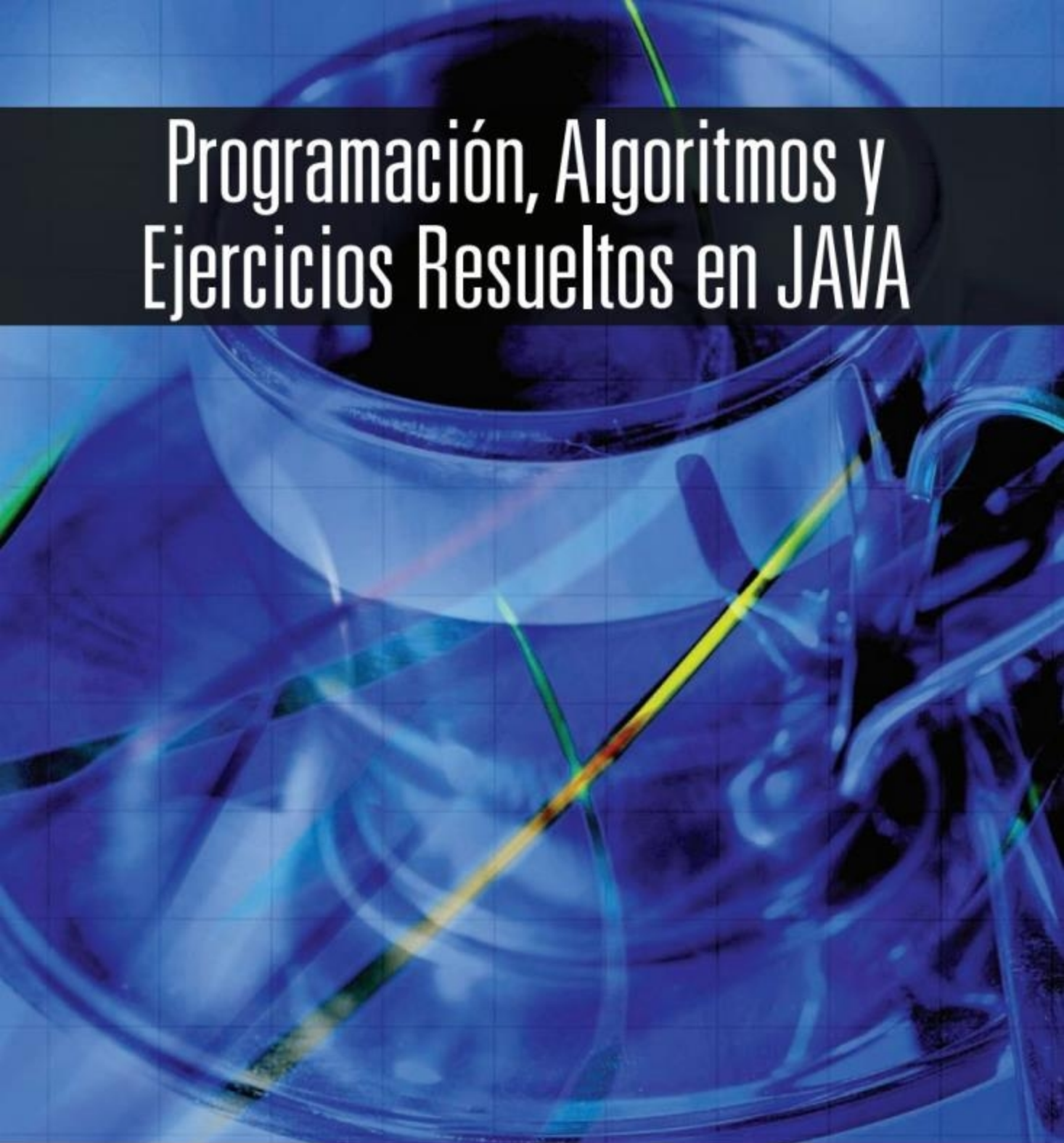




Programación, Algoritmos y Ejercicios Resueltos en JAVA



David Camacho (Coordinador) • José María Valls • Jesús García
José Manuel Molina • Enrique Bueno

Programación, algoritmos y ejercicios resueltos en Java

Programación, algoritmos y ejercicios resueltos en Java

David Camacho Fernández

Coordinador

José María Valls Ferrán

Jesús García Herrero

José Manuel Molina López

Enrique Bueno Rodríguez

Departamento de Informática

Universidad Carlos III de Madrid



Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima
Montevideo • San Juan • San José • Santiago • São Paulo • White Plains

Datos de catalogación bibliográfica

**CAMACHO FERNÁNDEZ, D. (Coordinador); VALLS
FERRÁN, J. M.; GARCÍA HERRERO, J.; MOLINA
LÓPEZ, J. M.; BUENO RODRÍGUEZ, E.**
**PROGRAMACIÓN, ALGORITMOS Y EJERCICIOS
RESUELTOS EN JAVA**
PEARSON EDUCACIÓN, S.A., Madrid, 2003

ISBN: 84-205-4024-2

MATERIA: Programación 519.6

Formato: 195 × 250 mm

Páginas: 424

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de la propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (arts. 270 y sgts. Código Penal).

DERECHOS RESERVADOS

© 2003 por PEARSON EDUCACIÓN, S.A.

Ribera del Loira, 28

28042 Madrid (España)

PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

**CAMACHO FERNÁNDEZ, D.; VALLS FERRÁN, J. M.; GARCÍA HERRERO, J.; MOLINA
LÓPEZ, J. M.; BUENO RODRÍGUEZ, E.**
PROGRAMACIÓN, ALGORITMOS Y EJERCICIOS RESUELTOS EN JAVA

ISBN: 84-205-4024-2

Depósito Legal: M. ??????-2003

Equipo editorial

Editor: David Fayerman Aragón

Técnico editorial: Ana Isabel García Borro

Equipo de producción:

Director: José A. Clares

Técnico: Diego Marín

Diseño de cubierta: Equipo de diseño de PEARSON EDUCACIÓN, S.A.

Composición: JOSUR TRATAMIENTOS DE TEXTOS, S.L.

Impreso por:

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Prólogo	ix
Capítulo 1 Fundamentos de programación en Java	1
1.1. Empezar con Java	1
1.1.1. Un poco de historia	1
1.1.2. Versiones de Java	2
1.1.3. Compilación y ejecución en Java	3
1.2. Fundamentos del lenguaje	6
1.2.1. Tipos básicos	6
1.2.2. Literales y constantes	11
1.2.3. Variables	12
1.2.4. Conversión de tipos	13
1.2.5. Uso básico de cadenas de caracteres	16
1.2.6. Arrays	17
1.2.7. Operadores	25
1.2.8. Control de flujo	30
1.2.9. Entrada/salida básica	38
1.2.10. Conceptos básicos de atributos y métodos	41
1.3. Ejercicios resueltos	45
Capítulo 2 Gestión de errores en Java	59
2.1. Introducción	59
2.2. Tipos de excepciones en Java	60
2.2.1. Gestión de excepciones	61
2.2.2. Clases derivadas de <code>Exception</code>	64
2.3. Sentencias <code>try/catch/finally</code>	64
2.3.1. Múltiples <code>catch</code>	66
2.3.2. Bloques <code>try/catch</code> anidados	69
2.3.3. Sentencia <code>finally</code>	71

2.4.	Sentencias <code>throw</code> y <code>throws</code>	73
2.4.1.	Sentencia <code>throw</code>	73
2.4.2.	Sentencia <code>throws</code>	75
2.5.	Declaración de excepciones propias	77
2.6.	Ejercicios resueltos	79
2.6.1.	Desarrollo de aplicaciones gráficas	79
2.6.2.	Ejercicios de redes	90
Capítulo 3	Algoritmos sobre arrays	101
3.1.	Introducción	101
3.2.	Algoritmos de búsqueda	102
3.2.1.	Búsqueda secuencial	102
3.2.2.	Búsqueda binaria	103
3.2.3.	Análisis de la eficiencia de los algoritmos de búsqueda	103
3.3.	Algoritmos de inserción	104
3.4.	Algoritmos de ordenación	105
3.4.1.	Métodos directos	106
3.4.2.	Métodos avanzados	107
3.4.3.	Medición experimental de la eficiencia	111
3.5.	Ejercicios resueltos	112
3.5.1.	Búsqueda	112
3.5.2.	Inserción	119
3.5.3.	Ordenación. Métodos directos	125
3.5.4.	Ordenación. Métodos avanzados	136
Capítulo 4	Algoritmos recursivos	161
4.1.	Introducción	161
4.2.	Conceptos básicos de recursividad	161
4.3.	Cuándo debe utilizarse la recursividad	164
4.4.	Algoritmos de <i>backtracking</i>	167
4.5.	Ejercicios resueltos	168
Capítulo 5	Programación con ficheros en Java	191
5.1.	Introducción	191
5.1.1.	Conceptos básicos sobre ficheros	191
5.1.2.	Operaciones sobre ficheros	194
5.1.3.	Tipos de ficheros	195
5.2.	Ficheros en Java	196
5.2.1.	Clases básicas para la manipulación de ficheros	196
5.2.2.	Declaración de un fichero en Java	197
5.2.3.	Flujos de entrada/salida en Java	200
5.3.	Gestión de excepciones en ficheros	217
5.4.	Algoritmos sobre ficheros	221
5.4.1.	Algoritmo de mezcla directa	221
5.4.2.	Algoritmo de mezcla natural	223

5.5.	Ejercicios resueltos	225
5.5.1.	Ejercicios básicos sobre ficheros	226
5.5.2.	Ejercicios avanzados sobre ficheros	233
Capítulo 6	Objetos y clases en Java	261
6.1.	Objetos como paradigmas de encapsulación	261
6.1.1.	Redefinición de los objetos	263
6.1.2.	Atributos y métodos	264
6.2.	Clases y objetos en Java	265
6.2.1.	Instancias de una clase. El operador <code>new</code>	266
6.2.2.	Acceso a los miembros de una clase	267
6.2.3.	Miembros públicos y privados	268
6.2.4.	Constructores	268
6.2.5.	Sobrecarga de identificador de funciones	269
6.2.6.	La referencia <code>this</code>	270
6.2.7.	Métodos de comparación	270
6.2.8.	Definición de constantes de clase, constantes de objeto, variables globales y métodos de clase	271
6.2.9.	Interfaces	272
6.3.	Arrays y listas de objetos	273
6.3.1.	Creación de arrays de objetos	273
6.3.2.	Estructuras de datos implementadas mediante arrays	273
6.3.3.	Listas en memoria dinámica	276
6.4.	La herencia	279
6.5.	Clases derivadas	282
6.5.1.	Implementación de clases derivadas	282
6.5.2.	Constructores en clases derivadas	283
6.5.3.	Acceso a los miembros heredados	285
6.5.4.	Métodos heredados y sobreescritos	285
6.6.	Polimorfismo	286
6.7.	Ejercicios resueltos	287
6.7.1.	Ejercicios de definición de objetos	287
6.7.2.	Ejercicios de clases y objetos en Java	291
6.7.3.	Ejercicios de arrays y listas de objetos	309
6.7.4.	Ejercicios de herencia	331
6.7.5.	Ejercicios de polimorfismo	343
Capítulo 7	Diseño de aplicaciones en Java	351
7.1.	Relaciones entre clases	351
7.1.1.	Relaciones jerárquicas y semánticas	351
7.1.2.	Clases derivadas y agregadas. Representación de diagramas de clases	353
7.1.3.	Clases abstractas e interfaces	356
7.2.	Diseño orientado a objetos (DOO)	359
7.2.1.	Diseño estructurado y diseño orientado a objetos	359

7.3.	Ejercicios resueltos	363
7.3.1.	Desarrollo de una aplicación gráfica	363
7.3.2.	Desarrollo de una aplicación de representación de mapas	378
7.3.3.	Desarrollo de un juego basado en personajes	384
7.3.4.	Desarrollo de una aplicación de simulación	393
Referencias bibliográficas		403
Índice analítico		405



Prólogo

Este libro está orientado a aquellas personas que están comenzando en el mundo de la programación. En particular, los autores consideran que se trata de un libro que puede resultar especialmente interesante para alumnos de primeros cursos de Ingeniería, o para aquellas personas que disponiendo de conocimientos de otros lenguajes de programación desean dar el salto a la programación en Java™. Independientemente del lenguaje utilizado, el principal objetivo del libro es el de tratar de desarrollar los conceptos más importantes en el proceso de creación de un programa. Por tanto, el texto hará un especial hincapié en el estudio y análisis de aquellas características de la programación que permiten la implementación de algoritmos capaces de resolver diferentes tipos de problemas.

La elección del lenguaje de programación utilizado para desarrollar el objetivo del libro (Java), presenta varias ventajas e inconvenientes. En primer lugar, y como principales ventajas, pueden enumerarse hechos como que la amplia utilización de este lenguaje en entornos profesionales, y académicos, facilita que una mayor cantidad de personas puedan estar interesadas en el mismo. En segundo lugar, la utilización de técnicas de programación orientada a objetos (POO) permite el desarrollo de programas más robustos y complejos. Además, este paradigma de programación se considera hoy día como básico. Y en tercer lugar, el aprendizaje de la programación, si es mediante la utilización del lenguaje Java, permitirá al lector aprender desde un principio los conceptos básicos de la POO. La facilidad de encontrar documentación, software, información en la Red, etc., también debe tenerse en cuenta como un factor muy positivo para el lector, dado que será más simple poder consultar múltiples fuentes de información y mejorar de esa forma su aprendizaje.

Sin embargo, algunas de las anteriores ventajas pueden transformarse en inconvenientes si se analizan con detenimiento. Por ejemplo, la utilización de la Programación Orientada a Objetos permite desarrollar y crear software de una calidad indudablemente mejor que otros paradigmas de programación, como por ejemplo, la programación estructurada utilizada en lenguajes muy populares como Pascal, Fortran, o C. En particular, si el software creado es muy complejo, debe

ser desarrollado por gran cantidad de personas, y/o mantenido durante una gran cantidad de tiempo. Sin embargo, la mejor calidad de los programas creados tiene un coste. Este coste puede resumirse en el hecho de que es más complejo para una persona que comienza con Java (o que no tiene experiencia con este tipo de paradigma de programación) crear programas utilizando la POO. El uso de Java también tiene un coste en cuanto a la eficiencia de los programas creados, aunque ese aspecto no será abordado en este libro (los programas creados en Java son, generalmente, varias veces más lentos, en tiempo de ejecución, que si son creados en otros lenguajes como C). Otro aspecto importante que podría convertirse en un serio problema, es el de la enorme cantidad de documentación existente: libros de texto, manuales técnicos y de referencia, libros especializados en temas concretos, herramientas disponibles para trabajar con Java, etc. Este problema puede ser estudiado desde dos perspectivas diferentes. En primer lugar, y de forma general, el hecho de que exista una gran cantidad de información no tiene por qué ser algo positivo (este hecho únicamente es positivo cuando se puede encontrar de una forma sencilla y rápida la información que se necesita, si se produce un exceso de información, posiblemente nos veamos desbordados). Si hemos de ser sinceros, hay tal cantidad de documentación en Java que la pregunta más habitual que los profesores de Java suelen responder es: ¿qué libro me recomendarías para empezar?

Otro posible problema, más específico, nos afecta como autores y podría resumirse en tratar de decidir qué aporta de nuevo otro libro más sobre Java. Efectivamente, se pueden contar por docenas (tal vez por centenares) el número de libros publicados (en castellano) que tratan sobre Java. De hecho, muchos de ellos son excelentes libros de referencia, que a diferentes niveles, deberían ser consultados por cualquier persona que desee, o necesite, desarrollar buenos programas en este lenguaje. Algunos de estos libros, considerados de especial utilidad por los autores, aparecen referenciados en la sección de "Referencias bibliográficas". Normalmente la mayoría de los libros publicados de Java podrían clasificarse en dos grandes subconjuntos.

- **Libros básicos, o destinados a la docencia.** Este tipo de libros como por ejemplo: [ArnowWeiss00], [Caro 02], o [Wu 01], tienen como principal objetivo el de enseñar adecuadamente los conceptos más importantes del lenguaje. Suelen hacer desarrollos muy minuciosos y completos de la sintaxis del lenguaje, e introducir la creación de programas mediante la utilización de la POO. Estos libros suelen emplear problemas muy simples para demostrar cada uno de los aspectos que pretenden asimilarse, planteando como problemas a desarrollar por el alumno aquellos que se consideran más complejos.
- **Libros profesionales.** Este tipo de textos suele diferenciarse por la complejidad, o profundidad, que se alcanza en el libro. Es habitual hablar de libros de nivel intermedio como [Naughton 97], o profesionales (niveles avanzados) como [Eckel 02]. También es muy frecuente encontrarse con libros muy especializados en el desarrollo de determinados tipos de problemas, como la creación de aplicaciones gráficas, desarrollo de aplicaciones para bases de datos, programas para Internet (desarrollo de Applets, Servlets/JSP), etc. Estos libros suelen presuponer ciertos conocimientos de programación, que ya se conocen los rudimentos básicos de la POO, o incluso que se tienen conocimientos de Java. Estos textos pueden caracterizarse por hacer un análisis más completo, y profundo, del conjunto de librerías, o paquetes, utilizables en Java. Los ejemplos expuestos suelen ser más complejos, aunque dado que su objetivo no es el docente, debe tenerse en cuenta que no es de esperar que dis-

pongan de un conjunto de ejercicios para ayudar a una mejor comprensión de un determinado tipo de problema. Estos libros, normalmente, no son libros de programación, son muy útiles para entender con profundidad las características concretas del lenguaje, pero desde luego no se trata de textos que hayan sido diseñados para la enseñanza de la programación a cualquier persona.

Además, debe tenerse en cuenta que existen miles de manuales y otro tipo de documentos relacionados con Java que pueden ser descargados desde Internet. Por tanto, tal vez se debería intentar responder a la siguiente pregunta: ¿qué puede aportarnos un libro, como el presente, frente a la enorme cantidad de documentación ya existente? Las ideas que han tratado de desarrollarse en este libro, podrían enumerarse de la siguiente forma:

1. En primer lugar, y a pesar de la enorme cantidad de documentación, es difícil encontrar libros de Java donde el enfoque de los mismos esté orientado hacia la resolución de un conjunto de problemas concretos. Los autores creen que disponer de un texto donde, además de encontrar los aspectos fundamentales de la programación, cualquier lector pueda encontrar un conjunto de ejercicios de diferentes complejidades resueltos puede resultar muy útil en el proceso del aprendizaje del lenguaje considerado. Esto es debido a que los autores creen que la mejor forma de aprender a programar (se trate del lenguaje que se trate), es mediante la práctica en la resolución de diferentes ejercicios.
3. En segundo lugar, el libro no pretende ser exhaustivo en todos los aspectos del lenguaje, ni siquiera textos profesionales de muy reconocida calidad como [Eckel02] lo logran. Esto es debido a la enorme complejidad de Java. Los autores han tratado de resumir y condensar todos los aspectos teóricos de importancia (indicando otras fuentes bibliográficas donde el lector podrá profundizar en cada uno de esos temas), buscando un enfoque eminentemente práctico.
4. Los aspectos de programación, incluyendo el análisis del problema, planteamiento de las técnicas algorítmicas y definición de estructuras de datos más adecuadas, tienen un peso fundamental en este libro. Así, el desarrollo de todos los elementos del lenguaje se presenta como una aplicación práctica a problemas concretos, particularizando finalmente su implementación a una codificación en Java.
5. Por último, este libro pretende estudiar con cierta profundidad diversos temas básicos, desde dos puntos de vista diferentes. En primer lugar, desde el punto de vista del lenguaje como la sintaxis, tratamiento de errores, fichero, o los principios fundamentales de la POO. Y en segundo lugar, desde el punto de vista de la programación, estudiando diferentes algoritmos para la ordenación o búsqueda de elementos, algoritmos recursivos, etc.

Con todo lo anterior, este libro podría clasificarse como un libro docente con un enfoque eminentemente práctico, donde lo principal es aprender a programar mediante el uso de ejercicios prácticos que están resueltos y que pueden ser consultados a posteriori por el lector. Se pretende superar el enfoque de la mayoría de los manuales, restringido a una revisión de los aspectos concretos del lenguaje, para presentar una visión aplicada de las principales técnicas de programación. Desde las principales técnicas algorítmicas clásicas, hasta los diseños basados en jerarquías de clases, todas aparecen orientadas a la resolución de ejercicios prácticos.

El libro consta de siete capítulos, en cada uno de los cuales puede encontrarse un conjunto de ejercicios que son planteados al lector. El grado de complejidad de estos ejercicios puede clasificarse en varias categorías:

- **Ejercicios simples.** Son utilizados para mostrar aquellas características que se consideran esenciales.
- **Ejercicios propuestos.** Al final de cada capítulo se puede encontrar un conjunto de ejercicios que permitirán aplicar de forma práctica los conceptos teóricos estudiados en el mismo.
- **Propuestas de prácticas.** Se trata de un conjunto de ejercicios que pueden ser utilizados (de hecho, han sido utilizado por los autores) como ejercicios de laboratorio en asignaturas de programación.
- **Ejercicios de examen.** Este conjunto de problemas ha sido utilizado en exámenes de diferentes asignaturas de programación. Pueden utilizarse para ver aplicaciones concretas de algunos de los algoritmos explicados en la teoría.

A continuación, se describe muy brevemente el contenido de cada uno de los capítulos que forman el libro:

- En el Capítulo 1 se realiza una breve introducción a los principios básicos de Java. Inicialmente se muestra cómo se crean, compilan y ejecutan programas en este lenguaje, para a continuación, describir la sintaxis y los rudimentos básicos del lenguaje.
- El Capítulo 2 estudia y analiza cómo se realiza la gestión y el control de errores en Java, denominados "excepciones". Se analizan en detalle todas las posibilidades del lenguaje, y se proporciona un conjunto de problemas complejos (que emplean paquetes como `java.awt`, `java.applet` o `java.net`) donde el lector puede observar cómo puede utilizar los conceptos adquiridos para crear programas robustos. Es decir, programas que no terminen de forma abrupta o inesperada por la aparición de un error inesperado.
- En el Capítulo 3 se estudian las que quizá sean las actividades más fundamentales utilizadas en computación: la búsqueda y la ordenación. Son algoritmos básicos que se han aplicado a arrays de números enteros, aunque, al comprender su funcionamiento, pueden aplicarse sin gran dificultad a otras estructuras de datos más complejas. Junto a los algoritmos fundamentales de búsqueda y ordenación se han incluido algunos algoritmos de inserción en arrays. Se han incorporado ejercicios donde se hacen seguimientos detallados del funcionamiento de casi todos los algoritmos con arrays concretos.
- El Capítulo 4 estudia un tema que, con toda seguridad, es uno de los más difíciles de entender por los estudiantes de computación: la recursividad. Esta técnica de programación permite resolver de forma elegante y sencilla problemas de gran complejidad. Se explican los fundamentos teóricos de la recursividad y se resuelven ejercicios de diverso grado de dificultad. Entre ellos incorporamos problemas clásicos como las torres de Hanoi o el salto del caballo.
- En el Capítulo 5 se realiza un estudio con detalle del tratamiento de ficheros en Java. Aunque inicialmente se describe de forma general (independientemente del lenguaje considerado) el concepto y las operaciones habituales sobre ficheros, la siguientes secciones del

capítulo enfocan la creación y gestión de los mismos sobre Java. El tratamiento de este tipo de estructuras de información, y el de la Entrada/Salida (E/S) en general, viene encapsulado por un conjunto de clases Java pertenecientes al paquete `java.io`. Dado que se trata de un tema de cierta complejidad no se analizan todas las clases posibles, sino sólo aquellas que más frecuentemente son utilizadas para resolver problemas básicos en el tratamiento de ficheros, y de E/S, en Java. Este capítulo pretende reunir diversos conceptos como la gestión de ficheros con tipo, ficheros de texto, la serialización de objetos (almacenamiento de objetos sobre ficheros), o algunos algoritmos de ordenación de ficheros en un único tema, que normalmente suelen aparecer de una forma bastante dispersa en otros libros de texto.

- En el Capítulo 6 se presenta la creación de clases como estructuras apropiadas para representar los elementos que aparecen en aplicaciones reales. Aparecen nuevos tipos de datos creados por el programador que permiten agrupar información de distintos tipos básicos, como enteros, reales, caracteres, etc., e incluso objetos de otras clases agregadas, desarrollándose además los algoritmos apropiados para trabajar con esos tipos de datos. La combinación de diversas informaciones (atributos) en una única variable, unificando el uso de todas ellas a través de una interfaz pública que proporciona una serie de servicios constituye el concepto básico de *objeto*. En este capítulo se profundizará acerca de estos conceptos, presentando además la abstracción y encapsulamiento como metodologías eficaces para tratar la complejidad y desarrollar programas mejor estructurados y fácilmente mantenibles.
- En el Capítulo 7 se da el paso desde el diseño de clases aisladas hacia el desarrollo de programas que contengan estructuras de clases interrelacionadas, con el objeto de abordar problemas de mayor complejidad. Se pretende aprovechar los mecanismos de herencia y polimorfismo para programar, de manera que el código sea encapsulado, ocultando los datos irrelevantes a cada nivel de abstracción, y dicho código sea al mismo tiempo potente, flexible y fácilmente reutilizable. Se presentan una serie de aplicaciones de complejidad intermedia, que sirven para ilustrar el diseño con estructuras de clases interrelacionadas.

Marzo, 2003

David Camacho Fernández
José María Valls Ferrán
Jesús García Herrero
José Manuel Molina López
Enrique Bueno Díaz

Departamento de Informática
Universidad Carlos III de Madrid

Fundamentos de programación en Java

1.1. Empezar con Java

Java es ante todo un lenguaje de programación moderno. Fue diseñado en la década de los noventa, y eso se nota en cuanto uno empieza a trabajar con él, nos proporciona una potencia, una robustez y una seguridad que muy pocos lenguajes pueden igualar, sin olvidar su rasgo más conocido: es totalmente portable. Todas estas características y otras que iremos descubriendo a lo largo de estas páginas, hacen de Java un lenguaje necesario que cubre un hueco enorme en el panorama de la programación moderna.

Si bien Java tiene su base en lenguajes como C y C++, los supera con creces y sería un error pensar que es una simple evolución de éstos. Java tiene entidad propia y características novedosas y potentes que hacen de él no sólo una apuesta de futuro, sino también un lenguaje con un presente claro y asentado. No obstante, toda la potencia que Java proporciona tiene un coste; es necesario asimilar muchos conceptos, técnicas y herramientas que en muchos casos son totalmente nuevas y hacen que la curva de aprendizaje sea pronunciada. Sin embargo, una vez superados los primeros escollos, los resultados son espectaculares y merece la pena el esfuerzo.

1.1.1. Un poco de historia

En 1991 un grupo de ingenieros de Sun Microsystems liderados por Patrick Naughton y James Gosling comienza el desarrollo de un lenguaje destinado a generar programas independientes de la plataforma en la que se ejecutan. Su objetivo inicial nada tiene que ver con lo que hoy en día es Java, sus creadores buscaban un lenguaje para programar los controladores utilizados en la electrónica de consumo. Existen infinidad de tipos de CPU distintas, y generar código para cada una de ellas requiere un compilador especial y el desarrollo de compiladores es caro.

Después de dieciocho meses de desarrollo aparece la primera versión de un lenguaje llamado OAK que más tarde cambiaría de nombre para convertirse en Java. La versión de 1992 está ampliada, cambiada y madurada, y a principios de 1996 sale a la luz la primera versión de Java. Los inicios son difíciles, no se encuentran los apoyos necesarios en Sun y el primer producto que sale del proyecto, un mando a distancia muy poderoso y avanzado, no encuentra comprador. Pero el rumbo de Java cambiaría debido a una tecnología completamente ajena a los controladores de electrodomésticos: Internet.

Mientras Java se estaba desarrollando, el mundo de las comunicaciones crecía a una velocidad de vértigo, Internet y principalmente el mundo World Wide Web dejaban los laboratorios de las universidades y llegaban a todos los rincones del planeta. Se iniciaba una nueva era y Java tuvo la suerte de estar allí y aprovechar la oportunidad. En 1993 con el fenómeno Internet en marcha, los desarrolladores de Java dan un giro en su desarrollo al darse cuenta de que el problema de la portabilidad de código de los controladores es el mismo que se produce en Internet, una red heterogénea y que crece sin parar, y dirigen sus esfuerzos hacia allí. En 1995 se libera una versión de HotJava, un navegador escrito totalmente en Java y es en ese mismo año cuando se produce el anuncio por parte de Netscape de que su navegador sería compatible con Java. Desde ahí otras grandes empresas se unen y Java se expande rápidamente.

No obstante, las primeras versiones de Java fueron incompletas, lentas y con errores. Han tenido que pasar varios años de desarrollo y trabajo para que Java sea un lenguaje perfectamente asentado y lleno de posibilidades. Actualmente es ampliamente utilizado en entornos tanto relacionados con Internet como completamente ajenos a la Red.

El mundo Java está en constante desarrollo, las nuevas tecnologías surgen y se desarrollan a gran velocidad haciendo de Java un lenguaje cada día mejor y que cubre prácticamente todas las áreas de la computación y las comunicaciones, desde teléfonos móviles hasta servidores de aplicaciones usan Java.

1.1.2. Versiones de Java

En algunos momentos, dado lo cambiante de la tecnología Java y sobre todo por la ingente cantidad de siglas que aparecen relacionadas directa o indirectamente con ella, surgen confusiones a raíz de las denominaciones de los productos o incluso sobre las versiones de los mismos.

Java ha cambiado a lo largo de los años, habiéndose liberado a día de hoy cerca de cuarenta versiones del entorno de desarrollo y de ejecución. Todas las versiones se pueden agrupar en tres grandes grupos. Cada uno de estos grupos representa un salto cuantitativo y cualitativo del producto.

- **Java 1.0.** Como se ha comentado anteriormente, la primera versión de Java se libera en 1996, nace la versión 1.0. Dos meses después aparece la versión 1.02, solucionando algunos problemas. Es el inicio del lenguaje y en estos momentos poco más que un sencillo applet era posible hacer con éste.
- **Java 1.1.** La siguiente revisión importante es la 1.1; el lenguaje empieza a tomar forma, la biblioteca de clases que acompaña al lenguaje es cada vez más completa.

- **Java 2.** Es en 1998 cuando Java da un verdadero paso adelante con la aparición de la versión 1.2; con ella nace Java 2 y es cuando el lenguaje se estabiliza definitivamente.

Dentro de Java 2 se han liberado hasta el momento tres grupos de versiones, la propia 1.2, la 1.3 y recientemente la 1.4. Cada una de ellas proporciona un pequeño avance sobre lo definido por Java 2, soluciona problemas, incrementa la velocidad y sobre todo hace crecer la biblioteca de clases. En este momento el lenguaje está estable y todos los esfuerzos se centran en ampliar las bibliotecas y proporcionar nuevas API para controlar, tratar o manejar cualquier tipo de dispositivo, dato o estructura imaginable.

Existe también cierta confusión con la denominación de los productos o tecnologías relacionados con Java. El entorno de desarrollo y ejecución de Java utilizado en este libro es el Java 2 Standard Edition (J2SE), que permite la creación de programas y de applets y su ejecución en la máquina virtual Java. En el momento de escribir este libro, la última versión estable de Java es la J2SE 1.4.1.

1.1.3. Compilación y ejecución en Java

El proceso de compilación y ejecución en Java requiere de la utilización de dos componentes del entorno de desarrollo; por un lado debemos compilar el código java y por otro debemos ejecutar el programa generado. En otros lenguajes de programación el resultado de la compilación es directamente ejecutable por el sistema operativo; pero en Java, el resultado de la compilación es un código que no es ejecutable por un procesador concreto, es un código que es interpretado por una máquina virtual que lo hace totalmente independiente del hardware en el que se ejecute ésta máquina virtual. Este código se denomina normalmente *bytecode* y la máquina virtual es conocida como JVM.

El *bytecode* generado por la compilación de un programa en Java es exactamente igual, independientemente de la plataforma en la que se ha generado, y por ello es posible, por ejemplo, compilar un programa en una máquina Sun basada en tecnología Sparc y después ejecutar el programa en una máquina Linux basada en tecnología Intel. En el proceso de ejecución es la JVM la que toma ese *bytecode*, lo interpreta y lo ejecuta teniendo en cuenta las particularidades del sistema operativo. El resultado: total portabilidad del código que generamos. Evidentemente esta capacidad de Java lo hace muy útil en entornos de ejecución heterogéneos como es Internet.

El conjunto de herramientas utilizadas en la compilación de un programa Java se conocen genéricamente como SDK (*Software Development Kit*) o entorno de desarrollo; la máquina virtual Java y todas las clases necesarias para la ejecución de un programa se conoce como JRE (*Java Runtime Environment*). El SDK de Java contiene el JRE, no tendría sentido poder compilar un programa y no poder ejecutarlo después; pero el JRE se distribuye por separado, existen personas que sólo quieren ejecutar programas y no les interesa el desarrollo.

Existen versiones del SDK para plataformas Solaris, Windows, Linux y Mac principalmente, pero es posible encontrar máquinas virtuales en otros muchos sistemas operativos, recordemos que Java se diseñó pensando en ejecutar programas en entornos heterogéneos.

Compilación

El proceso de compilación en Java es similar al de otros lenguajes de programación; la principal diferencia es que en lugar de generar código dependiente de un determinado procesador, como haría un compilador de C++ por ejemplo, genera código para un procesador que no existe realmente, es virtual: la JVM.

Una característica del SDK de Java que sorprende a los programadores que se topan la primera vez con Java es que el SDK de Java no tiene un entorno gráfico, ni tan siquiera un entorno para editar los programas y compilarlos, todo se hace desde la línea de comandos, a la manera tradicional. Esto al principio es engorroso e incómodo pero tiene un beneficio claro: sólo debemos preocuparnos por conocer Java, no es necesario gastar tiempo en entender un entorno de desarrollo complejo antes de incluso saber escribir nuestro primer programa. Con el tiempo, y cuando dominemos el lenguaje, podremos elegir entre los entornos gráficos y no gráficos para desarrollar en Java. Para escribir un programa en Java, incluso con los gráficos más complejos, sólo es necesario un editor de texto y el SDK.

Para poder compilar nuestro primer programa en Java necesitamos escribirlo; y como no podía ser de otra forma será sencillo y sólo mostrará un mensaje de bienvenida a la programación en Java. A pesar de que el programa es de sólo cinco líneas, encierra conceptos importantes, la mayoría de ellos deben ser explicados posteriormente; pero vamos a ver por encima la estructura del programa sabiendo que algunos puntos del programa quedarán oscuros en este momento.

```
public class Inicio {  
    public static void main(String [] args){  
        System.out.println("¡Bienvenido a Java!");  
    }  
}
```

En primer lugar debemos escribir el programa con nuestro editor de textos favorito y guardarlo en un fichero con el nombre `Inicio.java`. Es necesario que el nombre del fichero coincida con el de la clase siempre que ésta sea una `public`. Dentro de un fichero `.java` (unidad de compilación) pueden aparecer tantas clases como queramos; pero sólo una de ellas puede ser `public`.

En Java todo son clases y si queremos hacer un programa que sólo escriba un mensaje por pantalla debemos escribir una clase, en nuestro caso la clase se llama `Inicio`. Es importante resaltar que Java es sensible a mayúsculas y minúsculas, y por tanto, la clase `Inicio` es distinta de la clase `inicio`. Una clase se define utilizando la palabra reservada `class` y comprende todo el código encerrado entre las dos llaves más externas.

Dentro de la clase `Inicio` vemos el método `main`. Este método es especial. Por él comienza la ejecución de cualquier programa en Java, siempre tiene esta estructura y es conveniente respetarla para evitar problemas. Sin entrar en muchos detalles vamos a comentar brevemente cada una de las partes del método `main`.

- **public**. Indica que el método es público y que puede accederse desde fuera de la clase que lo define. El método `main` tiene que ser invocado por la máquina virtual Java, por lo que debe ser público.

- **static.** Indica que el método es estático y que no es necesario que exista una instancia de la clase `Inicio` para poder ser ejecutado. Esto también es necesario ya que como hemos dicho el método es llamado desde la JVM.
- **void.** Informa al compilador de que el método `main` no devuelve ningún valor tras su ejecución.
- **String[] args.** Define un parámetro del método `main`. Utilizaremos los parámetros para enviar y recoger información de los métodos. En este caso `args` contendrá los posibles parámetros de la línea de comando de la ejecución de la clase `Inicio`.

Lógicamente toda esta información es confusa en este momento, no se preocupe, no es necesario comprender totalmente todos estos conceptos para continuar. Algunas veces, es necesario introducir conceptos que no es posible explicar hasta más adelante con el fin de poder continuar. En este momento, es necesario escribir un programa en Java sin saber nada de Java. Es lógico que no se entienda todo.

Por último, dentro del método `main` nos encontramos una línea de código que será ejecutada cuando se invoque el método. Nos limitaremos a decir que esa línea permite imprimir en la salida estándar, normalmente el monitor de nuestro computador, el mensaje “¡Bienvenido a Java!”. Para ello utiliza el método `println` del objeto `out` de la clase `System`. Una vez más todo esto se explicará más adelante.

Para compilar utilizaremos el compilador `javac`. La ejecución de `javac` dependerá del sistema operativo en el que estemos trabajando, pero en general se realizará desde la línea de comandos de nuestro sistema operativo.

En Windows

```
C:\> javac Inicio.java
```

En Solaris/GNU Linux/Unix

```
$ javac Inicio.java
```

Si no se encuentran errores en la compilación, el resultado de ésta será un fichero con el bytecode correspondiente a la compilación de `Inicio.java`; este resultado se almacena en un fichero con extensión `.class` que tiene como nombre el mismo que el fichero fuente. Por tanto, en nuestro directorio tendremos un fichero llamado `Inicio.class`. Ya tenemos compilado nuestro primer programa en Java, ahora tenemos que ejecutarlo.

Ejecución

La ejecución de un programa Java involucra a la máquina virtual que es la encargada de interpretar y ejecutar cada una de las instrucciones (bytecode) contenidas en el fichero `.class`.

Para ejecutar el programa tenemos que utilizar el entorno de ejecución de Java (*Java Runtime Environment*, JRE). El JRE nos permite ejecutar el bytecode de nuestros programas en la máquina virtual Java. Para ejecutar un programa en Java tenemos que invocar el entorno de ejecución pasándole como parámetro el nombre de la clase que queremos ejecutar.

En Windows

```
C:> java Inicio
```

En Solaris/GNU Linux/Unix

```
$ java Inicio
```

Si todo ha ido bien veremos el resultado de la ejecución después de la ejecución del programa. Es necesario que nos encontremos en el mismo directorio que contiene el fichero `.class` para que la JVM lo encuentre. También es importante recordar que no debemos poner la extensión del fichero, ya que no estamos indicando el nombre de un fichero sino el nombre de una clase, la máquina virtual Java se encargará de encontrar el fichero con el bytecode.

1.2. Fundamentos del lenguaje

Cuando se comienza a estudiar un nuevo lenguaje de programación es necesario ver dos bloques fundamentales. Por un lado necesitamos conocer qué datos es capaz de manejar, qué posibilidades de manejo de esos datos nos proporciona y por otro lado, qué herramientas para controlar la ejecución y la interacción con el usuario nos ofrece. En esta sección vamos a ver todo esto, pero al contrario que en otros lenguajes, en Java no haremos nada más que empezar a ver sus posibilidades. Tendremos que llegar a conocer y dominar otros muchos elementos, relacionados con la programación orientada a objetos la mayoría, antes de poder decir que lo conocemos.

Comenzaremos viendo la forma de representar y manejar la información en Java, después descubriremos cómo podemos controlar el flujo de ejecución de nuestros programas y finalizaremos con unos breves conceptos de entrada/salida a consola y métodos.

1.2.1. Tipos básicos

Java es un lenguaje fuertemente tipado, todas las variables que se definen tienen un tipo declarado y este tipo es controlado y comprobado en todas las operaciones y expresiones. A pesar de que en algunos momentos tantas comprobaciones pueden ser un poco frustrantes, éstas hacen de Java un lenguaje muy poco propenso a errores exotéricos, derivados de un tipo mal expresado o incorrectamente usado, que en otros lenguajes se producen con relativa frecuencia.

Disponemos de ocho tipos básicos divididos en cuatro bloques dependiendo de su naturaleza. En un principio puede parecer que un lenguaje tan completo como Java debería tener más tipos de datos, pero estos ocho tipos cubren perfectamente las necesidades de representación de la información, ya que sirven de base para crear estructuras más complejas y potentes.

Los bloques en los que se encuentran divididos los tipos básicos en Java son los siguientes:

- Enteros. Son cuatro tipos que nos permiten representar números enteros.
- Coma flotante. Son dos tipos usados para representar datos reales.

- Caracteres. Un tipo que nos permite representar caracteres de cualquier idioma mundial.
- Lógicos. Un tipo para representar valores lógicos.

A diferencia de lo que ocurre en otros lenguajes, los tipos básicos en Java siempre tienen los mismos tamaños y capacidades, independientemente del entorno en el que estemos trabajando. Esto garantiza que no será necesario comprobar la arquitectura en la que nos encontramos para decidimos por un tamaño de entero o por otro, un tipo `int` tendrá 32 bits en un PC y en una estación Sun.

Enteros

Los números enteros en Java son siempre con signo, no existen tipos enteros sin signo ni modificadores para eliminarlo. Los cuatro tipos enteros: `byte`, `short`, `int` y `long`, se muestran en la Tabla 1.1. con su tamaño y su rango de valores representables.

Tabla 1.1. Tipos enteros.

Nombre	Tamaño	Rango
<code>long</code>	64 bits	-9,233,372,036,854,775,808L a 9,233,372,036,854,775,807L
<code>int</code>	32 bits	-2,147,483,648 a 2,147,483,647
<code>short</code>	16 bits	-32,768 a 32767
<code>byte</code>	8 bits	-128 a 127

A diferencia de lo que ocurre en otros lenguajes, los tipos básicos en Java siempre tienen las mismas capacidades de almacenamiento, independientemente del entorno en el que estemos trabajando. Esto garantiza que no será necesario comprobar la arquitectura en la que nos encontramos para decidimos por un tamaño de entero o por otro.

Por defecto las constantes enteras son de tipo `int`. Si queremos forzar que una constante de tipo entero sea tomada como un `long` debemos añadir al final una L.

Coma flotante

Los dos tipos utilizados en Java para representar valores reales son: `float` y `double`, en la Tabla 1.2. podemos ver sus características de almacenamiento.

El tipo `double` es, generalmente, más usado que el `float`, pero éste es un poco más rápido en las operaciones y ocupa menos, por lo que puede ser muy útil en algunas circunstancias en las que la velocidad de cálculo sea prioritaria.

Tabla 1.2. Tipos en coma flotante.

Nombre	Tamaño	Rango
float	32 bits	$\pm 3.40282347\text{E}+38\text{F}$
Double	64 bits	$\pm 1.79769313486231570\text{E}+308$

Al igual que las constantes enteras son por defecto de tipo `int`, las constantes reales son por defecto de tipo `double`. Podemos forzar un tipo `float` si añadimos al final del número una `F`.

Los tipos `float` y `double` disponen de tres valores especiales: infinito positivo, infinito negativo y NaN (*Not a number*). Estos valores nos permiten representar situaciones como desbordamientos y errores.

```
public class Rangos {
    public static void main(String [] args){
        System.out.println(Math.sqrt(-1));
        System.out.println(1.1e200*1.1e200);
        System.out.println(-1.1e200*1.1e200);
    }
}
```

La ejecución del programa anterior da como resultado la impresión de los tres valores especiales. El método `sqrt()` de la clase `Math` nos permite calcular la raíz cuadrada de un número.

```
NaN
Infinity
-Infinity
```

Caracteres

En Java los caracteres no se almacenan en un byte como en la mayoría de los lenguajes de programación. En Java se usa Unicode para representar los caracteres y por ello se emplean 16 bits para almacenar cada carácter. Al utilizar dos bytes para almacenar cada carácter, disponemos de un total de 65.535 posibilidades, suficiente para representar todos los caracteres de todos los lenguajes del planeta. El estándar ASCII/ANSI es un subconjunto de Unicode y ocupa las primeras 256 posiciones de la tabla de códigos, con lo que es posible la compatibilidad entre los dos sistemas de representación.

Muchas veces es necesario representar caracteres en forma de constante. En Java las constantes de tipo carácter se representan entre comillas simples. Existen secuencias de escape para representar algunos caracteres especiales como el retorno de carro, el tabulador, etc. como se muestra en la Tabla 1.3.

Tabla 1.3 Secuencias de escape.

Secuencia	Descripción
<code>\b</code>	Retroceso
<code>\t</code>	Tabulador
<code>\r</code>	Retorno de carro
<code>\n</code>	Nueva línea
<code>\'</code>	Comilla simple
<code>\"</code>	Comilla doble
<code>\\</code>	Barra invertida

Podemos también expresar caracteres a través de su código Unicode o su código ASCII/ANSI tradicional. Para ello utilizamos `'uxxxx'` donde `xxxx` es el código Unicode del carácter en hexadecimal. También podemos utilizar el código ASCII/ANSI en octal de la forma `'ooo'` o en hexadecimal con la expresión `'0xnn'`.

```
'A'    '\u0041'    '\0x41'    '\101'
```

Lógicos

En Java existe un tipo especial para representar valores lógicos, el tipo `boolean`. Este tipo de datos sólo puede tomar dos valores: verdadero y falso. El tipo `boolean` se emplea en todas las estructuras condicionales y es el resultado de las operaciones realizadas utilizando operadores relacionales.

Existen dos palabras reservadas para representar los valores lógicos, `true` y `false`, que pueden utilizarse libremente en los programas. A diferencia de otros lenguajes, el tipo `boolean` es un tipo distinto de los demás y, por tanto, incompatible con el resto. Java es rígido en esto y si espera un tipo `boolean` no aceptará un `int` en su lugar.

```
boolean b;  
  
b=true;  
if (b) System.out.println("Es cierto");
```

Envoltorios

En Java todo son clases y objetos, excepto los tipos básicos. Esto hace que en algunas circunstancias tengamos que convertir estos tipos básicos en objetos. Para realizar esta conversión utili-

zamos envoltorios que recubren el tipo básico con una clase, a partir de este momento el tipo básico envuelto se convierte en un objeto.

Existen nueve envoltorios para los tipos básicos de Java, como se puede ver en la Tabla 1.4., cada uno de ellos envuelve un tipo básico y nos permite trabajar con objetos en lugar de con tipos básicos.

Tabla 1.4 Envoltorios.

Tipo	Envoltorio
int	Integer
long	Long
float	Float
double	Double
short	Short
byte	Byte
char	Character
boolean	Boolean
void	Void

Una de las razones más importantes para utilizar envoltorios es poder emplear las clases de utilidad que Java proporciona en su biblioteca de clases. Estas clases necesitan utilizar objetos para funcionar y no aceptan tipos de datos básicos. Si queremos por ejemplo crear una pila de números reales, tendremos que envolver los números reales para tener objetos que poder guardar en la pila.

En los envoltorios existen algunos métodos que nos permiten convertir cadenas de caracteres en tipos básicos. Así, podemos convertir la cadena "123" en el número entero 123 utilizando el método `parseInt()` de la clase `Integer`.

```
int num=Integer.parseInt("123");
```

En Java 2, dentro de cada clase envoltorio, excepto `Boolean`, `Character` y `Void`, existe un método `parse` que nos permite convertir una cadena en el tipo básico correspondiente.

```
public class Envoltorios {  
    public static void main(String [] args){  
        System.out.println(Integer.parseInt("124"));
```



```

System.out.println(Long.parseLong("165"));
System.out.println(Byte.parseByte("21"));
System.out.println(Short.parseShort("45"));
System.out.println(Float.parseFloat("45.89"));
System.out.println(Double.parseDouble("1.5e8"));
}
}

```

Los tipos básicos envueltos por las clases proporcionadas por Java son inmutables, es decir, que no pueden modificar su valor sin destruir el objeto. Existen situaciones, como se verá más tarde, en las que es necesario cambiar este comportamiento y deberemos definir nuestros propios envoltorios.

1.2.2. Literales y constantes

Un literal es un dato que se considera constante y que es expresado de diferentes formas dependiendo de su naturaleza con el fin de evitar ambigüedades. En Java disponemos de tantos tipos de literales como tipos de datos básicos. Además podemos escribir literales de tipo cadena de caracteres, que internamente Java convierte al tipo `String`, utilizado en Java para manejar las cadenas. Todos los tipos de literal se encuentran en la Tabla 1.5.

Tabla 1.5. Tipos de literal.

Tipo	Literal	Comentarios
<code>int</code>	123	Todos los enteros por defecto son <code>int</code>
<code>long</code>	123L	Es necesario indicar una L
<code>char</code>	'a'	Comillas simples
<code>float</code>	5.9F	Es posible usar también la notación exponencial 1.8E9
<code>double</code>	7.9	Todos los reales por defecto son <code>double</code> . Se pueden finalizar con una D
<code>boolean</code>	true	<code>true</code> y <code>false</code> son los únicos valores válidos
<code>String</code>	"hola"	Comillas dobles

Es posible utilizar constantes numéricas expresadas en octal y en hexadecimal. Para indicar que una constante está expresada en octal, ésta debe comenzar por 0, si comienza por 0x o por 0X será una constante hexadecimal.

```

Octal      015
Decimal    13
Hexadecimal 0x0D

```

Existe la posibilidad de definir constantes mediante el uso de la palabra reservada `final` situada antes del tipo en la definición de una variable. Al utilizar `final` en la declaración hacemos que esta variable únicamente pueda cambiar su valor una vez y, por tanto, la convertimos en constante.

```
final double PI=3.141592653589793;
```

En Java las constantes se suelen escribir en mayúsculas para diferenciarlas de las variables.

1.2.3. Variables

La forma más sencilla de almacenar información en Java es utilizar variables. Posteriormente veremos que disponemos de elementos más complejos para representar información, pero las variables son la base de todos ellos.

En Java antes de usar cualquier variable, independientemente de su tipo, es necesario declararla. Desde ese momento puede ser usada sin más restricciones que las impuestas por su tipo, su ámbito y su tiempo de vida, características todas ellas que exploraremos seguidamente.

Declaración

La estricta comprobación de los tipos en Java hace que la declaración de variables sea obligatoria; ésta puede realizarse en cualquier parte de una clase o método (su denominación cambia pero son variables en cualquier caso) y a partir de ese momento la variable podrá ser utilizada.

La forma de declaración de una variable en Java básicamente indica el nombre y el tipo de la misma, pero puede ir acompañada de más información, como el valor inicial o la declaración de más variables del mismo tipo.

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

En la declaración anterior, `tipo` es uno de los tipos legales de Java, es decir, tipos básicos, clases o interfaces. Estos dos últimos se verán más adelante. El identificador tiene que ser único y puede contener cualquier carácter UNICODE.

Es posible inicializar la variable con un valor distinto del que Java emplea por defecto. Este valor debe ser del mismo tipo que la variable o un tipo compatible. El valor de la inicialización puede ser cualquier expresión válida, no tiene por qué ser una constante, es, por tanto, legal utilizar una expresión cuyo valor no sea conocido en tiempo de compilación.

```
int i=0,j;  
double d = Math.sqrt(i*5);
```

Ámbito y tiempo de vida

Todas las variables tienen dos características que definen su comportamiento: su ámbito y su tiempo de vida. Generalmente, estos dos conceptos van unidos y no es posible entender el uno sin el otro, pero son dos características diferentes.

La declaración de una variable se debe producir dentro de un bloque de código y ese bloque de código determina su ámbito, es decir, en qué parte del código la variable puede ser accedida. Un bloque es una porción de código encerrado entre dos llaves ({ y }). Hemos visto bloques de código en algunos ejemplos anteriores, cuando definíamos una clase o cuando definíamos el método `main`; pero no son los únicos lugares donde aparecen bloques, la mayor parte de las veces aparecen unidos a sentencias de control, pero pueden aparecer solos para delimitar un fragmento de código o más concretamente un ámbito. Es posible anidar bloques y, por tanto, ámbitos.

```
1. {  
2.     int a;  
3.     a=9;  
4.     {  
5.         int b=a+1;  
6.     }  
7.     a=10;  
8. }
```

Vemos en el ejemplo anterior que la variable `a` está definida en el bloque que comienza en la línea 1 y finaliza en la línea 8, por tanto, es legal acceder a la variable `a` en la línea 5. La variable `b` se define dentro del bloque de las líneas 4, 5 y 6, si intentamos acceder a su contenido en la línea 7 se producirá un error ya que su ámbito no llega a la línea 7 del programa.

Dada la naturaleza orientada a objetos de Java, no existe el concepto de ámbito global y local como en otros lenguajes. Existen otros ámbitos más adaptados a la programación orientada a objetos como son el ámbito de clase y el de método; no son los únicos pero sí los más claros en este momento. Es sencillo deducir que el ámbito de clase corresponde con las líneas de código de una clase y el ámbito de método con las de un método.

El tiempo de vida de una variable es el tiempo (código) que transcurre entre la declaración de la variable y su destrucción. Generalmente, coincide con el ámbito pero existen variables cuya vida no finaliza con la llave que cierra el ámbito donde fueron definidas.

1.2.4. Conversión de tipos

Cuando se evalúa una expresión en la que se mezclan datos con distintos tipos, es necesario realizar conversiones de tipo con el fin de que las operaciones se realicen de una forma coherente; en algunos casos estas conversiones son sencillas ya que afectan a tipos que tienen características comunes, como pueden ser dos tipos enteros; pero en otras ocasiones esto no es tan sencillo y es necesario tomar decisiones que afectan a la fiabilidad de los datos involucrados, por ejemplo, cuando tenemos que convertir un número real en un entero.

Por todo ello, los procesos de conversión de tipo son bastante complejos y no siempre pueden ser automáticos. Cuando es posible realizar la conversión de forma automática, Java la realiza. En los demás casos, no es posible realizar la conversión sin que se pierda información, y es necesario que sea forzada por el programador.

Conversión automática

La conversión automática es la deseable, en la mayoría de los casos el programador no quiere tener que preocuparse por cambios en los tamaños de los enteros, sólo quiere sumar dos enteros. Existen dos reglas básicas para determinar si se puede realizar una conversión automática de tipos en Java:

- Los dos tipos son compatibles.
- El tipo destino es más grande que el tipo origen.

Cuando se cumplen estas dos condiciones se produce una promoción del tipo origen para adaptarlo al tipo destino. En esta promoción nunca se pierde información, por lo que Java puede realizarlo sin problemas y sin que el programador tenga que indicarlo. Es fácil aumentar el tamaño de un `byte` para convertirlo en un `int`.

Las reglas de compatibilidad son :

- Todos los tipos numéricos son compatibles entre sí, sin importar que sean enteros o reales.
- El tipo `char` es compatible con `int`.
- El tipo `boolean` no es compatible con ningún otro tipo.

Cuando se inicializa una variable `long`, `short` o `byte` con una constante entera, Java realiza la conversión de esa constante entera al tipo destino de forma automática. En este caso la constante entera debe estar en el rango del tipo destino, de no ser así se producirá un error de compilación.

Para ilustrar mejor el comportamiento de Java en las conversiones vamos a estudiar algunos ejemplos, para ello definiremos algunas variables y las inicializaremos convenientemente.

```
char c='a',c2;  
int i=23,i2;  
short s;  
double d;
```

Veamos las asignaciones:

```
i2=c;
```

La asignación es correcta, los tipos de `i2` y `c` son compatibles y el valor de `c` cabe en el tipo de `i2`.

```
s=c;
```

Esta asignación es incorrecta, recordemos que en Java el tipo `char` utiliza el mismo número de bits que el `short` para guardar la información, pero en el tipo `short` hay signo lo que le impide representar todos los valores posibles de un `char`. En este caso necesitaremos utilizar otro tipo de conversión que veremos a continuación.

```
d=c;
```


En este caso la asignación sí es correcta, los dos tipos son compatibles y el origen cabe en el destino.

```
s=678;
```

El literal entero 678 se convierte automáticamente a `short` y como está dentro del rango permitido se realiza la asignación.

Conversión explícita. Casting

En los casos en los que la conversión de tipos no puede llevarse a cabo de forma automática, pero necesitamos que la conversión se realice, tenemos que forzar el cambio utilizando una conversión explícita o `casting`. Éste se realiza anteponiendo al dato que queremos cambiar el tipo destino encerrado entre paréntesis.

Utilizando `casting` podemos forzar las conversiones a pesar de que se puede perder información en el cambio. Esta posible pérdida de información es la razón de que no sea automática, Java nos cede la responsabilidad de decidir si queremos o no sacrificar parte en el cambio.

La conversión se realizará siguiendo unas sencillas reglas:

- Entre números enteros, si el destino es mayor que el origen, el valor resultante será el resto (módulo) de la división entera del valor con el rango del tipo destino.
- Si el origen es un número real y el destino un entero, la parte decimal se trunca, además si la parte entera restante no cabe en el destino, se aplica en criterio del módulo.
- Entre números reales, se guarda el máximo valor posible.

Veamos unos ejemplos de conversiones que requieren `casting`:

```
double dou=123.67;  
int dest=(int) dou;
```

La variable `dest` contendrá 123 después de la asignación ya que se trunca la parte decimal al pasar de un número real a un número entero.

```
dou=3.40282347E+50;  
float fl=(float) dou;
```

En este caso el resultado es un poco más sorprendente, pasamos de un tipo `double` a un tipo `float` y el valor que queremos guardar en la variable `fl` supera el rango de un tipo `float`. El resultado de la asignación es un valor infinito, y para representarlo utiliza el valor especial `Infinity` como veíamos anteriormente.

```
int in=257;  
byte b;  
b=(byte) in;
```

Después de la asignación, la variable `b` contendrá 1 ya que se aplica el resto de la división entera para calcularlo y $257 \bmod 256 = 1$.

Promoción en expresiones

Al evaluar una expresión, también se producen conversiones de tipo si es necesario. La regla general cuando esto sucede es que Java convierta automáticamente los operandos al tipo mayor de los presentes en la expresión y utilizará este tipo para evaluar la expresión. Si, por ejemplo, se va a realizar una suma entre un tipo `double` y un tipo `int`, el `int` será promocionado a `double` y la operación será realizada entre dos `double`. El resultado, por tanto, será un `double`.

Las reglas que controlan estas promociones son las siguientes:

- `byte` y `short` se promocionan a `int`.
- Si un valor es `long` la expresión se promociona a `long`.
- Si un valor es `float` la expresión se promociona a `float`.
- Si un valor es `double` la expresión se promociona a `double`.
- Un valor `char` en una expresión numérica se promociona a `int`.

Vamos a realizar una operación utilizando tipos `byte`.

```
byte b,b2,b3;  
b2=10;  
b3=100;  
  
b=(byte) (b2*b3);
```

A pesar de que `b2` y `b3` son del mismo tipo que `b`, la operación se realiza como si fueran `int` y el resultado de la misma es lógicamente un `int`. Como consecuencia, necesitamos utilizar un `casting` para que la asignación se realice sin problemas.

1.2.5. Uso básico de cadenas de caracteres

En Java no existe un tipo básico para almacenar cadenas de caracteres, a pesar de que es uno de los tipos más usados en la mayoría de los programas; en su defecto se utiliza una clase de la biblioteca estándar, la clase `String`. Cada vez que necesitamos una cadena de caracteres instanciamos un objeto de la clase `String`. El concepto de instanciación, de objetos y clases se verá posteriormente. Instanciar un objeto es muy parecido (visto desde fuera) a declarar una variable de un tipo básico.

Al no ser el tipo `String` un tipo básico hace que el manejo de cadenas en toda su potencia requiera manejar clases, objetos y métodos, y por ello vamos a ver cómo utilizar cadenas de forma muy simple y posteriormente ampliaremos la información.

Es posible definir literales de tipo cadena entrecomillando texto con comillas dobles, imprimir con `System.out.println()` estas cadenas e incluso concatenarlas usando el operador `+`.

```
System.out.println("Hola"+" mundo");
```

También podemos definir “variables” de tipo cadena (realmente son objetos) y usarlas para guardar cadenas de caracteres y realizar con ellas operaciones básicas de asignación y concatenación.

```
1. String a,b;  
2.  
3. a="Hola";  
4. b=" mundo";  
5. String c=a+b;  
6. System.out.println(c);
```

En el ejemplo podemos ver en la línea 1 que la declaración de una “variable” de tipo `String` es similar a la declaración de cualquier variable de tipo básico. En las líneas 3 y 4 se realizan asignaciones de literales a las variables definidas. Las líneas 5 y 6 crean una nueva cadena como resultado de la concatenación de `a` y `b` e imprimen el resultado respectivamente.

Como se puede ver en el ejemplo anterior, el tratamiento de cadenas es sencillo en Java, pero no se debe perder de vista que no estamos tratando con tipos básicos y que como se verá posteriormente las operaciones que se realizan con cadenas implican operaciones entre objetos más complicadas que las realizadas con tipos básicos.

1.2.6. Arrays

En ocasiones nos encontramos con un conjunto de datos homogéneos y relacionados entre sí que debemos almacenar. Con lo que hemos visto hasta ahora, la única solución para almacenar esta información es declarando variables independientes y guardando en cada una de ellas un dato. Con un número de variables pequeño esta solución es relativamente buena, pero qué ocurre si tenemos que manejar los datos referentes a las horas de luz en Madrid durante un mes; tendríamos que definir una treintena de variables y el programa se volvería complicado de leer y muy propenso a errores. Para poder afrontar este tipo de problemas necesitamos un tipo de datos más poderoso que los tipos básicos. Necesitamos un array.

Un array (o matriz) es un conjunto de datos homogéneos que ocupan posiciones de memoria contiguas y que es posible referenciar a través de un nombre único. Cada uno de los elementos de los que está compuesto el array es direccionable individualmente a través del nombre de ésta y un índice que indica el desplazamiento a lo largo de los elementos de los que está compuesta.

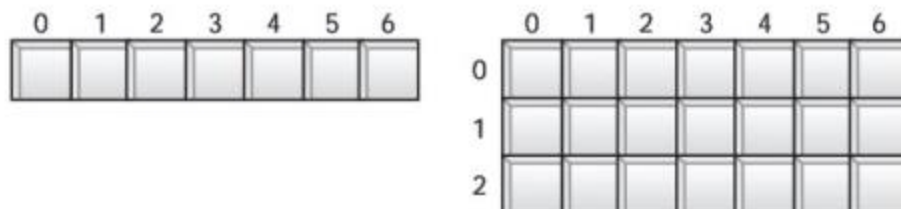


Figura 1.1. Arrays de una y dos dimensiones.

Podemos definir arrays de cualquier tipo de datos, sea éste un tipo básico o como veremos posteriormente tipos más complejos. También podemos aumentar el número de dimensiones y crear tablas o cubos. No existe una limitación en el número de dimensiones que puede tener un array en Java, pero trabajar con estructuras de más de cuatro dimensiones es algo reservado a muy pocos.

Utilizando arrays, manejar un conjunto de datos relacionados se simplifica y es posible mantener unidos los datos de las horas de luz como veíamos antes, o relacionar las ventas mensuales de varios comerciales en forma de tabla.

Declaración de arrays de una dimensión

En Java la declaración de arrays es un poco distinta que en otros lenguajes ya que consta de dos partes; por un lado tenemos que declarar una referencia al array que queremos crear. Una vez que tenemos la referencia, asociarle el array, es decir, primero necesitamos una variable para "apuntar" a los datos del array y después reservamos la memoria necesaria para almacenar esos datos.

La declaración de la referencia del array se realiza especificando el tipo del array seguido de [], después viene el nombre de la variable.

```
int[] a;
```

Una vez que tenemos la referencia, tenemos que reservar la memoria para almacenar el array, para ello utilizamos el operador `new`. Usaremos `new` cuando queramos crear un nuevo elemento de forma dinámica dentro de nuestros programas, en este momento necesitamos crear un array, posteriormente crearemos objetos. El operador `new` debe ir seguido del tipo del array y entre corchetes la dimensión (número de elementos) del mismo.

```
a=new int[10];
```

Esta sentencia crea un array de diez enteros y éste queda referenciado por la variable `a`. A partir de este momento la variable `a` apunta al array creado y la utilizaremos para acceder al contenido del array.

Es posible realizar las dos sentencias en una, y además existen dos formas de declarar un array. También es posible que el tamaño del array en el momento de la definición no sea conocido en tiempo de compilación.

Como se puede ver en los siguientes ejemplos, la línea 1 declara y define un array de veinte números reales, referenciados por la variable `arr`. En las líneas 3 y 4, creamos el array `arr2` de la misma forma, pero en este caso, el tamaño del mismo viene dado por el valor de una variable lo que hace que su valor no sea conocido hasta que se produzca la ejecución del programa.

```
1. float[] arr=new float[20];  
2.  
3. int tam=9;  
4. float arr2[]=new float[tam];
```


Cuando creamos un array, Java por defecto inicializa todas sus posiciones a 0. Esta inicialización es, además de un trabajo que no tenemos que hacer, una medida de seguridad que evita problemas al acceder a posiciones en las que puede existir cualquier valor existente previamente en la memoria. No obstante, la inicialización por defecto no es siempre la mejor opción y por eso es posible a la hora de declarar el array darle valores por defecto distintos de cero, como podemos ver en la siguiente declaración.

```
int[] numeros={1,2,3,4,5,6,7,8,9};
```

El array `numeros` contendrá, al finalizar su creación, los números del 1 al 9. Como se puede ver, en la declaración de este array no se ha indicado su tamaño, Java es capaz de calcular este dato a partir del número de elementos que encuentre para la inicialización. En este caso, creará un array de nueve posiciones.

El tamaño de un array se indica en el momento de su definición y no puede modificarse posteriormente. Si necesitamos que el array cambie de tamaño, debemos crear un nuevo array y pasarle la información del array inicial. Es posible conocer el número de elementos de un array utilizando la propiedad `length` como vemos en el siguiente ejemplo que imprimirá en la salida estándar un 25.

```
int[] a=new int[25];  
System.out.println(a.length);
```

Acceso a un array de una dimensión

El acceso al array se realiza, normalmente, a través de la referencia usada en su declaración. Es posible acceder al array en su totalidad, utilizando sólo la referencia del array, o a cada uno de los elementos que lo constituyen mediante un índice único, en este caso se añade el índice a la referencia encerrándolo entre corchetes. El índice del array es un número entero comprendido entre 0 y la dimensión -1. Java se encarga de comprobar que todos los accesos que se realizan estén comprendidos en ese rango, cualquier intento de acceso fuera de los valores permitidos provocará un error indicado en forma de excepción.

En el siguiente ejemplo podemos ver un sencillo ejemplo de acceso a un array de enteros declarado y definido en la línea 1 del programa.

```
1. int[] a=new int[10];  
2.  
3. a[0]=1;  
4. a[2]=5;  
5. a[4]=6;  
6.  
7. System.out.println(a[2]);  
8. System.out.println(a[4]);  
9. System.out.println(a[6]);
```

Las líneas 3, 4 y 5 cambian los valores de las posiciones 0, 2 y 4 del array cambiando los ceros de la inicialización por defecto por otros valores. En las líneas 7, 8 y 9 los contenidos de las posiciones 2, 4 y 6 son mostrados por pantalla. El resultado de la ejecución de este fragmento de código se muestra a continuación.

```
5
6
0
```

El último número mostrado es un 0 ya que la posición 6 del array no ha sido cambiada desde su declaración y conserva la inicialización por defecto. Las posiciones 2 y 4 contienen los valores guardados en las asignaciones de las líneas 4 y 5.

Dado que los arrays se manejan utilizando referencias, si asignamos un array a otro, no copiamos los valores de uno en otro, lo que hacemos es apuntar las dos referencias al mismo array.

```
1. int a[]={1,2,3};
2. int b[]={2,4,6};
3. b=a;
4. System.out.println(b[1]);
5. a[1]=99;
6. System.out.println(b[1]);
```

Tras la asignación de la línea 3, a y b apuntan al mismo array, el que contiene los datos {1, 2, 3}, el array al que apuntaba la variable b, se pierde y es eliminado por el recolector de basura de Java. El `println` de la línea 4 imprimirá un 2, pero si realizamos una asignación como la de la línea 5 y volvemos a imprimir el contenido de la posición 1 del array b, veremos que ha cambiado y se imprimirá un 99. Esto es debido a que a y b apuntan al mismo conjunto de datos, no hemos realizado un copia de un array en otro.

Si lo que queremos es copiar los valores de un array en otro y que los dos arrays sean independientes, deberemos recorrerlos y copiar cada uno de los elementos de un array en el otro. De este modo, los datos del array origen y los del array destino no ocuparán el mismo lugar de almacenamiento, siendo, por tanto, independientes.

En el siguiente ejemplo vemos cómo copiar un array a otro sin tener problemas.

```
public class CopiaArraysFor{
    public static void main(String [] args)    {
        int [] origen={1,3,5,7};
        int [] destino= new int[origen.length];
        for (int i=0; i<origen.length ; i++ )
            destino[i]=origen[i];
        System.out.println("Después de la copia");
        System.out.print("origen=[ ");
        for (int i=0; i<origen.length ; i++ )
```

```

        System.out.print(origen[i]+" ");
        System.out.println("");
        System.out.print("destino=[ ");
        for (int i=0; i<destino.length ; i++ )
            System.out.print(destino[i]+" ");
        System.out.println("");
        origen[1]=99;

        System.out.println("\nDespués de la asignación");
        System.out.print("origen=[ ");
        for (int i=0; i<origen.length ; i++ )
            System.out.print(origen[i]+" ");
        System.out.println("");
        System.out.print("destino=[ ");
        for (int i=0; i<destino.length ; i++ )
            System.out.print(destino[i]+" ");
        System.out.println("");
    }
}

```

El resultado de la ejecución del programa anterior se muestra a continuación. Tras realizar la copia del array origen en el array destino, ambos arrays contienen la misma información, pero de forma independiente, de tal forma que si modificamos uno, el otro no se ve afectado.

```

Despues de la copia
origen=[ 1 3 5 7 ]
destino=[ 1 3 5 7 ]
Despues de la asignación
origen=[ 1 99 5 7 ]
destino=[ 1 3 5 7 ]

```

Dado que copiar un array en otro es una tarea muy común, existe un método en la clase `System` que podemos usar para copiar los valores de un array en otro.

```
System.arraycopy(origen, indiceOrigen, destino, indiceDestino, num);
```

Este método copia desde el índice `indiceOrigen` del array origen al array destino, comenzando en la posición `indiceDestino` copia `num` elementos. El array destino debe tener suficiente espacio para la copia.

```

public class CopiaArray {
    public static void main(String [] args) {
        int[] origen=new int[10];
        int[] destino={1,1,1,1,1};
        origen=new int[] {2,2,2}; // Array anónimo
    }
}

```

```
        System.arraycopy(origen,0,destino,1,origen.length);  
        for(int i=0;i<destino.length;i++)  
            System.out.println("destino["+i+"]="+destino[i]);  
    }  
}
```

El resultado de la ejecución del programa será:

```
destino[0]=1  
destino[1]=2  
destino[2]=2  
destino[3]=2  
destino[4]=1
```

Declaración de un array multidimensional

En algunas ocasiones necesitamos representar información relacionada en varias dimensiones, es muy frecuente, por ejemplo, comparar en forma de tabla dos conjuntos de datos relacionados. En este caso necesitamos dos arrays. Es cierto que podríamos utilizar estos dos arrays de forma independiente, pero si además estos dos arrays se encontraran unidos bajo el mismo nombre, su tratamiento se facilitaría enormemente.

Tradicionalmente cuando se piensa en una tabla (array de dos dimensiones) imaginamos un conjunto de arrays de una dimensión unidos. En Java, esto es literalmente cierto. Así, para definir un array bidimensional, definimos un array de arrays. La forma de definirlo se ve en la siguiente línea:

```
int[][] tabla=new int[2][3];
```

Declaramos un array de dos filas y tres columnas, o lo que es lo mismo un array de dos elementos, siendo cada uno de ellos a su vez un array de tres elementos.

Al definir un array multidimensional, sólo es obligatorio indicar el número de filas, después se puede reservar memoria para el resto de forma independiente.

```
int[][] tabla=new int[2][];  
tabla[0]=new int[3];  
tabla[1]=new int[3];
```

Cuando tenemos más dimensiones todo continúa funcionando de la misma forma, sólo es necesario añadir otro grupo más de corchetes para poder declarar cada nueva dimensión. Veamos un ejemplo de declaración de un array de tres dimensiones.

```
int[][][] tres;  
tres=new int[2][][3];
```



```
tres[0]=new int [2] [];  
tres[1]=new int [2] [];  
  
tres[0][0]=new int[2];  
tres[0][1]=new int[2];  
tres[1][0]=new int[2];  
tres[1][1]=new int[2];
```

Evidentemente, no es necesario realizar un proceso tan largo para definir un array de tres dimensiones, podemos hacerlo en una sola línea.

```
int[][][] tres=new [2][2][2];
```

No obstante, la posibilidad de declarar independientemente cada dimensión nos permite crear estructuras irregulares en las que cada fila puede tener una dimensión distinta.

```
int[][] test;  
test=new int[3];  
test[0]=new int[4];  
test[1]=new int[7];  
test[2]=new int[2];
```

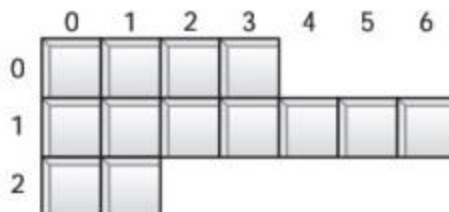


Figura 1.2. Array irregular.

Acceso a un array multidimensional

El acceso a cada uno de los elementos del array es similar al acceso en una dimensión, sólo es necesario añadir un nuevo grupo de corchetes con el índice de la siguiente dimensión.

```
System.out.println(tabla[1][1]);
```

Al igual que con los arrays de una dimensión, en las matrices multidimensionales es posible conocer el número de elementos utilizando la propiedad `length`.

```
public class ArraysDosDim {  
    public static void main(String [] args){  
        int [][] raro;
```

```

int filas=(int) ((Math.random()*10)%5)+1; // Un valor entre 1 y 5
System.out.println("Creamos un array de "+filas+" filas");
raro=new int[filas][];

int columnas;
for (int i=0; i<raro.length; i++){
    columnas=(int) (Math.random()*10+1); // Un valor entre 1 y 10
    System.out.println("Creamos un array de "+columnas+
        " columnas para la fila "+i);
    raro[i]=new int[columnas];
}
System.out.println("Rellenamos el array con datos aleatorios");
for (int i=0;i<raro.length ; i++ )
    for (int j=0;j<raro[i].length ; j++)
        raro[i][j]=(int) (Math.random()*10); // Un valor entre 0 y 9
for (int i=0;i<raro.length ; i++ ){
    for (int j=0;j<raro[i].length ; j++)
        System.out.print(raro[i][j]+" ");
    System.out.println();
}

}
}

```

Un resultado de la ejecución del programa anterior se muestra a continuación.

```

Creamos un array de 5 filas
Creamos un array de 10 columnas para la fila 0
Creamos un array de 10 columnas para la fila 1
Creamos un array de 5 columnas para la fila 2
Creamos un array de 2 columnas para la fila 3
Creamos un array de 5 columnas para la fila 4
Rellenamos el array con datos aleatorios
6 6 0 8 6 9 5 1 9 2
9 9 8 6 3 8 1 8 6 4
0 8 4 9 4
6 2
3 9 3 0 4

```

También es posible inicializar un array multidimensional en el momento de la declaración indicando los valores para cada una de las dimensiones encerrados entre llaves. Cada pareja de llaves que se abre corresponde con una dimensión y separamos los elementos dentro de la misma dimensión con comas.

```
int[][] tabla={{1,2,3},{4,5,6}};
```

1.2.7. Operadores

En Java disponemos de un conjunto de operadores bastante grande, lo que permite realizar prácticamente cualquier operación de una manera directa. Disponemos de operadores para realizar desde operaciones aritméticas sencillas hasta operaciones a nivel de bit.

Operadores aritméticos

Los operadores aritméticos deben usarse con operandos de tipo numérico, sin importar si son enteros o reales, o con operadores de tipo char.

Tabla 1.6. Operadores aritméticos.

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo
++	Incremento
--	Decremento

El operador de división produce un resultado entero si los operadores son enteros y un resultado real si los operadores son reales. Posteriormente, veremos cómo usar la conversión de tipos explícita para cambiar este comportamiento.

Los operadores de incremento y decremento pueden utilizarse como prefijo o como sufijo. Si lo usamos como prefijo, el operando se incrementa o decrementa antes de que el valor se utilice en la expresión. Si lo usamos como sufijo, la operación de decremento o incremento se realiza después de usar el valor en la expresión.

```
public class Operadores {  
    public static void main(String [] args){  
        int a=9;  
        int b=9;  
        System.out.println(a++);  
        System.out.println(++b);  
    }  
}
```

El resultado de la ejecución del programa anterior ilustra perfectamente las diferencias existentes entre el operador de incremento (y también el de decremento) usado como prefijo o como sufijo.

Operadores relacionales

Los operadores relacionales se usan siempre que necesitamos establecer una comparación de cualquier tipo entre dos operandos. El resultado de la evaluación de cualquier operador relacional es un tipo de datos `boolean`.

Tabla 1.7. Operadores relacionales.

Operador	Descripción
<code>=</code>	Igual
<code>!=</code>	Distinto
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual
<code><=</code>	Menor o igual

Los operadores relacionales pueden ser usados sobre cualquier tipo de datos básico. Si son usados sobre tipos más complejos, como arrays u objetos, los resultados no son correctos. Por ejemplo, el operador `==` aplicado sobre dos cadenas de caracteres, únicamente comprueba que ambas cadenas ocupen la misma posición de memoria. Obviamente ésta no es una buena manera de comprobar si dos cadenas son iguales. Por ello, para comparar la igualdad de dos cadenas utilizaremos el método `equals()`, el cual devolverá `true` si las dos cadenas son iguales y `false` en caso contrario.

Es posible utilizar `equals()` con variables o con constantes:

```
String s="cadena";
String p="otra cadena";
boolean res = "cadena".equals(s); // True
boolean res2 = s.equals(p);       // False
```

Si necesitamos que la comparación se realice ignorando las diferencias entre mayúsculas y minúsculas podemos utilizar `equalsIgnoreCase()`.

```
String cad="Cadena";
String cad2="CADENA";
boolean res=cad.equals(cad2);           // False
boolean res2=cad.equalsIgnoreCase(cad2); // True
```

Operadores lógicos

Los operadores lógicos se aplican sobre operandos `boolean` y como resultado se obtiene un valor también `boolean`. Este tipo de operadores proporciona la capacidad de crear expresiones relacionales más complejas al permitir unir varias expresiones simples en una expresión compleja.

Java dispone de operadores AND y OR en modo normal y en modo cortocircuito. En el modo normal, todos los operandos involucrados en la operación lógica son evaluados independientemente de que el resultado de la operación se conozca de antemano. Esto ocurre, por ejemplo, en el siguiente fragmento de código:

```
int a,b;
boolean r;
a=3;
b=8;
r=a!=0 | b>a;
```

En este ejemplo, no es necesario evaluar la expresión `b>a`, ya que al ser `a!=0` cierto, la expresión entera lo es. No obstante, al usar el operador `|`, se evalúa completamente la expresión.

Tabla 1.8 Operadores lógicos.

Operador	Descripción
&	AND
	OR
^	XOR
&&	AND en cortocircuito
	OR en cortocircuito
!	NOT

Si usamos la variante en cortocircuito, la expresión se evaluará hasta que se conozca el valor seguro del resultado. Esto nos permite realizar comprobaciones ligando el resultado de la parte derecha de una expresión al cumplimiento de la parte izquierda de la misma. Por ejemplo:

```
int num;
num=-9;

if (num<0 && Math.sqrt(num) > 3)
    ...
```

En este ejemplo, si utilizamos el operador `&` en lugar de `&&` obtendremos un valor NaN al tratarse el resultado de un número imaginario.

Operadores a nivel de bit

Los operadores a nivel de bit se pueden aplicar a todos los tipos enteros, es decir, `int`, `long`, `short`, `char` y `byte`. Como se mencionó anteriormente, en Java todos los enteros tienen signo (con la excepción del tipo `char`), por lo que habrá que tenerlo en cuenta en las operaciones de desplazamiento de bits para evitar sorpresas al recoger el resultado.

Tabla 1.9. Operadores a nivel de bit.

Operador	Descripción
~	NOT
&	AND
	OR
^	XOR
>>	Desplazamiento a la derecha
>>>	Desplazamiento a la derecha sin signo
<<	Desplazamiento a la izquierda

En las expresiones los valores `byte` y `short` promocionan a `int`, con lo que pueden producirse resultados inesperados cuando el número que promociona es negativo. En este caso, el signo se extiende rellenando con unos la parte más significativa del número.

```
byte b=64,a;
int i;
i=b << 2;
a=(byte) (b<<2);
System.out.println("Valor de b:"+b);
System.out.println("Valor de i:"+i);
System.out.println("Valor de a:"+a);
```

El resultado de este fragmento de código es :

```
Valor de b:64
Valor de i:256
Valor de a:0
```

En el ejemplo anterior, el valor de `b` promociona a `int` al realizar el desplazamiento, por lo que al truncarlo para guardar el resultado en `a` se obtiene 0.

Los desplazamientos a la derecha pueden ser con signo o sin signo, por lo que los resultados de algunos desplazamientos también pueden parecer curiosos.

```
i=-8;
System.out.println(i>>1);
System.out.println(i>>>1);
```

La ejecución del código anterior produce como salida:

```
-4
2147483644
```

Operadores de asignación

El operador de asignación es el signo igual (=). Asigna la expresión de la derecha a la variable de la izquierda. Además de este operador, existe en Java la posibilidad de escribir en forma reducida operaciones en las que la variable que se encuentra en la parte izquierda de una asignación, aparece también en la parte derecha relacionada con un operador aritmético o lógico. Así, expresiones como :

```
a=a+5;
```

se suelen escribir como

```
a+=5;
```

Todas las formas reducidas se encuentran en la Tabla 1.10.

Tabla 1.10. Formas reducidas de los operadores.

Operador	Descripción
~	NOT
+=	Suma y asignación
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
&=	AND y asignación
=	OR y asignación
^=	XOR y asignación
<<=	Desplazamiento a la izquierda y asignación
>>=	Desplazamiento a la derecha y asignación
>>>=	Desplazamiento a la derecha sin signo y asignación

El operador ternario

El operador ?: es el operador ternario. Puede sustituir a una sentencia `if-then-else`. Su sintaxis es:

```
exp1 ? exp2 : exp3;
```

donde, `exp1` es una expresión booleana. Si el resultado de evaluar `exp1` es `true`, entonces se evalúa `exp2`, en caso contrario la expresión que se evalúa es `exp3`.

Por ejemplo, podemos calcular el valor absoluto de un número con la siguiente expresión:

```
int va, x=-10;
va=(x>=0)? x : -x;
```

Precedencia de los operadores

La Tabla 1.11, muestra la precedencia de todos los operadores de Java. Esta tabla se aplica en aquellos casos en los que sea necesario decidir qué operador se aplica antes. En caso de que dos operadores tengan la misma precedencia se evaluarán de izquierda a derecha, a no ser que tengan asociatividad derecha a izquierda.

Tabla 1.11. Precedencia de los operadores.

Operador	Asociatividad
() [] .	izquierda a derecha
++ - ! +(unario) -(unario) () (cast) new	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
>> >>> <<	izquierda a derecha
> >= <= > instanceof	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
	izquierda a derecha
	izquierda a derecha
&&	izquierda a derecha
	izquierda a derecha
?:	izquierda a derecha
= += -= *= /= %= &= = = <<= >>= >>>=	izquierda a derecha

1.2.8. Control de flujo

El control de la ejecución del programa se realiza en Java utilizando unas pocas sentencias. Estas sentencias son básicamente las mismas que en C/C++, es un uno de los aspectos que más similitudes se encuentran entre estos dos lenguajes.

Sentencias condicionales

En Java disponemos de dos sentencias condicionales, la sentencia `if-else` para condiciones simples y la sentencia `switch` para realizar selecciones múltiples.

Sentencia `if-else`

La sentencia `if-else` sirve para tomar decisiones, es posiblemente la estructura de control más usada en programación. Nos permite decidir entre dos posibles opciones excluyentes.



La sintaxis es la siguiente, donde la parte `else` es opcional:

```
if (expresión)
    sentencia-1
[else
    sentencia-2]
```

La expresión que acompaña al `if` debe producir, al ser evaluada, un valor booleano. Si este valor es `true` entonces la `sentencia-1` es ejecutada; si el valor resultante de la evaluación es `false` se ejecutará la `sentencia-2`. Dado que la parte `else` es opcional, si el resultado de la evaluación de la expresión es `false`, y no hay parte `else`, la ejecución continuará en la siguiente línea al `if`.

Es posible anidar sentencias `if`. Si lo hacemos tenemos que poner especial cuidado a la hora de emparejar las partes `else` con sus correspondientes `if`. Los `else` se emparejan con el `if` más cercano dentro del mismo bloque que no tenga ya asociado un `else`.

```
int resultado;
int valor=3;
if (valor>3)
    resultado=1;
else if (valor==3)
    resultado=0;
else
    resultado=-1;
```

En todas las sentencias de control es posible sustituir una sentencia por un grupo de sentencias encerradas entre llaves.

```
int resultado;  
int valor=4;  
if (valor>3){  
    resultado=valor+10;  
    resultado*=10;  
}  
else{  
    resultado=valor-1;  
    valor=0;  
}
```

Sentencia switch

Existen situaciones en las que una estructura `if-else` se queda corta ya que tenemos que decidir entre más de dos alternativas. Si se presenta este problema podemos optar por dos soluciones:

- Utilizar una secuencia de `if` anidados.
- Utilizar la sentencia `switch`.



La sintaxis de la sentencia `switch` es la siguiente:

```
switch(expresión) {  
    case valor1: sentencia;  
                sentencia;  
                ...  
                [break;]  
    case valor2: sentencia;  
                sentencia;  
                ...  
                [break;]  
}
```

```
...
[default: sentencia;
      sentencia;]
}
```

El resultado de la expresión debe ser un tipo simple de Java, los valores que acompañan a las partes `case` deben ser constantes y ser tipos compatibles con el resultado de la expresión.

El funcionamiento de la sentencia `switch` es muy sencillo, se evalúa la expresión y en caso de coincidir el valor de la expresión con el valor de una de las ramas `case`, se ejecuta el conjunto de sentencias que sigue. El flujo de ejecución continúa desde ese punto, hasta el final de la estructura `switch` o hasta que se encuentra una sentencia `break`. La parte `default` es opcional, si existe, se ejecutan las sentencias que le siguen en caso de que no se encuentre coincidencia entre el resultado de la expresión y todas las partes `case` del `switch`. Si no existe `default` y no se encuentra coincidencia en ninguna rama `case` no se hace nada.

No existe ninguna restricción sobre el orden en el que deben aparecer los `case` y `default`, si bien, las opciones deben ser todas diferentes.

```
class PruebaSwitch {
    public static void main(String args[]){
        int dia=4;
        String diaSemana;
        switch (dia) {
            case 1:
                diaSemana="Lunes";
                break;
            case 2:
                diaSemana="Martes";
                break;
            case 3:
                diaSemana="Miercoles";
                break;
            case 4:
                diaSemana="Jueves";
                break;
            case 5:
                diaSemana="Viernes";
                break;
            case 6:
                diaSemana="Sábado";
                break;
            case 7:
                diaSemana="Domingo";
                break;
            default:
```

```
        diaSemana="Dia incorrecto";  
    }  
    System.out.println(diaSemana);  
}  
}
```

Bucles

Bucle while

La sentencia iterativa más simple es un bucle `while`. En este bucle una serie de sentencias se repiten mientras se cumple una determinada condición. Una característica que define a este tipo de bucle es que el cuerpo del bucle (conjunto de sentencias) se ejecuta 0 o N veces, ya que si la condición del bucle no se cumple no se entra a ejecutar las sentencias.



La sintaxis Java del bucle `while` es la siguiente:

```
while (expresión)  
    sentencia
```

Su funcionamiento es sencillo, se evalúa la expresión y si como resultado se obtiene un valor `true`, se ejecuta la sentencia, en caso contrario se continúa la ejecución del programa por la línea siguiente al `while`. Como siempre, es posible sustituir la sentencia, por un grupo de sentencias encerradas por llaves. A la sentencia o conjunto de sentencias que se ejecutan repetidamente dentro del bucle se las conocen como cuerpo del bucle.

```
class PruebaWhile{  
    public static void main(String args[]){  
        int valor=250;  
        while (valor>0){  
            System.out.println(valor);  
            valor-=10;  
        }  
    }  
}
```


Bucle for

El bucle `for` es el tipo de bucle más potente y versátil de Java. Se comporta como un bucle `while` a la hora de comprobar la condición de control, pero permite realizar asignaciones y cambios en la variable de control dentro del mismo bucle, no obstante, un bucle `for` es equivalente a un bucle `while`.



La sintaxis del bucle `for` es un poco más complicada que la del bucle `while`:

```
for (exp1;exp2;exp3)
    sentencia
```

Puede omitirse cualquiera de las tres expresiones del bucle `for`, pero los puntos y coma deben permanecer. Las expresiones `exp1` y `exp3` son asignaciones o llamadas a función y `exp2` es una expresión condicional. En caso de que no exista `exp2` se considera que la condición es siempre cierta.

`exp1` se utiliza para inicializar la variable que controla el bucle, con `exp2` controlamos la permanencia en el mismo y con `exp3` realizamos modificaciones sobre la variable que controla el bucle para poder llegar a salir de éste.

```
class PruebaFor {
    public static void main(String args[]){
        int i;
        for (i=0;i<100;i++)
            if ((i % 2)==0)
                System.out.println(i + " es divisible entre 2");
    }
}
```

La mayor parte de las veces, la variable de control del bucle se usa solamente dentro de él, por ello, se suele declarar dentro del bucle, con lo que su ámbito y tiempo de vida coinciden con el del bucle.

```
for(int i=0;i<10; i++)
    System.out.println(i);
```

Un bucle `for` puede escribirse como un bucle `while` de la siguiente forma:

```
exp1;
while(exp2) {
    proposicion
    exp3;
}

class PruebaForWhile {
    public static void main(String args[]){
        int i;
        i=0;
        while(i<100) {
            if ((i % 2)==0)
                System.out.println(i + " es divisible entre 2");
            i++;
        }
    }
}
```

Ocurre también con cierta frecuencia que es necesario utilizar dos o más variables dentro de un bucle `for`:

```
class PruebaFor1{
    public static void main(String args[]){
        int a,c=0;
        a=9;
        for(int b=0;b<a;b++)
            c=a+b;
        System.out.println(c);
    }
}
```

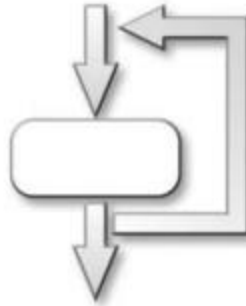
Es posible realizar varias inicializaciones y varias operaciones sobre las variables de control del bucle utilizando el operador coma (,) para separar cada una de las operaciones.

```
class PruebaFor2{

    public static void main(String args[]){
        int c=0;
        for(int b=0,a=9;b<a;b++)
            c=a+b;
        System.out.println(c);
    }
}
```

Bucle do-while

Con los bucles `while` y `for` es posible que el cuerpo del bucle no llegue nunca a ejecutarse, debido a que antes de entrar en ellos se comprueba la condición y si ésta no se cumple se continúa la ejecución sin entrar en el bucle. Existen algunos casos en los que es necesario que el cuerpo del bucle se ejecute al menos una vez, en estos casos, podemos usar un bucle `do-while` en el que la comprobación de la condición del bucle se evalúa después de ejecutado el cuerpo del bucle.



La sintaxis del bucle `do-while` es la siguiente:

```
do
    sentencia
while (expresión);
```

Este tipo de bucles es particularmente útil para controlar menús, ya que en la impresión del menú (cuerpo del bucle) debe mostrarse al menos una vez, y una vez mostrado es cuando el usuario tiene opción a elegir y su petición puede ser atendida y procesada.

La sentencia `break`

En algunos casos es necesario salir de un bucle en un momento que no corresponde con la comprobación de la condición de salida del mismo. En estos casos podemos usar `break`. Si dentro de un programa aparece un `break`, el flujo del mismo sale del bucle (o `switch`) más interior, eso quiere decir que si tenemos bucles anidados únicamente saldremos del más interno.

```
class PruebaBreak{
    public static void main(String args[]){
        for (int i=0;i<100;i++){
            System.out.println(i);
            if (i==50) break;
        }
        System.out.println("Fin del bucle");
    }
}
```

El uso de `break` en los programas sólo debe producirse en situaciones en las que no sea posible salir a través de la condición normal de salida o que para hacerlo sea necesario complicar la lógica del programa demasiado. Generalmente, su uso está desaconsejado.

La sentencia `continue`

Cuando dentro de un bucle encontramos una sentencia `continue` obligamos al programa a realizar la comprobación de condición de salida en ese momento, sin necesidad de concluir la iteración en curso.

```
class PruebaContinue{
    public static void main(String args[]){
        int suma=0;
        for (int i=0;i<100;i++){
            if ((i%2)==0) continue;
            suma+=i;
        }
        System.out.println("La suma de los impares es "+suma);
    }
}
```

El uso de `continue` no es necesario en la inmensa mayoría de los programas, y lo mismo que ocurría con `break`, sólo en las situaciones de especial complejidad.

