tablas que contiene, número de filas o registros, etc.

Statement, permite ejecutar sentencias SQL sin parámetros.

PreparedStatement, permite ejecutar sentencias SQL con parámetros de entrada.

CallableStatement, permite ejecutar sentencias SQL con parámetros de entrada y salida, generalmente procedimientos almacenados.

ResultSet, contiene las filas o registros obtenidos al ejecutar una sentencia SELECT.

ResultSetMetadata, Permite obtener información sobre un **ResultSet**, como el número de columnas, sus nombres, etc.

La aplicación que se desarrolla a continuación va a permitir crear una tabla en el servidor, en este caso una base de datos *MySQL*, utilizando el driver JDBC de protocolo nativo proporcionado por *MySQL* en un fichero JAR que debe colocarse en el directorio *$JAVA_HOME/jre/lib/ext*. La estructura de todas las aplicaciones sigue un patrón semejante: establecer la conexión a la base de datos, ejecutar las sentencias SQL previstas y recuperar los resultados. La fuente de casi todos los problemas suele ser la conexión, por lo que el lector deberá extremar los cuidados a la hora de configurar su sistema.

El diagrama de la figura 19.8 relaciona las cuatro clases principales que va a usar cualquier programa Java con JDBC, y representa el esqueleto de cualquiera de los programas que se desarrollan para atacar a bases de datos.

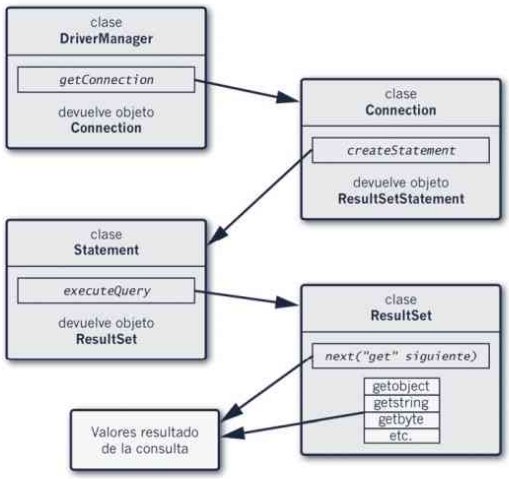


Figura 19.8

La aplicación Java1901.java es un ejemplo en donde se aplica el esquema anterior: establece una conexión, crea una tabla y rellena algunos datos iniciales.

```
class Java1901 {
  static public void main( String[] args ) {
    Connection conexion;
    Statement sentencia;
    ResultSet resultado;
    System.out.println( "Iniciando programa." );
    // Se carga el driver JDBC para MySQL
    try {
       Class.forName("com.mysql.jdbc.Driver").newInstance();
     } catch( Exception e ) {
      System.out.println( "No se pudo cargar el driver JDBC: "+
        e.getLocalizedMessage() );
      return;
      }
    try {
      // Se establece la conexión con la base de datos
      conexion = DriverManager.getConnection(
        "jdbc:mysql://localhost/tutorial?"+
        "user=tutorial&password=tutorialdb" );
      sentencia = conexion.createStatement();
      try {
        // Se elimina la tabla en caso de que ya existiese
        sentencia.execute( "DROP TABLE IF EXISTS LIBROS" );
      } catch( SQLException e ) {
        }
      // Esto es código SQL
      sentencia.execute( "CREATE TABLE LIBROS ("+
        " TITULO VARCHAR(50) NOT NULL, "+
        " AUTOR VARCHAR(30) NOT NULL, "+
        " PRECIO FLOAT NOT NULL, "+
        " PUBLICACION DATETIME) " );
      sentencia.execute( "INSERT INTO LIBROS "+
        "VALUES('JavaServer Pages: Manual de Usuario y Tutorial',"+
        "'A. Froufe',18.00,'2001-12-28')" );
      sentencia.execute( "INSERT INTO LIBROS "+
        "VALUES('JAVA 2: Manual de Usuario y Tutorial',"+
        "'A. Froufe',35.75,'2002-07-28')" );
      sentencia.execute( "INSERT INTO LIBROS "+
        "VALUES('J2ME: Manual de Usuario y Tutorial',"+
        "'A. Froufe, P. Jorge',31.90,'2003-12-28')" );
      sentencia.execute( "INSERT INTO LIBROS "+
        "VALUES('Java Pocket Guide',"+
        "'R. Liguori, P. Liguori',4.5,'2008-03-22')" );
    } catch( Exception e ) {
      System.out.println( e );
      return;
      }
    System.out.println( "Creacion finalizada." );
    }
  }
```

Seguidamente se revisan un poco más detenidamente las partes más interesantes del código, comenzando por las líneas que cargan el driver JDBC para acceder al esquema *MySQL*, utilizando el método *forName()* de la clase **Class**.

```
try {
   Class.forName("com.mysql.jdbc.Driver").newInstance();
 } catch( Exception e ) {
  System.out.println( "No se pudo cargar el driver JDBC: "+
    e.getLocalizedMessage() );
  return;
  }
```

No es necesario crear una instancia del driver y registrarlo con **DriverManager**, ya que la llamada al método *forName()* lo hace automáticamente. Si se crease una instancia, aunque no ocurriría ningún desastre, se estaría creando un duplicado innecesario.

Al indicar el driver a usar en el código, si se quiere cambiar a otro tipo de DMBS, por ejemplo, *PostgreSQL*, es necesario regenerar la llamada al método *forName()* y recompilar el programa. La llamada a este método en el caso de PostgreSQL sería:

```
Class.forName( "postgresql.Driver" );
```

O en el caso de utilizar el driver embebido de *JavaDB*, la llamada se convierte en:

```
Class.forName( "org.apache.derby.jdbc.EmbeddedDriver" );
```

El driver JDBC, además de poder ser codificado en el código, lo que restringe el uso del applet o aplicación a ese driver específico, también se puede cargar indicándolo en la línea de comandos a la hora de lanzar la aplicación:

```
java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver Programa
```

La sentencia anterior cargaría el driver correspondiente al puente JDBC-ODBC que proporciona directamente la plataforma Java 2. Este método solamente funciona con aplicaciones, no con applets. Sin embargo, la aplicación no está restringida al uso de un tipo específico de DBMS, sino que basta con indicar otro driver diferente para cambiar de sistema de base de datos, sin necesidad de recompilar la aplicación.

Una vez cargado el driver, ya es posible realizar la conexión al DBMS. Esta conexión se realiza solicitándolo a **DriverManager**. En este caso, se accede a la base de datos, o esquema en el lenguaje *MySQL*, **tutorial**, especificándolo mediante el parámetro correspondiente al driver de *MySQL*, **jdbc:mysql://localhost/tutorial**.

```
conexion = DriverManager.getConnection(
  "jdbc:mysql://localhost/tutorial?"+
  "user=tutorial&password=tutorialdb" );
```

Los dos últimos argumentos de la llamada, `user` y `password`, corresponden al nombre del usuario y la clave que permiten el acceso a la base de datos. La información que proporciona el fabricante del driver que se esté utilizando, generalmente indica la forma de acceder al DBMS que soporta. En el caso del puente JDBC-ODBC, la sintaxis es más simple ya que no es necesario que indicar el *host* en el que se encuentra la base de datos, ni tampoco el *puerto* de ese *host* habilitado para

el acceso, de forma que la URL JDBC de la base de datos sería **jdbc:odbc:Tutorial**, mientras que la conexión a PostgreSQL por ejemplo, en su sintaxis más compleja, la conexión se indicaría como:

```
jdbc:postgresql://host:puerto/base_datos
```

La siguiente línea es la que crea un objeto de tipo **Statement**, a través del cual se van a ejecutar todas las sentencias SQL, utilizando la conexión anterior.

```
sentencia = conexion.createStatement();
```

Para ejecutar las sentencias, están disponibles dos métodos, en función de que sea necesario o no que la ejecución de la sentencia SQL devuelva algún tipo de información:

execute(String sentencia), permite ejecutar una petición SQL que no devolverá ningún tipo de datos.

executeQuery(String sentencia), para ejecutar una consulta SQL que devuelve un objeto de tipo **ResultSet**.

La primera de las sentencias que se ejecutan en el programa es el borrado de cualquier tabla que existiese. Como el lector seguramente probará varias veces la ejecución de este programa, la eliminación de la tabla si se había creado previamente en una ejecución anterior del ejemplo, deja el campo libre a una nueva creación. Sin embargo, si la base de datos no existiese, es posible que se produjese un error, por lo que se necesita envolver la ejecución de la sentencia en un bloque `try-catch` para prevenir tal circunstancia.

```
sentencia.execute( "DROP TABLE IF EXISTS LIBROS" );
```

La línea siguiente es la encargada de crear la tabla que va a permitir insertar registros correspondientes a la información que contendrá la tabla, en este caso correspondiente a libros. En esta sentencia SQL se indica el nombre de cada campo del registro, el tipo de información que almacenará y si se permite que el campo esté vacío o no. Solamente la fecha de publicación, en esta sentencia, se permite que pueda quedar en blanco.

```
sentencia.execute( "CREATE TABLE LIBROS ("+
  " TITULO VARCHAR(50) NOT NULL, "+
  " AUTOR VARCHAR(30) NOT NULL, "+
  " PRECIO FLOAT NOT NULL, "+
  " PUBLICACION DATETIME) " );
```

El resto de las sentencias SQL del ejemplo introducen información en la tabla a través de sentencias `INSERT`, que también hay que envolver en un bloque `try-catch` para capturar la excepción de tipo **SQLException** generada en caso de producirse algún error en la transacción o ejecución de la sentencia a través de JDBC.

Para recuperar la información almacenada en la tabla, se recurre al método *executeQuery()*, que permite ejecutar sentencias SQL de consulta hacia la base de datos. Si en el ejemplo anterior, las líneas correspondientes a las sentencias de inserción de datos se sustituyen por la ejecución de la consulta que se reproduce en el código siguiente, se presentará en pantalla el texto con los datos correspondientes a cada uno de los registros introducidos.

```
resultado = sentencia.executeQuery( "SELECT * FROM LIBROS" );
while( resultado.next() ) {
  String titulo = resultado.getString( "TITULO" );
  String autor = resultado.getString( "AUTOR" );
  float precio = resultado.getFloat( "PRECIO" );
  Date publicacion = resultado.getDate( "PUBLICACION" );
  System.out.println( "El libro " + titulo + " del autor " +
    autor + " (" + precio + " Euros), fue publicado en " +
    publicacion );
  }
```

El método *next()* recuperará la información correspondiente al primer registro de la tabla en su ejecución inicial, para luego recorrer los demás registros. Devuelve `true` o `false` si existe el registro o no, respectivamente, de forma que es sencilla la realización del bucle que recorre la tabla al completo.

Los métodos que se utilizan para recuperar la información de las columnas del registro corresponden a los distintos tipos de datos que se pueden almacenar en ellos. En el código de ejemplo anterior se utilizan *getString()*, *getInt()* y *getDate()*, pero hay disponibles métodos para el resto de los tipos.

El objeto **ResultSet** devuelto por el método *executeQuery()*, permite recorrer las filas obtenidas, pero no proporciona información referente a la estructura de cada una de ellas; para ello se utiliza **ResultSetMetaData**, que permite obtener el tipo de cada campo o columna, su nombre, si es de tipo autoincremento o no, si es sensible o no a mayúsculas, si se puede escribir en dicha columna, si admite valores nulos, etc.

En general y a modo de resumen, los objetos que se van a poder encontrar en una aplicación que utilice JDBC serán los que se indican a continuación.

Connection

Representa la conexión con la base de datos. Es el objeto que permite realizar las consultas SQL y obtener los resultados de dichas consultas. Es el objeto base para la creación de los objetos de acceso a la base de datos.

DriverManager

Encargado de mantener los drivers que están disponibles en una aplicación concreta. Es el objeto que mantiene las funciones de administración de las operaciones que se realizan con la base de datos.

Statement

Se utiliza para enviar las sentencias SQL simples, aquellas que no necesitan parámetros, a la base de datos.

PreparedStatement

Tiene una relación de herencia con el objeto **Statement**, añadiéndole la funcionalidad de poder utilizar parámetros de entrada. Además, tiene la particularidad de que la pregunta ha sido compilada antes de ser realizada, por lo que se denomina *preparada*. La principal ventaja, aparte de la utilización de parámetros, es la rapidez de ejecución de la consulta.

CallableStatement

Tiene una relación de herencia con el objeto **PreparedStatement**. Permite utilizar funciones implementadas directamente sobre el sistema de gestión de la base de datos. Teniendo en cuenta que éste posee información adicional sobre el uso de las estructuras internas, índices, etc.; las funciones se realizarán de forma más eficiente. Este tipo de operaciones es muy utilizado en el caso de tratarse de funciones muy complicadas o bien que vayan a ser ejecutadas varias veces a lo largo del tiempo de vida de la aplicación.

ResultSet

Contiene la tabla resultado de la pregunta SQL que se haya realizado. En párrafos anteriores se han comentado los métodos que proporciona este objeto para recorrer dicha tabla.

TRANSACCIONES

Anteriormente se ha tratado de la creación y uso de sentencias SQL, que siempre se obtienen llamando a un método de un objeto de tipo **Connection**, como *createStatement()* o *preparedStatement()*. El uso de transacciones también se controla mediante métodos del objeto **Connection**. Como también se ha indicado, **Connection** representa una conexión a una base de datos dada, es decir, representa el lugar adecuado para el manejo de transacciones, dado que éstas afectan a todas las sentencias ejecutadas sobre una conexión a la base de datos.

Por defecto, una conexión funciona en modo *autocommit*, es decir, cada vez que se ejecuta una sentencia SQL se abre y se cierra automáticamente una transacción que afecta únicamente a dicha sentencia. Es posible modificar esta opción mediante *setAutoCommit()*, mientras que *getAutoCommit()* indica si se encuentra en modo *autocommit* o no. Si no se está trabajando en modo *autocommit* será necesario cerrar explícitamente las transacciones mediante *commit()* si tienen éxito, o *rollback()* si fallan; obsérvese que, tras cerrar una transacción, la próxima vez que se ejecute una

sentencia SQL se abrirá automáticamente una nueva, por lo que no existe ningún método que permita iniciar una transacción.

Es posible también especificar el nivel de aislamiento de una transacción mediante *setTransactionIsolation()*, así como averiguar cuál es el nivel de aislamiento de la actual mediante *getTransactionIsolation()*. Los niveles de aislamiento se representan mediante las constantes que se muestran en la lista siguiente, en la cual se explica muy básicamente el efecto de cada nivel de aislamiento.

`TRANSACTION_NONE`, no se pueden utilizar transacciones.

`TRANSACTION_READ_UNCOMMITTED`, desde esta transacción se pueden llegar a ver registros que han sido modificados por otra transacción, pero no guardados, por lo que es posible llegar a trabajar con valores que nunca lleguen a guardarse realmente.

`TRANSACTION_READ_COMMITTED`, se ven sólo las modificaciones ya guardadas por otras transacciones.

`TRANSACTION_REPEATABLE_READ`, si se leyó un registro, y otra transacción lo modifica, guardándolo, y se vuelve a leer, se seguirá viendo la información que había cuando se leyó por primera vez. Es decir, evita la situación en que una transacción lee una fila, una segunda transacción altera esa fila y la primera transacción vuelve a leer la fila, obteniendo valores diferentes a la primera lectura. Esto proporciona un nivel de consistencia mayor que los niveles de aislamiento anteriores.

`TRANSACTION_SERIALIZABLE`, se verán todos los registros tal y como estaban antes de comenzar la transacción, independientemente de las modificaciones que otras transacciones hagan ni de que se haya leído antes o no. Si se añadió algún nuevo registro, tampoco se verá.

Además de manejar transacciones, el objeto **Connection** proporciona algunos otros métodos que permiten especificar características de una conexión a una base de datos; por ejemplo, los métodos *isReadOnly()* y *setReadOnly()* permiten averiguar si una conexión a una base de datos es de sólo lectura, o hacerla de sólo lectura. El método *isClosed()* permite averiguar si una conexión está cerrada o no, y el método *nativeSQL()* permite obtener la cadena SQL que el driver enviaría a la base de datos si se tratase de ejecutar la cadena SQL especificada, permitiendo averiguar qué es exactamente lo que se le envía a la base de datos.

INFORMACIÓN DE LA BASE DE DATOS

Falta aún una pieza importante a la hora de trabajar con la conexión a la base de datos mediante el objeto **Connection**, y es la posibilidad de poder interrogar sobre las características de una base de datos; por ejemplo, puede ser interesante saber si la base de datos soporta cierto nivel de aislamiento en una transacción, como puede ser `TRANSACTION_SERIALIZABLE`, que muchos gestores no soportan. Para esto hay otra de

las interfaces que proporciona JDBC, **DatabaseMetaData**, a la que es posible interrogar sobre las características de la base de datos con la que se está trabajando. Es posible obtener un objeto de tipo **DatabaseMetaData** mediante la llamada al método *getMetaData()* de la clase **Connection**.

DatabaseMetaData proporciona diversa información sobre una base de datos, y cuenta con varias docenas de métodos, a través de los cuales es posible obtener gran cantidad de información acerca de una tabla; por ejemplo, *getColumns()* devuelve las columnas de una tabla, *getPrimaryKeys()* devuelve la lista de columnas que forman la clave primaria, *getIndexInfo()* devuelve información acerca de sus índices, mientras que *getExportedKeys()* devuelve la lista de todas las claves ajenas que utilizan la clave primaria de esta tabla, y *getImportedKeys()* las claves ajenas existentes en la tabla. El método *getTables()* devuelve la lista de todas las tablas en la base de datos, mientras que *getProcedures()* devuelve la lista de procedimientos almacenados. Muchos de los métodos de **DatabaseMetaData** devuelven un objeto de tipo **ResultSet** que contiene la información deseada. El listado que se presenta a continuación muestra el código necesario para obtener todas las tablas de una base de datos.

```
String nombreTablas = "%";          // Listamos todas las tablas
String tipos[] = new String[1];     // Listamos sólo tablas
tipos[0] = "TABLE";
DatabaseMetaData dbmd = conexion.getMetaData();
ResultSet tablas = dbmd.getTables( null,null,nombreTablas,tipos );
boolean seguir = tablas.next();
while( seguir ) {
  // Mostramos sólo el nombre de las tablas, guardado
  // en la columna "TABLE_NAME"
  System.out.println(
    tablas.getString( tablas.findColumn( "TABLE_NAME" ) ) );
    seguir = tablas.next();
  };
```

Hay todo un grupo de métodos que permiten averiguar si ciertas características están soportadas por la base de datos; entre ellos, destacan *supportsGroupBy()* que indica si se soporta el uso de `GROUP BY` en un `SELECT`, mientras que *supportsOuterJoins()* indica si se pueden llevar a cabo *outer-joins*. El método *supportsTransactions()*, comentado antes, indica si cierto tipo de transacciones está soportado o no. Otros métodos de utilidad son *getUserName()*, que devuelve el nombre del usuario actual o *getURL()*, que devuelve la URL de la base de datos actual.

DatabaseMetaData proporciona muchos otros métodos que permiten averiguar cosas tales como el máximo número de columnas utilizable en un `SELECT`, etc. En general, casi cualquier pregunta sobre las capacidades de la base de datos se puede contestar invocando a los métodos del objeto **DatabaseMetaData**, que merece la pena que el lector consulte cuando no sepa si cierta característica está soportada.

TIPOS SQL EN JAVA

Muchos de los tipos de datos estándar de SQL'92, no tienen un equivalente nativo en Java. Para superar esta deficiencia, se deben *mapear*, o convertir, los tipos de datos SQL en Java utilizando las clases JDBC para acceder a los tipos de datos SQL. Es necesario saber cómo recuperar adecuadamente tipos de datos Java como `int`, `long`, o **String**, a partir de sus contrapartidas SQL almacenadas en base de datos. Esto puede ser muy importante a la hora de trabajar con datos numéricos, que necesiten control decimal con precisión, o con fechas SQL, que tienen un formato muy bien definido.

El mapeo de los tipos de datos Java a SQL es realmente sencillo, tal como se muestra en la tabla que acompaña a este párrafo. Observe el lector que los tipos que comienzan por `java` no son tipos básicos, sino clases que tienen métodos para trasladar los datos a formatos utilizables, que son necesarias porque no hay un tipo de datos básico que mapee directamente su contrapartida SQL. La creación de estas clases debe hacerse siempre que se necesite almacenar un tipo de dato SQL en un programa Java, para poder utilizar directamente el dato desde la base de datos.

Java	SQL
`String`	`VARCHAR`
`Boolean`	`BIT`
`Byte`	`TINYINT`
`Short`	`SMALLINT`
`Int`	`INTEGER`
`Long`	`BIGINT`
`Float`	`REAL`
`Double`	`DOUBLE`
`byte[]-byte array: imagenes, sonidos...`	`VARBINARY (BLOBs)`
`java.sql.Date`	`DATE`
`java.sql.Time`	`TIME`
`java.sql.Timestamp`	`TIMESTAMP`
`java.math.BigDecimal`	`NUMERIC`
`Clob`	`CLOB`
`Array`	`ARRAY`
`Struct`	`STRUCT`
`Ref`	`REF`
`clase Java`	`JAVA_OBJECT`

El tipo de dato `byte[]`, es un array de bytes de tamaño variable. Esta estructura de datos guarda datos binarios, que en SQL son `VARBINARY` y `LONGVARBINARY`. Estos tipos se utilizan para almacenar imágenes, ficheros de documentos y cosas parecidas. Para almacenar y recuperar este tipo de información de la base de datos, se deben utilizar los métodos para control de *streams* que proporciona JDBC: *setBinaryStream()* y *getBinaryStream()*.

Una de las aportaciones de la versión 2.0 del API JDBC fue proporcionar soporte para tipos definidos por el usuario (*UDT*). JDBC proporciona un mecanismo para mapear tipos SQL a clases Java a través de los métodos *getObject()* y *setObject()*, pero

además, también permite utilizar una instancia de **java.util.Map** para realizar un mapeo específico de *UDTs*. Las clases en este mapa propio implementan la interfaz **SQLData** que incluye métodos para operar sobre tipos definidos por el usuario. Además, define el tipo `JAVA_OBJECT` para objetos Java y proporciona métodos que devuelven información sobre estos objetos Java y *UDTs*.

La base de datos *PostgreSQL* hace uso de esta característica y proporciona tipos para algunos de los que dispone Java, de forma que cuando se utiliza *getObject()* sobre un objeto de tipo **ResultSet**, devuelve un tipo específico **PG_Object**, que puede ser: `box`, `circle`, `line`, `lseg`, `path`, `point` y `polygon`. Las siguientes líneas muestran un ejemplo en el que se recupera un `point` de la columna 5 de un determinado registro, siendo `res` el objeto **ResultSet**:

```
PG_Object obj = (PG_Object)res.getObject( 5 );
PGpoint punto = obj.getPoint();
System.out.println( "Coordenadas: x="+punto.x+", y="+punto.y );
```

En la tabla siguiente se muestran los tipos Java correspondientes a cada uno de los tipos SQL.

SQL	Java
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Como se puede observar en la tabla, la conversión de tipos en el sentido de SQL a Java puede no estar tan clara, ya que hay tipos SQL cuyo tipo Java correspondiente puede no ser evidente, como `VARBINARY`, o `DECIMAL`, etc.

Existe una constante para cada tipo de dato SQL, que se encuentra declarada en **java.sql.Types**; así, al tipo `TIMESTAMP` le corresponde la constante **TIMESTAMP**, por ejemplo.

Además, JDBC proporciona clases Java nuevas para representar varios tipos de datos SQL: éstas son **java.sql.Date**, **java.sql.Time** y **java.sql.Timestamp**.

MODELO RELACIONAL DE OBJETOS

Este modelo intenta fundir la orientación a objetos con el modelo de base de datos relacional. Como muchos de los lenguajes de programación actuales, Java por ejemplo, son orientados a objetos, una estrecha integración entre los dos podría proporcionar una relativamente sencilla abstracción a los desarrolladores que programan en lenguajes orientados a objetos y que también necesitan *programar* en SQL. Esta integración, además, debería casi eliminar la necesidad de una constante traslación entre las tablas de la base de datos y las estructuras del lenguaje orientado a objetos, que es una tarea bastante ardua.

A continuación, se muestra un ejemplo muy simple para presentar la base del Modelo. Supóngase que se crea la siguiente tabla en una base de datos:

apellido	nombre	telefono	num_empleado
González	José	928 98 1112	00001
Gómez	Pedro	928 98 1113	00012
Pérez	Gonzalo	928 98 1114	00045
López	Alejandro	928 98 1115	00023

Con una relativa facilidad se puede mapear esta tabla en un objeto Java, tal como se muestra en el siguiente trozo de código:

```
class Empleado {
  int Clave;
  String Nombre;
  String Apellido;
  String Telefono;
  int Num_Empleado;
  Clave = Num_Empleado;
  }
```

Para recuperar esta tabla desde la base de datos a Java, simplemente se asignarían las columnas respectivas al objeto **Empleado** que se crearía previamente a la recuperación de cada fila:

```
Empleado objEmpleado = new Empleado();
objEmpleado.Nombre = resultset.getString( "nombre" );
objEmpleado.Apellido = resultset.getString( "apellido" );
objEmpleado.Telefono = resultset.getString( "telefono" );
objEmpleado.Num_Empleado = resultset.getInt( "num_empleado" );
```

Con una base de datos más grande, incluso con enlaces entre las tablas, el número de problemas se dispara, incluyendo la escalabilidad debida a los múltiples `JOINs` en el modelo de datos y los enlaces cruzados entre las claves de las tablas. Pero, afortunadamente, hay productos disponibles que permiten crear este tipo de puentes

entre los modelos relacional y orientado a objetos; es más, hay varias de estas soluciones que están siendo desarrolladas para trabajar específicamente con Java.