1.2.9. Entrada/salida básica

La entrada/salida por consola en Java es bastante limitada y no se suele utilizar mucho, ya que normalmente los programas en Java utilizan, para realizar la comunicación con el usuario, interfaces más complicadas que la consola. Esto es debido a que la mayor parte de los programas en Java se ejecutan en entornos gráficos y en ellos tienen la posibilidad de realizar la entrada y la salida de datos de forma más elaborada.

No obstante, en los primeros programas, y en aquellos en los que la interacción con el usuario no es lo primordial, el uso de la entrada/salida por consola es una buena opción.

Flujos de datos

En Java la entrada/salida se realiza utilizando flujos (`streams`). Podemos entender un flujo como la representación de un productor/consumidor de información. Si disponemos de un flujo al que enviar información y éste es capaz de recogerla, como un sumidero, entonces diremos que es un flujo de salida, si por el contrario es el flujo el que produce la información, como una fuente, entonces tendremos un flujo de entrada.

Todos los flujos están unidos a un elemento, físico o no, que es el origen o el destino de la información que circula por el flujo. Los flujos se comportan de la misma forma, independiente-

mente de dónde estén conectados. Dispondremos de las mismas clases y métodos si conectamos un flujo a un fichero o lo hacemos a una conexión de red.

El uso de flujos en Java hace que la E/S sea homogénea y que no sea necesario escribir código independiente para tratar la entrada por la consola y la salida a un fichero. En Java, el tratamiento es siempre el mismo.

La entrada/salida de Java

Toda la entrada/salida se define dentro del paquete `java.io`. Dentro de este paquete se encuentran definidas todas las clases necesarias para controlar todas las eventualidades derivadas de la entrada/salida. Existen cerca de 30 clases distintas relacionadas con la entrada/salida; este volumen de clases tiene un doble análisis, por un lado disponemos de una potencia impresionante que cubre prácticamente todo, pero por otro lado es bastante complicado entender y usar la entrada/salida de Java.

Básicamente todas las clases relacionadas con la entrada/salida tienen relación con `InputStream` o con `OutputStream`. La primera clase define los parámetros básicos para la entrada y la segunda para la salida, ambas son abstractas.

Flujos predefinidos

Todos los programas en Java importan automáticamente el paquete `java.lang`. Este paquete contiene las clases básicas necesarias para escribir programas en Java. Una de estas clases es `System`, que proporciona acceso a algunas características propias del entorno de ejecución del programa, por ejemplo, acceso a la hora del sistema, el entorno de ejecución, etc.

Dentro de `System` existen tres atributos denominados `in`, `out` y `err` que son tres flujos asociados respectivamente a la entrada estándar, la salida estándar y la salida de errores. Estos tres flujos son accesibles desde cualquier parte del programa sin necesidad de declarar ni incluir nada (son `public static`).

`System.in` es un flujo de entrada y está asociado normalmente al teclado, `System.out` y `System.err` son dos flujos de salida y están asociados normalmente con la consola.

Salida estándar y salida de errores

Podemos enviar información a la consola mediante `System.out` y `System.err`. Utilizaremos `System.out` para enviar los mensajes referentes a la ejecución del programa, y `System.err` para enviar los mensajes producidos por errores en la ejecución. En algunos sistemas operativos no hay diferencia entre la salida estándar y la salida de errores.

Generalmente, utilizaremos los métodos `println` y `print`. La diferencia existente entre ellos radica en que el primero realiza un salto de línea al finalizar la impresión, cosa que `print` no hace.

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

```

        System.out.print("Hola ");
        System.out.println("mundo!");
    }
}

```

Entrada estándar

La entrada estándar en Java es bastante más complicada que en otros lenguajes de programación. En Java no disponemos de ningún método genérico de lectura de consola. Además es necesario que incluyamos el paquete `java.io` antes de usar cualquier método de entrada.

Para leer la entrada estándar utilizaremos `System.in`. El método de más bajo nivel de que disponemos es `read()`. Este método nos devolverá un byte del flujo de entrada en forma de entero, un `-1` si no hay más entrada disponible y no lo hará hasta que pulsemos la tecla ENTER. Esto hace de `read` un método poco usado. Además tenemos que tener en cuenta que `read()` puede elevar una excepción de tipo `IOException` como consecuencia de su uso, lo que nos obliga a tomar las medidas necesarias para que esta excepción sea convenientemente tratada o propagada.

```

import java.io.*;

public class Read {
    public static void main(String [] args ) throws IOException {
        char c;
        System.out.print("Teclea un caracter seguido de ENTER : ");
        c=(char) System.in.read();
        System.out.println("Has tecleado: "+c);
    }
}

```

Utilizando `throws IOException` en el método `main`, avisamos de la posibilidad de que se produzca una excepción que deberá ser tratada por el programa llamante, que en nuestro caso será el sistema operativo. Es necesario realizar un casting ya que `read()` devuelve un valor entero.

Para poder leer cadenas de caracteres tenemos que recurrir a utilizar varias clases de `java.io`. El método que utilizaremos es `readLine()` que pertenece a la clase `BufferedReader`. Para crear el objeto de esta clase utilizamos otra clase intermedia, `InputStreamReader` que nos permite cambiar un objeto de clase `InputStream` en un objeto de tipo `Reader`, necesario para crear un objeto de tipo `BufferedReader`. Veremos posteriormente la entrada/salida con detalle, por el momento el siguiente ejemplo muestra la forma de leer una cadena.

```

import java.io.*;

public class LeerCadena {
    public static void main(String [] args ) throws IOException {
        BufferedReader stdin=new BufferedReader(

```

```

        new InputStreamReader(System.in));
String cadena=new String();

System.out.print("Escribe una cadena: ");
cadena=stdin.readLine();
System.out.println("Has escrito: "+cadena);
    }
}

```

Si necesitamos leer otros tipos de datos debemos hacerlo a través de `readLine()` y posteriormente convertir la cadena leída al tipo deseado.

```

import java.io.*;
public class ESBasica {
    public static void main(String [] args ) throws IOException {
        BufferedReader stdin=new BufferedReader(
            new InputStreamReader(System.in));

        // Lectura de un entero. int, short, byte o long
        System.out.print("Escribe un numero entero: ");
        int entero=Integer.parseInt(stdin.readLine());
        System.out.println("Has escrito: "+entero);
        // Lectura de un número real. float o double
        System.out.print("Escribe un numero real: ");
        Float f=new Float(stdin.readLine());
        float real=f.floatValue();
        System.out.println("Has escrito: "+real);
        // Lectura de un valor booleano
        System.out.print("Escribe true o false: ");
        Boolean b=new Boolean(stdin.readLine());
        boolean bool=b.booleanValue();
        System.out.println("Has escrito: "+bool);
    }
}

```

1.2.10. Conceptos básicos de atributos y métodos

Los métodos en Java constituyen la lógica de los programas y se encuentran en las clases. La definición de un método es muy sencilla, sigue la siguiente sintaxis:

```

tipo nombre(parametros)
    cuerpo del método

```

donde `tipo` es el valor de retorno del método; este valor es devuelto por el método utilizando la palabra reservada `return`. El nombre del método es el utilizado para su invocación y debe ser un identificador único dentro de la clase.

Normalmente, las llamadas a los métodos se realizan a través de un objeto utilizando la siguiente sintaxis:

```
objeto.metodo();
```

Si la llamada se produce dentro de la misma clase, entonces el nombre del objeto no aparece, opcionalmente se puede poner `this`. El valor de retorno de un método puede ser cualquier tipo válido de Java.

Los datos dentro de una clase se llaman atributos. La definición de un atributo es la misma que la definición de cualquier variable, sólo que se realiza fuera de un método. Podemos definir atributos de cualquier tipo de datos válido. Los métodos de la clase pueden acceder a los atributos de la misma como si fueran variables definidas dentro de los mismos.

En algunas ocasiones, no resulta conveniente tener que crear un objeto para poder invocar un método. Cuando esto sucede tenemos la posibilidad de utilizar métodos estáticos, también llamados métodos de clase. Un método estático no está unido a una instancia de clase (objeto), es común a todos los objetos instanciados y puede ser invocado incluso cuando no exista ningún objeto de esa clase. Un ejemplo de método estático es `main`, cuando se ejecuta una clase que tiene `main` no existe ningún objeto para invocarlo.

Los métodos estáticos no pueden acceder a métodos no estáticos de una clase sin utilizar un objeto, ya que no es posible garantizar que exista un objeto de esa clase y los métodos no estáticos necesitan un objeto para poder ser ejecutados.

```
class Metodos {
    static void imprimirPares(){
        for (int i=1;i<=25;i++)
            if (i%2==0)
                System.out.println(i);
    }
    static void imprimirPrimos(){
        System.out.println("\n\n2");
        for (int i=2;i<25;i++){
            int divisor=2;
            while (i%divisor!=0 && divisor<i-1) divisor++;
            if (divisor==i-1)
                System.out.println(i);
        }
    }
}

public static void main (String [] args) {
    System.out.println("Números pares hasta 25:");
    imprimirPares();
}
```



```
        System.out.println("Números primos hasta 25:");  
        imprimirPrimos();  
    }  
}
```

Tanto de atributos como de métodos se tratará con mucha más extensión posteriormente, pero es necesario analizar en este momento algunos conceptos ya que serán necesarios para los ejemplos posteriores.

Paso de parámetros

Los parámetros indican qué información y de qué tipo se pasa al método. Es posible pasar como parámetro cualquier tipo de datos, tanto los tipos de datos básicos como los objetos más complicados, pero todos tienen que cumplir que el tipo pasado como parámetro debe coincidir con el tipo esperado.

Existen dos tipos de paso de parámetros: por valor y por referencia. En el primero, los valores que recibe el método son copias de los valores originales pasados al mismo, por tanto, cualquier modificación que se realice sobre ellos sólo será visible en el ámbito del método. En el segundo, el paso se realiza pasando una referencia al dato, esto hace que dentro del método se trabaje con el dato verdadero, no con una copia y las modificaciones no se refieran únicamente al ámbito del método.

En Java, todos los tipos básicos pasan por valor y todos los objetos pasan por referencia. Si pasamos un dato de tipo básico como parámetro, lo estamos pasando por valor y si este parámetro es modificado dentro del método, esa modificación se perderá al salir de ésta. En el siguiente ejemplo, se intenta obtener en un parámetro de la llamada el contador de números primos.

```
class Primos {  
    static void imprimirPrimos(int max,int num){  
        num=2;  
        System.out.println("1\n2");  
        for (int i=3;i<=max;i++){  
            int divisor=2;  
            while (i%divisor!=0 && divisor<i-1)  
                divisor++;  
            if (divisor==i-1){  
                System.out.println(i);  
                num++;  
            }  
        }  
    }  
    public static void main (String [] args) {  
        int maximo=25,numero=0;  
        System.out.println("Números primos hasta "+maximo+":");  
    }  
}
```

```

        // No obtendremos el valor que queremos.
        imprimirPrimos(maximo, numero);
        System.out.println("Total "+numero+" primos"); // Mal
    }
}

```

El resultado de la ejecución es claro, el valor de `numero` no ha cambiado dentro de `main`, a pesar de que el parámetro `num` sí lo ha hecho.

Números primos hasta 20:

```

1
2
3
5
7
11
13
17
19
23

```

Total 0 primos

Si necesitamos pasar por referencia un tipo básico debemos utilizar un envoltorio, recubrir el tipo básico con una capa de objeto, y de esta forma forzar su paso como parámetro por referencia.

```

class Primos2 {
    static class Envoltorio {
        int dato;
    }
    static void imprimirPrimos(int max, Envoltorio num) {
        num.dato=2;
        System.out.println("1\n2");
        for (int i=3; i<=max; i++) {
            int divisor=2;
            while (i%divisor!=0 && divisor<i-1)
                divisor++;
            if (divisor==i-1) {
                System.out.println(i);
                num.dato++;
            }
        }
    }

    public static void main (String [] args) {

```

```
int maximo=25;
Envoltorio numero=new Envoltorio();
numero.dato=0;

System.out.println("Números primos hasta "+maximo+":");
imprimirPrimos(maximo,numero);
System.out.println("Total "+numero.dato+" primos");
}
}
```

En este caso, el programa funciona correctamente. Al encerrar el tipo `int` dentro de un objeto, es posible modificar su contenido dentro del método y que ese valor modificado llegue al método `main`.

Números primos hasta 25:

1
2
3
5
7
11
13
17
19
23

Total 10 primos

1.3. Ejercicios resueltos

Ejercicio 1.1.

Enunciado Implementar un programa en Java que obtenga el número primo inferior a un valor introducido por teclado. Se recomienda implementar un método que determine si un número es primo o no, y utilizarlo para ir probando números desde el valor introducido hacia abajo.

Solución

```
import java.io.*;

public class PrimoMenor{
    public static void main(String args[]) {
        int numero=0;
        int ultimoPrimo=1;
        int cadaNumero=numero;

        InputStreamReader isr=new InputStreamReader(System.in);
```

```

BufferedReader br=new BufferedReader(isr);
String valor;
try {
    System.out.println("Introducir numero: ");
    valor=br.readLine();
    numero=Integer.valueOf(valor).intValue();
}
catch (IOException ioex) {
    System.out.println("error de entrada");
}

while (cadaNumero>=2){
    if (esPrimo(cadaNumero)){
        ultimoPrimo=cadaNumero;
    }
    cadaNumero--;
}
System.out.println("el numero primo inferior o igual a "+numero+" es:
                    +ultimoPrimo);
}

static boolean esPrimo (int par){
    int resto = 1;
    int n=2;
    while((n<par/2) && (resto!=0)){
        resto= (int) (par % n);
        n++;
    }
    if (resto!=0)
        return true;
    else
        return false;
}
}

```

Ejercicio 1.2.

Enunciado Escribir un programa que calcule y presente en pantalla el "triángulo de Tartaglia". Se muestra a continuación un ejemplo para 6 filas:

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

```


Para ello construya un array bidimensional triangular donde almacenar el triángulo de Tartaglia hasta la fila n y a continuación representarlo en pantalla. Obsérvese que en una fila, el valor en cada columna es el siguiente:

1 si es la primera o última columna de esa fila.

La suma de los valores en la fila anterior correspondientes a la misma columna y la situada a la izquierda.

Solución

```
import java.io.*;

public class TrianguloTartaglia{
    public static void main (String args[]) throws IOException
    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Introducir n: ");
        String valor=br.readLine();
        int n=Integer.parseInt(valor);

        int triangulo[][]=new int[n+1][];
        int k,l,m;
        for (k=0;k<n+1;k++){
            triangulo[k]=new int[k+1];
            for (l=k;l>=0;l--){
                if ((l==k) || (l==0))
                    triangulo[k][l]=1;
                else
                    triangulo[k][l]=triangulo[k-1][l]+triangulo[k-1][l-1];
            }
        }
        for (k=0;k<n+1;k++){
            for(l=0;l<k+1;l++)
                System.out.print(triangulo[k][l]+" ");
            System.out.print('\n');
        }
    }
}
```

Ejercicio 1.3.

Enunciado Escribir un programa en Java que lea una matriz de teclado (por filas) y, a continuación, calcule la potencia n -ésima (n también pedido por teclado) y la presente en pantalla.

Ejemplo

$$\begin{bmatrix} 1 & 0 & 1 \\ 2 & 1 & 3 \\ 0 & 1 & 1 \end{bmatrix}^9 = \begin{bmatrix} 2187 & 2187 & 4374 \\ 8748 & 8748 & 17496 \\ 4374 & 4374 & 8748 \end{bmatrix}$$

Solución

```

import java.io.*;
import java.util.StringTokenizer;
public class PotenciaMatriz {
    public static void main(String args[]) throws IOException{
        int n, p;
        int i,j;
        InputStreamReader isr=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(isr);
        String valor, cadaNum;

        System.out.println("Introducir tamaño: ");
        valor=br.readLine();
        n=Integer.valueOf(valor).intValue();

        int[][] m = new int[n][n];
        int[][] resultado = new int[n][n];

        //lectura de m
        for (i=0;i<n;i++){
            System.out.println("introduzca fila "+i);
            valor=br.readLine();
            StringTokenizer st=new StringTokenizer(valor);
            for (j=0;j<n;j++){
                if (i==j)
                    resultado[i][j]=1;
                else
                    resultado[i][j]=0;
                cadaNum=st.nextToken();
                m[i][j]=Integer.parseInt(cadaNum);
            }
        }
        System.out.println("matriz introducida:");
        ver(m);
        System.out.println("potencia ");
        valor=br.readLine();
        p=Integer.valueOf(valor).intValue();

        for (i=0;i<p;i++){
            resultado=producto(resultado,m);

            System.out.println("resultado:");
            ver(resultado);
        }
        public static int[][] producto (int[][] a, int [][] b){

```

```

int i,j,k;
int n=a.length;
int[][] resultado=new int[n][n];
for (i=0; i<n; i++)
    for (j=0; j<n; j++){
        resultado[i][j]=0;
        for (k=0; k<n; k++)
            resultado[i][j] = resultado[i][j] + a[i][k]*b[k][j];
    }
return resultado;
}

public static void ver (int[][] a){
    int i,j;
    int n=a.length;
    for (i=0;i<n ;i++ ){
        for (j=0;j<n ;j++ ){
            System.out.print(" "+ a[i][j]);
        }
        System.out.println();
    }
}
}

```

Ejercicio 1.4.

Enunciado Para almacenar las temperaturas medias de todos los días de un año no bisiesto, declarar un array de dos dimensiones de elementos de tipo `float`, que contenga 12 filas, y cada fila tendrá 28, 30 o 31 días, según corresponda al mes. Con esta estructura, escribir el código para:

- Rellenar la estructura con temperaturas simuladas. Se modelarán las temperaturas con una variación cíclica anual a la que se superponga una variación aleatoria de 3 grados. Se sugiere una expresión como la siguiente:

$$\text{temperatura}(\text{día_año}) = 20 - 10 \times \cos\left(\frac{\text{día_año} \times 2\pi}{365}\right) \pm 3^\circ$$

- Para cada uno de los meses, la temperatura media, máxima, mínima y desviación estándar

$$\text{media} = \frac{\sum_{i=1}^N a_i}{N} \quad \text{desviación estándar} = \sqrt{\frac{1}{N} \sum_{i=1}^N a_i^2 - \frac{1}{N^2} \left(\sum_{i=1}^N a_i\right)^2}$$

siendo a_i la temperatura de cada día y N el número de días en un mes

- Para un mes seleccionado por teclado, las temperaturas de todos los días, con 7 días por línea. Representar los valores con un máximo de dos decimales.


```

        minimo=temperaturas[mes][1];
    }
    suma+=temperaturas[mes][1];
    sumaCua+=(temperaturas[mes][1])*(temperaturas[mes][1]);
}
int N=temperaturas[mes].length;
media=suma/N;
desv=(float)Math.sqrt(sumaCua/N-suma*suma/(N*N));

System.out.print(meses[mes]+":\t media="+formato(media));
System.out.print("\tdesv: "+formato(desv));
System.out.print(",\tmaxima: "+formato(maximo));
System.out.println(",\tminima: "+formato(minimo));
} // for de meses
System.out.println("Mes para visualizar temperaturas ");
InputStreamReader isr=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(isr);
String mes_cadena;

int mesSelec=Integer.parseInt(br.readLine());

System.out.println("temperaturas mes "+meses[mesSelec-1]);
for (l=0;l<temperaturas[mesSelec-1].length;l++){
    System.out.print(formato(temperaturas[mesSelec-1][l])+"\t");
    if ((l+1)%7==0)
        System.out.print('\n');
}
}
static double formato(double v){
    return ((int)(100*v))/100.0;
}
}

```

Ejercicio 1.5.

Enunciado Escribir un procedimiento que intercambie la fila i -ésima con la j -ésima de una matriz $m \times n$.

Solución

```

class Matrices4 {
    public static void main(String [] args){
        int [][]matriz = {{3,5,-1,6},{2,6,8,2},{13,4,1,1},
                           {12,-2,0,2},{1,1,1,1}};

        int i = 2,j = 4,k;
        int aux;
        for (k=0;k<4;k++){
            aux = matriz[i][k];

```



```

        matriz [i][k] = matriz [j][k];
        matriz [j][k] = aux;
    }
    // visualizar la matriz
    for (int m=0;m<5;m++){
        for (int n=0;n<4;n++){
            System.out.print(matriz[m][n] + ", ");
            System.out.println();
        }
    }
}

```

Ejercicio 1.6.

Enunciado Escribir un programa que lea números por teclado mientras sean positivos y calcule su media y varianza.

Solución

```

class MediaVarianza {
    public static void main(String [] args){
        System.out.println("numeros");
        double sumax=0.0, sumax2=0.0, media=0.0, desv=0.0, cada=0.0;
        int numero=0;
        do{
            valor=br.readLine();
            cada=Double.parseDouble(valor);
            if (cada>=0){
                sumax+=cada;
                sumax2+=cada*cada;
                numero++;
            }
        } while(cada>=0);
        if (numero>0){
            media=sumax/numero;
            desv=Math.sqrt (sumax2/numero-media*media);
        }
        System.out.println("numero: "+numero+" media: "+media+" desv: "+desv);
    }
}

```

Ejercicio 1.7.

Enunciado Desarrollar un programa que implemente una calculadora, presentando los siguientes menús al usuario, leyendo del teclado los operandos y opciones seleccionadas, y sacando el resultado por pantalla:

Menú 1:

Indicar numero de operadores:

0 (salir)

1 operador (ir a Menú 1.1)

2 operadores (ir a Menú 1.2)

Menú 1.1

Introducir operando

Operación deseada:

1. logaritmo
2. exponente
3. seno
4. coseno

Menú 1.2

Introducir operando1

Introducir operando2

Operación deseada:

1. suma
2. resta
3. producto
4. división

Solución

```
import java.io.*;

public class Calculadora {
    public static void main(String args[]) throws IOException{
        int numero=1;
        char menu1, menu2;
        double op1, op2;
        InputStreamReader isr=new InputStreamReader(System.in);
        BufferedReader br=new BufferedReader(isr);
        String valor;

        do{
            System.out.println("Indicar numero de operadores: ");
            System.out.println("0 (salir)");
            System.out.println("1 operador");
            System.out.println("2 operadores");
            valor=br.readLine();
            menu1=valor.charAt(0);
            System.out.println("menu1="+menu1);
            switch (menu1){
```

```

case '1':
    System.out.println("introducir operando");
    valor=br.readLine();
    op1=Double.parseDouble(valor);

    System.out.println("Operacion deseada:");
    System.out.println("1.- logaritmo");
    System.out.println("2.- exponente");
    System.out.println("3.- seno");
    System.out.println("4.- coseno");
    valor=br.readLine();
    menu2=valor.charAt(0);//Integer.valueOf(valor).intValue();
    switch (menu2){
        case '1':
            System.out.println("log("+op1+")="+Math.log(op1));
            break;
        case '2':
            System.out.println("exp("+op1+")="+Math.exp(op1));
            break;
        case '3':
            System.out.println("sen("+op1+")="+Math.sin(op1));
            break;
        case '4':
            System.out.println("raiz("+op1+")="+Math.sqrt(op1));
            break;
    }
    break;
case '2':
    System.out.println("introducir operando1");
    valor=br.readLine();
    op1=Double.parseDouble(valor);
    System.out.println("introducir operando2");
    valor=br.readLine();
    op2=Double.parseDouble(valor);
    System.out.println("Operacion deseada:");
    System.out.println("1.- suma");
    System.out.println("2.- resta");
    System.out.println("3.- producto");
    System.out.println("4.- division");
    valor=br.readLine();
    menu2=valor.charAt(0);//Integer.valueOf(valor).intValue();
    switch (menu2){
        case '1':
            System.out.println(op1+"+"+op2+"="+op1+op2);
            break;

```

```

        case '2':
            System.out.println(op1+"-"+op2+"="+ (op1-op2));
            break;
        case '3':
            System.out.println(op1+"*"+op2+"="+ (op1*op2));
            break;
        case '4':
            System.out.println(op1+"/"+op2+"="+ (op1/op2));
        }
    } //switch principal
}while (menu1 > '0' && menu1 <= '2');

}
}

```

Ejercicio 1.8.

Enunciado Implementar un programa en Java que permita leer información por la entrada estándar (teclado) hasta que el usuario teclee la cadena "fin". Para mostrar la información por la salida estándar utilizar el flujo predefinido `System.out`.

Solución

```

import java.io.*;

public class LeerTeclado {

    public static void main(String [] args) throws IOException{
        //lectura desde teclado
        InputStreamReader flujoEntrada =new InputStreamReader(System.in);
        BufferedReader teclado = new BufferedReader(flujoEntrada);
        System.out.println("Introduce líneas.....");
        System.out.println("Teclea FIN para terminar.");
        boolean terminar=false;
        String linea=null;
        while (!terminar){
            linea = teclado.readLine();
            terminar = linea.equals("fin");
            if (terminar){
                System.out.println("no más líneas....."+linea);
            }else{
                System.out.println("has tecleado:: "+linea);
            }
        }
    }

} //fin main
} //fin LeerTeclado

```

Ejercicio 1.9.

Enunciado Implementar un programa en Java que permita mostrar información por la salida estándar (monitor). Debe utilizarse tanto el flujo predefinido de Java (`System.out`) como alguna de las clases disponibles en el API de Java 1.2 y que permitirían realizar la misma operación (se recomienda estudiar la clase `java.io.PrintWriter`).

Solución La salida estándar está definida en Java como un flujo (`System.out`), que habitualmente es el monitor (realmente se trata de la consola o aplicación que permita mostrar texto a través del monitor, por ejemplo, en el caso del sistema operativo Linux se trataría de un terminal). El siguiente ejemplo muestra dos formas muy simples de escribir por pantalla, la primera (y más habitual) utilizando el método `println()`; (línea 6) del flujo predefinido `System.out` (línea 5). La segunda establece una conexión entre un `PrintWriter` utilizando como flujo de salida el estándar, de los constructores disponibles es importante utilizar: `PrintWriter (OutputStream out, boolean autoFlush)`, y poner el segundo parámetro a `true` para forzar el vaciado automático del `Stream` (en caso contrario no se verá nada por pantalla).

```
import java.io.*;

public class EscribirTeclado {
    public static void main(String [] args) throws IOException{
        //escritura por pantalla
        PrintWriter teclado = new PrintWriter(System.out,true);
        System.out.println("Forma habitual de escribir por la salida estandar");
        String [] lineas = {"en un lugar","de la Mancha", "de cuyo nombre",
                           "no quiero acordarme"};
        for(int i=0; i<lineas.length;i++){
            teclado.println(lineas[i]);
        }
    }
}
```

Ejercicio 1.10.

Enunciado Escribir un programa que tome un número entero por teclado y muestre por pantalla el número de bits a 1 y a 0 que tiene su representación binaria.

Solución

```
import java.io.*;

class ContarBits {
    public static void main(String [] args) throws IOException{
        InputStreamReader flujoEntrada =new InputStreamReader(System.in);
        BufferedReader teclado = new BufferedReader(flujoEntrada);

        int numero;
        System.out.print("Dame un numero: ");
        numero=Integer.parseInt(teclado.readLine());
    }
}
```



```
int mascara=1,numeroUnos=0;

// Cogemos el número de bits por exceso
int numBits=(int) (Math.log(numero)/Math.log(2))+1;

// Comprobamos con un AND si el bit está puesto a 1
for (int i=0;i<=numBits;i++){
    if ((numero & mascara) != 0 )
        numeroUnos++;
        mascara<<=1;
    }
    System.out.print("El numero "+numero+" tiene "+numeroUnos+" bits a 1 y ");
    System.out.println(numBits-numeroUnos+" a 0 ");
}
}
```


Gestión de errores en Java

2.1. Introducción

Java dispone de un mecanismo para el control y la gestión de errores basado en objetos denominado excepciones. La idea básica que subyace tras este mecanismo consiste en la generación de un objeto que representa la situación anormal que se ha producido durante la ejecución del programa. Este control de errores puede resumirse de la siguiente forma:

*Una **excepción** es una condición anormal que surge en una secuencia de código durante la ejecución del programa. Cuando se produce una condición excepcional (error), se crea un objeto que representa la excepción y se le envía al método que lo ha provocado.*

Cuando un método ha provocado una excepción, tiene dos posibilidades:

- *Gestionarla él mismo.* Es decir, el propio método deberá **capturar**, o detectar, el error producido y tratar de resolverlo.
- *Pasarla a otro método.* El método que ha originado el error puede pasar el objeto de tipo excepción al método que lo invocase. De hecho, cualquier método es inicialmente ejecutado por el método `main()`, que a su vez es ejecutado (invocado) por el intérprete de Java. Luego en última instancia, será el propio intérprete el encargado de gestionar el error producido.

Una excepción puede ser generada por el intérprete de Java, o por el propio código (informa de alguna condición de error sobre el método que se está ejecutando). La gestión de excepciones en Java se gestiona mediante un conjunto de palabras reservadas del lenguaje que serán estudiadas en este capítulo, y que son: `try/catch/finally/throw/throws`. Estas palabras reservadas podrán ser utilizadas por el programador para crear su propio **gestor de excepción**, es decir, un bloque de código capaz de resolver o gestionar la situación anómala cuando ésta aparezca.

¿Por qué debe realizarse un control y una buena gestión de los posibles errores en un programa? La respuesta a esta pregunta es bastante sencilla, incluso obvia, cualquier persona que trabaje con una aplicación computacional no desea que ésta termine de forma inesperada o abrupta, perdiendo todo el trabajo realizado hasta el momento o debiendo volver a ejecutar el programa. La gestión de excepciones tiene dos importantes ventajas que podrían resumirse en:

1. *Se detectan y depuran los posibles errores.* Es decir, es posible generar un software robusto capaz de responder correctamente (e incluso resolver) aquellos errores que se consideran habituales (por ejemplo, la introducción errónea de datos por parte del usuario que manipula el programa), e incluso los errores inesperados (por ejemplo, un fallo en una red de computadores cuando se está realizando una transmisión de datos).
2. *Se evita que el programa termine de forma abrupta.* Esta característica es importante, no sólo por la posible pérdida de datos o información, sino por la sensación de inestabilidad, inseguridad, mal funcionamiento, etc., que genera el programa sobre aquellos usuarios que lo están utilizando.

2.2. Tipos de excepciones en Java

Cualquier excepción en Java es una subclase de la clase `Throwable`, que a su vez está dividida en:

- **Exception.** Se trata de un conjunto de excepciones que el programa del usuario debería capturar, es decir, resolver. Una subclase muy importante debido a su utilización es `RuntimeException`, y en especial sus clases derivadas como: `ArithmeticException`, `IndexOutOfBoundsException`, `NullPointerException`, etc., que representan un conjunto de errores muy habituales que suelen aparecer durante la ejecución de un programa.
- **Error.** Esta clase (y sus derivadas) representan fallos de tipo catastrófico, generalmente, no controlados (desbordamiento de pila, etc.), y que originan la parada del programa en ejecución. Sus clases derivadas son: `AWTError`, `LinkageError`, `ThreadDeath`, `VirtualMachineError`, que como puede verse tienen relación con errores muy graves, como por ejemplo: un error en la máquina virtual de Java, la finalización de un hilo (`Thread`), un error severo en el sistema de gestión de ventanas de Java, etc.

El siguiente esquema muestra parte de la jerarquía disponible en Java para la gestión de errores y excepciones:

```
java.lang.Object
|
+--java.lang.Throwable
|
+--java.lang.Exception
|
```

```

|
+java.lang.Exception.AWTException
|
+java.lang.Exception.IOException
|
+java.lang.Exception.RuntimeException
|
+java.lang.Exception.RuntimeException.ArithmeticException
|
+etc.
|
+java.lang.Error
|
+java.lang.Error.AWTError
|
+java.lang.Error.VirtualMachineError
|
+etc.

```

Dado que los objetos de tipo `Error` se consideran fallos no controlables por el programador, serán los objetos de tipo `Exception`, o excepciones, los más interesantes para tratar de crear programas robustos y tolerantes a fallos. Dentro de las posibles excepciones gestionables, puede diferenciarse entre dos tipos:

- **Excepciones no capturadas.** Si el programa no las trata, o captura, el propio intérprete de Java será el encargado de resolver el problema. Por tanto, una excepción se denomina no capturada en Java, cuando *no es necesario que exista* un gestor de excepciones propio. Es decir, en este caso si el programa no dispone de un código adecuado para resolver el problema, será el propio intérprete de Java el encargado de capturar y resolver la excepción; estas excepciones son las derivadas de `RuntimeException` o de `Error`.
- **Excepciones capturadas.** Son aquellos errores que el programador debe encargarse de capturar y resolver, si no lo hace el programa no compilará. Por tanto, en este tipo de excepciones *debe proporcionarse* un gestor de excepciones propio. Por ejemplo, cuando se define un método que manipula un fichero (véase Capítulo 5), será necesario indicar qué hacer si se produce un fallo de tipo `IOException` (es decir, un error de entrada/salida).

2.2.1. Gestión de excepciones

Una vez que se produce un error en un método, es generado un objeto que encapsula el tipo de error producido, ¿pero qué se hace con ese objeto? Java utiliza un mecanismo muy simple de paso de objetos para resolver el error. Cuando un error se produce en un **método i**, el intérprete de Java (encargado de generar en tiempo de ejecución este tipo de objetos) buscará un gestor adecuado dentro de ese método; si el gestor no existe buscará el gestor en aquellos métodos que hayan invocado (ejecutado) al método que ha generado la excepción. Este proceso continuará hasta encon-

trar un gestor que pueda resolver el problema. En última instancia, si ninguno de los métodos involucrados dispone de un gestor que sepa cómo resolver el problema, será el propio intérprete de Java el encargado de gestionarlo. La siguiente figura muestra de forma gráfica el mecanismo de gestión de excepciones.

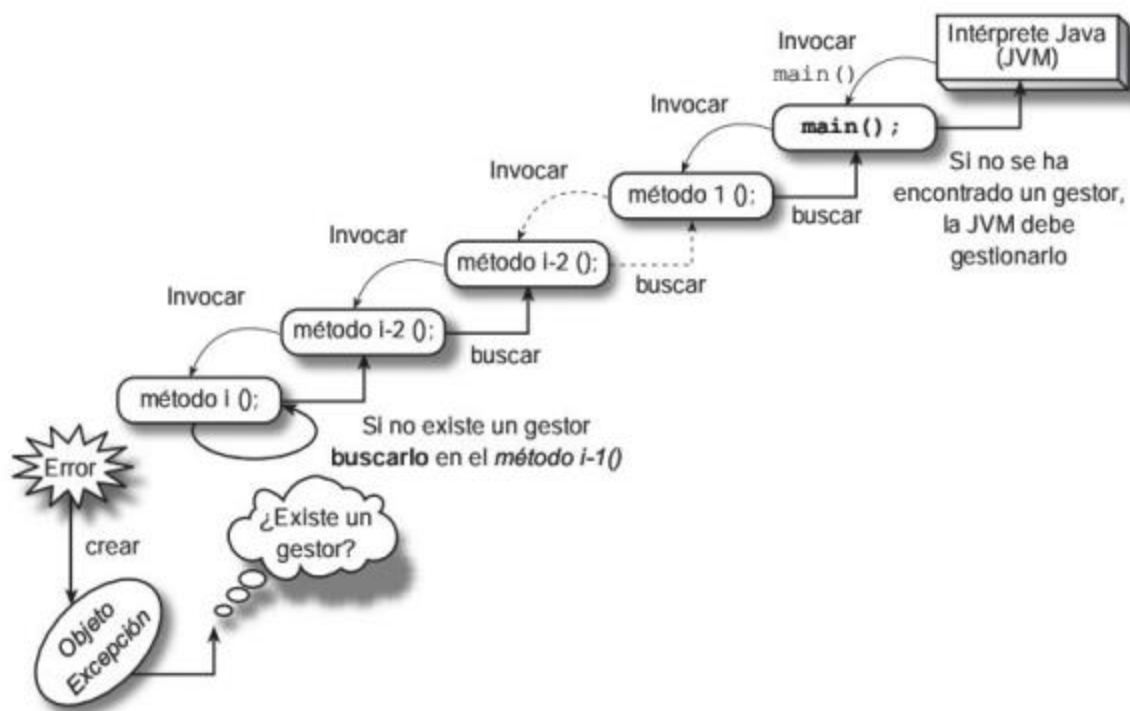


Figura 2.1. Gestión de una excepción de Java.

A continuación, se muestra un ejemplo concreto de una clase Java (ExcepcionNo Capturada) que lanzaría una excepción. Este código compilaría sin problemas, pero al tratar de ejecutarlo se produce un error debido al intento de división entre cero. Como puede verse no hay ninguna estructura, palabra clave, o bloque de código dentro del método que indique de alguna forma al intérprete qué debe hacer con el error que se produce (habitualmente se habla de **capturar** la excepción). Por tanto, el objeto de tipo `ArithmeticException` que se produce será simplemente pasado al intérprete de Java.

```

class ExcepcionNoCapturada {
    public static void main(String args[]){
        int a = 0;
        int b = 10/a;
    }
}
  
```

El intérprete de Java realiza la siguiente gestión del error aparecido: detiene todos los hilos que se encuentren en ejecución (en realidad, detiene el único hilo existente utilizado para ejecutar el `main()`), y muestra por pantalla la información disponible sobre el objeto de excepción que le ha llegado (habitualmente denominada **traza del objeto**). Para el ejemplo anterior, el intérprete de Java mostraría el siguiente mensaje de error por la salida estándar (monitor):

```
Exception in thread "main"
java.lang.ArithmeticException: /by Zero
at ExcepcionNoCapturada.main(ExcepcionNoCapturada.java:4)
```

Como puede verse en este ejemplo, la máquina virtual indica que se ha producido un error de tipo aritmético en un método llamado `main()`; de la clase `ExcepcionNoCapturada` en la línea 4 del código (`int b = 10/a`). La traza del objeto de excepción también indica que se ha producido un error de división por cero (`/by Zero`).

La siguiente figura muestra cómo la ejecución del programa anterior generaría una excepción del tipo adecuado; cuando ésta se produce (se genera el objeto) el intérprete de Java buscará un gestor de excepciones que pueda tratar el error aparecido. Para ello, en primer lugar busca un gestor dentro del propio método que ha provocado el error; dado que en este caso el error se produce directamente en el método `main()`, la excepción termina gestionándola el propio intérprete.

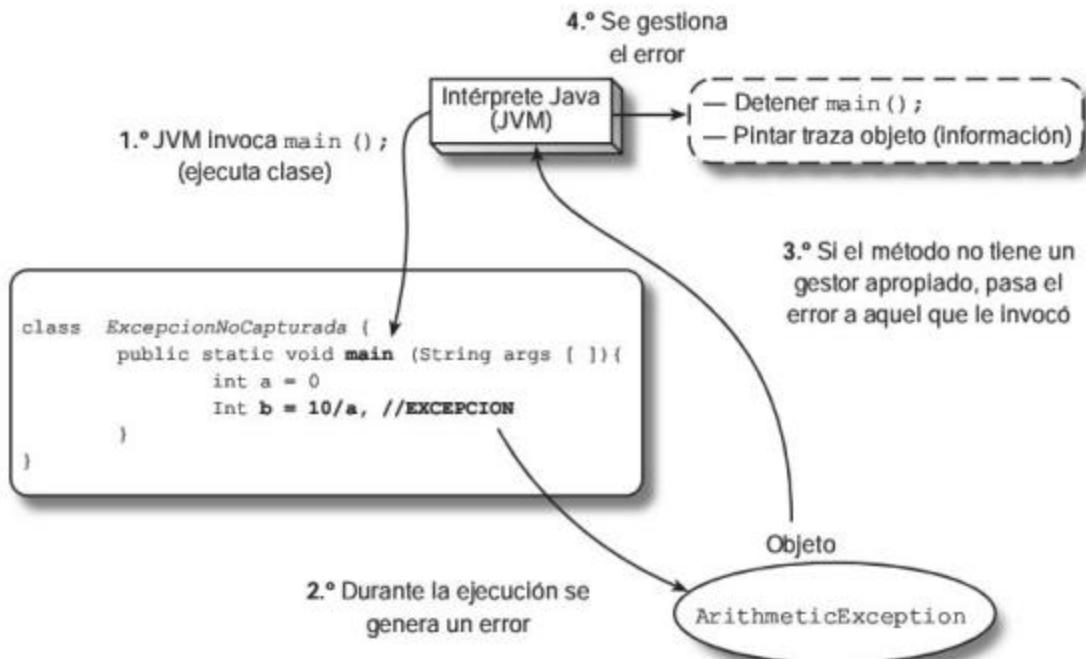


Figura 2.2. Ejemplo concreto de generación y gestión de una excepción en Java.

2.2.2. Clases derivadas de **Exception**

Este conjunto de clases son las derivadas de la clase `java.Object.Throwable.Exception`; se considera que estas clases representan el conjunto básico de errores que pueden aparecer al desarrollar cualquier programa. Este conjunto de clases se encuentran almacenadas en el paquete `java.lang` y es conveniente que cualquier programador de Java conozca su existencia. En esta sección se describirán muy brevemente algunas una de las clases (para la versión de Java 1.2) que posteriormente serán utilizadas a lo largo del texto.

- **AWTException**. Indica una excepción sobre algún elemento derivado del paquete `java.awt`. Este paquete es utilizado para implementar aplicaciones gráficas en Java, luego la clase muestra errores aparecidos sobre cualquiera de los elementos visuales (botones, campos de texto, etc.) utilizables.
- **ClassNotFoundException**. Error al tratar de utilizar una clase (construir un objeto de esa clase, ejecutar un método, etc.).
- **DataFormatException**. Error en el formato de los datos.
- **IllegalAccessException**. Se intenta acceder a una clase a la que no se tiene permiso (por ejemplo, ejecución de un método privado de otra clase).
- **IOException**. Excepciones producidas al realizar tareas de entrada/salida por el programa (véase Capítulo 5). Algunas de sus clases derivadas, y que serán utilizadas en diferentes ejemplos en el libro son: `EOFException`, `FileNotFoundException`, `MalformedURLException`, `SocketException`, `UnknownHostException`, `UTFDataFormatException`.
- **NoSuchFieldException**. No se encuentra un determinado atributo.
- **NoSuchMethodException**. No se encuentra un determinado método.
- **RuntimeException**. Excepciones, o errores, producidos en tiempo de ejecución. Este tipo de excepciones son las más utilizadas, debido a que representan una gran cantidad de errores gestionables por el programa y que pueden aparecer durante la ejecución del mismo. Algunas de las clases derivadas de `RuntimeException` son: `ArithmeticException`, `ClassCastException`, `IndexOutOfBoundsException`, `NegativeArraySizeException`, `NullPointerException`.

Para realizar un análisis pormenorizado de estas clases de excepción y de sus clases derivadas, se recomienda consultar el API de Java 1.2.

2.3. Sentencias **try/catch/finally**

Una vez estudiado qué tipos de excepciones existen en Java y cómo funciona el sistema de gestión de errores en el lenguaje, es necesario estudiar cómo pueden implementarse los gestores capaces de tratar los errores.

En primer lugar, se estudiará un conjunto de sentencias que permiten construir uno de estos gestores. Las sentencias `try/catch/finally` definen tres bloques de sentencias que pueden capturar y resolver un problema aparecido durante la ejecución de un método. Si se traducen estas palabras reservadas como: **intentar/capturar/finalmente** puede entenderse con mayor facilidad cómo funciona y cómo se define uno de estos gestores de excepciones. La idea puede resumirse en los siguientes pasos:

1. Cuando se desea controlar un bloque de programa que puede originar alguna excepción, las sentencias que podrían originar un error se incluyen dentro de un bloque `try`. Es decir, en un bloque `try` se **intenta** ejecutar un código que *podría* generar una o varias excepciones.
2. La excepción (si se produce) se **captura** por el bloque de código `catch` (pueden existir varios).
3. **Finalmente**, una vez se ha ejecutado el código especificado por `try` o por el `catch`, aquellas operaciones que deban ser siempre ejecutadas antes de salir del método aparecerán dentro del bloque `finally`.

La unión de estos bloques de código formará el gestor encargado de resolver los diferentes problemas aparecidos durante la ejecución del método. La sintaxis de estas sentencias se muestra a continuación:

```
try{
    //bloque de código peligroso
}
catch (TipoExcepcion1 exOb){
    //gestión de excepciones para TipoExcepcion1
}
catch (TipoExcepcion2 exOb){
    //gestión de excepciones para TipoExcepcion2
}
// etc.
finally {
    //bloque de sentencias que se ejecutan
    //antes de salir del método
}
```

El bloque `finally` es opcional, aparece después del último `catch` y proporciona un código que **siempre** se ejecuta, *sucedan o no las excepciones*. En cambio, los bloques `try/catch` forman una unidad lógica, dado que si existe un bloque `try`, debe existir por lo menos un bloque `catch` capaz de gestionar el error que podría aparecer durante la ejecución de las sentencias contenidas en el bloque `try`. Las sentencias contenidas en los bloques `catch` sólo se ejecutarán cuando el error que gestionan se produce.

El siguiente ejemplo muestra cómo un bloque de código `try/catch` puede ser utilizado para gestionar el error aparecido en el ejemplo anterior (class `ExcepcionNoCapturada`), de esta forma se evita el problema de la división entre 0 y la finalización inesperada del programa.

```

class ExcepcionCapturada {
    public static void main(String args[]){
        int a,b;
        try{    //controla el código peligroso
            a = 0;
            b = 10/a;
            System.out.println("esto ya no se imprime");
        } catch (ArithmeticException objetoExcep){
            System.out.println("División por cero"+objetoExcep);
        }
        System.out.println("continua el programa....");
    }
}

```

En este ejemplo (al igual que en la clase anterior), cuando el método `main()` es ejecutado vuelve a producirse la excepción de tipo aritmética. Sin embargo, ahora el error es capturado y gestionado. La gestión del error consiste simplemente en mostrar un mensaje de aviso por pantalla, pero a diferencia de la clase `ExcepcionNoCapturada`, ahora (y una vez el bloque `catch` ha finalizado) la ejecución del programa continúa de forma normal.

2.3.1. Múltiples catch

Es posible utilizar múltiples bloques de sentencias `catch` sobre un único bloque de sentencias `try`. Esto es debido a que el bloque `try` puede contener sentencias que originen diversos tipos de problemas. Cuando se tienen varios `catch`, el intérprete de Java los recorre secuencialmente buscando el primero que es adecuado para gestionar el error producido. Esto significa que debe tenerse mucho cuidado en el orden en el que se sitúan estos bloques dentro del código. Si por ejemplo se situase un `catch` que capturase una excepción de tipo `Exception` (la más general) en primer lugar se generaría un error, dado que el resto de bloques que pudiesen existir a continuación nunca se alcanzarían; de modo que los gestores más generales deben situarse después de los específicos.

El siguiente ejemplo muestra cómo pueden emplearse dos bloques de sentencias `catch` para capturar dos posibles errores que podrían originarse en el ejemplo de la división de dos números.

```

class MultiplesCatch{
    public static void main(String args[]) {
        int a;
        int b[];
        int c = args.length; //número de argumentos: línea de comandos.
        try{
            a = 10 / c; // Error si n° argumentos = 0
            System.out.println("valor de a = "+a);
            b = new int[5];
        }
    }
}

```



```

    for (int i = 0; i<a;i++){ // si c = 1, se produce un error
        b[i] = i;
        System.out.print("\t"+i);
    }
} catch (ArithmeticException e){
    System.out.println("Primera excepcion. División por cero:: "+e);
} catch (ArrayIndexOutOfBoundsException e){
    System.out.println("Segunda excepcion. Índice fuera de límites
        =>"+e);
}
}
}
}

```

En este nuevo ejemplo pueden realizarse diferentes ejecuciones del mismo, debido a que en función del número de argumentos que se le pasen la respuesta del mismo será una u otra. Por ejemplo, para el siguiente conjunto de ejecuciones de la clase `MultiplesCatch`:

Probar las siguientes ejecuciones:

1. c:\> java MultiplesCatch
2. c:\> java MultiplesCatch 1
3. c:\> java MultiplesCatch 1 2
4. c:\> java MultiplesCatch 1 2 3
5. c:\> java MultiplesCatch 1 2 3 4

Puede comprobarse que el programa genera las siguientes respuestas:

1. Primera excepcion. División por cero:: java.lang.ArithmeticException: / by zero
2. valor de a = 10

0	1	2	3	4
---	---	---	---	---

 Segunda excepcion. Índice fuera de límites => java.lang.ArrayIndexOutOfBoundsException
3. valor de a = 5

0	1	2	3	4
---	---	---	---	---
4. valor de a = 3

0	1	2
---	---	---
5. valor de a = 2

0	1
---	---

Si se modifica la clase `MultiplesCatch` y se le añade una excepción genérica antes que las otras dos más específicas, puede verse cómo el compilador de Java genera dos errores del tipo:

```

MultiplesCatchException.java [27:1] exception java.lang.ArithmeticException
has already been caught
    } catch (ArithmeticException e){
        ^

```

```

MultiplesCatchException.java                                [29:1]                exception
java.lang.ArrayIndexOutOfBoundsException has already been caught
    } catch (ArrayIndexOutOfBoundsException e){
        ^
2 errors
Errors compiling MultiplesCatchException.

```

Estos errores se producen debido a que el compilador de Java detecta que ambas excepciones están siendo ya capturadas. Por tanto, si se necesita utilizar excepciones más generales (para prevenir posibles errores no previstos por el programador), éstas deberán ser situadas tras los `catch` más específicos. A continuación, se muestra un ejemplo de la utilización correcta de un gestor `catch` genérico:

```

// Ejemplo con varias sentencias catch
class MultiplesCatchException{
    public static void main(String args[]) {
        int a;
        int b[];
        int c = args.length; //número de argumentos: línea de comandos.
        try{
            a = 10 / c; // Error si nº argumentos = 0
            System.out.println("valor de a = "+a);
            b = new int[5];
            for (int i = 0; i<a;i++){ // si c = 1, se produce un error
                b[i] = i;
                System.out.print("\t"+i);
            }
        }catch (ArithmeticException e){
            System.out.println("Primera excepcion. División por cero:: "+e);
        } catch (ArrayIndexOutOfBoundsException e){
            System.out.println("Segunda excepcion. Índice fuera de límites
                                =>" +e);
        }catch (Exception e){
            System.out.println("Excepcion GENERICA..." +e);
        }
    }
} //fin main
} //fin MultiplesCatchException

```

Aunque el gestor implementado para los ejemplos anteriores también detiene la ejecución del programa mostrando únicamente un mensaje de aviso. Es interesante observar que ahora no es el intérprete de Java el encargado de realizar esta gestión, sino que es el programador el que tiene la capacidad de reaccionar ante un posible fallo del programa.

2.3.2. Bloques try/catch anidados

Es posible anidar diferentes bloques de sentencias try/catch. Si se decide utilizar este tipo de control sobre el código generado, es muy importante entender que la búsqueda del gestor de errores se realizará de los catch **más internos hacia los más externos**, deteniéndose el proceso de búsqueda de un gestor en cuanto se halle el primero adecuado. Luego, si se anidan bloques de try/catch y se sitúan gestores muy genéricos en niveles internos de la jerarquía, deberá tenerse en cuenta que en determinadas situaciones estos gestores más externos podrían no ejecutarse. El siguiente programa muestra cómo pueden utilizarse dos bloques try anidados, el primero se encargaría de generar una posible excepción de tipo aritmético si se intenta dividir por cero, mientras que el segundo sólo generará una excepción si se produce un desbordamiento en los índices del array.

```
class TryAnidado{
    public static void main(String args[]) {
        try(
            int c = args.length; //n° de argumentos en la línea de comandos.
            int a = 10 / c; // Error si n° argumentos = 0
            System.out.print("a =      "+a);

            try {
                if (c == 1) a = a/(c-1);
                if (c == 2){
                    int[] b = {100,5}; // el array contiene dos elementos
                    b[3] = 0;//fuera de límites
                }
            } catch (ArrayIndexOutOfBoundsException e){
                System.out.println("Segunda excepcion.
                                   Índice fuera de límites =>" +e);
            }
        ) catch (ArithmeticException e){
            System.out.println("Primera excepcion. División por cero =>" +e);
        } //primer try
    } //main
} //TryAnidado
```

En este ejemplo no aparece el problema descrito anteriormente debido a que las excepciones consideradas no dependen una de la otra. Es decir, son excepciones que se producen por motivos diferentes y una no es subclase de la otra. Como en el caso de la clase `MultipleCatchException`, se podría incluir una excepción interna muy genérica situada después de la excepción más específica (como se vió previamente). A continuación, se ha modificado el ejemplo anterior y se ha implementado la clase `TryAnidadoException`, en la que se ha incluido dentro del try interno (seguida de la excepción `ArrayIndexOutOfBoundsException`) la excepción genérica (`Exception`). Puede observarse que el código compilará y se ejecutará, sin embargo, la excepción `ArithmeticException` sólo se lanza en el caso de que el número de

argumentos sea cero, si el número de argumentos es igual a uno se lanzará la excepción genérica y no la anterior.

```
import java.io.*;
public class TryAnidadoException {
    public static void main(String args[]) {
        try{
            int c = args.length; //número de argumentos: línea de comandos.
            int a = 10 / c; // Error si nº argumentos = 0
            System.out.println("valor de a = "+a);
            try{
                if (c==1)
                    a = a/(c-1);
                if (c==2){
                    int[] b = {100,5}; //el array contiene dos elementos
                    b[3] =0; //fuera de límites
                }
            }catch (ArrayIndexOutOfBoundsException e){
                System.out.println("Segunda excepcion.
                                   Índice fuera de límites=>" +e);
            }catch (Exception e){
                System.out.println("Excepcion GENERICA..." +e);
            }
        }catch (ArithmeticException e){
            System.out.println("Primera excepcion.
                               División por cero:: " +e);
        }
    }
}
//fin main
//fin TryAnidadoException
```

Para las siguientes ejecuciones de las clases TryAnidado y TryAnidadoException es interesante observar el comportamiento de cada uno de los dos programas. De esta forma puede comprenderse mejor el funcionamiento de los diferentes gestores try/catch cuando están anidados.

• **Probar las siguientes ejecuciones (clase TryAnidado):**

1. c:\> java TryAnidado
2. c:\> java TryAnidado 1
3. c:\> java TryAnidado 1 2
4. c:\> java TryAnidado 1 2 3

Salidas:

1. Excepcion externa. División por cero:: java.lang.ArithmeticException: / by zero
2. valor de a = 10

Excepcion externa. División por cero:: java.lang.ArithmeticException: / by zero

```
3. valor de a = 5
   Excepción interna. Índice fuera de
       límites=>java.lang.ArrayIndexOutOfBoundsException
```

```
4. valor de a = 3
```

• **Probar las siguientes ejecuciones (clase TryAnidadoException):**

```
5. c:\> java TryAnidadoException
6. c:\> java TryAnidadoException 1
7. c:\> java TryAnidadoException 1 2
8. c:\> java TryAnidadoException 1 2 3
```

Salidas:

```
5. Excepcion externa. División por cero:: java.lang.ArithmeticException: / by zero

6. valor de a = 10
   Excepcion GENERICA INTERNA...java.lang.ArithmeticException: / by zero

7. valor de a = 5
   Excepción interna. Índice fuera de
       límites=>java.lang.ArrayIndexOutOfBoundsException

8. valor de a = 3
```

2.3.3. Sentencia finally

Dado que las excepciones alteran la ejecución normal de un método (pueden hacer que termine prematuramente), si un método ha utilizado determinados recursos, como por ejemplo abrir un conjunto de ficheros, es necesario asegurarse que esos recursos sean restaurados (es decir, si se tiene un conjunto de ficheros abiertos se trataría de cerrar los archivos antes de finalizar la ejecución del método). A este tipo de problema se le conoce habitualmente por *fuga de recursos*. La sentencia `finally` permite crear un bloque de código que se ejecutará después de `try/catch` y que permitirá evitar la fuga de recursos en el programa.

La declaración de un bloque `finally` es opcional, pero en el caso de que sea declarado siempre se ejecutará, tanto si se produce la excepción (y se ejecuta el bloque `catch`) como si no se produce (y se ejecuta únicamente el bloque `try`). El siguiente ejemplo muestra cómo puede ser utilizada la sentencia `finally` y el efecto que tiene sobre un conjunto de métodos donde aparece definida.

```
class PruebaFinally{
    //lanzar excepción desde fuera del método
    static void metodo1() {
        try{
            System.out.println("en el metodo1");
            throw new RuntimeException("metodo1");
        } finally {
```



```

        System.out.println("finally del método1");
    }
}
//ejecuta return en try
static void metodo2() {
    try{
        System.out.println("en el método2");
        return;
    } finally {
        System.out.println("finally del método2");
    }
}
//ejecuta bloque en try normal
static void metodo3() {
    try{
        System.out.println("en el método3");
    } finally {
        System.out.println("finally del método3");
    }
}
} //fin metodo3
public static void main(String args[]) {
    try{
        metodo1();
    } catch (Exception e) {
        System.out.println("Excep Caturada");
    }
    metodo2();
    metodo3();
} //fin main
} //PruebaFinally

```

La ejecución del anterior ejemplo generará la siguiente salida:

```

en el método1
finally del método1
Excep Caturada
en el método2
finally del método2
en el método3
finally del método3

```

Como puede verse en la salida mostrada, independientemente de si las excepciones son o no lanzadas (el método 2 y el método 3 no lanzan excepciones, sin embargo, el método 1 lanza una excepción de tipo `RuntimeException`) los bloques `finally` son siempre ejecutados (incluso en el caso del método 2 que tiene un `return;` antes del bloque `finally`); de esta forma

puede garantizarse que cualquier operación que deba ser realizada por un método se ejecutará incluso en el caso de la aparición de la correspondiente excepción.

2.4. Sentencias `throw` y `throws`

No siempre tiene por qué interesar la implementación de un gestor de excepciones, puede incluso que al programador no le importe qué suceda con el posible error si éste aparece. En este caso Java dispone de la posibilidad de indicar que un determinado método *puede genera, o lanzar* (`throw`), un determinado tipo de excepciones. En el caso totalmente opuesto, puede que en una determinada situación (por ejemplo, la ejecución de un bloque de sentencias) sea necesario *generar, o lanzar* (`throw`), una excepción concreta. Este apartado estudiará cómo pueden implementarse en Java ambas situaciones.

2.4.1. Sentencia `throw`

La sentencia **`throw`** es utilizada para lanzar (crear) una excepción. A continuación, se muestra un ejemplo muy simple de un método `g()`; que simplemente lanzaría una excepción de tipo `a`.

```
int g() throws a,b,c{  
    //cuerpo del método  
    throw a; //se lanza la excepción  
}
```

El punto desde el cual se lanza la excepción se indica con `throw` (se lanza una excepción *explícitamente* desde el código). Hasta ahora las excepciones podían lanzarse:

- Automáticamente desde el intérprete.
- Desde un bloque de código con `try/catch`.

Utilizando la sentencia **`throw`** es posible lanzar una excepción de forma explícita. La sintaxis de esta sentencia se muestra a continuación:

```
throw InstanciaThrowable; //excepción a lanzar
```

- **InstanciaThrowable**. Debe ser de tipo `Throwable` o una de sus subclases. Los tipos simples de Java (`int`, `char`, etc.), o aquellas clases que no son de ese tipo (como por ejemplo `String`), no pueden ser utilizados como excepciones.

Para obtener un objeto de tipo `Throwable`, se puede utilizar un parámetro en una cláusula `catch` o crear uno con el operador `new` (es decir, utilizar el constructor de la clase correspondiente). El flujo de ejecución del programa se detiene después de la sentencia `throw` y cualquier sentencia posterior no se ejecutará (excepto, como se ha visto previamente, las sentencias contenidas en el bloque `finally`).

El siguiente ejemplo muestra cómo puede utilizarse esta sentencia para controlar la incorrecta asignación de una referencia a un determinado objeto.

```
class ThrowSimple{
    static void metodoDemo() throws Exception{
        try{
            throw new Exception("Fallo en Programa ThrowSimple");
        }catch(Exception e){
            System.out.println("Captura de la excepción lanzada");
            throw e; //se relanza la excepción
        }
    }
    public static void main(String args[]){
        try{
            metodoDemo();
        }catch(Exception e){
            System.out.println("Nueva captura de la excepción: "+e);
        }
    }
}
```

En el ejemplo anterior se muestra cómo el método `metodoDemo()` lanza una excepción de tipo `Exception`, que posteriormente es capturada por su bloque `catch` y vuelta a lanzar. Volver a lanzar una excepción hace que la excepción vaya al contexto inmediatamente más alto de los gestores de excepciones. Relanzar las excepciones permite conservar toda la información relativa al objeto en el punto donde la excepción se originó, de forma que el contexto superior que captura el tipo de excepción específico pueda extraer toda la información sin importar cuántas veces haya sido lanzada la excepción particular. Es decir, si se modifica el anterior ejemplo y el objeto de excepción no es relanzado (ya no es necesario que el método indique que va a lanzar ninguna excepción puesto que se supone que es gestionada dentro del método), puede estudiarse el nuevo comportamiento del programa.

```
public class ThrowSimpleNoRelanzada {
    static void metodoDemo() {
        try{
            throw new Exception("Fallo en Programa ThrowSimple");
        }catch(Exception e){
            System.out.println("Captura de la excepción lanzada");
        }
    }
    public static void main(String args[]){
        try{
            metodoDemo();
        }catch(Exception e){
            System.out.println("Nueva captura de la excepción: "+e);
        }
    }
}
```

```

    }
    }//fin main
  }//fin ThrowSimpleNoRelanzada

```

La salida provocada por la ejecución de la clase `ThrowSimple` origina la siguiente salida:

```

Captura de la excepción lanzada
Nueva captura de la excepción: java.lang.Exception: Fallo en Programa
ThrowSimple

```

Sin embargo, la ejecución de la clase `ThrowSimpleNoRelanzada` origina:

```

Captura de la excepción lanzada

```

La diferencia entre ambos ejemplos permite comprobar el efecto de volver a lanzar un objeto de excepción, permitiendo de esa forma que gestores de nivel superior puedan utilizar o acceder a la información que inicialmente lo originó.

2.4.2. Sentencia `throws`

Se utiliza para indicar que un determinado método podría generar un conjunto de excepciones **que posteriormente no manejan**. Por tanto, la sentencia `throws` permite listar los tipos de excepciones que un método puede lanzar (el listado de excepciones aparecerán separados por una coma). La sintaxis de esta sentencia sería:

```

tipo nombre_de_método (lista_parametros) throws lista_excepciones{
    //cuerpo del método
}

```

La clase `FalloThrows` muestra un ejemplo sencillo donde el método `main()` ejecuta a su vez un método (`LanzaExcep`) que lanza una excepción de tipo `IllegalAccessExcepcion`.

```

class FalloThrows{
    static void lanzaExcep() {
        System.out.println("Lanzamos una excepcion");
        throw new IllegalAccessExcepcion("error producido en lanzaExcep");
    }
    //fin lanzaExcep
    public static void main(String args[]){
        lanzaExcep();
    }
    //fin main
}
//fin FalloThrows

```

Si se intenta compilar el ejemplo anterior, el compilador de Java mostrará el siguiente mensaje de error:

FalloThrows.java [4:1] unreported exception java.lang.IllegalAccessException; must be caught or declared to be thrown

```
    throw new IllegalAccessException("error producido en lanzaExcep");
    ^
```

Como puede verse, el compilador indica que la línea 4 del código (que pertenece al método `lanzaExcep`) genera una excepción que debe ser capturada (mediante la utilización de bloques `try/catch`) o declarada (en una lista de posibles excepciones mediante `throws`). Los dos siguientes ejemplos (`SimpleThrows` y `SimpleThrows2`) muestran cómo se resolvería este problema, utilizando en primer lugar bloques de sentencias `try/catch`, y en segundo lugar la sentencia `throws`.

```
class SimpleThrows{
    static void lanzaExcep() throws IllegalAccessException{
    try{
        System.out.println("Lanzamos una excepcion");
        throw new IllegalAccessException("error producido
                                      en lanzaExcep");
    }catch(IllegalAccessException e){
        throw e; //se relanza la excepción
    }
    }
    public static void main(String args[]){
    try{
        System.out.println("antes de lanzar la excepción");
        lanzaExcep();
        System.out.println("después de lanzar la excepción");
    }catch (IllegalAccessException e){
        System.out.println("excepción capturada:: "+e);
    }
    }
    }
}
```

En el primer ejemplo, el método `lanzaExcep()` lanza y posteriormente captura una excepción de tipo `IllegalAccessException`. El método `main()`, que ejecuta al anterior, *está obligado a capturar* (mediante `try/catch` en este ejemplo) la posible excepción del método al que invoca. La ejecución del anterior programa originaría la siguiente salida:

```
antes de lanzar la excepción
Lanzamos una excepcion
excepción capturada:: java.lang.IllegalAccessException: fallo producido en lanzaExcep
```

Otra posible solución, si no se desean capturar las excepciones, consistiría en utilizar únicamente sentencias `throws`, lo que originaría que finalmente sea la máquina virtual de Java la encargada de gestionar y controlar el error producido.


```
class SimpleThrows2{
    static void lanzaExcep() throws IllegalAccessException{
        System.out.println("Lanzamos una excepción");
        throw new IllegalAccessException("error producido en lanzaExcep");
    }
    public static void main(String args[]) throws IllegalAccessException{
        lanzaExcep();
    } //fin main
} //fin SimpleThrows2
```

Es interesante observar las diferencias entre las dos posibles soluciones anteriores. La ejecución de la clase `SimpleThrows2` genera la siguiente salida:

```
Lanzamos una excepción
java.lang.IllegalAccessException: error producido en lanzaExcep
    at SimpleThrows2.lanzaExcep(SimpleThrows2.java:4)
    at SimpleThrows2.main(SimpleThrows2.java:7)
Exception in thread "main"
```

Como puede observarse, en ambos casos al ejecutarse el método `lanzaExcep()` se imprime por pantalla una sentencia que indica que se ha lanzado una excepción. En el primer ejemplo esta excepción es capturada y resuelta (simplemente se imprime otro mensaje por pantalla que indica el tipo de excepción que ha sido capturada). Sin embargo, en el segundo ejemplo el método `lanza` la excepción que es pasada al método que lo invocó (en este caso `main()`), y dado que este método tampoco captura la excepción lo pasa finalmente al intérprete de Java que muestra la traza del objeto de tipo `IllegalAccessException` que le ha llegado. Es interesante observar que la traza del compilador de Java muestra, en primer lugar, la información sobre el error disponible: `java.lang.IllegalAccessException: error producido en lanzaExcep`, para a continuación mostrar el lugar del código donde esa excepción se ha generado: `at SimpleThrows2.lanzaExcep(SimpleThrows2.java:4)`; esta información se va encadenando hasta finalmente llegar al método `main`: `at SimpleThrows2.main(SimpleThrows2.java:7)`.

2.5. Declaración de excepciones propias

Java permite definir nuestras propias excepciones generando una subclase de `Exception`, el siguiente ejemplo muestra cómo podría definirse un nuevo tipo de excepción.

```
class MyException extends Exception{
    private int dato;
    //constructor
    MyException(int param){
        dato = param;
    }
}
```

```

    }
    public String toString(){
        return "Mi propia excepción : "+dato;
    }
}

```

Para definir una nueva excepción únicamente será necesario crear una nueva clase que herede de la clase `Exception` (`extends Exception`) e implementar aquellos métodos que se consideren necesarios; en el ejemplo anterior, sólo se han definido dos métodos: el constructor que necesita un parámetro entero para construir el objeto de tipo `MyException` y un método `toString()` que devuelve una cadena de texto con el valor del atributo del objeto de tipo `MyException` creado. La anterior excepción podrá ser utilizada por cualquier otro programa al igual que el resto de excepciones definidas por Java. A continuación, se muestra una posible aplicación del mismo.

```

class DemoException{
    static void calcula (int p) throws MyException{
        System.out.println("Ejecuto calcula : "+p);
        if (p >10)
            throw new MyException(p);
        System.out.println("finalización correcta ");
    }
    public static void main(String args[]){
        try{
            calcula(-1);
            calcula(100);
        }catch (MyException e){
            System.out.println("Capturada : "+e);
        }
    }
}
//fin main
//fin DemoException

```

En el anterior programa, el método `calcula(int p)` lanza la excepción del tipo `MyException` en función del valor del parámetro con el que haya sido ejecutado. Como puede verse en el ejemplo, el método `main()` se ejecuta dos veces asignando diferentes valores a `calcula(int p)`, siendo la segunda llamada la que provoca la ejecución de la excepción, la salida de este programa mostraría:

```

Ejecuto calcula : -1
finalización correcta
Ejecuto calcula : 100
Capturada : Mi propia excepción : 100

```

Como puede verse en este ejemplo, al crearse la excepción se crea un objeto de tipo `MyException` cuando la excepción es capturada en el bloque `catch` que imprime el contenido de ese objeto (se utiliza el método `toString()` que sobrescribe al definido por la clase `Object`), para de esa forma mostrar el mensaje o información que previamente se había definido.

2.6. Ejercicios resueltos

En esta sección se muestran algunos ejemplos prácticos de la utilización de excepciones en Java. Debe tenerse en cuenta que dada la importancia de la gestión y control de errores para cualquier programa, la mayoría de los programas implementados a lo largo de este libro utilizarán de una forma u otra el sistema de gestión de errores descrito en este capítulo. El siguiente conjunto de ejercicios se ha dividido en dos grandes bloques:

1. *Ejercicios de desarrollo de aplicaciones gráficas.* Estos ejercicios plantean la implementación y desarrollo de Applets, por lo que será necesario utilizar el paquete `java.applet`.
2. *Ejercicios de redes.* Se utilizarán algunas de las clases asociadas al paquete `java.net` que permite el desarrollo de programas que posteriormente podrán ser ejecutados dentro de una red.

2.6.1. Desarrollo de aplicaciones gráficas

Ejercicio 2.1. Un divisor con control de errores

Enunciado a) Implementar un programa en Java que acepte dos números enteros como parámetros y que calcule su división. En el caso de que el denominador sea igual a cero, se capturará el error producido, y se mostrará un mensaje por pantalla.

Ejemplo c:\> java DivisionConExcepciones 45 16

Resultado (división entera):

División = 2

Ejemplo c:\> java DivisionConExcepciones 234 0

Resultado:

Excepción capturada, intento de división por cero.

- b) Sobre el ejemplo anterior controlar otro tipo de posibles errores, como por ejemplo, el hecho de que alguno de los números introducidos no se encuentre el formato adecuado, o la introducción de un menor número de argumentos de los esperados.

Ejemplo c:\> java DivisionConExcepciones 24 4f6

c:\> java DivisionConExcepciones 24

Solución

- a) El siguiente programa calcula la división entre dos números controlando mediante excepciones la división por cero. En este caso el método `main()` utiliza un bloque `catch` que detiene la ejecución del código y muestra un mensaje por pantalla.

```
public class DivisionByZero {
    public static void main (String[] args){
        //los argumentos de la división se pasan como parámetros
        int numerador, divisor, resultado = 0;
        try{
            numerador = Integer.parseInt(args[0]);
            divisor = Integer.parseInt(args[1]);
            resultado = numerador/divisor;
            System.out.println("Resultado de la división:: "+resultado);
        }catch(ArithmeticException e){
            System.out.println("Excepción, división por cero:: "+e);
        }
    }
} //fin main
} //fin DivisionByZero
```

- b) La siguiente clase muestra cómo utilizar el programa anterior para controlar otro conjunto de errores básicos en el proceso de división de dos números. Los errores controlados son: división por cero (`ArithmeticException`), la de formato erróneo en los argumentos (`NumberFormatException`), o un número inadecuado de argumentos (`ArrayIndexOutOfBoundsException`).

```
public class DivisionConExcepciones {
    public static void main (String[] args){
        //los argumentos de la división se pasan como parámetros
        int numerador, divisor, resultado = 0;
        try{
            numerador = Integer.parseInt(args[0]);
            divisor = Integer.parseInt(args[1]);
            resultado = numerador/divisor;
            System.out.println("Resultado de la división:: "+resultado);
        }catch(NumberFormatException e){
            System.out.println("Excepción, alguno de los argumentos
                               NO es un número:: "+e);
        }catch(ArithmeticException e){
            System.out.println("Excepción, división por cero:: "+e);
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("Excepción, error en el número
                               de argumentos:: "+e);
        }
    }
} //fin main
} //fin DivisionConExcepciones
```


El resultado de la ejecución de los ejemplos propuestos en los enunciados a) y b) sería:

1. Resultado de la división:: 2
2. Excepción división por cero:: `java.lang.ArithmeticException: / by zero`
3. Excepción, alguno de los argumentos **NO** es un número:: `java.lang.NumberFormatException: 4f6`
4. Excepción, error en el número de argumentos:: `java.lang.ArrayIndexOutOfBoundsException`

De esta solución deben mencionarse dos curiosidades:

1. En primer lugar, si se transformasen los argumentos de entrada a otro tipo numérico como `float` o `double`, las excepciones consideradas no aparecen, ¿por qué? Si se prueba a cambiar las operaciones que transforman las cadenas leídas en los argumentos a números enteros (`Integer.parseInt(argumento)`) por `Float.parseFloat(argumento)` (una vez declarados los atributos como `float`), cuando se intenta dividir entre cero, el programa responde con el valor **Infinity** y no genera ninguna excepción. Si se le pasa un argumento que no puede transformarse a un formato numérico (como la cadena "4f6"), responde asignando a la variable con un valor **NaN** (*Not a Number*) sin generar tampoco ninguna excepción. ¿Qué ha sucedido realmente?, ¿han dejado de funcionar las excepciones? La respuesta a este peculiar comportamiento tiene que ver con el formato de representación numérica utilizada en Java para los números en coma flotante. El estándar utilizado conocido por IEEE-754 dispone de representaciones (no numéricas) para situaciones especiales. En particular, este formato permite representar casos como infinito, o una representación no numérica; debido a ello, si se utilizasen parámetros de tipo `float` o `double`, esas excepciones no aparecerían simplemente porque no existen.
2. En segundo lugar, si se transforman los dos argumentos proporcionados al programa (`args[0]` y `args[1]`) fuera del `try/catch` el programa sigue funcionando exactamente igual. Pero cuando se produce una excepción de tipo `NumberFormatException` no son capturadas por el `catch` definido en el programa, sino que directamente es pasado al intérprete de Java. Esto es debido, a que las excepciones consideradas en este ejemplo son de tipo *no capturadas* (todas derivan de `RuntimeException`). Es decir, al situar ambas sentencias fuera de bloque `try/catch` se considera que esa posible excepción no se capturará, por lo que finalmente es la máquina virtual de Java la encargada de gestionar la excepción (como puede comprobarse, si se comenta el `try/catch`).

Ejercicio 2.2. Un divisor gráfico con control de errores. Implementación de un applet simple (propuesta de práctica)

Enunciado A partir del ejercicio anterior, implementar un **applet** en Java que realice la operación de división (suponer en este caso que los valores leídos son enteros, sin embargo, el resultado debe calcularse con un `double`). La salida gráfica del applet se muestra en la siguiente figura. Se dispondrá de dos campos de texto donde se introducen los valores del numerador y del denominador. El applet utilizará su propio tipo de excepciones (`DividePorCeroException`) que permitirá

comprobar el error de división entre cero, para el resto de errores se utilizarán las excepciones estándares de Java.

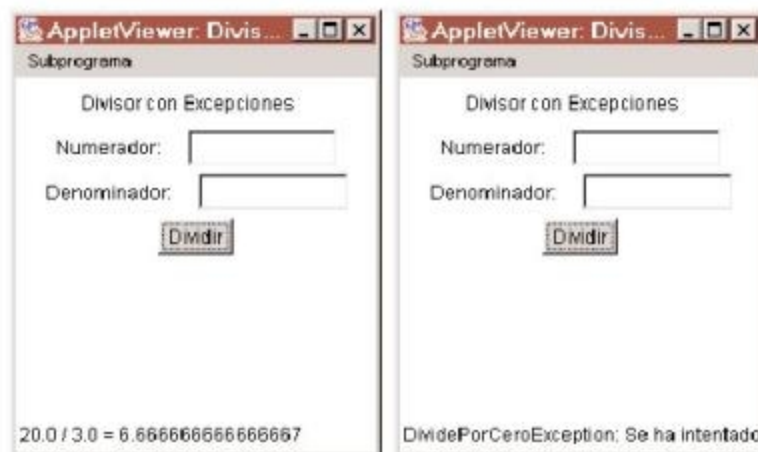


Figura 2.3. Applet que permite la división de dos números.

Solución

En primer lugar, se definirá una excepción propia que simplemente muestre un mensaje en el applet si se produce un error en la operación de división. Para ello se creará una clase que derive de `ArithmeticException` puesto que se trata de un caso particular de error aritmético. Como puede verse en la clase `DividePorCeroException`, únicamente se ha definido el constructor y se le pasa una cadena de texto al constructor de la clase padre.

```
public class DividePorCeroException extends ArithmeticException {
    public DividePorCeroException() {
        super("Se ha intentado dividir por cero....");
    }
}
//fin DividePorCeroException
```

Aunque este libro no estudia el diseño y construcción de aplicaciones gráficas con Java¹, deben darse unos conocimientos mínimos para permitir el desarrollo de este tipo de programas. En primer lugar, es necesario saber que es posible desarrollar dos tipos de programas totalmente diferentes en Java:

- Los primeros, denominados **aplicaciones** (la mayor parte de los programas mostrados en este libro son en realidad aplicaciones), se caracterizan por disponer de un método de ejecución llamado `main()`. Este método es el punto de arranque (o ejecución) para el programa. Sin embargo, cuando una aplicación es *gráfica* utilizará principalmente dos conjuntos de librerías; `java.awt` y/o `javax.swing`.

¹ Algunos de los libros referenciados utilizados como [Eckel02] o [Naughton97] tienen capítulos que estudian el desarrollo de aplicaciones gráficas en Java.

- El segundo tipo de programa en Java es el denominado **applet**. Los applets son programas (gráficos) almacenados y disponibles desde un servidor de Internet. Se transmiten a través de la red, se instalan y se ejecutan como parte de una página web en una máquina cliente. Este tipo de programas puede caracterizarse por:
 1. **No disponen** de un método `main()` (en su lugar pueden utilizar otros métodos como `init()`, `start()`, etc.).
 2. Estos programas necesitan extender (heredar) de la clase `java.applet.Applet`.
 3. Todo Applet es un programa gráfico y puede ser ejecutado por una aplicación denominada **appletviewer** (visor de Applets), o por cualquier navegador (NetScape, Explorer, etc.). De hecho, un applet es un programa creado para ser empotrado dentro de una página web que posteriormente será visualizada por un navegador (el `appletviewer` es un entorno gráfico muy simple que permite visualizar el aspecto del applet antes de dejarlo disponible para que pueda ser ejecutado por el navegador).
 4. Dado que un applet no deja de ser un trozo de código que se descarga y ejecuta dentro de un computador, las restricciones de seguridad sobre el applet son muy severas, restringiendo muy seriamente el conjunto de operaciones realizables por el applet una vez es descargada y ejecutada por el navegador.
 5. Todo applet implementa lo que se conoce por **ciclo de vida**, es decir, debe ejecutar un conjunto de métodos en un determinado orden. La siguiente figura muestra el ciclo de vida de cualquier applet:

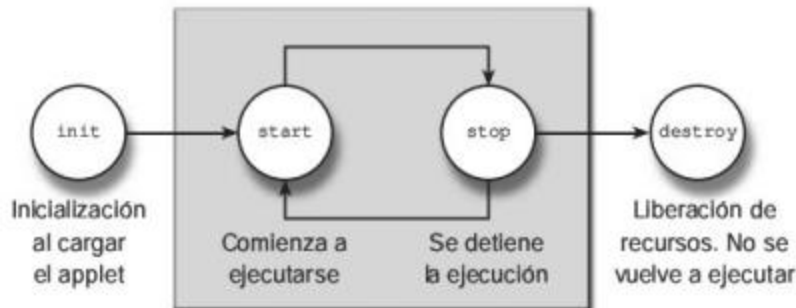


Figura 2.4. Ciclo de vida de un applet en Java.

Cada uno de los estados del ciclo de vida del applet corresponde a un método que puede ser implementado por el programa; los métodos que pueden utilizarse son:

- `init()`. Debe inicializar las variables. Sólo se ejecuta una vez durante el tiempo que se ejecuta el applet.
- `start()`. Se ejecuta cada vez que se muestra en pantalla el documento HTML que contiene el applet.
- `paint()`. Se ejecuta cada vez que se debe volver a pintar la salida de un applet. Tiene un parámetro de tipo `Graphics` que contiene el contexto gráfico en el que se ejecuta el applet.

- `stop()`. Se ejecuta cuando el navegador deja la página HTML. Suspende los hilos no necesarios cuando la applet no es visible.
- `destroy()`. Se ejecuta cuando el applet se borra de la memoria. Libera todos los recursos que utiliza el applet.

Debido a que este tipo de programas Java son gráficos, su funcionamiento estará muy relacionado con otros conjuntos de librerías utilizables en Java para construir elementos gráficos (`java.awt`, `javax.swing`) como botones, cuadros de texto, barras de desplazamiento, *checkbox*, etc. La correcta implementación y el funcionamiento de cualquier applet podría resumirse en:

a) **Implementación** de un applet:

- Debe importar los paquetes `java.applet` y `java.awt`.
- Se ejecuta utilizando un navegador (NetScape, Explorer, etc.) o un visualizador de applets (`appletviewer`).
- Es un programa basado en ventanas y en **eventos** (*cambios, acciones*) que el usuario produce sobre los elementos gráficos del applet.

b) **Funcionamiento** de un applet:

1. Se produce un evento² (suceso o acción sobre algún elemento del applet).
2. AWT notifica al applet el evento llamando a un gestor de eventos que el applet sobrescribe.
3. El applet hace algo y devuelve el control al AWT.

Una vez descritas muy brevemente las principales características de un applet, puede construirse uno de estos programas que resuelva el problema planteado. Se implementará únicamente un método `init()` que permitirá cargar los valores iniciales del mismo, y un método `action()` que permitirá responder ante los eventos que se produzcan en cualquiera de los elementos gráficos que forman el applet. La clase `DivisonConExcepcionesApplet` muestra cómo puede implementarse un applet que calcula la división de dos números.

```
import java.awt.*;
import java.applet.Applet;
public class DivisonConExcepcionesApplet extends Applet{
    //Atributos del applet
    Label  etiqeta1, etiqeta2, titulo;
    TextField campo1, campo2;
    Button boton;
    double numerol, numero2;
    double resultado;
```

² El evento está encapsulado por la clase `java.awt.Event`.

```
//inicialización del applet
public void init() {
    titulo = new Label("Divisor con Excepciones");
    etiqueta1 = new Label("Numerador:");
    campo1 = new TextField(10);
    etiqueta2 = new Label("Denominador:");
    campo2 = new TextField(10);
    boton = new Button("Dividir");
    //añadimos los campos
    add(titulo);
    add(etiqueta1);
    add(campo1);
    add(etiqueta2);
    add(campo2);
    add(boton);
}

//se procesan los eventos del interfaz gráfico del Applet
public boolean action(Event evento, Object objeto) {
    if (evento.target == boton) {
        try { //código a controlar
            numero1 = Double.parseDouble(campo1.getText());
            campo1.setText("");
            numero2 = Double.parseDouble(campo2.getText());
            campo2.setText("");
            resultado = cociente(numero1, numero2);
            showStatus(numero1 + " / " + numero2 + " = " +
                Double.toString(resultado));
        } catch (DividePorCeroException e) {
            showStatus(e.toString());
        } catch (NumberFormatException e) {
            showStatus(e.toString());
        }
    }
    return true;
}

//metodo cociente, se lanza la excepción si se detecta
public double cociente(double numerador, double denominador)
    throws DividePorCeroException {
    if (denominador == 0)
        throw new DividePorCeroException();
    return (double) numerador / denominador; //transforma a double
} //cociente
} //DivisionConExcepcionesApplet
```


En este caso debe observarse que no es necesario controlar la excepción `ArrayIndexOutOfBoundsException` debido a que en el ejemplo los valores son leídos de un conjunto de campos de texto, o `TextField`, (`Double.parseDouble(campo1.getText())`) y no del array de argumentos de entrada al programa (`String[] args`). Para leer los valores de tipo `String` se utilizará el método `getText()`; una vez leídos, los campos de texto se limpian (es decir, se borra su contenido) utilizando el método `setText(" ")`. En el peor de los casos uno de los campos podría no contener ningún tipo de información, y se generaría un error de conversión numérica (dado que trata de transformarse un carácter especial en un número), la excepción sería capturada y se mostraría el correspondiente error a través de la barra de estado del visor de applets o del navegador (`showStatus(e.toString())`). Finalmente, debe observarse que cuando se produce el evento sobre el botón (es decir, el usuario lo pulsa), es capturado y utilizado para decidir cuándo se realizan todas las operaciones (`evento.target == boton`). El método `cociente(numerador, denominador)` es el encargado de calcular la división de los parámetros leídos; en el caso de que el denominador sea cero lanzará explícitamente la correspondiente excepción.

El anterior ejemplo podría ejecutarse directamente utilizando el visor de applets desde la línea de comandos ejecutando:

```
c:\> appletviewer DivisionConExcepcionesApplet
```

o descargando el applet insertado dentro de una página web en un navegador. Para lograr que el applet sea ejecutado por un navegador, es necesario crear una página HTML que indique al navegador dónde se encuentra el código ejecutable del applet (`DivisorConExcepciones.class`). Para el ejemplo anterior, se podría crear la siguiente página que permitirá descargar y ejecutar el applet creado.

```
<HTML>
<HEAD>
  <TITLE>Divisor con Excepciones</TITLE>
</HEAD>
<BODY>
<H3><HR WIDTH="100%">Divisor con Excepciones<HR WIDTH="100%"></H3>
<P>
<APPLET code="DivisonConExcepcionesApplet.class"
  width=350 height=200> </APPLET>
</P>
</BODY>
</HTML>
```

Esta página HTML contiene una etiqueta llamada `<applet>` que permite indicar (mediante su campo `code`) dónde se encuentra el código Java a ejecutar. Dado que no se indica ninguna ruta para la clase correspondiente, se supone que el código compilado del applet (`DivisonConExcepcionesApplet.class`), y la página HTML, se encuentran almacenadas en el mismo directorio. La salida gráfica en el navegador se muestra en la siguiente figura.



Figura 2.5. Applet generado para el divisor con excepciones.

Ejercicio 2.3. Desarrollo de una calculadora básica (propuesta de práctica)

Enunciado Utilizando el ejercicio anterior, mejorar la funcionalidad del applet implementado e introducir nuevas funciones aritméticas básicas como: suma, resta, multiplicación o división. Utilizar un campo extra de texto que permita mostrar el resultado de las operaciones en lugar de utilizar la barra de estado.

Solución La salida gráfica del applet que se implementará se muestra en las siguientes figuras, en ellas se han incluido un conjunto básico de operaciones aritméticas como se pedía en el enunciado.



Figura 2.6. Applet calculadora.

El código del applet que implementa la calculadora es una modificación muy simple del ejemplo anterior, donde la excepciones (en caso de producirse) se muestran también en el campo del resultado (`campo3.setText(e.toString())`) y sólo el método `cociente` lanza excepciones de tipo `DivisionPorCeroException`.

```
import java.applet.Applet;
import java.awt.*;
public class AppletCalculadora extends Applet {
    //Atributos del applet
    Label   etiqueta1, etiqueta2, etiqueta3;
    TextField campo1, campo2, campo3;
    Button boton1, boton2, boton3, boton4;
    double numero1, numero2;
    double resultado;
    public void init() {
        etiqueta1 = new Label("Operando 1:");
        campo1 = new TextField(10);
        etiqueta2 = new Label("Operando 2:");
        campo2 = new TextField(10);
        etiqueta3 = new Label("Resultado:");
        campo3 = new TextField(10);
        boton1 = new Button("Sumar");
        boton2 = new Button("Restar");
        boton3 = new Button("Multiplicar");
        boton4 = new Button("Dividir");
        //añadimos los campos
        add(etiqueta1);
        add(campo1);
        add(etiqueta2);
        add(campo2);
        add(etiqueta3);
        add(campo3);
        add(boton1);
        add(boton2);
        add(boton3);
        add(boton4);
    }
    //procesamos los eventos de la GUI
    public boolean action(Event evento, Object objeto){
        try{
            numero1 = Double.parseDouble(campo1.getText());
            numero2 = Double.parseDouble(campo2.getText());
            if (evento.target == boton1)
                resultado = suma(numero1, numero2);
            if (evento.target == boton2)
                resultado = resta(numero1, numero2);
```

```

        if (evento.target == boton3)
            resultado = multiplicacion(numero1,numero2);
        if (evento.target == boton4)
            resultado = cociente(numero1,numero2);
        campo3.setText(Double.toString(resultado));
    }catch (DividePorCeroException e){
        campo3.setText(e.toString());
    }catch (NumberFormatException e){
        campo3.setText(e.toString());
    }
    return true;
} //fin action
//metodos de cálculo
public double suma (double numerador, double denominador){
    return numerador + denominador;
}
public double resta (double numerador, double denominador){
    return numerador - denominador;
}
public double multiplicacion (double numerador, double denominador){
    return numerador * denominador;
}
public double cociente (double numerador, double denominador)
    throws DividePorCeroException{
    if (denominador == 0)
        throw new DividePorCeroException();
    return (double)numerador/denominador;
}
} //fin AppletCalculadora

```

Como puede verse en este ejemplo, el método `action()` se utiliza para controlar los eventos producidos, en función del botón pulsado se ejecutará un método u otro (`evento.target == boton1`, `boton2`, etc.); es importante ver que ahora la salida del resultado no se realiza a través de la barra de estado del applet, sino que se utiliza el método `setText()` (`campo3.setText(Double.toString(resultado))`) para mostrar el resultado. La página HTML que podría utilizarse para descargar el contenido del applet en el navegador, junto con la salida mostrada en el mismo, se muestra a continuación:

```

<HTML>
<HEAD>
    <TITLE>Applet HTML Page</TITLE>
</HEAD>
<BODY>
<H3><HR WIDTH="100%">Applet HTML Page<HR WIDTH="100%"></H3>

```

```

<P>
<APPLET code="AppletCalculadora.class" width=350 height=200></APPLET>
</P>
</BODY>
</HTML>

```



Figura 2.7. Applet calculadora cargado en NetScape.

Como puede verse en este ejemplo todavía quedan muchas cuestiones abiertas acerca del desarrollo de applets. Por ejemplo, sería interesante lograr que la distribución de los objetos pudiese ser más uniforme, o permitir capturar una mayor cantidad de eventos o acciones realizables por el usuario, para de esa forma dotar de una mejor funcionalidad al programa desarrollado. Sin embargo, y dado que no es el objetivo principal de este capítulo, se recomienda a aquellas personas que deseen profundizar en la implementación de este tipo de programas la consulta de alguna de las referencias recomendadas en el libro como [Eckel02] o [Naughton97].

2.6.2. Ejercicios de redes

Ejercicio 2.4. Acceso automático a máquinas conectadas a una red

Enunciado Implementar un programa en Java capaz de recuperar la información disponible sobre una máquina conectada a una red como Internet. El programa aceptará como parámetro el nombre de la máquina en forma de URL, o como una dirección IP, y mostrará por consola (pantalla) la información relativa a la máquina local donde se está ejecutando el programa. Debe tenerse en cuenta que algunos de los nombres simbólicos utilizados por estas máquinas como `www.yahoo.com` utilizan varias direcciones IP diferentes; en ese caso, el programa deberá mostrar todas las posibles direcciones en caso de que existan.

Nota: se recomienda estudiar el paquete `java.net`, en particular la clase `InetAddress`.

Ejemplos

```
c:\> java CapturarDireccionesIP www.LordOfTheRings.com
c:\> java CapturarDireccionesIP 163.117.129.183
c:\> java CapturarDireccionesIP www.jahoo.com
```

Solución El siguiente conjunto de problemas trabajará sobre algunas de las clases disponibles en el paquete **java.net**. El paquete `java.net` proporciona los mecanismos básicos para trabajar con computadores conectados a una red. Java ofrece un conjunto de capacidades que facilitan el desarrollo de aplicaciones que pueden ser ejecutadas sobre máquinas conectadas a redes como Internet. La comunicación entre estos programas está basada en **sockets**. Una conexión basada en sockets permite a las aplicaciones manejar el trabajo en red como si fuera E/S sobre archivos (lectura/escritura de sockets). Es decir, uno de los computadores proporciona un flujo de salida de datos y el otro dispone de un flujo de entrada para poder leer esa información, Java permite crear dos tipos de sockets:

- **Sockets de flujo.** Conexión permanente entre dos procesos, ofrecen un servicio orientado a conexiones. Se implementa el protocolo TCP (*Transmission Control Protocol*).
- **Sockets de datagrama.** Se transmiten paquetes individuales de información, ofrecen un servicio sin conexiones. Se implementa el protocolo UDP (*User Datagram Protocol*).

En particular se desarrollarán problemas utilizando las siguientes clases de `java.net`: `HttpConnection`, `InetAddress`, `ServerSocket`, `Socket`, `URL`, `URLConnection`. Las excepciones contenidas en este paquete (y que deben utilizarse cuando se crean programas para trabajar en redes como Internet) son: `BindException`, `ConnectException`, `MalformedURLException`, `NoRouteToHostException`, `ProtocolException`, `SocketException`, `UnknownHostException`, `UnknownServiceException`.

Retomando el enunciado del ejercicio, y para resolver el problema planteado se utilizará la clase `InetAddress` que permite encapsular el concepto de dirección IP numérica y el nombre de dominio (cadena de texto que permite recordar de forma más simple la dirección numérica de la máquina) de esas direcciones. Por ejemplo:

- Dirección IP: 64.12.47.243
- Nombre Dominio: `www.theMatrix.com`

Es importante observar que desde el punto de vista de Java, `InetAddress` es una clase extraña; esto es debido a que no dispone de constructores (por tanto, esta clase se aparta de la filosofía de la programación orientada a objetos). Se utiliza un conjunto de métodos estáticos que devuelven una instancia de la clase. Por tanto, si se desea construir un objeto de tipo `InetAddress` será necesario que la aplicación utilice cualquiera de los siguientes métodos estáticos: `getLocalHost()`, `getByName()` o `getAllByName()`. La Tabla 2.1. muestra algunos de los métodos disponibles y sus funciones.

Las excepciones que los métodos estáticos (utilizados para crear objetos de este tipo) lanzan son:

- **UnknownHostException.** La excepción se lanza si la dirección IP no existe.
- **SecurityException.** En caso de existir un gestor de seguridad puede no permitirse la operación de conexión de la aplicación a la máquina correspondiente.

Tabla 2.1. Algunos métodos `java.net.InetAddress`.

Método	Función
<code>static InetAddress getLocalHost()</code>	Obtiene el host, o nombre de la máquina a la que se accede.
<code>static InetAddress getByName (String nomNodo)</code>	Obtiene el nombre de dominio.
<code>static InetAddress[] getAllByName (String nomNodo)</code>	Lee todas las direcciones que representan un determinado nombre de dominio.
<code>String getHostName ()</code>	Nombre del nodo asociado al objeto <code>InetAddress</code> .
<code>byte[] getAddress ()</code>	Matriz de bytes de cuatro elementos que representan la dirección IP del objeto.
<code>String toString ()</code>	Devuelve una cadena que contiene el nombre del nodo y la dirección IP, por ejemplo: <<www.starwars.com/206.251.0.173>>
<code>boolean equals(InetAddress otroNodo)</code>	Devuelve true si el objeto tiene la misma dirección IP que <i>otroNodo</i> .

El siguiente programa resuelve de forma simple el problema planteado, utilizando el método `getLocalHost()` que permite recuperar la información asociada a la máquina a la que se está accediendo en la red. Una vez creado el objeto `InetAddress` puede accederse al nombre de dominio y a la dirección IP de la máquina utilizando el método `toString()`. Cuando se le pasa un nombre de dominio o una dirección IP por parámetro al programa, es posible utilizar otros métodos como `getByName()`, o `getAllByName()`, para acceder al resto de información que se pedía en el enunciado del ejercicio.