Uno de los ejemplos, para *Linux*, de este tipo de herramientas es la base de datos *PostgreSQL*, ejemplos de la cual ya se han mostrado al lector en este capítulo, que es un sistema de base de datos relacional que une las estructuras clásicas de estos sistemas con los conceptos de programación orientada a objetos, es decir, se trataría de una base de datos objeto-relacional. En *PostgreSQL*, por ejemplo, las *tablas* de denominan *clases*, las *filas* se denominan *instancias* y las *columnas* se denominan *atributos*. Además, un concepto que aparece en *PostgreSQL* y que viene claramente de la programación orientada a objetos es la *Herencia*, de forma que cuando se crea una nueva clase heredada de otra, la clase creada adquiere todas las características de la clase de la que proviene más las características que se definan en la nueva clase. Por poner un ejemplo, si se tiene una tabla creada con la sentencia siguiente:

```
CREATE TABLA tabla1 (
  campo1 text
  campo2 int )
```

A continuación, se puede crear una segunda tabla con la sentencia SQL:

```
CREATE TABLA tabla2 (
  campo3 int )
INHERITS( tabla1 )
```

Como resultado de esta sentencia SQL, la nueva tabla tendrá los atributos *campo1*, *campo2* y *campo3*.

MODELO DE CONEXIÓN

La conexión a bases de datos relacionales a través de JDBC realmente es muy simple, tal como se ha podido comprobar a lo largo de las explicaciones previas. El objetivo en este apartado es proporcionar, o intentarlo al menos, una plantilla que se pueda reutilizar y personalizar para aprender a manipular bases de datos con Java. Una vez que el ejemplo sea comprendido por el lector, no habrá problemas a la hora de saber qué hacer y cómo hacerlo. Y cuando se necesite más información, la documentación que proporciona *Sun Microsystems* sobre JDBC la tiene.

De todas las cosas que hay que tener en mente, la conexión de Java con la base de datos relacional es la primera de las preocupaciones. En el ejemplo `Java1902.java` se muestra cómo se crea y establece la conexión. También contiene varios métodos que procesan sentencias SQL habituales de una forma simple y segura: las conexiones son abiertas y cerradas con cada sentencia SQL. Si el lector está construyendo aplicaciones en las que se prevean grandes flujos de transacciones, tendrá que establecer una estrategia más adecuada; por ejemplo, si se pretenden realizar actualizaciones sobre un registro una vez que se haya accedido a él, probablemente sea mejor mantener abierta la conexión con la base de datos hasta que hayan concluido todas las actualizaciones.

Como la conexión a una base de datos es un objeto, se puede mantener en una variable tanto tiempo como se necesite; con lo cual, la aplicación será capaz de procesar las acciones mucho más rápidamente, pero corriendo el riesgo de que otros usuarios tengan bloqueado el acceso hasta que las conexiones que estén bloqueadas concluyan.

El ejemplo, evidentemente, asume que hay una base de datos disponible para usar y su esquema es muy simple, utilizando la base de datos del primer ejemplo del capítulo, con solamente tres campos. Mucha parte del desarrollo de este capítulo ha sido desarrollado en *Linux* utilizando la base de datos *MySQL*, aunque exactamente el mismo código, solamente cambiando la conexión a la base de datos se puede utilizar con *Microsoft Acc*ess, por ejemplo, sobre entornos *Windows*.

La parte cliente del ejemplo anterior es la que se ha codificado en el ejemplo `Java1903.java`, implementado como applet, que permite introducir una consulta en el campo de texto, y enviarla al servidor implementado en la clase **Java1902**, que a su vez enviará la consulta a la base de datos y devolverá el resultado al applet, el cual mostrará la información resultante de su consulta en la parte inferior.

En estos dos ejemplos se muestran los fundamentos de la interacción con bases de datos, de una forma un poco más complicada, de tal modo que no se ataca a la base de datos desde el mismo programa sino que se proporciona al lector una visión más amplia de la capacidad y potencia que se encuentra bajo la conjunción de Java y las bases de datos. El fichero `java1903.html` permite lanzar el applet mediante el visor *appletviewer*.

```
import java.io.*;
import java.net.*;
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Java1903 extends Applet {
  static final int puerto = 6700;
  String cadConsulta = "No hay consulta todavia";
  String cadResultado = "No hay resultados";
  Button boton;
  TextArea texto;
  List lista;

  public void init() {
    setLayout( new GridLayout( 5,1 ) );
    texto = new TextArea( 20,40 );
    lista = new List();
    boton = new Button( "Ejecutar Consulta" );
    boton.addActionListener( new MiActionListener() );
    add( new Label( "Escribir la consulta aqui..." ) );
    add( texto );
    add( boton );
    add( new Label( "y examinar los resultados aqui" ) );
    add( lista );
    resize( 800,800 );
    }
```

```
  void abreSocket() {
    Socket s = null;
    try {
      s = new Socket( getCodeBase().getHost(),puerto );
      BufferedReader sinstream =
      new BufferedReader(new InputStreamReader(s.getInputStream()));
      PrintStream soutstream = new PrintStream( s.getOutputStream() );
      soutstream.println( texto.getText() );
      lista.removeAll();
      cadResultado = sinstream.readLine();
      while( cadResultado != null ) {
        lista.add( cadResultado );
        cadResultado = sinstream.readLine();
        }
    } catch( IOException e ) {
      System.err.println( e );
    }    finally {
      try {
        if( s != null )
          s.close();
      } catch( IOException e ) {
        }
      }
    }

  class MiActionListener implements ActionListener {
    public void actionPerformed( ActionEvent evt ) {
      abreSocket();
      }
    }
  }
```

Una vez comprendidas las ideas básicas que se presentaban en los ejemplos anteriores, será fácil atreverse a escribir programas que manejen esquemas de bases de datos mucho más complicados. No obstante, si el lector tiene una buena base en el conocimiento de bases de datos relacionales, estará satisfecho con esta simplicidad.

El ejemplo `Java1904.java` es una simple interfaz que sirve para atacar a cualquier tipo de base de datos. Solamente se ha codificado el driver JDBC a utilizar, pero a través de la ventana que se presenta, se puede acceder a cualquier base de datos y cualquier tabla. La figura 19.9 muestra la ventana, una vez que se ha accedido a la base de datos que se ha estado utilizando a lo largo del capítulo.



Figura 19.9

En el código del ejemplo, que el lector puede consultar en el soporte digital que se incluye con el libro, se han incluido comentarios para que la comprensión sea sencilla.

JDBC ROWSETS

Un objeto **RowSet** contiene información tabular obtenida de una fuente de datos de una forma más flexible que **ResultSet**, porque al contrario que los objetos **ResultSet**, los objetos de tipo **RowSet** pueden operar sin estar conectados a la base de datos, hoja de cálculo o cualquier otra fuente de datos tabulares, resultando mucho más ligeros que los objetos **ResultSet**.

Los objetos **RowSet** siguen el modelo JavaBean en lo que respecta a propiedades y notificación de eventos. *Sun Microsystems* define varias interfaces para implementar otras tantas clases de objetos **RowSet**, proporcionando implementaciones para cada una de ellas. No obstante, los fabricantes de bases de datos también han desarrollado sus propias implementaciones de las interfaces **RowSet** adaptadas a las características específicas de sus productos.

Todos los objetos **RowSet** derivan de la interfaz **ResultSet** por lo que comparten todas sus capacidades, incorporando características nuevas cada uno de los tipos de **RowSet**. En principio, se pueden dividir las interfaces **RowSet** en dos grupos, por un lado las que generan objetos que necesitarían estar conectados a la base de datos y, por otro lado, las que generan objetos que pueden funcionar sin conexión directa a la base de datos, o fuente de datos que proporciona la información.

Solamente una de las implementaciones de *Sun* funciona con conexión a la base de datos, es ***JdbcRowSet***. Es muy similar a **ResultSet** y se utiliza, por ejemplo, para seleccionar un driver JDBC mediante una interfaz gráfica.

Las otras implementaciones funcionan sin conexión permanente a la base de datos. Además de las características que proporciona la interfaz **RowSet** de la que derivan, incorporan nuevas capacidades, como por ejemplo el permitir la serialización

de objetos, que junto con su capacidad para funcionar sin conexión a la fuente de datos, hace que estas implementaciones sean ideales para enviar datos a través de una red o para enviar datos a dispositivos con capacidades limitadas, como PDA o teléfonos móviles.

Las diferentes implementaciones que proporciona *Sun Microsystems*, además de **JdbcRowSet**, son las que se describen en las secciones siguientes.

CachedRowSet

Proporciona todas las capacidades de un objeto **JdbcRowSet** y además permite obtener una conexión a una fuente de datos y ejecutar una consulta, manipular los datos obtenidos haciendo cambios mientras está desconectado de la fuente de datos; reconectarse con la fuente de datos y volcar los cambios realizados y comprobar los conflictos y resolverlos con la fuente de datos.

La colección de datos que obtiene de la fuente de datos se mantiene en memoria mientras se realizan los cambios en los datos y cuando se vuelca de nuevo a la fuente de datos, se realiza la sincronización, si es necesaria. Por ello, no es adecuada para soportar grandes conjuntos de datos.

WebRowSet

A las capacidades aportadas por **CachedRowSet**, añade la posibilidad de volcar los datos como un documento XML y la capacidad de leer un documento XML que describa un objeto de tipo **WebRowSet**.

JoinRowSet

Proporciona todas las capacidades de un objeto de tipo **WebRowSet** y, por tanto, también de un objeto **CachedRowSet**, incorporando además la capacidad de ejecutar la sentencia `SQL JOIN` de datos provenientes de múltiples **RowSet** sin necesidad de tener que conectarse a la fuente de datos.

FilteredRowSet

También proporciona las capacidades de un objeto **WebRowSet**, permitiendo además aplicar filtros o criterios de selección de datos sobre los datos visibles, para obtener un subconjunto de esos datos. Equivale a ejecutar una consulta sobre un objeto **RowSet** sin tener que utilizar un lenguaje de consulta o tener que conectarse a la fuente de datos.

La aplicación `Java1905` utiliza un objeto de tipo **CachedRowSet** para mostrar la utilización de *RowSets*, porque es la implementación más general de las que proporciona *Sun Microsystems*.

Un objeto **CachedRowSet** se puede crear invocando a su constructor por defecto y rellenarlo a partir de los datos obtenidos mediante un **ResultSet**. Aunque esta no es una solución ideal, porque el resultado se mantiene en memoria y está ligado a la capacidad de ésta, piense el lector por ejemplo, en los datos generados por una consulta sobre el sistema de información de parcelas agrícolas, fácilmente podría desbordar la capacidad de la memoria del sistema.

El código necesario para crear un objeto **CachedRowSet** consiste simplemente en la invocación de su constructor sin argumentos.

```
CachedRowSet crs = new CachedRowSet();
```

Ahora es posible fijar las propiedades que satisfagan las necesidades para las que ha sido creado el objeto a través de los métodos *get/set* de la clase. Algunas de las propiedades que se pueden indicar incluyen el comando a ejecutar, concurrencia, tipo, fuente de datos, dirección url, usuario, contraseña, nivel de transacción, tamaño máximo del campo, número máximo de filas a recuperar, tiempo de espera, etc. Por ejemplo, el valor asignado por defecto a la propiedad de concurrencia es `CONCUR_READ_ONLY`, por lo que sí es necesario permitir la actualización de los valores de los registros que se recuperen en el objeto **CachedRowSet**, es necesario utilizar la siguiente sentencia:

```
crs.setConcurrency( ResultSet.CONCUR_UPDATETABLE );
```

También se puede indicar que se permita el desplazamiento por los registros que forman parte del objeto **CachedRowSet** generado en la ejecución del comando SQL. Esta característica se fija con la sentencia siguiente:

```
crs.setType( ResultSet.TYPE_SCROLL_INSENSITIVE );
```

Para recuperar las filas de la base de datos, hay que indicar la consulta a realizar. Para ello se fija la propiedad `Command`, de forma que cuando se invoque al método *execute()*, el comando sea ejecutado y el objeto **CachedRowSet** se rellene con los datos obtenidos. Pero antes de poder invocar a este método, es necesario indicar los parámetros que conciernen a la conexión como son la fuente de datos, el nombre del usuario y su contraseña de acceso a esa fuente de datos.

```
crs.setCommand();
crs.setDataSource();
crs.setUsername();
crs.setPassword();
```

Las demás propiedades son opcionales; por ejemplo, si se desea que no se puedan leer datos que no hayan sido grabados en la base de datos, para no obtener lecturas *sucias*, se puede fijar el nivel de transacción como se indica a continuación, pero si no se desea o tiene importancia, no es necesario fijar dicha propiedad.

```
crs.setTransactionIsolationLevel(
  Connection.TRANSACTION_READ_COMMITTED );
```

Como el objeto **CachedRowSet** sigue la especificación *JavaBean*, también tiene la posibilidad de participar en la notificación de eventos. Por ejemplo, en este caso podría haber un objeto al que fuese necesario indicar que los datos de **CachedRowSet** han sido actualizados. El programador del *Bean* que desee recibir notificaciones de **CachedRowSet**, debe implementar los métodos que define **RowSetListener**, que son *rowChanged()*, *rowSetChanged()* y *cursorMoved()*. Y esto se registraría en la instancia de **CachedRowSet** como un receptor de eventos:

```
crs.addRowSetListener( receptorEventos );
```

Para recuperar filas de la base de datos hay que invocar al método *execute()*.

```
crs.execute();
```

Esta sentencia generará un objeto **CachedRowSet** que contendrá los mismos datos que el objeto **ResultSet** generado por la misma sentencia que se ha indicado en la propiedad `Command` de **CachedRowSet**. La diferencia se encuentra en las propiedades que se hayan establecido para **CachedRowSet**. No obstante, lo anterior es cierto siempre que el driver JDBC soporte las características correspondientes a las propiedades que se hayan indicado, por ejemplo, la posibilidades de actualizar el objeto **CachedRowSet** no tendría efecto alguno si el driver JDBC no soporta esa característica.

Un método alternativo para obtener datos en un objeto **CachedRowSet** es volcar el resultado que se obtiene mediante un objeto **ResultSet**, utilizando el método *populate()*.

```
crs.populate( resultSet );
```

El desplazamiento por los registros de un objeto **CachedRowSet** es exactamente igual que en un objeto **ResultSet**. El cursor está posicionado por defecto antes de la primera fila y la primera llamada al método *next()* lo coloca apuntando al primer registro. Sucesivas llamadas a este método desplazarán el cursor por todos los registros que se hayan recuperado en el **CachedRowSet**. Por ejemplo, el siguiente fragmento de código itera sobre el objeto completo **CachedRowSet**, imprimiendo el título de cada uno de los libros recuperados de la tabla de la base de datos.

```
while( crs.next() ) {
  System.out.printf( "%s%n",crs.getString("TITULO") );
  }
```

Si el objeto **CachedRowSet** no permite el desplazamiento, la iteración se limita a una única vez y en dirección de inicio a fin. Con la propiedad de desplazamiento, el cursor se puede mover en cualquier dirección y puede colocarse sobre un registro tantas veces como se quiera. En este caso, es necesario permitir el desplazamiento para realizar las actualizaciones de los datos más fácilmente.

También con los objetos **CachedRowSet** las actualizaciones de datos son semejantes a las que se realizan sobre objetos **ResultSet**. Los métodos *updateXxx()*,

insertRow() y *deleteRow()* son heredados de **ResultSet** y se utilizan con el mismo propósito. Por ejemplo, si se quiere cambiar el precio del libro que ocupa la tercera fila, es necesario colocar el cursor sobre dicho registro y actualizar el campo correspondiente.

```
crs.absolute( 3 );
crs.updateFloat( 4,10.49f );
crs.updateRow();
```

La última sentencia es la encargada de concluir los cambios realizados en el registro. En este momento, el receptor de eventos registrado sobre el objeto **CachedRowSet**, si hay alguno, será notificado del cambio producido en dicho objeto.

En este instante, los cambios todavía no han sido volcados a la base de datos, porque solamente se han realizado en el objeto **CachedRowSet**, que no tiene conexión permanente con su base de datos. Para que los cambios se almacenen de forma permanente, es necesario invocar al método *acceptChanges()*.

```
crs.acceptChanges();
```

Este método restablece la conexión con la base de datos y, antes de grabar los nuevos valores en la base de datos, el componente **RowSet** compara los valores originales con los que se proporcionan en la actualización. En caso de que no haya conflicto, esos valores serán actualizados.

El ejemplo `Java1905.java` muestra la utilización de **CachedRowSet** para crear una interfaz SQL a fin de pasar datos entre clases y propagar los cambios realizados de nuevo a la base de datos.

En el escenario que muestra la aplicación, el objeto **CachedRowSet** se puede utilizar para pasar un conjunto de registros a un cliente ligero, como puede ser un dispositivo móvil o un ordenador portátil, y realizar las actualizaciones pertinentes en los registros, devolviendo los datos actualizados a la base de datos para su almacenamiento definitivo.

JDBC Y SERVLETS

Esta parte del capítulo, dedicada a JDBC, explora el mundo de los *servlets* en el contexto de JDBC. Si el lector no está cómodo con el uso de *servlets* o con los conocimientos que posee, en el capítulo anterior del Tutorial se trató a fondo este tema, al que puede recurrir y luego regresar a este punto de su lectura.

Hasta ahora se ha presentado la parte cliente del uso de JDBC, y en lo que se pretende adentrar al lector es en el uso de JDBC en el servidor para generar páginas Web a partir de la información obtenida de una base de datos. Si el lector está familiarizado con CGI, podrá comprobar que los *servlets* son herramientas más

potentes a la hora de generar páginas Web dinámicamente, por varias razones: velocidad, eficiencia en el uso de recursos, escalabilidad, etc.

La arquitectura de los servlets hace que la escritura de aplicaciones que se ejecuten en el servidor sea relativamente sencilla y se construyan aplicaciones muy robustas. La principal ventaja de utilizar servlets es que se puede programar sin dificultad la información que va a proporcionar entre peticiones del cliente. Es decir, se puede tener constancia de lo que el usuario ha hecho en peticiones anteriores e implementar funciones de tipo *rollback* o *cancel transaction* (suponiendo que el servidor de base de datos las soporte). Además, cada instancia del servlet se ejecuta dentro de una tarea Java, por lo que se pueden controlar las interacciones entre múltiples instancias; y al utilizar el identificador de sincronización, se puede asegurar que los servlets del mismo tipo esperan a que se produzca la misma transacción antes de procesar la petición; esto puede ser especialmente útil cuando mucha gente intenta actualizar al mismo tiempo la base de datos, o bien si hay muchos usuarios pendientes de consultas a la base de datos cuando ésta se encuentra en pleno proceso de actualización.

El ejemplo `Java1906.java`, es un servlet, que junto con la página Web asociada, `java1906.html` (renombrada a *index.html* en el despliegue sobre ***Tomcat***) y las dos tablas que utiliza, conforman un servicio completo de noticias o artículos. Los requisitos de la aplicación son simples, indicando que debe haber un número determinado de usuarios con autorización para enviar artículos, y otro grupo de usuarios que puedan ver los últimos artículos. Por supuesto, se podría complicar indicando que también se pudiese almacenar la fecha y la hora en que se colocan los artículos, y posiblemente también fuese útil la categorización, es decir, colocarlos dentro de una categoría como deportes, internacional, nacional, local, etc. De este modo, lo que aquí se esboza como la simple gestión de artículos, podría ser la semilla de un verdadero *servicio de noticias*. Pero no se pretende llegar a eso, sino simplemente presentar la forma en que se puede aunar la potencia de JDBC y los servlets para poder acceder a una base de datos, introduciendo algunas características como la comprobación de autorización, etc.; por ello, se ha huido conscientemente del uso de la palabra *noticias*, aunque el lector puede implementar sin demasiada dificultad un sistema de ese tipo.

Toda la información estará almacenada en una base de datos de la forma mejor posible; de modo que se puede escribir un servlet que almacene la información que le llegue en ficheros para luego proporcionarla a quien la consulte.

Además, también se quieren almacenar las contraseñas para el acceso al sistema de artículos en una tabla de la base de datos en vez de en la configuración del servidor Web. Hay dos razones fundamentales para ello: por un lado, porque es previsible que mucha gente utilice este servicio, y por otro lado, que debe ser posible asociar cada envío con una persona en particular. La inclusión en una tabla ayudará a manejar gran cantidad de usuarios así como a controlar quién envía artículos. Una mejora que se

puede hacer al sistema es el desarrollo de un applet JDBC que permita añadir y quitar usuarios, o bien asignar privilegios a quien haya enviado algún artículo al sistema.

Las siguientes líneas de código muestran las sentencias utilizadas en la creación de las tablas e índices que se van a utilizar en la aplicación que se desarrollará para completar el *sistema de Artículos*. La aplicación se ha pensado para atacar una base de datos *MySQL*, si el lector desea portarlo a otro driver JDBC, tendría que asegurarse que las sentencias SQL que se utilizan están soportadas por él.

```
CREATE TABLE usuarios (
  usuario VARCHAR(16) NOT NULL,
  nombre VARCHAR (60) NOT NULL,
  empresa VARCHAR (60) NOT NULL,
  admitirEnvio CHAR(1) NOT NULL,
  clave VARCHAR (8) NOT NULL, PRIMARY KEY(usuario) );
CREATE INDEX usuario_key ON usuarios(usuario);

CREATE TABLE articulos (
  titulo VARCHAR(255) NOT NULL,
  usuario VARCHAR(16) NOT NULL,
  cuerpo BLOB );
CREATE INDEX articulo_key ON articulos(usuario);
CREATE INDEX titulo_key ON articulos(titulo,usuario);
```

Como se puede observar, solamente hay dos tablas. Si se quisiera implantar una lista de categorias, sería necesario incorporar una nueva tabla, o bien añadir un campo más a la tabla de artículos. La clave primaria de la tabla de usuarios es el identificador de cada usuario, y en la tabla de artículos hay un índice compuesto formado por el título del artículo y el usuario que lo envió. Se usa un tipo `BLOB` (que en *Access*, ha de ser `MEMO`), para guardar el cuerpo del artículo. Una mejora, en caso de convertirlo en un sistema de noticias, consistiría en añadir la fecha en que se ha enviado la noticia, guardarla en un campo en la tabla de artículos y añadir el campo al índice compuesto de esa tabla, con lo cual se podrían presentar las noticias correspondientes a una fecha determinada sin aparente dificultad.

En la tabla de usuarios se guarda el identificador a través del cual el sistema va a reconocer al usuario, junto con la contraseña que determine para comprobar su identificación, más su nombre completo, la empresa a que pertenece y si ese usuario tiene autorización para el envío de artículos al sistema. Si un usuario no dispone de autorización para el envío, solamente podrá acceder a la lectura de artículos. Y por supuesto, si alguien no aparece en esta tabla, no tendrá acceso alguno al sistema.

Lo primero que hay que desarrollar es la página que va a dar acceso al sistema, para ver la forma de procesar los datos que van a llegar. La página es muy simple (figura 19.10), implementada en el fichero `java1906.html`.

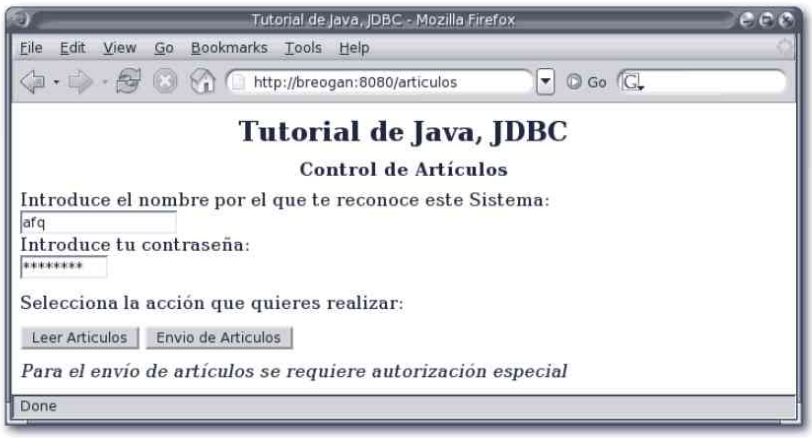


Figura 19.10

```
<HTML>
<HEAD><TITLE>Tutorial de Java, JDBC</TITLE></HEAD>
<BODY>
<FORM ACTION="http://localhost:8080/articulos/java1906"
  ENCTYPE="x-www-form-encoded" METHOD="POST">
  <H2><CENTER>Tutorial de Java, JDBC</CENTER></H2>
  <P><CENTER><B>Control de Artículos</B></CENTER></P>

  <P>Introduce el nombre por el que te reconoce este Sistema:<BR>
  <INPUT NAME="usuario" TYPE="text" SIZE="16" MAXLENGTH="16"><BR>
  Introduce tu contraseña:<BR>
  <INPUT NAME="clave" TYPE="password" SIZE="8" MAXLENGTH="8"></P>
  Selecciona la acción que quieres realizar:
  <P>
  <INPUT NAME="accion" TYPE="submit" VALUE="Leer Articulos">
  <INPUT NAME="accion" TYPE="submit" VALUE="Envio de Articulos">
  </P>
  </FORM>
<I>Para el envío de artículos se requiere autorización especial</I>
</BODY>
</HTML>
```

Lo primero que se necesita aclarar es cómo se van a procesar los datos del formulario. La página no puede ser más sencilla, tal como se puede ver en la captura de su visualización en el navegador, con dos botones para seleccionar la acción a realizar. Si se quiere enviar un artículo, no es necesario introducir el nombre y clave en esta página, ya que el servlet enviará una nueva página para la introducción del contenido del artículo, y ya sobre ésa sí que se establecen las comprobaciones de si el usuario está autorizado o no a enviar artículos al sistema.

La página que envía el servlet es la que reproduce la figura 19.11. Esta página se genera en el mismo momento en que el usuario solicita la inserción de un artículo. En caso de que haya introducido su identificación en la página anterior, el servlet la colocará en su lugar, si no, la dejará en blanco.

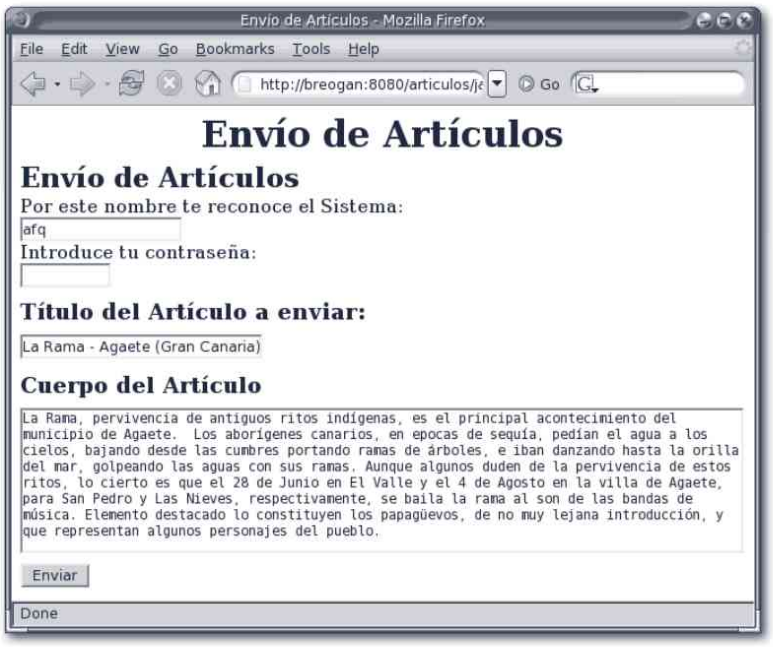


Figura 19.11

La imagen reproduce la página ya rellena, lista para la inserción de un nuevo artículo en el sistema. Cuando se pulsa el botón de envío de artículo y el servlet recibe la petición de inserción del artículo en el sistema, es cuando realiza la comprobación de autorización; por un lado, si es un usuario reconocido para el sistema y, en caso afirmativo, si está autorizado al envío de artículo, o solamente puede leerlos.

Debe recordar el lector que los nombres de los campos de entrada de datos se pueden obtener con *getParameter()* y recuperar la información que contienen. Los parámetros *nombre* y *clave* en el formulario se mapean en las variables *usuario* y *clave* en el servlet, que serán las que se utilicen para realizar las comprobaciones de acceso del usuario.

El código completo del servlet se implementa en el fichero `Java1906.java`, cuyo contenido el lector puede consultar en el soporte digital proporcionado con el libro.

Seguidamente, se repasan los trozos de código más interesantes del servlet, tal como se ha hecho en muchos de los ejemplos del Tutorial. Una de las primeras cosas necesarias consiste en reconocer cuándo se ha recibido una petición correctamente, y para ello se utiliza el código que aparece en la línea siguiente:

```
res.setStatus( res.SC_OK );
```

La línea que sigue a ésta es la que indica que el contenido es HTML, porque la intención del servlet es enviar respuestas en este formato al navegador. Esto se indica en la línea:

```
res.setContentType( "text/html" );
```

Otro trozo de código interesante es el utilizado para saber cuál de los botones de la página inicial se ha pulsado, en donde se recurre al método *getParameter()* sobre el nombre del botón (**accion**), buscando los valores que se han asignado en el código fuente, y procesar aquel de los dos que se haya pulsado. Las líneas de código siguientes son las que realizan estas acciones.

```
String accion = req.getParameter( "accion" );
try {
  autorizado = autorizacion( req );
  if( accion.equals( "Leer Articulos" )
    && !autorizado.equals( "ACCESO DENEGADO" ) ) {
    leerArticulo( req,res );
  } else if( accion.equals( "Enviar" )
    && autorizado.equals( "POST" ) ) {
    enviarArticulo( req,res );
  } else if( accion.equals("Envio de Articulos" ) ) {
  . . .
  }
} catch( SQLException e ) {
  . . .
  }
```

Como se ha especificado en la página inicial de acceso un valor para cada uno de los botones, se sabe cuáles deben buscarse y qué hacer para procesarlos. También se pueden recuperar todos los parámetros que hay en un formulario a través del método *getParameterNames()*.

Una vez que se conocen tanto el usuario como la clave, hay que comprobar esta información contra la base de datos. Para ello se utiliza una consulta para saber si el usuario figura en la base de datos. El código que realiza estas acciones es el que se reproduce en las siguientes líneas.

```
public String autorizacion( HttpServletRequest req )
  throws SQLException {
  Statement stmt = con.createStatement();
  String consulta;
  ResultSet rs;
  String valido = "";
  String usuario = req.getParameter( "usuario" );
  String clave = req.getParameter( "clave" );
  String permiso="";
  consulta = "SELECT admitirEnvio FROM usuarios WHERE usuario = '"+
    usuario;
  consulta += "' AND clave = '"+clave+"'";
  rs = stmt.executeQuery( consulta );
  while( rs.next() ) {
    valido = rs.getString(1);
    }
  rs.close();
  stmt.close();
  if( valido.equals( "" ) ) {
    permiso = "ACCESO DENEGADO";
    }
  else {
```

```
    // Permiso sólo para lectura de artículos
    if ( valido.equals( "N" ) ) {
      permiso = "GET";
      // Permiso para lectura y envío de artículos
      }
    else if ( valido.equals( "S" ) ) {
      permiso = "POST";
      }
    }
  return permiso;
  }
```

Para realizar las consultas a la base de datos, es necesario crear una conexión con ella. Para hacerlo se utiliza el método *init()* de la clase **HttpServlet**, en donde se instancia la conexión con el servidor de base de datos, como se muestra en las líneas de código siguientes.

```
  String DBurl = "jdbc:mysql://localhost/tutorial";
  String usuarioDb = "tutorial";
  String claveDb = "tutorialdb";
. . .
      Class.forName( "com.mysql.jdbc.Driver" );
      con = DriverManager.getConnection(
        DBurl+"?user="+usuarioDb+"&password="+claveDb );
. . .
```

Ésta es solamente la parte de la conexión, hay más código implicado, pero ya se han visto bastantes ejemplos. No obstante, hay que tener en cuenta que la conexión se puede romper normalmente por exceso de tiempo, es decir, que si no se realizan acciones contra la base de datos en un tiempo determinado, la conexión se rompe. Así que una de las cosas que debe controlar el lector si va a utilizar este código es introducir la reconexión en la parte de código que trata la excepción de SQL.

Una vez creada la conexión, hay que realizar consultas para la lectura de datos. El siguiente código hace esto, consultando la base de datos y recogiendo la información de todos los artículos. Se puede incorporar fácilmente un nuevo campo con la fecha, para obtener los artículos ordenados por la fecha en que han sido enviados, o bien realizar consultas diferentes para obtener los artículos ordenados por usuario, etc.

```
consulta = "SELECT articulos.cuerpo,articulos.titulo," );
consulta += articulos.usuario,usuarios.nombre,usuarios.empresa ";
consulta += "FROM articulos,usuarios WHERE " );
consulta += articulos.usuario=usuarios.usuario";
rs = stmt.executeQuery( consulta );
PrintWriter out = new PrintWriter( res.getOutputStream() );
out.println( "<HTML>" );
out.println( "<HEAD><TITLE>Artículos Enviados</TITLE></HEAD>" );
out.println( "<BODY>" );
while( rs.next() ) {
  out.println( "<H2>" );
  out.println( rs.getString(1) );
  out.println( "</H2><p>" );
  out.println( "<I>Enviado desde: "+rs.getString(5)+"</I><BR> " );
  out.println( "<B>"+rs.getString(2)+"</B>, por "+rs.getString(4) );
```

```
  out.println( "<HR>" );
  }
out.println( "</BODY></HTML>" );
out.flush();
rs.close();
stmt.close();
}
```

Aquí se llama al método *getString()* de cada columna de cada fila, donde cada fila corresponde a un artículo. La figura 19.12 muestra el resultado de la ejecución de una consulta de este tipo.

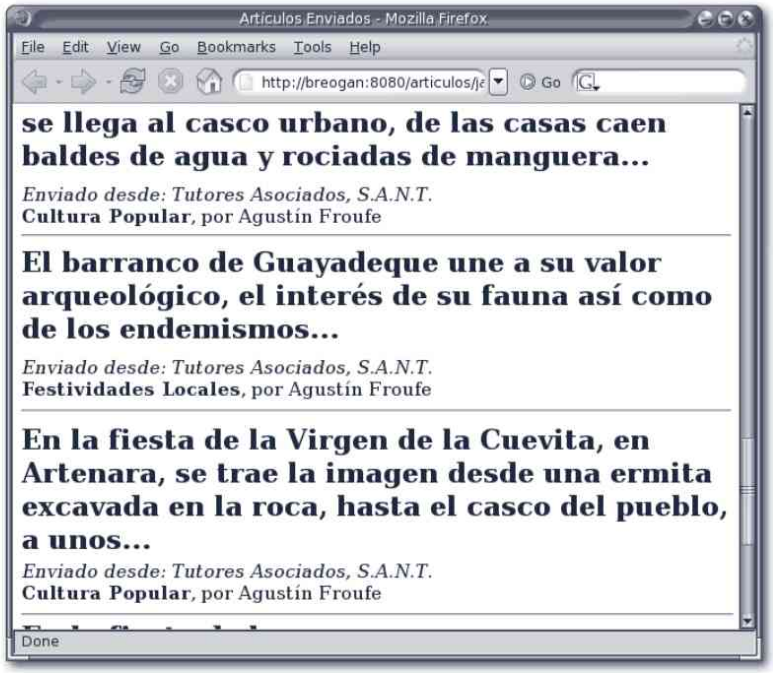


Figura 19.12

Otra parte interesante del código es la que se encarga del envío de los artículos y su inserción en la base de datos. Esto se realiza en las líneas que se muestran seguidamente. Las cuestiones de autorización se encargan a otro método, así que no hay por qué considerarlas en éste.

```
consulta = "INSERT INTO articulos VALUES( '";
consulta += req.getParameter( "titulo" )+"','"+usuario+"','";
consulta += req.getParameter("cuerpo")+"')";
int result = stmt.executeUpdate( consulta );
if( result != 0 ) {
  out.println( "Tu artículo ha sido aceptado e insertado "+
    "correctamente." );
  }
else {
  out.println( "Se ha producido un error en la aceptación de "+
    "tu artículo.<BR>" );
  out.println( "Contacta con el Administrador de la base de "+
    "datos, o consulta<BR>" );
  out.println( "el fichero <I>log</I> del servlet." );
```

```
  }
out.println( "<INPUT TYPE=button name=Submit value=' Atrás ' ");
out.println( "onClick='history.go(-1);'> " );
out.println( "</BODY></HTML>" );
out.flush();
stmt.close();
```

Con esto, se ha presentado al lector un ejemplo en el que se accede a la base de datos desde el servidor, y la parte cliente se limita a utilizar los recursos del servidor Web para acceder a la información de esa base de datos. El ejemplo es específico para servlets HTTP, aunque se pueden escribir servlets que devuelvan tipos binarios en lugar de una página *html*; por ejemplo, se puede fijar el tipo de contenido a '*image/jpg*' y utilizar el controlador de **OutputStream** para escribir una imagen *jpg* que se construya en el mismo momento al navegador. De este modo se pueden pasar imágenes que están almacenadas en la base de datos del servidor a la parte cliente, en este caso, el navegador.

JAVADB

Sun Microsystems, junto con el JDK correspondiente a J2SE 6, distribuye la base de datos **JavaDB**, desarrollada completamente en Java, que es una derivación del proyecto *opensource* **Derby** de la comunidad *Apache*.

La base de datos JavaDB es un desarrollo de poco más de 2 megabytes y puede ser incluida en las aplicaciones Java sin apenas dificultad. De este modo, las aplicaciones Java pueden acceder a un potente motor de base de datos, incluyendo incluso *triggers*, procedimientos almacenados y soporte SQL.

En esta sección se describe la instalación de JavaDB y se implementa una aplicación de *agenda personal*, para mostrar al lector cómo se trabaja utilizando JavaDB cómo motor de base de datos y cómo se integra JavaDB en aplicaciones Java.

La aplicación que se desarrolla consiste pues, en una simple **Agenda** para guardar información acerca de personas. Solamente se almacenará el nombre, número de teléfono, dirección de correo electrónico y país. Permitirá las acciones correspondientes a la inserción de datos de una nueva persona, la edición de los datos de personas existentes y su eliminación. La interfaz gráfica que presentará es la que muestra la figura 19.13.

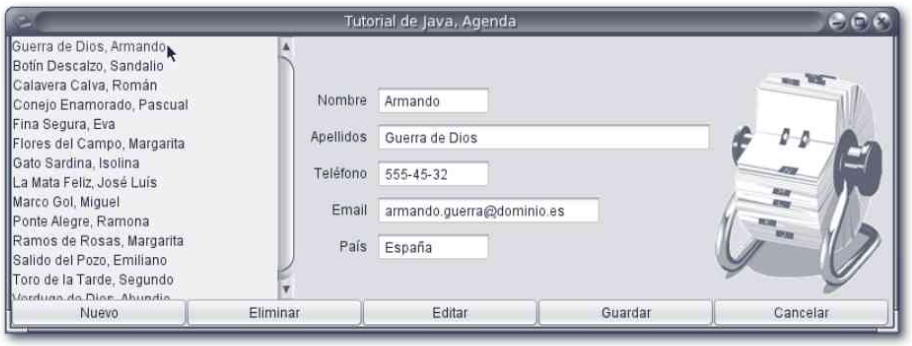


Figura 19.13

La aplicación consiste en un objeto de tipo **JFrame** principal que contiene a todos los componentes gráficos y actúa como controlador de todos los eventos generados por dichos componentes. Estos componentes son subclases de **JPanel** y cada cual se encarga de una acción específica. En el código de las clases que implementan los componentes, se indica la responsabilidad asignada a cada uno de ellos.

INSTALACIÓN

Para instalar la base de datos basta con la ejecución de los binarios que acompañan a la distribución del JDK.

La puesta en marcha de la base de datos consiste en incluir el fichero **derby.jar** como parte del `CLASSPATH` de la aplicación y ya la base de datos quedará accesible a la aplicación. La forma más sencilla es colocar el fichero *derby.jar* en el directorio *jre/lib/ext* de instalación del JDK. El fichero **derby.jar** se encuentra en el directorio *lib* del árbol de instalación de JavaDB.

DRIVER

El manejo de la base de datos se realiza a través del driver JDBC correspondiente, que se encuentra en el fichero **derby.jar** y su nombre es *EmbeddedDriver*, que ha de pasarse como parámetro al método *forName()* de la clase **Class** de la forma:

```
Class.forName( "org.apache.derby.jdbc.EmbeddedDriver" );
```

En la aplicación Agenda que se implementa en esta sección, el nombre del driver se recupera de una propiedad de configuración y se pasa al método *cargarDriver()*. Toda la funcionalidad de la Agenda se encapsula en un objeto que sigue el patrón **DAO** (*Data Access Object*).

El driver es el encargado de proporcionar las conexiones a la base de datos. La URL de conexión en el caso de JavaDB, siempre tiene la forma:

```
jdbc:derby:<base_de_datos>[propiedades]
```

En donde `base_de_datos` identifica a la base de datos que se va a utilizar. Esta base de datos puede estar localizada en diferentes sitios: un directorio local, un directorio en el `CLASSPATH`, un fichero *JAR*, un directorio absoluto o un directorio raíz para todas las bases de datos JavaDB. Esta última localización es la más utilizada y se puede especificar a través de la propiedad `derby.system.home`.

En este caso, la base de datos se llama *AgendaDB* y se localiza donde se indique en la propiedad anterior que se fija al subdirectorio *.agenda*, en el directorio local del usuario que ejecuta la aplicación. Se acude a la clase **System** para conocer el directorio local del usuario y luego se fija la propiedad `derby.system.home`, tal como muestra el código siguiente:

```
private void setDirSistemaBD() {
  // Recuperamos el directorio del perfil del usuario
  String dirUsuario = System.getProperty("user.home", ".");
  // Fijamos la propiedad de la base de datos en que indicamos dónde está
  // su directorio raíz
  String dirAgenda = dirUsuario + "/.agenda";
  System.setProperty( "derby.system.home",dirAgenda );
  // y lo creamos
  File dirSistemaBD = new File( dirAgenda );
  dirSistemaBD.mkdir();
   }
```

La parte `propiedades` de la URL consiste en una serie de argumentos que se pueden pasar al sistema de base de datos. Se pueden indicar como tales argumentos, separando cada propiedad por un punto y coma (`;`) o también se puede pasar como un objeto de tipo **Properties**. Las propiedades que se pueden indicar son:

```
create = true
databaseName = nomreBaseDatos
user = usuario
password = claveDelUsuario
shutdown = true
```

CREAR BASE DE DATOS

La aplicación **Agenda** no dispone de una base de datos preinstalada, sino que la propia aplicación la crea cuando se ejecuta por primera vez.

La ventaja de las bases de datos *embebidas* es que todo el control se puede ejercer desde la aplicación, sin contar con el usuario, de modo que se puede controlar que la base de datos exista, qué tablas se van a crear y qué permisos se van a conceder. En este caso, la base de datos se crea pasando el argumento `create=true` en la conexión, de la forma:

```
dbProp.put( "derby.driver","org.apache.derby.jdbc.EmbeddedDriver" );
String driver = dbProp.getProperty( "derby.driver" );
try {
  Class.forName( driver );
} catch( ClassNotFoundException e ) {
```

```
      e.printStackTrace();
      }
```

Una vez que la base de datos está creada, la aplicación ya puede crear las tablas que necesita. EL código siguiente SQL es el que crea la tabla AGENDA.

```
create table APP.AGENDA (
  ID       INTEGER NOT NULL PRIMARY KEY GENERATED ALWAYS
    AS IDENTITY (START WITH 1, INCREMENT BY 1),
  NOMBRE    VARCHAR(30),
  APELLIDOS VARCHAR(50),
  TELEFONO  VARCHAR(20),
  EMAIL     VARCHAR(50),
  PAIS      VARCHAR(30) )
```

Cada registro estará identificado por un ID, que será generado por JavaDB cuando se cree y se utilizará como clave primaria. Para ejecutar el código y crear la tabla, se utiliza el mismo objeto **dbCon** que representa la conexión a la base de datos. Se crea un objeto **Statement** al que se pasa un objeto **String** conteniendo el código SQL anterior y se ejecuta.

```
  private boolean creaBaseDeDatos() {
    boolean creada = false;
    Connection conn = null;
    Statement stm = null;
    String dbUrl = getUrlDB();
    dbProp.put( "create","true" );
    try {
      conn = DriverManager.getConnection( dbUrl,dbProp );
      stm = conn.createStatement();
      // Ejecutamos la sentencia SQL que crea las tablas
      stm.execute( strCrearTablasAgenda );
      creada = true;
    } catch( SQLException e ) {
      }
    dbProp.remove( "create" );
    return creada;
    }
```

En este instante ya existirá la base de datos en el directorio reservado a ella. Si los ficheros contenidos en este directorio se manipulan directamente, se puede destruir la integridad de la base de datos.

USAR BASE DE DATOS

Una vez que se dispone ya de la base de datos y las tablas necesarias, la aplicación creará nuevas conexiones y sentencias para ejecutar las tareas de gestión que corresponden a la creación de nuevos registros o entradas en la agenda, la eliminación de entradas ya grabadas, la edición de entradas, el almacenamiento de entradas nuevas o modificadas y la cancelación de ediciones de registros o de la creación de nuevos registros.

En la interfaz gráfica de la aplicación, todas las acciones anteriores se pueden invocar desde los botones situados en la parte inferior de la ventana. El lector puede revisar el código fuente de la aplicación, en donde se implementa cada una de estas acciones y se comentan las sentencias que las componen y que se invocan para llevar a cabo las funciones correspondientes a dichas acciones.

DISTRIBUCIÓN

Una vez que la aplicación está implementada y funcionando, es necesario distribuirla a los usuarios, para que la desplieguen en sus propias máquinas.

Java permite el uso de distintas estrategias de distribución y despliegue de aplicaciones; ya sea mediante *Java WebStartt*, el uso de *applets* o ficheros *JAR*. Esta última opción es la que se va a comentar.

En este caso, pues, para distribuir la aplicación será necesario proporcionar la propia aplicación en un fichero *JAR*, **agenda.jar**, e incluir los ficheros **derby.jar** que corresponde a la base de datos y **nimbus.jar** que corresponde al *Look&Feel* de la aplicación, que se colocarán en el directorio *lib*.

En muchas ocasiones, cuando se incluyen librerías de terceros en las aplicaciones Java, como en este caso la librería correspondiente a la base de datos, se proporcionan *scripts* externos para la ejecución de la aplicación. Estos *scripts* colocan los ficheros en el CLASSPATH y luego ejecutan la aplicación. Este método es un poco engorroso, porque requiere la creación de múltiples *scripts*, típicamente uno para cada plataforma o sistema operativo sobre el que se pretenda ejecutar la aplicación, por ejemplo **run.bat** para *Windows*, **run.sh** para *Linux* o *Solaris*, etc.

La aplicación **Agenda** va a incluir la información del CLASSPATH en el fichero **MANIFEST** que se incluye en **agenda.jar**, así la aplicación se podrá ejecutar directamente desde la línea de comandos pasando **agenda.jar** como argumento o haciendo doble clic sobre ella, en caso de que se ejecute sobre un entorno gráfico.

En el contenido del fichero **manifest.mf**, se puede indicar la clase principal de la aplicación que debe ser ejecutada y los ficheros adicionales que forman parte del CLASSPATH de la aplicación.

```
Manifest-Version: 1.0
Main-Class: agenda.Java1907
Class-Path: lib/derby.jar lib/nimbus.jar
```

Una vez que se ha creado el fichero **MANIFEST** y se ha incluido en la distribución, se puede generar un fichero comprimido de tipo *ZIP*, que será el que se proporcione como distribución de la aplicación a los usuarios finales. Estos usuarios solamente tendrán que descomprimir el fichero y ejecutar **agenda.jar**, que a través del contenido del fichero **manifest.mf** indicará al sistema los ficheros que debe colocar en el CLASSPATH y cual es la clase principal de la aplicación. Evidentemente, como se

utiliza JavaDB, en ese CLASSPATH debe figurar *lib/derby.jar* para que la agenda pueda funcionar como se ha descrito.

CÓDIGO INDEPENDIENTE Y PORTABLE

Tenga el lector presente que lo que a continuación se refleja son simples opiniones y sugerencias, que solamente están destinadas a que los problemas que se presenten sean los menos posibles cuando se intente programar con Java y, en este caso, con JDBC.

Uno de los objetivos fundamentales a la hora de diseñar JDBC fue obtener la máxima portabilidad posible entre distintos Sistemas de Gestión de Bases de Datos. Distintos gestores tienden a utilizar distinta sintaxis para las mismas cosas, como por ejemplo para especificar una fecha, ejecutar un procedimiento almacenado, etc. JDBC proporciona una serie de cláusulas de escape de modo que se pueda escribir una fecha, etc., de forma portable; posteriormente, será el driver el que se encargue de convertir la información al formato que requiere la base de datos. Todas las cláusulas de escape se escriben siempre entre llaves, "{...}".

Una fecha se especifica utilizando el formato {d 'aaaa-mm-dd'}, donde d indica que se trata de una fecha, y aaaa serán los cuatro dígitos correspondientes a un año, mm los dos dígitos del mes y dd los dos dígitos del día. Una hora se escribirá utilizando el formato {t 'hh:mm:ss'}, donde se usa hh para la hora, mm para los minutos y ss para los segundos. Para un valor de fecha/hora se utilizará {ts 'aaaa-mm-dd hh:mm:ss.f . . .'}, donde f, la parte fraccionaria de los segundos, es optativa.

No sólo se utilizan cláusulas de escape para las constantes de fecha y hora, también para llamar a un procedimiento almacenado hay una sintaxis especial que permite aislarse de la sintaxis concreta de cada base de datos: el formato será {call nombre_proc[(?, ?,...)]}, donde call indica que se está invocando un procedimiento almacenado, nombre_proc será su nombre y los distintos parámetros, si los hay, se indicarán mediante una serie de caracteres de interrogación, ?, encerrados entre paréntesis.

También para invocar una función SQL existe una cláusula de escape, con la sintaxis {fn upper("Texto")}, donde **fn** indica que estamos llamando a una función, de nombre upper y a la que se le pasa el argumento "Texto".