```
import java.net.*;

public class CapturarDireccionesIP {
    public static void main(String args[]) throws UnknownHostException{
        //máquina local
        InetAddress direccionIP = InetAddress.getLocalHost();
        System.out.println ("La máquina local tiene la siguiente
                               dirección IP: "+direccionIP.toString());
        //argumento que se le ha pasado al programa
        direccionIP = InetAddress.getByName(args[0]);
        System.out.println ("encontrarás el servidor de
                               +direccionIP.getHostName()
                               +" en :"+
                               +direccionIP.toString());
        InetAddress VariasDirecciones[] = InetAddress.getAllByName(args[0]);
        for (int i=0; i < VariasDirecciones.length; i++)
            System.out.println(VariasDirecciones[i]);
    }
}
//fin main
}
//fin CapturarDireccionesIP
```

La salida del programa anterior, mostraría la siguiente información para las diferentes máquinas que se ha pasado como direcciones URL:

```
La máquina local tiene la siguiente dirección IP: merlin.uc3m.es/163.117.129.183
//args[0] = www.LordOfTheRings.com
encontrarás el servidor de
en :www.LordOfTheRings.com/198.74.34.22
//args[0] = www.yahoo.com
www.yahoo.com/64.58.76.176
www.yahoo.com/64.58.76.177
www.yahoo.com/64.58.76.178
www.yahoo.com/64.58.76.179
www.yahoo.com/64.58.76.222
www.yahoo.com/64.58.76.223
www.yahoo.com/64.58.76.224
www.yahoo.com/64.58.76.225
www.yahoo.com/64.58.76.227
www.yahoo.com/64.58.76.228
www.yahoo.com/64.58.76.229
www.yahoo.com/64.58.76.230
```

Si se prueba a introducir una dirección de IP errónea, o un nombre de dominio inexistente el programa generará la siguiente excepción:

```
La máquina local tiene la siguiente dirección IP: merlin.uc3m.es/163.117.129.183
java.net.UnknownHostException: www.fallo.com
    at java.net.InetAddress.getAllByName0(InetAddress.java:566)
    at java.net.InetAddress.getAllByName0(InetAddress.java:535)
    at java.net.InetAddress.getByName(InetAddress.java:444)
    at CapturarDireccionesIP.main(CapturarDireccionesIP.java:18)
Exception in thread "main"
```

Ejercicio 2.5. Acceso a una dirección URL

Enunciado Implementar un programa en Java que permita acceder a una dirección web (URL) y consultar la información básica de la misma como: el nombre de la máquina donde se encuentra almacenada (host), protocolo que utiliza la URL, o el puerto de la máquina.

Solución Todo recurso en Web se puede identificar mediante una URL (*Uniform Resource Locator*, identificador de recursos uniforme) que *identifica de forma única* dónde se encuentra ese recurso dentro de la Red. El formato de una URL, está formado por cuatro elementos:

- **Protocolo** http, ftp, gopher, file, https, etc.
- **Nombre del nodo o dirección IP.**

- **Número de puerto** (normalmente el 80).
- **Nombre del archivo real** (por defecto index.html).

Por lo tanto, cualquier URL tendrá el siguiente aspecto:

protocolo://NombreHost[:número_puerto]/nombre_archivo

Java define dos clases que nos permiten acceder y manipular los diferentes recursos que existen en la Web: **URL**, y **URLConnection**. La clase URL dispone de seis constructores y todos pueden lanzar la excepción de tipo **MalformedURLException**. A continuación, se muestra la sintaxis de algunos de estos constructores y métodos disponibles para la clase URL.

- **Constructores:**

```
URL(String nombre)
URL(String protocolo,String nodo,int puerto, String archivo)
URL(String protocolo,String nodo, String archivo)
URL(URL contexto,String nombre)
```

Tabla 2.2 Algunos métodos java.net.Innet.URL.

Método	Función
getProtocol()	Captura el protocolo utilizado por la URL.
getPort()	Puerto utilizado por la URL, devuelve -1 si el puerto no está asignado. El puerto por defecto es el 80 para el protocolo http.
getHost()	Nombre del servidor.
getFile()	Nombre del fichero html que se visualiza.
toExternalForm()	Construye un String que representa la URL.

Una vez descritas brevemente las principales características de la clase URL, el siguiente programa utiliza dos de los constructores de la clase para acceder a diferente información de las páginas web que son pasadas por parámetro o directamente como una cadena de texto.

```
public class EjemploURL {
    public static void main(String args[]) throws MalformedURLException{
        URL paginal = new URL(args[0]); //args[0] = http://www.uc3m.es
        URL pagina2 = new URL("http","scalab.uc3m.es",80,"index.html");
        //Primera Prueba
        System.out.println("Primera pagina");
        System.out.println("Protocolo: "+paginal.getProtocol());
        System.out.println("Puerto: "+paginal.getPort());
        System.out.println("Host: "+paginal.getHost());
        System.out.println("Archivo: "+paginal.getFile());
    }
}
```

```

        System.out.println("External form: "+pagina1.toExternalForm());
        //Segunda prueba
        System.out.println("Segunda pagina");
        System.out.println("Protocolo: "+pagina2.getProtocol());
        System.out.println("Puerto: "+pagina2.getPort());
        System.out.println("Host: "+pagina2.getHost());
        System.out.println("Archivo: "+pagina2.getFile());
        System.out.println("External form: "+pagina2.toExternalForm());
    } //fin main
} //fin EjemploURL

```

Como puede verse en la clase EjemploURL, se han empleado diversos métodos como `getProtocol()`, `getPort()`, o `getFile()`, para acceder a la información solicitada en el enunciado. La salida de este programa sería:

```

Primera pagina
Protocolo: http
Puerto: -1 //el puerto no está asignado
Host: www.uc3m.es
Archivo:
External form: http://www.uc3m.es
Segunda pagina
Protocolo: http
Puerto: 80
Host: scalab.uc3m.es
Archivo: index.html
External form: http://scalab.uc3m.es:80index.html

```

Ejercicio 2.6. Extracción de información de una página web

Enunciado Implementar un programa en Java que permita acceder a una página web y obtener información sobre el fichero o recurso físico relacionado con la URL correspondiente. El programa deberá mostrar datos como la fecha de la última modificación, el tamaño del fichero HTML asociado y también deberá mostrar por pantalla el fichero HTML que corresponde a la página descargada.

Solución Una vez se conoce cómo se encapsula el concepto de **URL** en Java, es necesario poder abrir una conexión si se desea acceder a la información almacenada en esa dirección. La clase `URLConnection` permite acceder a los atributos de un recurso remoto. Una vez establecida la conexión, pueden examinarse las propiedades del objeto remoto antes de transferirlo. Sólo tiene sentido para objetos URL que utilizan el **protocolo HTTP**.

La clase `URLConnection` dispone de un único constructor que necesita como parámetro un objeto de tipo URL. Luego, para poder establecer una conexión con una página web, será necesario crear previamente uno de estos objetos. Algunos de los métodos disponibles en `URLConnection` aparecen descritos en la Tabla 2.3.

Tabla 2.3 Algunos métodos `java.net.Innet.URLConnection`.

Método	Función
<code>openConnection()</code>	Sobre un objeto URL, abre una conexión.
<code>getExpiration()</code>	Obtiene la fecha y hora en la que ese recurso quedará obsoleto.
<code>getContentType()</code>	Devuelve el tipo MIME del recurso.
<code>getContentLength()</code>	Tamaño en bytes del recurso.
<code>getDate()</code>	Fecha de la creación de la página.
<code>getInputStream()</code>	Buffer de entrada.

Como puede verse en el siguiente programa, el método `main()`; deberá lanzar dos excepciones; la primera es debida a que se va a utilizar un flujo de E/S para mostrar por pantalla el contenido del fichero html (throws `IOException`). La segunda es debida a la posibilidad de que la URL considerada no exista (throws `MalformedURLException`).

Una vez creado el objeto de tipo URL (new `URL("http://scalab.uc3m.es/")`), y abierta la conexión contra la página (nodo.`openConnection()`), es posible acceder a información concreta del recurso como la fecha de la última modificación (Conex_nodo.`getLastModified()`), el tipo MIME (*Multipurpose Internet Mail Extensions*) del recurso (página html, ficheros, imágenes, vídeo, música, etc.), referenciado por la URL (Conex_nodo.`getContentType()`), o el tamaño en bytes de la página (Conex_nodo.`getContentLength()`). Finalmente, se utiliza la conexión abierta como un flujo de entrada que es dirigido contra la salida estándar³.

```
public class EjemploURLConnection {
    public static void main(String args[]) throws IOException,
                                                MalformedURLException{

        int c;
        URL nodo = new URL("http://scalab.uc3m.es/");
        URLConnection Conex_nodo= nodo.openConnection();
        System.out.println("Fecha: "+new Date(Conex_nodo.getDate()));
        System.out.println("Contenido: "+Conex_nodo.getContentType());
        System.out.println("Expira: "+Conex_nodo.getExpiration());
        System.out.println("Última modificación: "
                           +new Date(Conex_nodo.getLastModified()));
        int longitud = Conex_nodo.getContentLength();
        System.out.println("Longitud del contenido: "+longitud);
        //mostrar el contenido del fichero HTML
    }
}
```

³ El Capítulo 5 "Programación de ficheros en Java" muestra con detalle cómo se realizan este tipo de procesos en Java.


```

    if (longitud > 0){
        System.out.println("##### CONTENIDO #####");
        InputStream entrada = Conex_nodo.getInputStream();
        int i = longitud;
        while ((c = entrada.read()) != -1) && (--i > 0)){
            System.out.print((char) c);
        }
        entrada.close(); //se cierra el flujo
    } else {
        System.out.println("Contenido no disponible");
    }
} //fin main
} //fin Ejemplo URLConneciton

```

La ejecución del anterior programa mostraría:

```

Fecha: Wed Jan 08 19:50:41 CET 2003
Contenido: text/html
Expira: 0
Última modificación: Sat Feb 24 13:28:56 CET 2001
Longitud del contenido: 173
##### CONTENIDO #####
<html>
<head>
<meta http-equiv="refresh" content="0;url=http://scalab.uc3m.es/~scalab">
<!meta http-equiv="refresh" content="0;url=http://scalab.uc3m.es/~docweb">
</head>

```

Como puede verse en la salida del programa, se ha utilizado la clase `java.util.Date` para construir un objeto de tipo fecha y mostrar de esa forma una fecha en un formato comprensible por el usuario. Para construir estos objetos se han utilizado los correspondientes métodos de `URLConnection` que devuelven un valor `long` que representa la fecha; ese valor es posteriormente utilizado para construir los objetos de tipo `Date`.

Ejercicio 2.7. Conexiones HTTP seguras (propuesta de práctica)

Enunciado Implementar un programa en Java que permita una conexión segura con una página web, el programa solicitará por teclado la URL a la que se quiere acceder. Si el recurso no existe, o aparece cualquier error identificable mediante un **código de respuesta**, el programa mostrará un mensaje de aviso. Los códigos de respuesta son números enteros que se utilizan para suministrar información a la máquina que accede a la información, un número superior a 300 indica algún tipo de error en la URL; por ejemplo, el código 404 se utiliza para indicar que ese recurso no existe.

Solución Para resolver el anterior problema se utilizará la clase `HttpURLConnection`, derivada de `URLConnection`, que permite modelar las conexiones HTTP sobre recursos web identificados

mediante una URL. Esta clase dispone de un único constructor que necesita un objeto de tipo URL como parámetro, la sintaxis del constructor es:

```
protected HttpURLConnection(URL u);
```

Esta clase dispone de una gran cantidad de atributos estáticos que permiten acceder a información sobre el protocolo http. A continuación, se muestran algunos de los métodos disponibles en la clase:

Tabla 2.4 Algunos métodos `java.net.Innet.HttpURLConnection`.

Método	Función
<code>String getRequestMethod()</code>	Devuelve el tipo de método (CGI, POST, etc.) utilizado en la conexión.
<code>int getResponseCode()</code>	Lee el código de respuesta (estado del servidor).
<code>String getResponseMessage()</code>	Muestra un mensaje asociado al código de respuesta.

El siguiente programa utiliza dos métodos estáticos, el primero `pedirURL()` devuelve un `String` que identifica el recurso al que se desea acceder. El segundo `comprobarURL(String url)` devuelve un valor entero que indicará si la URL es, o no, correcta (si el código es mayor de 300 se detecta algún problema). El programa principal se ejecuta hasta que el usuario teclea la cadena "fin". El siguiente programa muestra cómo se ha resuelto el problema.

```
import java.net.*;
import java.io.*;
public class ConexHTTPSicura {
    //atributos
    private static boolean URLCorrecta = true;
    private static String cadenaURL = "";
    private static int codigoRespuesta= 0;
    //métodos
    static String pedirURL() throws IOException{
        InputStreamReader flujoEntrada =new InputStreamReader(System.in);
        BufferedReader teclado = new BufferedReader(flujoEntrada);
        System.out.println("Dame una URL (fin para terminar)");
        return teclado.readLine();
    }//fin pedir URL

    static int comprobarURL(String cadURL) {
        try{
            URL u = new URL(cadURL); //se construye el objeto URL
            //se abre la conexión
            HttpURLConnection uConn = (HttpURLConnection) u.openConnection();
```



```
Dame una URL (fin para terminar)
http://www.wlw.es
Método utilizado en la conexión: GET
Tipo MIME del recurso: null
Host desconocido....:java.net.UnknownHostException: wlwlwlw.es
La URL introducida: http://wlwlwlw.es ha generado algún error
Codigo de respuesta: 901

Dame una URL (fin para terminar)
http://www.lordOfTheRings.com
URL mal formada....:java.net.MalformedURLException: unknown protocol: http
La URL introducida: http://www.lordOfTheRings.com ha generado algún error
Codigo de respuesta: 900

Dame una URL (fin para terminar)
http://www.LordOfTheRings.com
Método utilizado en la conexión: GET
Tipo MIME del recurso: text/html
Codigo de respuesta: 200
Mensaje asociado al código de respuesta: OK
Dame una URL (fin para terminar)
fin
FIN conexión HTTP segura
```

Como puede verse en la salida, la primera de las cadenas (`http://www.wlw.es`) que es una URL teóricamente bien formada genera una excepción debido a que ese servidor no existe; la segunda cadena (`http://www.lordOfTheRings.com`) genera un error de tipo `MalformedURLException`, debido a que el protocolo `http` no existe. Finalmente, la última cadena (`http://www.lordOfTheRings.com`) corresponde a una URL correcta, por lo que puede accederse a la información de esa URL.

Algoritmos sobre arrays

3.1. Introducción

En la vida diaria, el concepto de “conjunto ordenado de elementos” tiene una gran importancia. Piénsese, por ejemplo, en actividades tan cotidianas como buscar un número de teléfono en la guía telefónica, para lo cual es necesario localizar a una persona concreta entre cientos de miles, y sin embargo, es algo que se hace con cierta rapidez. Por supuesto, esta búsqueda se facilita enormemente gracias a que los nombres contenidos en la guía telefónica están ordenados alfabéticamente. Lo mismo ocurre con diccionarios, enciclopedias, ficheros de bibliotecas, etc. Es posible imaginar lo extraordinariamente lenta que sería la búsqueda manual de datos si las guías telefónicas, diccionarios, etc., no estuvieran ordenados.

Por la misma razón puede pensarse que para que la búsqueda de datos computacionales sea eficiente es importante que los datos estén ordenados. Los programas de computación emplean una parte significativa de su tiempo realizando operaciones de búsqueda de datos. Por tanto, las actividades de búsqueda y ordenación son fundamentales en programación de computadores. Estas operaciones normalmente se realizan en arrays o ficheros.

En este capítulo se tratarán algunos de los algoritmos de búsqueda y ordenación más utilizados con arrays. Como se ha dicho, son métodos básicos y fundamentales, y están muy relacionados. Además se han incluido algunos algoritmos básicos de inserción de elementos en arrays tanto ordenados como no ordenados.

Aunque, generalmente, los datos se ordenan para poder realizar operaciones de búsqueda sobre ellos, en este capítulo se estudiarán primero los métodos de búsqueda porque son más sencillos, pasando después a ver los algoritmos de inserción y por último, los de ordenación que son más complicados y sofisticados.

Puede verse un análisis exhaustivo de los algoritmos de ordenación en [Wirth80] y en [Brassard90].

En este capítulo se utilizará la notación asintótica $O(f(n))$ para indicar la eficiencia de un algoritmo. Cuando se dice que la eficiencia de un algoritmo es $O(f(n))$ ("O grande de $f(n)$ "), o también que es **de orden $f(n)$** , significa que el tiempo que tarda en ejecutarse el algoritmo en función del número de elementos n es aproximadamente proporcional a $f(n)$ para un valor de n lo suficientemente grande. Por ejemplo, si la eficiencia de un determinado algoritmo de ordenación es $O(n^2)$ el tiempo de ejecución es aproximadamente proporcional al cuadrado del número de elementos a ordenar, de modo que si en ordenar 1.000 elementos se tarda 10 ms, en ordenar 2.000 se tardará aproximadamente 4 veces más, 40 ms.

No se ha dado una descripción formal de esta notación porque se sale fuera de los objetivos de este libro, pero un estudio formal de la eficiencia y de la notación asintótica se puede encontrar en [Brassard90].

3.2. Algoritmos de búsqueda

Aunque hay muchas formas de buscar datos en un array, en este capítulo se estudiarán las dos más importantes: la búsqueda secuencial y la búsqueda binaria.

La **búsqueda secuencial** puede realizarse en cualquier tipo de arrays (no necesita que estén ordenados), y es muy sencilla, pero es un método poco eficiente: si se quiere buscar secuencialmente un valor en un array de n elementos, en el peor caso deberán realizarse n comparaciones. Es decir, su eficiencia es $O(n)$. Por el contrario, la **búsqueda binaria** sólo puede utilizarse sobre arrays ordenados, y es algo más complicada, pero es altamente eficiente: sólo se necesitarán $\log_2 n$ comparaciones para buscar un valor en un array de n elementos utilizando este método, por tanto su eficiencia es $O(n \log n)^1$.

3.2.1. Búsqueda secuencial

En **arrays no ordenados**, la única búsqueda posible es la secuencial. Consiste en ir comparando cada elemento del array con el elemento a buscar, comenzando por el principio. Finalizará la búsqueda cuando se localice el elemento o cuando se llegue al final del array. Si se encuentra el elemento, se devolverá como resultado su posición; si no, se devolverá un código de error que indique que el elemento no se encuentra en el array. En el peor caso, deberá recorrerse toda la lista de valores. Es un método sencillo, pero poco eficiente. Su complejidad es $O(n)$.

Como analogía, piénsese en la búsqueda de una determinada palabra en un hipotético diccionario que no estuviera ordenado alfabéticamente: no quedará más remedio que ir buscando la palabra una a una desde el principio hasta el final.

En arrays ordenados también se puede aplicar este método, aunque se dejaría de buscar cuando se encontrara el elemento o cuando encontrara uno mayor, (en el caso de ordenación ascen-

¹ Se demuestra que $O(\log_a n) = O(\log_b n)$, siendo a y b dos bases cualesquiera. Por tanto, en la notación asintótica no se especificará la base del logaritmo.

dente). De todos modos, siempre es preferible utilizar la búsqueda binaria cuando los arrays están ordenados.

3.2.2. Búsqueda binaria

Los arrays ordenados permiten una búsqueda mucho más eficiente: **la búsqueda binaria**. En este método se aprovechará el hecho de que el array está ordenado, descartando en cada iteración la mitad de los elementos. La gran ventaja de este método es su eficiencia, es muchísimo más rápido que el anterior. Otra ventaja de este método es que aunque el elemento no se encuentre en el array, podemos determinar el lugar donde le correspondería estar si se quisiera insertar. Esta ventaja del método se utilizará más tarde en los algoritmos de inserción.

Se puede pensar en la siguiente analogía: buscar la palabra "Jaén" en un diccionario; abrimos el diccionario aproximadamente por la mitad, y vemos que nos encontramos con palabras que comienzan por "M", por lo que descartamos la segunda mitad del diccionario, y volvemos a dividir, ahora la primera mitad. Supongamos que nos encontramos con palabras que empiezan por "G", entonces descartaremos la primera mitad y nos centraremos en la segunda, y así sucesivamente hasta que en relativamente pocos pasos encontramos la palabra que estamos buscando.

El algoritmo consiste en lo siguiente: se examina primero el elemento que ocupa el centro de la lista, si es el que se busca ya se ha logrado la solución y si no, se determina si el elemento buscado está en la primera mitad (porque es inferior al elemento central), en cuyo caso se descarta la segunda mitad, o si está en la segunda mitad (porque es superior), descartando la primera mitad. Se repite el proceso con la sublista correspondiente hasta que se encuentre el elemento o bien hasta que se determine que no está porque se ha llegado a una sublista de 0 elementos.

3.2.3. Análisis de la eficiencia de los algoritmos de búsqueda

A continuación, se realizará un análisis básico de la eficiencia de los dos algoritmos de búsqueda estudiados. En ambos casos se estudiará la peor situación, que será aquella donde el elemento buscado no se encuentra en el array. Supondremos que al array tiene N elementos.

En la **búsqueda secuencial**, en el peor caso, el número de comparaciones que deberán realizarse será N .

En la **búsqueda binaria**, en cada comparación se puede eliminar la mitad de los elementos del vector. Así, el número de comparaciones que deben realizarse en el peor caso será el número de veces que el vector puede dividirse en mitades sucesivas. Antes de realizar la primera comparación, se busca en una lista de N elementos. Después de la primera comparación se buscará en una lista de $N/2$ elementos. Después de la segunda, la lista donde se busca tiene $N/4$ elementos, y así sucesivamente. Puede expresarse en la siguiente tabla:

Número de comparaciones	Elementos de la lista donde buscamos
0	N
1	$N/2$
2	$N/2^2$
3	$N/2^3$
...	...
C	$N/2^C$

Por tanto, si $N = 2^C$, después de hacer la comparación C la lista se habrá reducido a 1 elemento y se necesitarán $C + 1$ comparaciones en el peor caso, hasta comprobar que el elemento no se encuentra en la lista.

Por tanto, si $N = 2^C$, $C = \log_2 N$.

Cuanto más grande sea el array, mayor será la diferencia entre los dos algoritmos de búsqueda.

Ejemplo:

N	Número de comparaciones	
	Búsqueda secuencial	Búsqueda binaria
8	8	3
1024	1024	10
1048576	1048576	20

Como se dijo en la introducción, la complejidad o eficiencia de la búsqueda secuencial es $O(n)$ mientras que la de la búsqueda binaria es $O(\log n)$.

3.3. Algoritmos de inserción

Estos métodos consisten en añadir un nuevo elemento al array. Siempre debe verificarse que hay espacio disponible para el nuevo elemento. Si el array no está ordenado, habrá que especificar la posición donde se quiere insertar el nuevo elemento. Si está ordenado, el propio algoritmo determinará la posición que le corresponde al nuevo elemento.

Consideraremos dos casos diferentes:

- **Inserción en un array no ordenado.** Es necesario indicar la posición que debe ocupar el nuevo elemento. Para insertar un elemento en una determinada posición de un array, deberá abrirse un hueco en dicha posición, moviendo los elementos posteriores un lugar, empe-

zando por el último. Después se inserta el elemento en ese hueco. Siempre se deberá comprobar que el array no está lleno. Además se comprobará que la posición deseada será menor o igual que la última más 1, porque el elemento puede ir a continuación del último elemento, siempre que el array no esté lleno.

- **Inserción de un elemento en un array ordenado.** No hay que indicar ninguna posición, sino que ocupará la posición que le corresponda según su valor. Pueden contemplarse dos situaciones:
 - **Los elementos no pueden encontrarse repetidos en la lista.** En este caso, si el elemento se encuentra en la lista, se genera un mensaje o código de error indicando que el elemento no se puede insertar. En caso contrario, se averigua la posición que le corresponde y se inserta en ésta, de forma similar a la utilizada en el caso de arrays no ordenados.
 - **Los elementos pueden repetirse.** En este caso, nos da igual que el elemento a insertar se encuentre ya en el array. Simplemente se utiliza el método **binario** para determinar la posición donde se encuentra el elemento o donde se debería encontrar, y en esa posición se inserta el nuevo elemento.

En el Apartado 3.5.2. se ven varios ejercicios de inserción resueltos, donde se contemplan todas las posibilidades citadas.

3.4. Algoritmos de ordenación

Una de las tareas más frecuentes en el procesamiento de datos es la ordenación, que consiste en la reorganización de un conjunto dado de objetos en una secuencia especificada, con respecto a alguno de los campos de los elementos del conjunto. El objetivo fundamental de la ordenación es facilitar la búsqueda de los elementos del conjunto ordenado. Es una actividad fundamental que se realiza universalmente. Los elementos ordenados aparecen en las guías telefónicas, en las enciclopedias y diccionarios, en las fichas académicas, etc. Aparecen allí donde haya objetos que necesiten ser buscados y recuperados. Una vez que una secuencia de datos está ordenada se puede buscar de forma muy rápida cualquier objeto individual, como se vio en el Apartado 3.2.2.

En este apartado se tratarán cinco métodos de ordenación, aunque existen muchos más. No se pretende tratar de forma exhaustiva el tema sino mostrar algunos de los métodos más representativos. Los métodos de ordenación son un ejemplo excelente de cómo una tarea puede resolverse mediante algoritmos muy diferentes, cada uno de ellos con ventajas e inconvenientes. Cada algoritmo deberá valorarse frente al resto para decidir cuál será conveniente usar en cada caso concreto.

Los métodos se dividen en dos categorías, según la estructura de datos utilizada:

- **Métodos de ordenación interna:** consisten en la ordenación de arrays, que se almacenan en memoria interna, y son de acceso aleatorio: cada elemento es 'visible' y accesible individualmente.

- **Métodos de ordenación externa:** consisten en la ordenación de ficheros secuenciales. Los ficheros se colocan en almacenamientos externos, discos o cintas, y son más lentos. Su acceso será secuencial, es decir, para acceder a un elemento necesitaremos acceder a todos los anteriores.

En este capítulo solamente se tratarán los **métodos de ordenación interna** a los que se aplicará una restricción sobre el uso de la memoria: como es importante la utilización económica de la memoria disponible, las permutaciones de elementos deben realizarse sobre el mismo array, por tanto, no se utilizará un array origen y un array resultado.

Se pueden clasificar los métodos de ordenación interna en función de su eficiencia en términos del tiempo utilizado en la ordenación. Generalmente, nos fijaremos en el número de comparaciones en el de movimientos a realizar en función del número de elementos a ordenar. Pueden considerarse dos grupos de algoritmos de ordenación interna:

- **Métodos directos.** Son métodos sencillos, pero poco eficientes, su complejidad es $O(n^2)$, siendo n el número de elementos a ordenar. Estos algoritmos son cortos y fáciles de entender, y para arrays pequeños son suficientemente rápidos. Sin embargo, no deben utilizarse con arrays grandes puesto que el tiempo de ordenación crece demasiado. Dentro de este grupo, se estudiarán los siguientes algoritmos:
 - Ordenación por intercambio directo (*BubbleSort* o burbuja).
 - Ordenación por inserción directa.
 - Ordenación por selección directa.
- **Métodos avanzados.** Son algoritmos mucho más eficientes, eficiencia $O(n \log n)$, aunque son más sofisticados y difíciles de entender. Su uso es necesario cuando se trata de arrays grandes. Se estudiarán los siguientes algoritmos avanzados:
 - Método rápido (*QuickSort*).
 - Método del montículo (*HeapSort*).

Al final del capítulo se muestra una tabla con los valores del tiempo empleado en la ordenación de arrays de diferentes tamaños, utilizando todos los métodos estudiados.

3.4.1. Métodos directos

Método de Intercambio directo (burbuja, o *BubbleSort*)

La característica principal de este método es el intercambio de pares de elementos adyacentes. Consiste básicamente en hacer varias pasadas sobre el array comparando en cada una de ellas todos los elementos adyacentes de forma que si no están ordenados se intercambian. En el peor caso es necesario hacer $n - 1$ pasadas sobre el array para garantizar que queda ordenado.

Existen varias versiones. En la que se explica en el Ejercicio 3.9. del Apartado 3.5.3., en cada pasada, se arrastra el elemento de más valor hasta el extremo derecho.

Método de inserción directa

Este método consiste en insertar cada elemento, comenzando por el segundo y hasta el final, en el lugar que le corresponde en la secuencia ordenada que se va formando a su izquierda.

En la primera pasada se selecciona el segundo elemento y se inserta en la secuencia ordenada formada por el primer elemento. En la segunda pasada, se inserta el tercer elemento en la secuencia ordenada formada por los dos primeros, y así sucesivamente hasta llegar al final del array. Por tanto, el método consiste en $n - 1$ inserciones en una lista ordenada.

El algoritmo detallado para ordenar ascendentemente un array de n elementos por el método de inserción directa se explica en el Ejercicio 3.12, del Apartado 3.5.3.

Método de selección directa

Este método se basa en el siguiente principio: se selecciona de la lista completa el elemento con menor valor y se intercambia con el primero, con lo que el primer elemento queda definitivamente ordenado. A continuación, se busca el elemento de menor valor de la sublista comprendida entre el segundo y el último, y se intercambia con el segundo elemento, y así sucesivamente hasta que sólo quede un elemento, que quedará definitivamente ordenado en el último lugar.

El algoritmo para ordenar ascendentemente un array de n elementos por el método de selección directa se explica con detalle en el Ejercicio 3.15, del Apartado 3.4.1.

3.4.2. Métodos avanzados

Los métodos directos son sencillos pero son poco eficientes, su complejidad es $O(n^2)$. Son útiles para ordenar arrays pequeños pero cuando se trata de arrays con gran número de elementos resultan excesivamente lentos a efectos prácticos. Por este motivo son necesarios otros algoritmos más sofisticados y muy eficientes. En este apartado se estudiarán el método de ordenación rápida, también llamado *QuickSort*, y el método del montículo, también llamado *HeapSort*. Ambos algoritmos mejoran drásticamente la eficiencia de los métodos directos, son de orden $(n \log n)$, lo que quiere decir que prácticamente son lineales para valores grandes de n .

Método de ordenación rápida (*QuickSort*)

La mejora de este algoritmo es tan efectiva, que es el mejor método para ordenar arrays conocido hasta el momento. Su creador fue Hoare (1960) y lo llamó método rápido (*QuickSort*) [Hoare62]. Este algoritmo es fácil de implementar, y es el más eficiente de todos los métodos conocidos. En la mayoría de las situaciones su eficiencia es $O(n \log n)$, aunque en determinados casos, puede convertirse en un algoritmo lento, siendo su eficiencia $O(n^2)$. Estos casos son muy improbables y se analizarán en este apartado.

Descripción del algoritmo

Consiste en particionar el array de forma que los valores de la parte izquierda sean todos menores o iguales que un determinado elemento del array llamado **pívote** y los de la parte derecha, mayores o iguales que el pivote. Una vez hecho esto, se particionan del mismo modo las partes izquierda y derecha, haciendo lo mismo con las partes resultantes, y así sucesivamente hasta que todas las partes consten de un elemento, momento en el cual se habrá conseguido ordenar el array. En principio, podemos elegir como pivote cualquier elemento de la secuencia que vamos a particionar, aunque como se verá, la eficiencia del algoritmo dependerá de la acertada elección del pivote. Este algoritmo hace uso de la recursividad, concepto que se tratará con detalle en el capítulo siguiente. Existen versiones no recursivas del mismo, pero son mucho más complicadas y apenas se utilizan; por este motivo se ha decidido incluir la versión recursiva del algoritmo, ya que es, con diferencia, la versión más utilizada de todos los algoritmos de ordenación.

En el Ejercicio 3.17, del Apartado 3.5.3, se explica este algoritmo con todo detalle, y se incluye la codificación en Java.

Hay que resaltar la importancia de la elección del pivote. El caso óptimo es aquél donde el pivote coincide con la mediana de los valores, de forma que la partición nos dé como resultado dos mitades aproximadamente iguales. Esto debe ocurrir en todas las llamadas recursivas. El caso peor sería aquél donde el pivote es el mayor o el menor elemento de los valores a particionar, siendo, por tanto, el resultado muy asimétrico: una partición de un solo elemento, y la otra con el resto de los elementos. Si esto ocurre en todas las llamadas recursivas, lo cual es altamente improbable, este algoritmo se vuelve muy lento con un orden de complejidad $O(n^2)$.

Podría pensarse en calcular la mediana para poder elegir el pivote óptimo, pero esta sobrecarga computacional haría que el algoritmo perdiera eficiencia.

Aunque en la codificación presentada en el Ejercicio 3.17 se elige como pivote el elemento que ocupa la posición central del array o subarray a particionar, puede elegirse cualquiera: el primero, el último o cualquiera elegido al azar.

Método del montículo (*HeapSort*)

Este método es una variante del método de ordenación por selección, donde la búsqueda del elemento mínimo de un array se realiza mediante técnicas basadas en la construcción de un montículo o *heap*.

Veamos primero qué es un montículo y qué relación tiene con un array: cualquier array puede representarse en forma de árbol. Por ejemplo, un array cuyos elementos ocupan las posiciones 0 a N podemos representarlo como un árbol binario, donde se cumple que para un nodo que ocupa la posición i del array, su hijo izquierdo ocupa la posición $2*i+1$, y su hijo derecho la posición $2*i+2$. En la Figura 3.1, vemos la representación como árbol binario de un array de 8 elementos.

Pues bien, un montículo es un árbol binario que cumple ciertas condiciones. Consideraremos dos tipos de montículos:

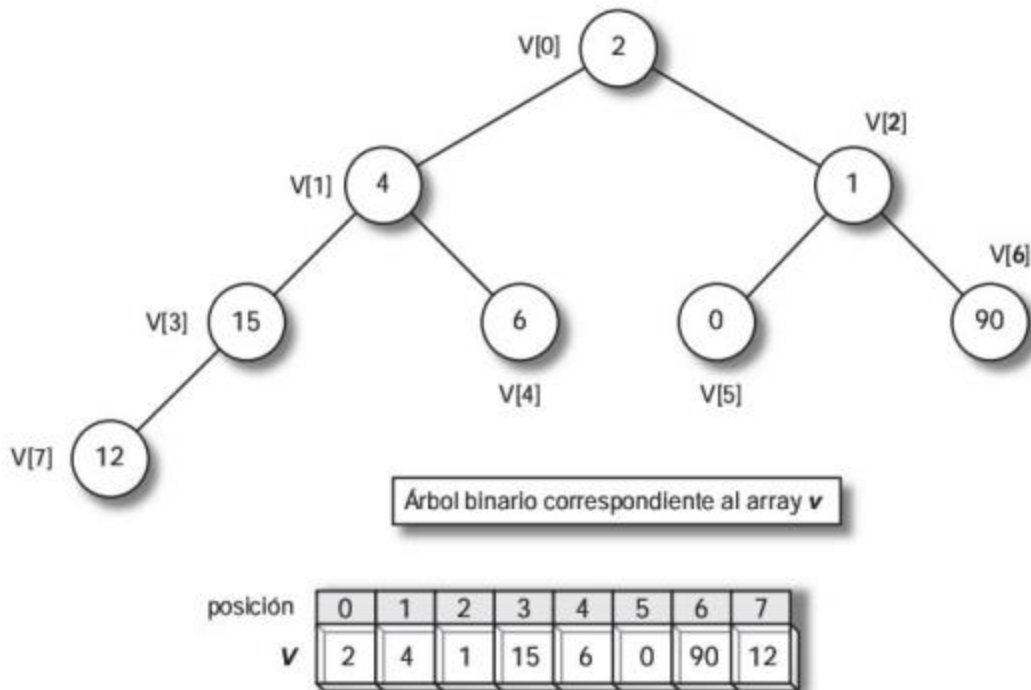


Figura 3.1. Representación de un array como árbol binario.

- Montículo ascendente (también llamado *min heap*)

En ellos se cumple que cada nodo es menor o igual que sus hijos.

Es decir, constan de una secuencia de valores $h_0, h_1, h_2, h_3, \dots, h_n$ tales que

$$h_i \leq h_{2i+1}$$

$$h_i \leq h_{2i+2}$$

para todos los nodos i que tengan dos hijos, y $h_i \leq h_{2i+1}$ para el nodo que sólo tenga un hijo.

Puede observarse que la raíz del árbol contiene el menor elemento del *heap* y que, además, cualquier camino de la raíz a una hoja, es una lista ordenada ascendente.

- Montículo descendente (o *max heap*)

Se cumple que cada nodo es mayor o igual que sus hijos. Constan de una secuencia de valores $h_0, h_1, h_2, h_3, \dots, h_n$ tales que

$$h_i \geq h_{2i+1}$$

$$h_i \geq h_{2i+2}$$

para todos los nodos i que tengan dos hijos, y $h_i \geq h_{2i+1}$ para el nodo que sólo tenga un hijo.

También puede observarse que la raíz del árbol contiene el mayor elemento del *heap* y que, además, cualquier camino de la raíz a una hoja, es una lista ordenada descendente.

Veamos ahora en qué consiste el **algoritmo de ordenación del montículo o *HeapSort***

En el método de selección directa, se selecciona el menor elemento del array $v[0]..v[n]$ y se intercambia con $v[0]$, luego el menor elemento del array $v[1]..v[n]$ y se intercambia con $v[1]$, y así sucesivamente hasta llegar al elemento $v[n-1]$. De esta forma se obtendrá un array ordenado ascendentemente. Evidentemente, para encontrar el mínimo entre n elementos se necesitan $n - 1$ comparaciones. El método de búsqueda del mínimo elemento puede mejorarse significativamente utilizando un proceso de búsqueda en un árbol. El método *HeapSort* fue ideado por J. Williams (1964) [Williams64] y consiste en la siguiente idea: se construye un montículo ascendente, o *min heap*, $v[0]..v[n]$, por tanto, la raíz $v[0]$ es el elemento menor, y se intercambia con $v[n]$. Luego se construye un montículo $v[0]..v[n-1]$, y el menor elemento ($v[0]$) se intercambia con $v[n-1]$, y así sucesivamente hasta que sólo quede un elemento en el montículo. De esta forma, se obtendrá un array ordenado descendentemente. Si se quiere realizar una ordenación ascendente, deberá utilizarse un montículo descendente o *max heap*. Hay que observar que inicialmente debe construirse completamente el montículo a partir del array desordenado, pero cuando empiezan los intercambios, el único elemento que cambia es la raíz, no siendo, por tanto, necesaria la construcción completa del *heap*, sino sólo una reconstrucción parcial mucho más rápida.

Se considerarán, por tanto, dos fases:

Fase 1: Construcción inicial del montículo

Se construye el montículo inicial a partir del array original.

Fase 2: Ordenación

- Se intercambia la raíz con el último elemento del montículo. A partir de ahora este último elemento queda definitivamente ordenado y ya no pertenece al montículo.
- Se restaura el montículo haciendo que se "hunda" el primer elemento hasta la posición que le corresponda. Ahora la raíz vuelve a ser el menor del montículo. Volver al punto a). Esto se repite hasta que en el montículo sólo quede un elemento.

Ahora se verán estas dos fases con detalle:

Fase 1: Construcción inicial del montículo

Para **construir el montículo inicial**, puede observarse qué parte de la lista inicial ya es un montículo: desde la mitad + 1 hasta el final, los elementos son nodos hoja, por lo que no incumplen la propiedad del montículo. Por tanto, partiendo de la posición $(N-1) / 2$ hasta la posición 0 se van incorporando elementos al montículo (de uno en uno) aplicando un procedimiento llamado **criba** que hace que la raíz del árbol, al cual se aplica, se "hunda" hasta encontrar la posición que le corresponda, en caso de que incumpla la propiedad del montículo. Si un elemento no es menor o igual que sus hijos, se "hunde" por la rama que corresponde al menor de sus hijos. Este procedimiento, que se desarrollará con detalle en el Ejercicio 3.20., recibe tres parámetros: el vector que contiene los elementos a ordenar, la posición del elemento a "hundir" y la última posición ocupada por el montículo.

El pseudocódigo correspondiente a la fase 1 sería:

```
desde i = (ultimo - 1) / 2 hasta 0
    criba (vector,i,ultimo);
```

Fase 2: Ordenación

Una vez construido el montículo inicial, la raíz (posición 0) contiene al menor elemento del array. Por tanto, se intercambiará con el último elemento del montículo quedando esta última posición ya definitivamente ordenada, por lo que se extrae del montículo; es decir, el montículo ahora sólo llega hasta la penúltima posición. Al haber cambiado el elemento de la raíz, puede ocurrir que ahora incumpla la propiedad del *min heap*, por lo que habrá que regenerarlo hundiendo este elemento hasta que encuentre su posición.

Una vez regenerado el montículo volverá a encontrarse en la raíz su valor mínimo, por lo que se intercambiará con el elemento que ocupa la última posición del montículo que será la penúltima del array. Al quedar el elemento definitivamente ordenado, el montículo se acorta y deberá regenerarse de nuevo, siguiendo con el mismo proceso hasta que el montículo tenga sólo un elemento.

El pseudocódigo correspondiente sería:

```
desde i = ultimo hasta 1 hacer
    // intercambiar la raíz con el elemento de la posición i
    aux = vector[0]; vector[0] = vector[i]; vector[i] = aux;
    // reconstruir el montículo entre 0 e i-1.
    // El último elemento se ha quitado del árbol
    criba(vector,0,i-1)
fin-desde
```

Si se quiere ordenar ascendentemente, debe utilizarse un *max Heap*, de manera que se forme un montículo donde el nodo raíz sea el mayor elemento en lugar del menor. Así, al intercambiar la raíz con el último elemento del montículo se va formando una secuencia ordenada ascendentemente. Para hacer esto, sólo hay que hacer ligeras modificaciones en el procedimiento *criba*, como se puede observar en el Ejercicio 3.22.

3.4.3. Medición experimental de la eficiencia

A modo de resumen, se muestra una tabla con los tiempos que han tardado en ordenarse arrays de enteros de diferentes tamaños utilizando todos los métodos de ordenación vistos en este capítulo. Los arrays se han generado aleatoriamente. En las columnas figura el número de elementos del array a ordenar y en las filas los diferentes métodos. Los tiempos se dan en milisegundos. Obsérvese la diferencia entre el tiempo requerido por el algoritmo de la burbuja y el requerido por *QuickSort* para ordenar el array de 40.000 enteros : unos 12 minutos frente a algo menos de 1 segundo. También es interesante observar que para ordenar un array de 10.000 elementos, la

burbuja es 552 veces más lento que *QuickSort*, mientras que para ordenar el array de 40.000 elementos, la burbuja es 1.939 veces más lento.

	10.000	15.000	20.000	25.000	30.000	35.000	40.000
Burbuja	41.924	116.538	170.098	266.772	391.321	534.290	697.952
Inserción	13.311	31.966	55.043	86.794	127.112	197.123	272.311
Selección	16.962	39.192	69.083	108.254	157.849	226.166	324.750
QuickSort	76	123	163	212	267	449	360
HeapSort	176	290	389	493	692	720	844

3.5. Ejercicios resueltos

3.5.1. Búsqueda

Ejercicio 3.1. Búsqueda secuencial

Enunciado Desarrollar el algoritmo de búsqueda secuencial explicado en la teoría, y codificarlo en Java.

Solución En primer lugar, veamos el algoritmo expresado en pseudocódigo:

En la **cabecera** se indican los siguientes parámetros:

- **lista**: el array que contiene la lista de elementos.
- **elemento**: el elemento que queremos buscar.
- **primero, último**: primera y última posición entre las que efectuar la búsqueda.

Además, se necesitarán las siguientes **variables locales**:

- **i**: índice entero para recorrer el array.
- **encontrado**: variable booleana para determinar cuándo se ha encontrado el elemento buscado.

```
procedimiento busquedaSecuencial (lista, elemento, primero, ultimo)
```

 Inicializamos las variables anteriores:

```
    i = primero
    encontrado = falso
```

 Entramos en un bucle y permaneceremos en él **mientras** se cumplan las siguientes condiciones:

- que no encontremos el elemento,
- que no alcancemos el final del array.

Si alguna de ellas no se cumple, finalizará el bucle.

Dentro del bucle, se comparará el elemento de la posición *i* con el que estamos buscando: si coinciden, se le da el valor cierto a la variable booleana *encontrado* y si no coinciden, se incrementa el índice *i* para buscar en la siguiente posición.

```
mientras (elemento no encontrado y i<=ultimo)
    si elemento = lista [i] entonces
        encontrado = cierto
    si-no
        incrementar i
fin-mientras
```

Una vez fuera del bucle, hay que comprobar cuál de las dos condiciones ha provocado la salida: porque se ha encontrado el elemento, en cuyo caso se devuelve la posición del índice *i*, o porque se ha llegado al final del array sin encontrar el elemento, devolviendo -1 como código de error.

```
si encontrado entonces
    devolver i
si-no
    devolver -1
```

A continuación, se muestra el algoritmo completo codificado en Java:

```
public static int secuencial (int [] vector, int elem, int p, int u) {
    // recibe una matriz no ordenada, un elemento a buscar
    // y los limites del rango donde buscar(p y u)
    // devuelve la posicion del elemento o -1 si no esta
    int i;
    boolean encontrado;
    i=p;
    encontrado=false;
    while (i<=u&& !encontrado)
        if (elem == vector[i])
            encontrado = true;
        else
            i++;
    if (encontrado)
        return i;
    else return -1;
    // no se ha encontrado
}
```

Ejercicio 3.2. Búsqueda binaria

Enunciado Desarrollar el algoritmo de búsqueda binaria explicado en teoría, y codificarlo en Java.

Solución El algoritmo detallado expresado en pseudocódigo se explica a continuación:

En la **cabecera** se indican los siguientes parámetros:

- **lista**: el array que contiene la lista de elementos.
- **elemento**: el elemento que queremos buscar.
- **p, u**: primera y última posición entre las que efectuar la búsqueda.
- **posicion**: posición donde se encuentra el elemento buscado, o posición donde debería estar en el caso de que no se encontrara.
- **Encontrado**: variable booleana para determinar si el elemento está en la lista.

Además, se necesitarán las siguientes **variables locales**:

- **mayor, menor**: índices enteros para delimitar las posiciones superior e inferior de la lista o sublista donde se busca el elemento.
- **central**: posición que se encuentra en el centro de la lista delimitada por mayor y menor.

Procedimiento `búsquedaBinaria (lista, elemento, primero, ultimo, posicion, encontrado)`

Se inicializan las variables **menor** y **mayor** a los valores de **primero** y **último**, y **encontrado** a falso:

```
menor = primero;
mayor = ultimo;
encontrado = falso;
```

Entramos en un bucle y seguiremos iterando mientras se cumplan las condiciones:

- no encontramos el elemento,
- no se crucen los índices mayor y menor.

Mientras (no encontrado) y (mayor >= menor) hacer

Dentro del bucle se hará lo siguiente:

- Se determinará la posición del elemento central:

```
(central = (mayor + menor ) div 2)
```

- Se comprueba si el elemento buscado es el que se encuentra en la posición central, en cuyo caso asignaremos a la variable **encontrado** el valor cierto.

```
si lista [central] = elemento entonces
    encontrado = cierto
```

- Si el elemento buscado no coincide con el central, comprobaremos si es superior o si por el contrario es inferior: si es superior, habrá que buscar en la sublista superior, para lo cual moveremos el índice **menor** colocándolo en la posición **central + 1**: ahora la sublista superior en la que buscaremos estará delimitada por **menor** y **mayor**. Si es inferior, el índice que hay que mover es **mayor**, colocándolo en la posición **central - 1**.

```
si no
    si elemento > lista [central] entonces
        menor = central + 1
    si no
        mayor = central - 1
    fin si
fin si
```

Una vez que se salga del bucle tendremos que determinar cuál ha sido el motivo: si hemos encontrado el **elemento** o bien se han cruzado los índices mayor y menor. En el primer caso, la posición buscada es la del índice central, y en el segundo caso, el elemento no está en el array, pero la posición buscada es la que le correspondería si quisiéramos insertarlo, siendo la indicada por el índice menor.

```
si encontrado entonces
    posicion = central
si no
    posicion = menor
fin si
```

Codificación en Java

El código completo en Java sería:

```
public static void binaria (int [] vector,int elem, int p,int u,
Entero posicion, Booleano encontrado){
    // recibe una matriz ordenada, un elemento a buscar
    // y los limites del rango donde buscar(p y u)
    // recibe dos referencias con un booleano para devolver si el
    // elemento se encuentra en la matriz, y con un entero para devolver
    // la posicion donde esta o la que le corresponde si no esta

    int i;
    int menor = p, mayor = u, medio=0;
    encontrado.v = false;
    while (!encontrado.v && mayor >= menor){
```

```

    medio = (mayor + menor) / 2;
    if (elem == vector[medio])
        encontrado.v=true;
    else if (elem > vector[medio])
        // buscar en la mitad superior
        menor = medio + 1;
    else
        // buscar en la mitad inferior
        mayor = medio - 1;
}
if (encontrado.v)
    posicion.v = medio;
else
    posicion.v = menor;
}

```

Ejercicio 3.3. Traza del algoritmo de búsqueda binaria

Enunciado Realizar un seguimiento detallado del algoritmo de búsqueda binaria cuando se desea buscar el número 20 en el array 'lista' formado por los siguientes números enteros: 3, 4, 10, 15, 16, 20, 80, 92.

Solución En la Figura 3.2. se puede ver el array, los valores de los índices (primero, ultimo, menor, central y mayor), y el elemento a buscar (20).

Se inicializan los índices menor y mayor a primero y último, y se calcula el valor del central según se ve en la Figura 3.2. (izquierda) y a continuación, se compara elemento con el valor de la posición central, Figura 3.2. (derecha), y comprobamos que es mayor.

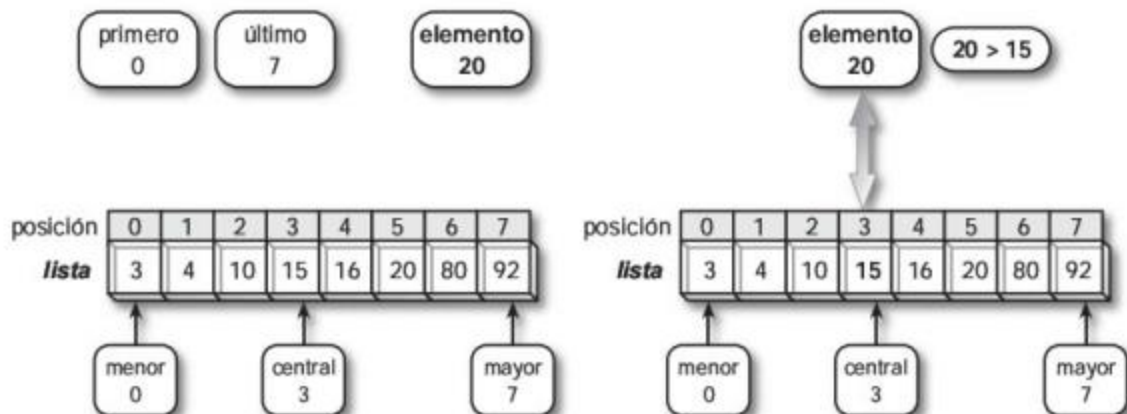


Figura 3.2. Traza del algoritmo de búsqueda binaria cuando el elemento buscado existe (a).

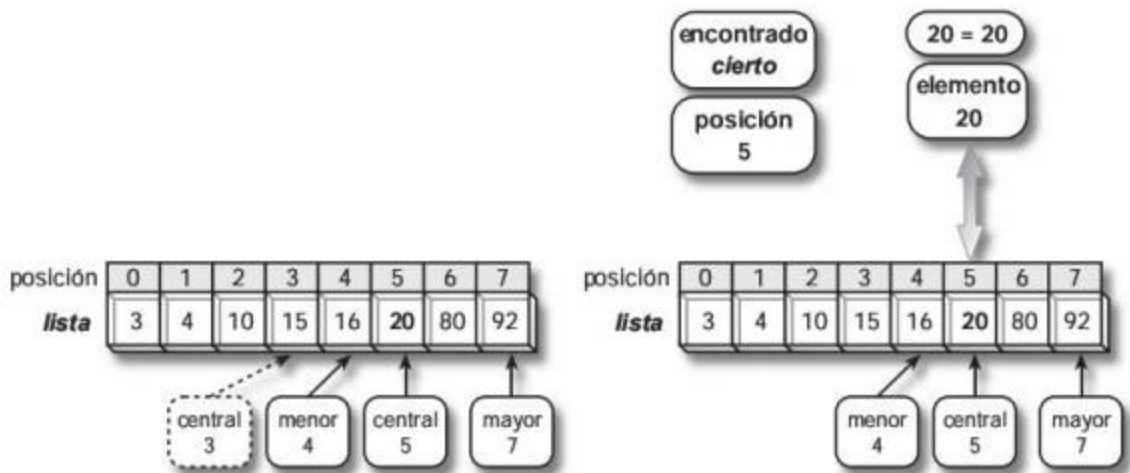


Figura 3.3. Traza del algoritmo de búsqueda binaria cuando el elemento buscado existe (b).

Por tanto, deberemos buscar en la mitad superior, es decir, la delimitada por las posiciones 4 y 7. Calculamos el nuevo valor de menor que será 4, como podemos ver en la Figura 3.3. (izquierda). Ahora, central será $(7 + 4) / 2 = 5$. Al volver a comparar el elemento buscado con el situado en la posición central, comprobamos que coinciden, con lo que la búsqueda finaliza dando como resultado los siguientes valores:

POSICIÓN = 5

ENCONTRADO = CIERTO

En el siguiente ejercicio se detallará el funcionamiento del algoritmo cuando el elemento a buscar no se encuentra en el array.

Ejercicio 3.4. Traza del algoritmo de búsqueda binaria

Enunciado Realizar un seguimiento detallado del algoritmo de búsqueda binaria cuando se desea buscar el número 5 en el array 'lista' formado por los siguientes números enteros: 3, 4, 10, 15, 16, 20, 80, 92.

Solución Ahora haremos el seguimiento del mismo ejemplo donde el elemento a buscar es el 5, que no se encuentra en el array. En la Figura 3.4. (izquierda) vemos la situación de los índices. En Figura 3.4. (derecha) comparamos el elemento con el de la posición central comprobando que el que buscamos es menor.

En Figura 3.5. (izquierda) vemos cómo se coloca el índice mayor en la posición 2. Por tanto, el nuevo índice central será el 1. En Figura 3.5. (derecha) se hace la comparación del central con el elemento buscado comprobando que éste es mayor, debiéndose mover ahora el índice menor, como se puede ver en Figura 3.6. (izquierda).

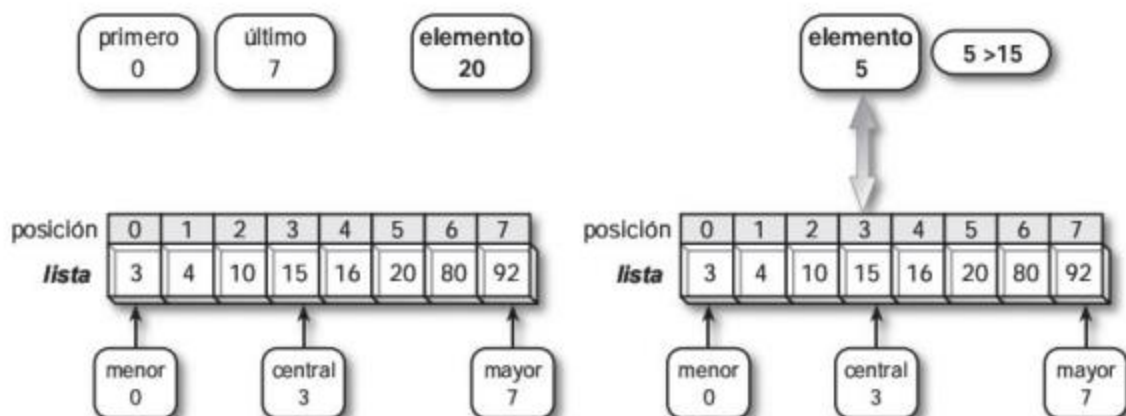


Figura 3.4. Trazo del algoritmo de búsqueda binaria cuando el elemento buscado no existe (a).

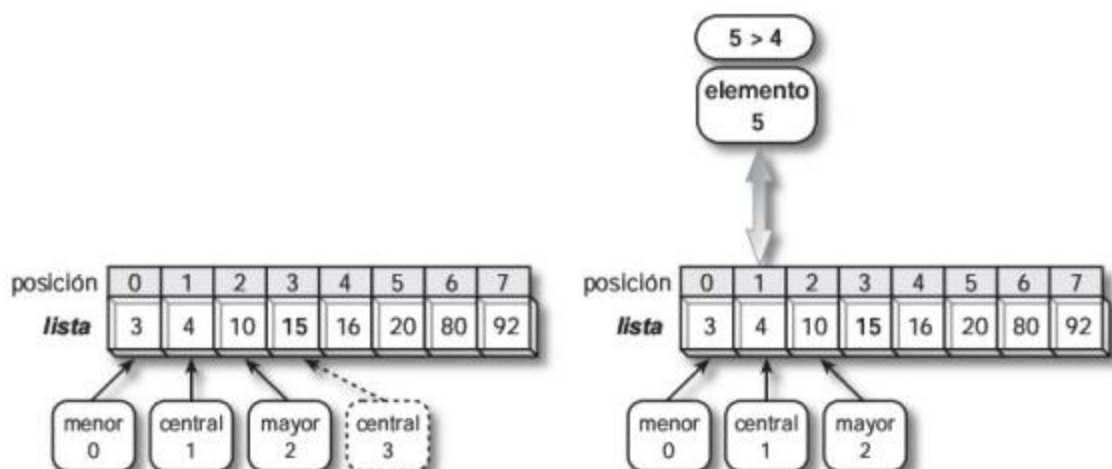


Figura 3.5. Trazo del algoritmo de búsqueda binaria cuando el elemento buscado no existe (b).

Se ve que ahora mayor y menor coinciden en la posición 2, y por tanto, central también tendrá el valor 2. En Figura 3.6, (derecha) se ve la comparación, resultando que elemento es menor que el valor apuntado por central.

Entonces, debe moverse el índice mayor a la posición 1 (Figura 3.7.-izquierda). En este momento mayor y menor se han cruzado, y se sale del bucle. Se comprueba que encontrado es falso, y por tanto, la posición buscada es la apuntada por menor, es decir, la posición que le correspondería al valor 5 (que no se encuentra en la lista) es la 2.

Resultado:

POSICIÓN = 2
ENCONTRADO = FALSO

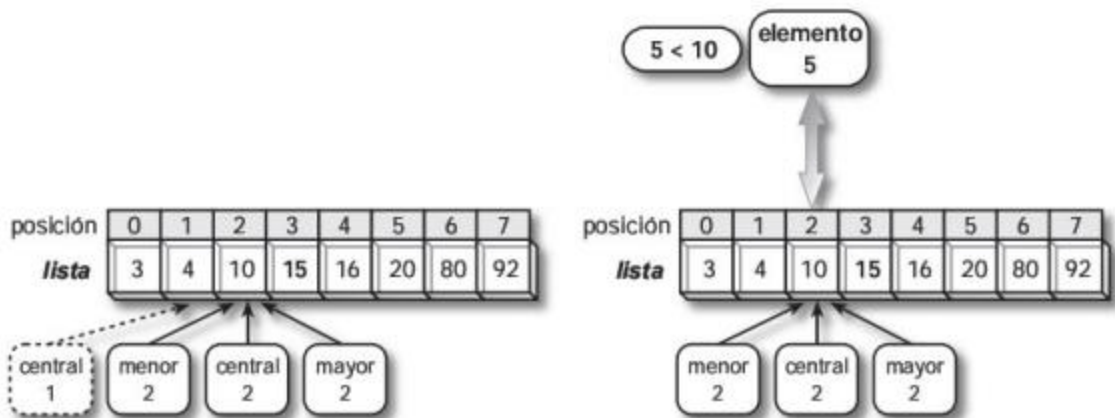


Figura 3.6. Trazas del algoritmo de búsqueda binaria cuando el elemento buscado no existe (c).

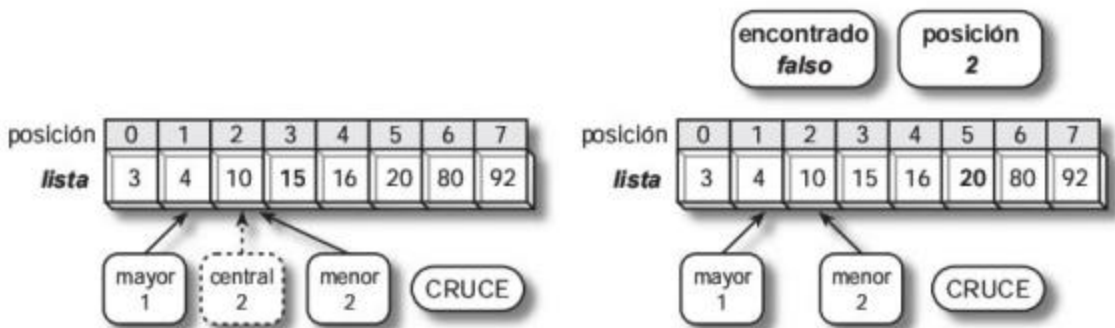


Figura 3.7. Trazas del algoritmo de búsqueda binaria cuando el elemento buscado no existe (d).

3.5.2. Inserción

Ejercicio 3.5. Inserción en arrays no ordenados

Enunciado Desarrollar en pseudocódigo el algoritmo de inserción en arrays no ordenados. Codificar en Java este algoritmo.

Solución Como se comentó en el Apartado 3.3., para insertar un elemento en una determinada posición de un array no ordenado, deberemos abrir un hueco en dicha posición, moviendo los elementos posteriores un lugar, empezando por el último. A continuación, se insertará el elemento en ese hueco, comprobando que el array no está lleno. Veamos el algoritmo expresado en pseudocódigo:

En la **cabecera** se indican los siguientes parámetros:

- **lista** el array que contiene la lista de elementos,
- **elemento** el elemento que queremos insertar,

- **k** la posición donde deberá insertarse el elemento. Se supone que es una posición válida,
- **último** última posición ocupada del array, que deberá incrementarse una vez realizada la inserción.

Además, se utilizará la siguiente **variable local**:

- **j**: índice entero para recorrer el array.

```
procedimiento insertarNoOrdenada (lista, elemento, k, ultimo)
```

En primer lugar se comprobará que la lista no está llena. Si lo está se devolverá un código de error o se sacará un mensaje de error.

```
si listallena entonces
    escribir ('error')
```

Si no lo está, movemos todos los elementos un lugar, comenzando por el último, hasta la posición k, es decir, cada elemento lo copiamos en la siguiente posición.

```
desde j = ultimo hasta k (incremento -1)
    lista [j+1] = lista [j]
fin-desde
```

Ahora se copia el elemento en la posición k del array, y se actualiza el índice último, porque ahora hay un elemento más en el array.

```
lista [k] = elemento
incrementar ultimo
```

Codificación del algoritmo en Java

En este método, se recibe como parámetro por referencia ultimo, que indica la posición del último elemento del array.

```
public static int insertarNoOrdenada (int [] vector,int elemento,int
pos, Entero ultimo){
    // recibe una matriz, un elemento a insertar, y la posicion de
    // insercion. Tambien recibe un objeto num de tipo Entero con la última
    // posición ocupada de la matriz. Se supone que el valor de
    // pos es correcto
    if (ultimo.v == vector.length - 1)
        return -1; // codigo de error, matriz llena
    else {
        for (int i = ultimo.v; i >= pos; i-- )
            vector[i+1] = vector[i];
        vector[pos] = elemento;
        ultimo.v ++;
        return 0;
    }
} // fin de insertarNoOrdenada
```


Ejercicio 3.6. Traza del algoritmo de inserción

Enunciado Se dispone de un array de 8 posiciones, donde están ocupadas las 6 primeras con los siguientes valores: 4, 3, 10, 15, 6, 20. Realizar un seguimiento detallado del algoritmo de inserción visto en el problema anterior, cuando se desea insertar el valor 95 en la posición 2.

Solución En la Figura 3.8. se puede ver el array, el elemento a insertar (95), el valor de la posición de inserción $k = 2$, y el valor de la variable `ultimo` que indica la posición ocupada por el último valor.

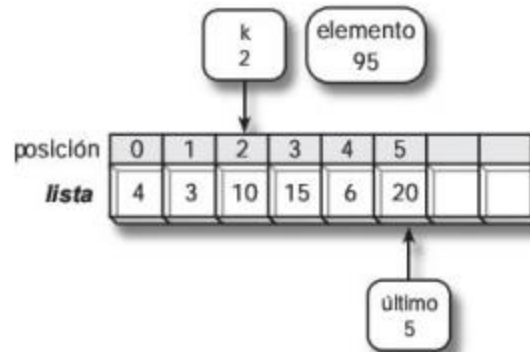


Figura 3.8. Inserción de un elemento en un array no ordenado. Situación inicial.

En primer lugar comprobamos que el array no está lleno, y movemos los elementos de posiciones ≥ 2 a la posición siguiente, pero teniendo cuidado de empezar por el último, es decir el que está en la posición 5 pasa a la 6, el que está en la 4 pasa a la 5 y así sucesivamente, hasta que movemos el que está en la posición 2 para pasar a la 3 (véase Figura 3.9.).

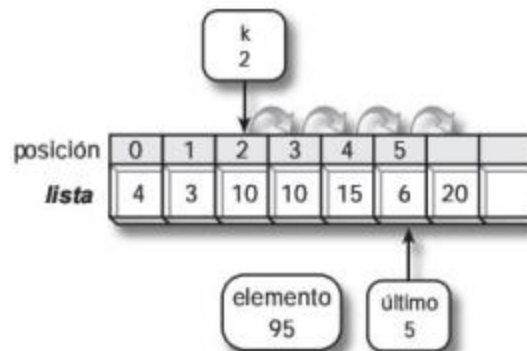


Figura 3.9. Inserción de un elemento en un array no ordenado. Movimiento de elementos.

Y por último, insertamos el elemento cuyo valor es 95 en la posición 2 del array, y modificamos el índice `ultimo`, de forma que ahora vale 6 (véase Figura 3.10.).

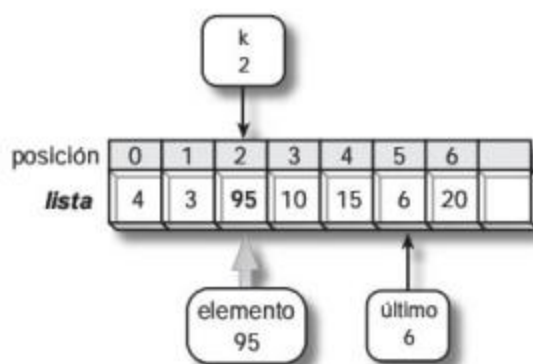


Figura 3.10. Inserción de un elemento en un array no ordenado. Situación final.

Ejercicio 3.7. Inserción en arrays ordenados

Enunciado Desarrollar en pseudocódigo el algoritmo de inserción en arrays ordenados, teniendo en cuenta que no se permite que haya elementos repetidos. Codificar en Java dicho algoritmo.

Solución Como se explicó en el Apartado 3.3., no hay que especificar la posición de inserción porque, al estar el array ordenado, el elemento deberá ocupar una posición determinada que el propio método determinará utilizando el algoritmo de búsqueda binaria explicado en el Apartado 3.2.2.

Teniendo en cuenta que los elementos no pueden repetirse, si el elemento ya se encuentra en la lista, se genera un mensaje o código de error indicando que el elemento no se puede insertar. En caso contrario, se averigua la posición que le corresponde y se inserta en ésta.

Veamos el algoritmo expresado en el pseudocódigo correspondiente al algoritmo:

En la **cabecera** se indican los siguientes parámetros:

- **lista**: el array que contiene la lista de elementos,
- **elemento**: el elemento que queremos insertar,
- **último**: última posición ocupada del array, que deberá incrementarse una vez realizada la inserción.

Como en el caso de arrays no ordenados, se utilizará la siguiente **variable local**:

- **j**: índice entero para recorrer el array.

```
procedimiento insertarOrdenada (lista, elemento, ultimo)
```

En primer lugar se comprobará que la lista no está llena. Si lo está se devolverá un código de error o se sacará un mensaje de error.

```
si listallena entonces
    escribir ('error')
```

A continuación, se utilizará el procedimiento de búsqueda binaria para comprobar que el elemento no se encuentra en el array y averiguar la posición que le corresponde.

```
binaria (lista,elemento,l,ultimo,posicion, encontrado)
si encontrado entonces
    escribir ('ERROR: El elemento ya se encuentra en el array')
```

Ahora, igual que en el Apartado 2.3.1. movemos todos los elementos un lugar, comenzando por el último, hasta la posición indicada por la variable **posicion**, es decir, cada elemento lo copiamos en la siguiente posición.

```
si no
    desde j = ultimo hasta posicion (incremento -1)
        lista [j+1] = lista [j]
    fin-desde
fin si
```

Ahora se copia el elemento en la posición indicada en la variable **posicion**, y se actualiza el índice último, porque ahora hay un elemento más en el array.

```
lista [posicion] = elemento
incrementar ultimo
```

Codificación en Java

```
public static int insertarOrdenada1 (int [] vector,int elemento,
Entero ultimo){
    // recibe una matriz ordenada y un elemento a insertar
    // que debe insertarse en la posición que le corresponda
    // tambien recibe un objeto entero con el numero de
    // elementos utiles en la matriz
    // los elementos no pueden estar repetidos
    Entero posicion = new Entero();
    Booleano encontrado = new Booleano();
    if (ultimo.v == vector.length - 1)
        return -1; // codigo de error, matriz llena
    else{
        // buscar la posicion
        Busqueda.binaria(vector,elemento,0,ultimo,posicion,encontrado);
        if (encontrado)
            return -2; // codigo de error, elemento repetido
        else{
            for (int i = ultimo; i >= posicion.v; i- )
                vector[i+1] = vector[i];
            vector[posicion.v] = elemento;
            ultimo.v ++;
```

```

        return 0; //codigo de terminacion OK
    } // fin del else interno
} // fin del else externo
} // fin de insertar

```

Ejercicio 3.8. Inserción en arrays ordenados

Enunciado Desarrollar en pseudocódigo el algoritmo de inserción en arrays ordenados, teniendo en cuenta que se permite que haya elementos repetidos. Codificar en Java dicho algoritmo.

Solución En este caso, nos da igual que el elemento a insertar se encuentre ya en el array. Simplemente se utiliza el método **binario** para determinar la posición donde se encuentra el elemento o donde se debería encontrar, y en esa posición se inserta el nuevo elemento.

La cabecera es idéntica:

```

procedimiento insertarOrdenada (lista, elemento, ultimo)

```

Como siempre que se realiza una inserción, se comprueba que la lista no está llena. Si lo está se devolverá un código de error o se presentará el correspondiente mensaje.

```

si listallena entonces
    escribir ('error')

```

A continuación, se utilizará el procedimiento de búsqueda binaria para averiguar la posición que le corresponde al elemento, independientemente de que se encuentre o no en el array.

```

    binaria (lista,elemento,l,ultimo,posicion, encontrado)

```

Ahora, igual que en el caso 1 movemos todos los elementos un lugar, comenzando por el último, hasta la posición indicada por la variable **posición**, es decir, cada elemento lo copiamos en la siguiente posición.

```

desde j = ultimo hasta posicion (incremento -1)
    lista [j+1] = lista [j]
fin-desde

```

Ahora se copia el elemento en la posición indicada en la variable **posición**, y se actualiza el índice último, porque ahora hay un elemento más en el array.

```

lista [posicion] = elemento
incrementar ultimo

```

Codificación del algoritmo en Java

```

public static int insertarOrdenada2 (int [] vector,int elemento,
Entero ultimo){
    // recibe una matriz ordenada y un elemento a insertar

```



```
// que debe insertarse en la posición que le corresponda
// tambien recibe un objeto entero con el numero de
// elementos utiles en la lista
// se permiten elementos repetidos en el array
Entero posicion = new Entero();
Booleano encontrado = new Booleano();
if (ultimo.v == vector.length - 1)
    return -1; // codigo de error, matriz llena
else{
    // buscar la posicion
    Busqueda.binaria(vector, elemento, 0, ultimo, posicion, encontrado);
    // nos da igual que el elemento este repetido
    for (int i = ultimo; i >= posicion.v; i--)
        vector[i+1] = vector[i];
    vector[posicion.v] = elemento;
    ultimo.v++;
    return 0;
} // fin del else
} // fin de insertar
```

3.5.3. Ordenación. Métodos directos

Ejercicio 3.9. Algoritmo de la burbuja

Enunciado Desarrollar en pseudocódigo el algoritmo de ordenación *BubbleSort* o algoritmo de la burbuja, de forma que ordene ascendentemente n elementos y que en cada pasada 'arrastre' el valor mayor hacia las últimas posiciones del array. Codificar en Java dicho algoritmo.

Solución Se necesitan los siguientes parámetros:

- **lista**: el array que contiene la lista de elementos a ordenar.
- **n**: el número de elementos que queremos ordenar.

Por tanto, la cabecera es:

Procedimiento Burbuja (lista, n)

Se utilizarán las siguientes **variables**:

- **i**: nos indicará el número de pasada,
- **j**: índice para recorrer el array,
- **aux**: variable auxiliar para realizar el intercambio de los elementos del array.

Se utilizará un bucle controlado por **i** que realizará $n - 1$ iteraciones, tantas como el número de pasadas necesarias en el peor caso.

desde $i \leftarrow 1$ hasta $N-1$ hacer

Para cada iteración del bucle, se comparará cada elemento, comenzando por el primero, con el siguiente, de forma que si no están ordenados se intercambian utilizando una variable auxiliar.

```

desde  $j \leftarrow 0$  hasta  $N-1-i$  hacer
    //hasta el penúltimo elemento
    Si  $L[j] > L[j+1]$  entonces
        // intercambio
        aux  $\leftarrow L[j]$ 
         $L[j] \leftarrow L[j+1]$ 
         $L[j+1] \leftarrow aux$ 
    fin-si
fin-desde

```

Se puede observar que en la primera pasada, el elemento de más valor queda colocado en la última posición, en la segunda pasada, el segundo elemento de más valor queda colocado en la penúltima posición y así sucesivamente, de forma que se va generando una sublista ordenada a la derecha del array. Por tanto, en la primera pasada, j tiene que llegar hasta la penúltima posición para poder comparar el elemento j con el $j+1$. En la segunda pasada sólo hay que llegar hasta la antepenúltima posición, porque en la última ya está colocado definitivamente el mayor elemento. De esta forma, cada vez es menor el recorrido de cada pasada.

Codificación en Java

```

public static void burbuja(int [] vector, int num){
    // ordena por el algoritmo de la burbuja una matriz de num elementos
    // utiles.
    int aux;
    for (int i=1; i<num; i++){
        for (int j=0; j<num - i; j++){
            if (vector[j] > vector [j+1]){
                aux = vector [j];
                vector [j] = vector [j+1];
                vector [j+1] = aux;
            }
        }
    }
}

```

Ejercicio 3.10. Algoritmo de la burbuja mejorado

Enunciado Codificar en Java un método que ordene ascendentemente un array de enteros por el método de la burbuja, de forma que si el array queda ordenado en una pasada intermedia, no sea necesario continuar hasta la pasada $n - 1$.

Solución Modificaremos el método anterior, añadiendo una variable booleana "intercambio" que controlará si se ha realizado algún intercambio de elementos en la pasada. Si en alguna pasada no se realiza intercambio, es porque el array ya está ordenado y, por tanto, no es necesario seguir.

```

public static void burbujaMejorada(int [] vector, int num){
    int aux;
    int i=1; // num de pasada
    boolean intercambio = true;
    while (intercambio){
        intercambio = false;
        for (int j=0;j<num - i; j++){
            if (vector[j] > vector [j+1]){
                aux = vector [j];
                vector [j] = vector [j+1];
                vector [j+1] = aux;
                intercambio = true;
            }
        }
        i++;
    }
}

```

Ejercicio 3.11. Traza del algoritmo de la burbuja

Enunciado Realizar un seguimiento detallado del algoritmo de la burbuja visto en el Ejercicio 3.9. para ordenar ascendentemente el siguiente array: 2, 4, 1, 15, 6, 0, 90, 12.

Solución En la Figura 3.11. se pueden observar paso a paso las comparaciones que se realizan en la primera pasada, comenzando por la comparación entre el primer elemento y el segundo, los cuales,

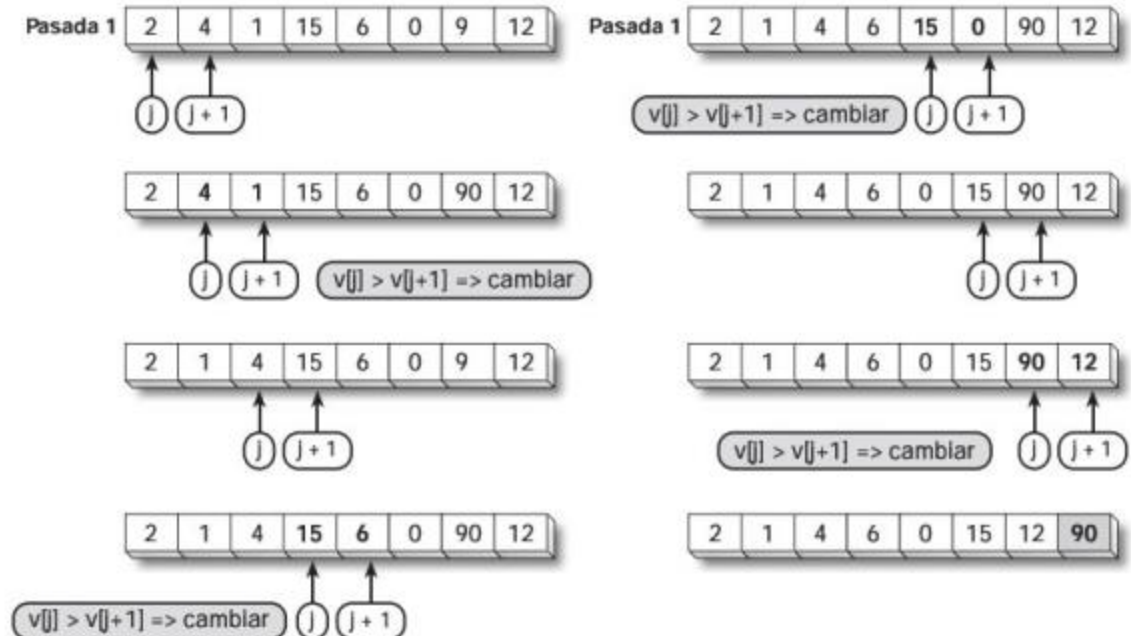


Figura 3.11. Traza del algoritmo de la burbuja. Pasada 1.

al estar ya ordenados ascendentemente, no se intercambian. Después se compara el segundo con el tercero y en este caso se intercambian por no estar ordenados, y así sucesivamente hasta que se llega a comparar el penúltimo con el último realizándose el intercambio, con lo que la pasada completa requiere 7 comparaciones ($n - 1$). En la parte inferior de la Figura 3.11. (derecha) se puede ver el resultado de la primera pasada habiéndose marcado el último elemento (90) que es el mayor del array, y ha quedado definitivamente ordenado.

En la Figura 3.12. se muestra el proceso correspondiente a la segunda pasada. Es similar al de la primera pasada pero ahora no hace falta llegar a comparar con el último elemento porque, como dijimos antes, ya está definitivamente ordenado. Ahora el recorrido requiere sólo 6 comparaciones. En la parte inferior de la Figura 3.12. (derecha) se ve el resultado de la segunda pasada.

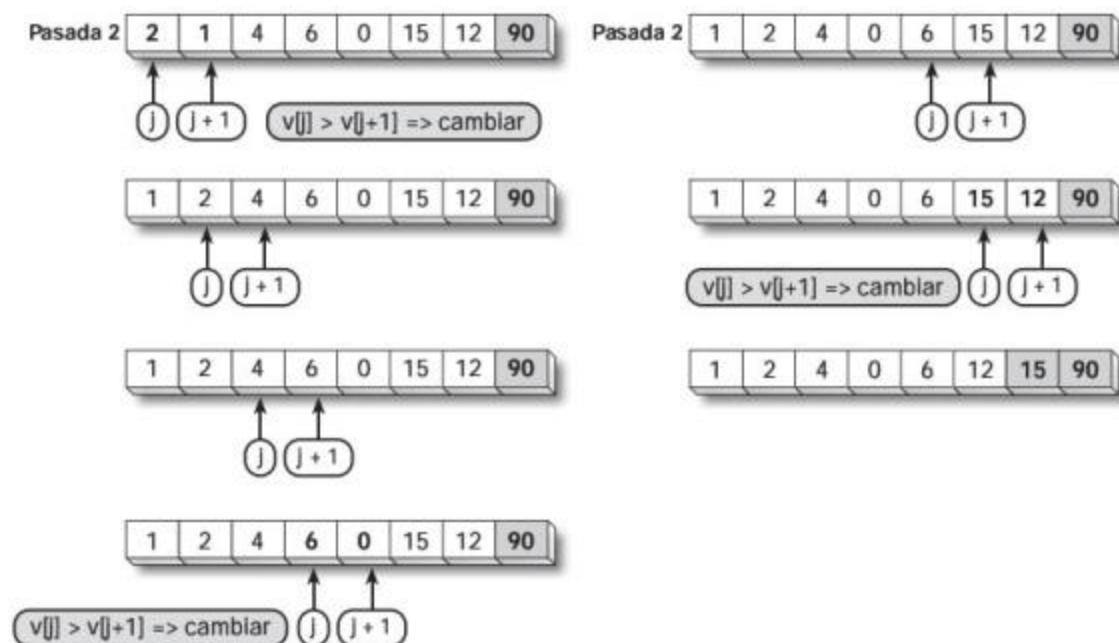


Figura 3.12. Trazo del algoritmo de la burbuja. Pasada 2.

En la Figura 3.13. se observa el proceso correspondiente a la tercera pasada. Ahora sólo es necesario realizar 5 comparaciones.

En la Figura 3.14. (izquierda) se ve el resultado de la pasada 4, donde hay que realizar 4 comparaciones, y quedan 4 elementos definitivamente ordenados a la derecha del array. En la Figura 3.14. (derecha) se muestran las 3 comparaciones necesarias correspondientes a la pasada 5.

Por último, en la Figura 3.15. se muestra el proceso de la pasada 6 con 2 comparaciones y de la pasada 7 con solamente 1 comparación y donde el array queda definitiva y completamente ordenado.

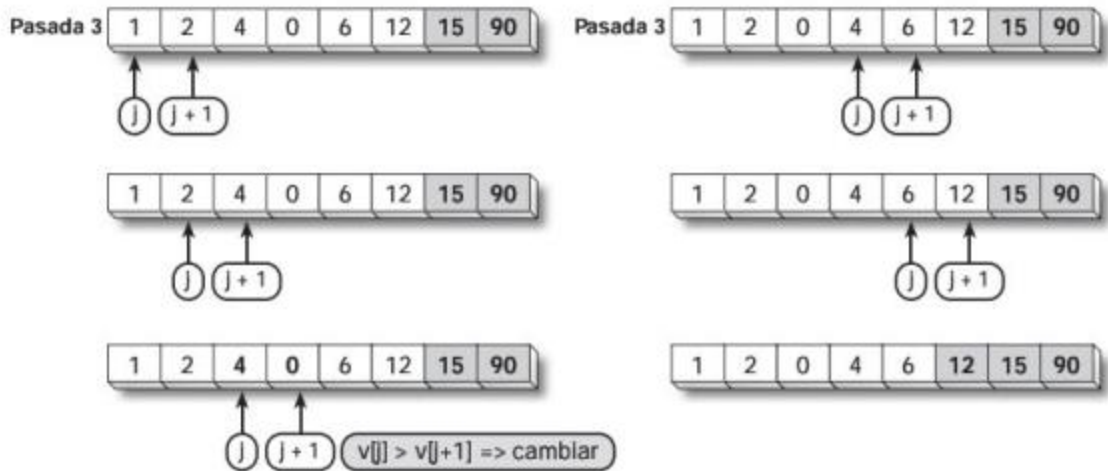


Figura 3.13. Traza del algoritmo de la burbuja. Pasada 3.

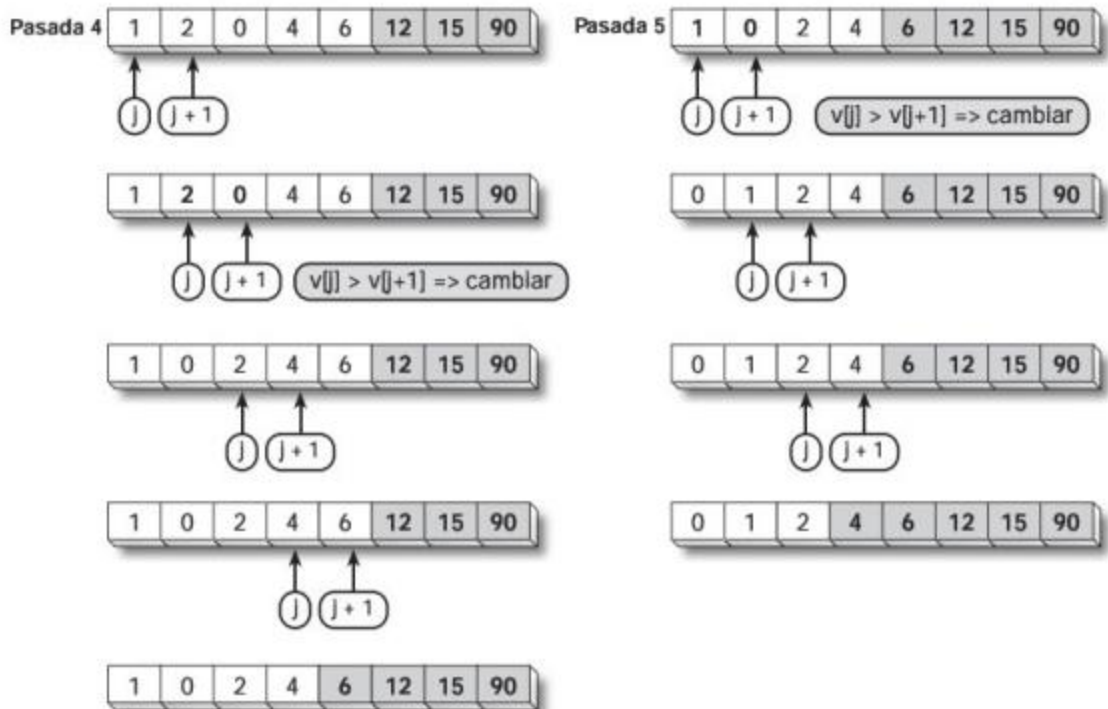


Figura 3.14. Traza del algoritmo de la burbuja. Pasadas 4 y 5.

En el ejemplo anterior se puede observar que las pasadas 6 y 7 no son realmente necesarias porque el array ya está ordenado y sin embargo, se siguen haciendo comparaciones que consumen tiempo de computación y no producen ningún resultado.

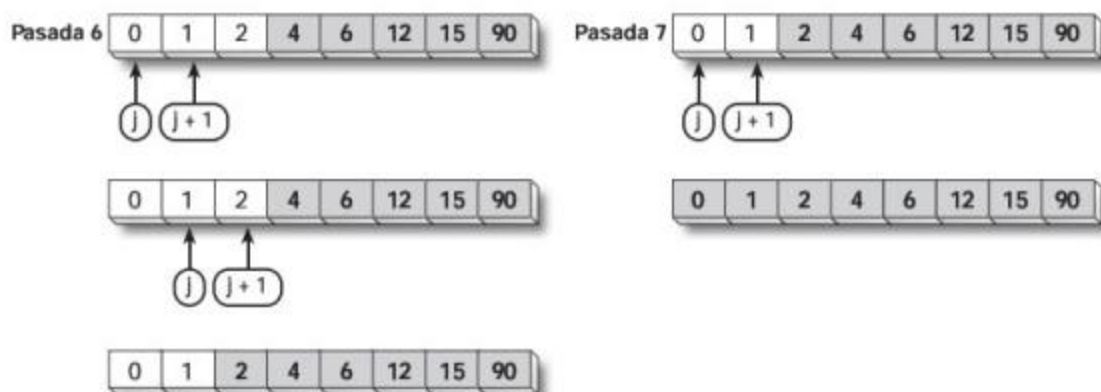


Figura 3.15. Traza del algoritmo de la burbuja. Pasadas 6 y 7.

Ejercicio 3.12. Algoritmo de inserción directa

Enunciado Desarrollar en pseudocódigo el algoritmo de ordenación por Inserción Directa, de forma que ordene ascendente n elementos de un array. Codificar en Java dicho algoritmo.

Solución En la **cabecera** se indican los parámetros **lista** y **n**, igual que en el ejercicio anterior.

Procedimiento InserciónDirecta (lista, n)

Se utilizarán las siguientes **variables**:

- **i**: nos indicará la posición del elemento a insertar.
- **j**: índice para recorrer la sublista ordenada donde se debe insertar el elemento a insertar.
- **aux**: variable auxiliar para guardar el elemento lista[i].

Comenzando por el segundo elemento del array, se recorren todos hasta el último. Tengamos en cuenta que en Java, la primera posición es la 0 y, por tanto, la última será la $n-1$:

desde $i \leftarrow 1$ hasta $n-1$ **hacer**

Para cada iteración, guardaremos el elemento lista[i] en una variable auxiliar **aux**, e inicializaremos el índice **j** a la posición anterior a **i**:

```
aux ← lista[i]
j ← i - 1
```

Se recorre con **j** la sublista ordenada formada por los elementos de las posiciones anteriores a **i**, buscando el lugar donde debe insertarse **aux**. Como vimos en el Apartado 2.3, ("algoritmos de inserción") una vez se encuentre su posición habrá que mover todos los elementos posteriores para dejar libre la posición comenzando por el elemento de la posición más alta. Como la búsqueda de la posición comienza en la posición más alta, lo más razonable es mover el elemento a

la vez que se hace la búsqueda. Las condiciones que deben cumplirse para permanecer en el bucle que realiza la búsqueda son las siguientes:

- El índice j no se sale del array.

```
j >= 0
```

- El elemento a insertar (aux) es menor que el elemento del array que estamos inspeccionando.

```
aux < lista[j]
```

El pseudocódigo correspondiente a esta búsqueda y desplazamiento sería:

```
mientras j >= 0 y aux < L[j] hacer
    lista [j + 1] ← lista [j]
    decrementar j
fin mientras
```

Una vez que la posición se ha localizado se sale del bucle y se inserta el elemento en la posición $j + 1$ (j se ha decrementado) en la última línea del bucle.