También hay cláusulas de escape para las composiciones *externas (outer-joins)*, utilizando la sintaxis {oj outer-join}, donde outer-join tiene la forma :

```
tabla LEFT OUTER JOIN {tabla | outer-join} ON condición_de_búsqueda
```

A la hora de utilizar dentro de un SELECT la palabra reservada LIKE se pueden usar dos caracteres como comodín, (_) y (%). Para buscar en la base de datos un texto en que aparezcan estos dos caracteres, basta con poner delante de ellos un carácter

especial que indique que en ese lugar se utilizan como cualquier otro carácter, no como comodines. Para ello, se debe indicar cuál es dicho carácter especial utilizando la sintaxis `{escape 'carácter'}`, por ejemplo:

```
SELECT nombre FROM Empleados WHERE nombre LIKE '@_' {escape '@'}
```

En este ejemplo, se indica que el carácter especial de escape es `@`, por lo que la cadena `LIKE '@_'` indica que se deben buscar todos los nombres que comiencen por '_', en lugar de usarlo como comodín.

Si se quiere escribir código JDBC para que sea portable e independiente de la máquina y motor de base de datos que se va a atacar, deberían, además de tenerse en cuenta las recomendaciones anteriores, seguirse unas sencillas reglas a la hora de escribir código. Algunas de ellas son las que se enumeran en los siguientes párrafos.

- Presérvese siempre el uso de mayúsculas y minúsculas de la salida de los métodos *getTables()* y *getColumns()*.
- Llamar al método *getIdentifierQuoteString()* y utilizar la cadena que devuelva para delimitar los identificadores.
- Tener en cuenta que no todos los sistemas soportan SQL '92 estrictamente.
- Utilícese *setNull()*

```
PreparedStatement stmt = con.prepareStatement(
  "update nombre set departamento=? where empleado=00012" );
stmt.setString( 1,"" ); // Erróneo!!
stmt.setNull( 1 );
stmt.executeUpdate();
```

- Comprobar los límites de la sesión invocando a los métodos *getMaxStatements() y getMaxConnections()*. Por ejemplo, Microsoft SQL restringe a una sola sentencia activa y hay algunos drivers que limitan las conexiones a una sola.
- Utilizar las comillas... pero no en todas partes.
- Las pseudo-columnas nunca deben ir entre comillas. Por ejemplo, *filaid*, *regid*, *usuario*, *empleado*.
- Tener cuidado con algunas características de SQL'92:

```
"select * from Empleados where empleado = NULL"
// nunca devuelve filas!
"select * from Empleados where empleado is NULL"
// puede devolver filas
"select * from Empleados where empleado = ?"
// usar setNull para el parámetro
// nunca devolverá filas
```

- Tener cuidado con los Nombres, si *supportSchemasInDataManipulation()* indica que sí se pueden soportar esquemas, utilizar siempre un punto (`.`) entre el nombre del esquema y el nombre de la tabla. ¡No olvidar las comillas!
- A la hora de crear una tabla usar *getTypeInfo()* y buscar entre el resultado las mejores opciones. Por ejemplo, si se quiere crear una columna `DECIMAL(8,0)`,

buscar un `SQL DECIMAL` o `SQL NUMERIC`, luego `SQL INTEGER`, luego `SQL DOUBLE` o `SQL FLOAT`, luego `SQL CHAR` o `SQL VARCHAR`; e ignorar tipos que no coincidan con los requisitos: moneda, autoincremento, etc.

- Utilizar parámetros en lugar de constantes en las sentencias SQL.
- Algunas bases de datos no soportan constantes cadena para columnas `LONG`.
- Usar *setObject()*, que reduce la complejidad del programa. Por ejemplo, si se usan buffers de cadenas, entonces es necesario *setObject()*.
- Para parámetros de salida, no olvidar utilizar *registerOutParameter()*.

CAPÍTULO 20

RMI

La Invocación de Métodos Remotos (**RMI**, *Remote Method Invocation*) de Java, permite que un objeto que se ejecuta bajo el control de una Máquina Virtual Java pueda invocar métodos de un objeto que se encuentra en ejecución bajo el control de otra Máquina Virtual Java diferente. Estas dos Máquinas Virtuales pueden estar ejecutándose como dos procesos independientes en el mismo ordenador o, lo que es más interesante, estar lanzadas en ordenadores distintos conectados a través de una red TCP/IP. Es decir, que como Internet es en último término una red TCP/IP, lo anterior se puede traducir en que una máquina cliente en cualquier rincón del mundo es capaz de invocar métodos de un objeto que se encuentre en ejecución sobre un servidor en cualquier otro rincón del mundo. La potencia que esta afirmación puede tener seguro el lector, que sobrepasa con creces su imaginación.



Figura 20.1

La máquina que contiene el objeto cuyos métodos se pueden invocar se llama *servidor*, y la máquina que invoca métodos sobre el objeto remoto recibe el nombre de *cliente*. En el cliente siempre debe haber una línea de código para recuperar la referencia al objeto remoto. Una vez que el cliente consigue esa referencia, la invocación de métodos sobre el objeto remoto no difiere en absoluto de la llamada a cualquier método de un objeto local (sin tener en cuenta la velocidad, por supuesto).

Tal como el lector ya supondrá, el código del servidor debe definir la clase e instanciar un objeto remoto de esa clase. Además de eso, solamente se necesitan unas

cuantas líneas más de código para registrar el objeto y dejar accesibles los métodos a los clientes, para que éstos puedan invocarlos remotamente.

Tanto el cliente como el servidor deben definir, o tener acceso, a una interfaz común, en donde se declaren los métodos que pueden ser invocados remotamente, y también el controlador de seguridad que va a tener a su cargo el control de que tanto servidor como cliente tengan los niveles de seguridad adecuados a las acciones que quieren realizar.

Cuando se invocan métodos sobre objetos remotos, el cliente puede pasar objetos como parámetros y los métodos de los objetos remotos pueden devolver objetos. Esto es posible gracias a la capacidad de *serialización* de objetos de Java. Por otro lado, como cliente y servidor están escritos en Java, el único requisito en cuanto a las plataformas en que se ejecutan es que en ambas se ejecuten Máquinas Virtuales Java compatibles.

HOLAMUNDO REMOTO

En primer lugar se va a introducir al lector en la utilización de RMI, para después explicar cómo funciona. Al igual que siempre, hay que recurrir al manido "Hola Mundo!", como la mínima aplicación que se despacha.

La implementación de esta pequeña aplicación requiere la creación de cuatro ficheros de código fuente y la ejecución de dos utilidades. Con ellos se consigue un mínimo de seis ficheros `.class` que deben ser instalados entre cliente y servidor, y algunos de ellos en ambos. Al final, habrá instalados tres ficheros `.class` en el cliente y cinco en el servidor. Por supuesto, esto es lo mínimo de lo mínimo, la cantidad de ficheros dependerá de la complejidad de la aplicación y de las clases que se usen.

La única diferencia con el uso local de la aplicación es la compilación en dos pasos diferentes. En el primer paso se utiliza el compilador Java para compilar el código fuente del servidor y del cliente, conteniendo las interfaces remotas y sus implementaciones. La ejecución del compilador *rmic* sobre el objeto remoto produce un fichero especial que se conoce como *stub*. Este fichero *stub* debe ser instalado en el cliente y es una representación del objeto remoto, comportándose como el método en un lado y como un programa de comunicaciones en el otro.

Lo habitual en aplicaciones reales que utilizan RMI es colocar todos los ficheros en paquetes; sin embargo, en este caso concreto sólo contribuiría a complicar la explicación del funcionamiento. En toda la secuencia de eventos que se produce en la ejecución de los programas de ejemplo se utiliza la misma máquina, de forma que cliente y servidor se ejecutan sobre una única plataforma, en procesos separados. De este modo, cada proceso, cliente y servidor, es ejecutado por una instancia distinta de la Máquina Virtual Java. Sin algo como RMI, el programa que se ejecuta en uno de

estos procesos no podría ser capaz de tener comunicación útil con el otro programa, y mucho menos si estuviesen corriendo en máquinas distintas.

El programa cliente ejecuta un método llamado *objRemotoHola()* sobre un objeto ejecutándose en el proceso servidor, pasándole la cadena de texto "Mundo". El método remoto recibe la cadena de texto que completa anteponiéndole el saludo y se la devuelve al programa cliente que la presentará en pantalla. En el programa de lotes, el programa cliente se ejecuta dos veces para demostrar al lector que el objeto remoto y su método continúan disponibles incluso después de que el programa cliente haya concluido su ejecución.

La batería de ficheros necesaria para la compilación y ejecución del ejemplo es la siguiente:

- `HolaMundoRmiI`, la interfaz
- `HolaMundoRmiO.java`, el objeto
- `HolaMundoRmiS.java`, el servidor
- `HolaMundoRmiC.java`, el cliente
- `HolaMundoRmi.bat`, el proceso de lotes para *Windows*
- `java.politicas`, el fichero de políticas de seguridad

Antes de explicar ninguna otra cosa, es necesario comentar la presencia de este último fichero, `java.politicas`. Debido a que los requisitos de seguridad en la plataforma Java 2 son más exigentes y estrictos que en anteriores versiones del JDK, es necesaria la utilización de un fichero de normas o políticas de seguridad que garantice los permisos necesarios para acceder a las operaciones que se realizan utilizando *sockets*. Sin el concurso de este fichero, no se podrá establecer la comunicación entre el programa cliente y el proceso servidor.

El otro fichero que tiene poco que ver con RMI es el fichero de proceso por lotes, `HolaMundoRmi.bat`. Se incluye para facilitar al lector la compilación y ejecución del ejemplo. En ejemplos posteriores, también se proporcionará al lector un fichero de características semejantes. En el fichero primero se encuentra la compilación del código fuente correspondiente al servidor y al cliente. Luego se ejecuta el programa de utilidad `rmic.exe` sobre el fichero que contiene la implementación del objeto remoto para generar el fichero `_Stub` adicional.

Hay otro programa de utilidad, `rmiregistry`, que se ejecuta para registrar los objetos sobre el servidor, proporcionándole la capacidad de crear y mantener un registro de objetos cuyos métodos pueden ser invocados remotamente. Antes de arrancar esta utilidad, hay que asegurarse de que la variable de entorno `CLASSPATH` actual no está apuntando al directorio que contiene las clases de los objetos remotos, incluyendo el fichero `stub`. Si la utilidad encuentra estas clases al arrancar, no las cargará cuando se ejecute el proceso servidor de los ejemplos, lo cual generará muchos problemas cuando los clientes intenten descargar las clases desde el servidor remoto.

Los siguientes comandos liberan la variable de entorno CLASSPATH y arrancan el registro RMI sobre el puerto 1099, que es el utilizado por defecto, aunque se puede indicar cualquier otro añadiéndolo al comando; aunque si el lector lo hace así, deberá tener en cuenta que ha de cambiar el código de los ejemplos para que se comuniquen a través de ese nuevo puerto.

Desde una ventana MS-DOS se invocaría como:

```
unset CLASSPATH
start rmiresgitry
```

y desde una plataforma Unix, usando *bash* como shell, se invocaría como:

```
unset CLASSPATH
rmiregistry &
```

No se olvide el lector de devolver el valor original a la variable de entorno CLASSPATH en caso de que estuviese fijada.

Otra forma de crear el registro de objetos remotos es desde el propio código del programa, aunque esto no es aconsejable debido a la posibilidad de que dos o más programas intenten crear múltiples registros. No obstante, ésta es una función del administrador, que debería ejecutarse solamente una vez.

Volviendo al proceso por lotes, el siguiente paso después del registro de objetos es el lanzamiento del servidor, que intenta ser automático y arrancarse en una ventana diferente. Sin embargo, si el sistema del lector es diferente o no permite este tipo de lanzamiento, deberá abrir una ventana distinta y arrancar manualmente el proceso servidor en esa ventana. Y esto mismo deberá hacer con el cliente, para que se ejecute en su propia ventana.

Aquí el proceso se vuelve semiautomático, porque al abrirse una ventana nueva, ésta gana el foco, así que cuando aparezca el mensaje de que el objeto remoto ya está listo, hay que pulsar con el ratón sobre la ventana original y pulsar una tecla para que se ejecute la última parte del proceso por lotes. Esta última parte consiste en el lanzamiento por dos veces del programa cliente, para demostrar que el objeto remoto sigue vivo y disponible incluso después de que un cliente lo haya utilizado.

El fichero Interface

Éste es un fichero minúsculo, pero es el fichero clave de todo el sistema, su contenido se reproduce a continuación.

```
public interface HolaMundoRmiI extends Remote {
  String objRemotoHola( String cliente ) throws RemoteException;
  }
```

En la definición de esta interfaz se extiende la interfaz **Remote**, que sirve para identificar a todos los objetos remotos. Cualquier objeto remoto ha de implementar esta interfaz, directa o indirectamente. Solamente aquellos métodos especificados en una interfaz remota estarán disponibles para ser llamados remotamente. En las clases de implementación se puede implementar cualquier número de interfaces remotas y se pueden extender otras clases de implementación.

La interfaz **Remote** no declara método alguno, pero es imprescindible para que un objeto pueda ser remoto. En este ejemplo el objeto remoto que se va a crear implementará la interfaz **HolaMundoRmiI**, y por tanto, implementará indirectamente la interfaz **Remote** por herencia.

Si la clase desde la cual se instancia el objeto remoto define métodos que no están declarados en la interfaz que extiende a **Remote**, entonces esos métodos no podrán ser invocados remotamente. En este caso solamente se declara el método *objRemotoHola()*.

La declaración del método indica que puede lanzar excepciones de tipo **RemoteException**. Esto es un requisito. Esta clase de excepción es la superclase de otros tipos de excepciones que tienen que ver con problemas RMI.

El fichero Objeto Remoto

En este fichero se encuentra la implementación de la interfaz **HolaMundoRmiI**, implementando indirectamente la interfaz **Remote**, a través de la herencia, para satisfacer uno de los requisitos necesarios para que un objeto sea tratado como remoto y se puedan invocar sus métodos remotamente.

Otro requisito es que extienda **UnicastRemoteObject**, que extiende a su vez la clase **RemoteServer**. La declaración de la clase cumple estos requisitos.

```
public class HolaMundoRmiO extends UnicastRemoteObject
  implements HolaMundoRmiI {
```

RemoteServer proporciona un mecanismo general de comunicación para los procesos RMI. Soporta a **UnicastRemoteObject** que proporciona un mecanismo de comunicación *punto-a-punto* de referencias de objetos activos a través de canales TCP. Es decir, extendiendo esta clase, el objeto instanciado hereda la posibilidad de constituirse en un punto de comunicación con el propósito de hacer sus métodos alcanzables desde otros lugares.

Lo que implica el párrafo anterior es que hay que estar conectado a una red TCP/IP para que todo funcione correctamente, incluso aunque se ejecuten servidor y clientes como distintos procesos sobre la misma máquina.

El siguiente fragmento de código muestra el constructor del objeto remoto, que también debe poder lanzar excepciones de tipo **RemoteException**. La llamada al método *super()* es por poner algo; no es necesaria, ya que cuando se invoca a un constructor sin argumentos esa llamada se produce por defecto.

```
public HolaMundoRmiO() throws RemoteException {
  super();
  }
```

Y lo que resta del fichero es la definición del método que se declaraba en la interfaz, y que será la que se pueda invocar desde distintos clientes. El método solamente completa el mensaje de saludo y lo devuelve.

```
public String objRemotoHola( String cliente )
  throws RemoteException {
  return( "Hola "+cliente );
  }
```

El cliente y este método son transportados a través de un canal de comunicaciones de dos vías, a través del cual se pasan objetos en ambas direcciones. Desde luego, un objeto **String** es muy sencillo y el movimiento de cadenas de un ordenador a otro no comporta grandes cambios. Sin embargo, incluso este programa tan simple permitirá al lector atisbar el poder de RMI, ya que los métodos remotos pueden, en general, recibir y devolver objetos de cualquier tipo por muy complejos que sean.

Esto se consigue a través de la *serialización*. La serialización de objetos se utiliza para descomponer un objeto en un array de bytes consistente que pueda ser transportado a través de una red IP y ser reconstruido en el otro lado partiendo de ese array de bytes. Esto no es empresa trivial, ya que puede haber objetos que contengan objetos, que a su vez contengan objetos, que también contengan objetos, etc., etc.

El fichero Servidor

El siguiente fichero que interviene en esta discusión es el que contiene el código para implementar el servidor. El primer fragmento de código instala el controlador de seguridad e intenta instanciar el objeto remoto dentro de un bloque `try-catch` que captura la excepción **RemoteException**, que es una excepción de comprobación que puede lanzar el objeto remoto y debe ser capturada para que el programa compile.

```
try {
  // Se instala el controlador de seguridad
  if( System.getSecurityManager() == null ) {
    System.setSecurityManager( new RMISecurityManager() );
    }
  HolaMundoRmiO objRemoto = new HolaMundoRmiO();
```

El controlador de seguridad protege contra el acceso a los recursos del sistema de código no fiable que se ejecute dentro de la Máquina Virtual Java. Todos los programas que utilicen RMI deben instalar un controlador de seguridad, o RMI no

descargará las clases (fuera de las clases locales) para objetos recibidos como parámetros, valores de retorno o excepciones en llamadas a métodos remotos. Esta restricción asegura que las operaciones que realice el código descargado pasan todos los controles de seguridad.

La siguiente sentencia en el método *main()* del servidor, es la que realmente coloca el objeto remoto en el registro y lo deja visible al cliente para que éste pueda invocar sus métodos.

```
Naming.rebind( "ObjetoHola",objRemoto );
```

El método *rebind()* es un método estático de la clase **Naming**. El mecanismo por el cual se obtienen las referencias a objetos remotos sigue la sintaxis de las direcciones URL, en donde el protocolo es ***rmi*** y el formato de la referencia es:

```
rmi://servidor:puerto/nombre
```

En donde, *rmi* es el protocolo, *servidor* es el nombre o dirección IP de la máquina que ha registrado el objeto remoto (por defecto la propia máquina), *puerto* es el puerto de comunicaciones del protocolo *rmi* (por defecto el 1099) y *nombre* es el identificador asignado al objeto remoto.

En la sentencia *rebind()* anterior, el objeto referenciado por la variable `objRemoto` se registra con el nombre **ObjetoHola**. La clase **Naming** dispone de otros métodos, entre los cuales hay varios que permiten obtener información de los objetos que se han registrado y las direcciones de donde provienen. Por ejemplo, en el cliente se usará el método *lookup()* para obtener una referencia del objeto remoto.

El resto del código del servidor consiste en la presentación de un mensaje indicando que el objeto ha sido registrado y está listo para ser invocado y la captura de la excepción que se puede generar.

El fichero Cliente

El fichero cliente consiste solamente en el método *main()*, cuya primera sentencia es la asignación, por conveniencia, a un objeto **String** de la dirección URL del servidor, que luego se completará con el nombre del objeto a la hora de construir la dirección completa del objeto remoto. Ésta es una sentencia para economizar cuando hay varios objetos remotos, y en este caso es más ilustrativa que útil.

```
String direccion = "rmi://localhost/";
```

El código que sigue es ya la petición al servidor de una referencia al objeto remoto del cual se quieren invocar métodos. Para ello, se llama al método estático *lookup()* de la clase **Naming**, pasándole la dirección del objeto como parámetro. Por supuesto, si esto falla, se lanza una excepción, por lo que esta sentencia es imprescindible que se encuentre dentro de un bloque `try-catch` para poder compilar.

Si todo ha ido bien, y ya se tiene la referencia al objeto remoto, esta referencia se puede utilizar para invocar los métodos de ese objeto remoto. Pero antes de hacerlo, hay que tener en cuenta que el método *lookup()* devuelve objetos de tipo **Remote**, por lo que hay que moldear el objeto a la interfaz implementada en la clase del objeto. Esto significa que también hay que obtener esa información

```
try {
  HolaMundoRmiI hm =
    (HolaMundoRmiI)Naming.lookup( direccion+"ObjetoHola" );
  System.out.println( hm.objRemotoHola( "Mundo" ) );
```

Una vez que ya se tiene la referencia al objeto remoto correcto, la última línea del fragmento anterior invoca al método *objRemotoHola()* sobre ese objeto remoto, pasándole como parámetro una cadena para completar el mensaje de saludo, e imprime el resultado, la cadena de saludo completa.

El resto del código se centra en la captura y tratamiento de la excepción y la conclusión del programa.

RMI CON MÚLTIPLES OBJETOS DEL MISMO TIPO

En el ejemplo **Java2001x** se profundiza un poco más en el proceso RMI, utilizando en este caso dos objetos remotos del mismo tipo, que permiten el acceso a dos métodos, un saludo de bienvenida y un saludo de despedida. Los ficheros que intervienen en este ejemplo son, al igual que en el ejemplo inicial: el fichero de la interfaz, el correspondiente al objeto remoto, el servidor, el cliente y dos ficheros de proceso por lotes para *Windows* y *Linux*, que se encargan de compilar y ejecutar la aplicación.

El fichero Interface

Ahora la interfaz, `Java2001I.java`, declara dos métodos en lugar de uno como en el ejemplo anterior, por lo demás, nada ha cambiado. La expansión a múltiples objetos remotos (del mismo tipo) con varios métodos no varía en absoluto la estructura básica del software.

```
String hola( String cliente ) throws RemoteException;
String adios( String cliente ) throws RemoteException;
```

El fichero Objeto Remoto

El fichero que contiene el código fuente de la clase a partir de la cual se instancian objetos remotos, `Java2001O.java`, difiere del ejemplo anterior en que el constructor recibe un parámetro que guarda en una variable de instancia para uso posterior y en que la clase define dos métodos distintos.

El cambio en el constructor no tiene nada que ver con la expansión a dos métodos, se ha hecho para que sea más significativo y se vea más claramente que se están utilizando dos objetos diferentes, porque a pesar de tratarse de objetos del mismo tipo, su funcionalidad es distinta al ser su estado inicial diferente.

```
public Java2001O( String idObj ) throws RemoteException {
  // Almacenamos el identificador del objeto
  this.idObj = idObj;
  }
```

Por lo demás, la definición de la clase para los objetos remotos es esencialmente la misma que la del ejemplo inicial del capítulo, teniendo en cuenta que los métodos que implementan los objetos son dos.

```
public String hola( String cliente ) throws RemoteException {
  return( "Hola " +cliente+ " desde el " +idObj );
  }

// Método para el saludo de despedida
public String adios( String cliente ) throws RemoteException {
  return( "Adios " +cliente+ " desde el " +idObj );
  }
```

Cada método recibe una cadena como parámetro, y devuelve un objeto **String** generado a partir de la cadena de saludo preescrita, la cadena que se pasa como parámetro y la cadena que se ha guardado en el momento de instanciar el objeto.

El fichero Servidor

El código de este fichero, `Java2001S.java`, también difiere en dos aspectos respecto al correspondiente del ejemplo anterior. Instancia dos objetos remotos del mismo tipo pasándoles un objeto **String** diferente para que la inicialización sea distinta y se pueda identificar a cada objeto. Registra los dos objetos para que los clientes puedan acceder a sus métodos.

```
Java2001O objRemoto1 = new Java2001O( "Objeto Remoto 1" );
Java2001O objRemoto2 = new Java2001O( "Objeto Remoto 2" );
Naming.rebind( "ObjetoRemoto1",objRemoto1 );
Naming.rebind( "ObjetoRemoto2",objRemoto2 );
```

El fichero Cliente

En el cliente, `Java2001C.java`, se obtienen dos referencias a objetos remotos diferentes y se invoca a cada uno de los métodos de esos objetos.

```
Java2001I refObj1 =
  (Java2001I)Naming.lookup( direccion+"ObjetoRemoto1" );
Java2001I refObj2 =
  (Java2001I)Naming.lookup( direccion+"ObjetoRemoto2" );
System.out.println( refObj1.hola( "Agustin" ) );
System.out.println( refObj1.adios( "Agustin" ) );
```

```
System.out.println( refObj2.hola( "Agustin" ) );
System.out.println( refObj2.adios( "Agustin" ) );
```

De este modo se obtienen los mensajes de saludo y despedida de ambos objetos, comprobando que efectivamente RMI soporta múltiples métodos sobre múltiples objetos del mismo tipo.

RMI CON MÚLTIPLES OBJETOS DE DISTINTO TIPO

En el ejemplo **Java2002x** se actualizará el ejemplo anterior para mostrar el uso de RMI con dos objetos de diferente tipo, conteniendo cada uno de ellos dos métodos. El principal cambio con respecto a los ejemplos previos, es el número de ficheros que intervienen, que al implicar a objetos remotos de tipo diferente obliga a crear los ficheros interfaz y de clase distintos para cada uno de los tipos de objetos.

En los ficheros de lotes para *Windows* y *Linux*, puede observar el lector que hay que ejecutar la utilidad **rmic** sobre cada uno de los tipos de objetos, por lo que se incluye dos veces la llamada. También podrá notar la ausencia de la ejecución del otro programa de utilidad, **rmiregistry**; porque en este ejemplo se mostrará la forma de registrar el objeto desde el código del servidor.

Los ficheros Interface

Debido a que en esta aplicación se utilizan dos tipos de objetos diferentes, es necesario disponer de dos ficheros que declaren los métodos que van a poder ser invocados remotamente para cada uno de los tipos de objeto: `Java2002Ia.java` y `Java2002Ib.java`. Excepto por esta diferencia, y que las interfaces deben tener nombres diferentes, no hay ninguna otra diferencia significativa con los ficheros que declaraban una sola interfaz de los ejemplos anteriores.

Los ficheros Objetos Remotos

Por el hecho de que la aplicación utiliza dos objetos de tipos distintos, necesita la definición de una clase para poder instanciar cada uno de esos objetos: `Java20020a.java` y `Java20020b.java`. De nuevo, excepto por la razón de que son objetos distintos y tienen nombres diferentes, no hay ninguna diferencia significativa con las definiciones de clases simples utilizadas en los ejemplos anteriores.