```
lista [j + 1] ← aux
```

Codificación en Java

```
public static void insercionDirecta(int [] lista, int num){
    // ordena por el algoritmo de insercion directa una matriz de
    // num elementos utiles.

    int i,j;
    int aux;
    for (i = 1; i<num; i++){
        aux = lista[i];
        j= i - 1;
        while (j >= 0 && aux < lista[j]) {
            lista [j + 1] = lista [j];
            j--;
        }
        lista [j + 1] = aux;
    } // for
}
```

Ejercicio 3.13. Traza del algoritmo de inserción directa

Enunciado Realizar un seguimiento detallado del algoritmo de Inserción Directa visto en el Ejercicio 3.12. para ordenar ascendentemente el siguiente array: 2, 4, 1, 15, 6, 0, 90, 12.

Solución En la Figura 3.16. se pueden ver las 4 primeras pasadas, donde se insertan los elementos 4, 1, 15 y 6. Se observa cómo se forma una sublista ordenada en las posiciones anteriores a las indicadas por el índice i .

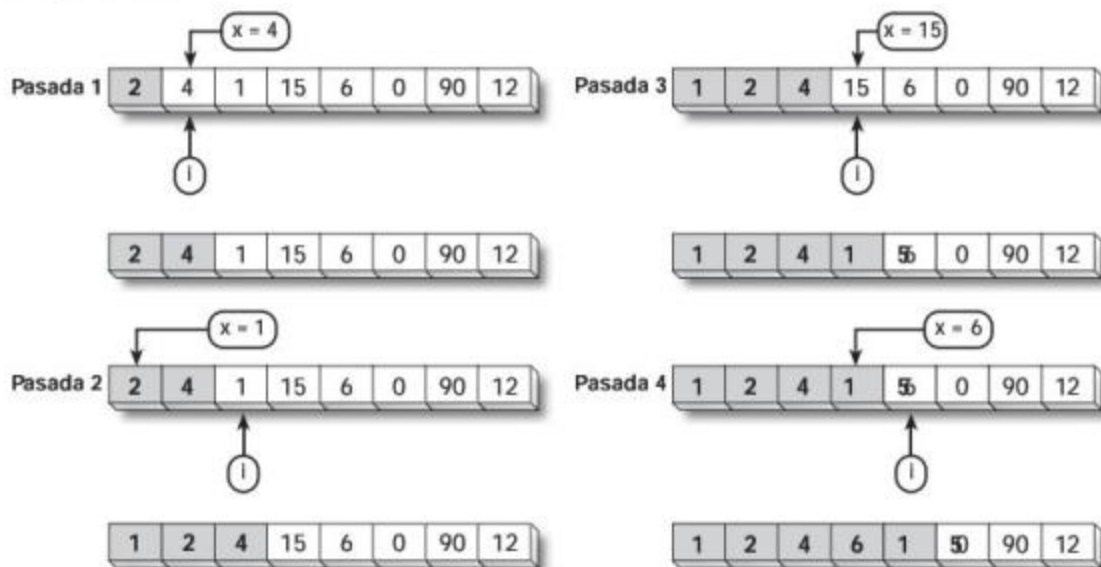


Figura 3.16. Trazo del algoritmo de inserción directa. Pasadas 1, 2, 3 y 4.

En la Figura 3.17. se ve el resultado de las pasadas 5, 6 y 7 y el resultado final.

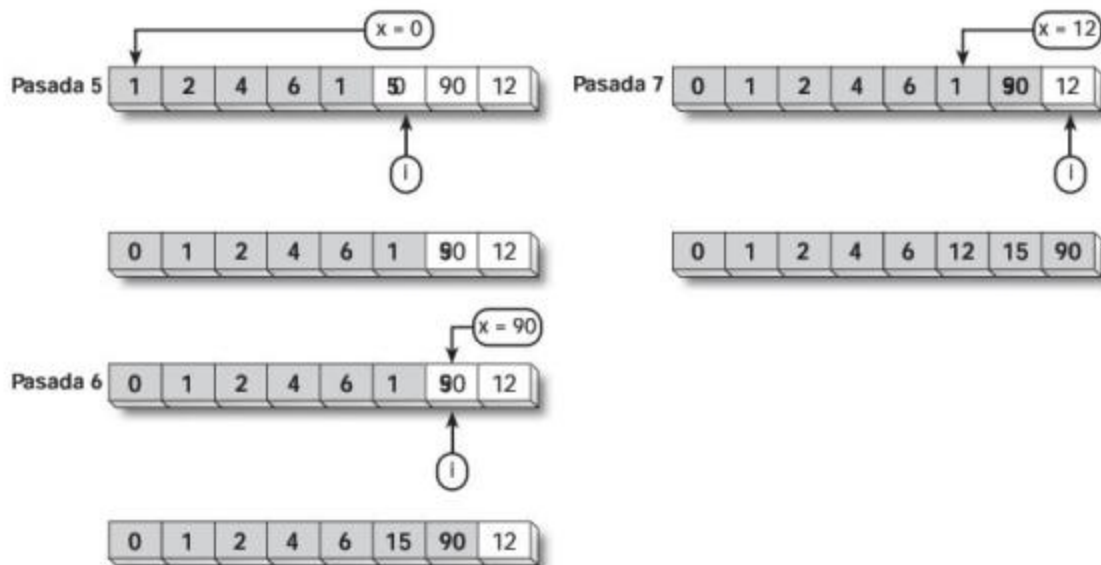


Figura 3.17. Trazo del algoritmo de inserción directa. Pasadas 5, 6 y 7.

Ejercicio 3.14. Inserción directa con búsqueda binaria

Enunciado Dado que el algoritmo de inserción directa se basa en la inserción de un elemento en una lista ordenada, parece razonable utilizar búsqueda binaria en lugar de búsqueda secuencial, para insertar dicho elemento. Razónese si realmente merece la pena la utilización de la búsqueda binaria en este algoritmo de ordenación.

Solución A primera vista parece que la ganancia es muy grande puesto que la búsqueda binaria es de complejidad $O(\log n)$, pero ocurre lo siguiente:

- La búsqueda del lugar de inserción es muy rápida pero de todos modos hay que mover los elementos, con lo que la complejidad del algoritmo completo sigue siendo del orden de n^2 .
- En el caso peor, cuando los datos están ordenados inversamente, se gana algo de tiempo aunque no es una mejora tan importante como parece a simple vista.
- En el caso mejor, se pierde tiempo porque:
 - En inserción directa, el lugar se encuentra inmediatamente, sólo hay que hacer una comparación y movimiento por cada valor de i .
 - En inserción binaria, el número de comparaciones es independiente de la ordenación inicial de los elementos, por tanto, hay que hacer el mismo número de comparaciones que en el peor caso, aunque sólo habría que mover un elemento.

Ejercicio 3.15. Algoritmo de selección directa

Enunciado Desarrollar en pseudocódigo el algoritmo de ordenación por selección directa, de forma que ordene ascendentemente n elementos de un array. Codificar en Java dicho algoritmo.

Solución Como en los métodos anteriores los parámetros serán `lista` y `n`.

Procedimiento `SeleccionDirecta (lista, n)`

Se utilizarán las siguientes **variables**:

- **i**: nos indicará la posición donde vamos a colocar el menor elemento de la sublista correspondiente.
- **j**: índice para recorrer la sublista donde buscaremos el menor elemento.
- **menor**: variable auxiliar para guardar el menor elemento de la subelemento `lista[i]`.
- **posición**: variable auxiliar para guardar la posición del elemento menor.

Se recorre el array desde el primer elemento hasta el penúltimo ($n-2$) utilizando el índice i . En cada iteración se buscará el menor de la lista comprendida entre la posición i y la última y para ello se inicializa la variable `menor` al primer elemento de la sublista y la variable `posición` al valor de i :

```
desde i ← 0 hasta n-2 do  
    menor ← lista[i]; posicion i
```

En cada iteración del bucle externo, se utiliza otro bucle, controlado por el índice j , para buscar el elemento menor:

```
desde  $j \leftarrow i+1$  hasta  $n$  do
    si  $L[j] < \text{menor}$  entonces
        menor  $\leftarrow L[j]$ ; posición  $j$ 
    fin si
fin desde
```

Una vez fuera del bucle interno, se intercambian los valores de la posición i y del elemento menor:

```
L[posición] L[i]
L[i] menor
```

Codificación en Java

```
public static void seleccionDirecta(int [] vector, int n){
    // ordena por el algoritmo de insercion directa una matriz de
    // n elementos utiles.

    int menor;
    int i,j,pos;
    for (i = 0; i < n-1; i++){
        menor = vector[i];
        pos = i;
        // buscamos el menor de la sublista derecha
        for (j = i+1; j < n ; j++){
            if (vector[j] < menor){
                menor = vector[j];
                pos = j;
            }
        }
        vector[pos] = vector[i];
        vector[i] = menor;
    }
}
```

Ejercicio 3.16. Traza del algoritmo de selección directa

Enunciado Realizar un seguimiento detallado del algoritmo de selección directa visto en el Ejercicio 3.15., para ordenar ascendentemente el siguiente array: 2, 4, 1, 15, 6, 0, 90, 12.

Solución En la Figura 3.18. vemos las 4 primeras pasadas. En la pasada 1, se localiza el valor mínimo del array, el 0, en la posición 5 y se intercambia con el valor de la posición 0. En la pasada 2, se localiza el mínimo en la posición 2 y se intercambia con el valor de la posición 1.

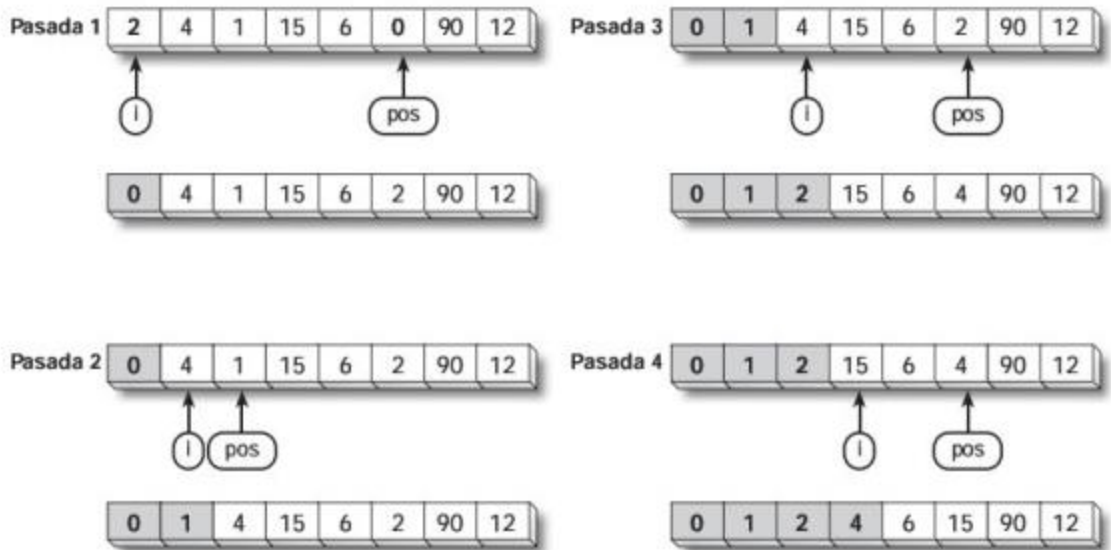


Figura 3.18. Traza del algoritmo de selección directa. Pasadas 1, 2, 3 y 4.

Como se puede ver en las Figuras 3.18. y 3.19., se realizan 7 pasadas hasta conseguir el array totalmente ordenado. También se puede observar cómo se va formando una sublista ordenada a la izquierda a medida que se van efectuando las correspondientes pasadas.

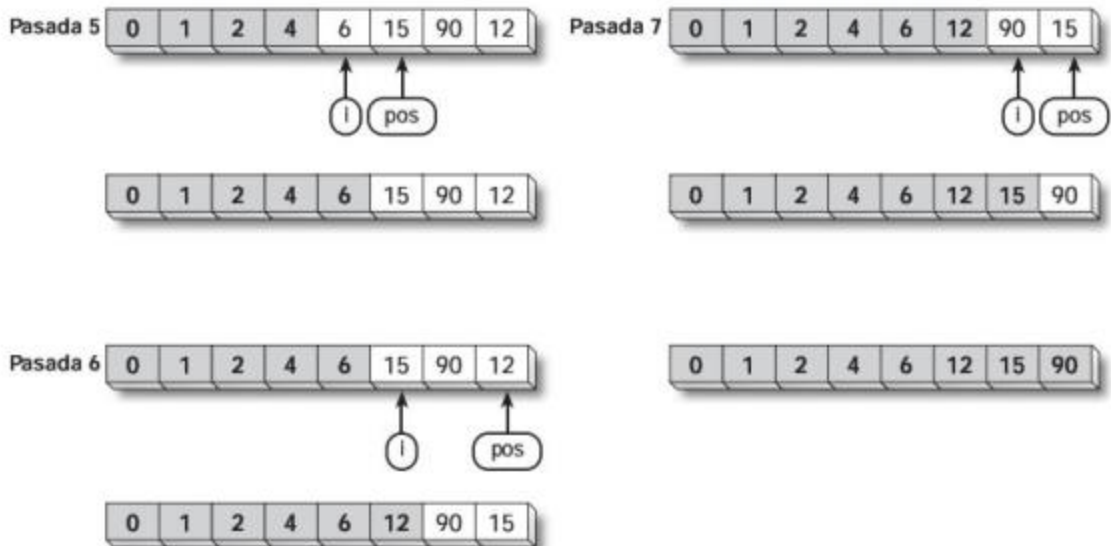


Figura 3.19. Traza del algoritmo de selección directa. Pasadas 5, 6 y 7.

3.5.4. Ordenación. Métodos avanzados

Ejercicio 3.17. Algoritmo de ordenación rápida

Enunciado Desarrollar en pseudocódigo el algoritmo de ordenación rápida o *QuickSort*, de forma que ordene ascendentemente n elementos de un array. Codificar en Java dicho algoritmo.

Solución En el pseudocódigo que veremos a continuación utilizaremos una cabecera donde se indican los siguientes parámetros:

- **lista**: el array que vamos a ordenar,
- **izq, der**: las posiciones izquierda y derecha del array respectivamente.

procedimiento quickSort (lista, izq, der)

Se utilizarán las siguientes variables:

- **i, j**: índices para recorrer el array.
- **pivote**: almacenará el pivote seleccionado.

En primer lugar inicializamos un índice i a la posición izquierda del array, y un índice j a la posición derecha.

```
i=izq
j=der
```

Tomamos arbitrariamente un elemento del array que será el pivote. Una opción puede ser elegir el elemento que ocupa la posición central.

```
pivote lista [(izq + der)/2]
```

Vamos recorriendo el array con el índice i de izquierda a derecha hasta encontrar un elemento lista [i] mayor o igual que el pivote.

```
Mientras lista [i] < x hacer
  incrementar i
```

Recorremos el array de derecha a izquierda con el índice j hasta encontrar un elemento lista [j] que sea menor o igual que el pivote.

```
Mientras lista [j] > x hacer
  decrementar j
```

Ahora el índice i apunta a un elemento que debe estar en la partición derecha y el índice j apunta a un elemento que debe estar en la partición izquierda. Por tanto, deben intercambiarse

siempre que los índices i y j no se hayan cruzado. Una vez hecho el intercambio, los dos índices avanzan una posición.

```

si  $i \leq j$  entonces
    Intercambiar lista  $[i]$  con lista  $[j]$ 
    incrementar  $i$ 
    decrementar  $j$ 
fin-si

```

Continuamos con el proceso de recorrido e intercambio, hasta que los índices i , j se crucen obteniendo como resultado un array partido en dos donde en su parte izquierda todos sus elementos son menores o iguales que x , y en su parte derecha todos sus elementos son mayores o iguales que x .

Una vez que el array está particionado las partes resultantes son las siguientes:

```

izquierda: lista desde izq hasta  $j$ .
derecha: lista desde  $i$  hasta der.

```

Este proceso de partición debe ahora realizarse **recursivamente** con las partes resultantes si tienen más de un elemento:

```

Si  $izq < j$  entonces quickSort(lista, izq, j)
Si  $der > i$  entonces quickSort (lista, i, der)

```

Codificación en Java

```

public static void quickSort(int [] vector, int izq, int der){
    // ordena por el algoritmo de quick sort una matriz de enteros
    // comprendidos entre las posiciones izq y der
    int i,j; // indices
    int pivote;
    int aux; // var auxiliar para intercambio
    pivote = vector[(izq+der) / 2];
    i = izq; j = der;
    do{
        while (vector [i] < pivote) i++;
        while (vector [j] > pivote) j--;
        if (i<=j) {
            aux = vector [i];
            vector[i] = vector[j];
            vector[j] = aux;
            i++;
            j--;
        } // end del if
    } while (i<=j);
}

```

```

        if (izq < j) quickSort(vector, izq, j);
        if (i < der) quickSort(vector, i, der);
    } // fin de quickSort

```

Ejercicio 3.18. Algoritmo de ordenación rápida

Enunciado Modificar el método *QuickSort* del ejercicio anterior de forma que ordene el array descendente.

Solución Basta con cambiar el sentido de los operadores relacionales que se encuentran en la condición de los bucles *while* internos:

```

while (vector [i] > pivote) i++;
while (vector [j] < pivote) j--;

```

de este modo, el índice *i*, que recorre la parte izquierda del array, se detiene cuando encuentra un valor que debe estar en la parte derecha, es decir cuando es menor que el pivote. Del mismo modo, el índice *j* se detiene cuando encuentra un valor que debe estar en la parte izquierda, es decir cuando es mayor que el pivote. Como antes, si se encuentra un valor igual al pivote el índice se debe detener.

Ejercicio 3.19. Traza del algoritmo de ordenación rápida

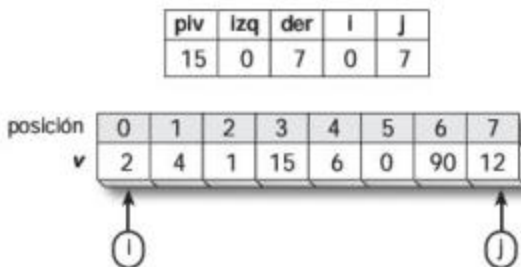
Enunciado Realizar un seguimiento detallado del algoritmo *QuickSort* visto en el Ejercicio 3.17., para ordenar ascendentemente el siguiente array: 2, 4, 1, 15, 6, 0, 90, 12.

Solución En la Figura 3.20. (izquierda) se hace la primera llamada al método *QuickSort*, enviando como parámetros el vector *v*, y los extremos del array 0 y 7. El pivote seleccionado es el valor 15. En la Figura 3.20. (derecha) se ve como se recorre el array con el índice *i* de izquierda a derecha hasta que se encuentra un valor mayor o igual que el pivote (el 15). Con *j* se recorre de derecha a izquierda hasta encontrar un elemento menor o igual que el pivote. En este caso no avanza ninguna posición y se queda apuntando al elemento 12. Se intercambian los valores y avanzan los índices.

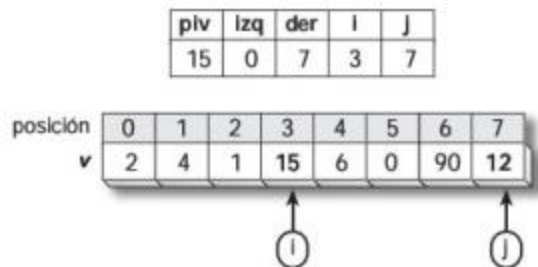
Los índices siguen avanzando hasta que se cruzan, como se puede ver en la Figura 3.21. (izquierda), donde se muestra el array particionado: parte izquierda: (0,5) y parte derecha: (6,7). Se hace la segunda llamada: *QuickSort(v, 0, 5)*, que corresponde a la primera llamada recursiva (Figura 3.21. derecha). Se pueden observar los nuevos valores de *izq* (0), *der* (5) y pivote (1).

En la Figura 3.22. (izquierda) se inicia la búsqueda de los elementos a intercambiar, que coinciden con los que están en ambos extremos de la sublista (2 y 0). El proceso sigue hasta encontrar nuevos elementos a intercambiar (4 y 1). Se intercambian y avanzan los índices de forma que se cruzan, como se ve en la Figura 3.22. (derecha), quedando la sublista particionada en: parte izquierda: (0,1) y parte derecha: (2,5).

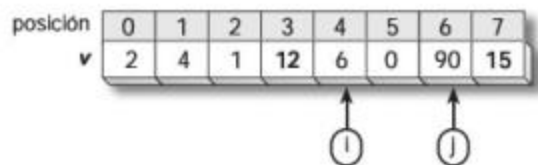
En la Figura 3.23. (izquierda) se realiza la llamada recursiva a *QuickSort(v, 0, 1)* para particionar la sublista (0,1), y se muestran los nuevos valores de *izq* (0), *der* (1) y pivote (0). Al realizar

1.ª llamada: *QuickSort*(v,0,7)

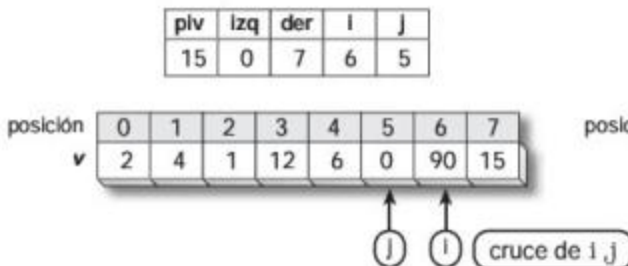
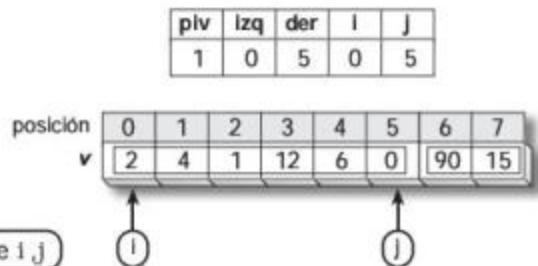
Búsqueda



Intercambio y avance de i, j

Figura 3.20. Traza del algoritmo *QuickSort* (1).

Búsqueda

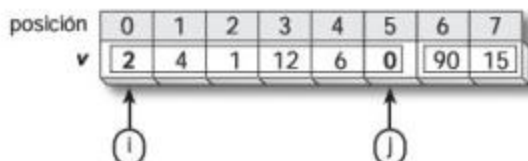
2.ª llamada: *QuickSort*(v,0,5)Figura 3.21. Traza del algoritmo *QuickSort* (2).

la búsqueda de los elementos a intercambiar, el índice *i* se queda en la posición 0 y *j* avanza hasta la posición 0. Se produciría un intercambio del 0 con él mismo, y los dos índices avanzarían de forma que *j* toma el valor -1 (se sale fuera del array) e *i* toma el valor 1. Se cruzan y en este caso no hay llamadas recursivas porque la partición izquierda no existe y la derecha es de 1 elemento: **parte derecha: (1,1)**. Por tanto, este procedimiento termina y se produce el retorno de la 3.ª llamada *QuickSort*(v,0,1), quedando los valores que ocupan las posiciones 0 y 1 definitivamente ordenados, como se indica en la Figura 3.23. (derecha). Ahora se realiza la llamada a *QuickSort*(v,2,5) para particionar la sublista (2,5). El nuevo pivote es el 12 y los valores de *izq* y *der* son 2 y 5 respectivamente.

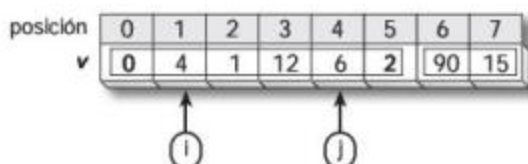
En la Figura 3.24. (izquierda) se comienza la búsqueda de los elementos a intercambiar y los índices quedan posicionados en las posiciones 3 y 5. Se intercambian los elementos y avanzan *i* y *j* quedando ambos apuntando a la posición 4. El índice *i* avanza hasta la posición 5 y el *j* se

Búsqueda

piv	izq	der	i	j
1	0	5	0	5

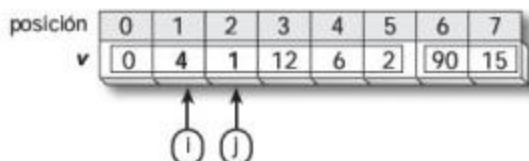


Intercambio y avance de i, j



Búsqueda

piv	izq	der	i	j
1	0	5	1	2



Intercambio y avance de i, j

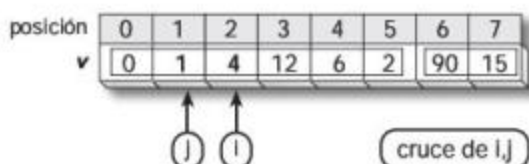
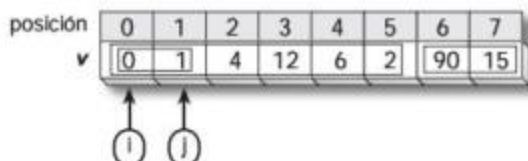


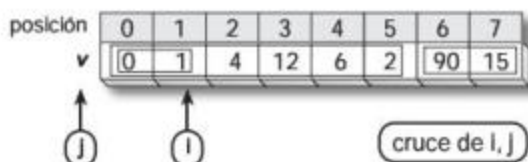
Figura 3.22. Traza del algoritmo QuickSort (3).

3.ª llamada: QuickSort(v,0,1)

piv	izq	der	i	j
0	0	1	0	1



Intercambio y avance de i, j



4.ª llamada: QuickSort(v,2,5)

piv	izq	der	i	j
12	2	5	2	5

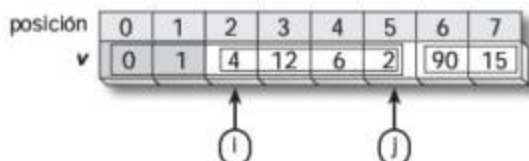
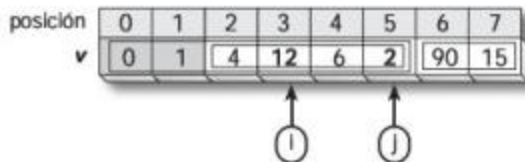


Figura 3.23. Traza del algoritmo QuickSort (4).

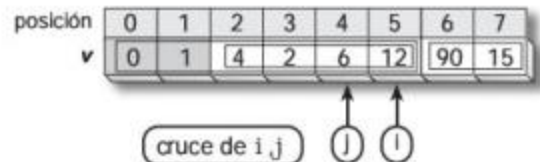
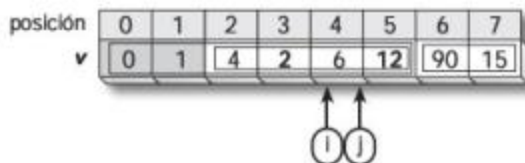
queda apuntando a la posición 4 (Figura 3.24, derecha). Como se han cruzado, se detiene el proceso de búsqueda quedando particionada la lista en **parte izquierda: (2,4)** y **parte derecha (5,5)**. En la Figura 3.25, (izquierda) se produce la llamada *QuickSort*(v,2,4) para particionar la sublista (2,4) tomando el pivote el valor 2. En la Figura 3.25, (derecha) los índices apuntan a las posiciones 2 y 3 cuyos valores deberán intercambiarse. Tras el intercambio, los índices avanzan de forma que se cruzan determinando las nuevas particiones: parte izquierda: (2,2) y parte derecha: (3,4).

Búsqueda

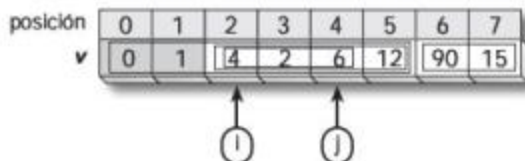
piv	izq	der	i	j
12	2	5	3	5

**Búsqueda y cruce**

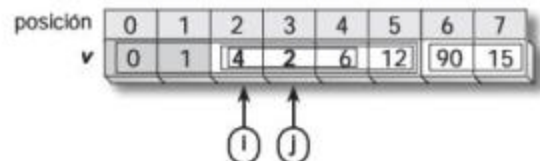
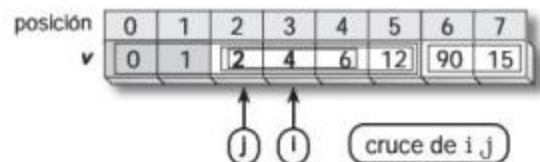
piv	izq	der	i	j
12	2	5	5	4

**Intercambio y avance de i, j**Figura 3.24. Traza del algoritmo *QuickSort* (5).**5.ª llamada: *QuickSort*(v,2,4)**

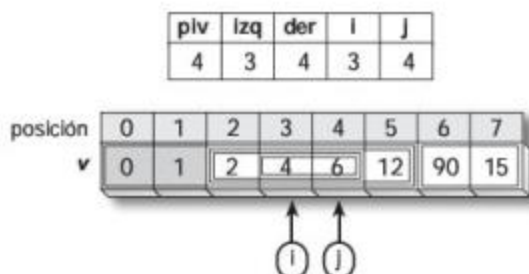
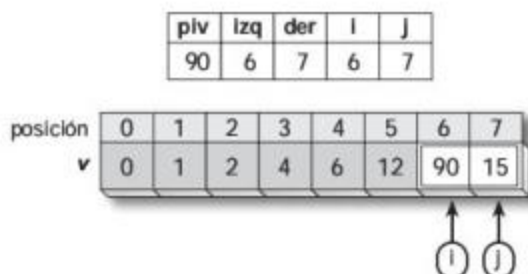
piv	izq	der	i	j
2	2	4	2	4

**Búsqueda y cruce**

piv	izq	der	i	j
12	2	5	5	4

**Intercambio y avance de i, j**Figura 3.25. Traza del algoritmo *QuickSort* (6).

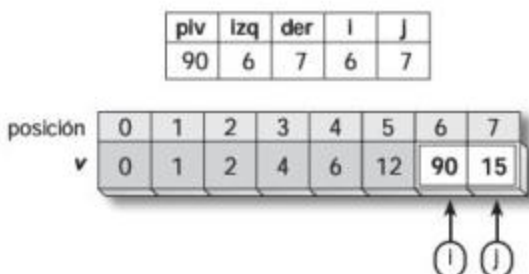
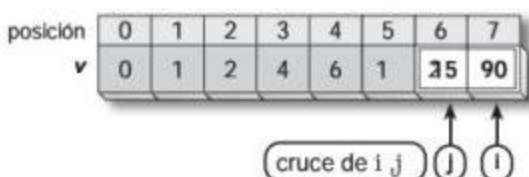
La sublista (2,2) al ser de un elemento ya está ordenada, como se indica en la Figura 3.26. (izquierda). Ahora se llama a *QuickSort*(v,3,4) para ordenar la sublista (3,4) siendo el nuevo pivote el valor 4. Esta llamada no produce ningún intercambio, pues los índices se cruzan sin encontrar valores que correspondan a la partición contraria. Aunque no se ve en la figura, las nuevas particiones serán: parte izquierda: (3,3) y parte derecha: (4,4), por lo que al ser ambas de un elemento, ya están ordenadas y no se producen las correspondientes llamadas recursivas. Por tanto, se retorna de la llamada

6.ª llamada: **QuickSort(v,3,4)**7.ª llamada: **QuickSort(v,6,7)**Retorno de 6.ª llamada: **QuickSort(v,3,4)**Retorno de 5.ª llamada: **QuickSort(v,2,4)**Retorno de 4.ª llamada: **QuickSort(v,2,5)**Retorno de 2.ª llamada: **QuickSort(v,0,5)**Figura 3.26. Traza del algoritmo *QuickSort* (7).

QuickSort(v,3,4) y también de las anteriores llamadas *QuickSort*(v,2,4), *QuickSort*(v,2,5) y *QuickSort*(0,5) como se indica en la Figura 3.26. (izquierda). En la Figura 3.26. (derecha) se muestra la llamada a *QuickSort*(v,6,7) que quedó pendiente cuando se realizó la primera partición. El nuevo pivote es el 90.

En la Figura 3.27. (izquierda), los índices *i* y *j* apuntan a los valores que deben intercambiarse. Se produce el intercambio, y al avanzar *i* y *j* se cruzan de forma que las nuevas particiones son: **parte izquierda: (6,6)** y **parte derecha: (7,7)**. Al ser ambas de un solo elemento no se producen las correspondientes llamadas recursivas, de forma que se retorna de la 7.ª llamada, quedando el array completamente ordenado como se muestra en la Figura 3.27. (derecha).

Búsqueda

Retorno de 7.ª llamada: **QuickSort(v,6,7)**Intercambio, avance y cruce de *i* y *j*Figura 3.27. Traza del algoritmo *QuickSort* (8).

Ejercicio 3.20. Traza del algoritmo de ordenación *HeapSort*

Enunciado Dado el siguiente array: {2, 4, 1, 15, 6, 0, 90, 12}, representado en la Figura 3.1., ordenarlo descendientemente utilizando el algoritmo del montículo o *HeapSort*, haciendo un seguimiento detallado de la evolución del algoritmo, representando el array y el árbol correspondiente.

Solución Como se explicó en el apartado teórico correspondiente al método *HeapSort*, se comienza con la construcción inicial del montículo, (fase 1), aplicando el método criba al primer nodo no hoja que es el $v[3]$. El 15 es mayor que su hijo izquierdo y por tanto, se hunde a la posición 7 ocupando su lugar el 12 como se puede ver en la Figura 3.28. (izquierda). En la Figura 3.28. (derecha) se aplica la criba al elemento que ocupa la posición 2, bajando hasta la posición 5 y subiendo el 0. Al aplicar la criba al nodo $v[1]$ no ocurre nada porque ya es menor o igual que sus hijos.

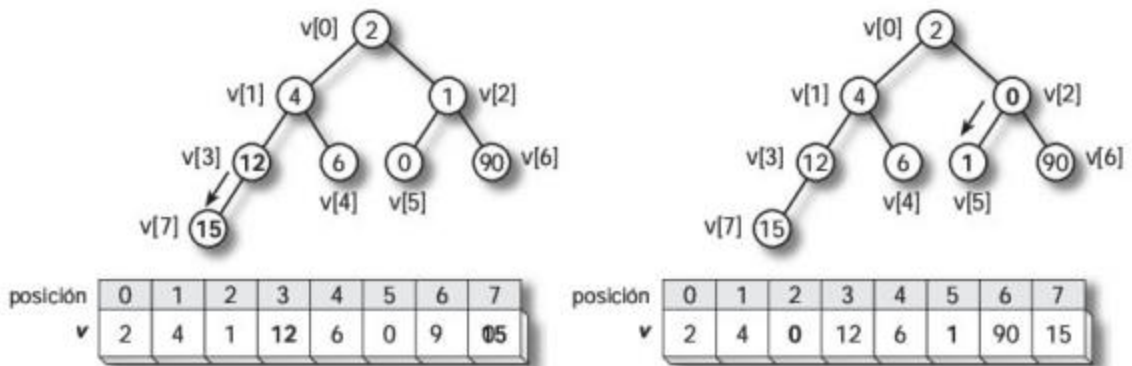


Figura 3.28. Traza del algoritmo *HeapSort*. Fase 1-1.

Y por fin le llega el turno al nodo raíz, que se hunde hasta la posición 5 como se puede observar en la Figura 3.29. El símbolo M indica que este árbol es un montículo, por lo que hemos terminado la fase 1. Se puede observar cómo el menor elemento del array, el 0, ocupa la posición 0.

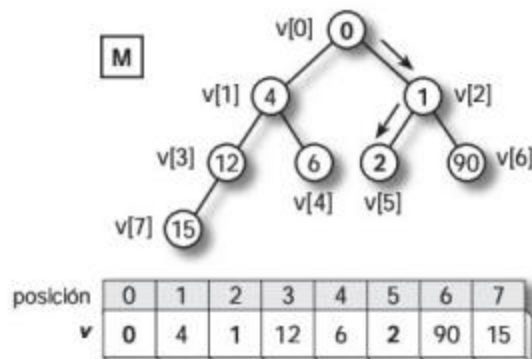


Figura 3.29. Traza del algoritmo *HeapSort*. Fase 1-2.

Ahora da comienzo la fase de ordenación, que consiste en intercambios y regeneraciones del montículo. En la Figura 3.30. (izquierda) se intercambia la raíz del montículo, con el último elemento, que ocupa la posición 7, quedando ya esta posición definitivamente ordenada y fuera del montículo. Pero vemos que el árbol resultante ya no es un *min heap* por lo que hay que regenerarlo. En la Figura 3.30. (derecha) se aplica la criba a la raíz y volvemos a obtener un montículo. La llamada al procedimiento criba sería: `criba (v, 0, 6)`. Se ve cómo el 15 ha bajado por la rama del hijo menor, subiendo el 1 hasta la raíz, pero el 15 debe seguir bajando hasta ocupar la posición 5, subiendo el 2 hasta la posición 2.

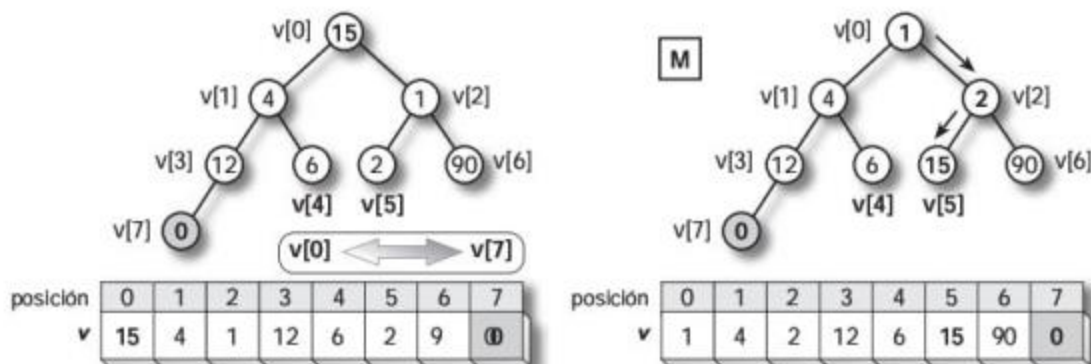


Figura 3.30. Trazo del algoritmo *HeapSort*. Fase 2-1.

En la Figura 3.31. (izquierda), se hace un nuevo intercambio con la raíz, pudiéndose ver sombreados los nodos $v[6]$ y $v[7]$ que ya quedan definitivamente ordenados y fuera del árbol. En la Figura 3.31. (derecha) se ve el resultado de aplicar la criba a la raíz para regenerar el montículo: `criba (v, 0, 5)`.

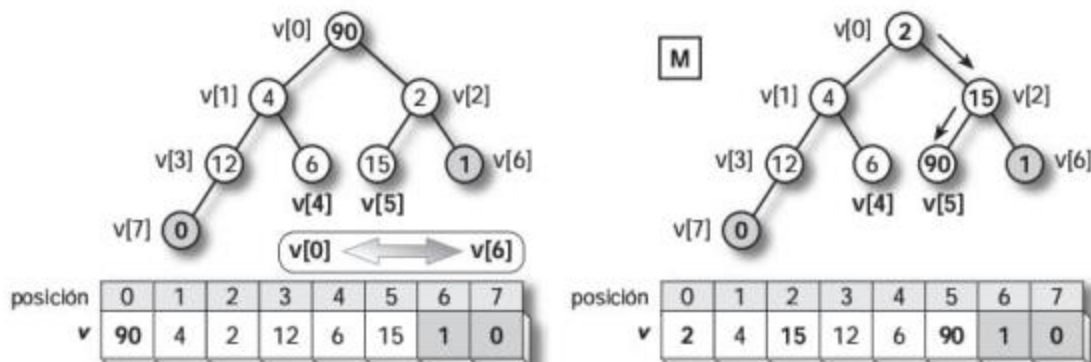


Figura 3.31. Trazo del algoritmo *HeapSort*. Fase 2-2.

En la Figura 3.32. (izquierda) se intercambian $v[0]$ con $v[5]$ quedando ya ordenados $v[5]$, $v[6]$ y $v[7]$. En la parte derecha de la figura se vuelve a regenerar el montículo aplicando la criba a la raíz: `criba (v, 0, 4)`.

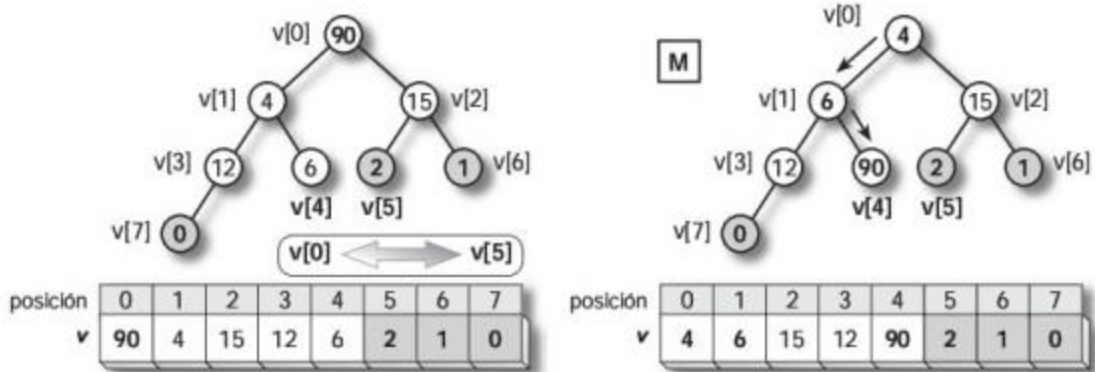


Figura 3.32. Traza del algoritmo *HeapSort*. Fase 2-3.

A continuación, se intercambia $v[0]$ con $v[4]$, quedando $v[4]$ fuera del árbol (véase Figura 3.33. izquierda). Luego se regenera el heap aplicando la criba a la raíz: `criba (v, 0, 3)` (véase Figura 3.33. derecha).

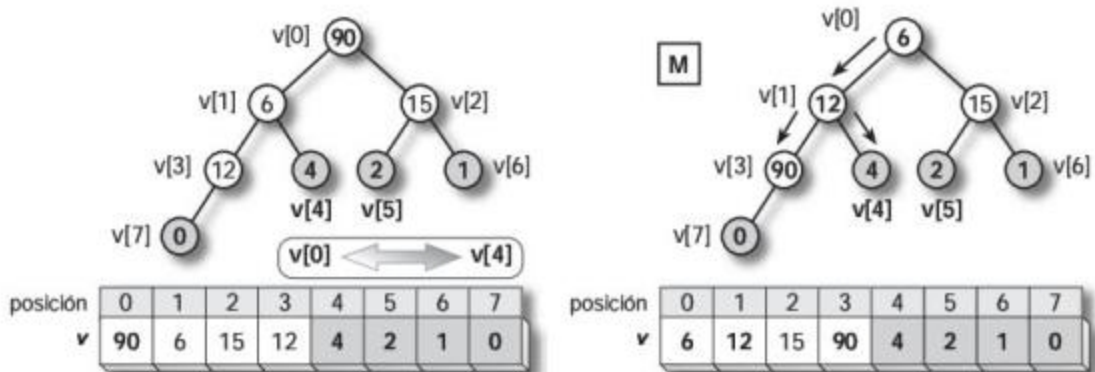
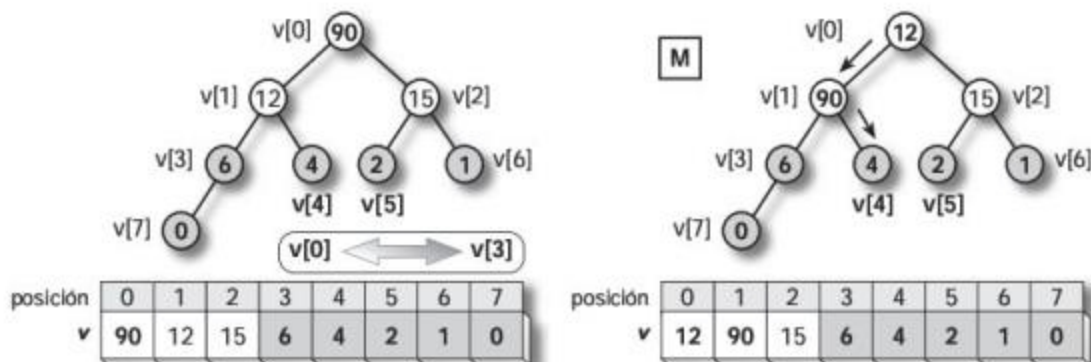
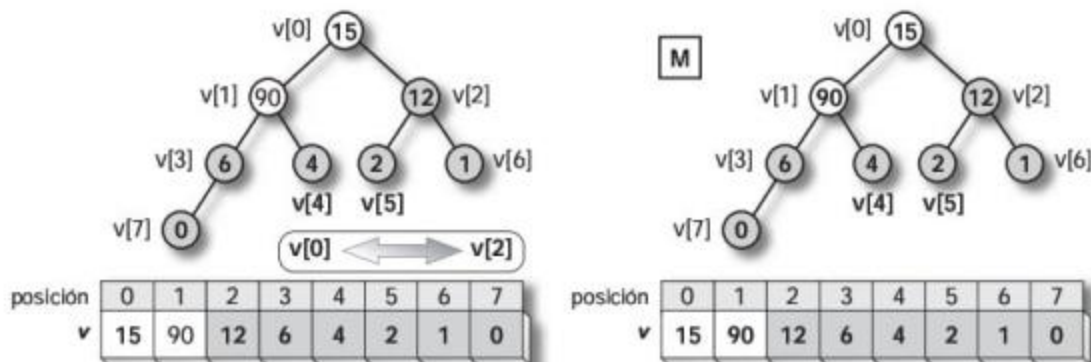


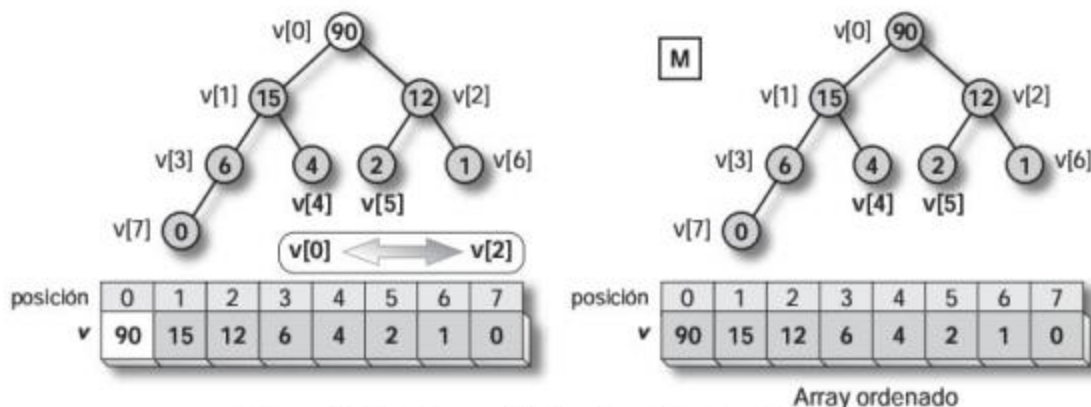
Figura 3.33. Traza del algoritmo *HeapSort*. Fase 2-4.

En la Figura 3.34. (izquierda) se intercambian $v[0]$ y $v[3]$ y se ve la sublista ordenada entre las posiciones 3 y 7, de forma que el árbol a regenerar ahora se reduce a 3 nodos. En la Figura 3.34. (derecha) se ve el resultado de aplicar la criba: `criba (v, 0, 2)`.

Se sigue con el proceso intercambiando $v[0]$ y $v[2]$ (véase Figura 3.35. izquierda) y vemos que, en este caso, al subir el 15 a la raíz el árbol sigue siendo un *min heap*. De todas formas seguimos aplicando la criba aunque ahora no tenga efecto: `criba (v, 0, 1)` (véase Figura 3.35. derecha).

Figura 3.34. Trazo del algoritmo *HeapSort*. Fase 2-5.Figura 3.35. Trazo del algoritmo *HeapSort*. Fase 2-6.

En la Figura 3.36, (izquierda) intercambiamos $v[0]$ con $v[1]$ y el árbol queda reducido a un solo elemento, con lo que el proceso termina. En la parte derecha vemos el array ordenado totalmente de forma descendente.

Figura 3.36. Trazo del algoritmo *HeapSort*. Fase 2-7.

Ejercicio 3.21. Algoritmo de ordenación *HeapSort*

Enunciado Desarrollar en pseudocódigo el procedimiento **criba** del algoritmo *HeapSort* para ordenación descendente. Codificar dicho algoritmo en Java.

Solución La cabecera será

```
procedimiento criba(vector, izq, der)
```

siendo los parámetros:

- **vector**: array que vamos a ordenar,
- **izq**: posición de la izquierda o raíz del árbol al cual se aplica el procedimiento criba. No tiene por qué coincidir con la primera posición del array,
- **der**: límite derecho del montículo. No tiene por qué coincidir con el extremo derecho del array.

Las variables locales utilizadas serán:

- **i, j**: índices para apuntar a nodos del árbol,
- **aux**: variable auxiliar para guardar el valor del elemento a "hundir",
- **salir**: variable booleana que determina cuándo podremos dejar de "hundir" el elemento.

Inicializaciones: hacemos que *i* apunte al nodo a hundir, y *j* a su hijo izquierdo. Guardamos en *aux* el valor del nodo a hundir, y hacemos que *salir* sea falso.

```
i izq  
j 2*i+1  
aux vector[i]  
salir false
```

Ahora entramos en un bucle para bajar por el árbol mientras no nos salgamos del montículo ($j \leq \text{der}$) y *salir* sea falso.

```
mientras (j <= der) y (no salir)
```

Si el nodo apuntado por *i* tiene dos hijos, hacemos que *j* apunte al hijo menor. Después, comprobamos si el valor de *aux* (valor a "hundir") es menor o igual que el hijo menor (apuntado por *j*) en cuyo caso, hacemos *salir* = cierto para finalizar el bucle. Si es mayor, debe seguir hundiéndose y entonces subimos el valor de *j* al lugar de *i* y hacemos que *i* apunte al lugar de *j* y *j* al hijo izquierdo de *i*.

```
Si (j < der)  
  Si (vector[j] > vector[j+1])  
    j++
```

```

        // j apunta al hijo menor
        fin-si
    fin-si

    Si (aux <= vector[j])
        salir true;
    Si-no
        vector[i] = vector[j]
        // subimos el valor del hijo menor
        i j
        j 2*i+1
        // j apunta a hijo izquierdo
    fin-Si
fin-mientras

```

Una vez fuera del bucle, se copia en la posición apuntada por *i* el valor que se ha hundido (*aux*)

```
vector[i]=aux;
```

Codificación en Java del algoritmo *HeapSort* completo para ordenación descendente

```

private static void criba(int [] vector, int izq, int der){
    // metodo criba utilizado por el metodo heapSort para ordenacion descendente
    int i,j;
    int aux; // variable auxiliar para intercambio de variables
    boolean salir;
    i=izq; j=2*i+1;
    // hijo izq: 2*i+1, hijo der: 2*i+2
    // i apunta al nodo padre y j al hijo izquierdo
    aux = vector[i];
    salir = false;
    while (j<=der && !salir){
        // mientras el indice j no se salga del monticulo y salir sea false
        if (j<der)
            if (vector[j] > vector[j+1]) j++;
            // j apunta al hijo menor
        if (aux <= vector[j]) salir= true;
        else {
            /*si el hijo (j) es menor que el padre (i), copiamos el
            * hijo en la posicion del padre y ahora el hijo se convierte
            * en padre (i) y apuntamos j a su hijo izquierdo
            */
            vector[i] = vector[j];

```



```

        i=j;
        j=2*i+1; // apunta a hijo izquierdo
    } // fin del else
} // fin del while
//al salir del bucle, copiamos en el ultimo i el elemento aux
vector[i]=aux;
} //fin del metodo criba

```

```

public static void heapSort(int [] vector, int num){
    // ordena por el algoritmo del monticulo (Heap Sort) una matriz de
    // num elementos utiles.
    int i,j;
    int ultimo = num-1;
    int aux; // variable auxiliar para intercambio
    // FASE 1: construir monticulo inicial
    for (i = (ultimo - 1) / 2; i>=0; i--) criba (vector,i,ultimo);
    // FASE 2: ordenacion
    for (i = ultimo; i>=1; i-){
        // cambiar el primero por vector[i]
        aux=vector[0]; vector[0]=vector[i]; vector[i]=aux;
        // y reconstruir el monticulo entre 0 y i-1. Hemos quitado el
        // ultimo del arbol
        criba(vector,0,i-1);
    }
}

```

Ejercicio 3.22. Algoritmo de ordenación *HeapSort*

Enunciado Modificar el algoritmo *HeapSort* para realizar una ordenación ascendente. Razonar la modificación y codificar el algoritmo en Java.

Solución Como se vio en la parte teórica, para ordenar ascendentemente un array basta con utilizar un montículo descendente o *max Heap*, de forma que se cumpla que cada nodo sea mayor o igual que sus hijos, es decir que esté formado por una secuencia de valores $h_0, h_1, h_2, h_3, \dots, h_n$ tales que

$$h_i \geq h_{2i+1}$$

$$h_i \geq h_{2i+2}$$

Así, se puede conservar todo lo que sea externo al procedimiento *criba*:

```

desde i = (ultimo - 1) / 2 hasta 0
    criba (vector,i,ultimo);
desde i = ultimo hasta 1 hacer

```

```

// intercambiar la raiz con el elemento de la posicion i
aux = vector[0]; vector[0] = vector[i]; vector[i] = aux;
// reconstruir el monticulo entre 0 e i-1.
criba(vector,0,i-1)
fin-desde

```

En el procedimiento *criba*, se deben realizar las siguientes modificaciones:

```

1. Si (j < der)
2.   Si (vector[j] < vector[j+1])
3.     j++
4.     // j apunta al hijo mayor
5.   fin-si
6. fin-si

7. Si (aux >= vector[j])
8.   salir = true;

```

En la línea 2, se cambia el sentido del operador relacional ">" por "<", de forma que el índice *j* apunte al hijo mayor.

En la línea 7, se cambia el operador relacional "<=" por el ">=", de forma que cambie la condición de salida del bucle: saldrá cuando el valor de *aux* sea mayor o igual que el hijo mayor.

Codificación en Java del algoritmo *HeapSort* completo para ordenación ascendente

```

private static void criba(int [] vector, int izq, int der){
    // metodo criba utilizado por el metodo heapSort para ordenacion ascendente
    int i,j;
    int aux; // variable auxiliar para intercambio de variables
    boolean salir;
    i=izq; j=2*i+1;
    // hijo izq: 2*i+1, hijo der: 2*i+2
    // i apunta al nodo padre y j al hijo izquierdo
    aux = vector[i];
    salir = false;
    while (j<=der && !salir){
        // mientras el indice j no se salga del monticulo y salir sea false
        if (j<der)
            if (vector[j] < vector[j+1]) j++;
            // j apunta al hijo mayor
        if (aux >= vector[j]) salir= true;
        else {
            //si el hijo (j) es mayor que el padre (i), copiamos el

```

```

        * hijo en la posición del padre y ahora el hijo se convierte
        * en padre (i) y apuntamos j a su hijo izquierdo
        */
        vector[i] = vector[j];
        i=j;
        j=2*i+1; // apunta a hijo izquierdo
    } // fin del else
} // fin del while
//al salir del bucle, copiamos en el ultimo i el elemento aux
vector[i]=aux;
} //fin del metodo criba

public static void heapSort(int [] vector, int num){
    // ordena por el algoritmo del monticulo (Heap Sort) una matriz de
    // num elementos utiles.
    int i,j;
    int ultimo = num-1;
    int aux; // variable auxiliar para intercambio
    // FASE 1: construir monticulo inicial
    for (i = (ultimo - 1) / 2; i>=0; i--) criba (vector,i,ultimo);
    // FASE 2: ordenación
    for (i = ultimo; i>=1; i-){
        // cambiar el primero por vector[i]
        aux=vector[0]; vector[0]=vector[i]; vector[i]=aux;
        // y reconstruir el monticulo entre 0 y i-1. Hemos quitado el
        // ultimo del arbol
        criba(vector,0,i-1);
    }
}

```

Ejercicio 3.23. Ordenación de un array de cadenas por longitud

Enunciado Aplicar el método de la burbuja a un array de `Strings`, de forma que las cadenas queden ordenadas por orden creciente de longitud.

Solución Como se vio en la parte teórica, todos los algoritmos de ordenación realizan permutaciones de elementos dependiendo del resultado de ciertas comparaciones. Hasta ahora sólo se han ordenado arrays de enteros, por lo que éstos se permutan dependiendo de las comparaciones de sus valores. En este ejercicio se permutarán cadenas atendiendo al resultado de la comparación de sus longitudes.

Se utilizará el método `length` de la clase `String`, que devolverá el número de caracteres que tiene la cadena.

En la versión que se vio en el Ejercicio 3.9., la sentencia que realizaba la comparación era:

```
if (vector[j] > vector [j+1])
```

Ahora se transformará en :

```
if (vector[j].length() > vector [j+1].length())
```

El código completo del método es el siguiente:

```
public static void burbujaLong(String [] vector, int num){
    // ordena por el algoritmo de la burbuja una matriz de num Strings
    // según su tamaño

    String aux;
    for (int i=1; i<num; i++)
        for (int j=0; j<num - i; j++)
            if (vector[j].length() > vector [j+1].length()){
                aux = vector [j];
                vector [j] = vector [j+1];
                vector [j+1] = aux;
            }
}
```

Aplicando el método a un array formado por las siguientes cadenas:

```
zona, hola, halo, boa, agua, miercoles, zorro, archidiocesis, alguacil
```

se obtendría la siguiente ordenación:

```
boa, zona, hola, halo, agua, zorro, alguacil, miercoles, archidiocesis
```

Ejercicio 3.24. Ordenación de un array de cadenas por orden alfabético

Enunciado Aplicar el método de la burbuja a un array de `Strings`, de forma que las cadenas queden ordenadas por orden alfabético.

Solución El ejercicio es muy parecido al ejercicio anterior, pero ahora el criterio de comparación es el orden alfabético o lexicográfico.

Se utilizará el método `compareTo()` de la clase `String` de la siguiente manera: si `a` y `b` son dos cadenas y las queremos comparar lexicográficamente, entonces se invocará dicho método por parte de una de las cadenas enviando la otra como parámetro:

a.compareTo(b) devolverá:

-1 si a es menor lexicográficamente que b.

0, si son iguales.

+1 si a es mayor lexicográficamente que b.

El método codificado en Java es el siguiente:

```
public static void burbujaAlfa(String [] vector, int num){
    String aux;
    for (int i=1; i<num; i++)
        for (int j=0; j<num - i; j++)
            if (vector[j].compareTo(vector [j+1]) > 0){
                aux = vector [j];
                vector [j] = vector [j+1];
                vector [j+1] = aux;
            }
}
```

Si se aplica el método al array del ejercicio anterior:

zona, hola, halo, boa, agua, miercoles, zorro, archidiocesis, alguacil

se obtiene la siguiente ordenación:

agua, alguacil, archidiocesis, boa, halo, hola, miercoles, zona, zorro

Ejercicio 3.25. Ordenación de un array de cadenas por varios criterios

Enunciado Aplicar el método de la burbuja a un array de `Strings`, de forma que las cadenas queden ordenadas por longitud, y en caso de tener la misma longitud, por orden alfabético.

Solución En este caso, las múltiples comparaciones que hay que realizar complican bastante el código, por lo que es aconsejable implementar un método que compare dos elementos del array, por los criterios especificados, y devuelva tres posibles valores: -1, si el primer elemento es menor (según dichos criterios) que el segundo, 0, si son iguales y 1 si el primero es mayor que el segundo.

La codificación en Java de ambos métodos sería la siguiente:

```
private static int comparar (String s1, String s2){
    if (s1.length()>s2.length())
        return 1;
    else if (s1.length()<s2.length())
        return -1;
```

```

        // son de la misma longitud, comparar lexicográficamente
    else if (s1.compareTo(s2)<0)
        return -1;
    else if (s1.compareTo(s2)>0)
        return 1;
    else return 0;
}

public static void burbujaLongAlfa(String [] vector, int num){
    // ordena por el algoritmo de la burbuja una matriz de num Strings
    // según su tamaño y para cadenas del mismo tamaño, según su orden
    // lexicográfico.

    String aux;

    for (int i=1; i<num; i++)
        for (int j=0; j<num - i; j++)
            if (comparar(vector[j], vector [j+1]) > 0){
                aux = vector [j];
                vector [j] = vector [j+1];
                vector [j+1] = aux;
            }
}

```

En esta solución se ha implementado el método *comparar* como estático, aunque en temas posteriores se verá cómo crear objetos, siendo entonces más conveniente utilizar métodos no estáticos, similares al método `compareTo ( )` de la clase `String`.

Si se aplica este método al array de los ejercicios anteriores:

zona, hola, halo, boa, agua, miercoles, zorro, archidiocesis, alguacil

se obtiene la siguiente ordenación:

boa, agua, halo, hola, zona, zorro, alguacil, miercoles, archidiócesis

Donde se ve que las palabras de letras se han ordenado por orden alfabético.

Ejercicio 3.26. Ejercicio resumen de métodos de ordenación

Enunciado Como ejercicio de recapitulación, se realizará un programa completo donde se aplicarán todos los métodos estudiados en este tema, para ordenar un array de cadenas por los criterios de longitud y orden lexicográfico, como se ha visto en el ejercicio anterior.

Solución Lo más importante de este ejercicio es detectar dónde están las sentencias de comparación en cada algoritmo y utilizar en ellas una llamada al método `comparar ( )` desarrollado en el Ejercicio 3.24., donde se ha implementado el método de la burbuja.

En el resto de los métodos, las sentencias de comparación para arrays de números enteros serán las siguientes:

Inserción directa:

```
while (j >= 0 && aux < a[j])
```

Selección directa:

```
if (vector[j] < menor)
```

QuickSort:

```
while (vector [i] < pivote) i++;
while (vector [j] > pivote) j--;
```

HeapSort (método criba):

```
if (vector[j] < vector[j+1]) j++;
if (aux >= vector[j]) salir= true;
```

y se sustituirán por:

Inserción directa:

```
while (j >= 0 && comparar(aux,a[j]) < 0)
```

Selección directa:

```
if (comparar(vector[j], menor) < 0)
```

QuickSort:

```
while (comparar(vector [i], pivote) < 0) i++;
while (comparar(vector [j], pivote) > 0) j--;
```

HeapSort (método criba):

```
if (comparar (vector[j],vector[j+1]) < 0) j++;
if (comparar (aux, vector[j]) >= 0) salir= true;
```

A continuación, se muestra el código completo del programa en Java:

```
public class OrdenacionStringsTodos {
    private static int comparar (String s1, String s2){
        if (s1.length()>s2.length())
            return 1;
```

```

        else if (s1.length()<s2.length())
            return -1;
        else if (s1.compareTo(s2)<0)
            return -1;
        else if (s1.compareTo(s2)>0)
            return 1;
        else return 0;
    }

    public static void burbuja(String [] vector, int num){
        // ordena por el algoritmo de la burbuja una matriz de num elementos
        // utiles.
        String aux;
        for (int i=1; i<num; i++)
            for (int j=0; j<num - i; j++)
                if (comparar(vector[j],vector [j+1]) > 0){
                    aux = vector [j];
                    vector [j] = vector [j+1];
                    vector [j+1] = aux;
                }
    }

    public static void burbujaMejorado(String [] vector, int num){
        // ordena por el algoritmo de la burbuja una matriz de num elementos
        // utiles.
        String aux;
        int i=1; // num de pasada
        boolean intercambio = true;
        while (intercambio){
            intercambio = false;
            for (int j=0; j<num - i; j++)
                if (comparar(vector[j],vector [j+1]) > 0){
                    aux = vector [j];
                    vector [j] = vector [j+1];
                    vector [j+1] = aux;
                    intercambio = true;
                }
            i++;
        }
    }

    public static void insercionDirecta(String [] a, int num){
        // ordena por el algoritmo de insercion directa una matriz de
        // num elementos utiles.
        int i,j;

```



```

String aux;
for (i = 2; i<num; i++){
    aux = a[i];
    j= i - 1;
    while (j >= 0 && comparar(aux,a[j]) < 0) {
        a[j + 1] = a[j];
        j--;
    }
    a[j + 1] = aux;
} // for
}

public static void seleccionDirecta(String [] vector, int num){
// ordena por el algoritmo de selección directa una matriz de
// num elementos utiles.

String menor;
int i,j,pos;
for (i = 0;i < num-1; i++){
    menor = vector[i];
    pos = i;
    // buscamos el menor de la sublista derecha
    for (j = i+1; j < num ; j++){
        if (comparar(vector[j], menor) < 0){
            menor = vector[j];
            pos = j;
        }
    }
    vector[pos] = vector[i];
    vector[i] = menor;
}
}

public static void quickSort(String [] vector, int izq, int der){
// ordena por el algoritmo de quick sort una matriz de enteros
// comprendidos entre las posiciones izq y der

int i,j; // indices
String pivote;
String aux; // var auxiliar para intercambio
pivote = vector[(izq+der) / 2];
i = izq; j = der;

do{
    while (comparar(vector [i], pivote) < 0) i++;
    while (comparar(vector [j], pivote) > 0) j--;
}

```

```

        if (i<=j) {
            aux = vector [i];
            vector[i] = vector[j];
            vector[j] = aux;
            i++;
            j--;
        } // end del if
    } while (i<=j);

    if (izq < j) quickSort(vector,izq,j);
    if (i<der) quickSort(vector,i,der);

} // fin de quickSort

private static void criba(String [] vector, int izq, int der){
    // metodo criba utilizado por el metodo heapSort

    int i,j;
    String aux; // variable auxiliar para intercambio de variables
    boolean salir;
    i=izq; j=2*i+1;
    // hijo izq: 2*i+1, hijo der: 2*i+2
    // i apunta al nodo padre y j al hijo izquierdo
    aux = vector[i];
    salir = false;
    while (j<=der && !salir){
        // mientras el indice j no se salga del monticulo y salir sea false
        if (j<der)
            if (comparar (vector[j],vector[j+1]) < 0) j++;
            // j apunta al hijo mayor
        if (comparar (aux, vector[j]) >= 0) salir= true;
        else {
            vector[i] = vector[j];
            i=j;
            j=2*i+1; // apunta a hijo izquierdo
        } // fin del else
    } // fin del while
    //al salir del bucle, copiamos en el ultimo i el elemento aux
    vector[i]=aux;
} //criba

public static void heapSort(String [] vector, int num){
    // ordena por el algoritmo del monticulo (Heap Sort) una matriz de
    // num elementos utiles.

```

```

int i,j;
int ultimo = num-1;
String aux; // variable auxiliar para intercambio
// FASE 1: construir monticulo inicial
for (i = (ultimo - 1) / 2; i>=0; i--) criba (vector,i,ultimo);
// FASE 2: ordenación

for (i = ultimo; i>=1; i-){
    // cambiar el primero por vector[i]
    aux=vector[0]; vector[0]=vector[i]; vector[i]=aux;
    // y reconstruir el monticulo entre 0 y i-1. Hemos quitado el
    // ultimo del arbol
    criba(vector,0,i-1);
}

}

public static void visualizarMatriz (String [] matriz,int num){
    for (int i=0; i<num; i++)
        System.out.print(matriz[i] + ", ");
    System.out.println();
}

public static void copiarMatriz (String [] m1, String [] m2, int n){
    // copia los n primeros elementos de m2 en m1
    for (int i=0;i<n;i++)
        m1[i]=m2[i];
}

public static void main(String args[]){
    String listaOriginal[] =
{"zona","hola","halo","boa","agua","miercoles","zorro","archidiocesis","alguacil"};
    int n=listaOriginal.length;
    String lista [] = new String [n];

    copiarMatriz(lista,listaOriginal,n);
    System.out.println("Matriz Original: ");
    visualizarMatriz(lista,n);
    System.out.println("Algoritmo de la burbuja: ");
    burbuja(lista,n);
    visualizarMatriz(lista,n);

    copiarMatriz(lista,listaOriginal,n);
    System.out.println("Algoritmo de la burbuja mejorado: ");
    burbujaMejorado(lista,n);
}

```

```

        visualizarMatriz(lista,n);

        copiarMatriz(lista,listaOriginal,n);
        System.out.println("Algoritmo de Insercion Directa: ");
        insercionDirecta(lista,n);
        visualizarMatriz(lista,n);

        copiarMatriz(lista,listaOriginal,n);
        System.out.println("Algoritmo de Seleccion Directa: ");
        seleccionDirecta(lista,n);
        visualizarMatriz(lista,n);

        copiarMatriz(lista,listaOriginal,n);
        System.out.println("Algoritmo Quick Sort: ");
        quickSort(lista,0,n-1);
        // desde la posicion 0 a la n-1
        visualizarMatriz(lista,n);

        copiarMatriz(lista,listaOriginal,n);
        System.out.println("Algoritmo Heap Sort: ");
        heapSort(lista,n);
        visualizarMatriz(lista,n);
    }
}

```

La salida de este programa será:

```

Matriz Original:
zona, hola, halo, boa, agua, miercoles, zorro, archidiocesis, alguacil,
Algoritmo de la burbuja:
boa, agua, halo, hola, zona, zorro, alguacil, miercoles, archidiocesis,
Algoritmo de la burbuja mejorado:
boa, agua, halo, hola, zona, zorro, alguacil, miercoles, archidiocesis,
Algoritmo de Insercion Directa:
boa, agua, halo, zona, hola, zorro, alguacil, miercoles, archidiocesis,
Algoritmo de Seleccion Directa:
boa, agua, halo, hola, zona, zorro, alguacil, miercoles, archidiocesis,
Algoritmo Quick Sort:
boa, agua, halo, hola, zona, zorro, alguacil, miercoles, archidiocesis,
Algoritmo Heap Sort:
boa, agua, halo, hola, zona, zorro, alguacil, miercoles, archidiocesis,

```


Algoritmos recursivos

4.1. Introducción

En este capítulo se presenta la recursividad como una poderosa herramienta de programación, siendo una alternativa a las estructuras de control iterativas. En general, las soluciones recursivas son menos eficientes que las iterativas; sin embargo, en muchos casos, la recursividad permite especificar soluciones simples, elegantes y naturales a problemas que de otro modo serían muy difíciles de resolver. La recursividad es, quizá, uno de los temas más difíciles de entender por los estudiantes que se inician en la programación y por eso, para facilitar una rápida y profunda comprensión de esta técnica y para que el estudiante se habitúe a su uso, se presenta acompañada de sencillos ejemplos e ilustraciones; se compara con las técnicas iterativas, explicando las ventajas e inconvenientes de cada una de ellas para que en todo momento sea posible decidir cuál es la más conveniente.

4.2. Conceptos básicos de recursividad

Se dice que un método es recursivo si contiene llamadas o invocaciones a sí mismo.

Un método recursivo tendría este aspecto:

```
... metodoRecursivo (...){  
    .....  
    metodoRecursivo(...);  
    //llamada recursiva  
    .....  
}
```

Esto implica que una llamada al método recursivo puede generar una o más invocaciones al mismo método, que a su vez genera otras invocaciones, y así sucesivamente.

A primera vista, parece que un método recursivo nunca terminará; pero si la definición está bien hecha y los parámetros de entrada son adecuados, alguna llamada activará un método que no generará nuevas llamadas, con lo cual este método terminará y devolverá el control a la llamada anterior, que a su vez terminará, y así sucesivamente, hasta que todas las llamadas terminen.

El concepto de recursividad es más general, y es ampliamente utilizado en Matemáticas donde muchas funciones se definen recursivamente, es decir, en términos de sí mismas. Una de las funciones matemáticas recursivas más sencillas es la **función factorial** que utilizaremos para ilustrar pormenorizadamente el funcionamiento de la recursividad como técnica de programación.

Se define el factorial de un número natural n , $n!$, de la siguiente forma:

$$n! = n * (n - 1)!, \text{ para } n > 1$$

$$1! = 1$$

Como se puede ver, la definición tiene una parte recursiva que es la primera línea, pues define el factorial de n como el producto de n por el factorial de $n - 1$, para n mayor que 1. Sin embargo, la segunda línea de la definición es no recursiva y define el caso base. En general, cualquier definición recursiva tendrá dos casos bien diferenciados: el caso recursivo y el caso base o no recursivo.

Teniendo en cuenta la definición anterior sería sencillo programar un método para calcular dicha función:

```
static int factorial( int n){
    if (n>1)
        return factorial(n-1)*n;
        // caso recursivo
    else
        return 1;
        // caso base
}
```

donde se distinguen claramente el caso recursivo y el caso base.

La Figura 4.1. ilustra lo que ocurre cuando se hace una llamada a este método con $n = 4$. Como se puede observar, en la primera llamada (paso 1) se produce la ejecución del método factorial con $n = 4$; al ser $n > 1$ se ejecuta la sentencia

```
return factorial(n-1)*n;
```

Esta sentencia no se puede completar hasta que no se evalúe la expresión "`factorial(n-1)*n`" que incluye la llamada *factorial* ($n - 1$), es decir *factorial* (3). Por tanto, se producirá una nueva activación del método factorial (paso 2), con una variable $n = 3$; de nuevo se ejecutará la sentencia

```
return factorial(n-1)*n
```

y provocará otra llamada a `factorial(2)` (paso 3), que producirá una nueva activación del método con $n = 2$. Otra vez se ejecutará la sentencia `return factorial(n-1)*n`, y provocará la llamada `factorial(1)` (paso 4), se ejecutará el método con $n = 1$, y por primera vez se entrará en el caso base ejecutándose la sentencia `return 1;`

Por tanto, esta cuarta copia activa del método `factorial` terminará y podrá devolver el resultado 1 (paso 5). Ahora la tercera copia del método que estaba esperando a que se completara la llamada `factorial(1)` podrá realizar el producto `2 * factorial(1)` que dará 2 como resultado y podrá devolver ese valor a la segunda copia del método (paso 6) que podrá entonces evaluar la expresión `3 * factorial(2)` dando 6 como resultado, valor que devolverá a la primera copia del método (paso 7), que por fin podrá devolver el resultado de la expresión `4 * factorial(3)`, que será 24 (paso 8).

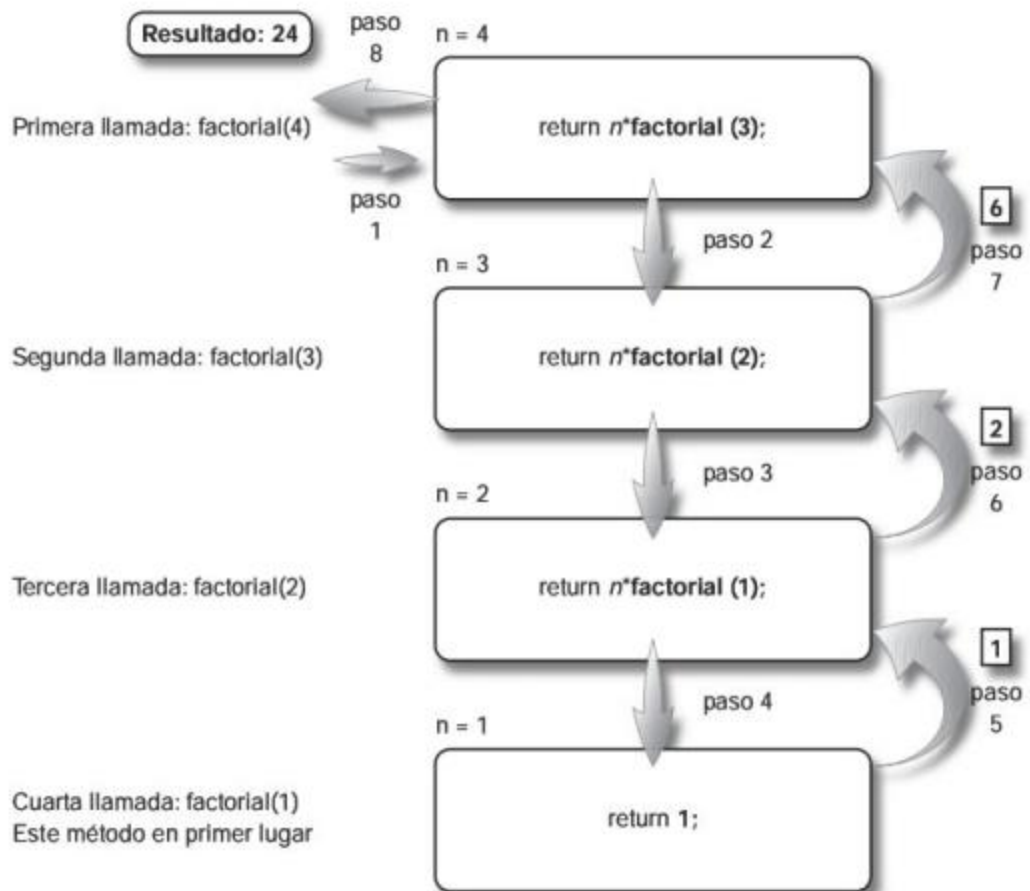


Figura 4.1. Seguimiento de la llamada recursiva `factorial(4)`.

La recursividad y la iteración son los dos mecanismos que suministran los lenguajes de programación para describir cálculos que deben repetirse un cierto número de veces. Normalmente puede pasarse de un planteamiento recursivo a uno iterativo y viceversa. En general, hay un conjunto de variables locales ligadas al método recursivo y cada vez que se activa recursivamente un método, se crea un nuevo conjunto de variables locales ligadas a él, siendo estas variables distintas, aunque tengan el mismo nombre.

Condiciones que debe cumplir todo método recursivo

Siempre hay que asegurar que existe una condición de salida, es decir, si se cumple esa condición, estaremos en el caso base, donde no se producen llamadas recursivas.

Cuando se diseña un algoritmo recursivo hay que identificar qué casos se pueden dar, y qué solución se aplica a cada caso.

En el ejemplo del factorial:

Casos	Solución
$n = 1$ (caso base)	return 1
$n > 1$	return $n * \text{factorial}(n - 1)$

Además, es conveniente comprobar que:

- Entre el caso base y los casos no base, se han cubierto todos los estados posibles.
- Cada llamada, en el caso no base, conduce a problemas cada vez más pequeños que necesariamente terminarán en el caso base.

4.3. Cuándo debe utilizarse la recursividad

Como se comentó anteriormente, todo problema recursivo se puede convertir en iterativo. Los algoritmos recursivos son especialmente apropiados cuando el problema a resolver o los datos a tratar se definen de forma recursiva. De esta forma se simplifica mucho el algoritmo. Pero, normalmente, la solución recursiva tiene un coste de tiempo y memoria mayor que la iterativa. Es decir, los programas recursivos en general son menos eficientes. Podríamos utilizar los siguientes consejos:

- Los algoritmos que por naturaleza son recursivos, y donde la solución iterativa es complicada y debe manejarse explícitamente una pila para emular las llamadas recursivas, deben resolverse por métodos recursivos.
- Cuando haya una solución obvia por iteración al problema, debe evitarse la recursividad.

A continuación, se verá un ejemplo donde no debe utilizarse la recursividad, aunque la solución recursiva es muy sencilla y natural pero extremadamente ineficiente. La solución iterativa es mucho más eficiente aunque no tan trivial.

Ejemplo Realizar un método recursivo que calcule el término n de la sucesión de Fibonacci. Para ello contamos con la siguiente definición recursiva:

$\text{Fibonacci}(n) = n$, si $n = 0$, o $n = 1$;

$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$, si $n > 1$

El algoritmo recursivo es muy sencillo de programar pero es extremadamente ineficiente:

```
public int fib (int n){  
    // siendo n un número entero no negativo  
    if (n > 1)  
        return fib(n-1) + fib(n-2);  
        // caso recursivo: para n>1  
    else  
        return n;  
        // caso base: para n=0 o n=1  
}
```

En el caso de que la primera llamada fuera $\text{fib}(6)$, las llamadas que se desencadenan son las que se ilustran en la Figura 4.2., donde se han marcado en tono oscuro las llamadas "terminales", o que terminan sin provocar otras llamadas recursivas.

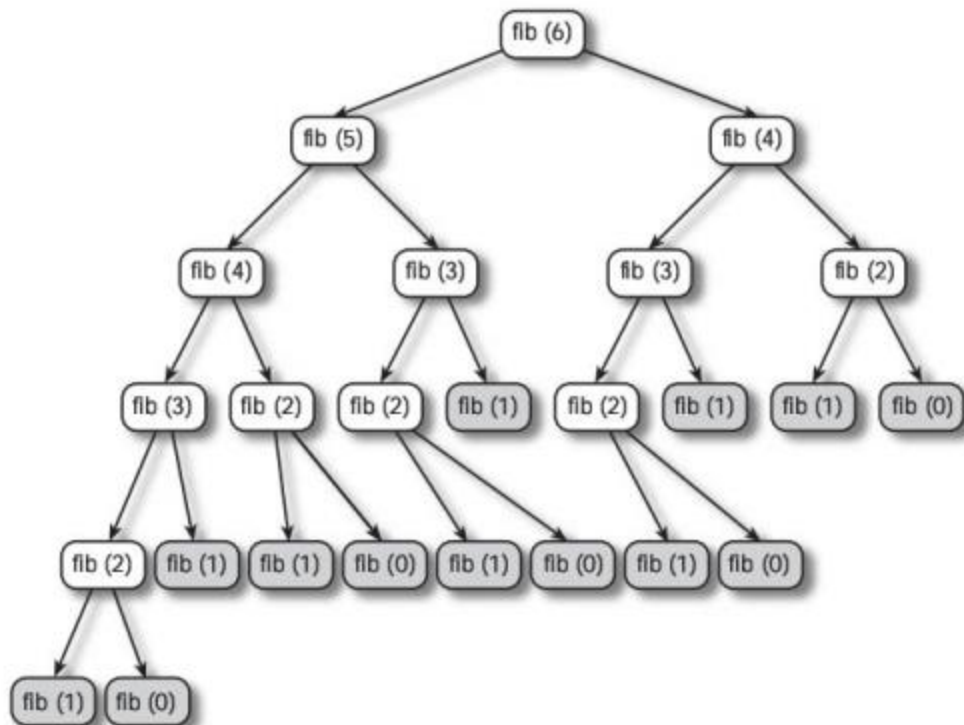


Figura 4.2. Llamadas producidas en el cálculo recursivo de $\text{fib}(6)$.

Como puede verse, se produce una "explosión" exponencial de llamadas recursivas repitiéndose muchos cálculos innecesariamente.

Veamos ahora la solución iterativa, donde se van calculando los términos de la serie desde el primero hasta n .

```
static int fib(int n){
    // n sera un número entero no negativo
    int i,x,y,z;
    if (n<=1)
        return n;
    else {
        x=0;
        y=1;
        for (i=2; i<=n; i++){
            z:=x+y;
            x:=y;
            y:=z;
        }
        return z;
    }
}
```

No parece una solución tan natural, pero es mucho más eficiente.

Proceso de activación de los métodos

Cuando se hace una llamada a un método, se guardan en la pila una serie de valores relacionados con ese método, entre otros, los parámetros formales y las variables locales. Cuando el método termina, se descargan de la pila esos valores. Si desde ese método se hace una llamada recursiva a él mismo, se cargan en la pila los valores de la segunda llamada sobre la primera. Si se vuelve a hacer una llamada, se vuelven a cargar sobre la segunda, y así sucesivamente.

A continuación, se mostrará un seguimiento de lo que ocurre en la pila con una llamada al método recursivo **factorial** con la llamada factorial (5): en la Figura 4.3, se muestran los diferentes estados por los que pasa la pila representados en cada columna de la tabla. Cuando se ha realizado la primera llamada, factorial (5), se cargan en la pila los valores asociados a ese método activado (véase paso 1). Cuando, dentro de ese método, se llama a factorial (4), se cargan en la pila sus valores asociados sobre la zona perteneciente a la primera activación del método (véase paso 2); y así sucesivamente, hasta que el método factorial (0) puede terminar ya que entra en su caso base; entonces, se descarga de la pila (véase paso 7), y los demás métodos pueden ir terminando y descargándose de la pila, hasta que termina el primer método cargado, factorial (5), y la pila queda vacía (véase paso 12).

En la Figura 4.4, se ve la evolución de la pila para el método recursivo que calcula la función de Fibonacci. En este caso la primera llamada será fib (4):

					fact(0)						
				fact(1)	fact(1)	fact(1)					
			fact(2)	fact(2)	fact(2)	fact(2)	fact(2)				
		fact(3)	fact(3)	fact(3)	fact(3)	fact(3)	fact(3)	fact(3)			
	fact(4)	fact(4)	fact(4)	fact(4)	fact(4)	fact(4)	fact(4)	fact(4)	fact(4)		
fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	fact(5)	
Paso 1	Paso 2	Paso 3	Paso 4	Paso 5	Paso 6	Paso 7	Paso 8	Paso 9	Paso 10	Paso 11	Paso 12

Figura 4.3. Estados de la pila asociados a la llamada al método **factorial (5)**.

				fib(1)	fib(0)							
			fib(2)	fib(2)	fib(2)	fib(2)	fib(1)			fib(1)	fib(0)	
		fib(3)	fib(3)	fib(3)	fib(3)	fib(3)	fib(3)	fib(3)	fib(2)	fib(2)	fib(2)	fib(2)
fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)	fib(4)
Paso 1	Paso 2	Paso 3	Paso 4	Paso 5	Paso 6	Paso 7	Paso 8	Paso 9	Paso 10	Paso 11	Paso 12	Paso 13

Figura 4.4. Estados de la pila asociados a la llamada al método **fib (4)**.

4.4. Algoritmos de *backtracking*

El *backtracking* o “vuelta atrás” es una técnica algorítmica de resolución general de problemas mediante una búsqueda sistemática de soluciones. Se descompone la tarea a realizar en tareas parciales de tanteo de posibles soluciones, y se prueba sistemáticamente cada una de éstas que a su vez se descompondrá en subtareas que habrá que probar, y así sucesivamente. Cuando al elegir una determinada tarea se comprueba que no conduce a la solución, se debe *volver atrás*, al punto donde se decidió elegir esta tarea, y probar con otra. Se deben guardar los “puntos de elección” para poder volver a ellos cuando por una determinada rama no se puede alcanzar la solución. Esto se suele hacer utilizando la recursividad, que permite abordar este tipo de problemas de forma natural.

La descripción de este proceso de búsqueda de soluciones se puede representar con un árbol de búsqueda donde se muestra cómo cada tarea se divide en subtareas que a su vez se vuelven a dividir, de forma que el número de caminos crece exponencialmente. Cada nodo en el árbol se corresponde con una llamada recursiva.

En los Ejercicios 4.7. y 4.8. del Apartado 4.5. se plantean dos problemas clásicos que se resuelven utilizando la técnica de *backtracking*: el salto del caballo y el problema de las ocho reinas.

4.5. Ejercicios resueltos

Ejercicio 4.1. Invertir un número

Enunciado Realizar un método recursivo que imprima en pantalla las cifras de un número dado, en orden inverso.

Solución Básicamente, la operación de invertir el número consiste en lo siguiente:

1. Si el número es menor que 10 lo escribimos y terminamos (caso base o condición de salida).
2. Si no,
 - 2.1. se escribe la última cifra, que se obtiene con el resto de dividir el número por 10,
 - 2.2. se vuelve a hacer la misma operación "invertir" sobre el número que queda al quitarle la última cifra.

Veamos un ejemplo: invertir el número 538:

1. Como no es menor que 10, seguimos el paso 2.1: Se escribe (538 modulo 10) $\rightarrow$ 8.
2. Se repite el proceso con (538 div 10), es decir con el 53.
 - Se escribe (53 modulo 10) $\rightarrow$ 3.
 - Se repite el proceso con (53 div 10), es decir con el 5.
 - Como 5 es menor que 10 se escribe y terminamos.

El método codificado en Java sería:

```
static void invertir(int n){
    if (n<10)
        System.out.println(n);
        // caso base
    else {
        System.out.println(n % 10);
        invertir (n /10);
        // caso recursivo
    }
}
```

Ejercicio 4.2. Máximo común divisor

Enunciado Realizar un método recursivo que calcule el máximo común divisor de dos números enteros positivos.

Solución Utilizaremos el algoritmo de Euclides, que consiste en lo siguiente: sean los dos números M y N, con $M > N$. Se divide M entre N y se obtiene el resto R1 y el cociente C1. Si el resto es 0, el

divisor es la solución buscada. Si no, se divide ahora N entre el resto $R1$, obteniendo el resto $R2$ y el cociente $C2$. Se sigue así hasta que se obtenga un resto nulo, en cuyo caso la solución será el último divisor.

Cociente		C1	C2	C3	...
Dividendo y divisor	M	N	R1	R2	...
Restos	R1	R2	R3		...

Se puede resolver este problema con técnicas iterativas y con técnicas recursivas. En la solución iterativa, en cada iteración se hace la división entera y se reasignan las variables de modo que el nuevo dividendo toma el valor del anterior divisor y el divisor el del resto; la condición de salida del bucle es que el resto de la división sea 0. En la solución recursiva, se hace lo mismo pero con llamadas recursivas donde la condición de salida o caso base es la situación donde el resto es 0, en cuyo caso se devuelve como resultado el valor del divisor. En cualquier otro caso se entra en el caso recursivo donde se hacen la llamadas variando los parámetros: el parámetro que corresponde al nuevo dividendo es el divisor actual, y el que corresponde al nuevo divisor es el resto de la división.

La codificación en Java será la siguiente:

```
static int mcd(int m,int n){
// m es mayor o igual que n
    if (m % n == 0)
        return n;
    // caso base
    else
        return mcd ( n , (m%n) );
    // caso recursivo
}
```

En la Figura 4.5. podemos ver un seguimiento de la ejecución de este método para la llamada `mcd (21,15)`.

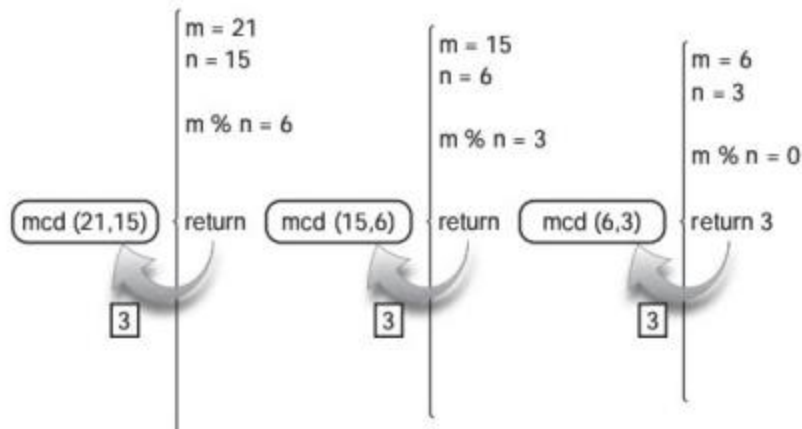


Figura 4.5. Seguimiento de la llamada `mcd (21, 15)`.

Ejercicio 4.3. Escribir un número en binario

Enunciado Realizar un método recursivo que reciba un número entero y lo escriba en binario.

Solución Para pasar un número decimal a binario se hace lo siguiente:

- Se divide el número por 2 y se guarda el resto. Se divide el cociente obtenido por 2 y se guarda el resto y así sucesivamente hasta obtener cociente 1, en cuyo caso se imprime 1.
- Se imprimen los restos obtenidos anteriormente en orden inverso.

Una solución iterativa consistiría en utilizar un bucle donde en cada iteración se vaya dividiendo el número por 2 y se obtenga el correspondiente resto. La condición de terminación del bucle se daría cuando se alcanzara el cociente 1, y entonces se mostrarían los restos en orden inverso al de su obtención.

Una solución recursiva a este problema parece más elegante: simplemente se trata de colocar la sentencia de escritura del resto obtenido **detrás** de la llamada recursiva: de este modo garantizamos que a medida que se sale de un método, se escribe a continuación el resto correspondiente, como se puede ver en la Figura 4.6.

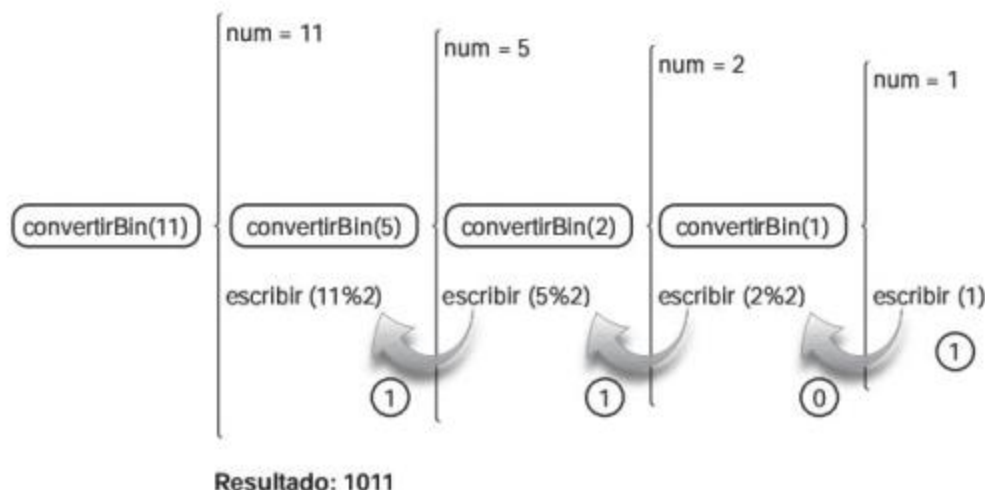


Figura 4.6. Segimiento de la llamada `converttirBin(11)`.

El método codificado en Java sería:

```
static void converttirBin(int num) {  
  
    if (num == 1)  
        System.out.print(num);  
    // caso base
```

```

    else{
        // caso recursivo
        convertirBin (num / 2);
        System.out.print(num % 2);
    }
}

```

En la Figura 4.6. se puede ver el seguimiento de la ejecución del método para la llamada **convertirBin (11)**.

Ejercicio 4.4. Multiplicación rusa

Enunciado Realizar una función recursiva que reciba dos números enteros y realice su multiplicación "a la rusa". El método consiste en formar dos columnas, una por cada operando. Las columnas se forman aplicando repetidamente los pasos siguientes:

- Dividir por 2 el multiplicando, y poner el cociente debajo.
- Duplicar el multiplicador y poner el resultado debajo.
- Cuando la columna del multiplicando llega al valor 1 paramos.
- Se suman los valores del multiplicador que se correspondan con valores impares de la columna de multiplicandos. El resultado es el producto.

Ejemplo

34	25	
17	50	
8	100	
4	200	
2	400	
1	800	850

Solución El caso de parada ocurrirá cuando el multiplicador sea 1. En otro caso, se hará una llamada recursiva al método con los parámetros siguientes: el multiplicando se dividirá por 2 (división entera) y el multiplicador se multiplicará por 2. Pero dentro de la parte recursiva hay que distinguir los casos de multiplicando par o impar. Cuando sea impar, hay que devolver la suma del multiplicando y el valor que devuelva la llamada recursiva. Cuando sea par, se devolverá simplemente el valor que devuelva la llamada recursiva, es decir, simplemente se arrastrará el valor que reciba. A continuación, se muestra el código en Java, y en la Figura 4.7. se puede ver el seguimiento detallado de la ejecución del método cuando la primera llamada es rusa (34,25).