El fichero Servidor

En este caso el fichero que corresponde al servidor, `Java2002S.java`, sí difiere del ejemplo anterior, ya que los objetos son de distinto tipo, donde antes eran del mismo tipo.

```
Java20020a objRemoto1 = new Java20020a( "Objeto Remoto A-1" );
Java20020b objRemoto2 = new Java20020b( "Objeto Remoto B-2" );
```

La siguiente línea de código en el programa ha sido añadida para mostrar el registro de objetos desde el programa.

```
LocateRegistry.createRegistry( 1099 );
```

El registro que se crea con esta sentencia es sobre el *host* local y sobre el puerto que se indica, en este caso el de defecto utilizado por RMI.

Por lo demás, tal como en el ejemplo anterior, se registran los dos objetos para que sus métodos puedan ser invocados remotamente.

```
Naming.rebind( "ObjetoRemoto1",objRemoto1 );
Naming.rebind( "ObjetoRemoto2",objRemoto2 );
```

El fichero Cliente

En este cliente, `Java2002C.java`, al igual que ocurría en el fichero cliente del ejemplo anterior, se obtienen dos referencias a objetos sobre el servidor, aunque ahora son de tipos distintos. Una vez que se consiguen las referencias a los objetos remotos, se invocan los métodos de saludo y despedida de cada uno de ellos.

```
Java2002Ia refObj1 =
  (Java2002Ia)Naming.lookup( direccion+"ObjetoRemoto1" );
Java2002Ib refObj2 =
  (Java2002Ib)Naming.lookup( direccion+"ObjetoRemoto2" );
System.out.println( refObj1.hola( "Agustin" ) );
System.out.println( refObj1.adios( "Agustin" ) );
System.out.println( refObj2.hola( "Agustin" ) );
System.out.println( refObj2.adios( "Agustin" ) );
```

Con esto el lector ha podido comprobar que una aplicación RMI soporta múltiples métodos sobre múltiples objetos de distinto tipo; lo cual puede extenderse a cualquier número de métodos sobre cualquier número de objetos remotos de cualquier tipo.

COMUNICACIÓN ENTRE OBJETOS

Para los propósitos de este capítulo, el lector debe tener claro que un objeto *remoto* es un objeto instanciado a partir de una clase que extiende la interfaz **Remote**, bien directa o indirectamente; y un objeto *normal* es un objeto de una clase que no extiende la interfaz **Remote**.

El autor recuerda que en sus tiempos de aprendizaje de los fundamentos de programación utilizando lenguajes como Pascal o C++, una gran parte del tiempo se consumía en la introducción de los conceptos de intercambio de parámetros por *valor* y por *referencia*. El paso de un parámetro por valor significa que la función o método recibe una copia del original, por lo que cualquier modificación en la copia no alterará el original. El paso por referencia significa que el método recibe una referencia, *puntero* en aquellos lenguajes, al original, de forma que cualquier cambio en el

parámetro estará alterando el original. Esto creaba situaciones de confusión, porque en Pascal o C++ cualquier parámetro se puede pasar por valor o por referencia. Y lo mismo ocurre con los valores de retorno de los métodos. La situación en Java es menos confusa, ya que todas las variables de tipo primitivo son pasadas por valor y todos los objetos se pasan por referencia.

Sin embargo, con RMI, se vuelven a complicar las cosas. Si un método que es invocado sobre un objeto remoto devuelve un objeto normal, se serializará una copia del objeto y se envía al cliente que invocó el método. Es decir, el objeto se devuelve por valor. Si el método en el cliente modifica el objeto, esta modificación no tendrá efecto alguno sobre el objeto original del servidor.

Por otro lado, si un método que es invocado sobre un objeto remoto devuelve un objeto remoto, no se devolverá una copia serializada del objeto, sino una referencia al objeto. Si esta referencia se utiliza para modificar el objeto, se estará modificando a la vez el objeto remoto original.

Por lo tanto, hay que ser cuidadoso con la invocación de métodos remotos bajo RMI, ya que pensar que se está modificando un objeto cuando en realidad se está modificando una copia del objeto, puede conducir a errores de bulto.

En los ficheros que componen el ejemplo **Java2003x** se intenta demostrar lo anterior. Se construye un objeto remoto que contiene dos variables de instancia. Una de estas variables es un objeto embebido, un objeto de una clase que implementa la interfaz **Remote**. Esta variable de instancia, siendo en sí misma un objeto, dispone de una variable de instancia de tipo `int` junto con un par de métodos para poder fijar y recuperar el valor de esa variable de instancia.

La otra variable de instancia del objeto remoto principal es un objeto de tipo **Date**. La clase **Date** no implementa la interfaz **Remote**, por lo tanto es una variable normal. Los ficheros que intervienen en el ejemplo son los mismos que en los ejemplos anteriores: interfaces, objetos, cliente, servidor y ficheros de proceso por lotes para la compilación y ejecución del ejemplo en *Windows* y *Linux*.

Los ficheros Interface

En esta aplicación, aunque se instancian dos objetos de diferentes tipos, solamente uno de ellos se va a exponer para que sus métodos puedan ser invocados desde clientes. No obstante, como son de tipos diferentes, se necesitan dos ficheros de interfaz distintos: `Java2003Ia.java` y `Java2003Ib.java`.

El siguiente fragmento de código corresponde a la definición de la interfaz para el objeto remoto principal.

```
public interface Java2003Ia extends Remote {
  Date getFecha() throws RemoteException;
```

```
Java2003Ib getObjeto() throws RemoteException;
}
```

El primer método declarado corresponde al que se encarga de obtener la fecha del objeto, que es una variable de instancia del objeto remoto principal. Este método devuelve un objeto de tipo **Date** y no hay ninguna sorpresa.

La declaración del método *getObjeto()* es un poco más compleja. En realidad, este método devuelve un objeto de tipo **Java2003Ob**; sin embargo, si se indica así, el programa no se ejecutará correctamente. Para que todo funcione como es debido, hay que declarar que el método devuelve **Java2003Ib**, que es el nombre de la interfaz implementada por la clase **Java2003Ob**.

La explicación de por qué esto es así es bastante compleja, pero se puede resumir en que la referencia que se pasa al cliente para representar al objeto solamente conoce a la interfaz **Remote**, y no sabe nada acerca del verdadero tipo del objeto.

El código siguiente muestra la interfaz **Remote** para el objeto que está embebido como variable de instancia en el objeto remoto principal. Éste es el objeto que devuelve el método *getObjeto()* nombrado antes.

```
public interface Java2003Ib extends Remote{
  void setDato( int dato ) throws RemoteException;
  int getDato() throws RemoteException;
  }
```

Tampoco hay sorpresas aquí. Se trata solamente de una interfaz que declara un par de métodos para fijar y recuperar el valor de una variable de instancia de la clase. Declarándolos bajo la tutela de **Remote**, se permite que sean invocados remotamente desde un cliente que pueda manejar este tipo de objetos. Todos los métodos declarados en la interfaz pueden ser invocados remotamente y todos aquellos que no estén declarados en la interfaz no podrán ser invocados desde ningún cliente.

Los ficheros Objetos Remotos

En este ejemplo hay dos tipos distintos de objetos remotos. Uno es el objeto remoto principal sobre el cual se invocan originalmente los métodos. Y el otro es el objeto que contiene una variable de instancia y que también es un objeto remoto.

Una vez que el cliente tiene acceso a la variable de instancia remota a través del método *getObjeto()* del objeto principal, está en disposición para invocar métodos sobre ambos objetos remotos, ya que los métodos están declarados en la interfaz **Remote**.

En el objeto principal se declaran dos variables de instancia: un objeto **Date** normal, porque la clase **Date** no implementa la interfaz **Remote,** y otra variable que sí es remota, porque la interfaz vista anteriormente extiende a **Remote**.

```
private Date fecha = new Date();
private Java2003Ob objRemoto;
```

Luego está el constructor y los métodos asociados a las variables de instancia. A la variable de instancia de tipo **Date** está asociado el método *getFecha()*, que cuando es invocado remotamente, serializa una copia del objeto y lo envía al cliente.

```
public Date getFecha() throws RemoteException {
  return( fecha );
  }
```

El método asociado a la otra variable es *getObjeto()*, que cuando es invocado devuelve una referencia al objeto. En este caso no se serializa nada, y el cliente tiene acceso al objeto original a través de la referencia que le pasa este método.

```
public Java2003Ib getObjeto() throws RemoteException {
  return( objRemoto );
  }
```

En el otro objeto, que está embebido en el objeto principal anterior, excepto por el hecho de que implementa la interfaz **Remote** indirectamente a través de la herencia, no hay nada destacable en la definición de la clase. Contiene una variable de instancia de tipo entero y un par de métodos para cambiarle el valor.

```
private int dato;
// Constructor del objeto
public Java2003Ob( int dato ) throws RemoteException{
  // Inicializamos el dato del objeto
  this.dato = dato;
  }
public void setDato( int dato ) throws RemoteException {
  this.dato = dato;
  }
public int getDato() throws RemoteException{
  return( dato );
  }
```

Lo único destacable es el hecho de que la clase implementa la interfaz **Remote**, por lo que los métodos declarados en esa interfaz pueden ser invocados por el cliente; es decir, que una vez que el cliente tenga la referencia a un objeto de este tipo, podrá obtener y cambiar el valor de la variable de instancia `dato`.

El fichero Servidor

En el código del servidor, `Java2003S.java`, la parte más importante no es la que se ve, sino la que no se ve. Por ejemplo, no se ven los dos objetos remotos que están siendo registrados para que los clientes puedan acceder a ellos; en su lugar, solamente se ve que el objeto principal será el que se expone al cliente almacenando su nombre en el registro.

```
Java2003Oa objRemoto = new Java2003Oa();
Naming.rebind( "ObjetoRemoto",objRemoto );
```

Sin embargo, el cliente puede invocar métodos de los dos objetos. A uno llega a través del acceso que le proporciona el registro y al otro a través del acceso que le proporciona el primer objeto. El segundo objeto es una variable de instancia del primero, que dispone de un método para dar acceso a esa variable, por lo tanto, una vez que el cliente consigue acceso al primer objeto puede invocar ese método para obtener una referencia al segundo objeto remoto, recibiendo una referencia a ese objeto remoto a través de la cual puede invocar sus métodos.

El fichero Cliente

El programa contiene varias salidas a pantalla para que se pueda realizar un seguimiento de las acciones que va realizando. Lo primero que se hace en el código del cliente, `Java2003C.java`, es ir al registro del servidor y obtener una referencia al objeto remoto principal.

```
Java2003Ia refObj =
  (Java2003Ia)Naming.lookup( direccion+"ObjetoRemoto" );
```

Esta referencia se utiliza para invocar al método *getFecha()* de ese objeto, que devuelve un objeto de tipo **Date**, que no es remoto (**Date** no implementa la interfaz **Remote**), por lo que se recibe una copia de ese objeto.

```
System.out.println( "Pedimos la fecha al servidor" );
Date fecha = refObj.getFecha();
System.out.println( "La fecha del Objeto Remoto es: "+fecha.toString() );
System.out.println( "Incrementamos la fecha 2 dias" );
Calendar cal = Calendar.getInstance();
cal.add( Calendar.DAY_OF_YEAR,2 );
fecha = cal.getTime();
System.out.println( "La fecha es ahora: "+fecha.toString() );
System.out.println( "Pedimos la fecha al servidor" );
fecha = refObj.getFecha();
System.out.println( "La fecha del Objeto Remoto es ahora: " +
  fecha.toString() );
```

En las sentencias anteriores está la secuencia de operaciones que se realiza con el objeto. Primero se imprime la fecha, luego se modifica, y se vuelve a imprimir, comprobando que efectivamente ha cambiado. Entonces se vuelve al objeto remoto para volver a imprimir su fecha y se observa que es la misma que antes de modificarla, es decir, que en el objeto remoto no se ha producido ningún cambio porque el cambio de fecha se ha realizado sobre la *copia* del objeto que se había obtenido al invocar al método remoto *getFecha()*.

Luego el programa realiza acciones similares con respecto al objeto remoto que es instancia del objeto remoto principal utilizado antes.

```
System.out.println( "Pedimos el objeto al servidor" );
java2003Ib obj = refObj.getObjeto();
System.out.println( "El dato del Objeto Remoto es: " +obj.getDato() );
System.out.println( "Fijamos el dato del objeto a 999" );
```

```
obj.setDato( 999 );
System.out.println( "El dato en el objeto es ahora: " + obj.getDato() );
System.out.println( "Pedimos el objeto al servidor" );
obj = refObj.getObjeto();
System.out.println( "El dato del Objeto Remoto es ahora: "+obj.getDato());
```

En este caso los resultados son diferentes. El código solicita el objeto y presenta en pantalla el dato que contiene. Luego cambia el valor del dato almacenado en el objeto y lo imprime de nuevo. Tal como era de prever, el valor ha cambiado. Luego se vuelve a obtener el objeto y se presenta el valor del dato. Esta vez, al contrario que con **Date**, los cambios se han hecho en la variable de instancia contenida en el objeto remoto (y no en ninguna copia), por lo que los cambios persisten. De hecho, en el fichero de lotes, el cliente se ejecuta una segunda vez, y se puede comprobar que los cambios realizados en la primera ejecución persisten, ya que el primer valor que se imprime en esta segunda ejecución es el 999, que se había fijado al ejecutar el cliente por vez primera.

RENDIMIENTO RMI

Cuando se desarrollan aplicaciones distribuidas, sea utilizando mecanismos RMI u otros, resulta de vital importancia minimizar las comunicaciones a la menor cantidad posible. Esto significa reducir el número de mensajes, agregando datos a otros. En otras palabras, es normalmente mejor enviar pocos mensajes grandes que muchos pequeños. Esto puede resultar difícil de conseguir cuando se utilizan objetos distribuidos, porque el modelo de programación orientada a objetos hace que los programas tiendan a ser no-distribuidos, con gran intercambio de mensajes entre ellos.

El ejemplo **Java2004*** muestra una de estas situaciones. Se trata de que un objeto local que contiene un componente **List**, que actualiza su contenido con los valores que le proporcione un objeto remoto. Un modo de hacerlo sería enviar cada elemento de la lista por separado, tal como el modelo de objetos sugiere; pero teniendo en cuenta las consideraciones anteriores, es mejor solución enviar todos los datos al cliente y dejar que éste actualice la lista.

A la hora de ejecutar el ejemplo, en pantalla aparecerá la ventana que se muestra a la izquierda en la figura 20.2, y una vez que se pulsa el botón, se establecerá comunicación con el objeto remoto, enviando éste el contenido de actualización de la lista, de forma que la nueva lista corresponderá a la mostrada en la ventana de la derecha.



Figura 20.2

La interfaz **Java2004I** es la que implementa el objeto remoto, y el servidor es una instancia de **Java2004S**, sobre el cual hay que ejecutar la utilidad `rmic`, antes de lanzarlo. En este caso, ya no se proporciona al lector un fichero de lotes para el lanzamiento del ejemplo, porque debe tener la suficiente soltura con RMI como para poder lanzar todo el proceso sin problema alguno.

El fichero Interface

En este ejemplo, en la interfaz `Java2004I.java` solamente se declara el método que devuelve los elementos de la lista con que se va a actualizar la lista local.

```
public interface Java2004I extends Remote {
  public String[] getDatos() throws RemoteException;
  }
```

El fichero Servidor

En esta clase, `Java2004S.java`, se implementa la interfaz, de forma que el método *getDatos()* devuelve el contenido de la lista de cadenas que se define en la primera sentencia:

```
String[] datos = { "Uno","Dos","Tres","Cuatro","Cinco" };
```

El método *main()* contiene las sentencias ya vistas de creación del objeto remoto y la activación de sus métodos para que puedan ser invocados desde los clientes. Todo englobado en un bloque `try-catch` para capturar posibles excepciones.

```
try {
  Java2004S lista = new Java2004S();
  Naming.rebind( "ListaRemota",lista );
  System.out.println( "Objeto remoto Lista preparado" );
} catch( Exception e ) {
  e.printStackTrace();
  return;
  }
```

El fichero Local

En este fichero, `Java2004L.java`, es donde se implementa la lista particular que se utiliza en el ejemplo. Es una clase derivada de la lista AWT, en la que se incluyen tres constructores y el método que permite cambiar el contenido de la lista, actualizando todos sus elementos.

```
// Primero borramos el contenido actual
removeAll();
// Y copiamos los elementos que se pasan como parámetro como
// nuevo contenido de la lista
for( int i=0; i < datos.length; i++ )
  add( datos[i] );
  }
```

En la acción de este método primero se elimina el contenido completo de la lista, que será sustituido por el array de cadenas que se le pasa como parámetro.

El fichero Cliente

Éste es el fichero, `Java2004C.java`, que genera la ventana en la que se inicializa la Lista con el contenido del array que se declara en la creación de la clase.

```
public static final String[] DATOS = {
  "Primero","Segundo","Tercero","Cuarto","Quinto"
  };
```

En este caso todas las sentencias que intervienen en la comunicación con el objeto remoto se encuentran en el método *actionPerformed()*, que es el receptor de eventos del objeto **Button** que se crea en la parte inferior de la ventana. En este método es donde se obtiene la referencia al objeto remoto y se invoca a su método *getDatos()*, para obtener el array de elementos con los que se va a sustituir la lista original de la ventana, pasando este array como parámetro al método *setDatos()* del objeto local, que es el que se muestra en la ventana.

```
try {
  listaRemota = (Java2004I)Naming.lookup( direccion+"ListaRemota" );
  lista.setDatos( listaRemota.getDatos() );
} catch( Exception e ) {
  e.printStackTrace();
  }
```

COMUNICACIÓN RMI

En esta sección solamente se trata de implementar un nuevo ejemplo de aplicación utilizando mecanismos RMI, aunque lo que se hace es utilizar como partida la misma aplicación desarrollada en el capítulo 9 en que se trataban los **Ficheros**, `Java902.java`, y adaptarla al uso de RMI. También se explica paso a paso la compilación y ejecución manual de la aplicación, sin ningún fichero de lotes, y en plataformas *Windows* y *Unix*, para que el lector disponga de un ejemplo completo, abarcando casi las circunstancias más generales que se pueden presentar.

El funcionamiento de la aplicación es muy sencillo. El primer cliente presenta una interfaz muy simple con un campo de entrada de texto y un botón. Cuando se pulsa ese botón, la cadena que se haya escrito en el campo de texto es enviada al segundo cliente a través de un objeto remoto. Cuando se pulsa el botón que presenta el segundo cliente, la cadena de texto escrita en el primer cliente se añade al texto que ya figuraba en el área de visualización del segundo cliente. La figura 20.3 muestra el estado de ejecución tras la inserción de unas cuantas cadenas de texto.

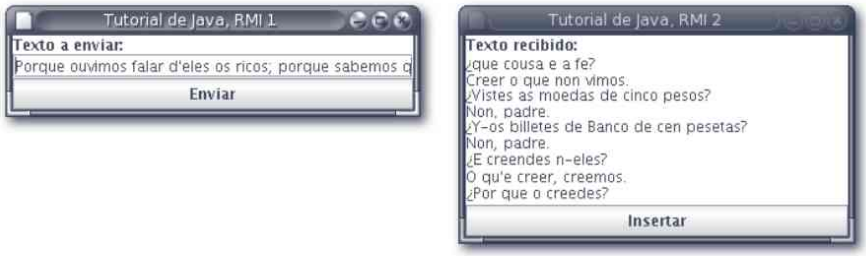


Figura 20.3

El funcionamiento del programa es el que muestra esquemáticamente la figura 20.4 y los ficheros de código fuente que componen la aplicación son los que se indican en la lista a continuación del diagrama.

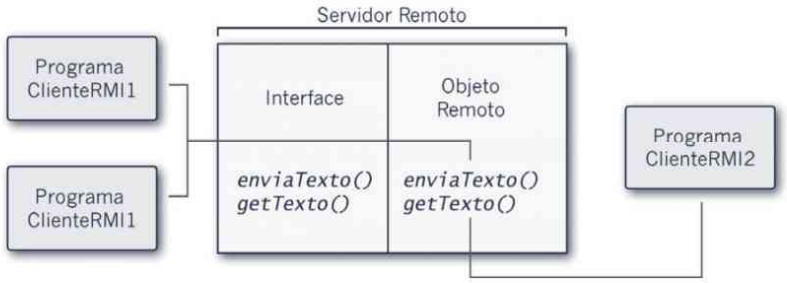


Figura 20.4

- `Java2005I.java`, la interfaz
- `Java2005S.java`, el servidor RMI
- `Java2005Ca.java`, el cliente que envía datos
- `Java2005Cb.java`, el cliente que recibe datos

Además del ya conocido fichero de normas o políticas de seguridad, que en este caso será `javaAwt.politicas`, que es el que garantiza los permisos necesarios para la ejecución del ejemplo. La diferencia existente de este fichero con el fichero de políticas de seguridad que se estaba utilizando, es que al estar utilizando AWT, que está controlado por eventos, es necesario garantizar que el servidor puede acceder a la cola de eventos de AWT para colocar sus propios eventos y poder establecer la comunicación entre servidor y cliente a través de los objetos remotos.

Compilar los fuentes

El entorno en que se compilan y ejecutan los programas corresponde al directorio local de la cuenta del usuario. La secuencia de comandos necesaria para la compilación de todos los ficheros es la que se muestra a continuación, tanto en plataformas *Unix* (Linux) como *Windows*, en cualquiera de sus versiones.

```
javac Java2005S.java
javac Java2005Cb.java
javac Java2005Ca.java
rmic -d . Java2005S
```

El primer comando compilará el servidor y la interfaz. El segundo compilará el cliente que recibe el texto. El tercero compilará el cliente que envía la cadena.

El último comando ejecuta la utilidad `rmic` sobre la clase del servidor, para generar el ficheros `_Stub`, a través de cuyas clases los clientes pueden invocar a los objetos remotos del servidor.

El primer cliente se invoca desde un directorio de la parte cliente y utiliza la parte servidor del servidor web y la parte cliente de la Máquina Virtual Java para descargar los ficheros públicamente accesibles. Y el segundo cliente es invocado del mismo modo, tal como se esquematiza en la figura 20.5.

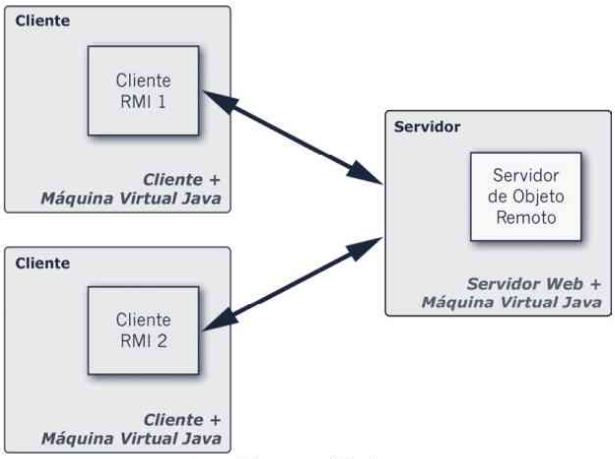


Figura 20.5

Registrar los objetos

Antes de que se puedan ejecutar los clientes, hay que lanzar el registro RMI, que es el repositorio de nombres, en el lado del servidor, que permite a los clientes obtener referencias a los objetos remotos. Es importante asegurarse de que previamente a la invocación de `rmiregistry`, la variable de entorno `CLASSPATH` no apunta al directorio que contiene las clases de los objetos remotos, ni de la clase *stub*, en ningún sitio del sistema. Si en el proceso de registro se encontraran estas clases, no se cargarían luego desde la parte servidora de la Máquina Virtual Java, lo cual podría originar problemas cuando los clientes intentasen descargar las clases del servidor remoto.

Por ello, en los siguientes comandos, primero se libera la variable `CLASSPATH`, y luego se arranca el registro RMI en el puerto por defecto, `1099`. Se puede especificar cualquier otro puerto, pasándolo como parámetro; teniendo en cuenta que si se lanza el registro sobre otro puerto, en la parte del servidor hay que indicar ese nuevo puerto.

Unix

```
unsetenv CLASSPATH
rmiregistry &
```

Windows

```
unset CLASSPATH
start rmiregistry
```

Hay que acordarse de volver a fijar los valores originales de la variable de entorno `CLASSPATH` a este nivel.

Arrancar el servidor

Antes de poder ejecutar los programas de ejemplo, es necesario lanzar el servidor. Si se lanzan primero los clientes, entonces no van a ser capaces de establecer una conexión porque el servidor no estará en ejecución. El comando en plataformas *Unix* y *Windows* es:

```
java -Djava.security.policy=javaAwt.politicas Java2005S
```

La opción `-D` que se proporciona al intérprete *`java`* indica el fichero de pólizas que contiene los permisos necesarios para la ejecución del servidor remoto y para poder acceder a la descarga de las clases.

Ejecutar los clientes

Solamente queda poner los clientes en marcha para poder utilizar la aplicación. Hay que indicar también a los clientes el fichero de políticas que deben utilizar, para poder tener acceso a las clases del servidor.

```
java -Djava.security.policy=javaAwt.politicas Java2005Ca
java -Djava.security.policy=javaAwt.politicas Java2005Cb
```

El fichero Interface

Se limita a declarar los métodos que implementa el servidor y que van a ser los que se puedan llamar remotamente.

```
public void enviaTexto( String texto ) throws RemoteException;
public String getTexto() throws RemoteException;
```

El fichero Servidor

La clase del servidor extiende a **UnicastRemoteObject** e implementa los métodos de la interfaz **Java2005I** que son accesibles remotamente.