```

static int rusa(int x, int y){
    if (x>1)
        // caso recursivo
        if (x % 2 != 0)    //x es impar
            return (y + rusa (x / 2, y * 2));
        else // x es par
            return (rusa (x / 2, y * 2));
    else
        // caso de parada: x es 1
        return y;
}

```

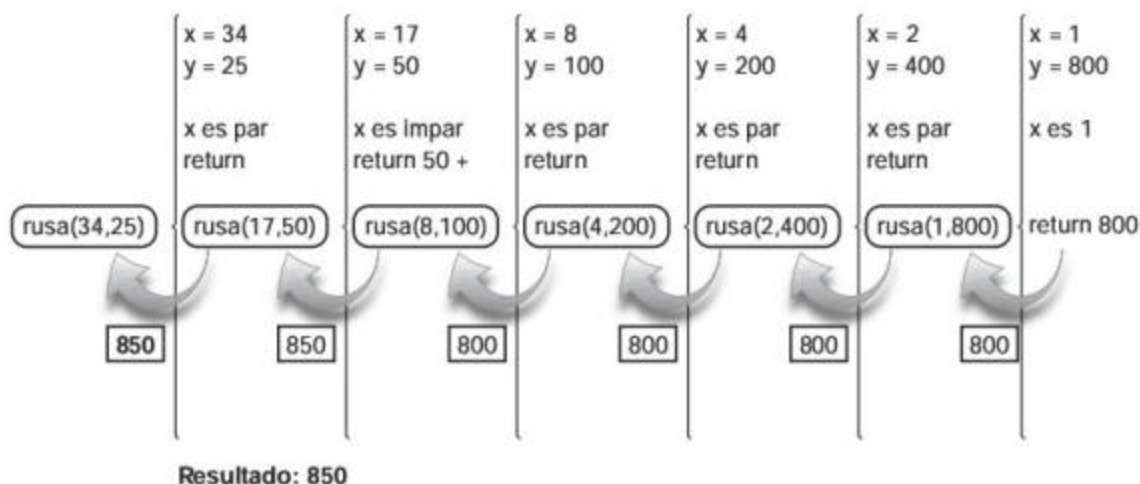


Figura 4.7. Seguimiento de la llamada `rusa(34, 25)`.

Ejercicio 4.5. Suma de los elementos de un array

Enunciado Escribir un método recursivo que calcule la suma de los elementos de un array.

Solución Evidentemente este problema tiene una clara y eficiente solución iterativa. Pero se trata de encontrar un planteamiento recursivo a la solución de este sencillo problema aunque, en este caso, no sea tan eficiente.

Supongamos que el índice de la última posición del array es u . El caso de parada ocurre cuando el array sólo tiene un elemento, es decir cuando el índice $u=0$, siendo el valor de la suma `lista[0]` (array de un elemento).

Si u no es 0, debemos añadir el valor del último elemento `lista[u]` a la suma del subarray con índices $0..u-1$. Dicho de otra forma, la suma de los n elementos de un array es:

- caso de $n = 1$: el valor del elemento.
- caso general ($n > 1$): el valor del último elemento más la suma de los elementos del subarray restante ($n - 1$ elementos).


```
static int sumarArray (int [] v, int u){  
    //v es el array y u es el valor del índice de la última posición  
    if (u==0)  
        // caso base  
        return (v[0]);  
    else  
        // caso recursivo  
        return (v[u] + sumarArray (v,u-1));  
}
```

Ejercicio 4.6. Problema de “las Torres de Hanoi”

Enunciado Este famoso juego nos permitirá aplicar la recursividad de una forma muy elegante y natural. Su solución es bastante complicada sin utilizar recursividad.

El juego dispone de tres postes: 1, 2 y 3 (véase Figura 4.8.). En el poste 1 se encuentran n discos de tamaño decreciente. El objetivo es mover todos los discos desde el poste 1 al poste 3 utilizando el poste 2 como almacén auxiliar, con las siguientes condiciones:

- Sólo se podrá mover un disco cada vez.
- Nunca podrá haber un disco encima de otro de menor tamaño.

Nuestro problema consiste en encontrar un algoritmo que lleve a cabo esta tarea, y codificarlo adecuadamente en Java.

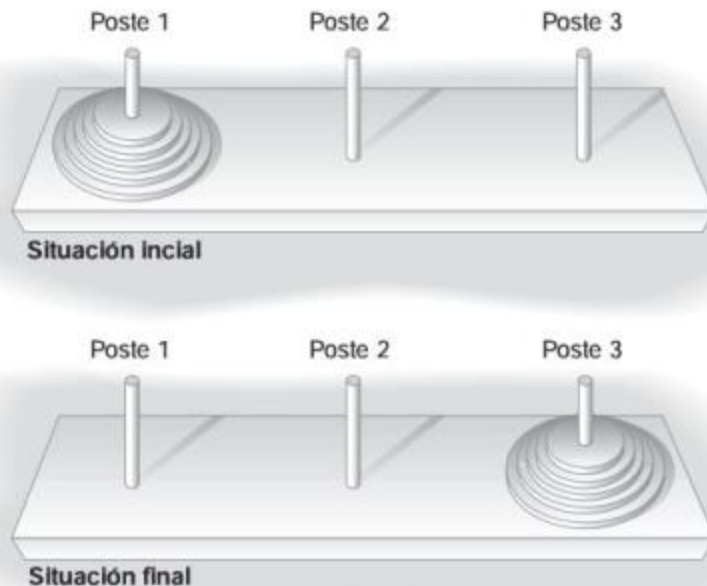


Figura 4.8. Juego de las Torres de Hanoi.

Solución Vamos a plantear la solución de forma recursiva:

- Caso base. Si sólo hay 1 disco, el problema es trivial: se pasa directamente el disco del poste 1 al 3.
- Caso general. Si hay más de un disco el problema se resuelve de la siguiente forma:
 1. Mover los $n - 1$ discos superiores de 1 a 2 utilizando el poste 3 como almacén auxiliar.
 2. Mover el disco n del poste 1 al 3.
 3. Mover los $n - 1$ discos del poste 2 al 3 utilizando el poste 1 como almacén auxiliar.

Como vemos, el problema de mover n discos se ha transformado en dos problemas más sencillos consistentes en mover $n - 1$ discos.

A su vez cada problema de mover $n - 1$ discos se transforma en dos problemas de mover $n - 2$ discos. Así, el problema de mover $n - 1$ discos de 1 a 2 utilizando 3 como auxiliar sería:

1. Mover los $n - 2$ discos superiores de 1 a 3 utilizando 2 como auxiliar.
2. Mover el disco $n - 1$ de 1 a 2.
3. Mover los $n - 2$ discos de 3 a 2.

De este modo vamos progresando, reduciendo en cada paso la complejidad del problema, hasta que el mismo consista sólo en mover un disco. La condición de terminación se cumple cuando el número de discos sea 1.

Se puede calcular matemáticamente el número de movimientos necesarios para mover n discos: $2^n - 1$. Es decir, el problema es de complejidad exponencial. La solución a este problema es de naturaleza recursiva, y no existe una solución iterativa directa. Es uno de los ejemplos más claros de problemas donde se debe utilizar un algoritmo recursivo, ya que una solución puramente iterativa sería extraordinariamente complicada.

Según lo dicho, la codificación en Java sería sencilla, teniendo cuidado de utilizar correctamente las variables que van a identificar los correspondientes postes. En esta implementación, estas variables, de tipo `int`, serán las siguientes:

- a para el poste origen del movimiento.
- c para el poste destino.
- b para el poste que se utilizará como almacén auxiliar.

La cabecera del método recursivo que resuelve el problema será:

```
static void hanoi (int n, int a, int c, int b);
```

donde n es el número de discos a mover. El método es el siguiente:

```
static void hanoi (int n, int a, int c, int b){
    // mover n discos de a a c pasando por b
    if (n==1)
        // caso base
        System.out.println ("Mover disco "+n+" desde "+a+" hasta "+ c);
    else{
```

```

    hanoi(n-1,a,b,c); // mover n-1 discos de a a b utilizando c
    System.out.println ("Mover disco "+n+" desde "+a+" hasta " + c);
    hanoi(n-1,b,c,a); // mover n-1 discos de b a c utilizando a
}
}

```

Veamos un ejemplo donde queremos mover 3 discos del poste 1 al poste 3 utilizando el poste 2 como almacén auxiliar:

```
hanoi(3,1,3,2);
```

Un seguimiento detallado de lo que ocurre cuando se produce esta llamada al método se puede ver en la Figura 4.9., donde el número encerrado en un círculo indica el orden en que se imprimirán los mensajes con las instrucciones para mover los discos.

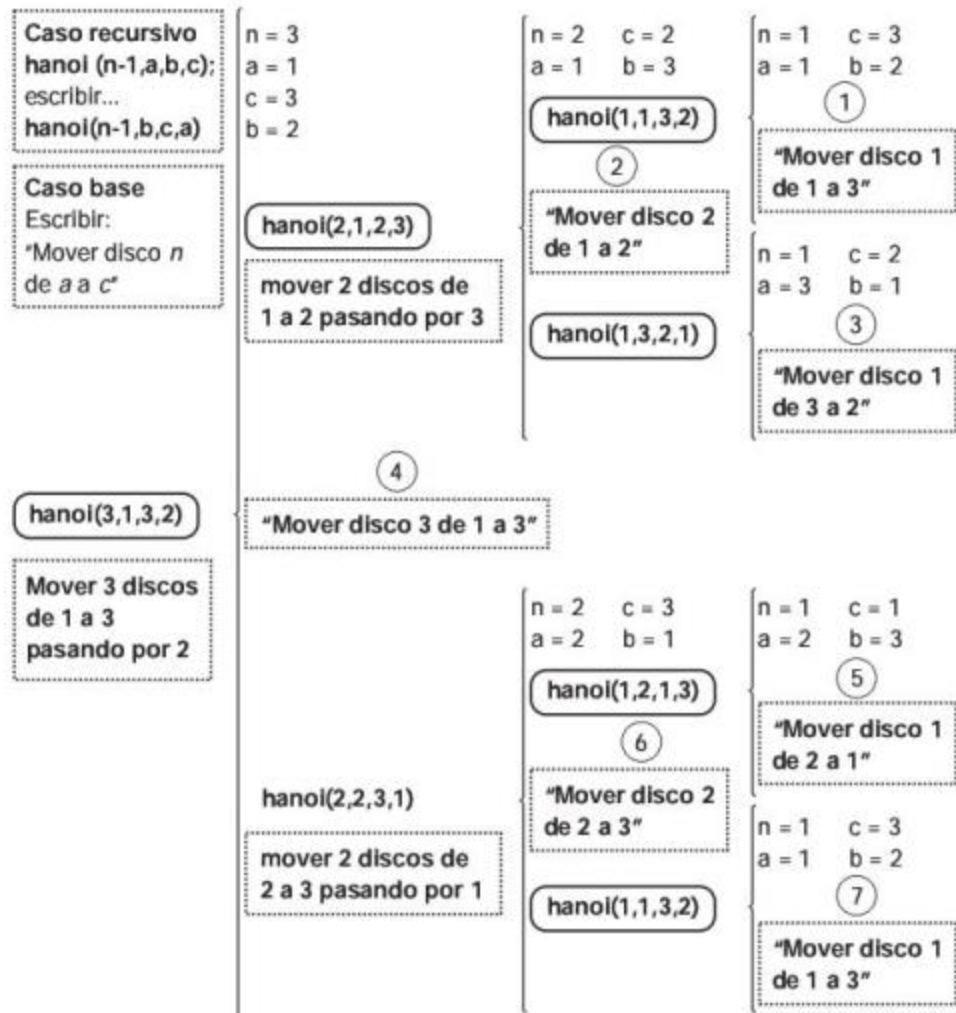


Figura 4.9. Seguimiento de la llamada `hanoi(3,1,3,2)`.

Ejercicio 4.7. Problema del “salto del caballo”

Enunciado Disponemos de un tablero de ajedrez de $N * N$ casillas. Situamos inicialmente un caballo en la casilla de coordenadas (X_0, Y_0) . El problema consiste en encontrar, si existe, un camino que permita al caballo pasar una sola vez por todas y cada una de las casillas del tablero.

Solución Se seguirá la técnica general de *backtracking*, haciendo una búsqueda sistemática de las posibles soluciones. En los algoritmos de vuelta atrás se prueban nuevos posibles caminos que conduzcan a la solución. En el caso de que un ensayo no nos lleve a la solución, se da marcha atrás, por tanto, se borra la anotación que se realizó en el ensayo y se vuelve a hacer otro ensayo, siempre que sea posible. En el caso del caballo, se puede probar con los 8 movimientos posibles desde una posición dada.

Básicamente, el algoritmo descrito en pseudocódigo es el siguiente:

```

Algoritmo Caballo (n, x, y)
// n: numero de movimiento (la primera vez será el 1)
// x, y: coordenadas de la celda que corresponde a ese movimiento
// la primera vez serán las celdas de partida
Inicio
  Anotar movimiento n en casilla (x,y)
  Si es el movimiento N*N habremos alcanzado la solución
  Para cada posible movimiento y mientras no alcancemos la solución
    Calcular las nuevas coordenadas (nx,ny)
    Si están en el tablero entonces
      Si no se ha pasado por esa casilla entonces
        Nuevo ensayo: Caballo (n+1,nx,ny)
      Fin-si
    Fin-si
  Fin-para
  Si no se alcanzó la solución entonces
    Borrar anotación anterior
  Fin-si
  Devolver solución
Fin

```

Para poder realizar el programa en Java, deberemos decidir la forma de representar tanto el tablero de ajedrez como los posibles movimientos del caballo. El tablero se representará mediante un array de enteros, para poder guardar el número de movimiento en el que pasa el caballo por esa casilla.

Es decir, una casilla $T[X][Y]$ contendrá:

- 0, si por la casilla (X,Y) no ha pasado el caballo,
- i, si por la casilla (X,Y) pasó el caballo en el movimiento i.

Para poder seleccionar un nuevo movimiento, hay que tener en cuenta los 8 posibles movimientos que puede realizar un caballo. En la Figura 4.10, se ven, numerados, estos movimientos partiendo de la posición marcada por la figura del caballo:

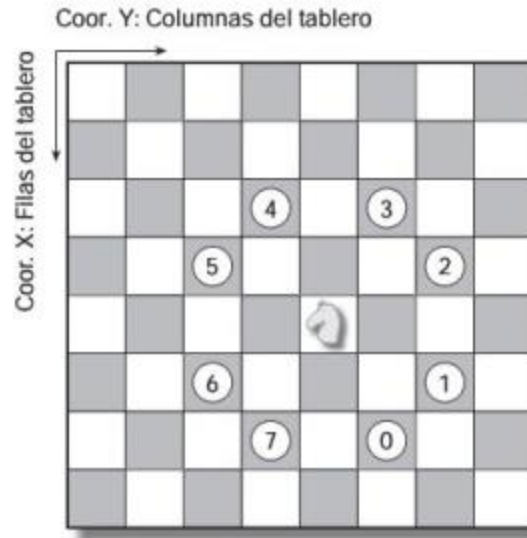


Figura 4.10. Posibles movimientos del caballo.

Estos movimientos se guardarán en una matriz de 2×8 de la siguiente forma:

- En la **fila 0** guardaremos las coordenadas X de los 8 posibles desplazamientos relativos, a partir de la posición actual del caballo: (2, 1, -1, -2, -2, -1, 1, 2). Las coordenadas X indicarán las filas de la matriz tablero.
- En la **fila 1**, guardaremos las coordenadas Y: (1, 2, 2, 1, -1, -2, -2, -1). Las coordenadas Y indicarán las columnas de la matriz tablero.

Ejemplo

El movimiento 0 tendrá un desplazamiento de +2 en las filas y de +1 en las columnas. El movimiento 2 tendrá un desplazamiento de -1 en filas y +2 en columnas.

	mov 0	mov 1	mov 2	mov 3	mov 4	mov 5	mov 6	mov 7
Desplaz de X (filas)	2	1	-1	-2	-2	-1	1	2
Desplaz de Y (columnas)	1	2	2	1	-1	-2	-2	-1

Veamos cómo se especifica en Java la condición de que el nuevo movimiento esté en el tablero:

```
(x < N && x >= 0) && (y < N && y >= 0)
```

hagamos lo mismo con la condición de que el caballo no haya pasado anteriormente por esa casilla:

```
(t[x][y] == 0)
```

y con la condición de que no se haya completado el tablero:

$i < N*N$, siendo i el número de movimiento, y $N*N$ el total de casillas.

Ahora tenemos todos los medios para poder programar en Java el algoritmo indicado anteriormente. La cabecera del método será:

```
boolean caballo (int i, int x, int y, int [][] tablero, int [][] despl)
```

siendo i el número del salto, x e y las coordenadas actuales del caballo, *tablero* la matriz que representa al tablero de ajedrez, y *despl* la matriz que guarda los 8 posibles saltos del caballo en forma de desplazamientos relativos. El método devolverá un valor booleano: true si ha encontrado la solución y false si no la ha encontrado.

A continuación se muestra el método codificado en Java junto con métodos auxiliares para visualizar el tablero y para inicializarlo, y un método *main* para ver un ejemplo de llamada inicial al método recursivo.

```
public class Caballo {
    final static int N=8;
    // lado del tablero
    final static int N_SALTOS=8;
    // numero de saltos posibles

    static boolean caballo (int i, int x, int y, int [][] tablero,
                           int [][] despl){

        // i:num de salto. x,y:coord actuales.
        //Devuelve un booleano: solución
        int nX,nY;
        // nuevas coordenadas x e y
        int k;
        // contador de posibles movimientos de este salto
        boolean solucion=false;
        // para controlar si se alcanza la solución
        tablero[x][y] = i;
        // anotamos movimiento i
        if (i==N*N) solucion = true; // hemos alcanzado la solucion
        for (k=0 ;k < N_SALTOS && !solucion; k++){
            // para cada posible movimiento
            // calculamos las coordenadas siguientes
```

```

        nX = x + despl[0][k];
        nY = y + despl[1][k];
        if (nX < N && nX >= 0 && nY < N && nY >= 0)
            // si las nuevas coord no se salen del tablero
            if (tablero[nX][nY] == 0)
                //si casilla no visitada
                //se produce un nuevo ensayo
                solucion = caballo (i+1, nX,nY,tablero,despl);
    } // fin del for
    // puede salir del for por dos motivos: por agotar los
    // movimientos o por encontrar la solución
    if (!solucion) tablero[x][y] = 0; // borrar anotación
    return solucion;
} // fin del metodo caballo

static void inicializarTablero (int [][] t){
    for (int i=0;i<N;i++)
        for (int j=0;j< N; j++)
            t[i][j] = 0;
}

static void visualizarTablero (int [][] t){
    for (int i=0;i<N;i++){
        for (int j=0;j< N; j++)
            System.out.print(t[i][j]+" ");
        System.out.println();
    }
}

public static void main(String args[]) throws IOException{
    int [][] tablero = new int [N][N];
    final int [][] desplazamientos = {{2,1,-1,-2,-2,-1,1,2},
                                      {1,2,2,1,-1,-2,-2,-1}};
    boolean solucion; // indicar si se ha alcanzado la solución
    inicializarTablero (tablero);
    // el caballo parte de la casilla (0,0) en el salto num 1
    solucion = caballo (1,0,0,tablero,desplazamientos);
    // Movimiento 1, coord actuales 0,0,
    if (solucion)
        visualizarTablero(tablero);
    else
        System.out.println("NO hay solución");
}
} // fin de la clase

```

A continuación, se muestra el resultado del programa anterior, es decir, cuando se parte de la casilla (0,0), esquina superior izquierda.

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

Ejercicio 4.8. Problema de las “ocho reinas”

Enunciado Este conocido problema consiste en colocar ocho reinas en un tablero de ajedrez, de forma que ninguna reina pueda amenazar a otra. Como se sabe, la reina del ajedrez puede comerse a una ficha contraria que esté accesible bien sea en vertical, en horizontal o en diagonal, como puede verse en la Figura 4.11.

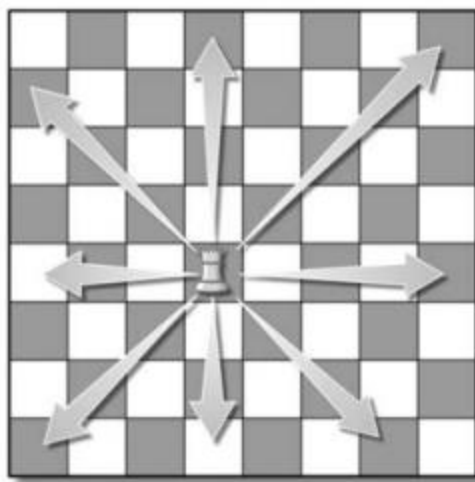


Figura 4.11. Casillas amenazadas.

Solución Siguiendo un esquema de “vuelta atrás”, se intentará colocar una reina en cada fila (puesto que no puede haber dos reinas en la misma fila) de la siguiente forma: se colocará la primera reina en la fila 1, columna 1. Se intentará colocar la reina 2 en la fila 2, columna 1, pero es una posición amenazada por la reina 1; se intentará colocar en la columna 2 pero también está amenazada; entonces la colocaremos en la columna 3 donde no hay problema. Después seguiremos con la reina 3, probando en las columnas 1, 2, 3, etc., hasta que se encuentre una columna válida, que en este caso es la 5. Seguimos aplicando este método a las sucesivas reinas, y en algún momen-

to veremos que no hay ninguna columna válida. Entonces tendremos que "dar marcha atrás" y recolocar la reina anterior.

El planteamiento anterior se puede expresar en pseudocódigo del siguiente modo:

```
colocarReina en fila f
    columna ← 0; solucion ← falso
    mientras (columna < 8) y (no lleguemos a la solucion) hacer
        si la posicion (f,columna) no esta amenazada entonces
            marcar posicion(f,columna)
            si es la ultima fila entonces
                solucion ← verdadero
            si no
                colocarReina en fila f+1
                si no se llego a la solucion entonces
                    borrar la marca en posicion (f,columna)
                fin-si
            fin-si
        fin-si
        columna ++
    fin-mientras
fin-metodo
```

El problema completo codificado en Java se muestra a continuación. Como puede verse, se ha incluido un método booleano `amenazada()` que indica si una determinada posición del tablero está amenazada por alguna reina que se encuentre ya en el tablero.

```
class OchoReinas{
    final static int M = 8;

    // para inicializar el tablero
    public static void inicializar(int t[][]){
        for (int i=0;i<M ;i++){
            for (int j=0;j<M ;j++ )
                t[i][j]=0;
        }

        // para visualizar el tablero
        public static void visualizar(int t[][]){
            for (int i=0;i<M ;i++){
                for (int j=0;j<M ;j++ )
                    System.out.print(t[i][j]+" ");
                System.out.println();
            }
        }
    }
}
```

```

// para determinar si una posición (f,c) está amenazada
public static boolean amenazada(int t[][],int f,int c){
    int nf=0,nc=0;

    // ver si hay otra en la misma fila
    boolean otraReina=false;
    for (nc=0;nc<M ;nc++ )
        if (t[f][nc]!=0)
            otraReina=true;
    if (!otraReina)

    // miramos la columna
    for (nf=0;nf<M ;nf++ )
        if (t[nf][c]!=0)
            otraReina=true;

    if (!otraReina)

    // miramos la diagonal directa hacia arriba
    for (nf=f,nc=c;nf>=0 && nc>=0 ;nf--,nc- )
        if (t[nf][nc]!=0)
            otraReina=true;

    // miramos la diagonal directa hacia abajo
    for (nf=f,nc=c;nf<M && nc<M ;nf++,nc++ )
        if (t[nf][nc]!=0)
            otraReina=true;

    // miramos la diagonal inversa hacia arriba
    for (nf=f,nc=c;nf>=0 && nc<M ;nf--,nc++ )
        if (t[nf][nc]!=0)
            otraReina=true;

    // miramos la diagonal inversa hacia abajo
    for (nf=f,nc=c;nf<M && nc>=0 ;nf++,nc- )
        if (t[nf][nc]!=0)
            otraReina=true;
    return otraReina;
}

public static boolean colocarReina(int [][] t, int f){
    // colocar la reina en la fila f
    // si se coloca con exito la reina devuelve true

```

```

boolean solucion=false;
int c=0;
while (c<M && !solucion){
    //intentamos colocar la reina de la fila f en la columna c
    if (!amenazada(t,f,c)){
        t[f][c]=f+1;
        // marcamos la casilla en la fila 0, la reina 1...
        if (f==M-1) // es la ultima fila
            solucion=true;
        else{
            solucion=colocarReina(t,f+1);
            if (!solucion)
                // al volver no ha encontrado la solucion
                t[f][c]=0; // borramos la anotacion
        } // fin del else
    } // fin del if !amenazada
    c++;
} // fin del while
return solucion;
}

public static void main(String[] args) {
    int t[][] = new int [M][M];
    inicializar(t);
    //visualizar(t);
    colocarReina(t,0); //colocar reina en la fila 0
    visualizar(t);
}
}

```

Existen 92 soluciones al problema de las ocho reinas. La solución que da este programa es la siguiente:

```

1 0 0 0 0 0 0 0
0 0 0 0 2 0 0 0
0 0 0 0 0 0 0 3
0 0 0 0 0 4 0 0
0 0 5 0 0 0 0 0
0 0 0 0 0 0 6 0
0 7 0 0 0 0 0 0
0 0 0 8 0 0 0 0

```

Correspondiendo los números del 1 al 8 a las reinas.

Ejercicio 4.9. Cuadrado mágico. (Ejercicio de examen)

Enunciado Se denomina "**cuadrado mágico**" a una colocación de n^2 números en filas y columnas formando un cuadrado, con n números en cada fila y en cada columna, de tal forma que la suma de los elementos de cada fila, cada columna y las dos diagonales del cuadrado dé el mismo valor. Habitualmente los números utilizados para colocar en un cuadrado de lado n son aquellos comprendidos entre el 1 y el n^2 .

En la fachada de la pasión de la catedral de la Sagrada Familia en Barcelona (diseñada por el arquitecto Josep María Subirachs) puede encontrarse el denominado "**criptograma de Subirachs**". Como puede observarse en las figuras, se trata de un cuadrado mágico ya que sus filas, columnas y diagonales suman 33, número correspondiente a la edad a la que supuestamente fue crucificado Jesús de Nazaret.

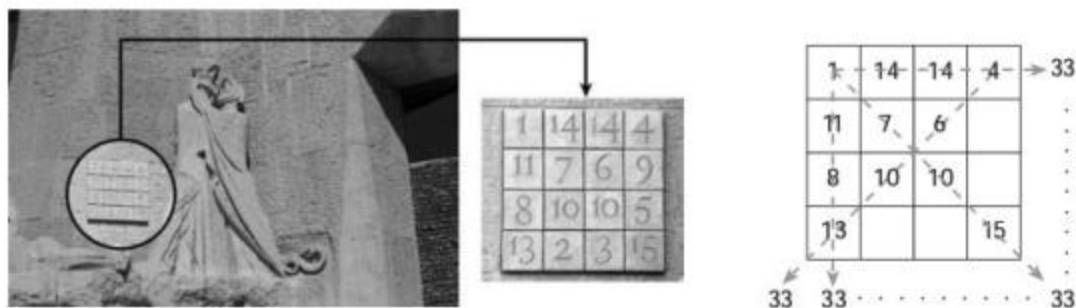


Figura 4.12. Criptograma de Subirachs.

Se pide:

1. Implementar el método **iterativo**: `public boolean esCMagico(int[][] cmagico)`. Este método utiliza un array bidimensional que almacena la matriz de números correspondiente, a partir del mismo comprueba que las filas, columnas y diagonales de la misma, verifican la condición de cuadrado mágico.
2. Implementar el método: `public boolean esCMagicoRekursivo(int[][] cript)`. Este método utilizará tres métodos que de forma recursiva comprobarán si filas, columnas y diagonales cumplen o no la condición.
3. Implementar el método que determine de forma recursiva si las **filas** cumplen la propiedad de cuadrado mágico:

```
private static boolean comprobarFilas(int[][] cmagico)
```

4. Implementar el método que determine de forma recursiva si las **columnas** cumplen la propiedad deseada:

```
private static boolean comprobarColumnas(int[][] cmagico)
```


5. Implementar el método que determine de forma recursiva si ambas **diagonales** cumplen la propiedad deseada:

```
private static boolean comprobarDiagonales(int[][] cmagico)
```

Solución

El Apartado 1 implementa el método booleano `esCMagico()` de forma iterativa, comprobando si las filas, columnas y diagonales cumplen la condición de cuadrado mágico. Se supondrá que existe un atributo de la clase que llamaremos `condicionCmagico` que contiene el valor correspondiente al cuadrado mágico.

```
public boolean esCMagico(int[][] cmagico){
    //método que recorre iterativamente una matriz comprobando la
    //condicion de cuadrado magico
    boolean cumple=true;
    int aux=0, i=0, j=0;
    int auxP=0, auxI=0;
    //se verifican las filas, y las diagonales
    while(i<cmagico.length && cumple){
        while(j<cmagico[i].length && cumple){
            aux = aux+cmagico[i][j];
            if(i==j) //diagonal principal
                auxP = auxP+cmagico[i][j];
            if(i+j==cmagico.length-1)//diagonal inversa
                auxI = auxI+cmagico[i][j];
            if (aux>condicionCmagico||auxP>condicionCmagico||auxI>condicionCmagico)
                cumple = false;
            j++;
        }
        if (aux!=condicionCmagico)
            cumple = false;
        i++;
        j=0;
        aux=0;
    }
    if (auxP!=condicionCmagico || auxI!=condicionCmagico)
        cumple = false;
    if (cumple){
        i=0;
        j=0;
        //se verifican las columnas
        while(j<cmagico[i].length && cumple){
            while(i<cmagico.length && cumple){
                aux = aux+cmagico[i][j];
                if (aux>condicionCmagico)
                    cumple = false;
            }
            j++;
            i=0;
        }
    }
}
```

```

        i++;
    }
    if (aux!=condicionCmagico)
        cumple = false;
    j++;
    i=0;
    aux=0;
}
}
return cumple;
} //fin esCmagicoIterativo

```

Apartado 2. Simplemente se harán las llamadas a los métodos recursivos que se desarrollan más adelante.

```

public boolean esCMagicoRecursivo(int[][] cmagico){
    return comprobarFilas(cmagico) && comprobarColumnas(cmagico)&&
        comprobarDiagonales(cmagico);
} //fin esCmagicoRecursivo

```

Apartado 3. Implementar el método que determine de forma recursiva si las **filas** cumplen la propiedad de cuadrado mágico:

```

private static boolean comprobarFilas(int[][] cmagico){
    int i=0;
    int filas = ORDEN-1;
    boolean cumple=true;
    while (i<filas && cumple){
        cumple = filaK(cmagico,i,0,0);
        i++;
    }
    return cumple;
}

private static boolean filaK(int[][] cmag, int fila, int columna, int
calculado){
    //verifica una fila recursivamente
    int aux=0;
    if (columna==ORDEN){
        return false;
    }else{
        aux = calculado+cmag[fila][columna];
        if(aux<condicionCmagico){
            return filaK(cmag,fila,++columna,aux);
        }else if(aux==condicionCmagico){
            return true;
        }
    }
}

```

```
        }else{
            return false;
        }
    }
}
//fin filaK
```

Apartado 4. Implementar el método que determine de forma recursiva si las **columnas** cumplen la propiedad deseada:

```
private static boolean comprobarColumnas(int[][] cmagico){
    int j=0;
    int columnas = ORDEN-1;
    boolean cumple=true;
    while (j<columnas && cumple){
        cumple = columnaK(cmagico,0,j,0);
        j++;
    }
    return cumple;
}
```

```
private boolean columnaK(int[][] cmag, int fila, int columna, int calculado){  
    //verifica una columna recursivamente  
    int aux=0;  
    if (fila==ORDEN){  
        return false;  
    }else{  
        aux = calculado+cmag[fila][columna];  
        if(aux<condicionCmagico){  
            return columnaK(cmag, ++fila, columna, aux);  
        }else if(aux==condicionCmagico){  
            return true;  
        }else{  
            return false;  
        }  
    }  
}  
//fin columnaK
```

Apartado 5. Implementar el método que determine de forma recursiva si ambas **diagonales** cumplen la propiedad deseada:

```
private static boolean comprobarDiagonales(int[][] cmagico){
return diagonalPrincipal(cmagico,0,0,0) && diagonalInversa(cmagico,0,3,0);
}
```

```
private static boolean diagonalPrincipal(int[][]cmag, int fila, int columna, int
```

```

calculado){
    //verifica una fila recursivamente
    int aux=0;
    if (columna==ORDEN){
        return false;
    }else{
        if(fila==columna)
            aux = calculado+ cmag [fila][columna];
        if(aux<condicionCmagico){
            //sólo se comprueban las casillas fila==columna
            return diagonalPrincipal (cmag,++fila,++columna,aux);
        }else if(aux==condicionCmagico){
            return true;
        }else{
            return false;
        }
    }
}
//fin diagonalPrincipal

private static boolean diagonalInversa(int[][]cmag, int fila, int columna,
int calculado){
    //verifica una fila recursivamente
    int aux=0;
    if (columna==ORDEN){
        return false;
    }else{
        if(fila+columna==ORDEN-1)
            aux = calculado+ cmag [fila][columna];
        if(aux<condicionCmagico){
            //sólo se comprueban las casillas fila+columna==ORDEN-1
            return diagonalInversa(cmag,++fila,-columna,aux);
        }else if(aux==condicionCmagico){
            return true;
        }else{
            return false;
        }
    }
}
}
//fin diagonalInversa

```

Ejercicio 4.10. Fiestas sociales. (Ejercicio de examen)

Enunciado Considérese una fiesta a la que asisten n personas: $p_1, p_2, p_3, \dots, p_n$. Puede ocurrir que la persona i conozca a la persona j , que la persona j conozca a la persona i , que la persona i no conozca a la persona j o que la persona j no conozca a la persona i .

Se quiere implementar una clase `Fiesta` que contenga las estructuras de datos necesarias para representar éstas relaciones entre las k personas asistentes a la reunión, así como una serie de métodos que nos darán información sobre las relaciones entre estas personas. Además, consideraremos que la persona i es famosa si es conocida por todas las demás, pero no conoce a nadie.

Se pide:

1. Definición de los atributos de la clase que permita representar los datos de forma adecuada. Hay que tener en cuenta que sólo nos interesa si una persona conoce a otra, no las características propias de cada persona. Por tanto, sólo hay que almacenar las relaciones entre todos los posibles pares de personas de la fiesta. Se aconseja utilizar estructuras sencillas, aunque sean poco eficientes.
2. Implementar el método boolean `esFamoso(int i){..}` que nos dice si la persona i es famosa.
3. Implementar el método boolean `esConocidoYConoce(int i){...}` que determine de forma recursiva si la persona i es conocida por todos y a la vez conoce a todos. No es necesario que este método sea recursivo, sino que pueden implementarse métodos recursivos auxiliares que son llamados desde este método.

Solución

Apartado 1. Definición de los atributos de la clase que permita representar los datos de forma adecuada.

Solamente se necesitará un atributo que será un array bidimensional de booleanos, de forma que si la persona i conoce a la persona j , la posición de la fila i , columna j será true.

En el constructor de la clase se inicializará la matriz.

```
private boolean [][] conoce;
```

Apartado 2. Implementar el método boolean `esFamoso(int i){..}` que nos dice si la persona i es famosa.

```
public boolean esFamoso(int i){
    int numPersonas = conoce.length;
    boolean famoso = true;
    int col=0;
    conoce[i][i]=false;
    // el valor de (i,i) es irrelevante, y simplifica bastante
    // el código si tiene el valor false.

    // comprobar que no conoce a nadie
    while (famoso && col < numPersonas){
        if (conoce[i][col]) famoso=false;
        else col ++;
    }
    // ver por que razon ha salido
```

```

    // dejamos conoce[i][i] como estaba
    conoce [i][i] = true;
    if (famoso){
    // ahora vamos a comprobar que le conoce todo el mundo
        int fil=0;
        while (famoso && fil < numPersonas)
            if (!conoce[fil][i]) famoso = false;
            else fil ++;
        }
    return famoso;
}

```

Apartado 3. Implementar el método boolean `esConocidoYConoce(int i){...}` que determine de forma recursiva si la persona `i` es conocida por todos y a la vez conoce a todos.

```

public boolean esConocidoYConoce(int i){
    int k=0;
    boolean esConocido = verColumna(0,i);
    boolean conoce = verFila(i,0);
    return esConocido && conoce;
}

```

```

public boolean verFila(int i , int col){
    boolean res=true;
    if (col < conoce.length){

        if (conoce[i][col])
            res = verFila (i, col+1);
        else res = false;

    }
    return res;
}

```

```

public boolean verColumna(int fila , int i){
    boolean res=true;
    if (fila < conoce.length){
        if (conoce[fila][i])
            res = verColumna (fila+1, i);
        else res = false;
    }
    return res;
}

```

Programación con ficheros en Java

5.1. Introducción

Cuando el volumen de información que puede manejar un programa puede aumentar de forma indefinida es necesario utilizar estructuras de datos especiales. Como ya se ha estudiado en capítulos anteriores, Java permite la utilización de arrays unidimensionales (listas), o multidimensionales (tablas), de tamaños prefijados de antemano para almacenar diferente tipo de información. Este tipo de estructuras de datos puede caracterizarse principalmente por dos hechos. En primer lugar, por encontrarse almacenados en **memoria principal**, es decir, los datos están almacenados en un tipo de memoria que permite acceder directamente a la información. En segundo lugar, por ser de tamaño fijo o limitado, predefinido en el momento de su declaración. Sin embargo, en muchas situaciones es necesario manipular datos o información que superan con creces la capacidad de almacenamiento del computador, piénsese por ejemplo en la posibilidad de manejar un listado que contiene la información personal de todos los habitantes de un país. En estos casos, es necesaria la utilización de estructuras de datos que permitan almacenar los datos en **memoria secundaria**. Habitualmente se denomina memoria secundaria a aquellos soportes físicos no volátiles que permiten el almacenamiento estable de información. Los elementos más habituales que forman este tipo de memoria suelen ser los discos duros, unidades de disco, CD-ROM, cintas magnéticas, etc. Este nuevo tipo de estructura de datos, denominadas habitualmente **archivos**, o **ficheros** (del inglés *File*), permitirá a un programa almacenar y procesar una cantidad, en principio, ilimitada de datos.

5.1.1. Conceptos básicos sobre ficheros

Un archivo, o fichero, puede definirse como una colección de datos homogéneos almacenados en un soporte físico del computador que puede ser permanente o volátil.

De la anterior definición pueden extraerse varias conclusiones de carácter general:

1. En primer lugar, todo archivo o fichero almacenará colecciones de datos del mismo tipo (homogéneos) como sucede en el caso de arrays o vectores.
2. En segundo lugar, al conjunto de elementos, o datos, almacenados en el fichero se le denomina **registro**, cada uno de estos registros pueden estar formado por un conjunto de **campos** que pueden ser de diferentes tipos (heterogéneos).

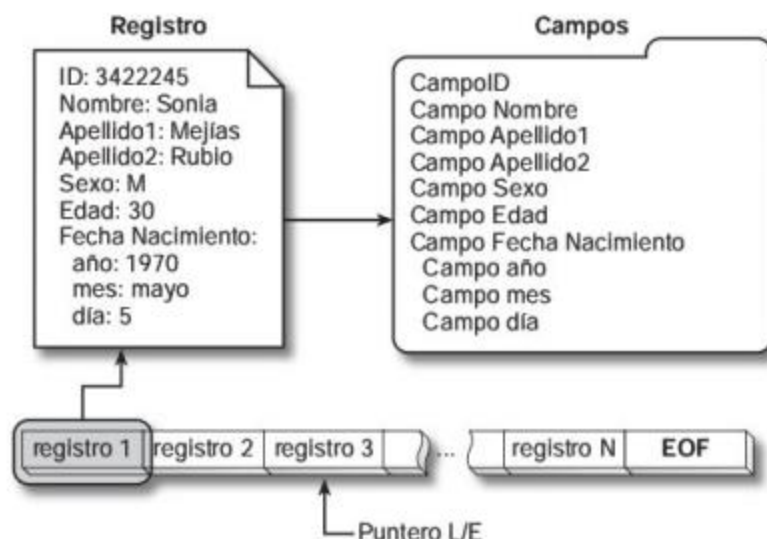


Figura 5.1. Relación entre los registros y campos almacenados en un fichero.

3. Por último, los ficheros pueden estar almacenados en soportes estables del computador (como ya se había indicado previamente en la introducción), como discos duros, unidades de disco, cintas magnéticas, etc... o en memoria principal (memoria RAM). Luego la capacidad real de almacenamiento de un fichero dependerá del soporte físico utilizado.

Otras características generales que deben ser tenidas en cuenta por el programador antes de utilizar este tipo de estructuras son:

1. Todos los elementos de un fichero poseen una determinada posición dentro de la secuencia de datos.
2. El tamaño de un fichero no es fijo, pudiendo cambiar dinámicamente el número de elementos que almacena.
3. Cada fichero será referenciado (nombrado) dentro del programa mediante un identificador, que será utilizado como una referencia lógica al fichero físico, o real, que se encuentra almacenado en memoria.
4. En cada operación de lectura/escritura sobre un fichero sólo podrá leerse/escribirse un único registro.

5. La operación de acceso a la información de un fichero puede realizarse de dos formas diferentes:
 - **Acceso secuencial.** El acceso al **registro i** del fichero necesita la exploración secuencial de los **(i - 1)** registros anteriores. Por tanto, el tiempo de acceso a un determinado dato vendrá determinado por la posición que ocupa dentro de la secuencia.
 - **Acceso directo, o aleatorio.** El acceso al registro deseado se realiza directamente indicando la posición que ocupa en la secuencia (por ejemplo, el acceso a los datos almacenados en un array es directo, pudiendo recuperar el dato deseado utilizando una variable denominada índice). La principal característica de este tipo de acceso consiste en que el tiempo medio de acceso a un registro es constante.

La Figura 5.2. muestra la diferencia entre realizar un acceso secuencial y un acceso directo a la información de un fichero.

Acceso secuencial sobre un fichero

Puntero L/E



ID: 3422245 Nombre: Ricardo Apellido1: Aler Apellido2: Mur Sexo: V Edad: 34 Fecha Nacimiento: año: 1968 mes: octubre día: 20	ID: 7485300 Nombre: Juan Apellido1: González Apellido2: Castillo Sexo: V Edad: 26 Fecha Nacimiento: año: 1980 mes: julio día: 8	...	ID: 97442245 Nombre: María Apellido1: García Apellido2: Pérez Sexo: M Edad: 54 Fecha Nacimiento: año: 1948 mes: enero día: 30	...	ID: 07642545 Nombre: Luisa Apellido1: Torres Apellido2: Gómez Sexo: M Edad: 10 Fecha Nacimiento: año: 1992 mes: julio día: 18
Posición 0	Posición 1		Posición J		Posición N

Buscar: García Pérez, María (ID = 97442245)

Tiempo = tiempo acceso 1 registro (t) * J (posición que ocupa) = t * J

Acceso directo sobre un fichero

Puntero L/E



ID: 3422245 Nombre: Ricardo Apellido1: Aler Apellido2: Mur Sexo: V Edad: 34 Fecha Nacimiento: año: 1968 mes: octubre día: 20	ID: 7485300 Nombre: Juan Apellido1: González Apellido2: Castillo Sexo: V Edad: 26 Fecha Nacimiento: año: 1980 mes: julio día: 8	...	ID: 97442245 Nombre: María Apellido1: García Apellido2: Pérez Sexo: M Edad: 54 Fecha Nacimiento: año: 1948 mes: enero día: 30	...	ID: 07642545 Nombre: Luisa Apellido1: Torres Apellido2: Gómez Sexo: M Edad: 10 Fecha Nacimiento: año: 1992 mes: julio día: 18
Posición 0	Posición 1		Posición J		Posición N

Buscar: García Pérez, María (ID = 97442245)

Tiempo = tiempo acceso 1 registro (t') = t'

Figura 5.2. Posibles tipos de accesos a la información almacenada en un fichero.

6. Cuando se realiza una operación de lectura/escritura sobre un fichero, existe un identificador de la posición del archivo sobre la que se está trabajando denominado **puntero de lectura/escritura** (puntero L/E).
7. Todo fichero dispone de un registro especial que indica el final del mismo, este carácter denominado Fin de Fichero, o EOF (*End Of File*), que es incluido automáticamente por el sistema.



Figura 5.3. Todo fichero utiliza un carácter especial (EOF) que indica el final del mismo.

5.1.2. Operaciones sobre ficheros

Una vez descritos los conceptos básicos sobre ficheros, es necesario analizar cuáles son las operaciones más habituales que pueden realizarse sobre los mismos. Una posible clasificación de estas operaciones, podría resumirse en:

1. Operación de **creación**. Esta operación permite relacionar un nombre lógico con el fichero físico. Este nombre lógico será utilizado dentro del programa para acceder a la información física almacenada en memoria. La operación de creación, o *asignación*, dependerá del lenguaje considerado, en el caso de Java se utilizará un constructor de la clase (o tipo de fichero) que se necesite.
2. Operación de **apertura**. Una vez el fichero ha sido creado, debe abrirse para permitir acceder a la información que tiene almacenada. En función del lenguaje utilizado, es posible abrir un fichero de diferentes formas, o *modos*, los más habituales suelen ser: *sólo lectura*, *sólo escritura*, o *lectura/escritura*, definiendo de esa forma qué tipo de operaciones son realizables sobre los datos almacenados. Como se estudiará más adelante, Java resuelve este problema al indicar el *tipo* del fichero que se construye mediante la clase empleada (si es un fichero que acepta un flujo de entrada (se crea un fichero para escritura, si el flujo es de salida (se tratará de un fichero para realizar operaciones de lectura).
3. Operaciones de **lectura/escritura**. Operaciones que permiten leer o escribir nuevos datos en el fichero.
4. Operaciones de **inserción/borrado**. Operaciones que permiten modificar los valores de alguno de los registros almacenados en el fichero, o simplemente borrarlos.
5. Operación de **renombrado/eliminación** de un fichero. Suele existir la posibilidad de cambiar el nombre físico de un fichero o incluso su completa eliminación.
6. Operación de **desplazamiento** en un fichero. Este tipo de operación suele referirse a la posibilidad de mover el puntero de lectura/escritura a lo largo de los diferentes registros que forman el fichero. En el caso de ficheros de acceso secuencial no existirán instruc-

ciones que permitan saltar o moverse al registro deseado, sin embargo, en aquellos ficheros que dispongan de un acceso directo o aleatorio, sí será posible situar el puntero de lectura/escritura sobre la posición deseada.

7. Operación de **cerrar**. Aunque lenguajes como Java cierran todos los ficheros abiertos automáticamente al finalizar la ejecución del programa, es muy conveniente que estos ficheros sean cerrados una vez terminan las operaciones sobre ellos. Esto es debido a que si se produce una terminación inesperada o anormal del programa (se produce una excepción, véase Capítulo 2) el contenido del fichero podría quedar inutilizable, es decir, los datos contenidos en el mismo podrían dañarse de forma irreparable no pudiendo recuperarse.

Generalmente, para trabajar correctamente sobre uno o varios ficheros, será necesario realizar un subconjunto de las operaciones anteriores. En particular, siempre que se trabaje con un fichero se deberían realizar las siguientes operaciones:

1. **Crear**, o asignar, un nombre lógico al fichero físico.
2. **Abrir** el fichero¹.
3. **Operar** sobre el fichero (lectura/escritura, inserción/borrado, etc...).
4. **Cerrar** el fichero.

En Java el conjunto de operaciones disponibles es muy variado y son implementadas a través de un conjunto de métodos implementados por la clase particular que se esté utilizando para construir el fichero. Por tanto, será necesario estudiar las diferentes clases disponibles, y sus métodos asociados, para manipular los contenidos del fichero. Este capítulo mostrará cómo es posible trabajar con ficheros en Java utilizando algunas de las clases más frecuentes.

5.1.3. Tipos de ficheros

Antes de comenzar a estudiar cómo manipular ficheros, es necesario conocer qué tipos de ficheros pueden existir. De esta forma, será más sencillo entender la relación que existe entre las diferentes clases disponibles en Java, y los datos sobre los que trabajan. Existen diferentes formas de clasificar los posibles tipos de ficheros, en general pueden clasificarse en función de:

- a) **Organización de los registros en memoria**. En este caso puede hablarse de ficheros con organización *secuencial*, donde los registros se encuentran almacenados de forma consecutiva en memoria. Organización *directa*, o *aleatoria*, donde el orden físico de los registros en memoria no tiene por qué coincidir con el orden lógico en el que han sido almacenados en memoria. Y finalmente, de organización *secuencial indexada*, en la que deben utilizarse dos ficheros, uno de ellos es el *fichero de datos* que contiene la información ordenada en función de un campo clave. El segundo, *fichero de índices*, contiene todas las claves almacenadas y es utilizado para lograr un acceso directo a la información almacenada en el fichero de datos.

¹ Debe recordarse que normalmente es posible abrir de diferentes formas el fichero: sólo lectura, sólo escritura, o lectura/escritura, en función del tipo de operación que desee realizarse sobre los datos almacenados.

- b) Acceso a la información almacenada** Como se ha visto anteriormente (y de forma muy relacionada con la organización de los mismos) suele diferenciarse entre ficheros de *acceso secuencial* y de *acceso aleatorio*.
- c) Tipo de información almacenada** Tradicionalmente suele diferenciarse entre *ficheros binarios* y *ficheros de texto*, siendo los primeros aquellos que almacenan secuencias de dígitos binarios (por ejemplo, ficheros que almacenan datos de un determinado tipo: ficheros de enteros, float, double, boolean, etc...). Por el contrario, los ficheros de texto son aquellos que almacenan caracteres alfanuméricos (ASCII, Unicode, UTF8, UTF16), es decir, ficheros que almacenan información alfanumérica en un formato estándar y que, por tanto, podrían ser portables entre diferentes máquinas. Los ficheros de texto se caracterizan porque pueden ser leídos, y/o modificados (en función de los permisos de lectura/escritura), por aplicaciones denominadas editores de texto.

5.2. Ficheros en Java

5.2.1. Clases básicas para la manipulación de ficheros

Basándose en las clasificaciones anteriores, donde se han utilizado características como la forma de acceso a la información (secuencial o directa), o el tipo de información almacenada (binarios o de texto), es posible realizar un breve resumen de algunas de las clases disponibles en Java 1.2 y que son ampliamente utilizadas para la gestión de ficheros. La Tabla 5.1 muestra algunas de las clases que serán utilizadas en este capítulo, junto con algunas de sus características básicas como el tipo de información almacenada por los ficheros, o el acceso que puede realizarse a los datos. Las clases disponibles para trabajar con ficheros en Java se encuentran almacenadas en el paquete `java.io` (de input/output).

La mayoría de estas clases pueden caracterizarse en función de cómo leen los datos (básicamente mediante flujos de bytes, o directamente leyendo/escribiendo datos u objetos), y por cómo deben ser definidos los ficheros asociados a estas clases (constructores de las clases asociadas). Como puede verse en la tabla anterior, existen diversas clases en Java que permiten trabajar con información binaria como por ejemplo `FileInputStream/FileOutputStream`, o sobre caracteres ASCII (texto), como `BufferedReader/BufferedWriter`. Otras clases como `DataInputStream/DataOutputStream` permiten leer o escribir datos de tipos básicos (`int`, `long`, `float`, `double`, etc...), en ficheros. Sin embargo, para leer o escribir cadenas de texto, será necesario utilizar clases como `PrintStream/PrintWriter`. Como puede verse en la tabla, la mayoría de las clases disponibles utilizan un acceso secuencial a los datos del fichero (aunque en los ejercicios del capítulo, se verá que muchas de estas clases implementan métodos como `skipBytes()`, o `skip()`, que permiten saltar, y de esa forma no leer, algunos bytes del fichero), excepto la clase `RandomAccessFile` que fue creada con el propósito de implementar el acceso directo, o aleatorio, sobre un fichero binario.

Tabla 5.1. Clases para la gestión de ficheros, características básicas.

Nombre Clase (java.io)	Tipo información almacenada	Lectura/ escritura información	Acceso datos
BufferedInputStream/BufferedOutputStream	binaria	bytes	secuencial
BufferedReader/BufferedWriter	texto	ASCII	secuencial
DataInputStream/DataOutputStream	binaria	datos	secuencial
InputStreamReader/OutputStreamWriter	texto	ASCII	secuencial
FileInputStream/FileOutputStream	binaria	bytes	secuencial
FileReader/FileWriter	binaria	int	secuencial
ObjectInputStream/ObjectOutputStream	binaria	objetos (bytes)	secuencial
PrintStream/PrintWriter	texto	String	secuencial
RandomAccessFile	binaria	bytes	directo

Las diferencias entre las clases anteriores (aun cuando lean o escriban el mismo tipo de información) se deben a los diferentes métodos disponibles para cada una de ellas. Estos métodos permiten realizar las diferentes operaciones sobre los datos almacenados de una forma más cómoda. Por tanto, será necesario estudiar no sólo las clases existentes, sino también de qué métodos disponen, para poder decidir cuál es la clase adecuada para resolver el problema que se plantee en cada situación.

El último apartado de este tema desarrollará diversos ejemplos empleando las clases anteriores. De esta forma, se pretende estudiar algunas de las clases Java más frecuentemente utilizadas para la implementación de programas cuyos datos son leídos y/o escritos contra ficheros.

5.2.2. Declaración de un fichero en Java

Un fichero en Java es un objeto o implementación física de una clase particular. En este caso, y desde las primeras versiones de Java, puede utilizarse la clase genérica `File` para construir un objeto que represente al fichero sobre el que se va a trabajar. Esta clase curiosamente no representa realmente un fichero, sino que es algo más general, la clase `File` puede emplearse para representar:

- El nombre de un archivo físico particular.
- Los nombres de un conjunto de archivos almacenados en un *directorio*.

Para construir un fichero en Java será necesario invocar uno de los posibles constructores de la clase considerada. Por ejemplo, si se utiliza la clase `File` pueden emplearse cualquiera de los siguientes constructores:

```
//1er constructor File (File objDirectorio, String nomArchivo);
File miRuta = new File("c:\\java\\ej5\\cap5\\");
```

```
File archivoEjemplo = new File (miRuta,"miFichero.txt");

//2º constructor File(String directorio);
File archivoEjemplo = new File("c:\\java\\ej5\\cap5\\miFichero.txt");

//3er constructor File(String directorio, String nomArchivo);
File archivoEjemplo = new File("/home/java/ejs/cap5/", "miFichero.txt");
```

La utilización del constructor permite establecer la relación entre el objeto y el fichero físico; es importante resaltar las siguientes características de los anteriores constructores:

1. El primer constructor utiliza una declaración del fichero mediante una ruta (o path) al que le concatena el nombre del fichero físico. Debe observarse que al constructor se le pasa un objeto (`objDirectorio`) de tipo `File`, que será utilizado para establecer la referencia entre el fichero físico y el objeto.
2. En la segunda declaración se utiliza directamente una cadena que representa el lugar físico donde se encuentra almacenado el fichero. En el caso del sistema operativo Windows se utiliza la barra invertida "`\`" (o *backslash*). El problema que aparece en Java (y otros lenguajes como C, o C++) es que esta barra habitualmente se utiliza para indicar al compilador determinados caracteres de control (por ejemplo `\n` suele indicar salto de línea, o `\t` una tabulación), por lo que si se emplease la cadena: "c:\java\ej5\cap5", el compilador trataría los caracteres "`\j`", "`\e`", "`\c`", como caracteres de control y no como una cadena de texto. Para evitar que esto suceda deben emplearse dos barras invertidas: "`\\`", luego la cadena anterior debe ser proporcionada al compilador como "c:\\java\\ej5\\cap5". En otros sistemas operativos como Linux se utiliza la barra de división "/" para indicar el lugar donde se encuentra el fichero (path), por lo que los problemas anteriores no aparecen ("/home/david/java/ejs/cap5/").
3. Finalmente, el último constructor permitirá definir un fichero mediante la concatenación de dos cadenas que representan la posición física del fichero dentro de la estructura de directorios (en el ejemplo mostrado se considera que el fichero se encuentra almacenado en un sistema operativo tipo Unix).

A continuación, se muestra un ejemplo donde se construyen tres ficheros empleando los constructores que acaban de describirse. Cuando se utiliza la clase `File`, es posible emplear métodos como `isFile()`, `list()`, `getName()`, o `isDirectory()`, que permiten mostrar diferentes características de los ficheros, o directorios, que representan.

```
import java.io.*;

public class DeclararFicheros {
    public static void main(String args[]) throws IOException{
        //declaración de Ficheros: clase File
        File miFich1 = new File("c:\\java\\ej5\\cap5\\ficheros\\fich1.txt");
        File ruta = new File("c:\\java\\ej5\\cap5\\ficheros\\");
        File miFich2 = new File(ruta,"fich2.txt");
        File miFich3 = new File("c:\\java\\ej5\\cap5\\ficheros\\", "fich3.txt");
```

```
String []nombres;
if (miFich1.isFile()&& miFich2.isFile() && miFich3.isFile()){
    System.out.println("ficheros creados");
    System.out.println("nombre fichero1: "+miFich1.getName();
    System.out.println("nombre fichero2: "+miFich2.getName();
    System.out.println("nombre fichero3: "+miFich3.getName();
}
if (ruta.isDirectory()){
    System.out.println(ruta.getName()+" representa un directorio");
    nombres = ruta.list();
    //se listan los ficheros que se encuentren en el directorio
    for(int i=0; i<nombres.length; i++){
        System.out.println("fichero "+i+" :"+nombres[i]);
    }
}
}
}
//fin main
}
//fin clase DeclararFicheros
```

En este ejemplo puede verse cómo se ha establecido la correspondencia entre los ficheros físicos existentes empleando los tres métodos de los que dispone la clase `File`. También se comprueba cómo esta clase puede utilizarse para representar un directorio. Como se explicará más adelante, el método `main` debe propagar la excepción de E/S (`IOException`) que puede aparecer al trabajar con ficheros (véase Capítulo 2). El anterior ejemplo muestra cómo puede utilizarse un objeto de tipo `File(ruta)` para identificar un directorio ("`c:\\java\\ej5\\cap5\\ficheros\\`"); una vez se ha creado el objeto es posible, por ejemplo, listar el contenido del directorio (`ruta.list()`). La ejecución del programa mostraría por pantalla:

```
ficheros creados
nombre fichero1: fich1.txt
nombre fichero2: fich2.txt
nombre fichero3: fich3.txt
ficheros representa un directorio
fichero 0 :fich1.txt
fichero 1 :fich2.txt
fichero 2 :fich3.txt
fichero 3 :fichDatos.dat
fichero 4 :fichDatosPrimitivos.dat
fichero 5 :Libros.dat
fichero 6 :FichAleatorio1.dat
fichero 7 :prueba.txt
fichero 8 :Personas.dat
fichero 9 :ObjetosPersona.dat
fichero 10 :DirectPersonas2.dat
fichero 11 :dos.txt
```

```
fichero 12 :uno.txt
fichero 13 :salida.txt
```

La salida anterior muestra cómo las variables `miFich1`, `miFich2` y `miFich3` representan a los ficheros físicos situados en un determinado directorio, mientras que la variable `ruta` representa un directorio y es posible realizar con ella otras acciones, utilizando métodos como: `isDirectory()`, `getName()`, o `list()`.

5.2.3. Flujos de entrada/salida en Java

Una vez abierto un fichero, asociándolo a un objeto de tipo `File`², puede accederse a la información almacenada en el mismo. Sin embargo, para trabajar sobre estas estructuras de datos antes deberá comprenderse la filosofía de Java para la gestión de ficheros. El concepto básico utilizado en Java para realizar cualquier operación de entrada/salida es el de *flujo de datos*. Un flujo, o *stream*, es simplemente una secuencia de datos, normalmente bytes de ocho bits, que pueden fluir desde una fuente de datos a un destino. Java dispone de dos flujos predefinidos, uno de entrada (`InputStream`), y otro de salida (`OutputStream`). Para poder leer, o escribir, desde un programa en Java es necesario establecer una serie de flujos que permitan conectar el fichero físico con el punto del programa que recibirá, o enviará, los datos al mismo. Java dispone de una gran cantidad de objetos de flujo (es decir, clases cuyos nombres incluyen las palabras *InputStream* o *OutputStream*) que permiten al programador crear el objeto o fichero adecuado a cada necesidad. La Figura 5.4. muestra el concepto de flujo de forma gráfica como un conjunto de tuberías que conectan las fuentes de datos con sus destinos.

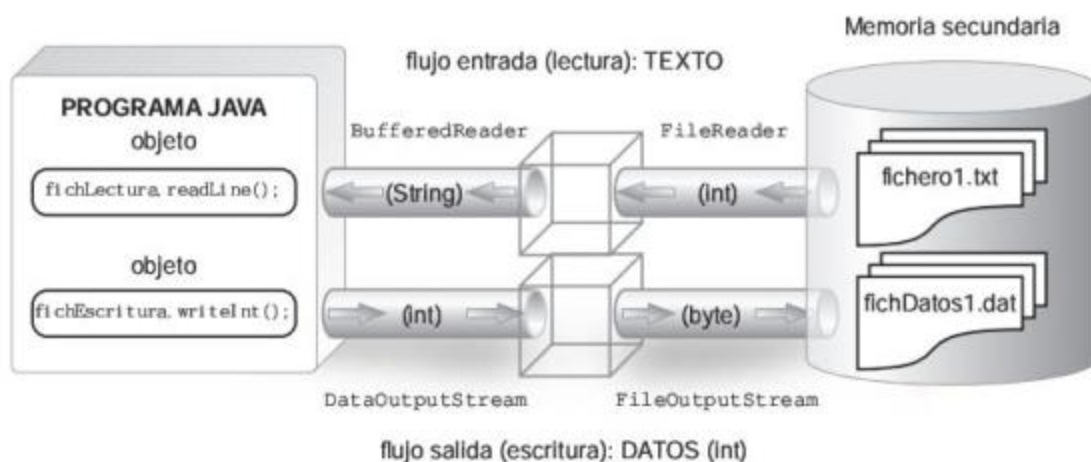


Figura 5.4. Flujos de entrada/salida sobre ficheros en Java.

² Como se verá en ejemplos posteriores, la mayoría de los constructores de las clases utilizadas aceptan como parámetro de entrada un `String` que representa al fichero, no siendo necesario utilizar un objeto de tipo `File`.

Como puede observarse en la figura, se han representado dos flujos básicos que permiten conectar el programa Java con dos ficheros que almacenan diferentes tipos de información, binaria en el caso de `fichDatos1.dat`, y de texto en el caso de `fichero1.txt`. Para conectar ambos ficheros es necesario establecer los correspondientes flujos que permitan de una forma cómoda la lectura o escritura de los datos, de forma transparente al programador. Los flujos son creados mediante la implementación de objetos pertenecientes a determinado tipo de clases. En la figura se ha diferenciado entre dos flujos, uno de salida (`DataOutputStream` que necesita para poder ser construido otro flujo de tipo `FileOutputStream`) que permite (a través de métodos como `writeInt()`, definidos en la primera clase) escribir directamente datos con tipo en el fichero, y otro de entrada (`BufferedReader` que necesita a su vez un flujo de tipo `FileReader`) para poder leer líneas de texto del fichero (utilizando el método `readLine()` de la clase correspondiente).

En Java pueden distinguirse entre varios tipos de flujos de entrada/salida, en función de la complejidad de la información leída o escrita:

- **Flujos de E/S de bajo nivel.** Estos flujos implementan la comunicación más básica de información, se leen o escriben directamente bytes en el fichero, estos ficheros se consideran siempre binarios.
- **Flujos de E/S de alto nivel.** En este caso se pueden leer o escribir directamente datos (por ejemplo, datos de tipos básicos: `int`, `long`, `char`, `float`, `boolean`,...), o texto.
- **Flujos de E/S para el almacenamiento de objetos.** Java permite definir flujos para almacenar y recuperar objetos pertenecientes a cualquier clase. Este tipo de almacenamiento puede emplearse para almacenar el estado de un programa (situación anterior a que una aplicación terminase) o para almacenar estructuras muy complejas de información.

Por tanto, para leer o escribir un dato sobre un fichero, es necesario establecer:

1. Tipo de fichero que va a utilizarse (binario o de texto). Dependiendo del tipo de fichero a manipular, será necesario emplear un tipo de clase Java u otra. Siempre será recomendable (en particular, cuando se está comenzando a trabajar con un lenguaje tan rico como Java), utilizar las clases más simples que se conozcan.
2. Establecer un flujo de entrada o salida. Las operaciones (y por tanto, las clases asociadas) de lectura necesitan flujos de entrada (`InputStream`, `Reader`, etc.), las de escritura necesitan flujos de salida (`OutputStream`, `Writer`, etc.).

Flujos de E/S de bajo nivel

El flujo de más bajo nivel implementado en Java es el orientado a leer o escribir bytes en un fichero. Este tipo de flujo permite garantizar la portabilidad de los datos almacenados. Sin embargo, las operaciones de lectura/escritura son generalmente muy engorrosas de realizar para el programador, que deberá conocer los diferentes formatos de los datos que desea leer o escribir.

El siguiente ejemplo muestra cómo, una vez creado un fichero, se establece un flujo de salida que permite la escritura de un conjunto de **bytes** sobre un fichero. Por tanto, una vez utilizada la clase `File` para construir el fichero, este objeto es pasado como parámetro al constructor

de la clase que crea el flujo de salida (en este caso `FileOutputStream`). Una vez establecido el flujo pueden emplearse algunos de los métodos de esta clase, como `write()`, para escribir los datos en el fichero.

```
import java.io.*;

public class EscribirBytesFichero {
    public static void main(String args[]) throws IOException{
        //declaración de ficheros: clase File
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichDatos = new File(ruta,"fichDatos.dat");
        // se crea el flujo de SALIDA...escritura
        FileOutputStream flujoSalida = new FileOutputStream(fichDatos);
        //información a almacenar.....secuencia de 1000 bytes
        byte []datos= new byte[1000];
        byte []datos2= new byte[100];
        for(int i=0;i<datos.length;i++){
            datos[i]=(byte)i; //almacenamos números entre 0 y 255
        }
        for(int i=0;i<datos2.length;i++){
            datos2[i]=(byte)i; //almacenamos números entre 0 y 255
            flujoSalida.write(datos2[i]); // se almacena byte a byte
        }
        flujoSalida.write(datos); //escribimos el array completo;
        flujoSalida.close(); //se cierra el fichero
    }
}

//fin main
//fin clase Escribir Bytes en Fichero
```

Como puede verse en el ejemplo anterior, una vez creado el objeto de tipo `File` (`fichDatos`), se establece el flujo de salida (`flujoSalida=new FileOutputStream(fichDatos)`) y se le pasa como parámetro `fichDatos` (si no existe un fichero físico con el nombre especificado, se creará uno vacío). A continuación, se crean dos arrays de bytes (`datos` y `datos2`) que serán utilizados para almacenar la información a escribir en el fichero de datos. Una vez almacenados los datos que se desean escribir en el fichero, puede utilizarse el método `write()` de la clase `FileOutputStream` permite escribir arrays completos (`flujoSalida.write(datos)`), o elemento a elemento (`flujoSalida.write(datos2[i])`), en el fichero.

Finalmente, y una vez los diferentes datos han sido almacenados en el fichero, antes de terminar la ejecución del programa, el flujo asociado al fichero de datos debe ser cerrado (`flujoSalida.close()`).

Análogamente, el siguiente ejemplo muestra un ejemplo que permitiría leer los bytes almacenados en el fichero anterior. Para ello, se crea un objeto de tipo `FileInputStream`, sobre el que pueden emplearse métodos como `read()`.

```
import java.io.*;

public class LeerBytesFichero {
```

```
public static void main(String[] args) throws IOException{
    //declaración de Ficheros, clase File
    String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
    File fichDatos = new File(ruta,"fichDatos.dat");
    //creamos el flujo de ENTRADA...lectura
    FileInputStream flujoEntrada = new FileInputStream(fichDatos);
    //información a leer.....secuencia de bytes
    //se calcula el tamaño del fichero
    int tamanoFich = (int)fichDatos.length();
    byte []datos = new byte[tamanoFich];
    // se leen todos los datos almacenados en el fichero
    flujoEntrada.read(datos);
    //se muestran por la salida estándar
    for(int i=0;i<datos.length;i++){
        System.out.println("dato "+i+" :"+datos[i]);
    }
    flujoEntrada.close(); //se cierra el fichero
} //fin main
} // fin clase LeerBytesFichero
```

En este ejemplo, se muestra como una vez creado el flujo de entrada (`flujoEntrada = new FileInputStream(fichDatos)`), se utiliza el método `length()` del fichero (`fichDatos.dat`) para determinar el número de bytes que deben ser leídos (`(int)fichDatos.length()`). La longitud del fichero (número de bytes almacenados) se utilizará para construir un array de bytes que almacene la información leída (`datos=new byte[tamanoFich]`). Finalmente, los datos son leídos (`flujoEntrada.read(datos)`) y mostrados por la salida estándar. Es importante remarcar para ambos ejemplos que una vez terminadas las operaciones sobre los ficheros, los flujos asociados a éstos, deben ser cerrados (`flujoEntrada.close()`). Realmente, la ejecución del método `close()` cierra el flujo asociado al fichero, por lo que toda la información que quedase almacenada en el mismo (denominado habitualmente buffer de lectura/escritura) se volcará contra el fichero. Debe tenerse en cuenta que si la ejecución del código finaliza correctamente, la propia máquina virtual de Java se encarga de cerrar los ficheros que hayan podido quedar abiertos; sin embargo, si se produjese una terminación abrupta, o inesperada, de la ejecución del programa estos ficheros podrían quedar abiertos y perderse la información almacenada.

Flujos de E/S de alto nivel

Como puede verse en los ejemplos anteriores, la lectura/escritura de información a un nivel tan básico puede ser un tanto engorrosa. Esto es debido a que si se desea leer o escribir datos de un determinado tipo, el programador deberá conocer cuál es el formato concreto de los datos que va a manipular, lo cual podría dificultar la implementación del programa. Si se desea que el proceso de lectura/escritura de los datos se transparente al programador (es decir, que no sea necesario conocer cuántos bytes utiliza un determinado dato para codificarse), deben utilizarse otros tipos de flujos (clases) que dispongan de métodos más adecuados, y que permitan abstraer el proceso de lectura/escritura de un determinado dato.

Este apartado se ha dividido en tres secciones diferenciadas por los tipos de datos considerados, y por tanto, por los tipos de ficheros asociados que se manipulan:

1. Ficheros que permiten el almacenamiento de cualquier tipo de *dato primitivo* en Java (estos ficheros serán de tipo binario).
2. Ficheros de *texto*, es decir, que permiten la lectura y escritura de caracteres ASCII.
3. Ficheros de *acceso aleatorio*, se mostrará cómo se implementan en Java este tipo de ficheros.

Ficheros para el almacenamiento de tipos primitivos de Java

Java dispone de un conjunto de clases que permite leer y escribir valores de tipos de datos primitivos. Las clases utilizadas para hacer la E/S son `DataInputStream/DataOutputStream`. La idea que rodea a este tipo de flujos es que estas clases se encargan de hacer transparente al programador el proceso de conversión a bytes de los tipos básicos utilizados. De esta forma el programador únicamente debe encargarse de leer o escribir el tipo de dato básico que desee. Por tanto, este tipo de clases dispondrá de métodos específicos como `readFloat()`, `readInt()`, `readChar()`, en el caso de `DataInputStream` y métodos como `writeFloat()`, `writeInt()`, `writeDouble()`, `writeBoolean()`, en el caso de `DataOutputStream`.

```
import java.io.*;

public class EscribirDatosPrimitivos {
    public static void main(String[] args) throws IOException{
        //declaración de Ficheros: clase File
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichDatos = new File(ruta,"fichDatosPrimitivos.dat");
        //se crea el flujo de SALIDA....bytes
        FileOutputStream flujoSalida = new FileOutputStream(fichDatos);
        //se conecta el flujo de bytes al flujo de datos
        DataOutputStream datosSalida = new DataOutputStream(flujoSalida);
        // se escriben algunos valores de tipos básicos en el fichero
        for(int i=0;i<100;i++){
            datosSalida.writeBoolean(true);
            datosSalida.writeChar('J');
            datosSalida.writeDouble(222222222D);
            datosSalida.writeInt(2003);
            datosSalida.writeFloat(999999999F);
            datosSalida.writeLong(777777777L);
            datosSalida.writeUTF("Hola Mundo");
        }
        datosSalida.close(); //se cierra el fichero
    }
}
//fin main
//fin EscribirDatosPrimitivos
```


Este ejemplo muestra cómo una vez son establecidos los flujos de salida necesarios y estos son asociados al fichero físico (`flujoSalida=new FileOutputStream(fichDatos)`), es posible escribir cualquier valor de un tipo básico de Java utilizando cualquiera de los métodos disponibles en la clase. Una vez almacenados los datos, el siguiente ejemplo muestra cómo leer la información almacenada en el fichero `fichDatosPrimitivos.dat`

```
import java.io.*;

public class LeerDatosPrimitivos {
    public static void main(String[] args) throws IOException{
        //declaración de Ficheros: clase File
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichDatos = new File(ruta,"fichDatosPrimitivos.dat");
        //se crea el flujo de ENTRADA....bytes
        FileInputStream flujoEntrada = new FileInputStream(fichDatos);
        //se conecta el flujo de bytes al flujo de datos
        DataInputStream datosEntrada = new DataInputStream(flujoEntrada);
        //información a leer.....tipos básicos
        for(int i=0;i<100;i++){
            System.out.println("dato:"+datosEntrada.readBoolean());
            System.out.println("dato:"+datosEntrada.readChar());
            System.out.println("dato:"+datosEntrada.readDouble());
            System.out.println("dato:"+datosEntrada.readInt());
            System.out.println("dato:"+datosEntrada.readFloat());
            System.out.println("dato:"+datosEntrada.readLong());
            System.out.println("dato:"+datosEntrada.readUTF());
        }
        datosEntrada.close(); //se cierra el fichero
    }
}
//fin LeerDatosPrimitivos
```

Estas clases disponen de una gran cantidad de métodos que permiten realizar las operaciones de lectura/escritura sobre el fichero, en este ejemplo sólo se muestran algunos de los métodos disponibles, puede consultarse el API de Java 1.2 para un análisis más detallado de las mismas. La clase `EscribirDatosPrimitivos` ha generado un fichero que contiene 700 datos de diferentes tipos (100 bloques de 7 tipos de datos primitivos en Java); la lectura de estos datos es realizada en `LeerDatosPrimitivos` que muestra por pantalla todos los datos almacenados. La salida de este programa sería:

```
dato:true
dato:J
dato:2.22222222E8
dato:2003
dato:1.0E9
dato:77777777
```

```

dato:Hola Mundo
dato:true
dato:J
dato:2.22222222E8
dato: ..... //etc...

```

Sin embargo, y aunque como muestran los ejemplos anteriores Java lo permite, no es muy aconsejable mezclar diferentes tipos de datos primitivos dentro de un mismo fichero debido a que si los datos se leyesen en un orden erróneo provocaría diversos errores. Por ejemplo, si se prueban a intercambiar las siguientes líneas de código se observa como los números leídos no corresponden con los inicialmente almacenados. Esto es debido a que al leer el flujo de bytes, el método correspondiente los transforma a la representación numérica destino:

```

System.out.println("dato:"+datosEntrada.readInt()); //se espera un Double
System.out.println("dato:"+datosEntrada.readDouble()); //se espera un Int

System.out.println("dato:"+datosEntrada.readLong()); //se espera un Float
System.out.println("dato:"+datosEntrada.readFloat()); //se espera un Long

```

La salida del programa modificado muestra cómo los valores numéricos no se han transformado según lo esperado:

```

dato:true
dato:J
dato:1101692335
dato:8.086349223907986E-174
dato:5651572401939415040
dato:5.0010166E-11
dato:Hola Mundo
dato:true
dato:J
dato:1101692335
dato:..... //etc...

```

Si por ejemplo, se cambiasen las líneas que leen una cadena por un número se generaría un error de incompatibilidad de tipos al no poderse transformar una cadena a un formato numérico:

```

System.out.println("dato:"+datosEntrada.readUTF()); //se espera un Long
System.out.println("dato:"+datosEntrada.readLong()); //se recibe un STRING!!!

```

La salida del programa obtiene los primeros datos correctamente, sin embargo, cuando trata de leer la cadena de texto se produce un primer problema (no muestra ningún carácter), el número de tipo long generado corresponde a los siguientes bytes leídos. Dado que a partir de ese momento la lectura de los bytes no va a coincidir con lo esperado, el siguiente valor del carácter mostrado, ya no es la “J” sino el signo de interrogación “?”; el número de tipo double tam-

poco coincide, etc... El proceso continúa hasta que se intenta transformar un valor numérico donde uno de los bytes leídos produce una excepción de tipo `EOFException`.

```

dato:true
dato:J
dato:2.2222222E8
dato:2003
dato:1.0E9
dato:                                     //no se lee ninguna cadena
dato:50972444393482 //este número no es el long almacenado
dato:true
dato:?                                     //el carácter ya no coincide
dato:7.162487406552596E159
dato:4866474
dato:2.9095035E37
dato:
dato:8604135353128
dato:false //ahora el valor booleano es erróneo también
dato:
dato:8.443923916584496E-308
dato:186937java.io.EOFException
    at java.io.DataInputStream.readFully(DataInputStream.java:153)
    at java.io.DataInputStream.readUTF(DataInputStream.java:521)
    at java.io.DataInputStream.readUTF(DataInputStream.java:491)
    at LeerDatosPrimitivos.main(LeerDatosPrimi3728 dato:2.5735328E8
                                tivos.java:30)

Exception in thread "main"

```

Ficheros para el almacenamiento de texto

Un caso muy importante en la E/S de alto nivel consiste en la lectura/escritura de **ficheros de texto**. Este tipo de ficheros almacena sus datos en formato ASCII. Este formato permite convertir todos los datos a caracteres, de tal forma que puedan ser leídos y modificados fácilmente utilizando otras aplicaciones software como los llamados procesadores de texto.

Java permite utilizar varias clases para gestionar ficheros de texto, se utilizará `PrintWriter/BufferedReader` para trabajar sobre este tipo de ficheros. `PrintWriter` es una clase derivada de la clase abstracta `java.io.Writer`. Si se analiza el API de Java 1.2, podrá observarse el conjunto de subclases heredadas de ésta, y que pueden ser utilizadas para trabajar con ficheros de texto. Estas clases son: `BufferedWriter`, `CharArrayWriter`, `FilterWriter`, `OutputStreamWriter`, `PipedWriter`, **`PrintWriter`**, `StringWriter`. En el caso de la lectura de ficheros de texto `BufferedReader` es una clase derivada de `java.io.Reader`, cuyas subclases son: **`BufferedReader`**, `CharArrayReader`, `FilterReader`, `InputStreamReader`, `PipedReader`, `StringReader`.

A continuación, se muestra un programa que permitiría la escritura sobre un fichero de texto, debe observarse que estas clases se caracterizan básicamente por leer y escribir cadenas

(String). Sin embargo, debe recordarse que la diferencia con el tipo de fichero anterior (binarios) reside en cómo es almacenada la información en el fichero físico. La diferencia entre las anteriores clases puede resumirse en el conjunto de métodos utilizables (por ejemplo, los métodos definidos por `Writer` escriben arrays de caracteres en el fichero, sin embargo, `PrintWriter` dispone de métodos que permiten escribir cadenas de texto).

```
import java.io.*;

public class EscribirTexto {
    public static void main(String[] args) throws IOException{
        //fichero de TEXTO
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichTexto = new File(ruta,"fich1.txt");
        //flujo de salida....
        FileOutputStream flujoSalida = new FileOutputStream(fichTexto);
        PrintWriter fSalida = new PrintWriter(flujoSalida);
        //se utilizan diversos métodos de PrintWriter
        char []cadena={'f','i','n'};
        fSalida.print(33333334D); //este número se almacena como texto
        fSalida.println(); //retorno de carro
        fSalida.println("En un lugar de la mancha");
        fSalida.print("de cuyo nombre ");
        fSalida.write("no quiero acordarme....");
        fSalida.write(cadena);
        fSalida.close(); //se cierra el fichero
    }
}
//fin main
//fin EscribirTexto
```

En el ejemplo anterior, se debe tener en cuenta que si el fichero existiese previamente, sus datos serían sobrescritos; la clase `PrintWriter` tiene como métodos principales: `print()`, `println()`, `write()`, pudiendo escribir prácticamente cualquier tipo de dato en forma de texto dentro del fichero. La siguiente clase muestra cómo puede utilizarse un flujo de tipo `BufferedReader` (que a su vez necesita un flujo de tipo `FileReader`), para leer la información almacenada en un fichero de texto.

```
1. import java.io.*;
2. public class LeerTexto {
3.     public static void main(String[] args) throws IOException{
4.         //fichero de TEXTO
5.         String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
6.         File fichTexto = new File(ruta,"fich1.txt");
7.         //flujo de entrada....
8.         FileReader flujoEntrada = new FileReader(fichTexto);
9.         BufferedReader fEntrada = new BufferedReader(flujoEntrada);
10.        //se utilizan diversos métodos de BufferedReader
11.        String cadena = fEntrada.readLine();
```



```
12.      System.out.println("Esto es una cadena: "+cadena);
13.      double dato = Double.parseDouble(cadena);
14.      System.out.println("Esto es un número: "+dato);
15.      //se lee el resto del fichero y se muestra por pantalla
16.      boolean leer=true;
17.      while (leer){
18.          cadena = fEntrada.readLine();
19.          System.out.println(cadena);
20.          if (cadena==null){
21.              leer=false; //fin del fichero
22.              //si se utiliza read().... char != -1
23.          }
24.          fEntrada.close(); //se cierra el fichero
25.      } //fin main
26.  } //fin LeerTexto
```

En este ejemplo debe observarse que la identificación del final del fichero depende del método utilizado; dado que se ha utilizado `readLine()` (que devuelve cadenas) la marca fin de fichero (EOF) se identifica mediante el valor `null`. Sin embargo, si se utilizase alguno de los métodos `read()`, que leen arrays de caracteres (bytes), una vez se alcanzase el final del fichero se devuelve el carácter `-1` lo cual modificaría la condición de salida del bucle.

Ficheros de acceso aleatorio

La clase `RandomAccessFile` encapsula y permite gestionar el concepto de archivo de acceso aleatorio en Java. Esta clase es anómala si se compara con las estudiadas hasta el momento, es una clase totalmente independiente que dispone de métodos únicos y que hereda directamente de `java.lang.Object`. Esta clase presenta gran cantidad de peculiaridades que podrían resumirse en:

- No está basada en el concepto de flujos de E/S como la mayoría de las clases pertenecientes a `java.io`.
- No deriva ni de `InputStream/OutputStream` ni de `Reader/Writer` (clases abstractas de las que suelen derivar la gran mayoría de las clases de E/S en Java).
- Implementa las interfaces de `DataInput/DataOutput` para definir las operaciones básicas de E/S encapsuladas en una única clase. Por tanto, al manejar un fichero de acceso aleatorio no será necesario establecer dos flujos diferentes.
- Al construir un fichero con esta clase es necesario especificar el tipo de acceso, que puede ser: *de sólo lectura*, o *de lectura/escritura*.
- Dispone de métodos específicos como **`seek(long posicion)`**, o **`skipBytes(int desplazamiento)`**, que permiten moverse de un registro a otro del fichero, o situarse directamente en una posición concreta del fichero.
- Los constructores de la clase `RandomAccessFile` son:

```
RandomAccessFile (File ObjArchivo, String ModoAcceso)
RandomAccessFile (String NombArchivo, String ModoAcceso)
```

En el primer constructor, `ObjArchivo` especifica el nombre del archivo mediante un objeto de tipo `File`. En el segundo, se utiliza el `String NombArchivo` para identificarlo. Ambos constructores necesitan especificar el tipo de acceso permitido (`String ModoAcceso`). El modo de acceso puede ser: **"r"** (*read*) si el fichero se abre en modo lectura (no podrá escribirse en él), o en modo **"rw"** (*read/write*) que indica que podrán leerse y escribirse datos en el fichero.

El conjunto básico de métodos que pueden utilizarse con esta clase es:

- **void seek(long posición).** Sitúa el puntero de lectura/escritura en la posición indicada por el parámetro *posición*, desde el comienzo del fichero.
- **long getFilePointer().** Devuelve la posición actual del puntero (en bytes) desde el principio del fichero.
- **int skipBytes(int desplazamiento).** Desplaza el puntero, desde la posición actual, el número de bytes indicado por el parámetro *desplazamiento*.
- **long length().** Devuelve la longitud, o tamaño, del fichero en bytes.

El siguiente ejemplo muestra cómo pueden leerse y escribirse datos sobre un fichero aleatorio en Java.

```
import java.io.*;
public class FicherosAccesoDirecto {
    public static void main(String[] args) throws IOException{
        //declaración de Ficheros: clase File
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichDatos = new File(ruta,"fichAleatorio1.dat");
        //fichero acceso directo
        RandomAccessFile fichAleatorio1 =
            new RandomAccessFile(fichDatos,"rw");
        RandomAccessFile fichAleatorio2 =
            new RandomAccessFile(
                "c:\\java\\ej5\\cap5\\ficheros\\fichAleatorio1.dat","r");
        //E/S desde teclado
        InputSteamReader flujoTeclado = new InputSteamReader(System.in);
        BufferedReader teclado=new BufferedReader(flujoTeclado);
        int tamano_double=8; //1 double = 8 bytes
        //leo datos desde teclado, se añaden al final del fichero
        fichAleatorio1.seek(fichAleatorio1.length()); //final del fichero
        boolean continuar=true;
        double dato=0.0;
        while(continuar){
            System.out.println("Dame números reales... -1 para terminar");
            dato = Double.parseDouble(teclado.readLine());
            //lo almacenamos en el fichero
```

```

    fichAleatorio1.writeDouble(dato);
    if (dato == -1)
        continuar = false;
}
fichAleatorio1.close(); //una vez grabado se cierra el fichero
//a continuación se determina el tamaño del fichero....
//primero en bytes, después el nº de registros almacenados
long longitud=fichAleatorio2.length(); //tamaño en bytes
long cantidadNumeros = longitud/tamano_double;
long numeroLeer = 0;
boolean correcto=false;
System.out.println("Nº de bytes almacenados: "+longitud);
System.out.println("Números almacenados: "+cantidadNumeros);
//pido leer un registro entre el primer y último elemento almacenado
while (!correcto){
    System.out.print("Dame un número que quieras leer del fichero: ");
    numeroLeer = Long.parseLong(teclado.readLine());
    if ((numeroLeer>0) && (numeroLeer<=cantidadNumeros)){
        System.out.println("número de registro erróneo");
        correcto=true;
    }
}
//se lee el registro solicitado
fichAleatorio2.seek((numeroLeer-1)*tamano_double);
double leído = fichAleatorio2.readDouble();
System.out.println(" El número que ocupa la posición "+numeroLeer+"
                    es: "+leído);
fichAleatorio2.close();
} //fin main
} //fin FicherosAccesoDirecto

```

En el ejemplo anterior es interesante observar cómo la simple declaración del fichero `fichAleatorio1` y `fichAleatorio2` permite tanto la escritura, como la lectura de datos sobre los respectivos ficheros (`fichAleatorio1 = new RandomAccessFile(fich-Datos,"rw")`).

Deben observarse varias cuestiones importantes sobre el ejemplo anterior. En primer lugar, si trata de abrirse un fichero para lectura, y éste no existe, se producirá un error de tipo `FileNotFoundException`. Cuando se abre un fichero de acceso aleatorio en modo lectura/escritura los datos no se sobrescriben, luego es posible añadir información al final del fichero si se sitúa el puntero adecuadamente (`fichAleatorio1.seek(fichAleatorio1.length())`). En segundo lugar, los datos se almacenan en forma de bytes, y muchos de los métodos de `RandomAccessFile` trabajan con desplazamientos o posiciones medidas en bytes, de ahí que para acceder al elemento *i*-ésimo, debamos multiplicar el elemento por el número de bytes que ocupa cada registro o dato almacenado (`fichAleatorio2.seek`

((numeroLeer-1)*tamano_double)). Para la inserción de datos se ha definido un flujo que permite conectar la entrada estándar de Java (System.in) con un objeto de tipo `BufferedReader` (`teclado=new BufferedReader(flujoTeclado)`) y de esa forma poder utilizar el método `readLine()` que permitirá leer cualquier línea de texto como un `String` (`Double.parseDouble(teclado.readLine())`).

Almacenamiento de objetos en ficheros (serialización)

Una vez estudiados los diversos flujos que permiten realizar un almacenamiento de datos, desde un nivel básico (basados en flujos de bytes), a un mayor nivel de abstracción (permitiendo manipular directamente datos de tipo primitivos o texto), podríamos preguntarnos si es posible, o incluso necesario, almacenar otro tipo de estructuras. Java, al igual que otros lenguajes de programación como C++, SmallTalk o Eiffel, utiliza para la creación de programas el paradigma orientado a objetos; esto supone que la estructura básica sobre la que va a construirse un programa es el objeto. Un objeto está formado por un conjunto de operaciones (métodos), y un conjunto de datos (atributos). Si el objeto constituye el elemento básico en cualquier programa Java, parece interesante que estas estructuras pudiesen ser directamente almacenadas en ficheros para poder recuperarlas cuando sean necesarias (o transmitir las entre diferentes máquinas para que puedan ser utilizadas por otros programas).

Java permite almacenar objetos completos y recuperarlos posteriormente para su tratamiento. Si la información que se desease almacenar fuese un conjunto de atributos de tipo básico (por ejemplo, un `String` seguido de un conjunto de números), aunque engorroso, sería posible realizarlo con las clases estudiadas hasta el momento. Sin embargo, si se crean objetos que almacenan referencias a otros objetos, su almacenamiento y posterior recuperación, ya no sería posible.

La técnica que permite en Java almacenar objetos se conoce como **serialización**, la serialización de objetos permite tomar cualquier objeto que implemente la *interfaz* **Serializable** y convertirlo en una secuencia de bits que posteriormente puede ser recuperada por el programa para regenerar el objeto original. La serialización es útil si se piensa que un determinado objeto podría ser creado por un programa Java que se esté ejecutando en Windows, y que posteriormente necesita ser utilizado en una máquina Unix. La serialización logra transformar ese objeto en un grupo de bits, enviarlo por una red, y almacenarlo en la máquina destino donde puede ser reconstruido sin problemas. Por tanto, la serialización producirá un fichero *binario* donde se encontrarán almacenados los bytes de información del objeto en cuestión.

La serialización de objetos en Java permite implementar una característica conocida como **persistencia**. Se dice que un objeto es persistente cuando su vida (tiempo en el que es utilizable) no viene determinada por la duración de la ejecución del programa. Es decir, un objeto persistente puede volver a ser utilizado con posterioridad a la finalización del programa. En este caso, se habla de *persistencia débil*, debido a que aunque los objetos pueden recuperarse (al encontrarse almacenados) desde un fichero, todos los detalles de almacenamiento, serialización, y restauración del objeto deben ser implementados por el programador.

Supongamos que se desea almacenar un conjunto de objetos en un fichero que representan libros de una biblioteca. La clase `FichaLibro` crea objetos muy sencillos que permiten repre-

sentar la información disponible sobre un libro. La clase `FechaPublicación` se utiliza para representar la fecha de publicación del mismo, mientras que la clase `FichaLibro` almacena la información concreta de un libro. Como puede verse en el ejemplo, un objeto de tipo `FichaLibro` tiene como uno de sus atributos un objeto de tipo `FechaPublicación`. Por tanto, será necesario serializar todos los datos primitivos y cualquier objeto que sea referenciado por el libro en cuestión. La Figura 5.5. muestra la relación entre los diferentes objetos, y las clases `LeerLibrosPersistentes/GrabarLibrosPersistentes` encargadas de realizar los procesos de lectura/escritura sobre los objetos serializados.

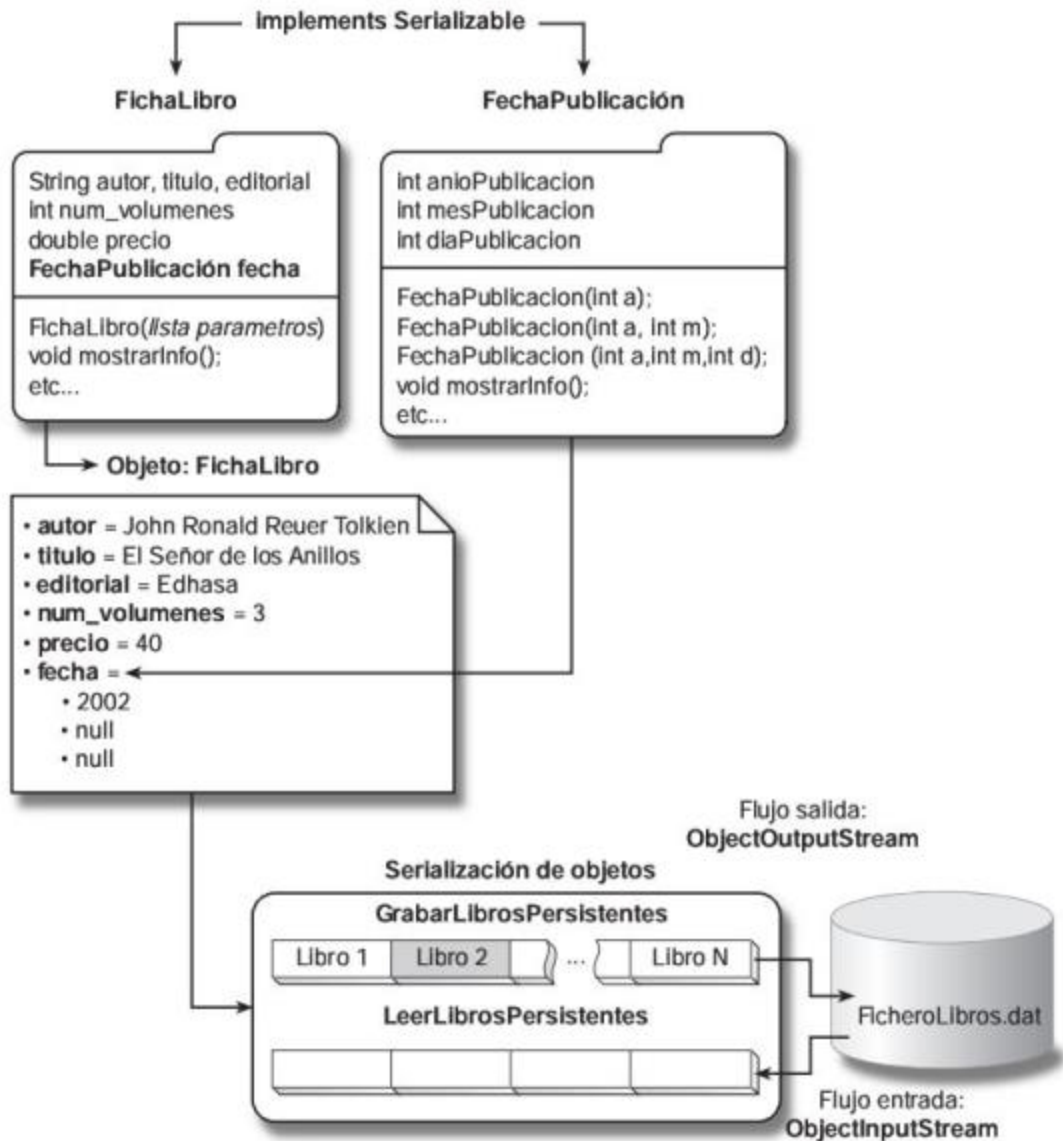


Figura 5.5. Serialización de objetos Java.

A continuación, se implementarán las diversas clases Java necesarias para realizar el proceso de serialización de un conjunto de libros.