La clase **UnicastRemoteObject** implementa una serie de métodos de la clase **Object** para que los objetos remotos y sus constructores y métodos estáticos tengan un objeto remoto disponible para recibir llamadas a sus propios métodos desde los programas cliente.

```
class Java2005S extends UnicastRemoteObject implements Java2005I {
  private String texto;
  // Constructor de la clase
  public Java2005S() throws RemoteException {
    super();
    }
  // Método que envía la cadena de texto al otro objeto Remoto
  public void enviaTexto( String texto ) {
    this.texto = texto;
    }
  // Método que recupera la cadena de texto
  public String getTexto() {
    return( texto );
    }
```

En el método *main()* lo primero que se hace es instalar el controlador de seguridad y abrir la conexión con el puerto de la máquina en que se está ejecutando este programa servidor. El controlador de seguridad determina, a través de los permisos fijados en el fichero de normas o políticas de seguridad, cuándo se debe conceder o no una descarga de código.

```
if( System.getSecurityManager() == null ) {
  System.setSecurityManager( new RMISecurityManager() );
  }
```

A continuación, se crea un nombre para el objeto en el servidor, y así poder identificarlo y se incorpora al registro invocando al método *bind()*.

```
Naming.rebind( "ObjetoRemoto",objRemoto );
```

El resto es la captura y tratamiento de las posibles excepciones que han de capturarse, por imposición.

Los ficheros Cliente

La clase correspondiente al primer cliente, **Java2005Ca**, establece una conexión con el servidor remoto y envía datos a ese objeto remoto. Todo el código para realizar las acciones se concentra en el método *actionPerformed()*, que es el que se invoca cuando se pulsa el botón de la parte inferior de la ventana.

```
// Se recupera el texto
String texto = campoTexto.getText();
String direccion = "rmi://localhost/";
try {
  Java2005I refObj =
    (Java2005I)Naming.lookup( direccion+"ObjetoRemoto" );
  // Se envía al objeto remoto
  refObj.enviaTexto( texto );
```

En el método *main()* solamente se crea una instancia del objeto y todos los adornos de la ventana.

En el otro cliente las acciones son semejantes, excepto en que en vez de obtener la cadena del campo de texto y luego enviarla a través del método del objeto remoto, se realiza el proceso contrario, que consiste en la recuperación de la cadena del objeto remoto y la invocación del método *append()* del área de texto para incorporar la cadena que había recogido el otro cliente a la información que ya figuraba en esa área de texto.

SERIALIZACIÓN DE OBJETOS

Uno de los aspectos básicos en los que se apoya RMI es en la *persistencia de objetos*, es decir, en que los objetos no se destruyan y puedan ser enviados entre máquinas en un entorno distribuido. En muchas aplicaciones, la persistencia de los datos se controla a través de ficheros de texto o bases de datos comerciales, dependiendo de la complejidad de la aplicación y de los recursos puestos a disposición de los programadores. Para aplicaciones sencillas, los ficheros de texto son suficientemente válidos, porque son flexibles, es muy fácil trabajar con ellos y no están limitados a su uso por un solo programa. Sin embargo, no son nada amigos de los objetos. Cuando el formato de un fichero se complica un poco más allá de una simple tabla o una lista de parámetros (lo que ocurre en casi todas las aplicaciones orientadas a objetos), el código para manejar estos ficheros se vuelve farragoso y consume mucho tiempo del programador.

Al otro lado del espectro están las bases de datos relacionales y orientadas a objetos que funcionan muy bien con programas que requieran características de bases de datos: transacciones, bloqueo de registros, índices, etc. Pero generalmente son muy caras, pueden llegar a ser difíciles de manejar y exigen grandes conocimientos. Actualmente se tiende a implementar la persistencia con bases de datos: Si un diseño requiere estados en que hay que guardar datos, entonces se asume que las bases de datos son la elección. En muchos casos, todo esto es un fichero con formato orientado a objetos integrado en el propio entorno de programación.

Una situación similar existe también en el entorno de la programación distribuida. Los *sockets* son flexibles y muy fáciles de utilizar, al igual que los ficheros, pero presentan los mismos problemas que éstos cuando se transmiten formatos de datos complejos. Las aplicaciones distribuidas basadas en CORBA, por ejemplo, disponen de facilidades para transmitir objetos, pero es una solución costosa.

La *serialización de objetos* en Java proporciona una solución intermedia para salvar objetos en ficheros y transmitirlos a través de la red. Incluso en grandes proyectos que utilicen bases de datos comerciales o comunicaciones *middleware*, puede ser un formato válido para ficheros auxiliares o comunicaciones variadas. Tanto RMI como el API de los *JavaBeans* utilizan la serialización para guardar y transmitir objetos. Por lo tanto, en toda aplicación Java en que se vea involucrada la persistencia o distribución de objetos, la serialización es una poderosa herramienta de programación.

La serialización de objetos en Java permite escribir y leer objetos en *streams*, sean éstos ficheros o *sockets*. Esta circunstancia proporciona a los programadores una forma sencilla de guardar tanto objetos individuales como grandes estructuras de objetos en ficheros, o enviarlos a través de la red.

Desde la perspectiva del programador, gran parte de este trabajo se realiza automáticamente. El mecanismo de serialización mantiene control sobre los tipos de los objetos, las referencias entre ellos y muchos detalles de cómo están almacenados los datos. El API de serialización está muy estructurado, de tal modo que en muchos casos se puede manejar directamente con facilidad, mientras permite realizar acondicionamientos muy complejos en caso necesario.

Grabar objetos

La mejor forma de entender los conceptos en torno a la serialización es ver un ejemplo en funcionamiento. En la mayoría de los casos, todo lo que se necesitará para que un objeto sea serializable es añadir en la declaración de la clase la implementación de **Serializable**, tal como se hace en la declaración de la clase **Arbol** del ejemplo `Java2006.java`.

```
class Arbol implements Serializable {
  Vector<Arbol> hijos;
  Arbol padre;
  String nombre;

  // Constructor de la clase, que implementa un Vector de
  // cinco elementos
  public Arbol( String s ) {
    hijos = new Vector<Arbol>( 5 );
    nombre = s;
    }

  // Método para incorporar elementos al Vector
  public void addRama( Arbol n ) {
    hijos.addElement( n );
    n.padre = this;
    }

  // Éste es el método que vuelca el contenido del vector
  // a un formato legible en pantalla, imprimiendo el contenido
  // de los elementos del Arbol
  public String toString() {
    Enumeration enu = hijos.elements();
    StringBuffer buffer = new StringBuffer( 100 );
    buffer.append( "[" +nombre+ ": " );
    while( enu.hasMoreElements() ) {
      buffer.append( enu.nextElement().toString() );
      }
    buffer.append("] ");
    return( buffer.toString() );
    }
  }
```

La interfaz **Serializable** no tiene métodos que deban ser implementados; sin embargo, cualquier campo de datos que implemente esta interfaz también debe ser serializable. Las referencias a otros objetos de tipo **Arbol**, como el que se guarda en el campo `padre` o cualquiera de los que se guarden en los elementos del vector `hijos`, son serializables por definición. Todos los tipos básicos como `int`, `String`, `Array`, `Vector` o `Hashtable` son serializables.

La clase que implementa el ejemplo engloba todas sus acciones en el método *main()*. En primer lugar, se construye un árbol de objetos **Arbol**.

```
Arbol raiz = new Arbol("raiz" );
raiz.addRama( new Arbol("izqda") );
raiz.addRama( new Arbol("drcha") );
```

El envío de objetos al canal de comunicaciones es realizado por la clase **ObjectOutputStream**, que implementa la interfaz genérica **ObjectOutput**. Esta interfaz es una extensión, a su vez, de la interfaz **DataOutput**, que le añade la capacidad de escribir objetos, así como tipos básicos como cadenas y números. El objeto **ObjectOutputStream** se crea por encima de cualquier otro **OutputStream**, como por ejemplo un fichero o un socket. En este ejemplo se crea como fichero.

```
FileOutputStream fOut = new FileOutputStream( "prueba.rmi" );
ObjectOutput out = new ObjectOutputStream( fOut );
```

Ahora ya se puede utilizar el método *writeObject()* para enviar datos al canal abierto hacia el fichero.

```
out.writeObject( raiz );
```

Y finalmente volcar todo el contenido y cerrar el canal.

```
out.flush();
out.close();
```

La clase **ObjectOutputStream** permite escribir un objeto dato en un canal, así como cualquier otro objeto al que se llegue a través del primero; es decir, permite enviar la estructura de objetos completa. Cuando se escribe un objeto en un canal, el **ObjectOutputStream** mantiene todas las referencias entre objetos, de forma que la complejidad de las estructuras de datos también se mantiene. Del mismo modo, **ObjectOutputStream** mantiene controlados todos los tipos de objetos, de forma que se puedan leer correctamente.

Leer objetos

Para leer objetos de un canal de entrada se utiliza la clase **ObjectInputStream**, para implementar la interfaz **ObjectInput**. Del mismo modo que con **ObjectOutputStream**, un objeto **ObjectInputStream** se crea a partir de un **InputStream**, utilizando el método *readObject()* para leer un objeto de ese canal.

```
FileInputStream fIn = new FileInputStream( "prueba.rmi" );
ObjectInputStream in = new ObjectInputStream( fIn );
Arbol n = (Arbol)in.readObject();
```

Para cada objeto que sea detectado en el canal, se crea otro nuevo en memoria, rellenando sus datos a partir del que se lee del canal de entrada. Esto incluye la recuperación de las referencias entre los objetos almacenados en el stream.

El método *readObject()* devuelve un **Object** que debe ser moldeado, en este caso a un tipo **Arbol** antes de poder utilizarlo como un objeto **Arbol**. La cuestión estriba en cómo saber el tipo real del objeto; y la respuesta consiste en solicitar más información al objeto **Object** devuelto sobre sí mismo, utilizando el método *getClass()*, o utilizando las capacidades de reflexión que permite la plataforma Java 2.

Personalizar la comunicación

A través de la interfaz **Serializable** la mayoría del trabajo pesado que interviene en la serialización de objetos se realiza automáticamente; sin embargo, el programador tiene varias opciones para poder configurar esa serialización.

- La palabra clave `transient` se puede utilizar para definir campos de datos que deben mantenerse a la hora de escribir en un canal. Cuando se lee un objeto de un canal, los datos de estos campos transitorios se inicializan a sus valores por defecto, como `0` para los números enteros o `null` para las cadenas. El programador puede restaurar los datos `transient` implementando un método *readObject()*.
- A un objeto se le pueden incorporar controles adicionales en su proceso de serialización implementando métodos propios *writeObject()* y *readObject()*. Esto puede ser útil para escribir una versión personalizada del objeto, o simplemente escribir datos adicionales en el canal.

Cuando un **ObjectOutputStream** escribe un objeto en un canal, siempre busca el método *writeObject()* en el objeto. Si hay uno, lo utiliza para enviar el objeto al canal. Un objeto que implemente este método puede utilizar la representación por defecto del objeto para escribir el objeto en el canal, o bien puede utilizar otros métodos de **ObjectOutput**, como *writeChar()* o *writeDouble()* para enviar los datos al canal.

En el otro lado de la comunicación, el programador puede implementar un método *readObject()* propio para tomar el control de la forma en que se va a leer el objeto del canal. Este método es llamado tras instanciar un nuevo objeto en memoria y antes de leer los datos del canal. De forma similar a *defaultWriteObject()* en la escritura, en este caso se puede utilizar el método *defaultReadObject()* de **ObjectInputStream** para leer la representación por defecto del objeto. En los dos casos, estos métodos solamente son responsables de la escritura y lectura de datos para el objeto en que están definidos, no para las subclases o superclases.

VALIDACIÓN DE OBJETOS

Un objeto que implemente el método *readObject()* puede registrarse con el **ObjectInputStream** para disponer de un método de validación después de que el objeto haya sido recuperado completamente del canal de entrada. Solamente el método *readObject()* puede registrar un método de validación. Para el ejemplo anterior, esto significaría que el método de validación debería invocarse después de que el árbol completo se haya reconstruido en memoria, pero antes de que la llamada a *ObjectInputStream.readObject()* retorne. El método de validación está definido en la interfaz **ObjectInputValidation** a través de un método *validateObject()* simple. Esto permite que el objeto pueda hacer trabajo extra, como comprobar si hay errores internos de consistencia entre los objetos, que no pueden realizarse dentro del método *readObject()*. En el ejemplo `Java2007.java` se muestra la forma de realizarlo.

Con la interfaz **Externalizable**, el programador asume toda la responsabilidad para la lectura y escritura de objetos en un *stream*, incluyendo sus subclases y sus superclases. Esto proporciona una flexibilidad total, sobre todo, cuando hay un formato de datos predefinido, o, para cualquier formato de datos que el programador tenga en mente; aunque, eso sí, requiere mayor esfuerzo de programación.

En el ejemplo `Java2007`, se utilizan dos campos transitorios, `instante` y `contador`, que se añaden al objeto **Arbol**. El campo `instante` guarda la fecha en que se instancia el objeto, que se actualizará cada vez que se lea el objeto del canal de entrada, por lo que no tiene sentido alguno hacerlo persistente. El `contador` almacena el número de veces que el objeto ha sido leído del canal, y se utiliza para demostrar cómo se leen y escriben datos extra en un objeto junto con los datos por defecto del objeto. Además, el método de validación se utiliza para comprobar las referencias a los objetos en el árbol.

Cuando un objeto se escribe en un canal y luego se lee, el objeto que devuelve el método *ObjectInputStream.readObject()* no es el mismo objeto que el que se guardó originalmente, sino que es un *clon* construido a partir de los datos recuperados del canal. Por esta razón, las identidades se vuelven más complejas al estar en un entorno persistente o distribuido, y han de ser consideradas en el diseño del programa.

El `contador` en este ejemplo indica la generación de un clon del **Arbol** que se está leyendo. Si el `contador` de un **Arbol** es cero, entonces es que ha sido instanciado utilizando el operador `new`. Si se trata de un número mayor que cero, entonces es que ha sido recuperado de un objeto *stream*. Las siguientes líneas de código muestran la declaración de la clase que incorpora la cláusula **ObjectInputValidation** y la declaración de los dos campos.

```
class Arbol implements Serializable,ObjectInputValidation {
  // Declaramos una variable para señalar el instante en que el objeto
  // es instanciado y el momento en que es leído del stream
  transient Date instante;
  // Declaramos otra variable, contador, para marcar el objeto cuando
```

```
  // se lee del stream
  transient int contador = 0;
```

La implementación del método *validateObject()* es la que se reproduce seguidamente, donde se comprueba la estructura del árbol, así como que cada hijo de un nodo tiene asignado el mismo padre. En la práctica, se pueden utilizar los métodos de validación para determinar si se han producido errores en el canal, o bien para comprobar que se trata de versiones correctas de los objetos, como se verá más adelante.

```
public void validateObject() throws InvalidObjectException {
  Enumeration enu = hijos.elements();
  while( enu.hasMoreElements() ) {
    if( ((Arbol)enu.nextElement()).padre != this )
      throw new InvalidObjectException( getClass().getName() );
    }
  }
```

Y el repaso al código completo del ejemplo concluye con la revisión de los métodos *writeObject()* y *readObject()*.

```
private void writeObject( ObjectOutputStream canal )
  throws IOException {
  try {
    canal.defaultWriteObject();
    // Se escribe el contador
    canal.writeInt( contador );
  } catch( Exception e ) {
    e.printStackTrace();
    }
  }

private void readObject( ObjectInputStream canal )
  throws IOException {
  try {
    canal.registerValidation( this,0 );
    canal.defaultReadObject();
    contador = canal.readInt()+1;
    instante = new Date();
  } catch( Exception e ) {
    e.printStackTrace();
    }
  }
```

En el método *writeObject()*, se utiliza la llamada a *defaultWriteObject()* para enviar al canal la representación por defecto del objeto. Después de esto, se escribe el `contador`. Aquí se muestra cómo se controlan los objetos que se escriben en el canal de salida. Como **ObjectOutputStream** implementa la interfaz **DataOutput**, los tipos de datos básicos como **String** o `int` se pueden escribir como si de objetos se tratase.

En la implementación del método *readObject()* se utiliza la llamada al método *registerValidation()* para la comprobación del objeto que se recupera del canal. El primer argumento de la llamada es el objeto sobre el que se va a invocar el método

validateObject(), y el segundo es la prioridad en caso de haber muchos métodos de validación. Los métodos con prioridad más alta son invocados en primer lugar.

Luego se utiliza el método *defaultReadObject()* para leer el objeto **Arbol**, el contador se lee como `int`, se incrementa y se asigna al campo correspondiente del nuevo objeto.

Problemas de versiones

Uno de los problemas más complicados con los que debe lidiar cualquier mecanismo de persistencia al utilizar objetos es la versión de esos objetos. Inevitablemente, las definiciones de las clases para todos los objetos cambian a lo largo del tiempo y esto significa que, en un mismo punto, el mecanismo de serialización puede estar leyendo un objeto cuya estructura está obsoleta con respecto a la versión actual del objeto que se está manejando.

En la especificación de los mecanismos de serialización se indican numerosos casos que pueden presentarse, y aunque a la larga se produzcan cambios en un objeto o cambios en la localización jerárquica de un objeto, siempre se requiere la intervención del programador para actualizar manualmente los objetos desfasados, aunque hay veces en que se puede automatizar este proceso.

Por ejemplo, un caso habitual es cuando se añaden nuevos campos de datos a la clase. En este caso, se creará un nuevo objeto sobre el que se volcarán los datos de la versión antigua en los campos adecuados y los nuevos se inicializarán a sus valores de defecto. La clase puede realizar una posterior inicialización implementando un método de lectura del objeto o utilizando el mecanismo de validación.

Uso de sockets para distribuir objetos

Como los canales **DataInput** y **DataOutput** de un socket funcionan de forma semejante a los canales que se establecen para ficheros, **ObjectOutputStream** y **ObjectInputStream** pueden escribir y leer objetos a través de ellos. En el ejemplo **Java2008x** se reconvierte el ejemplo anterior para que permita el envío desde una aplicación cliente al servidor utilizando una conexión establecida a través de sockets. El servidor añadirá un nuevo nodo al **Arbol** que se le pase y lo devolverá al cliente.

El uso de la serialización sobre sockets permite que dos aplicaciones Java puedan comunicarse fácilmente con cualquier otra en términos de objetos en lugar de caracteres, haciendo más sencilla la creación de aplicaciones distribuidas. El enlace de comunicación que es posible establecer entre un applet y un servidor ejecutándose concurrentemente sobre un servidor Web, extiende enormemente las capacidades de ese applet.

Por conveniencia, en el ejemplo se asume que tanto cliente como servidor se ejecutan sobre la misma máquina, aunque el código puede ser modificado fácilmente para que se ejecute sobre máquinas diferentes. El cliente inicia la comunicación estableciendo y abriendo un socket con el servidor.

```
cliente = new Socket( servidor,puerto );
```

A continuación, como en el ejemplo `Java2006.java`, construye un árbol:

```
Java2008T raiz = new Java2008T( "raiz" );
raiz.addRama( new Java2008T("izqda") );
raiz.addRama( new Java2008T("drcha") );
```

y luego crea el canal con el socket y lo escribe en ese canal de salida.

```
ObjectOutput out =
  new ObjectOutputStream( cliente.getOutputStream() );
// Ahora enviamos el árbol completo a través de ese canal
out.writeObject( raiz );
out.flush();
```

En el servidor, después de que la conexión a través del socket haya sido aceptada, se establecen los canales de salida y entrada:

```
// Se crea el canal de salida
out = new ObjectOutputStream( socket.getOutputStream() );
// Se crea el canal de entrada
in = new ObjectInputStream( socket.getInputStream() );
```

luego se lee el **Arbol** del canal de entrada, se le incorpora un nuevo elemento y se escribe en el canal de salida:

```
Java2008T n = (Java2008T)in.readObject();
n.addRama( new Java2008T( "Rama Servidor" ) );
out.writeObject( n );
out.flush();
```

Volviendo al cliente, el árbol ahora reflejará los cambios que ha introducido el servidor, leyendo el objeto que devuelve éste del socket e imprimiéndolo en pantalla.

```
// Creamos el canal de lectura desde el servidor
ObjectInputStream in= new ObjectInputstream(cliente.getInputStream());
// Recuperamos el objeto del canal
Java2008T n = (Java2008T)in.readObject();
// Imprimimos el resultado de la lectura del objeto recibido
// a través de ese canal
System.out.println("Arbol Leido: \n" + n.toString());
```

Seguridad

Uno de los aspectos de la serialización que requiere especial consideración, especialmente cuando se utilizan sockets, es la seguridad. Un objeto serializado viajando a través de la red está sujeto a las mismas violaciones que cualquier mensaje

de correo electrónico o cualquier comunicación sin encriptar. Con lo cual, puede ser fácilmente interceptado.

En general, datos críticos en objetos serializables, como descriptores de ficheros o controladores de recursos del sistema, se deben hacer privados. Esto evitará que sean escritos a la hora de serializar el objeto; y cuando este objeto se lea, solamente la clase puede asignar un valor a ese campo privado. También se puede utilizar un método de validación para comprobar la integridad de un grupo de objetos a la hora de leerlos de un canal de entrada.

No obstante, la mejor forma de evitar problemas de seguridad es encriptar los datos serializados que se envían al canal de salida. Esto se puede hacer implementando una versión propia de los métodos *readObject()* y *writeObject()*, o utilizando la interfaz **Externalizable**. En una solución más global, las clases **ObjectInputStream** y **ObjectOutputStream** pueden ser personalizadas para encriptar el objeto *stream* completo.

CAPÍTULO 21

JMX

La gestión de los recursos Java, tanto físicos (dispositivos de red, memoria, etc.) como virtuales (aplicaciones, drivers, etc.), a través de la red se realiza mediante una interfaz de gestión. Implementar un recurso significa crear un objeto Java que represente la interfaz de gestión de ese recurso, siguiendo el modelo JavaBean, que recibe el nombre del **ManagedBean** (***MBean***) . Este **MBean** es un objeto Java que representa a la parte del recurso gestionable, no al recurso en sí.

Toda la gestión de recursos se realiza a través de las *Extensiones de Gestión Java* (***JMX***), que definen una arquitectura, API, servicios, patrones de diseño, gestión de red y monitorización, utilizando lenguaje Java. Proporcionan mecanismos que permiten la creación de agentes y gestores para implementar aplicaciones de gestión distribuida o integrarlos en aplicaciones de gestión ya existentes.

ARQUITECTURA JMX

La arquitectura JMX está fundamentada sobre un modelo de tres capas:

1. *Nivel de instrumentación*. Añade la capacidad de gestión a cualquier recurso Java.
2. *Nivel de agente*. Aporta contenedores que ofrecen los servicios de gestión fundamentales, permitiendo su extensión de forma dinámica.
3. *Nivel de gestión*. Incorpora los componentes de gestión, que pueden actuar como gestores o como agentes, respecto a la distribución o consolidación de los servicios de gestión.

La arquitectura JMX comprende, pues, tres componentes fundamentales: el recurso gestionable, el agente y el gestor. Cada uno de ellos integrando una de las capas del modelo.

Un *recurso gestionable* JMX es aquel que ha sido creado siguiendo la especificación al nivel de instrumentación anterior. Debe estar desarrollado en Java directamente, o debe haber un recurso escrito en Java que actúe como intermediario ante el recurso real que se desee gestionar. El MBean representa al recurso gestionable, pudiendo ser insertado dentro de cualquier agente JMX.

El *agente* JMX es una entidad implementada de acuerdo con el nivel de agente de la arquitectura. Está formado por un servidor, una serie de recursos gestionables y al menos un adaptador de protocolo o conector, además de poder contener servicios de gestión. Todos los elementos anteriores se representan a través de MBeans.

El *servidor* funciona como un directorio de MBeans y es el encargado de suministrar los servicios que permiten su control. Las operaciones que se realizan sobre los servicios se hacen a través de interfaces que proporciona el servidor.

Los *conectores* y *adaptadores de protocolo* permiten que las aplicaciones de gestión puedan acceder a los agentes y puedan realizar las acciones sobre los elementos que están contenidos en el servidor. Sirven para representar los MBeans mediante otros protocolos y están implementados del lado del agente, el cual utiliza protocolos específicos como HTTP, SNMP, RMI, etc. para comunicarse con la aplicación de gestión.

Los conectores están constituidos por dos componentes, uno en el lado de la aplicación y otro en el lado del agente, permitiendo la comunicación entre los dos a través de una amplia variedad de protocolos, como HTTP, HTTPS, RMI, IIOP, SNMP, CORBA, AMI, etc. Los conectores están basados en interfaces, circunstancia que facilita la selección del conector más adecuado al entorno de la aplicación de gestión y, posteriormente, si fuese necesario, cambiar dinámicamente de conector. Por ejemplo, una aplicación puede comunicarse con un agente mediante HTTP en el momento de arrancar y luego continuar la comunicación mediante HTTPS.

El *gestor* JMX es una entidad de gestión cuya implementación sigue la especificación indicada en el nivel de gestión JMX. Proporciona una interfaz mediante la cual los agentes se comunican con las aplicaciones y se garantiza la seguridad de las operaciones que se realicen.