```
import java.io.*;

public class FechaPublicacion implements Serializable{
    //atributos
    private int anioPublicacion=-1;
    private int mesPublicacion=-1;
    private int diaPublicacion=-1;
    //constructores
    public FechaPublicacion(int newAnio){
        this.anioPublicacion=newAnio;
    }
    public FechaPublicacion(int newAnio, int newMes){
        this.anioPublicacion=newAnio;
        this.mesPublicacion=newMes;
    }
    public FechaPublicacion(int newAnio, int newMes, int newDia){
        this.anioPublicacion=newAnio;
        this.mesPublicacion=newMes;
        this.diaPublicacion=newDia;
    }
    //mostrar información de una fecha
    public void mostrarInfo(){
        if(this.anioPublicacion != -1)
            System.out.println("Año de Publicación : "+this.anioPublicacion);
        if(this.mesPublicacion != -1)
            System.out.println("Mes de Publicación : "+this.mesPublicacion);
        if(this.diaPublicacion != -1)
            System.out.println("Día de Publicación : "+this.diaPublicacion);
    }
    //etc....
} //fin FechaPublicacion
```

El anterior ejemplo muestra la definición de un objeto de tipo FechaPublicacion que dispone de tres constructores y de un método que permite mostrar el acceso a los atributos de la fecha (public void mostrarInfo()). Es muy importante observar que esta clase debe implementar la interfaz Serializable dado que los objetos de este tipo formarán parte de un objeto más general que será finalmente serializado.

```
import java.io.*;

public class FichaLibro implements Serializable{
    //Atributos
    private String autor;
    private String titulo;
```

```

private int num_volumenes;
private double precio;
private String editorial;
private FechaPublicacion fechaPub; //esto es un objeto
/** Creates a new instance of FichaLibro */
public FichaLibro(String autor, String titulo, int num_volumenes,
    double precio, String editorial, FechaPublicacion fPublicado){
    this.autor = autor;
    this.titulo = titulo;
    this.num_volumenes = num_volumenes;
    this.precio = precio;
    this.editorial = editorial;
    this.fechaPub = fPublicado;
}
//mostrar información de un libro
public void mostrarInfo() {
    System.out.println("Autor : "+this.autor);
    System.out.println("Titulo : "+this.titulo);
    System.out.println("Número de Volúmenes : "+this.num_volumenes);
    System.out.println("Precio : "+this.precio);
    System.out.println("Editorial : "+this.editorial);
    this.fechaPub.mostrarInfo();
}
//otros métodos....
} //fin FichaLibro

```

Esta clase define el objeto `FichaLibro` que finalmente será serializado y almacenado en el fichero. Al igual que en el caso de la definición del objeto `FechaPublicación`, esta clase también debe implementar la interfaz `Serializable`. La clase `GrabarLibrosPersistentes` que se muestra a continuación, construye varios libros (`lib1`, `lib2`, `lib3`), los serializa y finalmente los almacena en un fichero binario.

```

import java.io.*;
public class GrabarLibrosPersistentes {
    public static void main(String[] args) throws IOException{
        //declaración de Ficheros: clase File
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichDatos = new File(ruta,"Libros.dat");
        //se crea el flujo de SALIDA....bytes
        FileOutputStream flujoSalida = new FileOutputStream(fichDatos);
        //se conecta el flujo de bytes al flujo de datos
        ObjectOutputStream librosSalida = new ObjectOutputStream(flujoSalida);
        // se crean algunos libros
        FichaLibro lib1 = new FichaLibro("Henry Miller", "Sexus", 2, 6.05,
            "biblioteca el Mundo", new FechaPublicacion(2002));
    }
}

```

```

FichaLibro lib2 = new FichaLibro("JRR Tolkien", "El Señor de los Anillos",
    3, 40.5, "Edhasa", new FechaPublicacion(2003,3));
FichaLibro lib3 = new FichaLibro("Ernest Heminway",
    "Por quién doblan las campanas", 1, 22, "Espasa",
    new FechaPublicacion(2000,5,20));

//SERIALIZO y almaceno los libros
librosSalida.writeObject(lib1);
librosSalida.writeObject(lib2);
librosSalida.writeObject(lib3);
librosSalida.close(); //se cierra el fichero
} //fin main
} //fin GrabarLibrosPersistentes

```

Finalmente, la clase LeerLibrosPersistentes permite recuperar los objetos almacenados cuando sea necesario para su posterior utilización. Para lograr la lectura/escritura de los objetos serializados, se han utilizado las clases ObjectOutputStream/ObjectInputStream. Para crear los flujos que permiten la serialización y deserialización de los objetos, se necesita a su vez un flujo de bytes de bajo nivel, que como puede observarse en los ejemplos son FileOutputStream/FileInputStream. Una vez creados los flujos, leer y escribir objetos es tan simple como utilizar los métodos readObject()/writeObject() de las correspondientes clases.

```

import java.io.*;

public class LeerLibrosPersistentes {
    public static void main(String[] args) throws IOException,
        ClassNotFoundException{

        //declaración de Ficheros: clase File
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        File fichDatos = new File(ruta,"Libros.dat");
        //se crea el flujo de ENTRADA...bytes
        FileInputStream flujoEntrada = new FileInputStream(fichDatos);
        //se conecta el flujo de bytes al flujo de datos
        ObjectInputStream librosEntrada = new ObjectInputStream(flujoEntrada);
        // se leen (DESERIALIZAN) los libros almacenados
        FichaLibro[] listaLibros = new FichaLibro[3];
        for(int i=0; i<listaLibros.length;i++){
            listaLibros[i] = (FichaLibro) librosEntrada.readObject();
            listaLibros[i].mostrarInfo();
        }
        librosEntrada.close(); //se cierra el fichero
    } //fin main
} //fin LeerLibrosPersistentes

```

Al utilizar el método readObject() de la clase ObjectInputStream para poder reconstruir los objetos almacenados en el fichero puede generarse un nuevo tipo de excepción

llamada `ClassNotFoundException` (debido a la posibilidad que existe de no poder reconstruir el objeto recuperado), por lo que será necesario añadirla en la lista de posibles excepciones lanzadas por el método `main()` (`throws ClassNotFoundException`). Dado que el método `readObject()` devuelve objetos, será necesario transformar el objeto recibido al tipo que le corresponde, es decir, se debe realizar el cast de los datos binarios al tipo del objeto destino (`(FichaLibro) librosEntrada.readObject()`).

5.3. Gestión de excepciones en ficheros

La manipulación de un fichero puede generar una serie de errores o excepciones, para gestionar este tipo de posibles errores; Java define una clase genérica (abstracta) llamada `IOException`. Esta clase dispone de un subconjunto de clases derivadas que representan errores concretos que pueden aparecer al manipular el contenido de un fichero. El siguiente esquema muestra la relación jerárquica de algunas de las clases de errores gestionados cuando se produce un error de entrada/salida.

```
java.lang.Object
├─java.lang.Throwable
│   └─java.lang.Exception
│       └─java.io.IOException
│           ├──java.io.FileNotFoundException
│           ├──java.io.InterruptedIOException
│           ├──java.io.ObjectStreamException
│           ├──java.io.OptionalDataException
│           ├──java.io.StreamCorruptedException
│           ├──java.io.UnsupportedEncodingException
│           ├──java.io.UTFDataFormatException
│           └─java.io.WriteAbortedException
```

Como se estudió en el Capítulo 2, cuando se crea un programa Java pueden capturarse y tratarse los errores aparecidos durante la ejecución del mismo, mediante un gestor de código `try/catch`, o simplemente puede generarse un objeto del tipo de error creado, y pasarlo a otro método para que sea éste el que finalmente lo trate (cláusula `throws`). Como puede verse en el anterior esquema, la mayoría de los errores básicos (como que un fichero no exista, o que se produzca un error de entrada/salida durante la operación sobre el mismo) están contemplados por el lenguaje. Debe recordarse que cuando se realizan determinadas operaciones sobre un fichero (como leer o escribir un dato) pueden originarse excepciones, por lo que si se desea que el programa funcione adecuadamente deberán ser capturados.

A lo largo del presente capítulo no se ha prestado demasiada atención al tratamiento de las excepciones que pueden aparecer al trabajar con ficheros. Los ejemplos mostrados han pretendido reflejar cómo deben ser manipuladas las clases que permiten construir este tipo de estructuras, por lo que se ha utilizado directamente la cláusula `throws IOException` que permite

pasar directamente la excepción a la máquina virtual de Java (dado que se incluye directamente en el `main()`). Sin embargo, no debe olvidarse que la implementación de un software robusto y tolerante a fallos requiere del tratamiento adecuado de aquellos errores habituales que pueden aparecer durante la ejecución del programa.

Por ejemplo, supongamos que se desea evitar el clásico problema que aparece cuando trata de abrirse un fichero que no existe (en particular cuando se trata de realizar una operación de lectura sobre el mismo). A continuación, se estudiarán varias aproximaciones que tratan de crear un programa que solicite y abra un fichero de forma robusta. La primera aproximación consiste en detectar la existencia o no del fichero, mostrar un mensaje de error y terminar.

```
import java.io.*;

public class ExcepcionFicherosGenerica{
    public static void main(String[] args){
        String ruta = new String(".");
        File fichTexto = new File(ruta,args[0]);
        //flujo de entrada....
        try{
            FileReader flujoEntrada = new FileReader(fichTexto);
            BufferedReader fEntrada = new BufferedReader(flujoEntrada);
            System.out.println("El fichero : "+fichTexto.getName()+"
                               abierto con exito");

            fEntrada.close();
        }catch(IOException e){
            System.out.println("El fichero : "+fichTexto.getName()+"
                               no existe");
        }
        //fin catch
    }
    //fin main
}
//fin ExcepcionFicheros
```

El ejemplo muestra que trata de abrirse un fichero que se lo pasa como primer argumento (`new File(ruta,args[0])`), en caso de que el fichero no existiese al tratar de establecer el flujo de entrada con `FileReader` se produciría un error que es capturado por el correspondiente gestor `catch`. Como puede verse, este ejemplo es equivalente a utilizar la cláusula `throws` en la declaración del `main()`, dado que únicamente se proporciona un mensaje genérico de error (`System.out.println("El fichero : "+fichTexto.getName()+"no existe")`) y el programa termina.

Sin embargo, puede proporcionarse un mayor control sobre este problema si en lugar de utilizar una excepción genérica se emplea la excepción `FileNotFoundException`. Además, y dado que se conoce el error exacto que se ha producido, podría volver a solicitarse de nuevo el fichero. Si se supone que ahora el nombre del fichero es solicitado por teclado, la siguiente aproximación muestra cómo se resolvería el problema de que un usuario introduzca un nombre erróneo de fichero. Si se observa detenidamente el código, ¿seríamos capaces de descubrir qué nuevo error puede originarse?

```

import java.io.*;
public class ExcepcionFicherosEspecifica {
    public static void main(String[] args){
        //Fichero
        String ruta = new String(".");
        String nomFich = new String();
        try{
            InputStreamReader tecEntrada =new InputStreamReader(System.in);
            BufferedReader teclado = new BufferedReader(tecEntrada);
            System.out.print("Dame el fichero a abrir: ");
            nomFich =teclado.readLine();
            File fichTexto = new File(ruta,nomFich);
            //se abre el fichero
            FileReader flujoEntrada = new FileReader(fichTexto);
            BufferedReader fEntrada = new BufferedReader(flujoEntrada);
            System.out.println("El fichero : "+fichTexto.getName()+"
                               abierto con exito");

            fEntrada.close();
        }catch(FileNotFoundException e){
            //error detectado
            System.out.println("El fichero : "+nomFich+" no existe");
            System.out.println(e);
            //volvemos a intentarlo
            try{
                InputStreamReader tecEntrada =new InputStreamReader(System.in);
                BufferedReader teclado = new BufferedReader(tecEntrada);
                //intento volver a pedir el nombre del fichero
                System.out.print("Dame OTRA VEZ... el fichero a abrir: ");
                File fichTexto = new File(ruta,teclado.readLine());
                FileReader flujoEntrada = new FileReader(fichTexto);
                System.out.println("ahora sí : "+fichTexto.getName());
                flujoEntrada.close();
            }catch(IOException e2){
                System.out.println("Error de E/S: "+e2.toString());
            }
        }catch(IOException e){
            System.out.println("Error de E/S: "+e.toString());
        }
    }
}
//fin main
}
//fin ExcepcionFicherosEspecifica

```

En la clase `ExcepcionFicherosEspecifica` se comprueba si el fichero introducido por teclado no existe, se le vuelve a solicitar al usuario; sin embargo, este ejemplo presenta varios problemas:

- En primer lugar es un código confuso, donde se han anidado sucesivamente bloques try/catch con lo que la legibilidad y su depuración se hace más complicada.
- En segundo lugar, se repite una gran cantidad de código, puesto que si se alcanza el bloque del catch hay que redefinir nuevamente los flujos (try interno).
- Finalmente, y mucho más importante, ¿qué sucedería si la persona vuelve a equivocarse? Efectivamente el código volvería a generar una excepción y se pararía.

Dado que el objetivo es encontrar una forma robusta de tratar este error en particular, no parece que el anterior ejemplo sea la forma idónea de solicitar de forma robusta un fichero. Este problema, utilizando adecuadamente las excepciones proporcionadas por Java, podría resolverse de la siguiente forma:

```
import java.io.*;

public class PedirFicheroRobusto {
    public static File ficheroCorrecto(String ruta, String nomFich){
        //se abre el flujo de lectura
        return new File(ruta,nomFich);
    }//fin ficheroCorrecto

    public static void main(String[] args){
        //Fichero
        String ruta = new String("."); //directorio actual
        String nomFich = "";
        boolean fichExiste = false; //salida del bucle
        File nombreFichero = null;
        try{
            //flujo de E/S desde teclado
            InputStreamReader tecEntrada =new InputStreamReader(System.in);
            BufferedReader teclado = new BufferedReader(tecEntrada);
            while(!fichExiste){
                System.out.print("Dame el fichero a abrir: ");
                nomFich =teclado.readLine();
                nombreFichero = ficheroCorrecto(ruta,nomFich);
                fichExiste = nombreFichero.exists();
                if (!fichExiste)
                    System.out.println(nomFich+" : NO existe");
            }
            //ahora se establece el flujo
            FileReader flujoEntrada = new FileReader(nombreFichero);
            BufferedReader fEntrada = new BufferedReader(flujoEntrada);
            System.out.println("El fichero : "+nombreFichero.getName()+"
                               ha sido abierto con éxito");
            fEntrada.close(); //se cierra el fichero
        }catch(IOException e){
            System.out.println("Error de E/S: "+e.toString());
        }
    }
}

//fin main
}
//fin PedirFicheroRobusto
```


La clase `PedirFicheroRobusto` permite solicitar un fichero de forma robusta hasta verificar la existencia del mismo. Dado que el método `main()` solicita por teclado el nombre del fichero, deberá controlarse el error de tipo `IOException` que podría producirse durante ese proceso. La forma más sencilla de controlar el hecho que un fichero exista, o no, puede resolverse mediante un método (se ha definido como estático para que pueda ser utilizado dentro de la propia clase) que devuelva una referencia al objeto `File` creado. Una vez obtenida la referencia puede utilizarse el método `exists()` (`nombreFichero.exists()`), que permite comprobar la existencia, o no, en memoria secundaria del fichero.

5.4. Algoritmos sobre ficheros

Los algoritmos de ordenación que se mostraron en el Capítulo 3 sirven para ordenar arrays, y se denominan **algoritmos de ordenación interna**. Los arrays se almacenan en memoria interna (o memoria principal), y son de acceso aleatorio, es decir, cada elemento es “visible” y accesible individualmente. Todos los algoritmos ordenan el array *in situ*, y lo consiguen permutando elementos. Como la memoria interna, o principal, es por lo general escasa, no puede desperdiciarse utilizando arrays auxiliares. Por lo que si el volumen de información es muy grande, posiblemente sea necesario otro tipo de algoritmos para ordenar su contenido.

Si el número de elementos contenidos en un fichero, y que se desea ordenar, es suficientemente pequeño podrían cargarse en memoria principal (por ejemplo, en un array) y ordenarlos utilizando alguno de los algoritmos de ordenación interna estudiados. Pero, en general, puede suponerse que el número de elementos a ordenar no está restringido por el tamaño de la memoria, por lo que se deberá utilizar uno o varios ficheros para almacenar los datos. Además, debe suponerse que el acceso a los datos debe realizarse de forma secuencial, por tanto, se tendrán que emplear métodos de ordenación totalmente diferentes. Los algoritmos utilizados para la ordenación de ficheros se denominan **algoritmos de ordenación externa**. Este tipo de algoritmos se basan en la **mezcla de secuencias ordenadas**.

Una secuencia ordenada de m elementos puede mezclarse con una secuencia ordenada de n elementos, formando una secuencia ordenada de $m + n$ elementos.

Ejemplo:

- **Secuencia a:** [3,7,9,12,45,67].
- **Secuencia b:** [7,23,25,34].
- **Secuencia resultante:** [3,7,7,9,12,23,25,34,45,67].

5.4.1. Algoritmo de mezcla directa

Este algoritmo consiste en dividir una secuencia inicial de datos en dos subcadenas y mezclar elemento a elemento de forma ordenada; el proceso se repite hasta lograr que la secuencia inicial

quede totalmente ordenada. El algoritmo de mezcla directa puede dividirse en el siguiente conjunto de pasos:

1. Se divide la secuencia inicial de datos contenidos en el fichero **a** en dos mitades **b** y **c**.
2. Se mezclan **b** y **c** combinando elementos aislados para formar *pares* ordenados.
3. La secuencia resultante se almacena en el fichero **a** y se repiten los pasos 1 y 2, mezclando los pares ordenados para formar *cuádruplos* ordenados.
4. Se repiten los pasos anteriores combinando los cuádruplos ordenados para formar *octetos* ordenados y se continúa así, duplicando en cada iteración el tamaño de las subsecuencias ordenadas hasta que la secuencia total quede ordenada.

A continuación, se muestra un ejemplo práctico del funcionamiento del algoritmo de mezcla directa, se dispondrá de un fichero inicial **a** que almacena el conjunto de datos, y de dos ficheros auxiliares **b** y **c** utilizados para realizar el proceso de mezcla (se ha utilizado el símbolo ' para indicar el final de una subcadena ordenada).

Situación inicial:

fichero a: [44, 55, 12, 42, 94, 18, 6, 67]

fichero b: []

fichero c: []

1ª división:

fichero b: [44, 55, 12, 42]

fichero c: [94, 18, 6, 67]

1ª mezcla:

fichero a: [44, 94', 18, 55', 6, 12', 42, 67']

2ª división:

fichero b: [44, 94', 18, 55']

fichero c: [6, 12', 42, 67']

2ª mezcla:

fichero a: [6, 12, 44, 94', 18, 42, 55, 67']

3ª división:

fichero b: [6, 12, 44, 94']

fichero c: [18, 42, 55, 67']

3ª mezcla:

fichero a: [6, 12, 18, 42, 44, 55, 67, 97]

Se han necesitado tres pasadas, cada una con una fase de división y una de mezcla, para lograr ordenar la secuencia de números. Las fases de división no contribuyen a la ordenación, porque

no permutan elementos, pero constituyen la mitad de las operaciones de copiado. Estas fases de división pueden ser eliminadas si se mezclan directamente sobre los dos ficheros, es decir, el resultado de la mezcla se distribuye sobre dos ficheros, que serán la base de la pasada siguiente. Este algoritmo es conocido por el nombre de método de mezcla directa con fase única o mezcla compensada. El número de pasos (divisiones/mezclas) de este algoritmo dependerá del número de elementos a ordenar (en cada mezcla se ordenan el doble de elementos que en el anterior).

5.4.2. Algoritmo de mezcla natural

Uno de los algoritmos más sencillos es el **método de mezcla natural**, que consiste en realizar continuamente **distribuciones y mezclas** hasta obtener solamente una secuencia ordenada.

La siguiente figura muestra un esquema de cómo se realiza el método de mezcla natural:

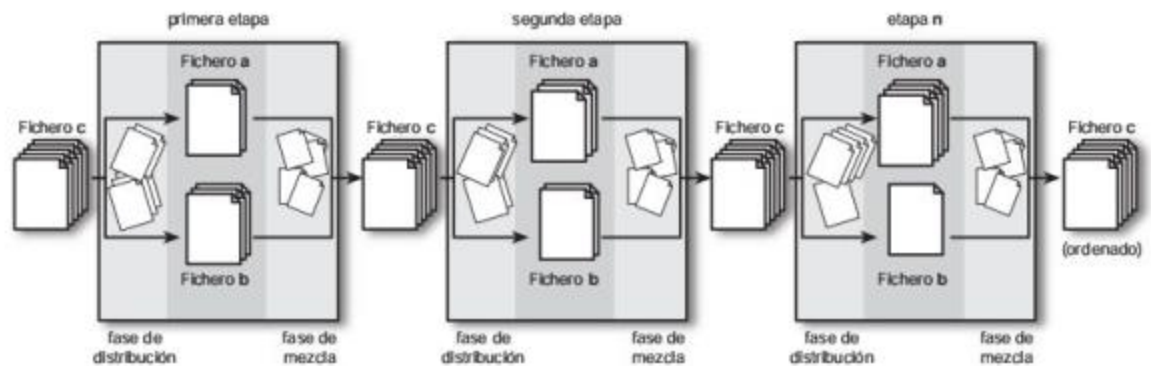


Figura 5.6. Etapas en el proceso de ordenación por mezcla natural.

Como puede verse en la figura anterior, el algoritmo de mezcla natural es un algoritmo iterativo que utiliza dos ficheros como estructuras auxiliares donde almacenar cada una de las partes del fichero origen que trata de ordenarse. A continuación, se explicará en detalle este algoritmo empleando varios ejemplos que permiten comprender cómo se realizan cada uno de los dos procesos de los que consta la mezcla natural.

Distribución

Supongamos que se desea almacenar la información almacenada en un fichero **c**. El método de mezcla natural utilizará otros dos ficheros **a** y **b** para realizar diversas distribuciones y mezclas hasta lograr ordenar los datos. La **distribución** consiste en repartir la secuencia original (almacenada en el fichero **c**) en dos secuencias, de forma que se pasarán aquellas **subsecuencias ordenadas (o tramos) de longitud máxima** alternativamente desde el fichero **c** al fichero **a** o al fichero **b**.

Ejemplo: supondremos que se parte de un fichero **c** que contiene números enteros. En la siguiente secuencia se muestran los valores almacenados en el mismo (los ficheros **a** y **b** comienzan vacíos):

fichero original c: [1,5,7,4,9,2,5,10,67,50,100]

fichero a: []

fichero b: []

El proceso de mezcla natural comenzará con la distribución de las subcadenas, por tanto, se pasará la subsecuencia ordenada "**1, 5, 7**" al fichero **a**, la subsecuencia "**4, 9**" al fichero **b**, la "**2, 5, 10, 67**" al fichero **a**, y la "**50, 100**" al fichero **b** (el proceso se continúa sucesivamente hasta alcanzar el final del fichero **c**). Una vez movidas las diferentes subsecuencias a cada uno de los dos ficheros, éstos almacenarían los siguientes datos:

fichero a: [1,5,7*, 2,5,10,67]

fichero b: [4,9, 50,100*]

Debe observarse que dos subsecuencias distintas pueden formar una sola cuando el primer elemento de la segunda es mayor que el último de la primera. Se ha utilizado el símbolo comilla (*), para indicar el final de la subsecuencia ordenada.

Mezcla

Una vez realizada la primera distribución es necesario **mezclar** las subsecuencias ordenadas de **a** y **b** dejando el resultado en **c**, de esta forma a partir de dos subsecuencias ordenadas se obtendrá una única secuencia.

Para el ejemplo anterior, y una vez realizado el proceso de distribución, se obtienen dos ficheros que contienen los siguientes datos:

fichero a: [1,5,7*,2,5,10,67*]

fichero b: [4,9, 50,100*]

Si se mezclan las subcadenas o tramos ordenados: "1, 5, 7" con la "4, 9, 50, 100", se obtendrá el nuevo tramo: "1, 4, 5, 7, 9, 50, 100" en el fichero **c**. Dado que todavía queda una subcadena en **a** (**b** ha utilizado todos sus datos en la primera mezcla), sólo deberá copiarse la subcadena "2, 5, 10, 67" en **c**, por lo que finalmente, y una vez realizado todo el proceso de mezcla (es decir, hasta que no quedan más datos en **a** o en **b**), el fichero **c** almacenará:

fichero c: [1,4,5,7,9,50,100*, 2,5,10,67*]

Es importante observar que en el fichero **c** existen dos subcadenas ordenadas identificadas gracias al símbolo comilla. A continuación, deberá realizarse una nueva distribución, y posteriormente una última mezcla, que logrará obtener el fichero ordenado en el fichero **c**. La secuencia de esta nueva distribución/mezcla sería:

Distribución 2:

fichero c: [1,4,5,7,9,50,100*, 2,5,10,67*]

fichero a: [1,4,5,7,9,50,100*]

fichero b: [2,5,10,67*]

Mezcla 2:

fichero c: [1,2,4,5,5,7,9,10,50,67,100]

fichero a: []

fichero b: []

A continuación, y a modo de resumen, se muestra un segundo ejemplo sobre una cadena de datos mayor donde aparecen intercaladas las diferentes operaciones de distribución y mezcla realizadas sobre los ficheros.

Secuencia inicial:

fichero c: [17 31 5 59 13 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61]

- Distribución:

fichero a: [17 31* 13 41 43 67* 3 7 71* 37 61*]

fichero b: [5 59* 11 23 29 47* 2 19 57*]

- Mezcla:

fichero c: [5 17 31 59* 11 13 23 29 41 43 47 67* 2 3 7 19 57 71* 37 61*]

- Distribución:

fichero a: [5 17 31 59* 2 3 7 19 57 71*]

fichero b: [11 13 23 29 41 43 47 67* 37 61*]

- Mezcla:

fichero c: [5 11 13 17 23 29 31 41 43 47 59 67* 2 3 7 19 37 57 61 71*]

- Distribución:

fichero a: [5 11 13 17 23 29 31 41 43 47 59 67*]

fichero b: [2 3 7 19 37 57 61 71*]

- Mezcla:

fichero c: [**2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 57 59 61 67 71**]

Como puede verse en el ejemplo resumen, el proceso termina cuando el número de tramos (subcadenas ordenadas) de **c** es 1. Como condición, se supone que al menos debe existir un tramo no vacío en el fichero inicial.

5.5. Ejercicios resueltos

Este apartado muestra una colección de ejercicios que se han dividido en dos tipos claramente diferenciados:

- Los primeros ejercicios (ejercicios básicos) consisten en una recopilación de casos prácticos muy utilizados cuando se trabaja sobre ficheros. Es decir, en muchas situaciones no será necesario realizar un estudio muy exhaustivo de las clases de `java.io`, sino que por el con-

trario con la siguiente colección de ejercicios resueltos se pretende mostrar al programador cómo se resuelven los problemas más habituales en el tratamiento de ficheros, pudiendo analizarse el API del paquete de E/S si se necesitase de otras clases más específicas.

- Los últimos ejercicios mostrarán cómo utilizar las clases de `java.io` para poder implementar algoritmos más complejos (ejercicios avanzados).

5.5.1. Ejercicios básicos sobre ficheros

Ejercicio 5.1. Lectura de ficheros de texto

Enunciado Implementar un programa en Java que permita leer el contenido de un fichero de texto y lo muestre por la salida estándar (monitor).

Solución El siguiente programa muestra cómo leer ficheros línea a línea de un fichero de texto para posteriormente mostrarlos por pantalla.

```
import java.io.*;

public class LeerLineasTexto {

    public static void main(String[] args) throws IOException{
        BufferedReader fEntrada = new BufferedReader(
            new FileReader(
                "c:\\java\\ejs\\cap5\\ficheros\\fich3.txt"));

        String cadena = new String();
        while ((cadena=fEntrada.readLine()) != null)
            System.out.println(cadena);
        fEntrada.close(); //se cierra el fichero
    } //fin main
} //fin LeerLineasTexto
```

Como puede verse en este ejemplo, se ha simplificado la declaración del fichero y al constructor del objeto `BufferedReader` se le pasa directamente el `FileReader`. Es interesante ver que en este ejemplo se utiliza directamente un `String` para indicar el lugar físico donde se encuentra el fichero (en lugar de utilizar un objeto de tipo `File`).

Ejercicio 5.2. Escritura de ficheros de texto

Enunciado Implementar un programa en Java que permita implementar la operación de copia de un fichero:

```
c:\> copy origen.txt destino.txt
```

donde tanto los ficheros `origen.txt` y `destino.txt` serán de tipo texto. Suponer que el nombre de ambos ficheros se proporciona al programa como parámetros.

```
c:\> java CopiarFicherosTexto origen.txt destino.txt
```

Solución El siguiente ejemplo muestra cómo pueden escribirse líneas desde un fichero de texto a otro (para ello se utilizará el ejercicio anterior). En este ejemplo se pasan dos parámetros desde línea de comandos, `args[0]` que representa el fichero origen, y `args[1]` que indica el fichero destino.

```
import java.io.*;

public class CopiarFicherosTexto {
    public static void main(String[] args) throws IOException{
        //args[0]: fichero origen
        //args[1]: fichero destino
        //flujo de entrada
        BufferedReader fEntrada = new BufferedReader(new FileReader(args[0]));
        //flujo de salida
        PrintWriter fSalida =
            new PrintWriter(new BufferedWriter(new FileWriter(args[1])));
        String linea = new String();
        while ((linea=fEntrada.readLine()) != null)
            fSalida.println(linea);
        fSalida.close(); //se cierra el fichero destino
        fEntrada.close(); //se cierra el fichero origen
    } //fin main
} //fin CopiarFicherosTexto
```

Ejercicio 5.3. Escritura de ficheros con tipo

Enunciado Implementar un programa en Java que permita escribir en un fichero con tipo un conjunto de atributos relativos a una persona. Los atributos que se almacenarán para cada persona estarán formados por:

- Dos campos `String`: nombre, apellidos.
- Un campo `int`: edad.
- Un campo `char`: sexo.

Solución En el siguiente ejemplo se utilizan las clases estudiadas anteriormente para guardar datos en un fichero. Dado que debe emplearse un fichero con tipo (binario) se utilizará la clase `DataOutputStream` y sus flujos asociados, una posible solución a este problema vendría dada por el siguiente programa:

```
import java.io.*;

public class EscribirPersonas {
    public static void main(String[] args) throws IOException{
        DataOutputStream fichSalida = new DataOutputStream(
            new FileOutputStream(
```

```

        "c:\\java\\ej5\\cap5\\ficheros\\
        Personas.dat"));
    // se escriben algunos valores en el fichero
    fichSalida.writeUTF("José");
    fichSalida.writeUTF("González Munoz");
    fichSalida.writeInt(23);
    fichSalida.writeChar((int)'v');
    fichSalida.writeUTF("Sonia");
    fichSalida.writeUTF("Mejías Rubio");
    fichSalida.writeInt(30);
    fichSalida.writeChar((int)'m');
    fichSalida.close(); //se cierra el fichero
} //fin main
} // fin EscribirPersonas

```

En el caso de que el fichero no exista, el constructor de la clase `DataOutputStream` creará uno nuevo. Es importante tener en cuenta que si el fichero ya existe sobrescribirá los datos.

Ejercicio 5.4. Lectura de ficheros con tipo

Enunciado Implementar un programa en Java que permita leer los diferentes registros creados y almacenados en el ejercicio anterior. Debe recordarse, que cada registro estará formado por dos campos `String` (de longitud variable), un atributo de tipo entero, y otro de tipo carácter.

Solución A partir de la solución mostrada en el ejercicio anterior, pueden leerse los datos primitivos almacenados en el fichero utilizando el flujo de datos de entrada definido en Java (`DataInputStream`).

```

import java.io.*;

public class LeerPersonas {
    public static void main(String[] args){
        try{
            DataInputStream fichEntrada =
                new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream(
                            "c:\\java\\ej5\\cap5\\ficheros\\Personas.dat")));

            //leer datos
            System.out.println(fichEntrada.readUTF());
            System.out.println(fichEntrada.readUTF());
            System.out.println(fichEntrada.readInt());
            System.out.println(fichEntrada.readChar());
            //segunda persona
            System.out.println(fichEntrada.readUTF());
            System.out.println(fichEntrada.readUTF());
            System.out.println(fichEntrada.readInt());

```



```

        System.out.println(fichEntrada.readChar());
        fichEntrada.close(); //se cierra el fichero
    } catch (FileNotFoundException e) {
        System.out.println("EL fichero no se encuentra");
    } catch (EOFException e) {
        System.out.println("Fin de fichero");
    } catch (IOException e) {
        System.out.println("Errores de E/S");
    }
}
} //fin main
} //fin LeerPersonas

```

Como se muestra en el ejemplo, se ha optado por la utilización de cláusulas `try/catch` que permiten lograr un mayor control sobre la gestión de errores. En este ejemplo se muestran dos de las excepciones más habituales al trabajar con un fichero, `FileNotFoundException` que se origina si la ruta indicada para el fichero no es correcta, y `EOFException` que se genera ante una terminación inesperada del fichero.

Ejercicio 5.5. Lectura/escritura de objetos en ficheros

Enunciado Implementar un programa en Java que sea capaz de almacenar (tanto para leer como para escribir) objetos en un fichero de datos. El programa almacenará los datos básicos de una persona; éstos son:

- **Nombre:** String.
- **Apellidos:** String.
- **Edad:** entero.
- **Sexo:** carácter (v,m).

Solución Supongamos que en lugar de almacenar tipos básicos se desean almacenar objetos que representan a las personas (este ejemplo es una simplificación del mostrado en el apartado sobre serialización). En ese caso, y como se estudió previamente, se deben utilizar las clases `ObjectInputStream` y `ObjectOutputStream`. Como puede verse en la solución mostrada, en primer lugar se ha definido la clase que permitirá construir los objetos persona (`PersonaSimple`), que posteriormente serán escritos o leídos del fichero `ObjetosPersona.dat`.

```

import java.io.*;

public class PersonaSimple implements Serializable{
    private String nombre, apellidos;
    private int edad;
    private char sexo;

    public PersonaSimple (String nomb, String apell, int ed, char sex) {
        this.nombre = nomb;
        this.apellidos = apell;
    }
}

```

```

        this.edad = ed;
        this.sexo = sex;
    } //fin constructor
    public void mostrarInfo() {
        System.out.println("Nombre: "+this.nombre);
        System.out.println("Apellidos: "+this.apellidos);
        System.out.println("Edad: "+this.edad);
        System.out.println("Sexo: "+this.sexo);
    } //fin mostrarInfo
} //fin Persona Simple

```

Una vez implementada la clase que permitirá construir diferentes personas, será necesario crear la clase que permite leer y escribir esos objetos que como puede verse en la declaración de la clase `PersonaSimple`, deberán implementar la interfaz `Serializable`.

```

import java.io.*;
public class LeerEscribirObjetosPersona{
    public static void main(String[] args) throws IOException,
                                                    ClassNotFoundException{
        //Escribir en el fichero
        ObjectOutputStream fichSalida =
            new ObjectOutputStream(
                new FileOutputStream(
                    "c:\\java\\ej5\\cap5\\ficheros\\ObjetosPersona.dat"));
        //se crean las personas1
        PersonaSimple p1 = new PersonaSimple("José", "González Muñoz", 23, 'v');
        PersonaSimple p2 = new PersonaSimple("Sonia", "Mejias Rubio", 30, 'm');
        //se almacenan
        fichSalida.writeObject(p1);
        fichSalida.writeObject(p2);
        fichSalida.close();
        //Leer del fichero
        ObjectInputStream fichEntrada =
            new ObjectInputStream(
                new FileInputStream(
                    "c:\\java\\ej5\\cap5\\ficheros\\ObjetosPersona.dat"));
        //leer datos
        PersonaSimple ob1 = (PersonaSimple) fichEntrada.readObject();
        PersonaSimple ob2 = (PersonaSimple) fichEntrada.readObject();
        ob1.mostrarInfo();
        ob2.mostrarInfo();
        fichEntrada.close();
    } //fin main
} //fin LeerEscribirObjetosPersona

```

Es importante recordar que cuando se trata con objetos serializados, al intentar reconstruirlos será necesario realizar un **cast** a la clase destino, por lo que es obligatorio (sólo en el caso de la lectura) capturar la excepción: `ClassNotFoundException`.

Ejercicio 5.6. Lectura/escritura en ficheros de acceso aleatorio

Enunciado Utilizando el ejercicio anterior, implementar un programa que permita la lectura y escritura de personas sobre un fichero de acceso aleatorio. Se reutilizará la clase `PersonaSimple`, para construir objetos de tipo persona que posteriormente serán accesibles directamente.

Solución Si se estudia el API de la clase `RandomAccessFile` se puede comprobar que tanto los métodos de lectura como los de escritura se han diseñado para tipos básicos de Java, luego no será posible mantener un fichero con acceso directo sobre un conjunto de objetos. Sin embargo, si se considera el Ejercicio 5.3., donde una persona es almacenada en el fichero como una colección de datos simples, puede crearse la clase `PersonasAccesoDirecto` que permite construir un fichero donde el acceso a los datos de cada persona se realizaría de forma directa (o aleatoria).

```
import java.io.*;

public class PersonasAccesoDirecto {
    public static void main(String[] args) throws IOException{
        RandomAccessFile fichAleatorio =
            new RandomAccessFile(
                "c:\\java\\ej5\\cap5\\ficheros\\DirectPersonas.dat", "rw");

        //se crean dos personas
        PersonaSimple p1 = new PersonaSimple("jose", "gonzalez garcia", 19, 'v');
        PersonaSimple p2 = new PersonaSimple("sonia", "mejias rubio", 30, 'm');
        fichAleatorio.seek(fichAleatorio.length()); //final del fichero
        //salvar datos en el fichero
        fichAleatorio.writeUTF(p1.getNombre());
        fichAleatorio.writeUTF(p1.getApellidos());
        fichAleatorio.writeInt(p1.getEdad());
        fichAleatorio.writeChar(p1.getSexo());
        fichAleatorio.writeUTF(p2.getNombre());
        fichAleatorio.writeUTF(p2.getApellidos());
        fichAleatorio.writeInt(p2.getEdad());
        fichAleatorio.writeChar(p2.getSexo());
        //ahora se trabaja en modo lectura
        long longitud=fichAleatorio.length();
        System.out.println("Nº de bytes almacenados: "+longitud);
        //desplazamientos primera persona
        int offset1 = 0; //nombre
        int offset2 = offset1+p1.tamNombre()+2; //Formato UTF, apellidos
        int offset3 = offset2+p1.tamApellidos()+2; //Edad
        int offset4 = offset3+p1.tamEdad(); //Sexo
        //desplazamientos segunda persona
```



```

    int offseta = offset4+2; //nombre, FORMATO UTF
    int offsetb = offseta+p1.tamNombre()+2; //Formato UTF, primer apellido
    int offsetc = offsetb+p1.tamApellidos(); //Edad
    int offsetd = offsetc+p1.tamEdad(); //Sexo
    //acceso directo a diferentes campos, en cualquier orden
    fichAleatorio.seek(offset4);
    System.out.println("Primera persona, sexo: "+fichAleatorio.readChar());
    fichAleatorio.seek(offset2);
    System.out.println("Primera persona, apell:"+fichAleatorio.readUTF());
    fichAleatorio.seek(offseta);
    System.out.println("Segunda persona, nomb: "+fichAleatorio.readUTF());
    fichAleatorio.seek(offsetc);
    System.out.println("Segunda persona, edad: "+fichAleatorio.readInt());
    fichAleatorio.seek(offset1);
    System.out.println("Primera persona, nomb: "+fichAleatorio.readUTF());
    fichAleatorio.seek(offsetd);
    System.out.println("Segunda persona, sexo: "+fichAleatorio.readChar());
    fichAleatorio.close();
} //fin main
} //fin PersonasAccesoDirecto

```

Como puede observarse en la solución, se han añadido nuevos métodos que permiten obtener información de los atributos de una persona como `getNombre()`, `getApellidos()`, `getEdad()`, o `getSexo()`, otros métodos como `tamNombre()`, `tamApellidos()`, `tamEdad()`, o `tamSexo()`, son necesarios para poder determinar cuantos bytes ocupan los diferentes campos almacenados en el fichero.

La clase `PersonasAccesoDirecto` muestra, en primer lugar, la declaración del fichero que permite realizar el acceso directo a la información (`new RandomAccessFile(nombreFichero, "rw")`). A continuación, son declarados dos objetos de tipo `PersonaSimple` (`new PersonaSimple(nombre, apellidos, edad, sexo)`), para pasar a almacenar la información de cada persona de forma ordenada en el fichero.

Sin embargo, acceder a la información es algo más complejo. Para leer la información directamente se necesita situar el puntero de lectura/escritura en la posición adecuada, esto se logrará utilizando el método `seek(posicion)`; el problema surge cuando se necesita calcular el valor de esa posición, dado que se han utilizado atributos de tipo `String` (de longitud variable) para almacenar los nombres y apellidos de las personas, y además se ha utilizado el método `writeUTF(String p)` para almacenar esas cadenas. Es necesario conocer cómo se almacenan exactamente los datos si se desea posicionar correctamente el puntero de lectura/escritura.

Cuando se utiliza el método `writeUTF(String p)`; debe conocerse que todos los caracteres son almacenados y representados por un único byte (UTF-8), y que por cada cadena almacenada se reservan los dos primeros bytes para indicar la longitud (en bytes) utilizada para almacenar el `String`. Esto significa, que cuando se desee leer un `String` almacenado con formato

UTF (*Universal Text Format*) el desplazamiento debe considerar esos **2 bytes extras** empleados por el formato de codificación de caracteres utilizado.

Si se observa la solución, cuando se calculan los diferentes desplazamientos (necesarios para poder acceder directamente a los diferentes campos almacenados), los parámetros `offset2`, `offset3`, `offseta` y `offsetb` tienen un incremento extra de 2 bytes debido a la utilización del formato UTF. Una vez los desplazamientos están correctamente calculados, es posible acceder a los campos deseados en cualquier orden.

5.5.2. Ejercicios avanzados sobre ficheros

Ejercicio 5.7. Cifrado de un fichero

Enunciado Implementar una clase en Java que permita cifrar de forma sencilla un fichero de texto. El programa obtendrá de los parámetros de entrada el fichero a cifrar (`origen.txt`), y el fichero destino (`cifrado.txt`). Una vez establecidos ambos flujos, solicitará por teclado un valor numérico (comprendido entre 1...255) que utilizará para realizar el proceso de cifrado. Este proceso consistirá en recorrer carácter a carácter el fichero origen y calcular la O-exclusiva (XOR) con el valor numérico proporcionado por el usuario, el carácter resultante será volcado en el fichero cifrado.

Solución Este ejercicio necesita establecer tres flujos, dos contra ficheros (`in` de tipo `BufferedInputStream`, y `out` de tipo `BufferedOutputStream`) que serán utilizados para leer caracteres del fichero origen y escribir el resultado del cifrado en el fichero destino, y un tercer flujo que será utilizado para permitir la lectura del dato suministrado por el usuario desde la consola. El siguiente ejemplo muestra una posible solución.

```
import java.io.*;

class CifrarSimple{
    public static void main(String args[])throws IOException{
        int b,clave;
        if (args.length != 2){
            System.out.println("Error en el nº de argumentos");
            return;
        }
        BufferedInputStream in = new BufferedInputStream (
            new FileInputStream(args[0]));
        BufferedOutputStream out = new BufferedOutputStream (
            new FileOutputStream(args[1]));
        BufferedReader br= new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Teclee la clave:(1..255) ");
        clave = Integer.parseInt(br.readLine());
        System.out.println("Cifrando fichero "+args[0]+ " en fichero "+args[1]);
        do{
            b=in.read(); //se lee un caracter
```

```

        if (b!=-1)
            out.write(b^clave); //cifrar
    }while (b!=-1);
    in.close();
    out.close();
} //fin main
} //fin CifrarSimple

```

Como puede verse en el anterior programa, el proceso de cifrado es realizado simplemente calculando (a nivel de byte) la XOR (^) con la clave proporcionada (`out.write(b^clave)`), una vez realizada la operación el carácter es almacenado en el fichero destino.

Ejercicio 5.8. Mezcla de ficheros ordenados

Enunciado Implementar una programa en Java que permita mezclar de forma ordenada dos ficheros de texto. Suponer que los ficheros de texto almacenan datos que pueden ser transformados a un formato numérico (un dato por línea), y que estos datos están ordenados. Por ejemplo, para los ficheros "uno.txt" y "dos.txt" que se muestran a continuación, debería obtenerse como resultado el fichero "salida.txt".

Uno.txt	Dos.txt	Salida.txt
2	1	1
4	3	2
9	9	3
9	11	4
13	15	9
22	15	9
25	80	11
51	100	13
82	101	

Solución Para resolver el problema anterior únicamente es necesario establecer tres flujos, dos de lectura que permitan leer los ficheros de datos, y otro de salida que permita almacenar los resultados. A continuación, se muestra una primera aproximación que permitiría resolver el problema de la mezcla de ficheros ordenados.

```

import java.io.*;
class FusionA{

```

```
public static void main(String[] args) throws IOException {
    String ruta = new String("c:\\\\java\\ejs\\cap5\\ficheros\\\\");
    BufferedReader fUno=new BufferedReader(
        new FileReader(ruta+"uno.txt"));
    BufferedReader fDos=new BufferedReader(new FileReader(ruta+"dos.txt"));
    PrintWriter fSalida=new PrintWriter(
        new BufferedWriter(new FileWriter(ruta+"salida.txt")));
    String cadUno,cadDos=new String();
    int enteroUno=0,enteroDos=0;
    boolean eofUno,eofDos=false;
    if (!(eofUno=((cadUno=fUno.readLine())==null)))
        enteroUno=Integer.parseInt(cadUno);
    if (!(eofDos=((cadDos=fDos.readLine())==null)))
        enteroDos=Integer.parseInt(cadDos);
    // Mientras no se acaben los ficheros, escribimos es valor
    // menor o igual y leemos del fichero que hemos escrito
    while (!eofUno && !eofDos) {
        if (enteroUno <= enteroDos) {
            fSalida.println(enteroUno);
            if (!(eofUno=((cadUno=fUno.readLine())==null)))
                enteroUno=Integer.parseInt(cadUno);
        }
        else {
            fSalida.println(enteroDos);
            if (!(eofDos=((cadDos=fDos.readLine())==null)))
                enteroDos=Integer.parseInt(cadDos);
        }
    }
    // Escribimos el resto del fichero que no ha provocado la
    // salida del bucle.
    if (eofUno)
        while(!eofDos) {
            fSalida.println(enteroDos);
            if (!(eofDos=((cadDos=fDos.readLine())==null)))
                enteroDos=Integer.parseInt(cadDos);
        }
    else
        while(!eofUno) {
            fSalida.println(enteroUno);
            if (!(eofUno=((cadUno=fUno.readLine())==null)))
                enteroUno=Integer.parseInt(cadUno);
        }
    //cerramos los ficheros
    fUno.close();
    fDos.close();
}
```

```

        fSalida.close();
    } //fin main
} //fin FusionA

```

Puede surgir un problema con el anterior algoritmo si los ficheros de texto almacenan algún retorno de carro extra después del último número almacenado. En ese caso, el programa trataría de transformar el carácter especial a un entero produciendo un error de formato numérico (`NumberFormatException`). Básicamente, el algoritmo consiste en recorrer cada uno de los ficheros (que como precondition se ha impuesto que estén ordenados), leyendo un único valor de cada uno de ellos y escribiendo en el fichero destino en primer lugar el de menor o igual valor. El proceso continúa hasta que uno de los dos ficheros termina; cuando esto sucede simplemente se recorre el resto del fichero que queda y se copia en el fichero destino. Una forma más simplificada del mismo algoritmo podría implementarse de la siguiente forma:

```

import java.io.*;

public class FusionB {
    public static void main (String args[]) throws IOException{
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        BufferedReader fUno = new BufferedReader(
            new FileReader(ruta+"uno.txt"));
        BufferedReader fDos = new BufferedReader(
            new FileReader(ruta+"dos.txt"));
        PrintWriter fSal = new PrintWriter(new BufferedWriter(
            new FileWriter(ruta+"salida.txt")));

        String cad1, cad2;
        int num=0;
        int int1=0, int2=0;
        cad1=fUno.readLine();
        cad2=fDos.readLine();
        while (cad1!=null && cad2!=null){
            int1=Integer.parseInt(cad1);
            int2=Integer.parseInt(cad2);
            if (int1<=int2){
                fSal.println(int1);
                cad1=fUno.readLine();
            }else{
                fSal.println(int2);
                cad2=fDos.readLine();
            }
        }
        while(cad1!=null){
            int1=Integer.parseInt(cad1);
            fSal.println(int1);
            cad1=fUno.readLine();
        }
        while(cad2!=null){

```



```

        int2=Integer.parseInt(cad2);
        fSal.println(int2);
        cad2=fDos.readLine();
    }
    fUno.close();
    fDos.close();
    fSal.close();
    }//fin main
} //fin FusionB

```

Ejercicio 5.9. Lectura robusta de datos con formato desde un fichero de texto

Enunciado Implementar un programa en Java (*LecturaFichero*) que permita leer de forma robusta un conjunto de datos con formato de un fichero de texto. Se controlarán los errores habituales como que el fichero no exista, o que se encuentre vacío. Además, se deberá implementar una clase (*DatosFicheroIncorrectos*) que en el caso de que alguna de las líneas no verifiquen el formato adecuado indique la posición del error. El fichero almacenará todos sus datos en una única línea de texto, donde los diferentes valores aparecen separados por algún carácter especial como un espacio en blanco o un tabulador:

fichero: [1 231 24 42 1243 321 431 24 1 2 3 4 5 6 7 8 0 -2 -5 34]

La clase a implementar, *LecturaFichero*, deberá leer los diferentes valores y almacenarlos en un array (una vez sean transformados a enteros). Por ejemplo, para el siguiente fichero:

fichero: [1 231 24 42 1243 ~~3666~~ 431 24 1 2 3 4 5 6 7 8 0 -2 -5 34]

Deberá lanzarse una excepción indicando que el dato que se encuentra en la posición 5 del fichero está en un formato inadecuado. El usuario podrá entonces cambiar el dato erróneo, salvarlo y continuar el proceso de transformación como si nada hubiese sucedido.

Nota: se aconseja revisar y estudiar la clase `java.util.StringTokenizer`.

Solución El siguiente ejemplo muestra cómo pueden utilizarse las excepciones para leer datos en un determinado formato, de forma robusta, de un fichero. Este ejemplo, permite al usuario editar y modificar el fichero hasta que éste sea aceptado por la aplicación, sin necesidad de detener la ejecución del programa.

En primer lugar, se ha creado una clase excepción propia, *DatosFicheroIncorrectos* que se lanzará cuando aparezcan errores en el acceso a un fichero de texto con datos. Esta excepción permite obtener un conjunto de detalles acerca del error concreto que generó la excepción. Esta información será utilizada por el programa para tratar adecuadamente el error.

Como estado interno de la excepción se indica si el fichero está vacío o no (`private boolean ficheroVacio`). En el caso de que un error se produzca se indicará la línea del fichero


```

int [] a;
FileReader fr = new FileReader(nombre);
BufferedReader br=new BufferedReader(fr);
String linea=null;
linea=br.readLine();
int k=0;
try{
    StringTokenizer st=new StringTokenizer(linea);
    int dim=st.countTokens();
    a=new int[dim];
    for (k=0;k<dim;k++)
        a[k]=Integer.parseInt(st.nextToken());
}catch (NullPointerException e){
    throw new DatosFicheroIncorrectos(true);
}catch (NumberFormatException e){
    throw new DatosFicheroIncorrectos(k);
}finally{
    br.close();
}
return a;
} //fin leerVector
} //fin LecturaFichero

```

Finalmente, el siguiente ejemplo muestra cómo puede utilizarse la clase anterior `LecturaFichero`. Así como la utilización de las excepciones propias y estándares para gestionar las posibles situaciones de error que se pudiesen producir. El siguiente programa (`PruebaFichero`) utiliza un bucle en el que intenta completar la operación de leer un vector de enteros de fichero hasta que no haya error. En caso de existir algún error en el fichero de datos, el programa indica al usuario qué problema específico ha detectado. Debe observarse cómo en el caso de que el fichero tenga un problema en el formato de una línea se indica explícitamente la posición donde debe realizarse la corrección del dato.

```

class PruebaFichero{
    public static void main(String[] args) throws IOException{
        LecturaFichero lf=new LecturaFichero();
        int [] a;
        boolean exito=false;
        while (!exito){
            try{
                a=lf.leerVector(args[0]);
                for (int k=0;k<a.length;k++){
                    System.out.print("a["+k+"]="+a[k]+" ");
                }
            }
        }
    }
}

```


Ejemplo ficheroDatos.txt:

```
1 231 24 42 1243
321    431    24    1
2 3 4 5 6 7
8      0      -2      -5      34
```

Nota: deberán detectarse los mismos errores que en el ejercicio anterior.

Solución La clase de excepciones propia, `DatosFicheroIncorrectos`, que fue generada en el anterior ejercicio puede reutilizarse completamente en este nuevo ejercicio. Dado que en este problema los datos estarán almacenados en varias líneas de texto, se ha modificado el método `leerVector2()` para permitir que lance una nueva excepción de tipo: `EOFException`. Esto se hace para detectar el final del fichero (`linea == null`), y de esa forma provocar el final del programa. Además, se ha modificado el método para que acepte el flujo de entrada del fichero (en lugar del `String` correspondiente a su nombre como se hacía en el ejercicio anterior).

```
class LecturaFichero2{
    public int[] leerVector2(BufferedReader br) throws FileNotFoundException,
                                                    IOException,
                                                    DatosFicheroIncorrectos,
                                                    EOFException {

        int [] a;
        String linea=null;
        linea=br.readLine();
        int k=0;
        try{
            if (linea == null){
                throw new EOFException ("fin de fichero...");
            }else {
                StringTokenizer st=new StringTokenizer(linea);
                int dim=st.countTokens();
                a=new int[dim];
                for (k=0;k<dim;k++)
                    a[k]=Integer.parseInt(st.nextToken());
            }
        }catch (NullPointerException e){
            System.out.println("error de formato");
            throw new DatosFicheroIncorrectos(true);
        }catch (NumberFormatException e){
            throw new DatosFicheroIncorrectos(k);
        }
    }
}
```

```

    }catch (EOFException e){
        throw e; //se relanza la excepción
    }
    return a;
} //leerVector
}

```

La clase de prueba es ahora la encargada de abrir y cerrar el flujo del fichero. Es interesante observar cómo ahora el bucle de lectura del fichero es interno al `try`, y la variable booleana "exito" nunca es puesta a `true`, luego la única forma de terminar este bucle es mediante una excepción que detecte el final del fichero.

```

class PruebaFichero3{
    public static void main(String[] args) throws IOException {
        String ruta = new String("c:\\\\java\\ejs\\cap5\\ficheros\\");
        String fichero = ruta + args[0];
        LecturaFichero3 lf = new LecturaFichero3();
        int [] a;
        int numLinea = 1;
        boolean exito=false, fin = false;
        FileReader fr = new FileReader(fichero);
        BufferedReader br=new BufferedReader(fr);
        while ((!exito) || (!fin)){
            try{
                exito = false;
                a=lf.leerVector2(br);
                System.out.println("Leída línea nº: "+numLinea);
                numLinea++;
                for (int k=0;k<a.length;k++){
                    System.out.print("a["+k+"]="+a[k]+" ");
                }
                System.out.println();
                exito=true;
            }catch (FileNotFoundException e){
                System.out.println("fichero "+args[0]+" no existe");
                System.out.println("pulsar retorno para continuar");
                System.in.read();
            }catch (DatosFicheroIncorrectos e){
                if (e.vacio()){
                    System.out.println("fichero"+args[0]+" esta vacio");
                    System.out.println("pulsar retorno para continuar");
                    System.in.read();
                }else{
                    System.out.println("error en posicion "+e.posicionError());
                }
            }
        }
    }
}

```

```

        System.out.println("pulsar retorno para continuar");
        System.in.read();
    }
} catch (EOFException e){
    System.out.println(" fin de fichero.....");
    exito = true;
    fin = true;
    br.close();
} //catch
} //while
} //fin main
} //fin PruebaFichero

```

Este programa genera un curioso comportamiento: si el fichero de datos está correctamente formateado funciona correctamente. Sin embargo, si se genera el siguiente fichero de datos, donde aparece un error en la tercera línea del fichero de datos:

```

1   231   24   422
1 231 24 422
42 error 1243 321
431 24

```

La ejecución del programa anterior generaría la siguiente salida:

```

Leída línea n.º: 1
a[0]=1 a[1]=231 a[2]=24 a[3]=422
Leída línea n.º: 2
a[0]=1 a[1]=231 a[2]=24 a[3]=422
error en posicion 1
pulsar retorno para continuar

```

Una vez modificado el error correspondiente (se le ha asignado un valor de 0), el programa terminaría generando la siguiente salida:

```

Leída línea n.º: 4
a[0]=431 a[1]=24
fin del fichero.....

```

donde como puede verse, se ignora la línea errónea y se continúa. Se propone al lector, la modificación de ambas clases para lograr que la línea (una vez modificada) no sea ignorada.

Ejercicio 5.11. Cálculo de estadísticas desde ficheros de datos (propuesta de práctica)

Enunciado Se pide implementar un programa en Java que lea datos almacenados en un fichero de texto que contenga las calificaciones obtenidas por un alumno en diferentes materias ("alumno.cal"). Las calificaciones aparecen en cinco filas, una fila por curso, con tantas columnas como materias (el número de columnas de cada fila es variable, y no es conocido de antemano), con valores numéricos entre 0 y 10. El programa deberá crear un nuevo fichero de salida para el alumno ("alumno.est"), donde aparezca la siguiente información:

- Estadísticas por cursos. Para cada curso, indicar el número de calificaciones, valores mínimo, máximo, media, desviación estándar y moda de cada una de las filas¹. Los valores se representarán con dos decimales.
- Estadística total del alumno. En este caso aparecerán el total de calificaciones y los valores de mínimo, máximo, media, desviación estándar y moda de todas las materias cursadas.

Para un conjunto de N valores, $a_i, i = 1 \dots N$, se tiene:

$$\text{media} = \frac{\sum_{i=1}^N a_i}{N} \quad \text{desviación estándar} = \left(\frac{\sum_{i=1}^N a_i^2}{N} - \text{media}^2 \right)^{1/2}$$

moda: valor repetido un mayor número de veces. Para obtener la moda de un conjunto de calificaciones, se determinará qué intervalo de los siguientes $\{[0,1), [1,2), [2,3), [3,4), [4,5), [5,6), [6,7), [7,8), [8,9), [9,10)\}$ es el más frecuente (se recomienda usar un array con 10 posiciones para contar frecuencias de cada intervalo).

Ejemplo Dado el siguiente fichero (NombreApellido.cal):

```
5.5 5.0 7.5 6.0 6.0 5.5
7.0 5.5 6.0 6.0 8.0
5.0 5.0 8.5 5.0 7.0 6.5 8.0
6.0 5.0 8.5 5.0 9.0 5.5 7.0 6.0
8.5 7.5 5.5 5.5 9.0 5.0 7.0
```

La salida generada en el fichero NombreApellido.est sería:

```
Curso: 1
numero: 6
minimo: 5.0 maximo: 7.5 media: 5.91 std: 0.78 moda: 5.0
```

¹ Se debe implementar un método genérico que devuelva los valores estadísticos a partir de un array de números de tipo `double`. Cada apartado se resolverá enviando el array de datos correspondiente (un array para cada curso, y un array global con todos las calificaciones de alumno) a dicho método.


```
Curso: 2
  numero: 5
  minimo: 5.5 maximo: 8.0 media: 6.5 std: 0.89 moda: 6.0
Curso: 3
  numero: 7
  minimo: 5.0 maximo: 8.5 media: 6.42 std: 1.37 moda: 5.0
Curso: 4
  numero: 8
  minimo: 5.0 maximo: 9.0 media: 6.5 std: 1.43 moda: 5.0
Curso: 5
  numero: 9
  minimo: 5.0 maximo: 9.0 media: 6.88 std: 1.28 moda: 7.0
TOTAL:
  numero: 35
  minimo: 5.0 maximo: 9.0 media: 6.48 std: 1.26 moda: 5.0
```

Solución

A continuación se muestra la solución al problema, en este caso sólo se han implementado dos métodos. El método `main()`, donde se lee como parámetro de entrada el fichero que contiene las calificaciones del alumno en las diferentes materias ("alumno.cal"), y el método **calcularEstadisticos**(double [] valores, double[] res), que realizará los correspondientes cálculos.

Como puede verse en el método `main()` es necesario crear un array de 5 filas con un número indeterminado de columnas (`new double[5][ ]`), dado que *a priori* no se conoce el número de asignaturas que el alumno puede haber cursado para cada uno de los cinco años. Posteriormente, y una vez creado el fichero, se utilizará la clase `StringTokenizer` (`java.util`) para identificar por cada fila el número de columnas (es decir, de asignaturas) que tiene, y construir el array bidimensional (no homogéneo). Una vez leídas todas las notas del alumnos se llama al método `calcularEstadisticos()` encargado de realizar los cálculos solicitados.

```
import java.util.StringTokenizer;
import java.io.*;

public class FicheroCalificaciones {
    public static void main (String args[]) {
        //para leer de fichero
        String nombreEntrada=args[0];
        String ruta = new String("c:\\java\\ej5\\cap5\\ficheros\\");
        String fichEntrada = ruta+nombreEntrada;
        String nombreSalida;
        int i, c, k, columnas, total=0;
        double notasAlumnoCursos[][]=new double[5][ ];
        double notasAlumnoTodo[];
```

```

try{
    FileReader fr = new FileReader(fichEntrada);
    BufferedReader br=new BufferedReader(fr);
    String cadaLinea, nota;
    fr=new FileReader(fichEntrada);
    br=new BufferedReader(fr);
    //se construye un array bidimensional no uniforme
    for (i=0;i<5;i++ ){
        cadaLinea=br.readLine();
        StringTokenizer valores=new StringTokenizer(cadaLinea," ");
        notasAlumnoCursos[i]=new double [valores.countTokens()];
        total = total + valores.countTokens();
        columnas=valores.countTokens();
        for(c=0;c<columnas;c++){
            nota=valores.nextToken();
            notasAlumnoCursos[i][c]=Double.parseDouble(nota);
        }
    }
    // se construye un array con todas las notas del alumno
    notasAlumnoTodo=new double[total];
    int cuenta=0;
    for (i=0;i<5;i++ ){
        for(c=0;c<notasAlumnoCursos[i].length;c++){
            notasAlumnoTodo[cuenta]=notasAlumnoCursos[i][c];
            cuenta=cuenta+1;
        }
    }
    // Fichero de salida:
    nombreSalida=fichEntrada.substring(0,fichEntrada.indexOf('.')+1)
        + ".est";
    FileWriter fw =new FileWriter(nombreSalida);
    PrintWriter pw=new PrintWriter(fw);

    // calcular: minimo, maximo, medio, std y moda, para cada fila:
    double [] estadisticos=new double[5];
    for (i=0;i<5;i++ ){
        calcularEstadisticos(notasAlumnoCursos[i], estadisticos);
        pw.println("Curso: "
            +(i+1)+"\n\t numero:"+notasAlumnoCursos[i].length
            +"\n\t minimo: "+ estadisticos[0]
            +"\t maximo: "+estadisticos[1]
            +"\t media: "+estadisticos[2]
            +"\t std: "+estadisticos[3]
            +"\t moda: "+estadisticos[4]);
    }
}

```

```

calcularEstadisticos(notasAlumnoTodo, estadisticos);
pw.println("TOTAL:"+"\\n\\t numero: "+notasAlumnoTodo.length
+"\\n\\t minimo: "+estadisticos[0]
+"\\t maximo: "+estadisticos[1]
+"\\t media: "+estadisticos[2]
+ "\\t std:"+estadisticos[3]
+"\\t moda: "+estadisticos[4]);
br.close();
pw.close();
}catch (FileNotFoundException fe){
    System.out.println("Error: no encuentra fichero "+fichEntrada);
}catch (IOException e){
    System.out.println("Error de entrada.: "+e);
}
}
}

//fin main

static void calcularEstadisticos(double [] valores, double[] res){
    int i, indice;
    double maximo=0, minimo=0, moda=0, maxModa=0;
    double media=0, medCuad=0;
    int repet[]=new int[10];

    for (i=0;i<10;i++ ) //contadores de frecuencia de cada intervalo
        repet[i]=0;

    for (i=0;i<valores.length;i++){
        if (i==0)
            maximo=minimo=valores[i];
        if (valores[i]<minimo)
            minimo=valores[i];
        if (valores[i]>maximo)
            maximo=valores[i];
        media=media+valores[i];
        medCuad=medCuad+valores[i]*valores[i];
        indice=(int)valores[i];
        if (indice>9)
            indice=9;
        repet[indice]++;
    }
    moda=0;
    maxModa=0;
    for (i=0;i<10 ;i++ ){
        if (i==0){ //inicialización
            moda=repet[i];
            maxModa=i;

```

```

        }else if (maxModa<repet[i]){
            maxModa=repet[i];
            moda=i;
        }
    }
    media=media/valores.length;
    res[0]=minimo;
    res[1]=maximo;
    res[2]=(int) (100*media)/100.0;
    res[3]=(int) (100*Math.sqrt((medCua-
                                media*media)/valores.length
                                -media*media))/100.0;

    res[4]=moda;
} //fin calcularEstadisticos
} //fin FicheroCalificaciones

```

El método **calcularEstadisticos** (`double [] valores, double[] res`), básicamente, consiste en recorrer el array de valores que se ha pasado como parámetro al método e implementar las diferentes fórmulas para la media, moda y desviación estándar que se pedía en el enunciado del problema. Como puede verse en este método, los sumatorios se transforman en bucles `for` y se utilizan los métodos de la clase `Math` (`java.lang`). Una vez calculados los diferentes valores estadísticos, éstos son devueltos en el parámetro **double[] res** que finalmente son almacenados en el fichero de salida con el formato que se solicitó (únicamente es necesario utilizar el método `println()` de la clase `PrintWriter`).

Ejercicio 5.12. Cálculo de expedientes desde ficheros de notas (propuesta de práctica)

Enunciado A partir del problema anterior se pide implementar un programa que lea desde un directorio “**expedientes**”, un conjunto de ficheros de calificaciones correspondientes a una serie de alumnos. Se pide realizar el cálculo de los valores estadísticos para cada un alumno (desarrollado en el problema anterior), y calcular las estadísticas de todos los alumnos contenidos en el directorio (se recomienda utilizar los métodos de la `File`). Finalmente, se deberá generar un fichero resumen (“**totales**”) en ese mismo directorio, con la calificación media de cada alumno, y los valores estadísticos globales de todos los alumnos, así como el nombre del alumno que tenga el mayor valor medio.

Solución En este caso se ha implementado un método `tratarFichero(String nombre)`, que permite extraer los datos para cada fichero de alumno, si y sólo si, la extensión del fichero termina en “.cal”. En caso de hallarse otros ficheros con diferentes extensiones no serían tenidos en cuenta. Este método, devolverá un array con toda la información que se ha encontrado para los ficheros de los alumnos almacenados en el directorio. Inicialmente, y una vez accedido al directorio en cuestión, se construirá un array (`notasAlumnos`) con tantas filas como ficheros contenga el directorio de expedientes. Posteriormente, y para cada fichero, los datos de cada alumno serán leídos y almacenados en el mismo.

```

import java.util.StringTokenizer;
import java.io.*;

```



```

public class DirectorioCalificaciones {
    public static void main (String args[]){

String ruta = new String("c:\\\\java\\ej\\cap5\\ficheros\\");
String ext, nombreDir=ruta+"expedientes";
    File dir=new File(nombreDir);
    String ficheros[] = dir.list(); // ficheros del directorio
    String nombre=null, alumno;
int i,c,cuenta;
    double mediaMax=0;
    int alumMax=0;
    double[][] notasAlumnos= new double [ficheros.length][];
    int totalDatos=0;
        for (i=0;i<ficheros.length;i++){
            ext=ficheros[i].substring(ficheros[i].indexOf('.'));
            if (ext.equals("cal")){
                nombre=nombreDir+"\\ "+ficheros[i];
                notasAlumnos[i]=tratarFichero(nombre);
                totalDatos+=notasAlumnos[i].length;
            }
        }
        try{
            nombre=nombreDir+"\\totales.est";
            FileWriter fw =new FileWriter(nombre);
            PrintWriter pw=new PrintWriter(fw);

            // se pasa las notas de cada alumno a un array global
            // se calcula el valor medio máximo
            double notasTotalAlumnos[]=new double [totalDatos];
            cuenta=0;

            double [] estadisticos=new double[5];
            for (i=0;i<ficheros.length;i++ ){
                ext=ficheros[i].substring(ficheros[i].indexOf('.'));
                if (ext.equals("cal")){
                    for(c=0;c<notasAlumnos[i].length;c++){
                        notasTotalAlumnos[cuenta]=notasAlumnos[i][c];
                        cuenta=cuenta+1;
                    }
                    calcularEstadisticos(notasAlumnos[i],estadisticos);
                    alumno=ficheros[i].substring(0,ficheros[i].indexOf('.'));
                    pw.println("alumno "+alumno+": "+estadisticos[2]);
                    if (i==0){
                        mediaMax=estadisticos[2];
                        alumMax=i;
                    }
                }
            }
        }
    }
}

```

```

        }
        if (mediaMax<estadisticos[2]){
            mediaMax=estadisticos[2];
            alumMax=i;
        }
    }
}

//fin for; para los alumnos almacenados en el directorio

//se almacenan los datos estadísticos para los alumnos
calcularEstadisticos(notasTotalAlumnos, estadisticos);
pw.println("TOTAL:\n\t numero: "
+notasTotalAlumnos.length
+"\n\t minimo: "
+estadisticos[0]
+"\t maximo: "
+estadisticos[1]
+"\t media: "
+estadisticos[2]
+"\t std: "
+estadisticos[3]
+"\t moda: "
+estadisticos[4]);
alumno=ficheros[alumMax].substring(0,ficheros[alumMax]
                                                                    .indexOf('.'));

//mejor alumno:
pw.println("El alumno con maximo valor medio es: "
+alumno
+" y valor : "
+mediaMax);

pw.close(); //se cierra el fichero
}catch (FileNotFoundException fe){
    System.out.println("Error: no encuentra fichero '"+nombre+"'");
}catch (IOException e){
    System.out.println("Error de entrada.: "+e);
}
}

//fin main

static double[] tratarFichero (String nombreEntrada){
    //para leer de fichero
    String nombreSalida;
    int i, c, k, columnas, total=0;
    double notasAlumnoCursos[][]=new double[5][];
    double notasAlumnoTodo[]=null;
    try{

```

```
FileReader fr=new FileReader(nombreEntrada);
BufferedReader br=new BufferedReader(fr);
String cadaLinea, nota;

fr=new FileReader(nombreEntrada);
br=new BufferedReader(fr);

for (i=0;i<5;i++ ){
    cadaLinea=br.readLine();
    StringTokenizer valores= new StringTokenizer(cadaLinea," ");
    notasAlumnoCursos[i]=new double [valores.countTokens()];
    total = total + valores.countTokens();
    columnas=valores.countTokens();
    for(c=0;c<columnas;c++){
        nota=valores.nextToken();
        notasAlumnoCursos[i][c]=Double.parseDouble(nota);
    }
}
// construye array con todas las notas del alumno
notasAlumnoTodo=new double[total];
int cuenta=0;

for (i=0;i<5;i++ ){
    for(c=0;c<notasAlumnoCursos[i].length;c++){
        notasAlumnoTodo[cuenta]=notasAlumnoCursos[i][c];
        cuenta=cuenta+1;
    }
}

// Fichero de salida:
nombreSalida=nombreEntrada.substring(0,nombreEntrada.indexOf('.'))
                                     +".est";

FileWriter fw =new FileWriter(nombreSalida);
PrintWriter pw=new PrintWriter(fw);
// ahora se calcula el minimo, maximo, medio, std y moda de cada
// fila:
double [] estadisticos=new double[5];
for (i=0;i<5;i++ ){
    calcularEstadisticos(notasAlumnoCursos[i], estadisticos);
    pw.println("Curso: "+(i+1)
        +"\n\t numero: "
        +notasAlumnoCursos[i].length
        +"\n\t minimo: "
        +estadisticos[0]
        +"\n\t maximo: "
```

```

        +estadisticos[1]
        +"\t media: "
        +estadisticos[2]
        +"\t std: "
        +estadisticos[3]
        +"\t moda: "+estadisticos[4]);
    }
    calcularEstadisticos(notasAlumnoTodo, estadisticos);
    pw.println("TOTAL:\n\t numero: "
        +notasAlumnoTodo.length
        +"\n\t minimo: "
        +estadisticos[0]
        +"\t maximo: "
        +estadisticos[1]
        +"\t media: "
        +estadisticos[2]
        +"\tstd:"
        +estadisticos[3]
        +"\t moda: "+estadisticos[4]);

    br.close();
    pw.close();
} catch (FileNotFoundException fe){
    System.out.println("Error: no encuentra fichero '"
        +nombreEntrada+"'");
} catch (IOException e){
    System.out.println("Error de entrada.: "+e);
}
return notasAlumnoTodo;
} //tratarFichero

static void calcularEstadisticos(double [] valores, double[] res){

    //.....ver problema anterior .....//
}

} //fin DirectorioCalificaciones

```

Una vez ejecutado el programa anterior, se obtendrán un conjunto de ficheros que corresponden a las diferentes estadísticas de cada alumno, además del fichero que contiene las estadísticas de todos los alumnos almacenados, el formato de este fichero (para cuatro ficheros de alumnos generados aleatoriamente) sería:

```

alumno alumno: 6.04
alumno alumno2: 5.94

```



```

alumno alumno3: 5.96
alumno alumnoMAX: 7.57
TOTAL:
    numero: 128
    minimo: 2.5    maximo: 9.0    media: 6.38    std: 1.68    moda: 5.0
El alumno con maximo valor medio es: alumnoMAX y valor : 7.57

```

Ejercicio 5.13. Ordenación por mezcla natural de un fichero

Enunciado Implementar la clase `OrdenarExterno.java` que acepte los siguientes parámetros:

- **args[0]**: valor entero entre 1 y 100. El valor de este parámetro se utilizará para generar un fichero que contenga tanto números de tipo **double** como el valor del parámetro indique.
- **args[1]**: valor entero entre 1 y 1000. El valor de este parámetro se utilizará para generar los números aleatorios que serán almacenados en el fichero. Es decir, si se introduce un valor de 900, se generarán números aleatorios comprendidos entre 0 y 900 que serán posteriormente almacenados.

Una vez generado aleatoriamente el fichero (`fichNumerosAleatorios.dat`), se pide ordenar su contenido utilizando el algoritmo de **mezcla natural** (los valores almacenados pueden estar repetidos). Se deberán implementar dos métodos que realicen los procesos de mezcla y distribución de los datos almacenados en el fichero origen:

```
public static void distribucion(File fichC, File fichA, File fichB) throws
IOException
```

```
public static void mezcla(File fichA, File fichB, File fichC) throws IOException
```

Nota: se recomienda consultar la clase `java.lang.Math`, y en particular su método `random()`; que genera números aleatorios comprendidos entre: **[0, 1)**.

Solución Una vez estudiado cuidadosamente el algoritmo descrito en el Apartado 5.4.2., y utilizando las clases `DataInputStream` y `DataOutputStream`, se creará aleatoriamente un fichero que almacene los datos a ordenar. En primer lugar, se leerá por teclado el valor de los diferentes parámetros, que serán utilizados para construir un fichero de números aleatorios de tipo `double`. Esta operación es realizada por el método `generaFicheroNumeros()`. Se ha implementado el método `listaFicheroNumeros(File fich)`, que permite visualizar por pantalla el contenido del fichero, a continuación se muestran ambos métodos:

```

//métodos generales
public static void generaFicheroNumeros(File fich,
                                         int numElement, double rangoVal){
    //se genera aleatoriamente el fichero de datos a ordenar
    try{
        DataOutputStream fichCOut = new DataOutputStream(
            new FileOutputStream(fich));

```

```

        for(int i=0; i<numElement;i++)
            fichCOut.writeDouble(Math.random()*rangoVal);
        fichCOut.close();
    }catch (IOException e){
        System.out.println("No se pudieron generar los datos del fichero");
    }
}
}

public static void listaFicheroNumeros(File fich){
    //se lista el contenido del fichero hasta EOF
    double dat = 0.0;
    int i = 0;
    try{
        DataInputStream fichIn = new DataInputStream(new FileInputStream(fich));
        while (!eof(fichIn)){
            dat = fichIn.readDouble();
            System.out.println("dato "+i+": "+dat);
            i++;
        }
        fichIn.close();
    }catch(IOException e){
        System.out.println("No se pudieron leer los datos del fichero... "+e);
    }
}
}

```

Una vez creado el fichero (`fichNumerosAleatorios.dat`), se utilizará como fichero de datos a ordenar. Se utilizarán otros dos ficheros auxiliares (`auxiliar1.dat` y `auxiliar2.dat`) para realizar el proceso de mezcla.

El algoritmo va a utilizar algunos métodos auxiliares muy importantes. En particular, el método `eof(File fich)`, que devuelve un valor booleano cuando se ha llegado al final del fichero (para ello se utiliza el método `available()` de la clase `DataInputStream`), y el método `leoDatoFich()` que lee el primer elemento de un fichero sin mover el puntero de lectura escritura.

El algoritmo de mezcla natural necesita poder comparar el elemento que acaba de leer (de cualquiera de los ficheros) con el siguiente que quedaría almacenado, para comprobar si se ha llegado al final, o no, de un tramo. Para ello es necesario acceder al primer elemento de un fichero sin mover el puntero de lectura/escritura (dado que si el puntero se mueve el dato se perdería). En este caso (para la clase considerada), se ha utilizado el método `skip(long desplazamiento)` que permite mover el puntero de lectura/escritura los bytes deseados.

En el caso del ejercicio planteado, y dado que se leen valores de tipo `double` (8 bytes), será necesario desplazar hacia atrás esos 8 bytes una vez leído el dato, de esa forma se consigue leer en una variable auxiliar el siguiente dato almacenado en el fichero sin que el puntero resulte desplazado.

```
//Método para detectar el final de un fichero
public static boolean eof(DataInputStream dis) throws IOException{
    if (dis.available() == 0)
        return true;
    else
        return false;
}

//Método para acceder al primer elemento de un fichero,
//sin mover el puntero de lectura escritura
public static double leoDatoFich(DataInputStream dis) throws IOException{
    double devolver;
    devolver=dis.readDouble(); //se lee el dato...
    dis.skip(-8); //se coloca el puntero nuevamente
    return devolver;
}
```

Una vez definidos los métodos básicos, los siguientes métodos permiten realizar el proceso de distribución de los tramos del fichero a ordenar. En primer lugar, se debe implementar el método que permite distribuir el fichero con los datos iniciales en los dos ficheros auxiliares (distribuyéndolos en sus respectivos tramos):

```
// ***** DISTRIBUCION *****//
public static boolean copiar(DataInputStream dis, DataOutputStream dos)
                                                                    throws IOException{

    //Copiar elementos de un fichero a otro
    double buf, aux;
    buf = dis.readDouble();
    dos.writeDouble(buf);
    if (eof(dis))
        return true;
    else{
        aux = leoDatoFich(dis);
        return (buf>aux);
    }
}

public static void copiTramo(DataInputStream dis, DataOutputStream dos)
                                                                    throws IOException{

    //se copia un tramo de dis a dos
    do{
        fdt = copiar(dis,dos);
    }while(!fdt); //se copia hasta fin de tramo
}
```

```

public static void distribucion(File fichC, File fichA, File fichB)
                                                    throws IOException{

    //se distribuye del fichero C a A y B
    //se establecen los flujos...
    DataInputStream _CIn = new DataInputStream(new FileInputStream(fichC));
    DataOutputStream _AOut = new DataOutputStream(new FileOutputStream(fichA));
    DataOutputStream _BOut = new DataOutputStream(new FileOutputStream(fichB));
    do{
        copiTramo(_CIn, _AOut);
        if (!eof(_CIn))
            copiTramo(_CIn, _BOut);
    }while(!eof(_CIn)); //se copia hasta terminar con el fichero C
    _CIn.close();
    _AOut.close();
    _BOut.close();
} //fin distribución

```

El método `distribucion()` acepta como parámetros los tres ficheros a manipular, establece tres flujos (uno de entrada para leer los datos y dos de salida para copiar los diferentes tramos), y copia los tramos almacenados en el fichero de datos hasta terminar.

El proceso de copia de los tramos lo realiza el método `copiTramo()`, que simplemente ejecutará al método `copiar()` mientras queden tramos en el fichero origen (sea cual sea). Como puede verse, el método de copia extrae uno a uno los diferentes elementos del fichero y lo compara con el siguiente elemento que queda almacenado para determinar si es el final o no de un tramo. Cuando se llega al final del fichero, o cuando el elemento que acaba de leerse es mayor que el siguiente elemento del fichero, se ha llegado al final de un tramo.

Una vez se ha logrado distribuir el fichero original en diversos tramos ordenados y almacenarlos en los ficheros auxiliares, se debe proceder a la mezcla ordenada de los tramos sobre el fichero original. Para ello, se han implementado los métodos `mezcla()`, y `mezTramo()` que permiten realizar el anterior proceso.

```

//***** MEZCLA *****/
public static void mezTramo(DataInputStream _A, DataInputStream _B,
                            DataOutputStream _C) throws IOException{
    //se mezclan los tramos de los ficheros A y B en C
    double datoA, datoB;
    do{
        datoA = leoDatoFich(_A);
        datoB = leoDatoFich(_B);
        if(datoA <= datoB){
            fdt = copiar(_A, _C);
            if (fdt)
                copiTramo(_B, _C);
        }else{

```



```

        fdt = copiar(_B,_C);
        if (fdt)
            copiTramo(_A,_C);
    }
}while(!fdt);
}

public static void mezcla(File fichA, File fichB, File fichC)
                                                                    throws IOException{
    //se mezclan los ficheros A y B en C
    DataInputStream _AIn = new DataInputStream(new FileInputStream(fichA));
    DataInputStream _BIn = new DataInputStream(new FileInputStream(fichB));
    DataOutputStream _COut = new DataOutputStream(new FileOutputStream(fichC));

    while ((!eof(_AIn)) && (!eof(_BIn))){
        mezTramo(_AIn,_BIn,_COut);
        numTramos++;
    }
    while(!eof(_AIn)){
        copiTramo(_AIn,_COut);
        numTramos++;
    }
    while(!eof(_BIn)){
        copiTramo(_BIn,_COut);
        numTramos++;
    }
    _AIn.close();
    _BIn.close();
    _COut.close();
} //fin mezcla

```

Como puede verse en los anteriores métodos, el método de mezcla está formado por tres bucles `while` que garantizan que ambos ficheros auxiliares (ficheros **a** y **b** del algoritmo) se recorran completamente hasta que no queden elementos en ninguno de ellos. Cuando en ambos ficheros quedan tramos sin mezclar (primer bucle), se ejecuta el método de mezcla de tramos que inspecciona los dos primeros elementos de cada fichero antes de decidir cuál de ellos debe ser copiado en primer lugar sobre el fichero destino. Este proceso se repite hasta terminar con el tramo de cada fichero considerado. Finalmente, si alguno de los dos ficheros contiene más tramos que el otro (es decir, se ha acabado la mezcla de tramos pero alguno de los dos ficheros todavía contiene elementos), simplemente se deben copiar los elementos restantes sobre el fichero **c**.

Finalmente, y una vez se ha comprobado el correcto funcionamiento de los anteriores métodos, el método `main()` acepta por teclado los diferentes parámetros descritos en el enunciado, genera un fichero de datos aleatorios a ordenar y realiza el proceso de ordenación de los datos.

```

import java.io.*;
public class OrdenarExterno{

//atributos
private static String ruta = new String("c:\\java\\ej\\cap5\\ficheros\\");
private static String fAux1 = "auxiliar1.dat";
private static String fAux2 = "auxiliar2.dat";
private static String ficheroOrdenar = ruta+"fichNumerosAleatorios.dat";
private static int numTramos = 0;
private static boolean fdt; //Fin De un Tramo

//métodos
public static void generaFicheroNumeros(File fich, int numElement,
double rangoVal){

.....
public static void listaFicheroNumeros(File fich){
.....
public static boolean eof(DataInputStream dis) throws IOException{
.....
public static boolean copiar(DataInputStream dis, DataOutputStream dos)
throws IOException{
.....
public static void copiTramo(DataInputStream dis, DataOutputStream dos)
throws IOException{
.....
public static void distribucion(File fichC, File fichA, File fichB)
throws IOException{
.....
public static void mezTramo(DataInputStream _A, DataInputStream _B,
DataOutputStream _C) throws IOException{
.....
public static void mezcla(File fichA, File fichB, File fichC)
throws IOException{
.....
//METODO MAIN
public static void main(String[] args){
try{
//lectura de los parámetros
int numeroElementos = Integer.parseInt(args[0]);
double rangoValores = Integer.parseInt(args[1]);
//ficheros de datos
File fichC = new File(ficheroOrdenar);
File fichA = new File(ruta+fAux1);
File fichB = new File(ruta+fAux2);

```

```
//se genera el fichero de datos a ordenar
System.out.println("Se genera el fichero de datos a ordenar...");
generaFicheroNumeros(fichC,numeroElementos,rangoValores);
listaFicheroNumeros(fichC);

//MECLA EXTERNA
do{
    distribucion(fichC,fichA,fichB);
    numTramos = 0;
    mezcla(fichA,fichB,fichC);
}while (!(numTramos == 1));

//Se lista el fichero ordenado
System.out.println("Fichero ordenado:");
listaFicheroNumeros(fichC);
}catch(Exception e){
    System.out.println("Excepcion....."+e.getMessage());
    e.printStackTrace();
}
}
} //fin main
} //fin OrdenarExterno
```

Como puede verse, en el método `main()` se ha implementado directamente el método de mezcla natural. Para implementar el algoritmo se realiza un conjunto de sucesivas distribuciones y mezclas hasta conseguir que el número de tramos almacenados en el fichero **c** sea igual a 1 (indica que todos los datos están ordenados y que, por tanto, el proceso de ordenación ha concluido). Debe observarse que después del proceso de distribución el número de tramos se pone a cero, esto es debido a que es el proceso de mezcla el que determina el número de tramos útiles que están almacenados en el fichero **c**.

Objetos y clases en Java

En los capítulos anteriores se ha trabajado con tipos de datos definidos en el lenguaje, como enteros, reales, caracteres, etc., desarrollándose diversos algoritmos que trabajan con esos tipos de datos. En la mayoría de las aplicaciones reales, además de la definición de algoritmos y su posterior programación, es necesaria la creación de nuevos tipos de datos que permitan agrupar información de distintos tipos básicos. La combinación de diversas informaciones (atributos) en una única variable, unificando el uso de todas ellas, se denomina *objeto*. Un ejemplo sencillo de esta necesidad es el concepto de fecha que agrupa en una única variable al menos tres atributos definidos mediante números enteros: uno que define el día, otro el mes y otro el año. En realidad, el objeto es algo más complejo que una mera agrupación de atributos, incluyendo un conjunto de funciones (métodos) que permiten utilizar las variables que lo componen. Un conjunto de objetos que comparten las mismas características (atributos y métodos) definirán lo que se denomina una *clase* que permitirá definir de manera abstracta cómo son los objetos de esa clase. En este capítulo se profundizará en estas ideas, e intentaremos responder a preguntas cómo: ¿qué es un objeto, qué es encapsular y cuál es su utilidad? ¿Qué es una clase, cómo se define, cómo se utiliza, cómo se crean objetos de la misma?

6.1. Objetos como paradigmas de encapsulación

Podemos definir un objeto como un conjunto complejo de datos y de funciones que pueden acceder a esos datos, dotado de una determinada estructura y que forma parte de una organización donde se relaciona con otros objetos.

La forma de indicar a un objeto que realice determinadas tareas es enviándole un mensaje, el cual le dice lo que tiene que hacer. El objeto receptor responde al mensaje de la siguiente manera:

1. Selecciona la operación que le indica el nombre del mensaje.
2. Ejecuta la operación seleccionada.
3. Devuelve el control al objeto que hizo la llamada.

Esta forma de trabajar pone de manifiesto las diferencias que hay entre los objetos y las variables tradicionales. Una variable se limita a esperar, de forma pasiva, a que se aplique sobre ella una función. Sin embargo, los objetos son activos ya que cada objeto tiene definidas un conjunto de operaciones que se pueden aplicar sobre las informaciones (datos) que contiene.

En teoría, no es posible acceder directamente a la información contenida en un objeto, a diferencia de lo que ocurre con las variables normales. De esta manera una función que se aplique a un objeto no puede modificar los valores del mismo. Esto es así porque los datos de un objeto son propiedad privada suya y la única forma de acceder a ellos es solicitar que lo haga el propio objeto, a través de sus propias funciones, que se denominan métodos.

Esta forma de integrar los datos y las funciones dentro de un objeto libera al programador que usa ese objeto de la responsabilidad de programar cada una de las funciones. De esta manera el programador que desarrolla un objeto y sus funciones se preocupa de decidir qué funciones son necesarias y de cómo se implementan, mientras que el programador que hace uso de ese objeto se preocupa únicamente de conocer las funcionalidades del objeto sin preocuparse de cómo se hace un determinado servicio.

Supongamos que deseamos desarrollar una aplicación gráfica para la cuál sería necesario definir distintas figuras geométricas bidimensionales: círculos, rombos, trapecios, cuadrados, etc. El programador de la aplicación gráfica podría recurrir a otro programador que desarrollara objetos para cada uno de los tipos de figuras. De esta manera el programador final hará uso de los objetos sin saber cómo están realmente implementados.

En definitiva, los objetos son agrupaciones de datos y operaciones, que se activan mediante llamadas a sus métodos. Esa agrupación da lugar al concepto de **encapsulamiento**. Desde el punto de vista de la orientación a objetos, la **encapsulación** consiste en definir una "cápsula" a modo de barrera conceptual que separa un conjunto de valores y operaciones, que poseen un substrato conceptual idéntico del resto del sistema. Es decir, los datos y programas que forman el objeto aparecen en el interior de esa cápsula.

La abstracción de un objeto debe preceder a las decisiones sobre su implementación y una vez que dicha implementación se ha realizado debe tratarse como un secreto y esconderse del resto del programa, es decir, se debe encapsular.

Teóricamente ninguna parte de un sistema complejo debería depender de la implementación, es decir, de los detalles internos de otra parte de ese mismo sistema. Se puede decir que la abstracción ayuda al hombre a pensar sobre lo que está haciendo y la encapsulación permite que los cambios que se realicen en el sistema no afecten a la estabilidad del mismo.

Abstracción y encapsulación son, por tanto, términos complementarios: la abstracción se centra en el exterior de los objetos y la encapsulación previene a los demás objetos de las variaciones en la implementación de los objetos. Así, cada objeto tiene dos partes bien diferenciadas, la interfaz y la implementación:

- La **interfaz** es pública, conocida por los demás objetos del sistema, y es el resultado de aplicar la abstracción al objeto.
- La **implementación** es el producto de aplicar la encapsulación al objeto en cuestión. En esta parte se explicitan y codifican los mecanismos (métodos) necesarios para responder a la interfaz especificada.

El hecho de que cada objeto sea una caja negra que contiene todos los datos y programas ligados a él, facilita enormemente que cualquier objeto determinado se pueda transportar con todo lo que contiene a otro punto del sistema, o incluso a otro sistema diferente. Si el objeto se ha construido bien, sus métodos seguirán funcionando en el nuevo entorno, como si no hubiera sido transplantado. Esta cualidad permite que la programación orientada a objetos favorezca la reutilización del código.

La encapsulación, en principio, establece una barrera conceptual, pero no fija si dicha barrera es transparente o no. La transparencia de la encapsulación sólo indica si los datos y operaciones del objeto encapsulado permanecen visibles o accesibles a los demás objetos del sistema. Aquí subyace el concepto de la **ocultación de la información**. Este concepto permite que cada objeto definido en nuestro sistema pueda ocultar su implementación y hacer pública su interfaz de comunicación.

La ocultación de la información no quiere decir que sea imposible conocer un objeto y su contenido, lo cual no tiene ninguna utilidad. Lo que indica es que las peticiones de información a un objeto deben cursarse por los cauces adecuados, es decir, a través de los métodos predefinidos de ese objeto. La respuesta de esos métodos será la información solicitada al objeto.

Los objetos, a su vez, pueden formar parte de otros objetos de manera que pueden definirse objetos que a su vez agrupen otros objetos. Por ejemplo, si se define un objeto fecha que encapsula tres números enteros para definir el día, el mes y el año, y un conjunto de métodos para acceder a esos valores, una vez definido el objeto fecha éste puede formar parte de otro objeto más complejo. Así, si se abstrae el concepto, persona uno de cuyos atributos es la fecha de nacimiento, puede utilizarse el objeto fecha y no es necesario definir tres atributos enteros dentro de persona. Esto hace que abstracciones más complejas hagan uso de abstracciones más simples, y la encapsulación de abstracciones simples permite que el código que se genere sea más compacto y esté libre de errores en los objetos más sencillos, liberando al programador de tener que reprogramar objetos que ya han sido programados anteriormente.

6.1.1. Redefinición de los objetos

Como se ha visto los objetos encapsulan y ocultan un conjunto de valores de manera que la información queda aislada del resto del programa. Debido a estas dos características, la programación con objetos tiene la propiedad de permitir que el código sea revisado o extendido sin afectar a la integridad del sistema.

Otra propiedad fundamental de los objetos fundamentada en la encapsulación es la facilidad de la depuración de los programas. Al ser responsabilidad del programador del objeto la implementación de cada uno de los métodos, una vez que éste se ha asegurado que el objeto funciona

correctamente cualquier programador que utilice dicho objeto no deberá preocuparse de si la implementación del mismo es la adecuada o no. Al componerse el programa de objetos aislados que se comunican mediante llamadas a métodos, la depuración del programa final es un proceso sencillo que se basa en la depuración de cada uno de los objetos que componen el programa final.

Evidentemente cada posible definición establece una manera de trabajar con el objeto, en algunos casos una definición permite que el objeto tenga unas propiedades diferentes.

6.1.2. Atributos y métodos

Las variables que almacenan la información de un objeto son los atributos del mismo. Estos atributos pueden ser de los tipos predefinidos en el lenguaje o ser a su vez objetos definidos por el programador. El acceso a estos atributos se realiza siempre a través de los métodos del objeto. De forma más general se suele hablar de mensajes enviados al objeto más que de los métodos del objeto. Esto es así porque el concepto método es más restringido y hace referencia al código que implementa dicha funcionalidad y el concepto mensaje es más amplio e incluye la idea de solicitar algo al objeto. De forma conceptual el acceso a un objeto implica los siguientes conceptos:

Mensaje

Está formado por el nombre de una operación y los argumentos requeridos. Cuando un objeto envía un mensaje a otro objeto, el emisor espera que el receptor realice la operación nombrada en el mensaje y que le devuelva algún tipo de información. Cuando el receptor recibe el mensaje, realiza la operación requerida según se especifica en su implementación privada. Al emisor no le interesa, en absoluto, la forma en que ha efectuado la tarea sino que sólo le interesa que lo que ha pedido esté hecho. Por tanto, se puede definir el comportamiento de un objeto mediante el conjunto de mensajes que ese objeto puede recibir. Habitualmente se suele hablar de dos tipos de mensajes: por un lado, aquellos que se utilizan para obtener información del objeto, denominados "preguntas" (*queries*), sin afectar al "estado" del objeto definido por los valores específicos de todos sus atributos. Por otro lado, aquellos que se utilizan para ejecutar una determinada acción, denominados "comandos" (*commands*), y que habitualmente sí modifican el estado del objeto.

Nombre del mensaje

Está constituido por el nombre de una determinada operación. Los nombres de los mensajes deben ser lo suficientemente descriptivos para que terceros programadores comprendan bien el comportamiento del objeto dado.

Signatura

La signatura está constituida por los tipos de sus parámetros y el tipo de objeto que el método devuelve. La signatura especifica tanto la entrada como la salida de un método.

Es el conjunto de operaciones necesarias para que un objeto realice una tarea específica. Se trata, pues, de un algoritmo ejecutado en respuesta a un mensaje. Por tanto, un método es un algo-

ritmo en el que se detallan, paso a paso, toda una serie de operaciones que se realizarán cuando el objeto reciba un mensaje cuyo nombre sea igual al del método a ejecutar. Como consecuencia del principio de ocultación de la información, un método pertenece a la parte privada de un objeto; nunca será parte de la interfaz pública de dicho objeto. Se puede pensar que el funcionamiento de un método es similar a la llamada y ejecución de funciones en un lenguaje tradicional, donde la llamada a una función sería el mensaje y el salto y ejecución de la función sería el método, sólo que se esconde la implementación de la rutina al resto del programa.

6.2. Clases y objetos en Java

En los apartados anteriores hemos estado hablando de los objetos como si cada uno de ellos fuera totalmente diferente de los demás. Sin embargo, los objetos pueden tener similitudes. Así, en el mundo real se pueden encontrar coches de diferentes marcas y colores, una gran variedad de vehículos, etc. Del mismo modo, una aplicación computacional puede necesitar de distintos ficheros, varias ventanas, cadenas de texto, etc. Algunos de estos objetos tendrán comportamientos diferentes, mientras que otros tendrán un comportamiento parecido.

De esta manera, aparece el concepto de **clase**, el cual sirve para definir a aquel conjunto de objetos que tienen un mismo comportamiento. Una clase es, pues, una especificación genérica para un número arbitrario de objetos similares. Las clases permiten crear una taxonomía de objetos a un nivel abstracto. Permiten describir el comportamiento genérico de un conjunto de objetos y crear objetos que se comporten de determinada manera cuando así lo necesitemos.

Una clase describe un grupo de objetos que contienen informaciones (atributos) similares y un comportamiento (métodos) común. Así, "Coches", "Figuras", "Animales" son en verdad clases de objetos. La mayoría de los objetos de una clase su individualidad deriva de la diferencia del contenido de sus atributos, pero es posible que existan objetos distintos con atributos iguales. Así por ejemplo, es posible definir la clase "Cuadrado", que posee los atributos `pto1`, `pto2` y `color` y que existan dos objetos (dos cuadrados) diferentes que posean los mismos valores.

Cada objeto conoce la clase a la que pertenece y la mayoría de los lenguajes orientados a objetos permiten determinar, en tiempo de ejecución, la clase a la que pertenece un determinado objeto. La clase a la que pertenece cada objeto es, por tanto, una propiedad implícita del objeto en cuestión.

El concepto de clase tiene mucho que ver con el concepto de abstracción. Realmente cuando agrupamos objetos dentro de una clase, en realidad estamos realizando una abstracción. Al abstraer las informaciones comunes de "Libro", "Revista", etc. y las operaciones que se pueden realizar como "Extraer Texto", "Escribir Texto", etc., estamos realizando la abstracción de la clase "Publicación".

Las definiciones comunes (como el nombre de la clase, los nombres de los atributos y los métodos) se almacenan solamente una vez en cada clase, sin importar cuántos objetos de dicha clase estén presentes en el sistema. Los conceptos de objeto y clase son paralelos y casi inseparables, puesto que no podemos hablar de un objeto sin hacerlo también de la clase a la que per-

tenece. Sin embargo, existe una diferencia clara entre objeto y clase. Un objeto es una entidad concreta que existe en un tiempo y en un espacio determinado, mientras que las clases sólo representan una abstracción, son la esencia de los objetos a los que definen.

Para identificar un cuadrado en particular, es decir, para identificar un objeto concreto de la clase "Cuadrado", debemos dirigirnos precisamente a ese cuadrado. Es decir, se utiliza un objeto que por definición existe en el programa y no es el producto de una abstracción. Así pues, un objeto es simplemente una *instancia* de una clase.

Para definir una clase en Java se utiliza la palabra reservada *class* y después el nombre de la clase. Los atributos y los métodos se definen posteriormente:

```
class <nombre de la clase>
{
    <tipo de dato> <nombre de atributo>;
    <tipo de dato> <nombre de atributo>;
    .....
    <tipo de dato> <nombre de atributo>;

    <tipo de dato de retorno> <nombre del método>(parámetros)
    {
        .....
    }
    <tipo de dato de retorno> <nombre del método>(parámetros)
    {
        .....
    }

    .....

    <tipo de dato de retorno> <nombre del método>(parámetros)
    {
        .....
    }

}
```

6.2.1. Instancias de una clase. El operador new

Cuando se tiene en Java la definición de una clase el compilador conoce los atributos de esa clase pero no reserva memoria para ella. Al definir la variable:

```
<nombre de clase> mivariable;
```

se crea una referencia para un objeto de dicha clase, considerando cada uno de los campos. Para declarar el objeto y que se reserve memoria para él es necesario utilizar el operador `new`:

```
mivariable = new <nombre de clase>();
```

El operador `new` es el encargado de crear instancias de una clase, es decir los objetos que tienen las características de la clase. El operador `new` de Java es capaz de reservar la memoria para un objeto de ese tipo sabiendo los atributos que tiene según la definición de la clase. De esta manera reserva un lugar en memoria para el objeto y enlaza cada método con la definición de la clase que se encuentra almacenada en otro lugar de memoria. Aunque en Java se habla de referencias, por no utilizar la palabra puntero que se utiliza en C y C++, realmente el concepto es el mismo. Una referencia es capaz de almacenar una posición de memoria. Al definir una referencia a un objeto de una clase determinada, lo que estamos haciendo es decir cuál es el tamaño de memoria que va a direccionar esa referencia, pero no le estamos asignando realmente ninguna memoria a esa referencia. Para que realmente esa referencia pueda contener valores que le son propios tenemos dos opciones, o bien la asignamos a otra referencia, con lo que la dirección de memoria pasa a ser la de la otra referencia y las dos comparten la misma memoria (una única instancia), o utilizamos el operador `new` para reservar nueva memoria que será asignada a la referencia en función del tamaño de la clase.

6.2.2. Acceso a los miembros de una clase

El concepto de acceso a los miembros de una clase define dos ideas que son diametralmente opuestas. Por un lado, cuando se habla de acceder a los miembros de una clase, realmente se habla de acceder a los miembros de una instancia (es decir, de un objeto) y por otra parte en Java pueden definirse atributos o métodos que son de la clase y no necesitan de una instancia u objeto para ser llamados. Pospondremos este segundo tipo de miembros, "de clase", al Apartado 5.2.8., y nos centraremos ahora en los miembros de objeto o instancia.

La primera idea expresada como "acceso a miembros" realmente lleva aparejada dos ideas en el acceso a los miembros de un objeto: aquella zona de memoria reservada expresamente para el objeto y donde va a almacenar valores particulares del objeto (atributos), y una zona de memoria reservada para la clase donde se encuentran los métodos comunes a todos los objetos de la misma clase. Para acceder a los miembros de una instancia, tanto atributos como métodos, debemos utilizar el operador `.`. No existe diferencia entre métodos y atributos. Ahora bien, para acceder a los miembros éstos deben estar definidos en la interfaz de la clase, es decir, deben ser públicos. El especificador `public` define un conjunto de miembros de la clase como de uso público, accesible desde fuera de la definición de la clase. Decir que los miembros de la clase son públicos, es lo mismo que decir que se puede acceder a ellos desde otras funciones, tanto situadas dentro como fuera de la clase. El especificador de acceso se aplica al miembro que viene después de él. A diferencia de C++, no se puede aplicar un especificador seguido de `:` para afectar a todos los miembros definidos a continuación.

Cuando una clase está compuesta a su vez de objetos de otra clase para acceder a los atributos del objeto que está contenido se utiliza el mismo operador `.`.

6.2.3. Miembros públicos y privados

Si recordamos el concepto de encapsulación, se debe prohibir el acceso a las estructuras internas usadas en la implementación de una clase. El acceso a los datos de una clase debe hacerse a través de métodos de clase especialmente diseñados e implementados para ello. Esto es así porque los atributos de una clase suelen definirse como privados a esa clase y el único camino para llegar a ellos es a través de la interfaz pública de la clase que suelen ser los métodos. Evidentemente, también pueden definirse atributos públicos de la clase, de manera que el acceso a estos campos es directo a través del nombre del objeto. Por defecto, si no se especifica nada, los atributos y métodos son públicos para el resto de clases existentes en el mismo paquete y son privados para cualquier clase que se encuentre fuera del paquete. A este calificador implícito que se utiliza en ausencia de otra indicación se le suele denominar calificador "amistoso" (*friendly*).

En resumen, una forma de mantener la encapsulación es aplicar un especificador `public` a las funciones miembro pero denegando el acceso a los datos miembro usados por esas funciones mediante el calificador `private`.

En todas las clases se deben definir métodos para acceso a los atributos privados, de esta manera cuando una clase contiene un objeto de otra clase el acceso a los atributos de este objeto se realiza mediante llamadas a métodos.

6.2.4. Constructores

La manera de inicializar un objeto con atributos privados está restringida a la definición de un método que asigne unos valores iniciales. Una forma mejor de realizar este procedimiento consiste en definir un *constructor* que no es más que una función miembro especial que tiene el mismo nombre que la clase. Este método se invoca automáticamente cuando se crea una instancia de la clase. El constructor puede inicializar los datos miembro así como realizar cualquier otra tarea de inicialización necesaria para ese objeto, en concreto, cuando los atributos de la clase son referencias para los que se requiere una petición de memoria. Una clase puede tener varios constructores. El constructor nunca devuelve un valor, puesto que siempre devolverá una referencia a una instancia de la clase.

Al realizar la definición se crea una instancia de la clase, es decir, se reserva memoria para un objeto del tamaño de "Cuadrado", y se invoca al constructor que procede a rellenar la estructura según lo indicado en el código del método. Esta construcción automática, cuando se crea una instancia con el operador `new` es una gran ventaja frente a los otros métodos de petición de memoria de otros lenguajes.

En general, cuando se crea una instancia de una clase no siempre se desean pasar los parámetros de inicialización al construirlo, por ello existe un tipo especial de constructores que son los llamados constructores por defecto. Estos constructores no llevan parámetros e inicializan los datos miembros asignándoles valores por defecto.

Si no se define un constructor para una determinada clase, el compilador generará un constructor por defecto, aunque éste no asignará valores de inicialización a los datos miembro. Cabe

decir en este punto que, en ausencia de una inicialización explícita, en Java se inicializan todas las variables numéricas a 0 y todas las referencias a objetos a `null`. Si una clase tiene un constructor por defecto (tanto explícitamente así definido como definido por el compilador) podemos definir un objeto sin necesidad de pasar parámetros.

Con respecto a los constructores por defecto, hay que tener en cuenta lo siguiente: únicamente crea el compilador un constructor por defecto cuando no existe ningún constructor definido por el usuario. En el caso de existir alguno, ya no se crean constructores por defecto, y únicamente pueden utilizarse los constructores existentes definidos por el usuario, respetando la interfaz especificada.

El uso de constructores permite la inicialización de los objetos de una clase utilizando algoritmos complejos. Además de los constructores que se han definido, habitualmente se suele definir un constructor de copia que permite generar el objeto a partir de otro objeto de la misma clase. En el caso de que una clase contenga objetos de otra clase, el constructor de la clase puede especificar valores para el objeto de la clase que contiene, invocando al constructor adecuado.

6.2.5. Sobrecarga de identificador de funciones

Hasta este punto se ha detallado cómo definir una clase que no es más que un conjunto de variables y un conjunto de métodos que el programador necesita. La definición de la clase le permite encapsular y ocultar la información específica de la clase, y a través de los constructores y de los métodos públicos cualquier otro programador puede crear objetos de esa clase y utilizarlos sin necesidad de conocer el código de la clase.

Cuando se desarrollan programas a veces se deben definir operaciones que realizan funciones similares con distintos tipos de datos. Este tipo de operación la hemos visto anteriormente en los constructores donde, en función de los parámetros del constructor, el comportamiento del constructor es distinto. La idea fundamental es que el mismo nombre del método se refiera siempre a la misma operación sobre el objeto, pudiendo ser el código de cada método distinto en función de los parámetros.

En Java se permite que múltiples funciones tengan el mismo identificador, siempre y cuando tengan diferentes parámetros; se distingue entre unas y otras por los diferentes tipos de parámetros que reciben. En la llamada, es el tipo de parámetros el que hace que se invoque a una u otra función. Por supuesto esto es algo pensado para funciones que hacen conceptualmente lo mismo.

Como vemos, según la interfaz empleada en la llamada se invoca automáticamente un método u otro. Para distinguir a qué método se llama se siguen las siguientes reglas:

- Si existe un método que se ajusta exactamente a la llamada (es decir mismo nombre y mismo tipo de cada parámetro) se llama a ese método.
- Si no existe ningún método se promueven las variables al tipo inmediatamente superior (de `char` a `int`, de `int` a `long`, etc.) y se busca algún método que coincida.

Los métodos no se distinguen por el valor de retorno. Es decir, dos métodos no pueden tener el mismo nombre, los mismo parámetros y distinguirse por el tipo del valor de retorno. Esto es así porque en Java está permitido invocar a un método sin asignar el valor de retorno a ninguna expresión, por ejemplo:

```
<nombre de objeto>.<método que retorna un valor>();
```

En ese caso sería imposible determinar cuál de los métodos habría que invocar realmente.

Si el programador desea llamar a un método y los tipos de las variables con las que va a llamar al método no coinciden, deberá hacer un `casting` explícito para que dicho método sea llamado. El compilador comprobará que los tipos (o las clases) de las variables a convertir son compatibles, y además se comprobará en tiempo de ejecución que la conversión es posible, generándose una excepción en caso contrario.

6.2.6. La referencia `this`

Dentro de las funciones miembro no se especifica más que el atributo de la clase que se desea utilizar, pero ¿qué dato es el que realmente está utilizando dicho método? La respuesta es sencilla, como hemos visto los métodos son llamados por instancias de la clase que se crean en primer lugar y cuyos datos, bien mediante un constructor bien mediante una función, se rellena con valores. Una vez creada la instancia, es esta misma la que llama a un método para que se ejecute, y el método está utilizando los valores de dicha instancia en su ejecución.

En realidad podríamos decir que la instancia es un parámetro que siempre figura al llamar al método, dicho parámetro puede ser explícitamente referenciado mediante la referencia `this`, que apunta a la instancia que llama al método.

6.2.7. Métodos de comparación

Además de los métodos que permiten acceder a los atributos privados y de los constructores existe otro tipo de métodos que resulta importante definir: los métodos de comparación. Los tipos básicos de cualquier lenguaje tienen la posibilidad de ser comparados para establecer una relación de menor y mayor. A partir de esta relación se establecen los criterios que permiten la ordenación, búsqueda, inserción y eliminación en estructuras en memoria o en fichero. Cuando se define una clase, el programador realmente está definiendo un nuevo tipo de dato que debe tener funciones para modificarlo, visualizarlo, inicializarlo y, por supuesto, compararlo con otro objeto del mismo tipo. El programador que define la clase define también cuál es el criterio de comparación.

A veces el desarrollador de una clase debe introducir distintas funciones de comparación para permitir que puedan compararse distintos atributos, dejando el uso de estos métodos al programador que vaya a utilizar posteriormente la clase.

6.2.8. Definición de constantes de clase, constantes de objeto, variables globales y métodos de clase

Como hemos visto, la abstracción y la encapsulación son los dos paradigmas de la programación orientada a objetos. Al llevar al límite estas dos ideas y desarrollar un lenguaje como Java totalmente orientado a objetos se pierde cierta flexibilidad de la programación convencional. Evidentemente esta flexibilidad se pierde para favorecer la calidad del software generado. En algunos casos poder romper las rigurosas normas de la POO permite desarrollar un programa de manera más rápida, aunque no es recomendable y, en otros casos, la propia POO es incorrecta para abordar algunos problemas como por ejemplo: ¿de qué objetos son las constantes?, ¿por qué la operación suma debe ser un método? Las constantes tienen un valor invariante en todo el programa y en sí mismas ni son objetos ni pertenecen a ningún objeto. Las operaciones entre objetos realmente no son métodos ya que deben ser simétricas para todos los objetos involucrados y según la POO un método es un mensaje a un objeto por lo que la operación se realiza sobre un objeto y no sobre los dos.

Los miembros constantes se definen en Java a través de la palabra reservada `final`, mientras que los miembros de clase se definen mediante la palabra reservada `static`. De esta manera, si consideramos los miembros atributo, tenemos cuatro posibles combinaciones en Java para indicar si un atributo es constante o no:

- atributos con calificador `static final`: son valores constantes de clase, asignados en la inicialización. Son comunes para todos los objetos de la clase, e incluso pueden invocarse aunque no exista ninguna instancia.
- atributos `final`: son valores constantes, pero potencialmente distintos en cada una de las instancias. Su valor se inicializa en la fase de construcción del objeto y ya no puede modificarse durante el tiempo de vida de éste.
- atributos `static`: toman valores comunes a todos los objetos existentes y potencialmente variables. Se pueden considerar como variables globales.
- resto: atributos variables, diferentes en cada objeto de la clase.

Los miembros que son globales, o que pertenecen a la clase independientemente de si existen instancias o no, se especifican en Java mediante la palabra reservada `static`. Como vimos, un caso particular son los valores de las constantes, que se definen una vez con la palabra reservada `final` y no pueden modificarse en todo el programa. Los atributos compartidos por todas las instancias de la clase deben inicializarse una sola vez, durante la declaración, y serán modificados por los métodos de la clase referenciados tanto por la clase como por las instancias.

Los atributos globales de la clase deben ser limitados ya que al ser globales dan lugar a múltiples errores de programación que no son fáciles de depurar y además rompen el concepto de la programación estructurada y de la programación orientada a objetos. De igual manera los métodos de clase también rompen la filosofía de la POO. Una manera de intentar controlar estos aspectos es modificar los valores de los atributos globales a través de las funciones de clase y no en los métodos que se ejecutan sobre instancias de clase.

Las constantes de objeto se definen mediante la palabra reservada `final` y son atributos que toman valor en el constructor del objeto y cuyo valor no puede modificarse en el resto del programa. De esta manera cada objeto tiene un atributo de ese tipo pero invariante para él. Este tipo de atributos sirve para identificar a cada objeto de forma única.

Los atributos de tipo `final` pueden inicializarse en la declaración o, en caso contrario, en cada uno de los constructores existentes.

Por último, los métodos de clase identifican operaciones que pertenecen a la clase y que son utilizados a través del nombre de la clase. Se definen mediante la palabra reservada `static` y se utilizan mediante la expresión `<nombre de clase>.<método static>`. Habitualmente son operaciones que no se realizan directamente sobre ningún objeto.

Es importante tener en cuenta que un método de clase únicamente podrá acceder a miembros de clase (estáticos), nunca a miembros de instancia, puesto que no tiene por qué existir ninguna instancia de la clase en el momento en que es invocado.

Por último recordar que:

- La definición de constantes requiere la pertenencia a una clase, por lo que debe definirse una clase para la cuál tenga significado dicha constante.
- La definición de métodos de clase está justificada cuando se requiere romper las normas de la POO para dar el verdadero significado a una operación. Es decir, debe ser una operación que requiera de simetría para los objetos involucrados.
- La definición de variables globales o constantes de objeto no está justificada y su uso debe ser muy limitado.

6.2.9. Interfaces

Una interfaz en Java no es una clase, es una declaración de un conjunto de métodos sin implementación. También puede definir constantes, que implícitamente serán `public`, `static` y `final`. Su utilidad es la de servir como referencia a los programadores de las clases para saber qué funciones deben desarrollar para implementar una interfaz.

Las interfaces son una creación del lenguaje Java que evita problemas en la herencia múltiple, y que al mismo tiempo invalida este concepto. Aunque la utilidad de las interfaces se verá al hablar de diseño orientado a objetos, la idea central es que muchas clases puedan compartir la misma interfaz; es decir, tiene un conjunto de funciones que les son comunes en su definición pero muy distintas en su implementación. En nuestro caso necesitaremos una interfaz que contenga todas las funciones que se van a utilizar para visualizar una figura, de esta manera todas las clases de figuras que desarrollemos pueden implementar dicha interfaz y podremos visualizar distintas figuras (con algoritmos de visualización diferentes) llamando a las mismas funciones.

Aunque habitualmente el uso de una interfaz está ligada al concepto de herencia, veremos en el próximo capítulo que la herencia no es exactamente lo mismo que el uso que se hace de las interfaces. Incluimos aquí un ejemplo de uso de interfaces y volveremos sobre el uso de interfaces cuando se hable de diseño orientado a objetos.

6.3. Arrays y listas de objetos

6.3.1. Creación de arrays de objetos

Al igual que se ha visto anteriormente la posibilidad de tener arrays de tipos de datos básicos del lenguaje, se pueden definir arrays con objetos de una clase. Existen dos sintaxis posibles:

```
NombreDeLaClase[] Objetos;  
NombreDeLaClase Objetos[];
```

Con esto se crea la referencia al array, "Objetos", y éste a su vez se construye especificando el tamaño deseado con la sintaxis:

```
Objetos = new NombreDeLaClase[N];
```

siendo N cualquier expresión válida con resultado entero (no tiene por qué ser un literal). Estas dos partes de la creación del array (que contendrá N referencias a objetos de la clase) pueden compactarse en una sentencia única:

```
NombreDeLaClase[] Objetos = new NombreDeLaClase[N];  
NombreDeLaClase Objetos[] = new NombreDeLaClase[N];
```

En cualquier caso, es muy importante tener en cuenta que cuando se crea el array el compilador crea una referencia para cada uno de los elementos del array, pero aún no existen las instancias de ninguno de ellos. Para crear los objetos, hay que llamar explícitamente al constructor por cada elemento creado.

Para trabajar con arrays de objetos es necesario definir funciones que permitan comparar objetos, esto es así porque no puede utilizarse directamente el valor de un atributo para realizar algoritmos de ordenación, mezcla, etc. Así para el ejemplo del cuadrado, si queremos poder trabajar con arrays, nos harán falta funciones que nos digan dados dos cuadrados cuál es mayor.

6.3.2. Estructuras de datos implementadas mediante arrays

Las estructuras básicas que permiten organizar la información cuando no está presente una clave que permite ordenar los elementos son las pilas y las colas. Cuando ordenamos los datos mediante una pila el último elemento que insertamos es también el primero que debe salir. La idea que se pretende representar mediante una pila es la de un conjunto de objetos colocados en columna de manera que ha medida que se van almacenando unos encima de otros no se puede tomar un objeto situado debajo hasta que no se quite el objeto superior (véase Figura 6.1.).

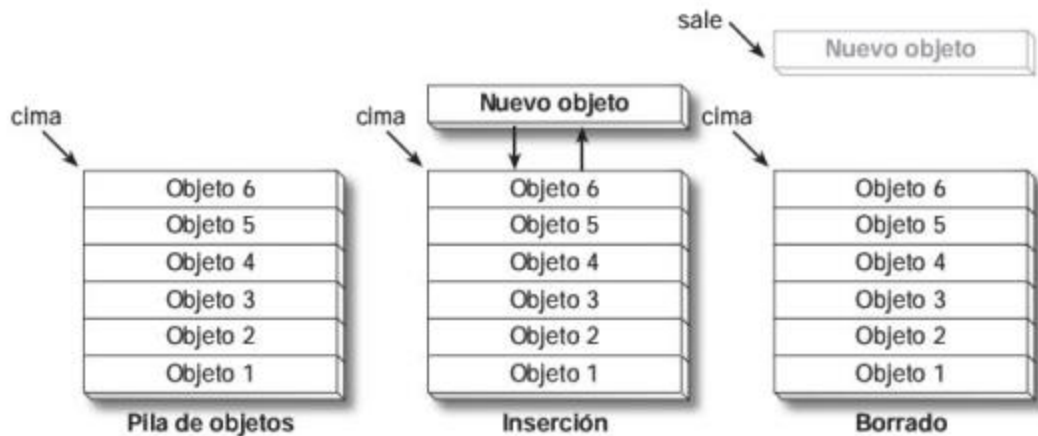


Figura 6.1. Representación de una pila.

La otra forma natural de organizar la información en una estructura dinámica es utilizar una cola. Cuando ordenamos los datos mediante una lista el último elemento que insertamos es también el último que debe salir, y el primero que insertamos debe ser el primero en salir. La idea que se pretende representar mediante una lista es la de un conjunto de objetos colocados en fila de manera que ha medida que se van almacenando unos se van colocando tras los anteriores. De este modo no se puede tomar un objeto situado detrás hasta que no se quite el objeto delantero (véase Figura 6.2.).

En ninguno de los dos casos es necesario buscar la posición donde se debe insertar/borrar/extraer el nodo, puesto que siempre será en la posición indicada por la referencia al principio. Esto hace que los algoritmos de inserción y de borrado/extracción sean más sencillos. En

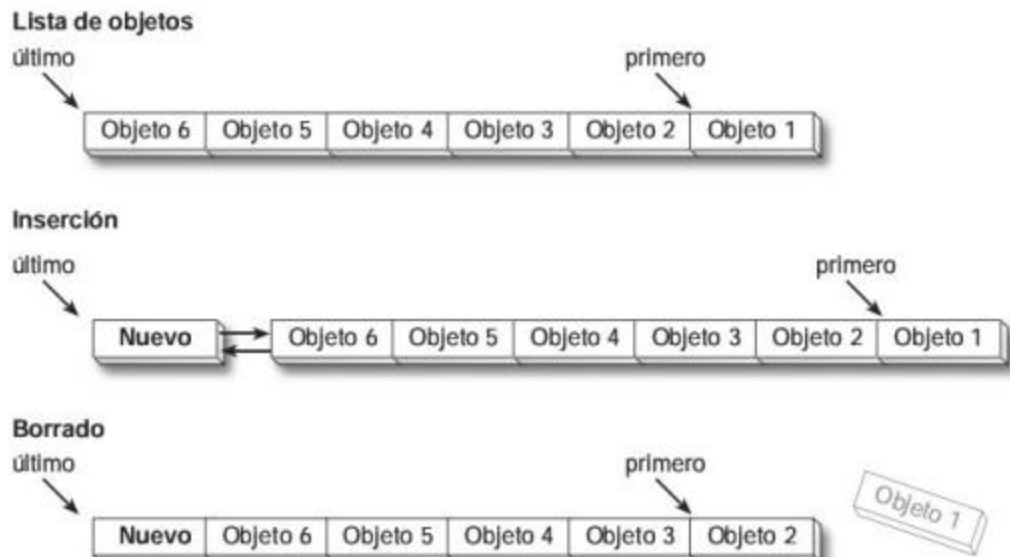


Figura 6.2. Representación de una cola.

general se suelen llamar "Poner" y "Quitar", puesto que no importa la posición real que ocupan. Una pila se puede implementar mediante un array o una lista. Si realizamos la pila mediante un array en memoria estática, la referencia que indica la posición del primer elemento de la pila, que suele denominarse "cima", es en realidad un índice a una posición del array, que indicará la posición del último elemento añadido a la pila. La función "Poner" hará que ese índice se incremente y, recíprocamente, la función "Quitar" que se decremente. Evidentemente, esta solución mediante arrays no es óptima puesto que tenemos que definir el tamaño de la pila *a priori*.

Si realizamos la cola mediante un array en memoria estática, una primera posibilidad sería hacer que la referencia que indique la posición del último elemento sea el índice 0 del array. La función "Poner" hace que se tengan que desplazar hacia la derecha todas las celdas del array, es decir, que tengan que incrementarse en una posición. La referencia que indique la posición del primer elemento será el índice del array, y "Quitar" simplemente decrementará este índice. Existe la posibilidad de implementarla en array al revés, es decir que la primera posición sea la 0 y la última el índice del array. En ese caso el movimiento de celdas se produciría en "Quitar", y "Poner" únicamente incrementará dicho índice del array.

Sin embargo, existe una posibilidad de implementar una cola eficiente con arrays que garantice tiempo constante para las operaciones de "Poner" y "Quitar". Esta solución se denomina implementación mediante "array circular", y define un par de índices, "frente" y "final", que marcan el primero y último de la cola. Estos índices se van desplazando con las operaciones de poner y quitar elementos en la cola, de manera que ésta va "circulando" por el array, y cada vez que uno de los índices alcanza la posición $N - 1$, vuelve a comenzar por la posición 0 (siempre que no se sobrepase el máximo número de elementos que caben en la cola, N). En la Figura 6.3. se ilustra la idea.

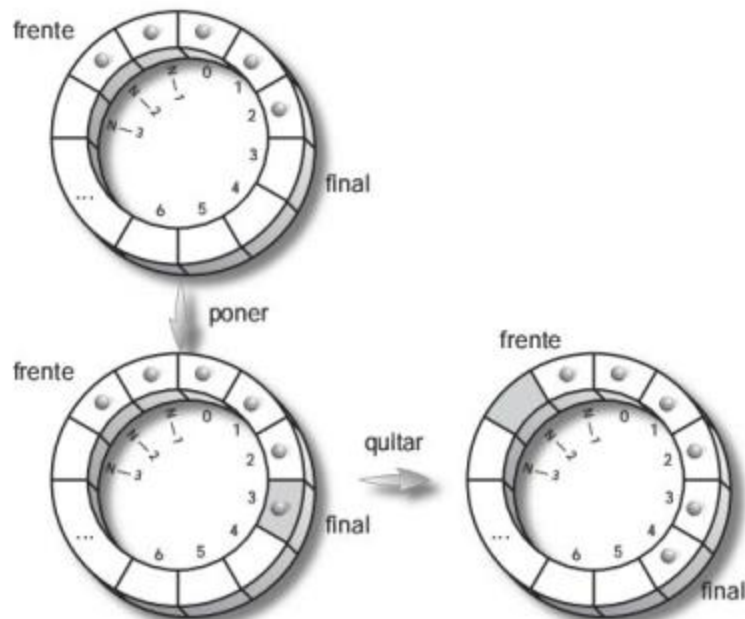


Figura 6.3. Implementación de una cola con un array circular.

Aunque aquí se ha desarrollado una pila y una cola haciendo uso de arrays, ésta no es la mejor manera de hacerlo ya que el número máximo de elementos está limitado desde el principio. Para desarrollar una pila o una cola que pueda crecer en función de la memoria disponible debe estar implementada sobre listas en memoria dinámica. En el siguiente apartado se estudiarán dichas listas.

6.3.3. Listas en memoria dinámica

Una lista es un conjunto de nodos que están relacionados. Cada nodo consta de dos partes: una donde se almacena el objeto (que contiene la información) y otra donde se guarda una referencia al siguiente nodo (Figura 6.4.). De este modo a medida que se van introduciendo elementos de información se pueden ir creando en memoria más nodos, de manera que todos estén relacionados. La ventaja fundamental de esta representación es que no se necesita conocer *a priori* la cantidad de datos que se van a manejar puesto que a medida que se van introduciendo se va creando la estructura en memoria.



Figura 6.4. Representación de un nodo de una lista.

Cuando decimos que se guarda una referencia a otro nodo estamos hablando de una referencia a un nodo, de este modo cada nodo apunta al siguiente nodo. Podemos hacer una analogía con el caso de la representación mediante arrays en la que la relación entre los elementos del array es una relación de orden indicada por el índice que permite indexar dicho array. La sintaxis de la definición de un nodo se compone de una clase donde se almacena la información y de la parte necesaria para guardar la referencia al siguiente:

```
class nodo
{
    <nombre de clase> <nombre de atributo>;
    nodo siguiente;
};
```

Si cada nodo únicamente tiene una referencia al nodo siguiente se dice que la lista está enlazada de forma simple. Cuando se tienen dos referencias, una al nodo anterior y otra al siguiente, se dice que la lista está doblemente enlazada. Una u otra forma de organización presenta sus ventajas y sus inconvenientes a la hora de insertar y borrar nodos. Los nodos de una lista doblemente enlazada tienen la siguiente sintaxis:


```

class nodo
{
    <nombre de clase> <nombre de atributo>;
    nodo siguiente;
    nodo anterior;
};

```

Podemos ver una representación gráfica de los tipos de listas en la Figura 6.5.

Lista simple



Lista doblemente enlazada



Figura 6.5. Representación de los distintos tipos de listas.

Además de la división de las listas en simples y doblemente enlazadas, se puede establecer otra división en función de la referencia que guarda el último elemento de la lista. Si la referencia del último elemento de la lista apunta al principio de la lista se dice que la lista es circular. Cuando esto no ocurre la referencia debe tener el valor 0, este valor concreto representa que no está apuntando a ninguna instancia de clase y es necesario para poder recorrer la lista.

En cualquier caso es imprescindible tener siempre una referencia al principio de la lista, si se pierde esa referencia se pierde el contenido de toda la lista. Todas las funciones sobre listas necesitan de esa referencia para conocer el comienzo. El final de la lista viene marcado por `null` en un caso o por la coincidencia con el principio en el caso de las circulares.

En particular vamos a estudiar los procedimientos de inserción y borrado en las listas simples (en las doblemente enlazadas el procedimiento es similar). En el caso de las listas, la inserción y el borrado no llevan asociada la idea de recolocación, como en el caso de los arrays, puesto que en este caso cualquiera de las dos operaciones produce una reordenación de

las referencias afectadas por la posición de inserción/borrado sin afectar a los nodos anteriores y posteriores.

Al trabajar con listas el almacenamiento de la información, se hace de forma ordenada debido a lo dificultoso de realizar una ordenación en una lista (el recorrido siempre es obligatoriamente secuencial). Por ello los algoritmos de inserción y borrado tienen siempre que primero buscar la posición dentro de la lista donde se va a realizar la operación. En el desarrollo que vamos a realizar se utilizarán listas con la última referencia a `null`, si se desean utilizar listas circulares no es necesario más que el cambio de la condición de terminación: "dirección del siguiente nodo = `null`" por "dirección del siguiente nodo = dirección del primer nodo".

Para insertar un nodo en una lista simple debemos tener una referencia al nodo inmediatamente anterior del que deseamos introducir. De ese modo podemos reorganizar las direcciones haciendo que la referencia del nuevo nodo apunte al siguiente del inmediatamente anterior y la referencia al siguiente nodo de la lista del inmediatamente anterior apunte al nuevo nodo.

La primera operación es una búsqueda en una lista del elemento anterior, para ello se van comparando las informaciones del nodo que deseamos insertar con las informaciones almacenadas en la lista. Una vez que conocemos el nodo anterior al que deseamos insertar apuntamos la referencia al siguiente del nodo nuevo al siguiente, puesto que debemos colocarlo entre medias de los dos. Una vez que ya tenemos colocada la referencia al nodo siguiente rompemos la referencia del nodo anterior con el siguiente, puesto que ahora el siguiente es el nodo nuevo. Gráficamente lo podemos observar en la Figura 6.6.

Además de la inserción en un sitio intermedio de la lista existen dos casos particulares, cuando el nodo debe ser insertado al principio y cuando debe ser insertado al final. El primer caso ocurre cuando la lista está vacía o cuando la posición del nodo es la primera de la lista, en este caso se modifica el valor de la referencia al principio de la lista. En el caso de insertar al final de la lista es más sencillo puesto que el algoritmo es idéntico con la salvedad de que la referencia de nuevo nodo debe apuntar a `null`. El algoritmo para insertar necesita de una función de búsqueda de la posición anterior que devolverá la posición anterior, 0 si debe ir al principio y un valor que indique si el nodo ya está en la lista.

Finalmente, para realizar el borrado de un elemento necesitamos buscar el elemento que se desea borrar (`nodo_borrar`) y una referencia al elemento anterior. En este caso basta con apuntar la referencia del nodo anterior al que se desea eliminar al siguiente del que se desea borrar. Obsérvese que tenemos un criterio contrario a la hora de buscar que en la función de Insertar, puesto que ahora la condición para poder borrar es encontrar el nodo, mientras que antes el criterio era no encontrar; como se ha indicado, generará el valor `null` si el nodo a borrar es el primero o no está en la lista el nodo buscado.

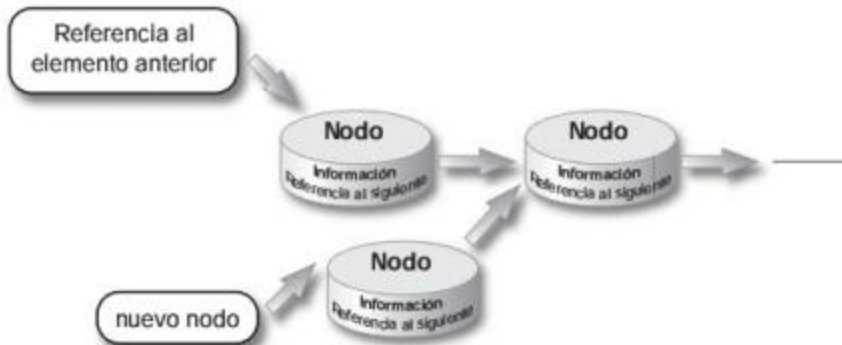
La implementación de esta estructura para una lista simplemente enlazada se propondrá como ejercicio más adelante. Es importante observar aquí que en Java, cuando una instancia no tiene referencia, se libera la memoria que ocupa de forma automática (*garbage collector*), por lo que cuando hablamos de borrar de la lista realmente hablamos de borrar de la memoria. Si se desea utilizar el nodo extraído, debemos tener otra referencia para que, antes de quitar la referencia de la lista a esta instancia, asignemos la referencia y podamos así devolverla al programa usuario de la lista.

Inserción en una lista simple

1. Buscamos la posición donde insertar



2. Colocamos la referencia del nuevo nodo



3. Colocamos la referencia al nuevo nodo

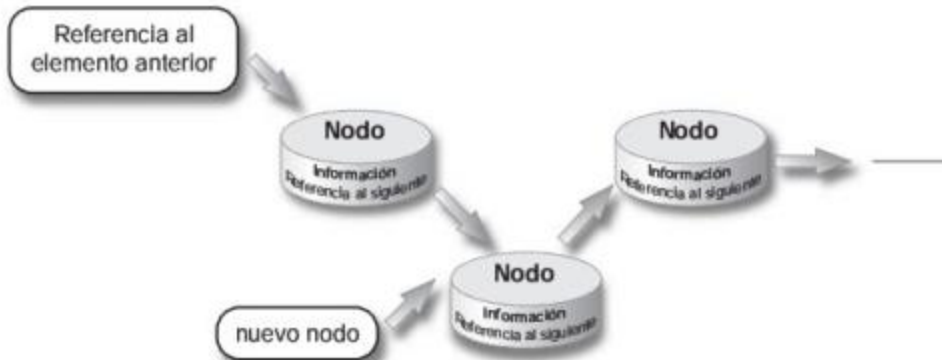


Figura 6.6. Inserción de un nuevo nodo en una lista simple.

6.4. La herencia

Hasta ahora se ha trabajado con clases y objetos definidos específicamente para encapsular un conjunto de variables y operaciones que están relacionadas y que dan lugar a un único concepto. En este capítulo vamos a analizar las ventajas de la POO para definir clases similares a otras y

poder aprovechar en parte el código construido. La herencia es la propiedad más novedosa de la POO y en particular el polimorfismo, que puede definirse como la capacidad del lenguaje para en tiempo de ejecución ensamblar con un código u otro en función de cómo se va desarrollando la ejecución, sin que esté programado *a priori*. Aunque la verdadera potencia de la POO se obtiene cuando se hacen uso de las clases y de las relaciones entre ellas, como se verá en el próximo capítulo, hemos considerado imprescindible analizar en detalle el concepto de herencia detallando las distintas posibilidades de implementación, las distintas relaciones que se establecen entre las clase base y la derivada y entre sus atributos y métodos.

La capacidad para derivar una clase a partir de otra ya definida es uno de los conceptos más importantes y, a la vez, más distintivos de la POO. Cuando se trabaja en un lenguaje procedimental no es posible definir clases, lo más cercano que puede llegarse al concepto de clase y al encapsulamiento es el concepto de estructura y procedimientos asociados. En este tipo de programación el encapsulamiento no es completo, puesto que no existen atributos privados, pero casi es equivalente al que se puede conseguir con una clase.

¿Dónde se encuentra la gran diferencia entre la programación procedimental y la orientada a objetos? Como hemos visto la diferencia fundamental no es el concepto de clase (como podría parecer por el nombre) y el encapsulamiento conseguido. La diferencia fundamental se centra en la capacidad de crear nuevas clases a partir de clases ya construidas reutilizando la mayor parte del código ya desarrollado. En un lenguaje procedimental, una modificación de un tipo definido por el usuario mediante una estructura obliga a modificar todas las funciones que se habían diseñado para dicha estructura. Así, por ejemplo, si consideramos la clase "Cuadrado" como una estructura y decidimos utilizarla para gestionar una aplicación visual con ventanas y deseamos que el cuadrado pueda moverse por la ventana, debemos introducir un atributo más que hará que la nueva estructura ya no sea compatible con la anterior. En los lenguajes procedimentales cada estructura tiene sus funciones y se encapsulan en librerías que no permiten una posterior redefinición de su uso sin una modificación del código. Esto hace que exista una librería para cada estructura que contiene sus procedimientos, cuando las estructuras son similares esta redefinición obliga a una repetición de código. Sin embargo, trabajando con un lenguaje orientado a objetos es posible incluir las modificaciones sin necesidad de modificar el código que ya se ha escrito mediante la herencia. De este modo se evita la duplicación de código y datos derivando una nueva clase a partir de otra ya existente.

Un ejemplo clásico y utilizado en todos los desarrollos de aplicaciones que utilizan Java o C++ aparece cuando se desarrollan interfaces gráficas de usuario. El desarrollo de una interfaz gráfica es un proceso muy laborioso y complicado. En primer lugar no es independiente totalmente del hardware, en segundo lugar cada vez que se abre una ventana hay que gestionar todo el proceso de pintado de la ventana, almacenamiento del trozo de pantalla ocupado por la ventana, movimiento de la ventana con el consiguiente problema de pintado de zonas que aparecen y almacenaje de nuevas, etc. Todos estos problemas de bajo nivel pueden llevar el 90 por ciento de la programación de una aplicación en un entorno visual. Gracias a la POO el tiempo de desarrollo se invierte, ya que los suministradores de compiladores ofrecen un conjunto de clases que permiten desarrollar interfaces rápidamente. ¿Cuál es el fundamento de estas clases? La herencia. Cuando un programador debe implementar un diálogo ya no debe programarlo desde el principio, existe una clase diálogo genérica que se encarga de gestionar todos los problemas gráficos

que puedan existir (incluyendo el cerrado del diálogo cuando se pulsa el botón OK); el programador únicamente deberá desarrollar una clase derivada de ésta en la cual insertará un conjunto de atributos que le permitan definir su diálogo, pero la gestión gráfica se realizará a través de los métodos que se encuentran en la clase base.

Vemos que la idea de heredar una clase de otra se centra en la idea de que dada una clase base, el programador deriva de ella agregando particularidades que no tiene la clase base. En una clase derivada se encuentran todos los atributos y métodos de la clase base, como puede observarse en la Figura 6.7.

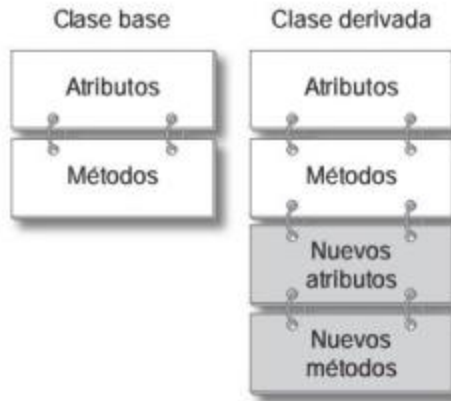


Figura 6.7. La herencia: clase base y derivada.

Como se observa en la Figura 6.7., una clase derivada contiene siempre a la clase base. Uno de los problemas más habituales cuando se comienza a programar con un lenguaje orientado a objetos suele ser la equivalencia entre clase base y derivada. Supongamos que tenemos dos referencias, *p* y *q*, donde *p* es una referencia a un objeto de la clase base y *q* una referencia a un objeto de la clase derivada. ¿Puede *p* referenciar un objeto de la clase derivada? ¿Puede *q* referenciar un objeto de la clase base? ¿*p* y *q* sólo pueden referenciar objetos de la clase para la que han sido definidos? La respuesta a estas preguntas es única, pero no parece ser trivial a la vista de la confusión que suele crear al programador. Habitualmente, cuando se hacen estas preguntas, el programador suele responder: como *q* apunta a un objeto "mayor" que *p*, parece que lo lógico es que *q* pueda apuntar a un objeto de la clase base. Sin embargo, esta respuesta no es válida, lógicamente si *q* pudiera referenciar a un objeto de la clase base, ¿cómo podría el compilador resolver las llamadas a los nuevos atributos y métodos?, eso sería imposible. La respuesta adecuada es que *p* puede referenciar a un objeto de la clase derivada, ya que el compilador siempre sabe que en un objeto de la clase derivada siempre se encuentran los atributos y métodos que existen en la clase base. Esta idea es fundamental para que posteriormente pueda entenderse la propiedad del polimorfismo que se verá al final del capítulo.

En la Figura 6.7. sólo se ha considerado la posibilidad de añadir nuevos atributos o métodos, pero también es posible sobrescribir métodos de la clase base en la clase derivada, como se muestra en la Figura 6.8. Lo que resulta imposible es eliminar los atributos o los métodos de la clase base.

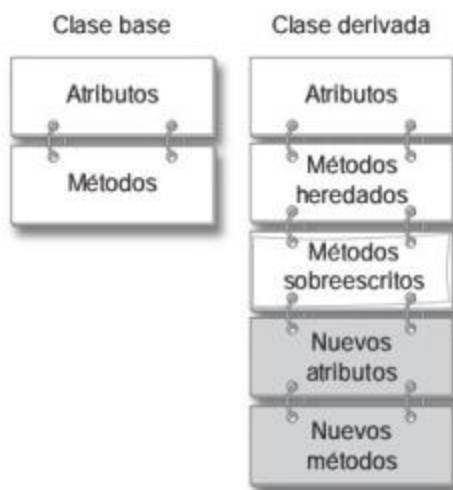


Figura 6.8. La herencia: métodos heredados y sobreescritos.

La capacidad para heredar métodos, es decir, utilizar el mismo código en la clase derivada, y de sobrescribirlos, es decir, modificar aquellos que no se adaptan a lo que se desea, es lo que facilita la reusabilidad del código. Además, aunque se sobrescriba un método es posible utilizar el método de la clase base desde el método sobreescrito, lo cual permite reutilizar parte del código aunque no se adapte exactamente.

Además de la idea de reusabilidad, otro de los posibles usos de la herencia es utilizar una clase base como patrón para un conjunto de clases que deriven de ella. La herencia fuerza a tener, al menos, los mismo atributos y métodos que la clase base. Haciendo uso de esta propiedad es posible definir una clase base (llegando al extremo podría ser abstracta y no poder instanciar ningún objeto de ella) que únicamente marque qué métodos y atributos deben tener un conjunto de posibles clases derivadas.

Hasta este punto hemos hablado del concepto de herencia en general, válido para cualquier lenguaje orientado a objetos; en los puntos que siguen se irán implementado estas ideas en Java.

6.5. Clases derivadas

6.5.1. Implementación de clases derivadas

La declaración de la herencia se hace a través de la palabra reservada `extends`. Para declarar una clase derivada se utiliza la sintaxis:

```
class <clase derivada> extends <clase base>
{
}
}
```

Dentro del ámbito de la nueva clase se definen los nuevos atributos, los nuevos métodos y los métodos sobrescritos:

```
class <clase derivada> extends <clase base>
{
    <calificador> <tipo> <nuevo atributo1>;
    .....
    <calificador> <tipo> <nuevo atributoN>;

    <calificador> <tipo devuelto> <nuevo metodo1>(<parámetros> )
    {
        .....
    }

    .....

    <calificador> <tipo devuelto> <metodo1 clase base>(<parámetros>)
    {
        <nuevo código>
    }

    .....
}
```

Cuando derivamos una nueva clase, como ya se ha dicho, ésta hereda todos los datos y métodos miembros de la clase existente. De esta manera, si no se introduce ningún nuevo atributo ni ningún nuevo método, la nueva clase es exactamente igual que la clase de la que deriva.

Uno de los usos de la herencia es la reutilización de código. También puede utilizarse como patrón de clases derivadas y para ello hay que implementar una clase abstracta que no tendrá ningún significado específico ni ningún método que sea realmente suyo, puesto que no se puede hacer nada sin saber detalles de su implementación, pero sí que puede definir cuál debe ser la implementación mínima de cualquier clase que derive de ella. La definición de esta clase es especial ya que se define así misma como una clase abstracta, mediante la palabra reservada `abstract`, que indica que no pueden crearse objetos de esa clase ya que no tiene sentido.

6.5.2. Constructores en clases derivadas

El constructor de una clase derivada es un método que debe, por un lado, construir una parte que está definida en la clase base, y por otro lado construir los nuevos atributos que se han añadido a la clase derivada. Evidentemente, si fuera necesario construir la clase base en la clase derivada, sería necesario conocer los atributos de la clase (y poder acceder a ellos). Si esto fuera así el

encapsulamiento en programación orientada a objetos desaparecería al aplicar la herencia. Para evitar romper el encapsulamiento de una clase, el constructor de la clase derivada sólo debe construir lo propio de la clase derivada, de manera que la parte correspondiente a la clase base debe ser construida a través de los métodos de la propia clase base.

Como vimos anteriormente se pueden definir funciones para dar valores a los atributos privados de las clases. Estos métodos pueden utilizarse en el constructor de la clase derivada. La definición de un constructor que utiliza métodos para asignar valores en la clase base puede ser sustituida por la utilización del propio constructor de la clase base. Para realizar la llamada al constructor de la clase base se utiliza la palabra reservada `super`.

Así, si un constructor no inicializa explícitamente la clase base, el compilador llama automáticamente al constructor por defecto de la clase base, y si ésta no tiene un constructor por defecto, el compilador producirá un error. Cuando se crea una instancia de una clase derivada el compilador llama a los constructores en el siguiente orden:

1. Constructor de la clase base.
2. Se ejecutan las líneas de código del constructor de la clase derivada.
3. Cada vez que se utilice la palabra `new` en el código del constructor se llamará al constructor correspondiente a ese objeto.

El orden anterior sólo se altera cuando se construyen los atributos en la propia definición de la clase, es decir, cuando tras el nombre del atributo se asigna un objeto

```
class <clase derivada> extends <clase base>
{
    <calificador> <clase> <nuevo atributo1> =
new <clase>(parámetros del constructor);
    .....
    <calificador> <clase> <nuevo atributoN> =
new <clase>(parámetros del constructor);
```

En este caso el orden de llamadas es:

1. Constructor de la clase base.
2. Constructores de los atributos en el orden en que han sido definidos.
3. Se ejecutan las líneas de código del constructor de la clase derivada.
4. Para aquellos objetos que no hayan sido contruidos en la definición de la clase, cada vez que se utilice la palabra `new` en el código del constructor se llamará al constructor correspondiente a ese objeto. Aquellos objetos que son contruidos en la definición no deberían volver a construirse en el constructor, pues se perderían los objetos instanciados la primera vez al volver a asignarse las referencias de los atributos a nuevos objetos.

Esta construcción de objetos en la definición se utiliza raras veces, obsérvese que el constructor es llamado con los mismos parámetros para todos los objetos de una clase dada.

6.5.3. Acceso a los miembros heredados

Aunque una clase derivada hereda los datos miembros de la clase base, no le es posible acceder a ellos de forma directa ya que fueron definidos como privados en la clase base. Esto permite que el encapsulado de cada clase sea el deseado en POO. Cada clase es responsable de los objetos que se crean de ella, y sólo el código de la clase (sus métodos) pueden acceder a los atributos privados de la misma.

Para facilitar la escritura del código, se puede romper el encapsulamiento (a costa de perder capacidad de depuración y limpieza del código) mediante la palabra `protected`. Se puede hacer que los datos miembro de la clase base se definan como `protected` en vez de `private` usando el correspondiente especificador de acceso. De este modo, no son accesibles más que para las clases derivadas.

En cualquier caso si los atributos son privados a una clase, éstos deben ser accesibles a través de métodos dispuestos para esa tarea. De esta manera, siempre que se desee acceder a los atributos desde una clase derivada se deberá hacer uso de esos métodos. Así los programadores de clases que luego son utilizadas por otros programadores pueden depurar el código de su clase y aunque posteriormente otros programadores deriven clases a partir de la suya, nunca se producirán errores en la clase base. Esta forma de programar es muy utilizada en las clases que se entregan con los compiladores para el desarrollo de interfaces gráficas. Para realizar una interfaz, los programadores deben derivar sus clases de las clases preparadas para desarrollo de interfaces pero no pueden acceder a los atributos privados de éstas, de manera que toda la gestión a bajo nivel queda oculta para el programador que sólo debe escribir unos métodos muy generales; que en ningún caso afectan a problemas con el sistema operativo, verdadero cuello de botella para el desarrollo de interfaces gráficas.

6.5.4. Métodos heredados y sobreescritos

Ya hemos visto que una clase derivada tiene los mismos atributos y los mismos métodos que la clase base de la que deriva. Además, se pueden incluir nuevos atributos y nuevos métodos. También se ha visto la necesidad de definir un constructor para la clase derivada y los problemas de acceso a los atributos de la clase base. Además de estos problemas puede ocurrir que alguno de los métodos que existen en la clase base no nos sirvan tal y como están programados.

Así podemos hablar de dos tipos de métodos heredados, aquellos que existen en la clase base y se utilizan con el código de la clase base (métodos heredados) y aquellos métodos de los que se hereda la definición pero no la implementación. Además, se puede redefinir una función de la clase base con una nueva versión en la derivada si ambas funciones son conceptualmente semejantes, pero tiene sentido utilizar la información específica que añade la clase derivada con respecto a la clase base. En resumen:

- Un método es heredado cuando en la clase derivada se puede utilizar de la misma manera que en la clase base. Por ejemplo, podríamos tener una clase "Empleado", derivada de una clase "Persona", en la que reutilizaríamos el método "calcularEdad" de "Persona", puesto que no necesitamos ninguna información específica añadida al pasar a "Empleado".

- Un método es sobrescrito o está reimplementado, cuando debe ser programado de nuevo para que ejecute un conjunto de instrucciones que tengan el mismo sentido para las dos clases. Por ejemplo, podríamos tener un método "visualizar", en el que se presentara el estado completo del objeto con los valores de todos sus atributos. En ese caso, para que tenga sentido visualizar los datos del objeto en cada caso (clase base y derivada) se deben incluir los nuevos atributos. Obsérvese que en este caso particular el método sobrescrito podría reutilizar el método de la clase base, y a continuación añadir los nuevos atributos. En Java podemos acceder a un método definido en la clase base que se haya sobrescrito empleando de nuevo la palabra reservada `super`:

```
class <clase derivada> extends <clase base>
{
    void visualizar()
    {
        super.visualizar();
        // instrucciones para visualizar los nuevos atributos
        ...
    }
    ...
};
```

Aquí hemos visto como los métodos de una clase base pueden ser sobrescritos en una clase derivada. Sin embargo, un aspecto a tener en cuenta es que los métodos de clase, caracterizados por el especificador `static`, así como los métodos con especificador `final` (métodos "constantes") no pueden ser redefinidos en clases derivadas.

6.6. Polimorfismo

La principal característica de la herencia entre clases se denomina polimorfismo. Ya hemos vistos las ventajas de heredar una clase y cómo los atributos y métodos de la clase base son utilizados por la nueva clase. El polimorfismo permite que el programador que utiliza estas clases pueda trabajar con la clase base y posteriormente utilizar la clase derivada sin necesidad de reprogramar nada.

Por ahondar más en el tema, centrémonos en el caso de una clase base, A, y una derivada, B; claramente, ambas clases son distintas. Eso quiere decir que son de dos tipos distintos. En cualquier lenguaje tipado (que es el caso de la mayoría de los lenguajes de programación tradicionales) para definir una variable de un tipo o de otro necesitaremos dos variables distintas. Esto es así porque cada variable ocupará un espacio distinto en función del tipo al que pertenece. En los lenguajes orientados a objetos, que es el caso de Java, un objeto (variable) de la clase derivada es también de la clase base, es decir, es una variable de dos tipos.

¿Cómo es posible que un objeto (variable) pueda ser de dos tipos al mismo tiempo?, esta pregunta podemos responderla desde dos puntos de vista diferentes: filosófico y práctico. Desde el

punto de vista filosófico, claramente, cualquier objeto de la clase B es un objeto de la clase A; es decir, un objeto de la clase B no es más que un objeto de la clase A con alguno añadidos por lo que, ¿cuál es el problema? Desde un punto de vista práctico, sin embargo, los problemas sí son más claros, porque ¿cómo es posible que un objeto (variable) de un tipo pueda ser equivalente a otro tipo que ocupa un tamaño de memoria diferente? La respuesta a todos estos interrogantes la tenemos en el concepto de referencia en Java (que en el caso de C++ se llamaría puntero). Como vimos en el apartado dedicado al operador `new`, este operador reserva la memoria para un objeto B. Como un objeto de la clase B es también de A, es válida la sentencia:

```
<clase A> miobjetoB=new <clase B>();
```

Como ya se vio en aquel apartado cuando se llama al operador `new` se enlaza la referencia con la zona de memoria. A su vez, la zona de memoria es del tamaño indicado por la clase a la que pertenece y los métodos ocupan las direcciones de memoria correspondientes. Las direcciones de memoria de cada atributo y método vienen determinadas internamente por la máquina, y a efectos del programador lo importante es que puede acceder a ellas utilizando el nombre del objeto seguido del operador `.` y seguido del nombre del atributo o método correspondiente. Entonces cuando tenemos una referencia a una clase base si hacemos que dicha referencia apunte a un trozo de memoria ocupado por un objeto de una clase derivada, evidentemente, habrá un desfase en el tamaño de memoria y en el código de los métodos; pero para el compilador es transparente, puesto que la sintaxis `<nombre del objeto seguido del operador "." y seguido del nombre del atributo o método correspondiente>` se mantiene para las dos clases y la resolución de qué zona de memoria exacta ocupa el objeto, no necesita ser conocida por el programador.

De esta manera es posible utilizar en un programa una referencia a un objeto de la clase A y luego realmente trabajar con un objeto de la clase B. Claro que si el programador ya sabe que es un objeto de tipo B, ¿por qué hacer todo esto?, ¿para qué sirve el polimorfismo? La verdadera utilidad de la herencia y el polimorfismo la veremos en el siguiente tema cuando se diseñen las clases y los programas en función de estas dos novedades que introduce la POO.

6.7. Ejercicios resueltos

6.7.1. Ejercicios de definición de objetos

Ejercicio 6.1. Definición de un objeto

Enunciado Definir un objeto que represente a un cuadrado. El objeto debe definirse mediante un conjunto de atributos y métodos, determinando qué constituye su interfaz pública.

Solución El programador de este objeto deberá analizar qué define una variable de ese tipo de manera que al final desarrolle una agrupación de variables que lo defina y un conjunto de métodos que permitan trabajar con él. Un objeto "Cuadrado" podría tener como datos propios: el punto superior izquierdo, el punto inferior derecho, el color, etc. Por supuesto, el desarrollador de la clase podría

definir otros datos propios que también sirvieran para representarlo (como el centro del cuadrado y su diagonal, etc.) sin que esa otra definición altere la forma de trabajar de un programador usuario, en virtud de la propiedad de encapsulamiento. Así, un programador de una aplicación gráfica tendría exactamente la misma funcionalidad y no tendría que cambiar su código aunque la representación interna del cuadrado cambiase.

Además de los atributos el desarrollador de la clase deberá pensar qué métodos pueden hacerle falta al programador de la aplicación que utilizará dicho objeto y, por ejemplo, podría definir como operaciones: *CalcularPerimetro*, *CalcularArea*, *CalcularDiagonal*, etc. Los datos y las operaciones de cada objeto "Cuadrado" estarán particularizados para cada objeto en concreto. Es decir que un objeto "Cuadrado" conoce sus datos propios (por ejemplo, su punto superior izquierdo, color, etc.) y conoce cómo ejecutar las operaciones que le son propias (por ejemplo, *CalcularPerimetro*).

Así, hemos encapsulado los datos (el punto superior izquierdo, el punto inferior derecho, el color) y las operaciones (*CalcularPerimetro*, *CalcularArea*, *CalcularDiagonal*) dentro de la misma barrera conceptual que ha surgido de la abstracción de la idea de una figura cuadrada al objeto "Cuadrado". De esta manera, como queda reflejado en la Figura 6.9., la encapsulación significa que el programador que utiliza un objeto sólo puede ver los servicios que están disponibles en el objeto, pero no la forma en que están realizados.

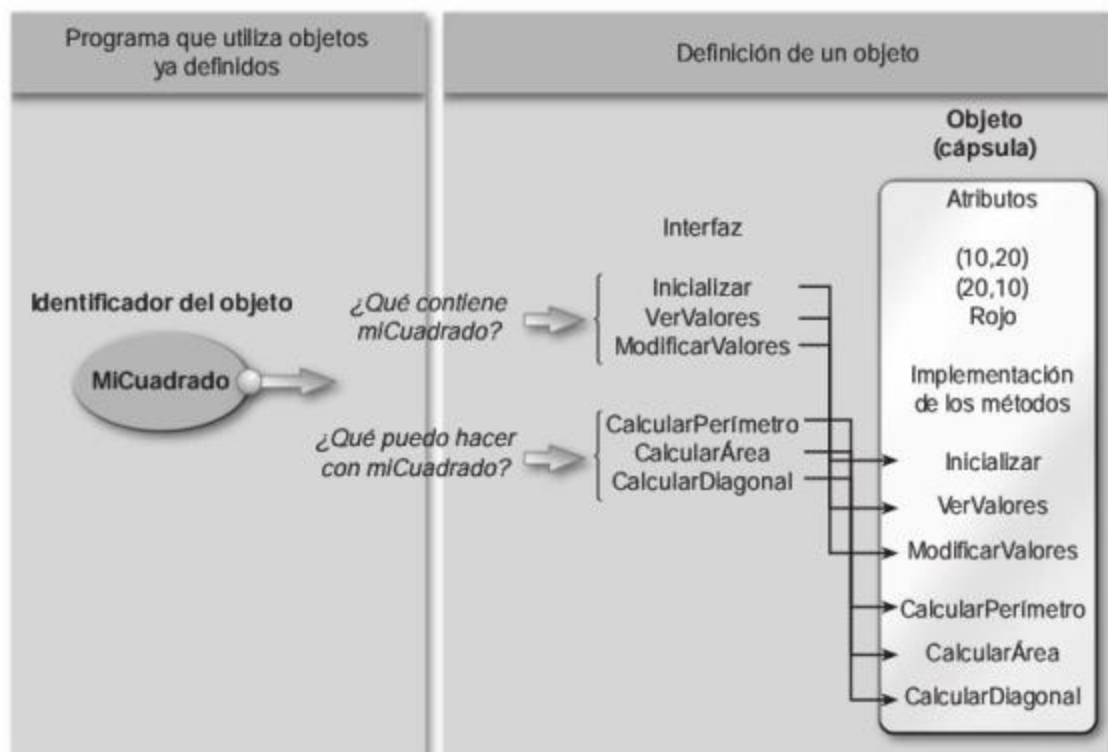


Figura 6.9. Puntos de vista del programador de la aplicación y del objeto "Cuadrado".

Ejercicio 6.2. Redefinición de un objeto

Enunciado Redefinir el objeto definido anteriormente de manera que represente a un cuadrado pero considerando como definición el mismo el centro, la diagonal y un ángulo. Analizar el efecto de la redefinición sobre los métodos definidos en su interfaz pública.

Solución Como queda reflejado en la Figura 6.10., el objeto "Cuadrado" se redefine mediante el centro, el valor de una diagonal y su ángulo respecto a la horizontal.

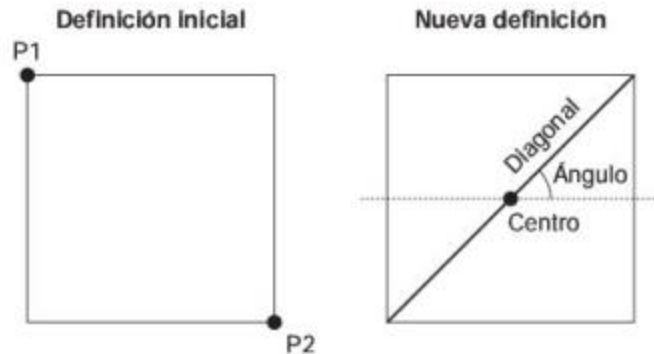


Figura 6.10. Redefinición interna de un objeto.

Puede observarse que si definimos un cuadrado mediante un punto y el valor de un lado (que sería la definición más sencilla posible), no se permitiría definir cuadrados que no estén alineados respecto a la horizontal, mientras que las otras dos definiciones propuestas permiten que el cuadrado pueda estar no alineado (véase Figura 6.11.). Esto supone mayor capacidad de representación a costa de incrementar la complejidad de los métodos para calcular la posición del cuadrado, área, perímetro, etc. Las limitaciones de una definición se establecen en la definición de requisitos que debe cumplir el objeto.

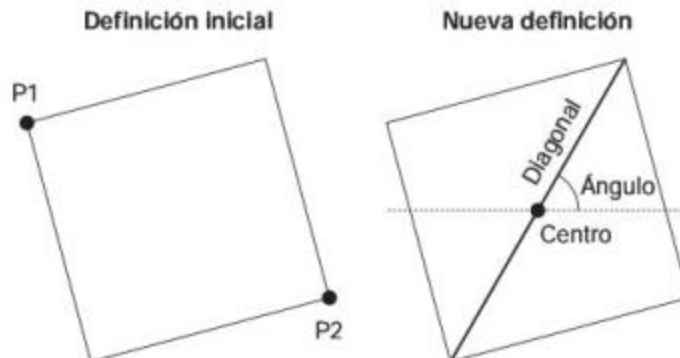


Figura 6.11. Diferentes posibilidades según la definición interna de un objeto.

En cualquier caso, la redefinición del objeto no afecta al programador que lo ha utilizado, puesto que no se ha modificado la declaración del mismo y la interfaz sigue conteniendo las mismas funciones. Ahora bien, el programador que ha desarrollado el objeto tendrá que volver a programar cada uno de los métodos para calcular el perímetro, el área y la diagonal a partir de los nuevos valores. Así por ejemplo, supongamos que el programador que ha utilizado el objeto ha definido el siguiente programa:

```
PROGRAMA EJEMPLO1-OBJETOS

EMPEZAR

DEFINIR OBJETO MiCuadrado

MiCuadrado.ModificarValores(Pto1, Pto2)

IMPRIMIR EN PANTALLA MiCuadrado.Perimetro

FIN
```

Consideramos ahora las dos posibilidades planteadas: definir el cuadrado mediante dos puntos o mediante el centro, la diagonal y el ángulo de giro. El programador del objeto deberá mantener la definición del método `ModificarValores` y del método `Perimetro` para las dos definiciones, de manera que el programador de la aplicación no tenga que modificar su programa. Claramente, la definición del objeto puede mantenerse pero no así su implementación:

- El método `ModificarValores` recibe en su definición dos puntos que directamente modificarán los dos puntos que aparecen como atributos de la clase. Sin embargo, al mantener el método y su interfaz en la segunda definición del objeto "Cuadrado", su implementación cambiará de manera que, para dar valores a los atributos privados, en el método se calcularán el centro, la diagonal y el ángulo a partir de los puntos de entrada, y esos valores se asignarán a los atributos privados del objeto.
- En cuanto a otros métodos, por ejemplo `Perimetro` en el primer caso utilizará los dos puntos para calcular la diagonal y a partir de ese valor calcular un lado y multiplicar por 4, mientras que en el segundo caso podrá calcular el valor del lado, directamente, a partir de la diagonal y luego multiplicar por 4.

Por tanto, la decisión de cómo representar un objeto deberá tener en cuenta la complejidad que supone implementar la funcionalidad del objeto, llegando a una solución lo más razonable posible.

Ejercicio 6.3. Mensajes de un objeto

Enunciado Definir para el método `ModificarValores` de la clase "Cuadrado" qué es un mensaje, el nombre del mensaje y su signatura.

Solución Mensaje

Objeto: Cuadrado
 Mensaje: *todo va bien* ModificarValores(*x_pt1, y_pt1, x_pt2, y_pt2, color*)
 Nombre: ModificarValores
 Signatura: *boolean* ModificarValores(*int, int, int, int, int*)

6.7.2. Ejercicios de clases y objetos en Java

Ejercicio 6.4. Definición de clase

Enunciado Implementar la clase Cuadrado en Java, representando cada objeto mediante dos puntos 2D, y con tres métodos que permitan calcular la diagonal, el perímetro y el área. Implementar a continuación una clase usuario que cree objetos de la clase Cuadrado.

Solución La clase que define el concepto de cuadrado sería:

```
class Cuadrado
{
    double x1, y1, x2, y2;
    //coordenadas 2D de los puntos en la esquina superior
    //izquierda y esquina inferior derecha

    double CalcularDiagonal()
    {
        ....
    }
    double CalcularPerimetro()
    {
        ....
    }
    double CalcularArea()
    {
        ....
    }
}
```

Cada uno de los métodos del concepto cuadrado estarán implementados de la siguiente manera:

- El método CalcularDiagonal calculará la longitud de la diagonal.

```
double CalcularDiagonal()

{
    return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
```

- El método `CalcularPerimetro` calculará la suma de la longitud de los cuatro lados, para ello en primer lugar calculará la diagonal (distancia entre los dos puntos), posteriormente calculará un lado haciendo uso del teorema de Pitágoras (dos lados y una diagonal definen un triángulo rectángulo) y, por último, multiplicará por cuatro el valor de un lado.

```
double CalcularPerimetro()
{
    double diagonal=Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
    double lado = diagonal/Math.sqrt(2);
    return 4*lado;
}
```

- El método `CalcularArea` calculará la superficie ocupada por un cuadrado. Este valor puede calcularse como el valor de un lado al cuadrado o lo que es lo mismo la mitad del cuadrado de la diagonal.

```
double CalcularArea()
{
    double cuadrado_diagonal=(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1);
    return 0.5*cuadrado_diagonal;
}
```

Como puede observarse en el código, la diagonal es calculada en los tres métodos. Para facilitar el trabajo del programador y evitar errores, se puede utilizar el método `double CalcularDiagonal()` en los otros dos métodos:

```
double CalcularPerimetro()
{
    return 4* CalcularDiagonal()/Math.sqrt(2);
}

double CalcularArea()
{
    return 0.5* CalcularDiagonal()*CalcularDiagonal();
}
```

Por último, para crear una instancia de la clase `Cuadrado`, que denominamos `miCuadrado`, utilizamos el operador `new`. El concepto de instancia de una clase es equivalente a decir que se crea un objeto de dicha clase. Así, el objeto `miCuadrado` ocupa su propio espacio de memoria y puede utilizarse para almacenar datos y realizar operaciones con ellos. Así, para crear los dos objetos que nos pide el enunciado tendríamos un programa que define instancias de la clase `Cuadrado`:

```
import Cuadrado;
class UsoCuadrado{
```



```
public static void main (String args []){  
  
    Cuadrado c1=new Cuadrado();  
    Cuadrado c2=new Cuadrado();  
  
}  
}
```

Ejercicio 6.5. Miembros públicos y privados

Enunciado Definir la clase `cuadrado` de manera que todos sus miembros puedan ser accesibles desde el programa principal, que aprovechará esta situación para asignar directamente valores a los atributos. A continuación, redefinir la clase `Cuadrado` con sus atributos privados de manera que sólo sean accesibles mediante unos métodos definidos a tal efecto para visualizarlos, modificarlos o inicializarlos.

Solución Reutilizando en parte el ejercicio anterior, podemos definir la clase `Cuadrado` de manera que pueda ser totalmente accesible en un programa exterior mediante la definición de todos sus componentes, atributos y métodos, como públicos:

```
class Cuadrado{  
public double x1, y1, x2, y2;  
//coordenadas 2D de los puntos en la esquina superior  
//izquierda y esquina inferior derecha  
  
public double CalcularPerimetro()  
{  
    double diagonal=Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));  
    return 4*diagonal/Math.sqrt(2);  
}  
  
public double CalcularArea()  
{  
    double cuadrado_diagonal=(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1);  
    return 0.5*cuadrado_diagonal;  
}  
  
public double CalcularDiagonal()  
{  
    return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));  
}  
}
```

Una vez que los miembros se han hecho públicos, podemos acceder a ellos mediante el operador `.`. Así, la siguiente clase sería un ejemplo de utilización de los métodos públicos de la clase `Cuadrado`:

```

class UsoCuadrado{
public static void main (String args []){

    Cuadrado miCuadrado=new Cuadrado();
    miCuadrado.x1 = 0;
    miCuadrado.y1 = 0;
    miCuadrado.x2 = 100;
    miCuadrado.y2 = 100;
    System.out.println("El perímetro es: "+
    miCuadrado.CalcularPerimetro());
}
}

```

De esta manera rompemos la encapsulación del objeto al acceder a su representación, lo que no es recomendable para asegurar una buena depuración o mantenimiento del código. El concepto de encapsulamiento se implementa si hacemos que la clase oculte su representación interna mediante atributos privados:

```

class Cuadrado{
private double x1, y1, x2, y2;
private int color;

public double CalcularPerimetro()
{
    double diagonal;
    diagonal=Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
    return 4*diagonal/Math.sqrt(2);
}

public double CalcularArea()
{
    double cuadrado_diagonal=(x2-x1)*(x2-x1)+(y2-y1)*(y2-y1);
    return 0.5*cuadrado_diagonal;
}

public double CalcularDiagonal()
{
    return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
}

```

En este caso, los datos son privados con lo que sólo podemos acceder a ellos desde la función miembro. Por tanto, al no poder acceder a los atributos privados de la clase, para poder leer los datos almacenados en los atributos debemos definir una función en la parte pública de la clase que obtenga dichos datos. Por ejemplo:

```

public float VerValor_x1( )
{

```

```

        return x1;
    }

    public float VerValor_y1( )
    {
        return y1;
    }

    ...

```

Por otro lado, si por ejemplo se desea mostrar por pantalla todos los atributos de un objeto, podríamos implementar el método siguiente:

```

public void VerValores()
{
    System.out.print("Atributos: x1="+x1+" y1="+y1+
        " x2="+x1+" y2="+y1+" color="+color);
}

```

Tanto el especificador `public` como `private` afectan únicamente a la declaración del método o atributos que van inmediatamente después. Así, las variables `x1`, `y1`, `x2`, `y2` son de tipo privado por estar contenidas en la misma declaración. Sin embargo, si la declaración hubiese sido

```

private double  x1, y1;
double         x2, y2;
private int     color;

```

las variables `x2`, `y2` no estarían afectadas por el especificador privado, y tendrían, por tanto, el especificador de acceso por defecto: público dentro del paquete y privado en el exterior.

A continuación, se muestra con un ejemplo los accesos permitidos a los miembros de la clase `Cuadrado`:

```

class UsoCuadrado{
    public static void main (String args []){

        Cuadrado miCuadrado=new Cuadrado();

        miCuadrado.color = 5; // ERROR: no se puede acceder a un
                               // miembro privado
        miCuadrado.CalcularPerímetro() // OK
    }
}

```

Aparecen errores al intentar acceder a miembros privados. Si no se puede acceder a los datos miembro, ¿cómo haremos para especificar los datos? Una buena forma de hacer esto consiste en

tener una función miembro definida como pública que reciba los valores deseados y que utilice esos valores para los datos miembro:

```
public void ModificarValores(float x1_p, float y1_p,
float x2_p, float y2_p, int color_p)
{
    x1=x1_p;
    y1=y1_p;
    x2=x2_p;
    y2=y2_p;
    color=color_p;
    // el sufijo '_p' se ha usado para denotar parámetros de entrada
    // al método
}
```

Esta función debe ponerse en la sección pública de la clase Cuadrado de manera que pueda llamarse desde cualquier punto del programa. De este modo tendríamos:

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

    Cuadrado miCuadrado=new Cuadrado();
    MiCuadrado.ModificarValores(0.0,0.0,2.0,2.0,2);
    System.out.println("El perímetro es: "+
    miCuadrado.CalcularPerimetro());
}
}
```

Finalmente, podríamos tener un método público de inicialización que asigne unos valores iniciales a los atributos de la clase mientras no existan unos valores concretos a asignar:

```
public void Inicializar( )
{
    x1=0;y1=0;
    x2=1;y2=1;
    color=0;
}
```

de manera que este método se puede invocar al principio, hasta que se asignen los valores específicos del objeto:

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

    Cuadrado miCuadrado=new Cuadrado();
```



```
        MiCuadrado.Inicializar();  
        ...  
    }  
}
```

Con esta definición las funciones miembro `Inicializar()`, `ModificarValores()` y `VerValores()` pueden acceder a los datos miembro privados de la clase `Cuadrado`. La principal ventaja que ofrece esta manera de definir las clases es la de permitir al diseñador de la clase probar la validez de cualquier valor que se asigne a un dato miembro, sin temor a que dicho dato sea modificado en alguna parte del programa sin llamar al mensaje correspondiente, con lo cual se evitan errores de programación. La otra ventaja es que la modificación del método de cambio de los valores puede ser ocultada al usuario de la clase, de modo que no es necesario conocer la implementación exacta del método para poder utilizarlo.

Ejercicio 6.6. Redefinición de una clase

Enunciado Redefinir la clase `Cuadrado` mediante el centro, la diagonal y el ángulo y modificar los métodos para que tengan el mismo significado que en la definición anterior. Mostrar mediante un ejemplo como redefinir una clase no implica modificar el código que utiliza objetos de dicha clase.

Solución Se modificará la representación del objeto "Cuadrado", como se vio en el Ejercicio 6.4., para en lugar de considerar los atributos que definen dos puntos opuestos (cuatro valores reales), consideraremos el centro, la diagonal y el ángulo (cuatro valores reales). Este cambio en Java no debería afectar a la interfaz de la clase, y de hecho no es necesario modificar las llamadas a las funciones de `Inicializar`, `ModificarValores` ni `VerValores`:

```
class Cuadrado{  
    private double    x, y, diagonal, angulo;  
    private int color;  
  
    public void Inicializar( )  
    {  
        x=0.5;y=0.5;  
        diagonal=Math.sqrt(2);  
        angulo=Math.pi/4;  
    }  
  
    public void ModificarValores(float x1_p, float y1_p, float x2_p, float y2_p,  
    int color_p)  
    {  
        //_p denota parámetros de entrada al método  
        // coordenadas del punto central  
        x=0.5*(x1_p+x2_p);  
        y=0.5*(y1_p+y2_p);  
    }  
}
```

```

        // longitud de la diagonal
        diagonal=Math.sqrt((x2_p-x1_p)*(x2_p-x1_p)+ (y2_p-y1_p)*(y2_p-y1_p));
        // ángulo con el eje horizontal
        angulo=Math.atan2(y2_p-y1_p, x2_p-x1_p);
        color=color_p;
    }

    public void VerValor_x1( )
    {
        return x-0.5*longitud*Math.cos(angulo);
    }
    public void VerValor_y1( )
    {
        return y+0.5*longitud*Math.cos(angulo);
    }
    ...
}

```

Las llamadas a estas funciones quedarán igual en el programa principal, sin necesidad de ninguna modificación adicional, una vez "acondicionada" la clase a la nueva representación:

```

class UsoCuadradoCaja{
public static void main (String args []){

    Cuadrado miCuadrado=new Cuadrado();
    MiCuadrado.Inicializar();
    MiCuadrado.ModificarValores(0.0,0.0,2.0,2.0,2);
    System.out.println("El perímetro es: "+
        miCuadrado.CalcularPerimetro());

    }
}

```

Ejercicio 6.7. Acceso a miembros de objetos agregados en una clase

Enunciado Definir una clase *Fecha* y una clase *Persona* que tenga un atributo de la clase *Fecha* para definir la fecha de nacimiento y realizar un programa que utilice objetos de dichas clases. A continuación, redefinir la clase *Fecha* con atributos privados de manera que sólo sean accesibles mediante métodos definidos a tal efecto. Así mismo, reescribir el código de la clase *Persona* que hace uso de la clase *Fecha*, ahora mediante los métodos públicos destinados al acceso a los atributos privados.

Solución Definiremos la clase para representar objetos de tipo fecha como:

```

class Fecha{
    public int dia,mes,anyo;
}

```

de manera que la clase `Persona` utiliza un objeto de tipo fecha para definir la fecha de nacimiento:

```
class Persona {  
    public String nombre;  
    public Fecha fecNac;  
}
```

Así, con todos los atributos públicos, cuando se define un objeto de tipo persona se puede acceder a la fecha de nacimiento si se utiliza el nombre del objeto y el del atributo:

```
class UsoPersona{  
    public static void main (String args []){  
  
        Persona miPersona=new Persona();  
        miPersona.fecNac.dia = 29;  
        miPersona.fecNac.mes = 1;  
        miPersona.fecNac.anyo = 1970;  
  
        System.out.println("La fecha es: "+  
            miPersona.fecNac.dia +  
            miPersona.fecNac.mes +  
            miPersona.fecNac.anyo);  
    }  
}
```

Ahora implementaremos ambas clases con métodos de acceso que faciliten su manipulación desde el exterior. Para la clase `Fecha` se incluyen métodos individuales para cada atributo, `dia`, `mes` y `anyo`, así como un método, `nuevaFecha`, que modifica todos de una vez:

```
class Fecha{  
    private int dia,mes,anyo;  
  
    public int devDia(){  
        return dia;  
    }  
    public modDia(int p_dia){  
        dia = p_dia;  
    }  
  
    public int devMes(){  
        return mes;  
    }  
    public modMes(int p_mes){  
        mes = p_mes;  
    }  
}
```

```

    public int devAnyo(){
        return anyo;
    }
    public modAnyo(int p_anyo){
        anyo = p_anyo;
    }
    public nuevaFecha(int p_dia, int p_mes, int p_anyo){

        modDia( p_dia);
        modMes( p_mes );
        modAnyo( p_anyo);
    }
}

```

Por tanto, la clase `Persona` a su vez introduce métodos para visualizar y modificar su fecha de nacimiento que accederán a los métodos públicos del atributo `fecNac`:

```

class Persona {
    private String nombre;
    private Fecha fecNac;

    public String devNombre() {
        return nombre;
    }
    public Fecha devFecha(){
        return fecNac;
    }
    public void modifNombre(String n){
        nombre = n;
    }
    public void modFecha(int p_dia, int p_mes, int p_anyo){
        fecNac.nuevaFecha(p_dia, p_mes, p_anyo);
    }
}

```

Ejercicio 6.8. Constructores y referencia `this`

Enunciado Definir dos constructores para la clase `Cuadrado`, uno que sea el constructor por defecto y otro que reciba los valores de los dos puntos y los introduzca en los atributos del cuadrado. Para este último, implementar dos versiones equivalentes, una con los atributos directamente y el otro utilizando la referencia `this`.

Solución Vimos en un ejemplo anterior que la función `Inicializar()` de la clase `Cuadrado` permite acceder a los datos miembro para introducir los datos en el objeto que acabamos de instanciar. Para definir el constructor por defecto podemos utilizar dicho método. La siguiente definición de clase tiene un constructor por defecto que inicializa todos los datos miembro a 0:


```
public Cuadrado()
{
    Inicializar();
}
```

Este método se invoca cada vez que se instancie un objeto de la clase Cuadrado:

```
Cuadrado    miCuadrado=new    Cuadrado();
```

Es importante notar que cuando se declara un objeto sin llamar al operador `new`, únicamente se crea una referencia al objeto, pero aún no se invoca a ningún constructor. Así, la sentencia:

```
Cuadrado    miCuadrado;        // no se ponen paréntesis
```

declara `miCuadrado` como una referencia a una instancia de la clase `Cuadrado`, pero que aún no se ha reservado memoria para el objeto `miCuadrado`. En el momento en el que se crea la instancia, se invocará obligatoriamente a un constructor:

```
miCuadrado=new Cuadrado();    //Se ponen paréntesis
```

Para definir el constructor que modifica los puntos que definen el cuadrado deberemos definir un constructor que tenga cuatro parámetros:

```
class Cuadrado
{
    private ...

    public Cuadrado(float x1_p, float y1_p, float x2_p, float y2_p, int c)
    {
        x1=x1_p;
        y1=y1_p;
        x2=x2_p;
        y2=y2_p;
        color=c;
        ...
    }
}
```

Esto podría hacerse de forma totalmente equivalente utilizando la referencia `this` para distinguir las variables argumento de los atributos del objeto:

```
class Cuadrado
{
    private

    public Cuadrado(float x1, float y1, float x2, float y2, int c)
    {
```

```

        this.x1=x1;
        this.y1=y1;
        this.x2=x2;
        this.y2=y2;
        this.c=c;
        ...
    }
}

```

Cuando se define el objeto se pasan los valores de los parámetros al constructor utilizando una sintaxis como la de cualquier llamada a función:

```
Cuadrado miCuadrado=new Cuadrado(2.0, 2.0 1.0, 1.0, 0);
```

Asimismo, la clase Cuadrado podría también utilizar en su constructor el método ModificarValores, con lo que quedaría:

```

public Cuadrado(float x1_p, float y1_p, float x2_p, float y2_p, int c)
{
    ModificarValores(x1_p, y1_p, x2_p, y2_p, c);
}

```

En este ejemplo concreto, únicamente podrían crearse objetos de la clase Cuadrado con los dos constructores definidos: Cuadrado() y Cuadrado(float, float, float, float, int). El compilador, por tanto, no creará ningún constructor por defecto.

Ejercicio 6.9. Constructores de copia

Enunciado Definir tres constructores para la clase Fecha: uno por defecto (sin argumentos), que genere aleatoriamente la fecha, otro al que se le pasen los valores del día, mes y año que definen la fecha, y un tercer constructor, "de copia", que genere una instancia copia del objeto apuntado por una referencia enviada a través de la interfaz.

Solución Como el número de días que puede tener un mes depende del mes en el que nos encontremos, el constructor no consiste únicamente en generar tres valores, sino que debe garantizar que la fecha sea válida (obsérvese la condición para saber si un año es bisiesto):

```

class Fecha{
    private int dia,mes,anyo;

    // Constructor por defecto:
    Fecha () {
        // genera aleatoriamente una fecha
        anyo = (int) (Math.random()*25 + 1970);
        mes = (int) (Math.random()*12+1);
    }
}

```

```

    if (mes == 1|mes==3|mes==5|mes==7|mes==8|mes==10|mes==12)
        dia=(int) (Math.random()*31+1);
    else if (mes==2)
        if ((anyo%4==0)&& (anyo%100!=0)|| (anyo%400==0))// es bisiesto
            dia=(int) (Math.random()*29+1);
        else dia=(int) (Math.random()*28+1);
    else // meses de 30
        dia=(int) (Math.random()*30+1);
    //System.out.println("Fecha generada aleatoriamente: ");
    //this.visualizar();
}