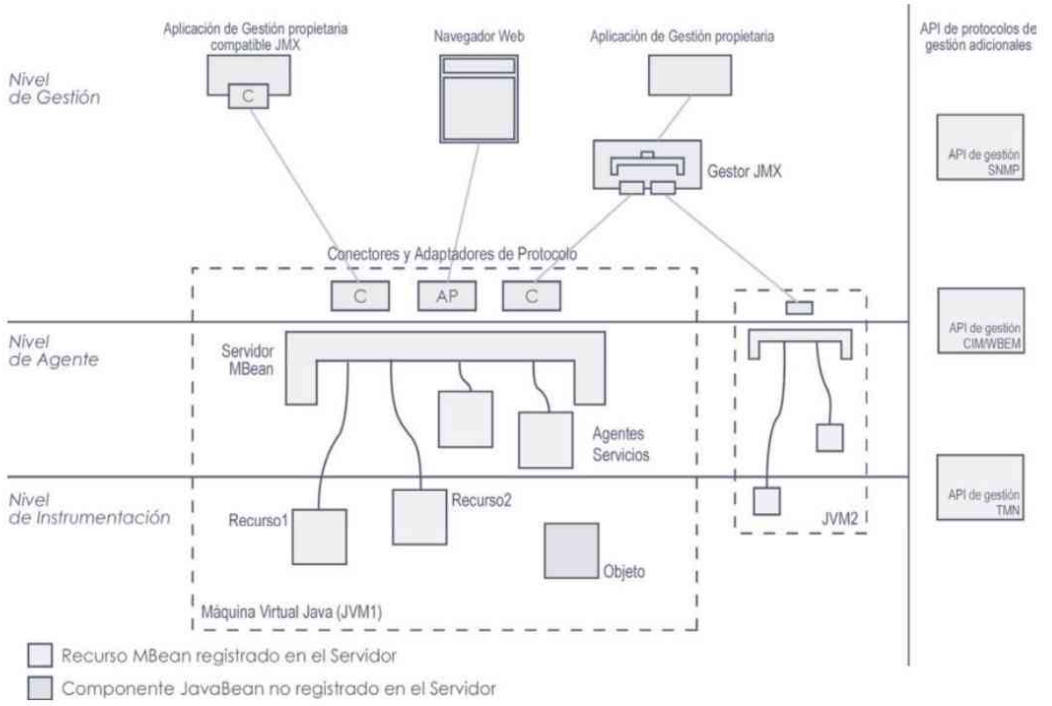


Figura 21.1

Los servicios se crean como MBeans y se pueden añadir y eliminar dinámicamente. La especificación JMX solamente define una interfaz para servicios básicos, como puede ser el registro de MBeans, búsquedas en ese registro, notificación de eventos, la carga dinámica, funciones de monitorización de atributos, creación de dependencias entre MBeans; aunque también es posible incluir otras como persistencia, seguridad, descubrimiento de agentes y gestores, etc.

INSTRUMENTACIÓN DE RECURSOS

Como se ha indicado anteriormente, un recurso es cualquier entidad física o virtual que se desee gestionar a través de la red y recibe el nombre de MBean. Un MBean puede tener atributos que pueden ser de lectura o escritura, operaciones que pueden ser invocadas y notificaciones que el MBean puede enviar. Cada MBean tiene un nombre, que es una instancia de la clase **ObjectName**. Este nombre pertenece a un dominio y puede tener una o más propiedades, por ejemplo:

```
com.tutorial: type=Configuracion
com.tutorial: type=Configuracion, name=configCache
```

en donde `com.tutorial` es el dominio y la parte derecha de los dos puntos (`:`) corresponde a las propiedades.

Para que un MBean resulte útil ha de registrarse ante un servidor mediante su **ObjectName** y, normalmente, se accede a los MBeans únicamente a través del servidor MBean. Es posible tener más de un servidor MBean en la misma máquina virtual Java.

Un recurso se puede instrumentar de cuatro formas distintas, tal como se describe en los siguientes puntos.

1. A través de MBean estándar (*Standard MBean*). Es la forma más sencilla. Sigue el modelo de los JavaBeans y su interfaz de gestión es estática, es decir, contiene métodos para obtener y asignar valores a los atributos. Estos atributos que expone el objeto son determinados en tiempo de compilación y no pueden ser modificados posteriormente en tiempo de ejecución.
2. A través de MBean dinámico (*Dynamic MBean*). No contiene métodos para asignar y obtener los valores de cada atributo, sino que proporciona métodos genéricos para obtener y asignar valores a los atributos e invocar operaciones. Puede ser implementado para acceder a objetos Java que no sigan el modelo JavaBeans o a recursos que no utilicen tecnología Java.
3. A través de MBean modelo (*Model MBean*). Constituye un MBean dinámico, genérico y configurable que se utiliza para instrumentar recursos en tiempo de ejecución.
4. A través de MBean abierto (*Open MBean*). Permite utilizar objetos gestionados que son descubiertos en tiempo de ejecución.

El servidor es el componente central del agente y consiste en un directorio de las instancias de los MBeans. Expone una interfaz genérica a través de la cual pueden interaccionar las aplicaciones. Las aplicaciones acceden a través del servidor a los MBeans, nunca directamente.

Como ya se ha indicado anteriormente, las aplicaciones de gestión acceden a los MBeans remotamente a través del servidor implementado en el agente, a través de los conectores o los adaptadores de protocolo. Las aplicaciones de gestión escritas completamente en Java utilizan los conectores para interactuar con los agentes, mientras que los adaptadores de protocolo son utilizados cuando se quiere integrar un agente JMX en un entorno de gestión ya existente.

A continuación se muestra la instrumentación de una aplicación utilizando MBeans. Para ello, se crea un MBean estándar simple que contenga una única operación y luego se registra en el servidor MBean.

Un MBean estándar se define escribiendo una interfaz en la que cada método define una operación o atributo del MBean y la clase que la implementa. En este caso se define la interfaz **Java2101MBean** y la clase **Java2101** que la implementa. El código correspondiente a la interfaz **Java2101MBean** se reproduce a continuación.

```
public interface Java2101MBean {
  // Atributos, con sus métodos set y get
  // Atributo de lectura/escritura para indicar el tamaño del caché
  public int getTamCache();
  public void setTamCache( int tamano );
  // Atributo de sólo lectura (eliminamos el método set) para el
```

```
// nombre de tipo String
public String getNombre();
// Método que genera un mensaje de salida
public void mensaje();
// Método que realiza la suma de dos argumentos
public int suma( int x,int y );
}
```

La operación *mensaje()* no necesita parámetros; sin embargo, el MBean puede tener métodos con parámetros e indicar el tipo de retorno, como se muestra en la definición del método *suma()*, al cual se pasan dos parámetros y devuelve su suma.

Los MBeans también pueden tener atributos, bien de lectura/escritura o de solo lectura. En la interfaz **Java2101MBean** se definen dos atributos, uno de tipo **String** de solo lectura y otro que indica el tamaño del caché de tipo `int`, creado como atributo de lectura y escritura.

Un MBean puede generar notificaciones, por ejemplo para indicar un cambio de estado, la llegada de un evento o un problema. Para generar notificaciones, un MBean debe implementar la interfaz **NotificationBroadcaster**. Normalmente, se hace extendiendo la clase **NotificationBroadcasterSupport**, de modo que para enviar una notificación basta con crear un objeto de tipo **Notification** y pasarlo como parámetro al método *sendNotification()*.

```
Notification n = new AttributeChangeNotification( this,
  numSecuencia++,System.currentTimeMillis(),
  "Tamano Cache Cambiado","TamCache","int",
  antTamCache,this.tamCache );
sendNotification( n );
```

Cada notificación tiene un *origen*, que es el **ObjectName** del Mbean que emite la notificación. Por conveniencia, un MBean puede emitir una notificación donde el origen es una referencia directa a sí mismo en lugar del **ObjectName**, dejando que el servidor de MBeans se encargue de sustituir la referencia por el **ObjectName** correcto.

También cada notificación debe tener un número de secuencia. Esto se hace para ordenar las notificaciones que tienen un mismo origen, y aunque es posible y admisible asignar un cero a todas ellas, es mejor incrementar el número de secuencia asignado a cada notificación.

Un MBean necesita también declarar un método que devuelva un objeto de tipo **MBeanNotificationInfo** que describa la notificación para que cualquier cliente de control sepa que el MBean ha lanzado el mensaje **AttributeChangeNotification**.

```
@Override
public MBeanNotificationInfo[] getNotificationInfo() {
  String[] tipos = new String[] {
    AttributeChangeNotification.ATTRIBUTE_CHANGE
    };
```

```
String nomClase = AttributeChangeNotification.class.getName();
String descripcion = "Un atributo de este MBean ha cambiado";
MBeanNotificationInfo info =
  new MBeanNotificationInfo( tipos,nomClase,descripcion );
return( new MBeanNotificationInfo[] {info} );
}
```

Como se puede observar en el código anterior, **MBeanNotificationInfo** define un nombre, que corresponde a la clase Java de la notificación y una lista de tipos. Cada notificación define una cadena que permite distinguir entre distintos tipos de notificación de una misma clase. En este caso, solamente hay un tipo, definido por la constante `ATTRIBUTE_CHANGE`. El objeto **MBeanNotificationInfo** así creado es el que devuelve el método *getNotificationInfo()*.

La aplicación `Java2101App.java` construye y registra un MBean de tipo **Java2101**. Si se ejecuta esta aplicación con el comando:

```
% java -Dcom.sun.management.jmxremote Java2101App
```

o con el comando siguiente, en caso de problemas con el `CLASSPATH`:

```
% java -cp .:JAVA_HOME/lib -Dcom.sun.management.jmxremote Java2101App
```

donde `JAVA_HOME` es el directorio de instalación del JDK.

Desde la aplicación *jconsole*, cuyo funcionamiento se describe en secciones posteriores, ya será posible visualizar e interaccionar con el MBean definido por la interfaz **Java2101MBean**.

Toda la discusión anterior hacía referencia a la instrumentación de recursos mediante un MBean estándar cuya interfaz se define en tiempo de compilación. El API JMX también permite el uso de MBeans dinámicos, cuya interfaz puede determinarse en tiempo de ejecución; por ejemplo, definiéndola mediante un fichero XML que sea analizado cuando la aplicación se esté ejecutando.

Para definir un MBean dinámico es necesario que la clase Java implemente la interfaz **DynamicMBean**. El ejemplo `Java2102.java` implementa un MBean dinámico. El único código interesante de la clase corresponde al método *invoke()*, que ha de comprobar si el nombre de la operación es *mensaje* y que el parámetro y *signature* de los arrays son `null` o vacíos. Si la comprobación es correcta enviará el mensaje y devuelve `null`; en caso contrario, se lanza una excepción.

```
public Object invoke( String actionName,Object[] params,
  String[] signature ) throws ReflectionException {
  // Solamente reconocemos la operación "mensaje", que no debe tener
  // parámetros. Si se trata de esta operación, escribimos el mensaje.
  // En caso contrario, lanzamos una excepción.
  if( actionName.equals("mensaje") &&
    ( params == null || params.length == 0 ) &&
    (signature == null || signature.length == 0) ) {
```

```
    System.out.println("Tutorial de Java");
    return( null );
    }
  else {
    Exception e = new NoSuchMethodException( actionName );
    throw new ReflectionException( e );
    // También se pueden lanzar excepciones de tipo RuntimeException
    // como IllegalArgumentException
    }
  }
```

La clase **Java2102** también implementa el método *getMBeanInfo()* que debe devolver un objeto de tipo **MBeanInfo** correspondiente a la interfaz del MBean, indicando sus atributos, operaciones, constructores y notificaciones. Este ejemplo es simple y no proporciona atributos o notificaciones y tampoco se necesita una lista de constructores. Lo único que necesita es la lista de operaciones. Se construye un objeto **MBeanOperationInfo** que describe la operación mensaje. Luego se construye un objeto **MBeanInfo** pasando `null` en los parámetros, excepto en las operaciones, en donde se pasa un array de tipo **MBeanOperationInfo**, que solamente contiene un elemento, el objeto que antes se había construido. El código siguiente muestra el método completo:

```
public MBeanInfo getMBeanInfo() {
  MBeanOperationInfo mensajeInfo =
    new MBeanOperationInfo( "mensaje",  // nombre del método
      "Imprime mensaje",                // descripción
      null,                             // signature
      "void",                           // tipo del retorno
      MBeanOperationInfo.ACTION         // impact
      );
  MBeanOperationInfo[] opInfo =
    new MBeanOperationInfo[] { mensajeInfo };
  MBeanInfo info =
    new MBeanInfo( "Java2102",  // nombre de la clase
      "MBean Dinamico",         // descripción
      null,                     // atributos
      null,                     // constructores
      opInfo,                   // operaciones
      null                      // notificaciones
      );
  return( info );
  }
```

El API de monitorización es muy fácil de utilizar. Por ejemplo, la aplicación `Java2103.java` muestra un informe detallado del uso de memoria por parte de la máquina virtual Java.

```
public class Java2103 {
  public static void main( String[] args ) {
    List<MemoryPoolMXBean> memoria =
      ManagementFactory.getMemoryPoolMXBeans();
    for( MemoryPoolMXBean m: memoria ) {
      System.out.printf( "Tipo de memoria= %s%n%s%n",
        m.getType(),m.getUsage() );
```

```
        }
    }
}
```

UTILIZACIÓN DE JCONSOLE

La aplicación *jconsole* es una aplicación Java que proporciona el JDK mediante la cual se pueden monitorizar los recursos que está utilizando la máquina virtual Java.

Para ejecutar esta aplicación, se puede hacer desde el directorio *bin* de la instalación del JDK, ejecutando en una ventana de comandos del sistema operativo la sentencia:

```
% jconsole
```

Cuando aparece la aplicación en pantalla, presenta la ventana de conexión. Para realizar la monitorización de la propia aplicación *jconsole*, como ejemplo, basta con seleccionar la pestaña *Remote*, aceptar los valores que presentan los campos *Host or IP*, *localhost*, *Port* y pulsar el botón *Connect*.

En pantalla aparecerá la ventana de *Java Monitoring & Management Console*, en la que se observan una serie de pestañas correspondientes a los distintos paneles de información. La figura 21.2 reproduce la aplicación en ejecución con el panel *Memory* seleccionado.

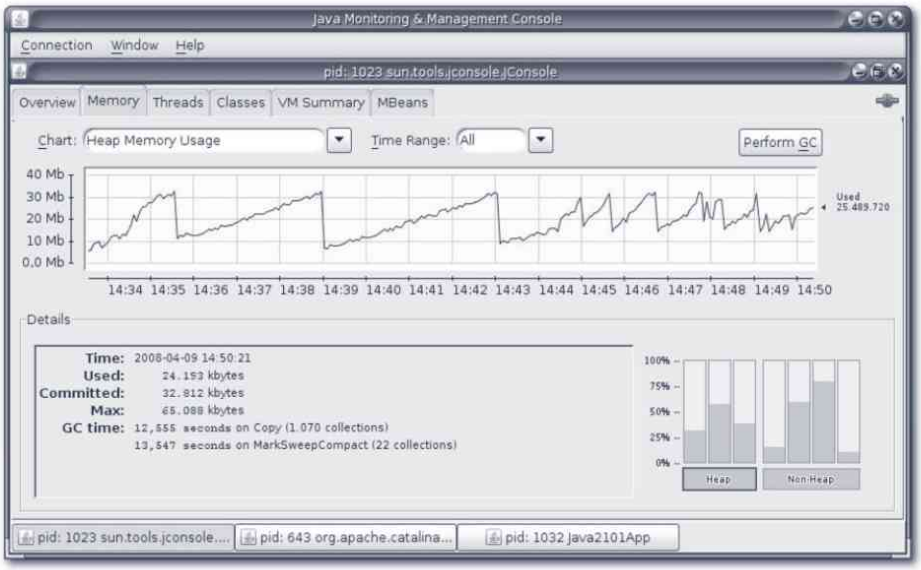


Figura 21.2

Los paneles más importantes de la aplicación son: *Memory*, *Threads* y *MBeans*.

El panel *Memory* muestra la utilización de la memoria, tanto de la memoria de la pila como de la memoria que utiliza la máquina virtual Java no perteneciente a la pila. Además, el menú desplegable de la opción *Chart* permite ver la gráfica de utilización

en rango de tiempo de las diferentes zonas de la memoria. También ofrece un botón para invocar al *garbage collector* directamente.

El panel *Threads* presenta una gráfica del número de tareas que se han lanzado a lo largo del tiempo. Permite seleccionar cualquiera de ellas y ver información adicional, incluso la historia de la llamada a métodos.

El panel *MBeans* permite monitorizar los MBeans bajo control de la máquina virtual, incluyendo los propios *beans* que se utilizan para mostrar información en los otros paneles.

A continuación se indica cómo monitorizar una aplicación local y una aplicación remota. La *monitorización local* es un método seguro, pero solamente permite monitorizar aplicaciones que se ejecuten en la misma máquina en que se ejecuta *jconsole*, porque todos los procesos de autenticación y control de acceso se dejan en manos del sistema operativo de la máquina. En este caso, no es necesario seleccionar ningún puerto único de comunicación o fijar contraseña alguna.

La aplicación `Java2101App.java` es la que se utilizará de ejemplo para su monitorización. Se ejecutará con el comando:

```
% java -Dcom.sun.management.jmxremote Java2101App
```

Una vez en ejecución, la aplicación solamente imprimirá en pantalla el mensaje de que se queda a la espera. En este momento, lanzar la aplicación *jconsole* y aparecerá la ventana de conexión; en caso de que ya se encuentre en ejecución *jconsole*, seleccionar en el menú superior la opción *Connection* y luego *New Connection*. En cualquiera de los dos casos, *jconsole* presentará la ventana de conexión en la cual, bajo la pestaña *Local*, aparecerá la aplicación que se había lanzado antes, que habrá que seleccionar.

Todas las pestañas presentan información correspondiente a la ejecución de la aplicación. Por ejemplo, la figura 21.3 muestra el contenido de la pestaña *MBeans*, en donde está desplegado el MBean correspondiente a la aplicación, mostrando los métodos de ésta que están accesibles.

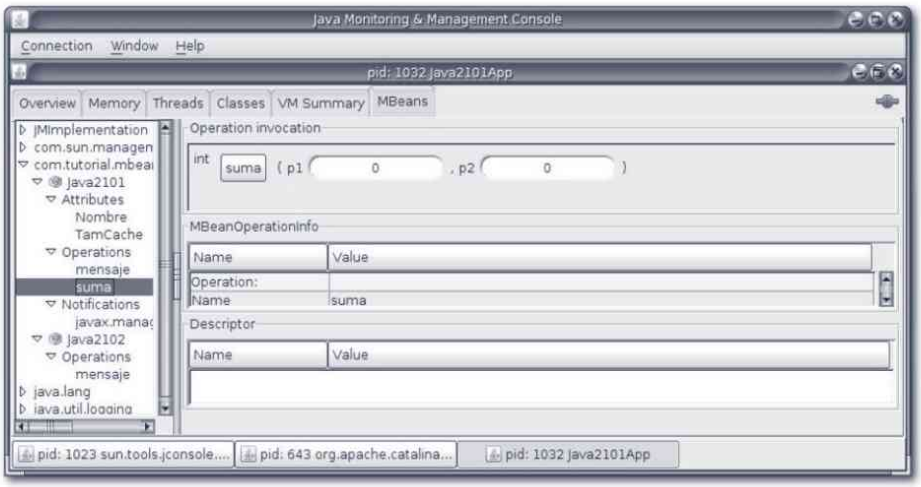


Figura 21.3

Si se pulsa en la ventana anterior el botón *mensaje*, en la consola en donde estaba lanzada la aplicación aparecerá el texto del mensaje que implementa el método *mensaje()* del MBean.

La *monitorización remota* se puede realizar de forma segura o de forma no segura. Esta última forma se emplea durante el desarrollo, porque no es necesario fijar contraseña alguna, aunque sí hay que indicar un puerto de comunicación, que será a través del cual se conecte *jconsole* al MBean. La forma segura se utiliza cuando el sistema requiere autenticación basada en contraseña, permitiendo incluso activar *SSL* para que esas contraseñas no se transmitan en claro. Las propiedades que se pueden indicar a la hora de lanzar la aplicación que invoca al MBean son:

```
com.sun.management.jmxremote.port
com.sun.management.ssl
com.sun.management.authenticate
com.sun.management.jmxremote.port
```

Esta última propiedad indica un fichero conteniendo la contraseña, para crear el cual *jre* proporciona una plantilla conteniendo instrucciones sobre su configuración.

APÉNDICE A

BIBLIOGRAFÍA

A continuación, se indican al lector algunos de los textos utilizados por el autor como referencia en el estudio de la tecnología Java y textos adicionales de lectura a los que puede recurrir para ampliar sus conocimientos. En general, cualquiera de los libros reseñados cubre el estudio básico y desarrollo de aplicaciones basadas en la tecnología Java, proporcionando abundantes ejemplos de uso, profundizando algunos de ellos en alguna parcela en concreto.

Los recursos de Internet que se relacionan pueden servir al lector para mantenerse informado y actualizado respecto a todo lo que se mueve en torno a la tecnología Java, además de encontrar ejemplos de código e información, aunque muy general en la mayor parte de los casos. Se indican los sitios web correspondientes a las herramientas de desarrollo Java más extendidas, aunque basta recurrir a un buscador para localizar infinidad de otros recursos que la comunidad Internet pone a disposición de los desarrolladores.

De consulta obligada es, por supuesto, la página Web de Java de *Sun* y el Tutorial de Java (en inglés) que el lector puede encontrar en esa página, fuente de conocimiento para todas las características de Java.

Libros

C. HORSTMANN, G. CORNELL, *Core Java, Volume 1, Fundamentals*
Prentice-Hall, 2007 ISBN: 0132354754

Libro que revisa por completo la programación orientada a objetos utilizando Java. Trata de forma muy especial todo lo referente a Swing. Dispone de gran cantidad de ejemplos de código fuente Java. Incluye secciones en las que se tratan los metadatos, objetos distribuidos, métodos nativos, JavaBeans, etc.

K. ARNOLD, J. GOSLING, D. HOLMES, *Java Programming Language, 4th Edition*
Prentice-Hall, 2006 ISBN: 0201704331

Es el libro de referencia de Java, escrito por los autores del lenguaje. Es la guía indispensable para conocer el lenguaje, tanto a nivel básico como en las características avanzadas que se han incorporado a Java desde su aparición como lenguaje de programación.

A. FROUFE, P. JORGE, *J2ME, Manual de Usuario y Tutorial*
Ra-Ma, 2003 ISBN: 8478975977

Libro que trata la programación de dispositivos móviles presentando los fundamentos de J2ME y la programación de MIDlets. Cubre casi la totalidad de conceptos involucrados en las configuraciones CLDC y CDC y perfiles MIDP 1.0 y MIDP 2.0, desarrollando numerosos ejemplos y aplicaciones, incluso contra contenedores web sirviendo páginas JSP y otros aspectos de Java.

M. HALL, L. BROWN, *CoreServlets and JavaServer Pages*
Prentice-Hall, 2003 ISBN: 0130092290

Libro dedicado al estudio de servlets, realizando una breve incursión en las páginas JSP proporcionando la descripción de las sintaxis de sus elementos y la integración con servlets, está basado en la especificación Servlet 2.4 y JSP 2.0, incluyendo el lenguaje de expresiones, conexión JDBC a bases de datos y la implementación de servlets en Tomcat, Resin y JRun.

P. HELLER, S.ROBERTS, *Complete Java 2 Certification Study Guide*
Sybex, 2003 ISBN: 0782142761

Libro orientado a los exámenes de certificación de *Sun Microsystems* para programadores y desarrolladores. Se centra en las partes que *Sun* ha definido como importantes: operadores, características básicas del lenguaje, palabras clave, *multithreading*, RMI y el diseño de interfaces de usuario utilizando los *layout managers*.

VARIOS, *The Java Tutorial: A Short Course on the Basics*
Prentice-Hall, 2006 ISBN: 0321334205

Libro completamente actualizado a la versión 6 de la plataforma Java 2, escrito por los miembros del equipo de Java. Además de capítulos para tratar las nuevas características de Java, también han reescrito toda la parte básica. También incorporan información para preparar el examen de certificación Java.

Recursos en Internet

http://java.sun.com/

Página oficial de Java de Sun Microsystems. En ella se presenta toda la información actualizada sobre Java, sobre su evolución, desarrollo, estudio, documentación, enlaces, productos, negocios, eventos, etc.

http://java.sun.com/products/jdk

Página oficial de la implementación de Java realizada por *Sun Microsystems*. En ella se encuentran descritos todos los APIs que componen la plataforma Java, en todas sus vertientes. Dispone además de enlaces a las páginas de especificación, descarga y documentación de cada uno de los componentes.

http://jakarta.apache.org/tomcat/

Tomcat es la implementación de Referencia de las tecnologías Java Servlet y *JavaServer Pages*, cuyas especificaciones están desarrolladas por *Sun Microsystems* bajo el *Java Community Process*. En este sitio es posible descargar la implementación de referencia, así como la documentación del API, información adicional y proporciona enlaces a otros sitios de interés.

http://www.jguru.com

Sitio de información sobre Java, en donde destacan por su calidad las diferentes FAQs sobre los más diversos aspectos de la tecnología Java.

http://www.javaworld.com

Fuente inagotable de artículos acerca de Java, algunos de ellos cubren aspectos muy específicos del uso de la tecnología Java.

http://www.mysql.com/

Página oficial de uno de los sistemas de bases de datos SQL Open Source más conocido, utilizado en este libro, que implementa por completo el estándar SQL:2003. Proporciona abundante información y herramientas.

http://www.eclipse.org/

Página oficial de la plataforma de desarrollo Open Source liderada por *IBM*. Es un entorno extensible en su funcionalidad mediante plug-ins y su objetivo es permitir crear aplicaciones, sitios web, EJB, etc., de forma sencilla, integrando todas las herramientas necesarias para realizar cualquiera de estos desarrollos.

http://www.netbeans.org/

Página oficial de la plataforma de desarrollo Open Source liderada por *Sun Microsystems*, que en su versión comercial corresponde a *Sun Java Studio*. Permite desarrollar aplicaciones para cualquier tipo de dispositivo y plataforma, incluyendo EJB, MIDlets, servicios Web, J2EE, etc. También incluye herramientas de comprobación del rendimiento de las aplicaciones.

http://www.google.com/

Recurso imprescindible para cualquier desarrollador.

APÉNDICE B

CONTENIDO DEL CD-ROM

Como complemento a esta publicación, se incluye un CD-ROM que contiene multitud de recursos para la programación Java, junto con el código fuente de todos los ejemplos mostrados en el libro, además de numerosas herramientas software y documentación adicional (en inglés) que serán de gran ayuda al lector.

En esta edición del Tutorial, se ha evitado la reproducción en texto del código completo de los ejemplos que ilustran los distintos conceptos que se introducen en cada uno de los capítulos. Solamente se presentan los trozos de código de dichos ejemplos que son necesarios para la comprensión del texto escrito. En el CD-ROM, el lector podrá consultar el código completo de cada uno de los ejemplos, que podrá imprimir y tener como consulta a la hora de leer las explicaciones en las secciones que correspondan. El lector debe tener en cuenta esta circunstancia, porque el código de los ejemplos en el libro, no está completo en casi ninguno de los casos.

Siempre que ha sido posible, se ha incluido el software del que se habla en el libro, bien en versiones *shareware* o versiones limitadas en tiempo, y por supuesto todas las que están acogidas a licencias de *software libre*. El código fuente de los ejemplos está organizado por capítulos, pero para aprovechar todas las facilidades que puede ofrecer este CD al lector, se han incluido páginas *HTML* para poder navegar a través de su contenido, por lo que se recomienda que utilice un navegador Web para visualizar el contenido del CD y acceder tanto al código fuente de cada uno de los ejemplos, como a la documentación o a los programas de instalación de las herramientas que se incluyen en él.

El documento que debe abrir el lector es "*index.html*", que se encuentra en el directorio raíz del CD. Una vez abierto, aparecerá la página principal, desde donde

será posible seguir todos los vínculos que le conducirán a diferentes páginas, entre las cuales podrá encontrar los ejemplos del código del libro, documentos de especificaciones de Java, documentos HTML, herramientas de desarrollo para lenguaje Java, entornos integrados de desarrollo, entornos para facilitar el uso del JDK, etc.

Para la instalación de la mayoría de los programas, el lector tendrá que abrir la carpeta correspondiente y seguir las instrucciones que aparezcan en pantalla. Hay software que se proporciona en formato comprimido, el cual debe ser expandido previamente a su instalación. Si el lector no tiene instalado ningún programa de descompresión de ficheros, en la carpeta de *Utilidades* del CD encontrará la versión *shareware* del programa ***WinZip***, que le permitirá descomprimir en el directorio que designe el contenido del fichero, y a partir de ahí deberá realizar la instalación del programa.

La figura siguiente muestra una de las páginas de descripción de aplicaciones, en donde puede observar el lector que en la zona izquierda dispone de enlaces a los grupos de aplicaciones y documentos que contiene el CD-ROM, mientras que a la derecha se encuentra la información correspondiente a dicha aplicación. Si dispone de conexión a Internet, el enlace que se proporciona le llevará directamente a la página principal de la aplicación en el sitio Web del fabricante o creador. Los enlaces correspondientes a las plataformas le permitirán instalar la aplicación en su propio ordenador. El logotipo del grano de café con sombrero andaluz, llevará al lector a la página inicial.



Figura B.1

En el CD-ROM se incluye el contenido completo del ***Tutorial de Java*** (en su primera versión, *histórica*) que fue el origen de la primera edición de este libro

Con este apéndice terminan las aventuras de programación con Java. Hemos estado junto al lector desde el comienzo de este libro y esperamos que el viaje le haya resultado ameno. Sobre todo, confiamos en que haya aprendido lo suficiente como para comenzar a realizar sus propias aplicaciones, puesto que se le ha mostrado la suficiente información para ello.

Y recuerde el lector, que el autor está a su disposición en cualquier momento a través de la dirección de correo electrónico `tutorialj2me@yahoo.es`.

Y, por supuesto, gracias por haber hecho este recorrido por el mundo Java con nosotros.

ÍNDICE ALFABÉTICO

A

abstract, 74
Adaptadores, 238
Animación, 463
  Bucle, 466
  Doble buffer, 465
  ImageConsumer, 470
  Imágenes, 469
  ImageProducer, 469
  Intervalo, 466
  Redibujo, 466
  Velocidad, 469
Anotaciones, 39
Anti-patrón, 106
Apache, 539, 608
Applet, 13, 55
  Archive, 61
  Ficheros, 167
  Firma, 169
  Marca APPLET, 56
  Parámetros, 59
  Seguridad, 167
  Signed, 169
Appletviewer, 16, 21, 55
Areas de texto, 281
Arrays, 35, 131, 151
Aserciones, 196
Autoboxing. *Ver* Conversiones Implícitas
Aviso, 50
AWT, 267
  Componentes, 270
  Contenedores, 270
  Escritorio, 324
  Estructura, 269
  Imprimir, 330
  Layouts, 308
  Menús, 301
  Pintar, 417

B

Barra de desplazamiento, 288
Bases de datos, 573
  Consultas, 574
  Formas normales, 574
  Información, 588
  Modelo de conexión, 593
  Modelo relacional, 592
Beans, 337
BorderLayout, 285
Botones, 271
  de comprobación, 273
  de pulsación, 271
  de selección, 272
break, 44
Broadcast, 532

C

CAB, 61
Cadena, 151
Cadenas, 38
Campos de texto, 279
Canvas, 284

Características de Java
Arquitectura neutral, 8
Difundido, 11
Dinámico, 10
Distribuido, 8
Interpretado, 10
Multitarea, 10
Orientado a objetos, 8
Portable, 10
Robusto, 8
Seguro, 9
Simple, 7
Cargador de clases, 9
case, 44
Casting, 33, *Ver* Moldeo
CGI, 538, 600
Clase, 67, 71
Abstracta, 101
Adapter, 232
AlphaComposite, 475
Applet, 20
AWTEventMulticaster, 247
BorderLayout, 310
BoxLayout, 322
Calendar, 351
Canvas, 284
CardLayout, 316
CaretListener, 354
Character, 111
CheckBox, 273
CheckboxMenuItem, 305
Choice, 272
Collection, 133
Color. *Ver* Color
ColorModel, 470
Comparator, 153
Component, 242
Container, 270, 289
DatagramPacket, 520, 534
DatagramSocket, 489, 522, 534
Dialog, 295
DirectColorModel, 472
DragSource, 398
EventSource, 239
File, 159
FilteredImageSource, 472
FlowLayout, 309
Font. *Ver* Fuente de caracteres
FontMetrics, 445
Formatter, 120
Frame, 290
Graphics, 19, 331, 415, 431
GridBagLayout, 315
GridLayout, 313
GroupLayout, 323
HttpServlet, 540
HttpSession, 553
Image, 454
ImageFilter, 472
ImageIcon, 358
IndexColorModel, 471
InetAddress, 491
Iterator, 134
JComponent, 343
JFileChooser, 391
JFormattedTextField, 352
JLabel, 343
JList, 347
JMenu, 362
JMenuItem, 362
JPasswordField, 352
JPopupMenu, 362
JRootPane, 423
JScrollPane, 348, 394
JSlider, 365
JSpinner, 350
JTabbedPane, 387
JTable, 382
JTextArea, 352
JTextField, 352
JTextPane, 352
JTree, 368
JViewPort, 396
Label, 283
List, 275
Listener, 232
Matcher, 119
Math, 109
MediaTracker. *Ver* MediaTracker
MemoryImageSource, 472
Menu, 301
MenuBar, 303
MenuComponent, 301
MenuItem, 302
MenuShortcut, 302
MulticastSocket, 488, 534
NetworkInterface, 494
OverlayLayout, 323
Panel, 298
Pattern, 119
PixelGrabber, 473
PopupMenu, 307
PrintGraphics, 336
PrintJob, 331
PrintStream, 15
Properties, 123
RepaintManager. *Ver* RepaintManager
ResultSet, 586
Runtime, 125
Scanner, 123
Scrollbar, 288
ServerSocket, 489, 508
Source, 232
SpinnerDateModel, 351
SpinnerListModel, 351
SpinnerNumberModel, 351
SplashScreen, 327
SSLServerSocket, 529
SSLSocket, 529
String, 34, 112
StringBuffer, 114
StringTokenizer, 116
System, 126
SystemTray, 328
TextArea, 281
TextComponent, 279

TextField, 279
Thread, 204
Tipos, 73
TitledBorder, 388
Toolkit, 330
TrayIcon, 328
TreeNode, 369
UIManager, 401
URL, 490, 495
URLConnection, 500
URLEncoder, 499
Vector, 131
Window, 290
CLASSPATH, 620
Clipping, 449
Cola de eventos, 253
ColdFusion, 538
Colecciones, 132, 143
AbstractSet, 143
BitSet, 137
Búsqueda, 151
Collection, 133, 142
Comparaciones, 153
Diccionario, 139
Hashtable, 139
List, 132, 143, 144
Listas, 155
Map, 132, *Ver* Map
Mapa, 142
Mapas de sólo lectura, 156
Mapas sincronizados, 156
Operaciones no soportadas, 149
Ordenación, 151
Queue, 132, 142, *Ver* Queue
Set, 132, 142, 144
Singleton, 157
Tipos, 136
Vector, 136
Collator, 155
Color, 450
Canal Alfa, 472
ColorModel, 470
DirectColorModel, 472
HSB, 450
IndexColorModel, 471
Píxel, 470
PseudoColor, 472
RGB, 450
sRGB, 450
TrueColor, 472
Comparable, 147, 153
Comparator, 147
Compilación
Applet, 20
javac, 16
Problemas, 17
Componentes, 270
Componentes solapados, 424
Comunicaciones en red. *Ver* Redes
Comunicaciones Entre Tareas, 219
Comunicaciones seguras, 529
Concurrencia, 224
Constantes, 34, 77
Constructor, 84
Contenedores, 289
Dialog, 293
Frame, 290
Panel, 298
Window, 290
Contexto gráfico, 432
Borrar, 438
Copiar, 438
Creación, 434
Traslación, 439
Control de acceso, 87, 89
package, 87
private, 87
protected, 87
public, 87
Control de flujo. *Ver* Sentencias
Controladores de posicionamiento, 308
Conversiones Implícitas, 96
Cookie, 552, 569
Corba, 639
CSS, 519

D

default, 44
Demonio, 218
Deprecated, 17
Derby, 608
DirectX, 268
DNS, 495
Doble buffer, 423, 430
doclets, 24
DOM, 178, 519
DriverManager, 584

E

Eclipse, 315, 661
EJB. *Ver* Enterprise JavaBeans
Enterprise JavaBeans, 554
Enumeraciones, 133
Escritorio
Barra de tareas, 328
Pantalla de presentación, 326
Etiqueta, 283
Eventos
de acción, 243
de bajo nivel, 242
de foco, 243
de usuario, 245
Fuentes, 232
KeyEvent, 303
KeyListener, 255
Modelo Delegación, 231
Receptores, 232
Swing, 258
Excepciones, 181
ArithmeticExcepcion, 184
Captura, 187
catch, 188
Clasificación, 181
ConcurrentModificationException, 157

Controlador, 37
finally, 189
Generación, 183
IllegalAccessExcepcion, 193
IllegalThreadStateException, 215
InterruptedExcepcion, 185, 186
InterruptedException, 462
IOException, 166
Manejo, 182
Propagacion, 194
RuntimeException, 166
ServletException, 570
SQLException, 585
throw, 191
throws, 192
try, 187
UnknownHostException, 491
UnsupportedOperationException, 143, 150
Exception handler. *Ver* Excepciones
Expresiones, 34
Extensible Markup Language. *Ver* XML

F

Ficheros, 159
Acceso aleatorio, 174
Buffers, 177
Canales, 178
Charsets, 179
Selectors, 179
final, 74, 90
Finalizador, 88
Firefox, 20, 55, 543, 564
FlowLayout, 298
Formulario, 537
GET, 549
POST, 548
Fuente de caracteres, 444
Fuente de eventos, 234, 236
Fuentes de caracteres
JAVA_FONTS, 445
True Type. *Ver* True Type

G

Gráficos
Animación. *Ver* Animación
Arco, 440
Clase Graphics, 431
Clipping. *Ver* Clipping
Componentes lightweight, 420
Contexto gráfico. *Ver* Contexto gráfico
Doble buffer. *Ver* Doble buffer
Línea, 440
Opacidad. *Ver* Opacidad
Optimización, 425
Polígono, 440
Polilínea, 440
Rectángulo, 440
Repintado síncrono, 429
Texto. *Ver* Texto
Transparencia. *Ver* Transparencia

H

Handles. *Ver* Identificadores
Hashtable, 131, 139
Herencia, 86, 93
HTTPS, 531
Hub, 534

I

Identificadores, 24
Imágenes, 449, 454
AlphaComposite, 475
FilteredImageSource, 472
ImageFilter, 472
MemoryImageSource, 472
PixelGrabber, 473
import, 19
Interfaces, 101
Constantes, 106
Declaración, 104
Definición, 104
Implementación, 105
Interfaz de usuario, 268
Internet, 66, 461, 539
Dirección, 493
Proveedor, 493
IPv4, 533
ISAPI, 538
Iterador, 134

J

J2ME, 660
Jakarta-Tomcat, 539
JAR, 62, 107
JarSigner, 171
Java 2 Enterprise Edition. *Ver* J2EE
Java 2 Micro Edition. *Ver* J2ME
Java 2 Standard Edition. *Ver* J2SE
Java Community Process, 661
Java Native Interface. *Ver* Métodos nativos
Java Plug-In, 169
Java Specification Request. *Ver* JSR
Java WebStart, 612
java.nio, 159, 176
JavaBean, 596, 649
JavaBeans, 554, *Ver* Beans
javac, 20
JavaDB, 581
javadoc, 24
javap, 9
JavaServer Pages, 554
jconsole, 654, 656
JCP, 11
JDBC, 382, 555, 660
2 Capas, 578
3 Capas, 578
CallableStatement, 587
Conectividad, 576
Connection, 586
DriverManager, 586
Drivers, 579

Java/Binario, 580
PreparedStatement, 587
Protocolo Independiente, 581
Protocolo nativo, 580
Puente JDBC-ODBC, 579
ResultSet, 587
RowSet, 596
Servlets, 600
Statement, 587
Transacciones. *Ver* Transacciones
JDK 1.1, 258
JFC, 401
JMX
Arquitectura, 649
Instrumentación de recursos, 651
jconsole, 656
Monitorización Local, 657
Monitorización Remota, 658
JNI. *Ver* Métodos nativos
JRun, 660
JSP. *Ver* JavaServer Pages
JSR, 11

L

Layouts, 308
BorderLayout, 310
BoxLayout, 322
CardLayout, 316
FlowLayout, 309
GridBagLayout, 315
GridLayout, 313
GroupLayout, 323
OverlayLayout, 323
Posicionamiento absoluto, 320
Z-Order, 323
Liberación de memoria, 69
List, 133, 144
ArrayList, 144
LinkedList, 144
Listas, 275
Look and Feel, 337, 400
Metal, 403
Ocean, 403
Synth, 405

M

ManagedBean, 649
Map, 145
ArrayMap, 147
HashMap, 146
LinkedHashMap, 146
TreeMap, 147
WeakHashMap, 146
Marimba, 539
MBean, 649
MediaTracker, 461
Menús, 301
Emergentes, 307
Metacaracteres, 118
Metal, 340, 401
Métodos, 67, 77
Argumentos, 78
de instancia, 81
Estáticos, 82
Nombre, 81
Paso de parámetros, 83
Retorno, 79
Sobrecarga, 81
Sobrescritura, 94
varargs, 78
Métodos nativos
Compilar, 480
Ejecución, 484
Fichero de Cabecera, 480
Función C, 481
Librería Dinámica, 482
Librerías del sistema, 484
Microsoft, 62
Middleware, 639
MIDlet, 661
MIME, 550
Modelo MVC, 337
Moldeo, 33
Mono-hilo, 202
Motif, 267
Mozilla, 20, 55, 268
MS-DOS, 22
Multicast, 532
Multihilvanado, 201
Multitarea, 156, 201
Multithreaded. *Ver* Multitarea
MVC, 368, 381
MySQL, 581, 661

N

Netbeans, 315, 661
Netscape, 55, 61
Nimbus, 401
NIO. *Ver* java.nio
NSAPI, 538

O

Object, 97
Objeto, 71
Objetos, 67
Objetos anónimos, 104
Ocean, 340, 401
ODBC, 576
ofuscador de código, 10
Opacidad, 424
Open Source, 564, 661
OpenGL, 268
Opera, 20, 55
Operadores, 28
Aritméticos, 29
Binarios, 28, 29
coma, 45, 46
Condicionales, 30
de asignación, 31
Moldeo, 33
Nivel de bits, 31
Relacionales, 30

Ternario, 32

P

Páginas dinámicas, 537
Palabras
  Clave, 24
  Literales, 25
  Reservadas, 25
  Separadores, 27
Paleta de Colores, 471
Paquete, 90, 91, 107
  import, 108
  Java, 108
Peer, 337
Perl, 119, 537
Pila, 138, 139
pkzip, 62
Plug-In, 168
PolicyTool, 512
Política de seguridad. *Ver* Póliza
Póliza, 168
  PolicyTool, 168
Porter-Duff, 475
PostgreSQL, 581
Preemptivo, 218
private, 90
Programación orientada a objetos, 19
protected, 91
public, 14, 73, 90

Q

Queue, 147

R

RDF, 519
Receptor de eventos, 236
Receptores
  ActionListener, 244
  FocusListener, 244
  ItemListener, 305
  MouseListener, 245
  TextListener, 282
  WindowListener, 245
Reciclador de memoria, 7, 68, 69
Redes
  Agente de seguridad, 509
  Cliente Eco, 502, 523
  Cliente Fecha, 505
  Cliente HTTP, 506
  Dirección IP, 492
  DNS. *Ver* DNS
  FTP, 496
  HTTP, 490
  Internet. *Ver* Internet
  localhost, 493
  Nodo, 487
  Paquetes, 487
  Protocolo, 490
  Servidor Eco, 514
  Servidor HTTP, 517
  Servidor UDP, 526
  SMTP, 490
  Sockets. *Ver* Sockets
  TCP, 488
  UDP, 488
  URL. *Ver* URL
RepaintManager, 427, 429
Resin, 660
RMI
  Comunicación, 627
  Fichero cliente, 623
  Fichero interfaz, 620
  Fichero objeto remoto, 621
  Fichero servidor, 622
  Rendimiento, 632
  Serialización. *Ver* Serialización
  Stub, 618
rmiregistry, 619
Round-Robin, 202
Router, 534
RowSet, 596
  CachedRowSet, 597
  FilteredRowSet, 597
  JdbcRowSet, 596
  JoinRowSet, 597
  WebRowSet, 597
RSA, 529
RSS, 519

S

Scheduler, 218
  Demonio, 219
  Prioridades, 219
  Servicios, 219
Scheduling, 218
Scripts, 537
Seguridad
  cacerts, 172
  JarSigner, 171
  KeyTool, 170, 530
  Plug-In, 169
  PolicyTool, 168, 512
  Política. *Ver* Póliza
  Restricción de accesos, 169
Sentencias, 43
  break, 49
  continue, 49
  de bucle, 45
    do/while, 48
    for, 45
    while, 48
  de salto, 43
    if/else, 43
    switch, 44
  Excepciones, 48
    try/catch, 48
  return, 50
Serialización, 639
  Distribución de objetos, 645
  Seguridad, 646
Servicios Web, 661

Servidor Web, 496
Servlets, 174
  Autenticación de usuarios, 565
  CGI. *Ver* CGI
  Comunicación, 570
  Concurrencia, 568
  Cookies, 569
  Correo Electrónico, 563
  Diccionario, 547
  Eventos, 545
  Filtros, 566
  GET, 549
  JDBC, 600
  JSP. *Ver* JavaServer Pages
  POST, 548
  Seguimiento de sesiones, 552
Set, 144
  ArraySet, 145
  HashSet, 145
  LinkedHashSet, 145
  TreeSet, 145
Sistema de coordenadas, 415
Skin, 405
Sockets, 488, 502
  Datagrama, 488
  Programación, 489
  Raw, 489
  Stream, 488
SQL, 573
  2003, 661
  BLOB, 602
  GROUP BY, 589
  JavaDB, 608
  JOIN, 613
  MEMO, 602
  Portabilidad, 613
  Tipos, 590
SSL, 529, 658
Stack, 131, *Ver* Pila
StarOffice, 268
static, 14
String, 137
Subclase, 89, 94
Sun Creador Studio, 315
Sun Java Studio, 661
super, 92
Superclase, 71, 87, 94
Swing, 268, 337
  Árboles, 367
  Arquitectura de Swing, 340
  Arrastar y soltar, 397
  Barras de progreso, 364
  Bordes, 342
  Botones, 344
  Botones de radio, 345
  Cajas combinadas, 347
  Diálogos predefinidos, 391
  Escalas, 364
  Etiquetas, 343
  Eventos, 258
  Grupos de botones, 345
  Iconos, 357
  JButton, 344
  JTable, 382
  Listas, 347
  Look and Feel. *Ver* Look and Feel
  Menús, 359
  Menús popup, 362
  Nuevos eventos, 261
  Paneles desplazables, 394
  Pestañas, 387
  Pintar, 423
  Selector de ficheros. *Ver* Selector de ficheros
  tablas, 382
  Teclado. *Ver* Teclado
  Texto, 352
  Tool Tips, 354
switch, 44, 49
synchronizable, 74
Synth, 405

T

Tarea, 201
  Agrupación, 211
  Arranque, 209
  Consumidor, 221
  Creación, 206
  Ejecutable, 216
  Estados, 215
  Manipulación, 210
  Monitor, 221
  Monitorización, 223
  Muerta, 217
  Nueva tarea, 215
  Parada, 210
  Productor, 220
  Suspensión, 210
Teclado, 393
Texto, 443
this, 92
Thread
  Métodos, 204
Throwable, 191
Tipos Enumerados, 38
Tipos Genéricos, 50
TLS, 529
Tomcat, 539, 551
Tool Tip, 337
Transacciones, 587
Transparencia, 423
TreeMap, 155

U

UI Delegate, 426
Unicast, 533
Unicode, 111, 448
URL, 490

V

Validación de objetos, 643
Variable, 34, 67
  Ámbito, 75
  Declaración, 75

Nombre, 75
Tipo, 75
Variables de instancia, 76
Variables estáticas, 76
Variables miembro, 74
Vector, 131, 136
Ventanas modales, 298
Verificador de ByteCode, 9

W

W3, 450
Warning. *Ver* Aviso

X

XAWT, 268
XHTML, 519
Xlib, 267
XML, 118, 178, 405, 519
DOM. *Ver* DOM
XSL, 519

Y

Yahoo, 496

Z

ZIP, 61, 107
Z-Order, 301, 323

JAVA™ 2

Manual de usuario y tutorial

5ª edición

TODO LO QUE NECESITA SABER SOBRE LA PLATAFORMA JAVA™ 2. Java ofrece un lenguaje de programación poderoso y flexible, a la vez que sencillo, potente, seguro, eficaz y universal, por lo que constituye el instrumento ideal para el desarrollador actual de aplicaciones.

Este libro le enseñará todo lo que Java puede hacer y cómo hacerlo. Java no está diseñado solamente para realizar *applets* o acceder a Internet. En este libro se describe toda su potencia, que le permitirá sentar las bases para llegar hasta donde nunca antes llegó ningún desarrollador de código.

La obra está estructurada en 21 capítulos; comienza con una introducción sobre el lenguaje Java para enseguida proporcionar información sobre los aspectos más importantes de la plataforma Java 2:

- Fundamentos del lenguaje
- Operadores
- Control del flujo de programación
- Expresiones
- Colecciones de datos
- Clases
- Interfaces
- Paquetes
- Ficheros
- Arquitectura NIO
- Multitarea
- *Scheduling*
- *Tipos genéricos*
- Anotaciones
- Modelo de delegación de eventos
- AWT
- *Swing*
- Gráficos
- Comunicaciones en red
- Sockets TCP/IP, UDP
- *Multicast*
- Comunicaciones seguras
- Servlets, páginas JSP
- JDBC, *Rowset*
- RMI
- JMX, *jconsole*

Se incluye una completa revisión del modelo de delegación de eventos, un estudio básico del desarrollo de ***Servlets***, comunicaciones en red a través de *sockets*, *multicast*, punto-a-punto, comunicaciones seguras *SSL*, acceso a bases de datos mediante **JDBC**, comunicaciones **RMI**, introducción a la mensajería **JMX**, etc. También se tratan las características más importantes aportadas por **Java2 SE 6** a Java: anotaciones, colecciones navegables, ventanas modales, *splash screen*, *system tray*, nueva API de escritorio, arrastrar y soltar, acceso avanzado a redes, uso de **JavaDB**, etc.



Con el libro se adjunta un CD-ROM que contiene el código completo de los más de 300 ejemplos que ilustran los conceptos explicados en el texto, permitiendo probarlos inmediatamente; también incluye herramientas de programación y entornos integrados de desarrollo para plataformas Linux y Windows.

9 788478 978755

ra-ma.es

Ra-Ma®