Fecha(int d, int m, int a){
    dia=d;
    mes=m;
    anyo=a;
}

// Constructor de copia:

Fecha(Fecha f){
    dia=f.devDia();
    mes=f.devMes();
    anyo=f.devAnyo();
}
}

```

En el último constructor, estamos accediendo a los atributos privados del objeto pasado por referencia, "f", a través de los métodos públicos, que es lo más recomendable. Sin embargo, cabe decir que el calificador privado afecta a todos los objetos de clases distintas (incluso las heredadas), pero no a los objetos que sean de la misma clase, que son totalmente accesibles; de manera que la siguiente implementación del constructor de copia también sería válida:

```

Fecha(Fecha f){
    dia=f.dia;
    mes=f.mes;
    anyo=f.anyo;
}

```

Ejercicio 6.10. Constructores

Enunciado Definir un conjunto de constructores para la clase `Persona` utilizando los constructores de la clase `Fecha`.

Solución En el caso de la clase `Persona` que contiene un objeto de la clase `Fecha`, el constructor de persona podrá encargarse de construir también el objeto "Fecha":

```

class Persona {
    private String nombre;
    private Fecha fecNac;

    Persona () {
        // genera aleatoriamente uno de los nombres de la tabla
        String [] tabla =
            {"luis", "pepe", "alberto", "andres", "juan",
             "mariano", "eduardo", "roberto", "beatriz",
             "raquel", "marisa", "julia", "silvia",
             "sonia", "laura", "ana"};
        nombre = new String(tabla[(int) (Math.random()*16)]);
        fecNac=new Fecha();
    }

    Persona (String n){
        nombre=new String(n);
        fecNac=new Fecha();
        // fecha generada aleatoriamente
    }

    Persona(String n,int d,int m, int a){
        nombre=new String (n);
        fecNac = new Fecha (d,m,a);
    }

    Persona (Persona p){
        // crea un objeto persona con los valores del que se le pasa como
        // parametro
        nombre = new String (p.devNombre());
        fecNac = new Fecha (p.devFecha());
    }
}

```

Además, con Java cabe otra posibilidad para inicializar los objetos agregados. Pueden inicializarse antes del constructor, en la propia sentencia de declaración, de manera que al entrar en cada constructor ya se habrían invocado los constructores, y por tanto, existirán instancias de cada uno de ellos. Así con este mismo ejemplo, podríamos haber declarado los objetos agregados "nombre" y "fecNac" como sigue:

```

class Persona {
    private String nombre = new String("");
    private Fecha fecNac = new Fecha();

    Persona () {...}
    Persona (String n){...}
    Persona(String n,int d,int m, int a){...}
}

```



```

    Persona (Persona p){...}
}

```

En el ejemplo se observa cómo el constructor de copia de la clase `Persona` hace uso del constructor de copia de la clase `Fecha`, por ello habitualmente suele definirse un constructor de copia para cada clase, lo que facilita enormemente el desarrollo de clases a partir de clases ya creadas.

Ejercicio 6.11. Sobrecarga de métodos

Enunciado A partir de la clase `Cuadrado`, deseamos definir una operación de giro sobre el cuadrado. Dicha operación quedará definida como: “operación que modifica los extremos del cuadrado de tal manera que el ángulo del cuadrado con la horizontal será distinto”. Ahora bien, dicha operación puede realizarse de muchas maneras en función de cómo se le diga al cuadrado que tiene que girar. Por ejemplo, se puede definir la función `giro` utilizando un ángulo, vectores y la misma operación utilizando un punto con el que se quiere que quede alineado su punto superior izquierdo después del giro. Realizar los dos métodos y utilizarlos en una clase `Ejemplo`.

Solución

```

public void giro(double angulo)
{
    x1=0.5*(x1+x2) - 0.5*(x2-x1)*Math.cos(angulo)
                - 0.5*(y2-y1)*Math.sin(angulo);
    y1=0.5*(y1+y2) - 0.5*(x2-x1)*Math.cos(angulo)
                + 0.5*(y2-y1)*Math.sin(angulo);
    x2=0.5*(x1+x2) + 0.5*(x2-x1)*Math.cos(angulo)
                + 0.5*(y2-y1)*Math.sin(angulo);
    y2=0.5*(y1+y2) + 0.5*(x2-x1)*Math.cos(angulo)
                - 0.5*(y2-y1)*Math.sin(angulo);
}

public void giro(double x1_p, double y1_p)
{
    double xc=0.5*(x1+x2);
    double yc=0.5*(y1+y2);
    x1=x1_p;
    y1=y1_p;
    x2=2*xc-x1;
    y2=2*yc-y1;
}

```

Estas dos funciones se podrían utilizar en un programa, como por ejemplo:

```

import Cuadrado;
class UsoCuadrado{
public static void main (String args []){

```

```

    Cuadrado c1=new Cuadrado();
    Cuadrado c2=new Cuadrado();

    c1.giro(0.73);          //En cada caso se llama a
    c2.giro(1.2,1.3);      //Una función diferente
}
}

```

Ejercicio 6.12. Métodos de comparación

Enunciado Definir un método de comparación para la clase `Fecha`. Comparará el objeto con otro que se le pasa como parámetro para retornar 0 si son iguales, y, en caso contrario, 1 o -1 en función de cuál de las dos fechas sea mayor.

Solución Esta función se ha implementado aprovechando la funcionalidad de ordenación alfanumérica de cadenas que proporciona la clase `String`. Para ello, se convierten las fechas a comparar en objetos `String`, asegurándose de que ambas se representan con el mismo número de dígitos, y a continuación se comparan con el método `compareTo`:

```

public int comparar (Fecha f){
    // devuelve -1 si esta fecha es anterior a f
    // devuelve 0 si las dos fechas son iguales
    // devuelve +1 si esta fecha es posterior a f
    // ponemos las fechas en cadenas del tipo aaaammdd
    // y las comparamos lexicograficamente
    String s1,s2;

    String d1,d2,m1,m2,a1,a2;
    if (dia<10) d1="0"+dia;
    else d1=String.valueOf(dia);
    if (mes<10) m1 = "0"+mes;
    else m1=String.valueOf(mes);
    a1=String.valueOf(ano);

    if (f.devDia()<10) d2="0"+f.devDia();
    else d2=String.valueOf(f.devDia());
    if (f.devMes()<10) m2 = "0"+f.devMes();
    else m2=String.valueOf(f.devMes());
    a2=String.valueOf(f.devAnyo());
    s1=a1+m1+d1;
    s2=a2+m2+d2;
    return s1.compareTo(s2);
}

```

Ejercicio 6.13. Métodos de comparación

Enunciado Utilizar el método de comparación de la clase `Fecha` en la clase `Persona` para realizar un método de comparación por edades de dos personas. Implementar, además, otro método de comparación basado en el nombre.

Solución La definición del método de comparación de la fecha permite que a la hora de utilizar la clase fecha en otra clase, como por ejemplo persona, al comparar objetos de esta nueva clase se pueda utilizar el método que compara fechas:

```
public int compararEdad(Persona p){
    // devuelve un entero < 0 si esta persona es mas joven que p
    // 0 si han nacido el mismo día
    // > 0 si esta persona es mayor que p
    // como al comparar fechas la fecha de nacimiento de la persona
    // mas joven es mayor
    // que la de la persona mayor, invertimos el signo del valor devuelto
    return - (fecNac.comparar(p.devFecha()));
}
```

Desarrollando un método de la clase `Persona` que compare el nombre, al realizar una aplicación que trabaje con personas el programador puede ordenar las personas por edad o por orden alfabético en función de lo que necesite. El método de comparación sería:

```
public int compararNombre(Persona p)
{
    // devuelve un entero < 0 si el nombre de esta
    // persona es menor lexicográficamente
    // que el nombre de p
    // devuelve 0 si son iguales y > 0 si es mayor
    return nombre.compareTo(p.devNombre());
}
```

Ejercicio 6.14. Constantes de clase, de objeto y variables globales

Enunciado Dentro de nuestra clase `Cuadrado`, definir la constante `PI`, que existirá con independencia de las instancias, definir un atributo compartido por todos los objetos que vaya contando el número de instancias que se han definido hasta entonces; definir un atributo constante pero distinto para cada objeto, que vaya identificando cada uno de los cuadrados que se han definido.

Solución Al utilizar los calificadores `static final` en el atributo `PI`, podemos acceder a su valor mediante la expresión `Cuadrado.PI`, sin necesidad de haber creado ningún objeto de la clase `Cuadrado`:

```
class Cuadrado{
    private double x1, y1, x2, y2;
    public static final double PI = 3.14159;
    ...
}
```

Hay que observar que el valor de la constante deberá inicializarse obligatoriamente en la declaración, pero no tiene por qué ser un valor literal (conocido en tiempo de compilación), sino que puede ser una expresión cuyo valor no se sepa hasta el tiempo de ejecución, por ejemplo:

```
static final double RAIZ_DOS = Math.sqrt(2.0);
```

En cuanto al atributo privado con el número de instancia, es una variable global accesible por todos los objetos:

```
class Cuadrado{
    private double    x1, y1, x2, y2;
    public static final double PI =3.14159;
    static private int numero;                // VARIABLE GLOBAL
    ...
};
```

Esta variable podría incrementarse cada vez que se invoca a un constructor:

```
Cuadrado(...)
{
    numero++;
    ...
}
```

Sin embargo, cualquier instancia podría manipular su valor y afectar al resto, por lo que el uso de variables globales debe ser muy limitado. Finalmente, el identificador constante de cada objeto se indica con un especificador `final`:

```
class Cuadrado{
    private double    x1, y1, x2, y2;
    public static final double PI =3.14159;
    static private int numero;                // VARIABLE GLOBAL
    final int identificador;                  // CONSTANTE DE OBJETO
    ...
};
```

Obsérvese que tenemos una constante que no ha sido inicializada, por ser dependiente del objeto concreto. Por tanto, su valor se asignará en el constructor, no estando permitida su modificación posterior

Ejercicio 6.15. Métodos de clase

Enunciado Definir la operación suma de dos cuadrados, que genere un objeto con su posición en las coordenadas resultado de sumar las coordenadas de cada argumento.

Solución Podría pensarse en hacer un método de instancia que tomara sólo un cuadrado como argumento para generar el resultado pedido a partir del argumento y de la propia instancia. Sin embargo, se implementa aquí un método de clase que recibe los dos cuadrados como argumento:


```
class Cuadrado{
.....

static public Cuadrado suma(Cuadrado c1, Cuadrado c2 )
{
    Cuadrado lasuma = new Cuadrado;
    lasuma.ModificarValores(c1.VerValor_x1( )+ c2.VerValor_x1( ),
        c1.VerValor_y1( )+ c2.VerValor_y1( ),
        c1.VerValor_x2( )+ c2.VerValor_x2( ),
        c1.VerValor_y2( )+ c2.VerValor_y2( ));
    return lasuma;
}
```

Obsérvese que sería ilegal invocar al método `ModificarValores` de la clase, porque es un método no estático que asume una instancia determinada con atributos a modificar. Sin embargo, al declarar el método `suma` como estático, no tiene por qué existir una instancia específica sobre la que se actúe, sino que se puede invocar sin existir ninguna (la llamada podría ser `Cuadrado.suma`). Hay que crear, por tanto, una instancia de un cuadrado, `lasuma`, sobre la que sí se pueden invocar métodos no estáticos. Por la misma razón, tampoco podría accederse a la referencia `this`.

6.7.3. Ejercicios de arrays y listas de objetos

Ejercicio 6.16. Definición de un array

Enunciado Crear un array de 10 objetos de tipo cuadrado.

Solución Las dos sentencias siguientes son válidas para crear un array de 10 objetos de tipo cuadrado:

```
Cuadrado[] MisCuadrados=new Cuadrado[10];
Cuadrado MisCuadrados[]=new Cuadrado[10];
```

Para crear los cuadrados correspondientes a cada una de las posiciones del array, es necesario utilizar el operador `new` por cada objeto:

```
Cuadrado[] MisCuadrados=new Cuadrado[10];

for (int i=0; i<10; i++)
    MisCuadrados[i]=new Cuadrado();
```

Para utilizar el array deberíamos realizar varias posibles comparaciones en función del color, de la posición que ocupa en el eje *x*, y de la posición que ocupa en el eje *y*.

Ejercicio 6.17. Gestión de un array

Enunciado Crear una clase que permita gestionar un array de objetos de la clase `Persona`, con métodos para creación, inserción, borrado, visualización y ordenación por nombre y edad.

Solución En muchos casos la utilización de arrays de objetos puede encapsularse mediante una clase cuyo atributo sea un array de objetos. Para crear una clase array de objetos debemos suministrar un constructor del array, métodos de inserción de elementos, de borrado, de ordenación y de visualización. La definición de los atributos sería:

```
class ListaOrdenada {
    private Persona [] lo;
    private int numPersonas;
    .....
}
```

De este modo, en el array cabe un número máximo de personas que deberá especificarse en el momento de la construcción, mientras que el atributo `numPersonas` indica cuántas hay realmente en el momento actual. Un constructor por defecto podría indicar un tamaño máximo del array y crearlo:

```
ListaOrdenada() {
    // constructor por defecto
    static final numMaximoPersonas=100;
    lo=new Persona[numMaximoPersonas];
    numPersonas=0;
}
}
```

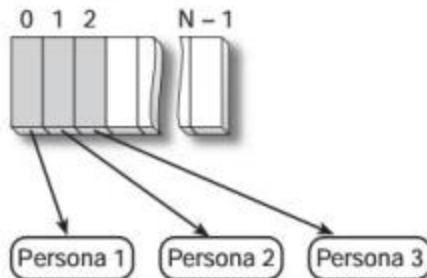
Con esta representación, los métodos para añadir y borrar elementos de la lista simplemente cambiarán el atributo `numPersonas`, y los valores que toman las referencias, tal y como se indica en la Figura 6.12.

Por tanto, un método para añadir simplemente extendería el número de personas en una unidad y lo insertaría por el final:

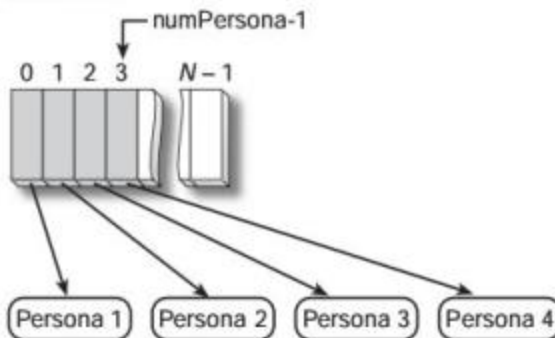
```
anyadir (Persona p)
{
    lo[numPersonas++]=p;
}
```

mientras que un método para borrar debería localizar a la persona y a continuación reubicar los elementos del array. Obsérvese que sólo hay que desplazar los que están a la derecha del elemento encontrado, cuyo nuevo índice estará entre la posición que ocupaba el elemento a borrar y la posición con el número de personas menos 2, puesto que se elimina un elemento.

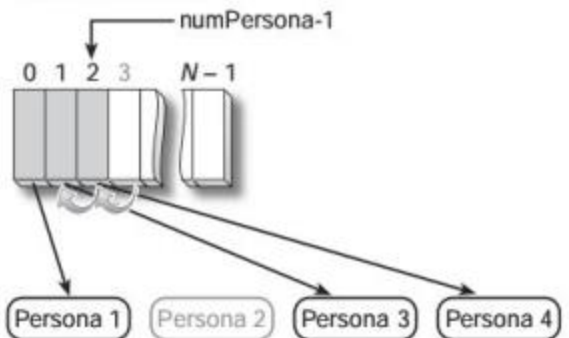
Lista de personas



Inserción



Borrar: Persona 2

Figura 6.12. Implementación de la lista de personas con un array de tamaño N .

```

borrar (Persona p)
{
    int i=0;
    boolean encontrado=false;
    while (!encontrado && i<numPersonas)
    {
        encontrado=(p.compararNombre(lo[i])==0) &&
                    (p.compararEdad(lo[i])==0);
        i++;
    }
    if (encontrado)
    {
        // i apunta a la posición del siguiente elemento
        // se desplazan una unidad a la izda todos los elementos
        for (int j=i-1; j<numPersonas-1; j++)
            lo[j]=lo[j+1];
        numPersonas--;
    }
}

```

Podemos incluir algoritmos de ordenación por distintos atributos, ya que como vimos anteriormente las personas pueden compararse por la edad o por el nombre:

```
public void ordenarNombre() {
    Persona aux;
    for (int i=1; i<lo.length; i++)
        for (int j=0; j<lo.length - i; j++)
            if (lo[j].compararNombre(lo[j+1])>0){
                aux = lo[j];
                lo[j]=lo[j+1];
                lo[j+1]=aux;
            }
}

public void ordenarEdad() {
    Persona aux;
    for (int i=1; i<lo.length; i++)
        for (int j=0; j<lo.length - i; j++)
            if (lo[j].compararEdad(lo[j+1])>0){
                aux = lo[j];
                lo[j]=lo[j+1];
                lo[j+1]=aux;
            }
}
```

Por último, podemos definir un constructor de copia que además de copiar los valores de un array de personas en nuestra clase los ordene por nombre, haciendo uso de las funciones anteriores:

```
ListaOrdenada(Persona [] l){
    // recibe un array de objetos Persona
    lo=new Persona[l.length];
    for(int i=0;i<l.length;i++)
        lo[i]=new Persona(l[i]);
    // copia en la lista ordenada los objetos Persona del array de objetos.
    ordenarNombre();
}
}
```

Sin embargo, obsérvese que en este caso el tamaño del array viene dado por el array de copia, y no se pueden añadir nuevos elementos.

Finalmente, podríamos definir dos métodos, uno que retorne el array, para poder trabajar con él de forma externa a la clase, y otro que lo visualice por pantalla:

```
public Persona[] obtener(){
    // devuelve una referencia al array de objetos Persona
```



```

        return lo;
    }
    public void visualizar(){
        // visualiza la lista
        for (int i=0;i<lo.length;i++)
            lo[i].visualizar();
    }
}

```

La utilidad de una clase que gestiona el array aparece cuando se utiliza en un programa donde ya no es necesario utilizar los algoritmos de ordenación sobre el array si no utilizar directamente los métodos definidos para ese fin. De esta manera uno puede crear un array de personas y crear un objeto `ListaOrdenada` a partir de ese array. Al crear el objeto el array se ordena por nombre (posteriormente se puede reordenar por fecha) y se puede gestionar desde el objeto `ListaOrdenada`:

```

public class ArrayObjetos{

    public static void main(String[] args){
        // primero construimos el array de referencias a objetos
        Persona [] arrayPersonas =new Persona [100];
        // ahora construimos cada uno de los 10 objetos
        for(int i=0;i<arrayPersonas.length;i++)
            arrayPersonas[i]=new Persona ();
        ListaOrdenada lista=new ListaOrdenada(arrayPersonas);
        System.out.println("Visualizar lista ordenada:");
        lista.visualizar();
        lista.ordenarEdad();
        System.out.println("Visualizar lista ordenada por edad:");
        lista.visualizar();

    }
}

```

Ejercicio 6.18. Estructuras de datos mediante arrays

Enunciado Desarrollar una clase que implemente una pila de enteros mediante un array y utilizarla en un ejemplo de programa que apile y desapile números.

Solución En este caso, como sólo está permitido acceder al elemento en la posición de la cima de la pila, únicamente basta con cambiar el atributo `cima`, que se corresponde con el número de elementos apilados menos 1:

```

class Pila{
    private int[] p;

```

```

private int cima;

Pila(int n){
    p=new int[n];
    cima=-1;
}
boolean vacia(){
    return cima==--1;
}
boolean llena() {
    return cima==p.length-1;
}
void Poner(int elem){
    if (!llena())
        p[++cima]=elem;
}
int Quitar() {
    if (!vacía())
        return p[cima--];
    return -1;
}
}

public class TestPila {
    public static void main(String[] args){
        Pila t=new Pila(3);

        t.Poner(1);
        t.Poner(2);
        t.Poner(3);
        t.Poner(4);
        System.out.println(t.llena());
        System.out.println(t.Quitar());
        System.out.println(t.Quitar());
        System.out.println(t.Quitar());
        System.out.println(t.Quitar());
    }
}

```

Ejercicio 6.19. Array circular

Enunciado Desarrollar una clase que implemente una cola de enteros mediante un array circular.

Solución Tal y como se indicó en la Figura 6.2. de este capítulo, basta con emplear un par de índices, `elFrente` y `elFinal`, para indicar las posiciones del primer y último elemento de la cola. Con

esto se permite que tanto para añadir, como para quitar elementos, únicamente haya que desplazar uno de estos índices, sin necesidad de mover los índices del resto de elementos. La clase emplea un método privado, *siguiente*, que determina la posición siguiente a un índice determinado con una operación de resto con respecto a la longitud del array. Los métodos *colaVacía* y *colaLlena* permiten al programador usuario evitar errores al poner datos cuando el número es el máximo (la longitud del array), o quitar datos si no hay ninguno.

```
class Cola {
    private int [] c;
    private int    numero;
    private int    elFrente, elFinal;

    Cola (int maximo)
    {
        c=new int[maximo];
        numero=0;
        elFrente=0;
        elFinal=c.length-1;
    }
    public boolean colaVacía()
    {
        return numero==0;
    }
    public boolean colaLlena()
    {
        return numero==c.length;
    }
    public void Poner (int p)
    {
        if (numeroPersonas<c.length)
        {
            elFinal=siguiente(elFinal);
            c[elFinal]=p;
            numero++;
        }
    }
    public int Quitar()
    {
        int res=0;
        if (numero>0)
        {
            res=c[elFrente];
            elFrente=siguiente(elFrente);
            numero--;
        }
    }
}
```

```

        return    res;
    }
    private int siguiente(int n){
        return (n+1)%cp.length;
    }
}

public class TestCola {
    public static void main(String[] args){
        Cola t=new Cola(4);

        t.Poner(1);
        t.Poner(2);
        t.Poner(3);
        t.Poner(4);
        System.out.println(t.colaLlena());
        System.out.println(t.Quitar());
        System.out.println(t.Quitar());
        System.out.println(t.Quitar ());
        System.out.println(t.Quitar ());
    }
}

```

Ejercicio 6.20. Listas en memoria dinámica

Enunciado A partir de la definición genérica de un nodo, desarrollar una clase lista dinámica que permita la inserción y borrado de dichos nodos genéricos, tal y como se indicó en el Apartado 5.3.3.

Solución Supondremos la existencia de una clase nodo:

```

class nodo
{
    <nombre de clase> <nombre de atributo>;
    nodo siguiente;
};

```

y de una clase lista:

```

class lista
{
    nodo principio;

    .....
};

```


La función que inserta un nodo se indica a continuación. Se invoca para ello a la función `BuscarNodoInsertar`, que puede devolver la posición del nodo anterior a la posición donde se debe insertar, o el valor `null` si ese elemento ya se encontraba en la lista. Como en el caso particular de que la posición a insertar sea la primera el valor del elemento anterior también es `null`, se analiza el caso de devolución de valor `null` por parte de la función `BuscarNodoInsertar`, para ver si es que se debe insertar al principio de la lista o simplemente no se debe insertar el elemento. El resultado de la inserción o no del elemento se devuelve como valor booleano. Obsérvese que no está implementado el método con un objeto concreto por lo que la comparación entre los objetos de la lista queda abierta y dependerá del tipo de objetos que tengamos, lo que se ha indicado con la expresión "<>".

```
// La función retorna un valor false si el nodo
// ya se encontraba en la lista
// Si la operación tiene éxito se retorna un valor true

boolean InsertaNodo (nodo nodonuevo)
{
    nodo anterior;
    boolean insertado;

    if( principio == null)
    {
        principio = nodonuevo;
        nodonuevo.siguiente = null;
        insertado=true;
    }
    else
    {
        anterior = BuscarNodoInsertar( nodonuevo );
        if( anterior!=null )
        {
            // se devuelve una posicion especifica para insertar,
            // distinta del comienzo de la lista
            nodonuevo.siguiente = anterior.siguiente;
            anterior.siguiente = nodonuevo;
            insertado=true;
        }
        else if( nodonuevo.<nombre de atributo>. Comparar(
            principio. <nombre de atributo>) < 0)
        {
            // el valor devuelto de anterior fue null, porque la
            // posicion para insertar nodonuevo es el principio
            // de la lista
            nodonuevo.siguiente = principio;
            principio = nodonuevo;
        }
    }
}
```

```

        insertado=true;
    }
    else
    {
        // se ha devuelto un valor de anterior null porque no
        // hay que insertar el nodo (esta repetido)
        insertado=false;
    }
}
return insertado;
}

```

La función `BuscarNodoInsertar`, retorna la referencia al nodo anterior, o un valor **null** tanto en el caso de que la posición a insertar sea el principio, o que el nodo se encontró en la lista (hay una ambigüedad que debe ser tratada en los métodos clientes, como se indicó anteriormente). El algoritmo codificado se muestra a continuación, obsérvese que no está implementado con un objeto concreto por lo que la comparación entre los objetos de la lista queda abierta y dependerá del tipo de objetos que tengamos:

```

nodo BuscarNodoInsertar(nodo nodo_buscado)
{
    nodo aux;
    int comparacion;

    aux = principio;

    comparacion = aux.<nombre de atributo>.Comparar(
        nodo_buscado.<nombre de atributo>);

    if( comparacion == 0 )
    {
        // el nodo buscado coincide con el colocado al principio
        return null;
    }
    else if( comparacion < 0 && aux.siguiente == null )
    {
        // el nodo se insertaría justo a continuación del
        // principio, que es el último

        return principio;
    }
    else if( comparacion > 0 )
    {
        // el nodo se insertaría al comienzo de la lista
    }
}

```

```

        return null;
    }
    // en caso contrario, el nodo va despues del principio, y
    // la lista contiene mas elementos que hay que recorrer

    while( aux.siguiente != 0 )
    {
        // Comparamos con el nodo siguiente y si el nodo_buscado
        // es menor que el siguiente retornamos el valor de anterior.
        // Si es igual a 0 retornamos null.
        // Si es mayor pasamos al nodo siguiente

        comparacion = aux.siguiente.<nombre de atributo>.Comparar(
            nodo_buscado.<nombre de atributo>);

        if( comparacion == 0 )
        {
            return null;

        }
        else if( comparacion > 0 )
        {
            //el nodo a insertar ira despues de aux
            return aux;
        }

        // Pasamos a la siguiente posición

        aux = aux.siguiente;
    }

    //se llego al final de la lista: se devuelve el ultimo nodo: aux
    return aux;
}

```

Finalmente, para realizar el borrado de un elemento necesitamos buscar el elemento que se desea borrar (`nodo_borrar`) y una referencia al elemento anterior. En este caso basta con apuntar la referencia del nodo anterior al que se desea eliminar al siguiente del que se desea borrar. El algoritmo de borrado utiliza ahora una función de búsqueda de la posición anterior del elemento, `BuscarNodoBorrar`, que devolverá la posición anterior al elemento, si lo encuentra, y un valor `null` para los casos en los que habría que borrar el principio de la lista, o no encuentra el nodo. Obsérvese que tenemos un criterio contrario a la función `BuscarNodoInsertar`, puesto que ahora la condición para poder borrar es encontrar el nodo, mientras que antes el criterio era no encontrar; como se ha indicado anteriormente generará el valor `null` si el nodo a

borrar es el primero, o no está en la lista el nodo buscado. La función que permite el borrado en una lista simple para el ejemplo anterior es:

```
// La función retorna un valor false si el nodo no se
// encontraba en la lista. Si la operación tiene éxito
// se retorna un true

boolean BorrarNodo(nodo nodo_borrar )
{
    nodo anterior;
    boolean borrado;

    if( principio == null )
    {
        // si la lista esta vacia no tiene sentido borrar un nodo
        borrado= false;
    }
    else
    {
        anterior = BuscarNodoBorrar(nodo_borrar);
        if(anterior!=null)
        {
            // encontrado nodo anterior al que se va a borrar
            anterior.siguiente =anterior.siguiente.siguiente;
            borrado = true;
        }
        else if( nodo_borrar.<nombre de atributo>. Comparar(
            principio.<nombre de atributo>) == 0)
        {
            //hay que borrar el principio de la lista
            principio = principio.siguiente;
            borrado = true;
        }
        else
        {
            // no hay que borrar nada, no esta el nodo
            borrado = false;
        }
    }

    return borrado;
}
```


La función `BuscarNodoBorrar`, retorna la referencia al nodo anterior al buscado, o un valor `null` tanto en el caso de que la posición a borrar sea el principio, o que el nodo no se encontró en la lista (al contrario que la función `BuscarNodoInsertar`).

```
nodo BuscarNodoBorrar(nodo nodo_borrar)
{
    nodo aux;
    int comparacion;

    aux = principio;

    comparacion = aux.<nombre de atributo>.Comparar(
        nodo_borrar.<nombre de atributo>);

    if( comparacion == 0 )
    {
        // el nodo buscado coincide con el colocado al principio.
        // borrar el principio
        return null;
    }
    else if( comparacion < 0 && aux.siguiente == 0 )
    {
        // el nodo del principio es el ultimo y no es el buscado

        return null;
    }
    else if( comparacion > 0 )
    {
        // el nodo seria anterior al principio. No esta
        return null;
    }
    // en caso contrario, el nodo va despues del principio, y
    // la lista contiene mas elementos que hay que recorrer

    while( aux.siguiente != 0 )
    {
        // Comparamos con el nodo siguiente y si el nodo_buscado
        // es menor que el siguiente retornamos el valor de anterior.
        // Si es igual a 0 retornamos null.
        // Si es mayor pasamos al nodo siguiente

        comparacion = aux.siguiente.<nombre de atributo>.Comparar(
            nodo_borrar.<nombre de atributo>);

        if( comparacion == 0 )
```

```

        {
            return aux;
        }
    else if( comparacion > 0 )
    {
        //el nodo a buscar estaria despues de aux y no esta
        return null;
    }

    // Pasamos a la siguiente posición

    aux = aux.siguiente;

}

// se llego al final de la lista y no se encontro: se devuelve
// null
return null;
}

```

Como ya se indicó, cuando una instancia no tiene referencia se libera la memoria, por lo que cuando hablamos de borrar de la lista realmente hablamos de borrar de la memoria.

Ejercicio 6.21. Listas en memoria dinámica

Enunciado A partir de la lista desarrollada anteriormente, desarrollar una clase que gestione listas dinámicas de personas ordenadas por edad, utilizando las clases *Fecha* y *Persona* propuestas en ejercicios anteriores.

Solución Presentamos el código específico para una lista de objetos de la clase *Persona* que hemos denominado *ListaPersonasEnlazada*. Por completitud, en primer lugar presentamos el código entero de las clases *Fecha* y *Persona*, incluyendo los métodos de comparación que se utilizarán para manejar las listas.

Clase Fecha

```

class Fecha{
    private int dia,mes,anyo;
    Fecha () {
        // genera aleatoriamente una fecha
        anyo = (int) (Math.random()*25 + 1970);
        mes = (int) (Math.random()*12+1);

        if (mes == 1||mes==3||mes==5||mes==7||mes==8||mes==10||mes==12)

```

```
        dia=(int) (Math.random()*31+1);
    else if (mes==2)
        if ((anyo%4==0)&& (anyo%100!=0)|| (anyo%400==0))// es bisiesto
            dia=(int) (Math.random()*29+1);
        else dia=(int) (Math.random()*28+1);
    else // meses de 30
        dia=(int) (Math.random()*30+1);
    //System.out.println("Fecha generada aleatoriamente: ");
    //this.visualizar();
}

Fecha(Fecha f){
    dia=f.devDia();
    mes=f.devMes();
    anyo=f.devAnyo();
}

public int devDia(){
    return dia;
}

public int devMes(){
    return mes;
}

public int devAnyo(){
    return anyo;
}

public void copiar(Fecha f){
    dia=f.devDia();
    mes=f.devMes();
    anyo=f.devAnyo();
}

public int comparar (Fecha f){
    // devuelve -1 si esta fecha es anterior a f
    // devuelve 0 si las dos fechas son iguales
    // devuelve +1 si esta fecha es posterior a f
    // ponemos las fechas en cadenas del tipo aaaammdd
    // y las comparamos lexicograficamente
    String s1,s2;

    String d1,d2,m1,m2,a1,a2;
    if (dia<10) d1="0"+dia;
    else d1=String.valueOf(dia);
    if (mes<10) m1 = "0"+mes;
    else m1=String.valueOf(mes);
    a1=String.valueOf(anyo);
```

```

        if (f.devDia() < 10) d2 = "0" + f.devDia();
        else d2 = String.valueOf(f.devDia());
        if (f.devMes() < 10) m2 = "0" + f.devMes();
        else m2 = String.valueOf(f.devMes());

        a2 = String.valueOf(f.devAnyo());
        s1 = a1 + m1 + d1;
        s2 = a2 + m2 + d2;
        //System.out.println(s1 + " " + s2);
        return s1.compareTo(s2);

    }

    public void visualizar() {
        System.out.print(dia + "/" + mes + "/" + anyo);
    }
}

```

Clase Persona

```

class Persona {
    private String nombre;
    private Fecha fecNac;

    Persona () {
        // genera aleatoriamente uno de los nombres de la tabla
        String [] tabla = {"luis", "pepe", "alberto", "andres", "juan", "mariano", "eduardo",
            "roberto", "beatriz", "raquel", "marisa", "julia", "silvia", "sonia", "laura", "ana"};
        nombre = new String(tabla[(int) (Math.random() * 16)]);
        fecNac = new Fecha();
        System.out.println("Persona generada aleatoriamente");
        this.visualizar();
    }

    Persona (String n) {
        nombre = new String(n);
        fecNac = new Fecha();
        // fecha generada aleatoriamente
    }

    Persona (String n, int d, int m, int a) {
        nombre = new String (n);
        fecNac = new Fecha (d, m, a);
    }

    Persona (Persona p) {
        // crea un objeto persona con los valores del que se le pasa como parametro
        nombre = new String (p.devNombre());
    }
}

```



```
        fecNac = new Fecha (p.devFecha());
    }

    public String devNombre() {
        return nombre;
    }
    public Fecha devFecha(){
        return fecNac;
    }
    public void modifNombre(String n){
        nombre = n;
    }
    public void modifFecha(Fecha f){
        // copia los valores de f en fecNac
        fecNac.copiar(f);
    }

    public void copiar(Persona p){
        nombre=p.devNombre();
        fecNac.copiar(p.devFecha());
    }

    public int compararNombre(Persona p){
        // devuelve un entero < 0 si el nombre de esta
        // persona es menor lexicograficamente
        // que el nombre de p
        // devuelve 0 si son iguales y > 0 si es mayor
        return nombre.compareTo(p.devNombre());
    }

    public int compararEdad(Persona p){
        // devuelve un entero < 0 si esta persona es mas joven que p
        // 0 si han nacido el mismo dia
        // > 0 si esta persona es mayor que p
        // como al comparar fechas la fecha de nacimiento de la persona
        // mas joven es mayor
        // que la de la persona mayor, invertimos el signo del valor devuelto
        return - (fecNac.comparar(p.devFecha()));
    }
    public void visualizar(){
        System.out.println(nombre);
        fecNac.visualizar();
        System.out.println();
    }
}
```

Finalmente, la clase `ListaPersonaEnlazada` se corresponde exactamente con la clase `lista` del ejercicio anterior, sin más que particularizar el método de comparación al método `compararEdad`, definido en la clase `Persona`.

Clase `ListaPersonasEnlazada`

```
class nodo
{
    Persona persona;
    nodo siguiente;
    int comparar(nodo nodo_p)
    {
        return persona.compararEdad(nodo_p.persona);
    }
};

class ListaPersonasEnlazada
{
    nodo principio=null;
    public boolean InsertarPersona(Persona p)
    {
        nodo nuevo=new nodo();
        nuevo.persona=p;
        return InsertaNodo(nuevo);
    }

    public boolean BorrarPersona(Persona p)
    {
        nodo nuevo=new nodo();
        nuevo.persona=p;
        return BorrarNodo(nuevo);
    }

    public boolean vacia()
    {
        return principio==null;
    }

    public void visualizar()
    {
        nodo auxiliar=principio;
        while (auxiliar!=null)
```

```
{
    auxiliar.persona.visualizar();
    auxiliar=auxiliar.siguiente;
}

}

boolean InsertaNodo (nodo nodonuevo)
{
    nodo anterior;
    boolean insertado;

    if( principio == null)
    {
        principio = nodonuevo;
        nodonuevo.siguiente = null;
        insertado=true;
    }
    else
    {
        anterior = BuscarNodoInsertar( nodonuevo );

        if( anterior!=null )
        {
            // se devuelve una posicion especifica para insertar,
            // distinta del comienzo de la lista
            nodonuevo.siguiente = anterior.siguiente;
            anterior.siguiente = nodonuevo;
            insertado=true;
        }
        else if( nodonuevo.comparar(principio) < 0)
        {
            // el valor devuelto de anterior fue null, porque la
            // posicion para insertar nodonuevo es el principio
            // de la lista

            nodonuevo.siguiente = principio;
            principio = nodonuevo;
            insertado=true;
        }
        else
        {
            // se ha devuelto un valor de anterior null porque no
            // hay que insertar el nodo (esta repetido)
            insertado=false;
        }
    }
}
```

```

        }
    }
    return insertado;
}

nodo BuscarNodoInsertar(nodo nodo_buscado)
{
    nodo aux;
    int comparacion;

    aux = principio;

    comparacion = aux.comparar(nodo_buscado);

    if( comparacion == 0 )
    {
        // el nodo buscado coincide con el colocado al principio
        return null;
    }
    else if( comparacion < 0 && aux.siguiente == null )
    {
        // el nodo se insertaria justo a continuacion del
        // principio, que es el ultimo

        return principio;
    }
    else if( comparacion > 0 )
    {
        // el nodo se insertaria al comienzo de la lista
        return null;
    }
    // en caso contrario, el nodo va despues del principio, y
    // la lista contiene mas elementos que hay que recorrer

    while( aux.siguiente != null )
    {
        // Comparamos con el nodo siguiente y si el nodo_buscado
        // es menor que el siguiente retornamos el valor de anterior.
        // Si es igual a 0 retornamos null.
        // Si es mayor pasamos al nodo siguiente

        comparacion = aux.siguiente.comparar(nodo_buscado);
        if( comparacion == 0 )

```



```
{
    return null;
}
else if( comparacion > 0 )
{
    //el nodo a insertar ira despues de aux
    return aux;
}

// Pasamos a la siguiente posición

    aux = aux.siguiente;

}

//se llego al final de la lista: se devuelve el ultimo nodo: aux
return aux;
}

boolean BorrarNodo(nodo nodo_borrar )
// La función retorna un valor false si el nodo no se
// encontraba en la lista. Si la operación tiene éxito
// se retorna un true

{
    nodo anterior;
    boolean borrado;

    if( principio == null )
    {
        // si la lista esta vacia no tiene sentido borrar un nodo
        borrado= false;
    }
    else
    {
        anterior = BuscarNodoBorrar(nodo_borrar);
        if(anterior!=null)
        {
            // encontrado nodo anterior al que se va a borrar
            anterior.siguiente =anterior.siguiente.siguiente;
            borrado = true;
        }
    }
}
```

```

        else if( nodo_borrar.comparar(principio) == 0)
        {
            //hay que borrar el principio de la lista
            principio = principio.siguiente;
            borrado = true;
        }
        else
        {
            // no hay que borrar nada, no esta el nodo
            borrado = false;
        }
    }

    return borrado;
}

nodo BuscarNodoBorrar(nodo nodo_borrar)
//La función "BuscarNodoBorrar", retorna la referencia al nodo
//anterior al buscado, o un valor null tanto para el caso de
//que la posición a borrar es el principio o bien el nodo no se
//encontró en la lista.

{
    nodo aux;
    int comparacion;

    aux = principio;

    comparacion = aux.comparar(nodo_borrar);

    if( comparacion == 0 )
    {
        // el nodo buscado coincide con el colocado al principio.
        // borrar el principio
        return null;
    }
    else if( comparacion < 0 && aux.siguiente == null)
    {
        // el nodo del principio es el ultimo y no es el buscado

        return null;
    }
    else if( comparacion > 0 )
    {
        // el nodo seria anterior al principio. No esta
        return null;
    }
}

```

```
}  
// en caso contrario, el nodo va despues del principio, y  
// la lista contiene mas elementos que hay que recorrer  
  
while( aux.siguiente != null )  
{  
    // Comparamos con el nodo siguiente y si el nodo_buscado  
    // es menor que el siguiente retornamos el valor de anterior.  
    // Si es igual a 0 retornamos null.  
    // Si es mayor pasamos al nodo siguiente  
  
    comparacion = aux.siguiente.comparar(nodo_borrar);  
  
    if( comparacion == 0 )  
    {  
        return aux;  
    }  
    else if( comparacion > 0 )  
    {  
        //el nodo a buscar estaria despues de aux y no esta  
        return null;  
    }  
  
    // Pasamos a la siguiente posición  
  
    aux = aux.siguiente;  
}  
  
// se llego al final de la lista y no se encontro: se devuelve  
// null  
return null;  
}  
}
```

6.7.4. Ejercicios de herencia

Ejercicio 6.22. Clases derivadas

Enunciado Derivar una nueva clase CuadradoCambiante a partir de Cuadrado, que añada nuevas funcionalidades, como por ejemplo la capacidad de desplazarse y de cambiar uniformemente su tamaño.

Solución En primer lugar, recordando la parte básica de la herencia, si definimos la clase CuadradoCambiante de la forma:

```
class CuadradoCambiante extends Cuadrado
{
}

```

estamos haciendo que `CuadradoCambiante` derive de la clase `Cuadrado`. Con esta definición tendríamos que la definición de la clase `CuadradoCambiante` estaría vacía, aunque poseería automáticamente las funciones miembro y los atributos de `Cuadrado`. La clase `CuadradoCambiante` se puede utilizar de la misma manera que `Cuadrado`.

```
import CuadradoCambiante;
class UsoCuadradoCambiante{
public static void main (String args []){

    CuadradoCambiante miCuadrado=new CuadradoCambiante();
    System.out.println("El perimetro es: "+
        miCuadrado.CalcularPerimetro());

    }
}

```

Para añadir nuevas funcionalidades a la clase derivada no se tiene más que incluir en la definición aquello que queramos ampliar. Por ejemplo:

```
class CuadradoCambiante extends Cuadrado
{
private float velocidadx, velocidady;
// almacena la velocidad con la que
// va a mover el cuadrado

private float escala; // Factor de crecimiento del Cuadrado

public void Mover(float tiempo)
{
    ModificarValores(VerValor_x1()+velocidadx*tiempo,
                    VerValor_y1()+velocidady*tiempo,
                    VerValor_x2()+velocidadx*tiempo,
                    VerValor_y2()+velocidady*tiempo,
                    VerColor() );
}

public void Escalar(void)
{
    float centro_x=0.5*(VerValor_x1()+VerValor_x2());
    float centro_y=0.5*(VerValor_y1()+VerValor_y2());
    ModificarValores(

```



```

        centro_x+escala*(VerValor_x1()-centro_x),
        centro_y+escala*(VerValor_y1()-centro_y),
        centro_x+escala*(VerValor_x2()-centro_x),
        centro_y+escala*(VerValor_y2()-centro_y),
        VerColor() );
    }

void PonEscala( float factor ){ escala = factor; }
void PonVelocidadx( float vel){ velocidadx = vel; }
void PonVelocidady( float vel){ velocidady = vel; }
}

```

Junto a los miembros heredados, la clase `CuadradoCambiante` tiene:

- Tres nuevos miembros privados: dos para la velocidad y uno para la escala (que indican la velocidad a la que va a ir moviéndose el cuadrado y un factor que indica una modificación del tamaño del cuadrado).
- Cinco nuevas funciones miembro de carácter público:
 - `PonVelocidadx(int vel)` y `PonVelocidady(int vel)`, que asignan un valor a la velocidad en cada una de las coordenadas cartesianas.
 - `PonEscala(int factor)`, la cual asigna un valor a escala.
 - `Mover(float tiempo)`, que hace que el cuadrado vaya cambiando la posición en horizontal según la velocidad.
 - `Escalar()`, que aplica la escala sobre el cuadrado actual modificando los valores de los puntos.

Es importante observar que los métodos `Mover` y `Escalar` de `CuadradoCambiante` necesitan acceder a los atributos de la clase superior, `Cuadrado`, pero al ser privados, deben hacerlo a través de métodos públicos. Esto redundará en una complicación a la hora de programar la clase derivada, pero facilita enormemente el depurado de la aplicación, ya que únicamente a través de los métodos de la clase base se pueden modificar sus valores, garantizando el encapsulamiento de esta clase al ser la manipulación de su estado interno responsabilidad exclusivamente suya. Un ejemplo claro de esta complicación es el método `Escalar` donde es necesario utilizar los métodos `ModificarValores`, `Ver_Valorx1`, `Ver_Valorx2`, `Ver_Valory1`, `Ver_Valory2`, para poder realizar una operación tan sencilla como escalar el cuadrado:

Ejercicio 6.23. Clases derivadas

Enunciado Redefina ahora los atributos de `Cuadrado` con el calificador `protected` y rehaga los métodos anteriores, observando la facilidad en su escritura.

Solución

```

class Cuadrado
{

```

```
protected double    x1, y1, x2, y2;
protected int color;
}
```

Al igual que un miembro privado, no puede accederse a un miembro protegido (`protected`) desde fuera de la clase. Por eso es ilegal el siguiente código:

```
//Ejemplo de programa principal
{
    Cuadrado miCuadrado;

    miCuadrado.color = 5; // ERROR: no se puede acceder a un
                          // miembro privado
}
```

Sin embargo, a diferencia de los miembros privados, puede accederse a un miembro protegido desde dentro de una clase que deriva de la clase en la cual se define el miembro. Con esto, podemos hacer que la clase `CuadradoCambiante` acceda directamente a los datos miembro definidos en `Cuadrado`:

```
class CuadradoCambiante extends Cuadrado
{
    .....

    public void Escalar(void)
    {
        float centro_x=0.5*(x1+x2);
        float centro_y=0.5*(y1+y2);
        x1 = centro_x+escala*( x1-centro_x);
        y1 = centro_y+escala*( y1-centro_y);
        x2 = centro_x+escala*( x2-centro_x);
        x3 = centro_y+escala*( y2-centro_y);
    }

    public void Mover(float tiempo)
    {
        ModificarValores(x1+velocidadx*tiempo,
                        y1+velocidady*tiempo,
                        x2+velocidadx*tiempo,
                        y2+velocidady*tiempo,
                        color );
    }
}
```

```

public void Escalar(void)
{
    float centro_x=0.5*(x1+x2);
    float centro_y=0.5*(y1+y2);
    x1 = centro_x+escala*( x1-centro_x);
    y1 = centro_y+escala*( y1-centro_y);
    x2 = centro_x+escala*( x2-centro_x);
    x3 = centro_y+escala*( y2-centro_y);
}
...
}

```

Ejercicio 6.24. Constructor en clases derivadas

Enunciado Definir el constructor de la clase `CuadradoCambiante` utilizando (a) los métodos definidos en la clase base, (b) accediendo al constructor de la clase base.

Solución Para construir el objeto de la clase derivada tenemos que dar valores a todos sus atributos. Si empleamos los métodos disponibles en la clase base y los nuevos, en el constructor, debemos llamar a cuatro métodos: `ModificarValores` (que pertenece a la clase base), `PonVelocidadx`, `PonVelocidady`, y `PonEscala` (que pertenecen únicamente a la clase derivada). De esta manera el constructor de `CuadradoCambiante` pondría los valores de todos los datos miembro al crearse una instancia de la clase:

```

public CuadradoCambiante(float x1_p, float y1_p,
    float x2_p, float y2_p, int color_p, float velx_p,
    float vely_p, float escala_p)
{
    ModificarValores(x1_p, y1_p, x2_p, y2_p, color_p);
    PonVelocidadx(velx_p);
    PonVelocidady(vely_p);
    PonEscala(escala_p);
}

```

Sin embargo, podemos acceder directamente al constructor base desde la clase derivada, quedando el código mucho más claro:

```

public CuadradoCambiante(float x1_p, float y1_p,
    float x2_p, float y2_p, int color_p,
    float velx_p, float vely_p, float escala_p){
    super(x1_p, y1_p, x2_p, y2_p, color_p);
    PonVelocidadx(velx_p);
    PonVelocidady(vely_p);
}

```

```
PonEscala(escala_p);
}
```

Como podemos ver, el constructor tiene una serie de parámetros que se utilizan para invocar al constructor de la clase base. Para ello, en el cuerpo del constructor de la clase derivada se llama al constructor de la clase base a través de la palabra reservada **super**. Además, el cuerpo del constructor llama a las funciones propias de la clase derivada para fijar el valor del dato miembro propio de la clase. Es importante notar que si no se especifica, el compilador añade automáticamente una llamada al constructor por defecto de la clase base. De esta manera, el siguiente constructor:

```
public CuadradoCambiante()
{
    velocidadx=0;
    velocidady=0;
    escala=0;
}
```

es automáticamente "traducido" a:

```
public CuadradoCambiante()
{
    super();
    velocidadx=0;
    velocidady=0;
    escala=0;
}
```

que como vimos antes, inicializa a cero todos los atributos de la clase superior Cuadrado. Sin embargo, esto sólo es válido si no existe ningún constructor para esa clase base, en cuyo caso se llamaría al creado por defecto para ella, o si ya existe un constructor sin parámetros, como ocurre en este caso, que inicializa todos los valores a cero. Si todos los constructores de la clase base necesitan parámetros, entonces es obligatorio invocar alguno de ellos en todos los constructores de clases derivadas en su primera instrucción dando, el compilador un error en caso contrario.

Ejercicio 6.25. Construcción de atributos en clases derivadas

Enunciado Supongamos ahora que la clase con el cuadrado cambiante incluye un objeto de una clase denominada Velocidad, en lugar de las dos variables de tipo float. Defina esta clase y especifique con dos alternativas posibles la construcción del objeto que representa la velocidad del cuadrado. Indique además el orden en el que se construyen los distintos atributos del objeto en cada caso.

Solución

En primer lugar, la clase `Velocidad` estará compuesta por dos variables de tipo `float`. Para mantenerla encapsulada, se definen como atributos privados, y métodos públicos para manipularlos:

```
class Velocidad
{
    private float Velx;
    private float Vely;

    Velocidad(){
        Velx = 0;
        Vely = 0;
    }

    Velocidad(Velocidad aux){
        this.Velx = aux.Velx;
        this.Vely = aux.Vely;
    }

    void ponVelx(float vel){ Velx= vel; }
    void ponVely(float vel){ Vely= vel; }
    float daVelx(){ return Velx; }
    float daVely(){ return Vely; }
}
```

En una primera posibilidad, la clase `CuadradoCambiante` puede construir su objeto `velocidad` en el momento de la llamada al constructor:

```
class CuadradoCambiante extends Cuadrado
{
    private Velocidad vel;
    private float escala;

    CuadradoCambiante()
    {
        super();
        escala = 0;
        vel = new Velocidad();
    }

    public void Mover(float tiempo)
    {
        .....
    }
}
```

```

public void Escalar(void)
{
    .....
}

void PonEscala( float factor ){ escala = factor; }
void PonVelocidadx( float velaux){ vel.ponVelx( velaux); }
void PonVelocidadx( float velaux){ vel.ponVely( velaux); }

}

```

Obsérvese que en ese caso, el atributo `vel` es únicamente una referencia hasta que se ha ejecutado el código del constructor de la clase `CuadradoCambiante` (que se ejecuta después de haberse invocado al constructor de la clase base, `Cuadrado`). Se crea el objeto atributo cuando se invoca a su constructor mediante el operador **new**

Otra posibilidad es invocar al constructor en la propia declaración del atributo:

```

class CuadradoCambiante extends Cuadrado
{
    private Velocidad vel = new Velocidad();
    private float escala;

    CuadradoCambiante()
    {
        super();
        escala = 0;
    }
}

```

En este caso se llamaría al constructor de `Velocidad` tras haber realizado la llamada al constructor de la clase base, de modo que antes de construirse el objeto de la clase `CuadradoCambiante`, pero después de haberse construido ya la parte correspondiente a la clase `Cuadrado`, se crea la instancia del atributo `vel`.

Ejercicio 6.26. Sobreescritura en clases derivadas

Enunciado Sobreescriba el método `VerValores` en la clase `CuadradoCambiante` para que pueda ser útil en dicha clase.

Solución En nuestro `CuadradoCambiante` el método `VerValores` de la clase base ya no nos es útil puesto que no permite visualizar los valores de los nuevos atributos. Únicamente presenta los atributos correspondientes a la clase `Cuadrado`. En vez de hacer una función nueva para visualizar los datos de los nuevos atributos, se puede redefinir `VerValores` para que incluyan los nuevos parámetros:

```

public void VerValores()
{
    System.out.print("Atributos: x1="+VerValor_x1()+"
y1="+VerValor_y1()+" x2="+VerValor_x2()+" y2="+VerValor_y2()+
" color="+VerColor()+" velocidadx="+velocidadx+"velocidady="+
velocidady+" escala="+escala);

    System.out.print(" velocidadx="+velocidadx+"velocidady="+velocidady+
" escala="+escala);
}

```

En este caso, la clase `CuadradoCambiante` dispone de dos versiones de la misma función, una heredada y otra explícitamente definida. Cuando una instancia de la clase `CuadradoCambiante` llame a la función `VerValores` se llamará a la versión redefinida que está explícitamente definida para esa clase.

Ahora bien, el hecho de redefinir una función no implica que no se pueda utilizar la de la clase base. Por ejemplo, en nuestra nueva función podemos llamar en primer lugar a la función `VerValores` de la clase `Cuadrado`, que está oculta por la redefinición que hemos hecho. Para poder llamarla se utiliza la sintaxis `super.Funcion`. De este modo se llamará a la función de la clase base que utilizará los atributos comunes a las dos clases. Así tendríamos:

```

public void VerValores()
{
    super.VerValores();
    System.out.print("velocidadx="+velocidadx+
" velocidady="+velocidady+" escala="+escala);
}

```

La expresión `super.metodo()` obliga al compilador a llamar a esa versión de `VerValores`. Si no se pone el especificador de campo el compilador llamaría a la versión de `VerValores` contenida en `CuadradoCambiante`. (De nuevo la versión de la función definida sobre la clase actual tiene prioridad frente a la función heredada.)

Ejercicio 6.27. Clases abstractas

Enunciado Se pide en este ejercicio definir una clase abstracta `Figura` que sirva como patrón para cualquier tipo de figura plana, con métodos para indicar el tipo de figura y para calcular el área y perímetro.

Solución Este ejemplo ilustra cómo utilizar la herencia no sólo para la reutilización de código, sino como patrón que especifica las funcionalidades a cubrir por las clases derivadas. Así, podemos pensar en una clase que se encuentre a un nivel de abstracción mayor que un cuadrado como es una figura genérica en el plano. Evidentemente, una figura genérica no tendrá ningún significado específico ni ningún método que sea realmente suyo, puesto que no se puede hacer nada sin saber qué

tipo de figura es (y sin la información específica de sus detalles de implementación), pero sí que puede definir cuál debe ser la funcionalidad mínima de cualquier figura. La definición de esta clase es especial ya que se define como una clase abstracta, mediante la palabra reservada **abstract**, que indica que no pueden crearse objetos de esa clase:

```
abstract class Figura
{
    private boolean poligono;
    Figura(){
        // no tendria sentido, pero lo hacemos para debug
        System.out.println("construyendo figura");
    }
    abstract double calcularArea();
    abstract double calcularPerimetro();
    abstract void queSoy();
}
```

En este ejemplo se está imponiendo que cualquier clase que derive de `Figura`, si se utiliza para crear objetos, tendrá al menos implementados los métodos `calcularArea`, `calcularPerimetro` y `queSoy`. Sin embargo, podría introducirse en la clase abstracta la implementación de alguno de sus métodos, que podrán ser utilizados posteriormente por las clases derivadas (utilizando así las dos ventajas de la herencia: reutilizar código y servir como patrón):

```
abstract class Figura
{
    private boolean poligono;
    Figura(){
        // no tendria sentido, pero lo hacemos para debug
        System.out.println("construyendo figura");
    }
    public double calcularArea(){
        // no tendria sentido, pero lo hacemos para debug
        System.out.println("calcularArea de figura");
        return 0;
    }
    abstract double calcularPerimetro();
    // esto es mas apropiado

    public void queSoy(){
        System.out.println("Soy una figura");
    }
}
```

Sin embargo, con la definición de este segundo caso ya no se estaría imponiendo a las clases derivadas implementar obligatoriamente los métodos `calcularArea` y `queSoy`, pudiéndose optar por reutilizar los de la clase base o sobrescribirlos.

Ejercicio 6.28. Clases derivadas

Enunciado Se pide ahora desarrollar una clase `Rectangulo` que implemente el patrón definido por la clase `Figura`. Por simplicidad, se tratará de rectángulos con dos de sus lados apoyados en los ejes de coordenadas (es decir, uno de sus vértices es siempre el origen). A continuación, se desarrollará una clase `Cuadrado` que herede de la anterior para el caso particular del rectángulo cuadrado.

Solución A partir de la clase `Figura` se deriva la clase `Rectangulo`, que añade como atributos específicos las coordenadas del punto que define un vértice (el otro es el origen de coordenadas). Se implementan los métodos (o se sobreescriben, en su caso) utilizando esta información:

```
class Rectangulo extends Figura
{
    // supondremos un rectángulo muy sencillo con la base horizontal
    // y el vertice inferior izq en el origen de coordenadas 0,0
    private double x,y;
    // el otro vertice
    Rectangulo () {
        System.out.println("Construyendo Rectangulo() por defecto ");
        x=0;
        y=0;
    }
    Rectangulo(double x, double y){
        System.out.println("Construyendo Rectangulo(int,int)");
        this.x=x;
        this.y=y;
    }

    public void modificar(double x, double y){
        System.out.println("Modificando Rectangulo(int,int)");
        this.x=x;
        this.y=y;
    }
    public double devX(){
        return x;
    }
    public double devY(){
        return y;
    }
    public double calcularArea(){
        System.out.println("calcularArea de Rectangulo");
        return x*y;
    }
}
```

```

    }

    public double calcularPerimetro(){
        System.out.println("calcularPerimetro de Rectangulo");
        return 2*(x+y);
    }

    public void queSoy(){
        System.out.println("Soy un Rectangulo");
    }
}

```

A partir del rectángulo es posible definir un cuadrado que en este caso es una simplificación del rectángulo ya que un cuadrado sólo necesita un valor para definir las dos coordenadas del vértice, que coinciden. La clase Cuadrado quedaría:

```

class Cuadrado extends Rectangulo
{
    // Un cuadrado ES-UN rectangulo

    private double lado;
    Cuadrado () {
        System.out.println("Construyendo Cuadrado");
    }
    Cuadrado(double lado){
        super(lado, lado);
        System.out.println("Construyendo Cuadrado(double lado)");
        this.lado=lado;
    }

    public void modificar(double l){
        System.out.println("Modificando Cuadrado(double)");
        this.lado=lado;
        super.modificar(lado,lado);
    }

    public void modificar(double x, double y){
        System.out.println("Modificando Cuadrado(double, double)");
        this.lado=x;
        super.modificar(x, x);
    }

    public double calcularArea(){
        System.out.println("calcularArea de Cuadrado");
    }
}

```

```

        return lado*lado;
    }

    public double calcularPerimetro(){
        System.out.println("calcularPerimetro de Rectangulo");
        return 4*lado;
    }

    public void queSoy(){
        System.out.println("Soy un Cuadrado");
    }
}

```

Obsérvese que en este caso estamos utilizando la herencia para restringir la utilización de la clase base. El hecho de encapsular el rectángulo (los atributos son `private`) permite asegurar que el cuadrado será siempre un cuadrado y que en ningún momento sea posible que los atributos de la clase base entren en conflicto con los de la derivada. Para ello, los constructores invocan a los de la clase base para imponerle que sus atributos sean coherentes con los de la derivada, y además se ha sobrescrito el método de modificación de atributos de la clase base para impedir que sea posible modificar los valores del cuadrado de manera que deje de serlo (es decir, es imposible a través de la interfaz del cuadrado modificar la x o la y de forma separada). Esta capacidad de encapsulación junto con la herencia es la característica más importante vista hasta ahora de la POO.

6.7.5. Ejercicios de polimorfismo

Ejercicio 6.29. Polimorfismo

Enunciado Verificar la posibilidad de que una misma variable referencia invoque a métodos distintos, según se asigne a una instancia de un objeto `Cuadrado` o a la de un objeto de la clase derivada, `CuadradoCambiante`. Hacer un ejemplo en el que sea imposible predecir en tiempo de compilación qué método será el invocado.

Solución A continuación, se muestra un ejemplo en el programa principal de una clase ejecutable:

```

import CuadradoCambiante;
class UsoCuadrados{
    public static void main (String args[])
    {
        Cuadrado miCuadrado;
        if (Math.random()>0.5)
            miCuadrado=new Cuadrado(1.0, 1.0, 0.0, 0.0, 0);
        else
            miCuadrado=new CuadradoCambiante(1.0, 1.0, 0.0, 0.0, 0, 2.0, 2.0,
                                                3.0);
        miCuadrado.verValores();
    }
}

```

Como puede verse, la última instrucción se asignará de forma dinámica a la clase adecuada en función del tipo final con el que se haya creado el objeto `miCuadrado`. Como esto depende de un valor aleatorio, mediante la llamada al método `Math.random`, no se determinará el método invocado hasta el momento de la ejecución.

Ejercicio 6.30. Construcción de un array polimórfico

Enunciado Utilizando las tres clases definidas en el Ejercicio 6.25., definir un array de referencias a figuras con diferentes objetos apuntados, de modo que la llamada a los métodos del patrón se materialice en diferentes llamadas según cada caso. Conviene trazar la salida del programa.

Solución En este ejemplo se van creando alternativamente un objeto cuadrado y un rectángulo que se asignan en posiciones consecutivas. Recordando que un cuadrado es un rectángulo que a su vez es una figura, una referencia a una figura puede referenciar tanto a un cuadrado como a un rectángulo:

```
class PruebaCuadrado
{
    public static void main(String[] args)
    {
        Figura [] f=new Figura[4];
        for(int i=0;i<4;i++)
            if (i%2==0)
                f[i]=new Cuadrado(2*i);
            else f[i]=new Rectangulo(i,2*i);
        for (int i=0;i<4;i++){
            f[i].queSoy();
            System.out.println("Area: "+f[i].calcularArea());
        }
    }
}
```

Lo primero que se puede observar es la capacidad de la POO para reutilizar una vez más el código ya que, si es necesario definir más tipos de figuras, basta con heredar de la clase `Figura` y podrán ser almacenadas también en ese mismo array. Lo segundo que se observa es que para gestionar el array sólo es necesario acceder a los métodos de la clase `Figura` (que existen en todas las clases que deriven de `Figura`). De esta manera el código que gestiona el array (el último `for` donde se imprime el área de cada `Figura`) es independiente del tipo real de `Figura` que se tenga en el array. Cualquier modificación de código en las clases `Cuadrado` y `Rectangulo`, cualquier nueva clase que derive de `Figura` y cuyos objetos hayan sido incluidos en el array, podrá ser gestionada por esta parte del programa. La salida por pantalla de ese programa es:

```
Salida
construyendo figura
Construyendo Rectangulo() por defecto
```



```

Construyendo Cuadrado(double lado)
Modificando Rectangulo(int,int)
construyendo figura
Construyendo Rectangulo(int,int)
construyendo figura
Construyendo Rectangulo() por defecto
Construyendo Cuadrado(double lado)
Modificando Rectangulo(int,int)
construyendo figura
Construyendo Rectangulo(int,int)
Soy un Cuadrado
calcularArea de Cuadrado
Area: 0.0
Soy un Rectangulo
calcularArea de Rectangulo
Area: 2.0
Soy un Cuadrado
calcularArea de Cuadrado
Area: 16.0
Soy un Rectangulo
calcularArea de Rectangulo
Area: 18.0

```

Ejercicio 6.31. Polimorfismo y agregación

Enunciado Se propone un ejemplo muy sencillo que resume de una manera didáctica la idea de agregación (definir un objeto como atributo de una clase), de herencia y de polimorfismo. Para ello se van a definir 4 clases denominadas Cero, Uno, Dos y Tres. La clase Cero únicamente tendrá un atributo booleano. La clase Uno contendrá un objeto de la clase Cero y un atributo entero. La clase Dos derivará de la clase Uno añadiendo una cadena. Por último, la clase Tres deriva de la Dos y añade un float. Se realizará un programa principal en el que se trazará la secuencia de llamadas a los constructores, la posibilidad de utilizar polimorfismo y cómo se gestionan las llamadas a los métodos.

Solución Lo primero que hacemos es definir una clase denominada Cero que va a almacenar si un número es positivo:

```

class Cero {
    private boolean positivo;
    Cero(int n){
        positivo=n>0;
        System.out.println("** Constructor de Cero **");
    }
    public boolean equals(Cero o){
        return positivo==o.positivo;
    }
}

```

Esta clase Cero forma parte de la clase Uno que contiene un número entero y su signo (mediante un objeto de la clase Cero). El objeto de la clase Cero es construido en el constructor de la clase Uno mediante la llamada al operador new. A su vez se han desarrollado dos métodos, uno de comparación (que hace uso del método de comparación de la clase Cero) y otro de incremento:

```
class Uno {
    private int dato;
    Cero cer;

    Uno(int a) {
        System.out.println("* Constructor de Uno *");
        dato=a;
        cer=new Cero(dato);
    }
    public void imprime() {
        System.out.println("\tEntero "+dato);
    }
    public void incrementa() {
        dato++;
    }
    public boolean equals(Uno o){
        return o.dato==dato && o.cer.equals(cer);
    }
}
```

La clase Dos deriva de la clase Uno y añade una cadena. Tiene dos constructores, uno que obvia el dato numérico de la clase Uno y, por tanto, lo construye con valor 0 (obsérvese la necesidad de llamar a super (0) para que no dé error) y otro que recibe un entero y llama al constructor de la clase base. Además implementa nuevos métodos y sobrescribe el método imprimir. La sobrescritura del método imprimir (que hace uso del imprimir de la clase anterior para imprimir la parte correspondiente a la clase Uno) se hace a propósito para buscar el polimorfismo en ejecución. La clase Dos es:

```
class Dos extends Uno {
    private String cad;

    Dos(String c){
        super(0);
        System.out.println("* Constructor de Dos *");
    }
    Dos(String c, int d){
        super(d);
        System.out.println("* Constructor de Dos *");
        cad=c;
    }
}
```

```

    }
    public void imprime() {
        System.out.println("\tCadena "+ cad);
        super.imprime();
    }
    public void tamanoCadena() {
        System.out.println("\tLongitud de la cadena "+cad.length());
    }
    public boolean equals(Dos o){
        return cad.equals(o.cad) && super.equals(o);
    }
}

```

La clase Tres deriva de la Dos y añade un atributo float. La clase Tres es similar a la Dos y únicamente resaltar nuevamente el método Imprimir que es nuevo y no sobrescribe el método Imprimir de Dos y Uno aunque hace uso del método de la clase base:

```

class Tres extends Dos {
    private float num;
    Tres(float n, String c, int d){
        super(c,d);
        System.out.println("* Constructor de Tres*");
        num=n;
    }
    public void imprime(String cad){
        System.out.println(cad);
        System.out.println("\tFlotante "+num);
        // usamos el método de la clase base para
        // imprimir los datos que faltan.
        imprime();
    }
    public boolean equals(Tres o){
        return num==o.num && super.equals(o);
    }
}

```

Por último, estas cuatro clases son utilizadas en el siguiente problema didáctico donde se observa el polimorfismo entre Uno y Dos con el método imprimir pero no con la clase Tres que no ha sobrescrito dicho método y, por tanto, únicamente lo ha heredado:

```

public class Herencia {
    public static void main(String[] args ){
        Cero ce1=new Cero(1);
        Cero ce2=new Cero(1);
        Uno a=new Uno(8);
    }
}

```

```

    Uno a2=new Uno(8);
    Dos b=new Dos("A",34);
    Dos b2=new Dos("A",34);
    Tres c=new Tres(8.9f,"TRES",8);
    Tres c2=new Tres(8.9f,"TRES",8);

    System.out.println("Comparaciones");
    // con el equals de la clase Object
    System.out.println(cel.equals(cel));
    System.out.println(cel.equals(ce2));
    System.out.println(a.equals(a2));
    System.out.println(b.equals(b2));
    System.out.println(c.equals(c2));
    System.out.println("\nObjeto a");
    a.imprime();
    a.incrementa();
    System.out.println("Objeto b");
    b.imprime();
    b.incrementa();
    System.out.println("Objeto c");
    c.imprime("Sobrecarga");
    System.out.println("Llamada al método de la clase base");
    c.imprime();
    c.incrementa();
    // Polimorfismo
    System.out.println("Objeto a");
    llamada(a);
    System.out.println("Objeto b");
    llamada(b);
    System.out.println("Objeto c");
    llamada(c);
}

public static void llamada(Uno d){
    System.out.println ("- Método llamada -");
    d.imprime();
    if (d instanceof Dos)
        ((Dos)d).tamanoCadena();
}
}

```

La salida del programa es:

```

Salida
* Constructor de Cero *
* Constructor de Cero *
* Constructor de Uno *
* Constructor de Cero *
* Constructor de Uno *

```



```
* Constructor de Cero *
* Constructor de Uno *
* Constructor de Cero *
* Constructor de Dos *
* Constructor de Uno *
* Constructor de Cero *
* Constructor de Dos *
* Constructor de Uno *
* Constructor de Cero *
* Constructor de Dos *
* Constructor de Tres*
* Constructor de Uno *
* Constructor de Cero *
* Constructor de Dos *
* Constructor de Tres*
Comparaciones
true
true
true
true
true

Objeto a
    Entero 8
Objeto b
    Cadena A
    Entero 34
Objeto c
Sobrecarga
    Flotante 8.9
    Cadena TRES
    Entero 8
Llamada al método de la clase base
    Cadena TRES
    Entero 8
Objeto a
- Método llamada -
    Entero 9
Objeto b
- Método llamada -
    Cadena A
    Entero 35
    Longitud de la cadena 1
Objeto c
- Método llamada -
    Cadena TRES
    Entero 9
    Longitud de la cadena 4
```


Diseño de aplicaciones en Java

Hasta ahora se ha trabajado con clases y objetos definidos específicamente para encapsular un conjunto de variables y operaciones que están relacionadas y que dan lugar a un único concepto, y posteriormente hemos visto las ventajas de la herencia y del polimorfismo. En este capítulo presentaremos el desarrollo de programas que contengan estructuras de clases interrelacionadas con el objeto de abordar problemas de mayor generalidad. Se pretende aprovechar los mecanismos de herencia y polimorfismo para programar, de manera que el código sea encapsulado, ocultando los datos irrelevantes a cada nivel de abstracción, y dicho código sea al mismo tiempo potente, flexible y fácilmente reutilizable.

7.1. Relaciones entre clases

Los objetos, como abstracción del mundo real, representando un concepto o idea, se relacionan con otros objetos para dar lugar a conceptos más elaborados. Las relaciones constituyen los enlaces que permiten a un objeto relacionarse con todos aquellos que forman parte de la misma organización. Las relaciones pueden ser divididas a grandes rasgos en dos tipos, jerárquicas o semánticas. Las relaciones que se utilizan con mayor profusión en las estructuras de clases son las de herencia y agregación, dando lugar a cuatro grandes tipos de clases: derivadas, agregadas, abstractas e interfaces. Estos conceptos se detallarán e ilustrarán con ejemplos sencillos a lo largo de esta sección.

7.1.1. Relaciones jerárquicas y semánticas

Los objetos forman siempre una organización jerárquica, en función del grado de abstracción que se desea representar. Por ejemplo el objeto "Persona" es más abstracto que los objetos "Hombre"

y "Mujer". Esta jerarquía provoca diferentes niveles de objetos, que van desde los que se sitúan en la raíz de la jerarquía (concepto muy abstracto y, por tanto, poco operativo) hasta los objetos terminales (objetos que realizan tareas específicas), pasando por los intermedios. Esta forma de organización presupone una estructura arbórea.

Estas relaciones se basan en el concepto de herencia, es decir en las relaciones de tipo padre-hijo entre objetos. El objeto *padre* es aquel que se encuentra situado inmediatamente encima de otro en el sistema del que ambos forman parte. Si un objeto A es padre de un objeto B, se dice también que B es *hijo* de A.

En estas jerarquías los objetos hijos heredan de sus padres algunas de sus características. Por ejemplo, si un objeto "Persona" tiene la capacidad de *andar y comer*, un objeto "Mujer", por ser hijo de "Persona" también tendrá la capacidad de *andar y comer*. Los hijos tienen además la posibilidad de adecuar dichas funcionalidades (métodos) a sus características, así el objeto "Mujer" tendrá su forma particular de *andar*, distinta de la del objeto "Hombre", mientras que ambos pueden utilizar el mismo método *comer* del objeto "Persona".

Los hijos, al estar diseñados para tareas más específicas que los padres, también pueden añadir nuevos métodos propios. Por ejemplo, el objeto "Mujer" puede tener un método *dar_a_luz*, que no estará definido en el objeto "Persona", porque sino lo heredaría el objeto "Hombre" y no tendría sentido.

En el ejemplo de organización en forma de árbol (véase Figura 7.1.), los objetos E y F son hijos de B mientras que D es padre de G, H e I. Por ejemplo el objeto A puede representar el concepto "Pintura" y los objetos B, C y D los conceptos de "Pintura sobre Tela", "Pintura sobre Papel" y "Pintura sobre Madera". Los objetos E y F podrían representar la "Pintura al Óleo" y la "Pintura con Acuarela".

Los objetos, a diferencia de las personas, pueden tener uno o varios padres. El ejemplo que aparece en el diagrama siguiente (véase Figura 7.2.) muestra como los objetos B y C son padres del objeto F. Siguiendo con el ejemplo anterior el objeto F podría ser una técnica de pintura que utilizara tanto pintura sobre tela como sobre papel.

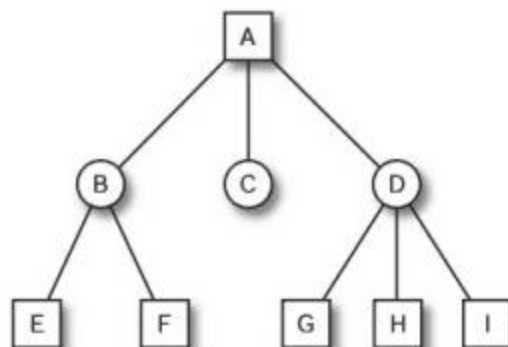


Figura 7.1. Organización en forma de árbol invertido de los objetos.

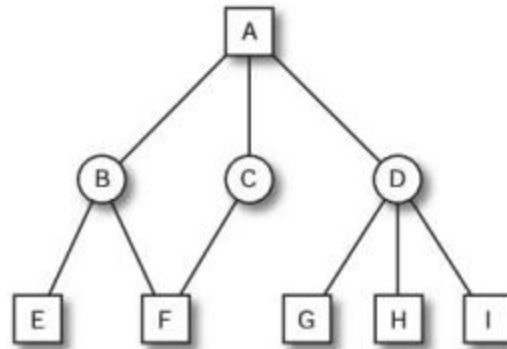


Figura 7.2. Organización no arbórea de los objetos.

Siguiendo la terminología utilizada en programación orientada a objetos, podemos establecer otros dos tipos de parentescos entre objetos: por un lado decimos que el objeto E es un *descendiente* del objeto A, si E es hijo de B y B es hijo de A. Por otro también se puede decir que el objeto A es un *antepasado* de B si A es padre de B, o si A es padre de un antepasado de B.

Las relaciones semánticas son aquellas cuyas propiedades sólo dependen de los objetos en sí mismos y no de su posición en la organización jerárquica. No importan donde estén, tan sólo se busca establecer algún vínculo semántico entre ellos. Por ejemplo, la relación que existe entre el objeto "Pintura" y el objeto "Pincel": uno no depende jerárquicamente del otro, pues no tienen propiedades comunes que puedan ser abstraídas de un concepto a otro, pero sí existirán distintos objetos hijos de "Pincel" que se relacionarán con los distintos objetos hijos de "Pintura", con una relación que podríamos denominar "Pintar".

7.1.2. Clases derivadas y agregadas. Representación de diagramas de clases

Las relaciones entre los objetos, y por tanto, entre las clases, se representan en diagramas de clases. Este tipo de diagramas aparece en las distintas metodologías de diseño orientadas a objetos, que aunque están fuera de los objetivos de este texto, ofrecen una visión práctica de las distintas clases. Sólo a modo de ejemplo se presentarán varios diagramas destinados a facilitar la comprensión de las posibles relaciones entre clases.

En primer lugar, lo importante es saber cómo se representa una clase, una relación y un objeto. Las dos relaciones más utilizadas son la de agregación y la de abstracción. La primera se utiliza cuando una clase contiene objetos de otra y la segunda da lugar a la creación de clases derivadas. Por ejemplo, en la Figura 7.3, se muestra las relaciones entre las clases Cero, Uno, Dos, Tres, introducidas en el capítulo anterior para ilustrar la herencia y el polimorfismo. Cada objeto de la clase Uno contiene un objeto de la clase Cero, lo que se indica mediante un rombo, la clase Dos hereda de la clase Uno y a su vez la clase Tres hereda de la clase Dos, relaciones indicadas mediante triángulos.

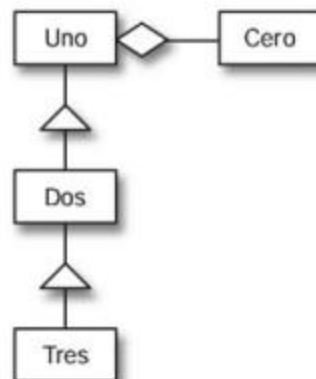


Figura 7.3. Ejemplo de relaciones de herencia y agregación.

Por otro lado, puede haber diferentes multiplicidades posibles para las relaciones, tal y como quedan recogidas en la Figura 7.4.

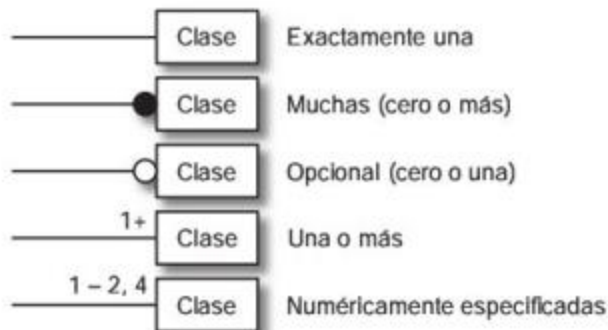


Figura 7.4. Definición de la multiplicidad en las relaciones.

A modo de ejemplo teórico, se muestra en la Figura 7.5. una relación de agregación entre clases con diferentes grados de multiplicidad, que representan los componentes de un computador. Según la figura, un computador se representa con una clase que contiene (relación de agregación): un teclado, una caja, opcionalmente un ratón, y uno o varios monitores. A su vez, la caja contiene una CPU, una o varias tarjetas, y una o varias unidades de memoria RAM.

Finalmente, es conveniente observar que los diagramas de clases que hemos presentado hasta ahora muestran relaciones estáticas entre clases, que en tiempo de ejecución se materializarán en relaciones de objetos específicos (instancias de esas clases). Así, a modo de ejemplo, en la Figura 7.5. se muestra una relación entre dos clases, Recta y Punto, que podríamos denominar “*recta pasa por*”.

La relación entre ambas clases está definida con un punto negro en cada extremo, lo que significa que es una relación de varios a varios y que se lee diciendo que una recta puede pasar por

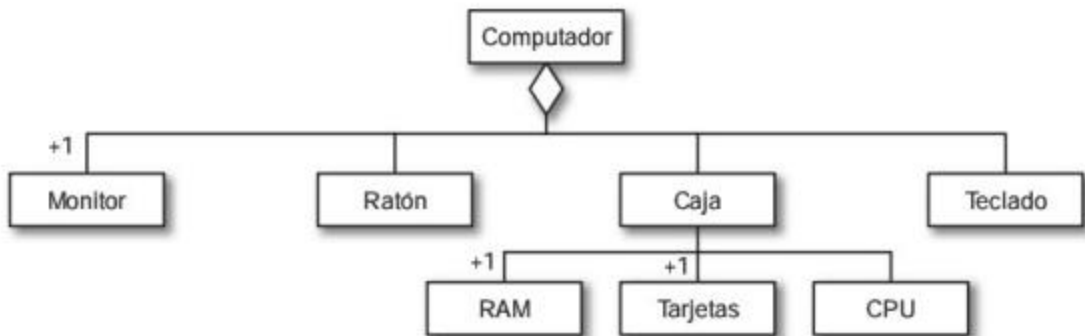


Figura 7.5. Ejemplo de relación de agregación con multiplicidad.

varios puntos y que por un punto pueden pasar varias rectas. A continuación, se muestra en la Figura 7.6. un ejemplo específico, que podría aparecer durante la ejecución, con un conjunto de objetos que cumplen esta relación. Obsérvese que los objetos se diferencian de las clases al representarse con bordes redondeados.

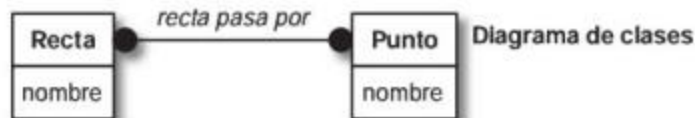
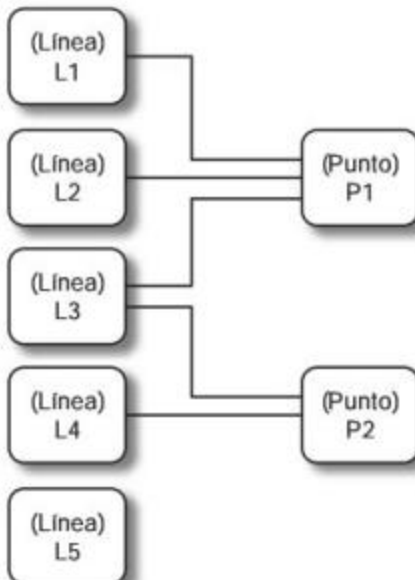


Figura 7.6. Ejemplo de relación de agregación con multiplicidad.

Diagrama de objetos



Datos ejemplo

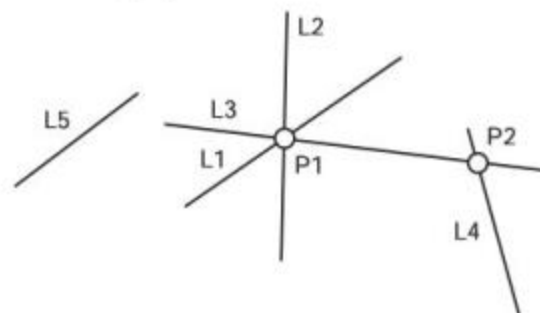


Figura 7.7. Ejemplo de clases, relaciones y objetos.

7.1.3. Clases abstractas e interfaces

Hemos visto que en las relaciones de generalización, representadas mediante árboles, los objetos pertenecen a una organización jerárquica, con diferentes niveles posibles de abstracción. Sin embargo, puede ocurrir que las clases que están en el nivel superior o niveles intermedios en la jerarquía, no tengan sentido como clases en sí mismas, es decir, que no pueden existir objetos de las mismas por no tener aún suficiente nivel de detalles para su implementación. En ese caso, estas clases únicamente servirán como referencias para las clases que deriven de ellas, dando una definición estándar que podría incluir la interfaz de los métodos que deben implementar las clases derivadas, de modo que el polimorfismo pueda aprovecharse hasta sus últimas consecuencias. En ese caso las clases se denominan abstractas, lo que significa que no pueden existir instancias de estas clases, únicamente de sus derivadas. Una clase abstracta, a su vez, puede contener métodos implementados (reutilizables por las clases derivadas o que pueden ser sobrescritos), y métodos abstractos, en cuyo caso únicamente se especifica la interfaz que obligatoriamente deberá implementarse en las clases derivadas de las que puedan generarse instancias. Así, un método abstracto será siempre polimórfico y generará una llamada al método particular de la clase derivada que se invoque en tiempo de ejecución, implementado en función de las características específicas de cada clase posible.

En la Figura 7.8. se muestra un ejemplo de jerarquía de clases para la representación geométrica de figuras, y que se propondrá como ejercicio en un apartado posterior. Todas las figuras geométricas planas con extensión bidimensional (con área no nula) descienden de una clase más general que es *FiguraDimension2*, y a su vez la *FiguraDimension2*, junto con otras de más o menos dimensiones puede generalizarse a la clase *Figura*. La existencia de clases abstractas se indica en el diagrama de clases mediante letra cursiva.

Un tipo de relación que involucra la generalización es la herencia múltiple, es decir la posibilidad de que una clase derive de dos clases simultáneamente. En Java este tipo de herencia no está permitida directamente entre las clases (en C++ sí) pero se puede soslayar ese problema mediante el uso de interfaces. En la Figura 7.9. se muestra un ejemplo de herencia múltiple.

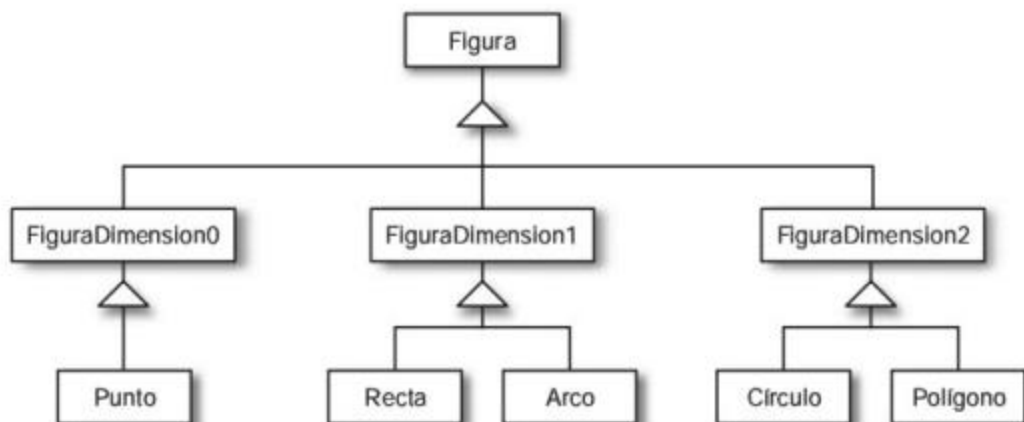


Figura 7.8. Ejemplo de relación de generalización.

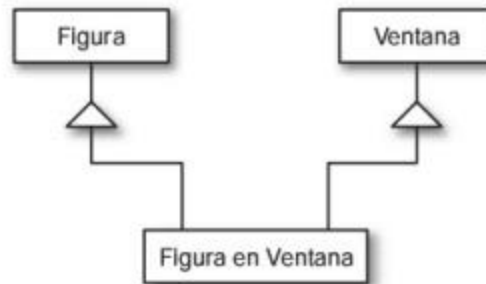


Figura 7.9. Ejemplo de herencia múltiple.

El concepto de interfaz representa una clase abstracta “pura”, en la que todos los métodos son abstractos y los atributos constantes, que sí permite multiplicidad para implementar el concepto de herencia múltiple en Java. Continuando con el ejemplo de las figuras, supongamos que queremos que los objetos construidos, además de representar figuras geométricas, incluyan propiedades que permitan su visualización gráfica en pantalla, definidos en una clase diferente, denominada `ClaseDibujo`. Para ello, una posibilidad sería que las clases derivadas de `Figura` a su vez heredasen los métodos de dibujo definidos en `ClaseDibujo`. La única posibilidad para implementar en Java esta herencia múltiple, es a través del mecanismo de interfaces. La clase `ClaseDibujo` es una interfaz que define unos métodos de dibujo para ser implementados por las clases derivadas de `Figura`, como se indica en la Figura 7.10.

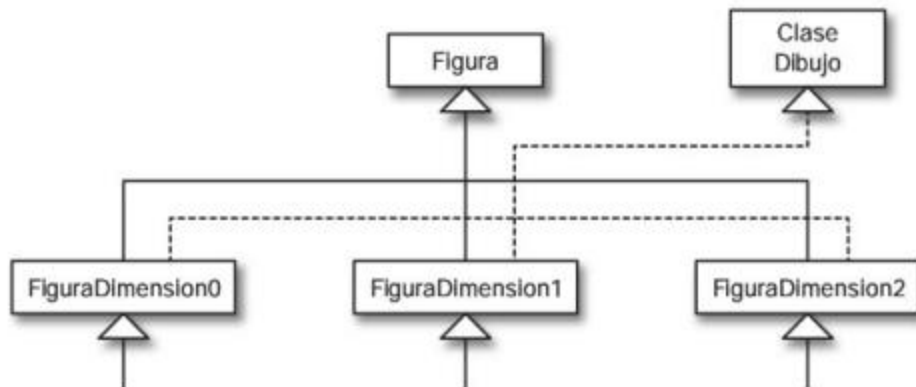


Figura 7.10. Herencia e Implementación de interfaz.

En este caso, `ClaseDibujo` define una interfaz que es implementada por cada una de las clases, también abstractas, derivadas de la clase `Figura`. Por tanto, los métodos definidos en la clase de dibujo deberán ser obligatoriamente implementados por cada una de las clases derivadas de `FiguraDimension0`, `FiguraDimension1` y `FiguraDimension2`. Obsérvese que las clases intermedias, `FiguraDimension0`, `FiguraDimension1` y `FiguraDimension2`, son a su vez abstractas, y no es obligatorio, por tanto, que implementen

los métodos de la interfaz `ClaseDibujo`. Únicamente aquellas clases de las que sí puedan instanciarse objetos (no abstractas) deberán obligatoriamente implementar todos los métodos abstractos definidos y no implementados en clases situadas por encima en la jerarquía. Además, todos los métodos que implementen los definidos en una interfaz deberán obligatoriamente contener el calificador de acceso **public**.

La relación de implementación de una interfaz se indica a través de línea discontinua en el diagrama de clases, y en el código Java se materializa a través de la palabra reservada **implements**. A su vez, la propiedad de una clase interfaz se indica a través de la palabra reservada **interface**, que impone como condiciones a la clase que sus métodos deben ser todos declarados y no implementados (abstractos). Todos los métodos de la interfaz son implícitamente públicos (aunque no se indique explícitamente), y deberán implementarse en las clases finales que deriven de la interfaz, obligatoriamente con el calificador **public**. Con clases finales nos referimos aquí a clases no abstractas, de las que sí pueden instanciarse objetos. En caso de que una clase abstracta implemente una interfaz, no es necesario que implemente los métodos de esta, pero sí lo harán sus clases derivadas finales. Por último, en caso de existir atributos en una interfaz, todos ellos serían implícitamente estáticos y constantes, es decir, de tipo **static** y **final**, no permitiéndose la existencia de atributos de instancia sobreescritos (con el mismo nombre que los atributos de la interfaz) en las clases que la implementen.

De este modo, la relación anterior (véase Figura 7.10.) se codificaría así:

```
public interface ClaseDibujo {...}
abstract class Figura {...}
abstract class FiguraDimension0 extends Figura implements ClaseDibujo
{...}
abstract class FiguraDimension1 extends Figura implements ClaseDibujo
{...}
...
```

Conviene en este punto mencionar que el uso de interfaces en las relaciones jerárquicas aporta una diferencia semántica frente a las clases derivadas, aparte de la razón práctica de su empleo en estructuras con herencia múltiple. Como hemos visto, en una estructura con relaciones jerárquicas tenemos que los niveles superiores presentan un mayor grado de abstracción, renunciando a los detalles específicos para agrupar las propiedades comunes de todas las clases que están por debajo en la jerarquía. Sin embargo, una interfaz presenta únicamente una funcionalidad de uso, que puede ser completamente independiente de las propiedades de las clases que la implementan. Así, dentro de este mismo ejemplo, podríamos tener que otras jerarquías de clases con naturaleza completamente distintas a las figuras geométricas, como podrían ser familias de gráficas de resultados, imágenes, etc., a su vez podrían también implementar la interfaz `ClaseDibujo` con el objeto de ser presentadas gráficamente en una ventana. Por tanto, vemos que una interfaz puede servir para agrupar funcionalidades de jerarquías de clases diferentes, manteniendo separadas las propiedades de cada una de ellas.

Siguiendo con el ejemplo, tal y como se ha indicado antes, es ilegal instanciar objetos de una clase abstracta o de una interfaz, por lo que las siguientes líneas serían incorrectas:

```
Figura miFigura=new Figura(); //ERROR
FiguraDimension2 figD2=new FiguraDimension2();// ERROR
ClaseDibujo dibujo=new ClaseDibujo();// ERROR
```

Sin embargo, es totalmente válido declarar objetos de una clase abstracta o interfaz e instanciarlos después recurriendo a una derivada:

```
Figura miFigura=new Circulo();
FiguraDimension2 figD2=new Poligono();
ClaseDibujo dibujo=new Circulo();
```

aunque es importante notar que estos objetos creados, `miFigura`, `figD2`, `dibujo`, han sido declarados como objetos de las clases genéricas `Figura`, `FiguraDimension2` y `ClaseDibujo`, por lo que únicamente podrían ser tratados como tales e invocar a los métodos de estas clases.

En la sección de ejercicios se propondrá el desarrollo completo de este ejemplo, que ilustra la utilización de interfaces para la construcción de una aplicación gráfica.

7.2. Diseño orientado a objetos (DOO)

En los apartados anteriores hemos realizado una introducción a la programación orientada a objetos. Esta metodología de programación lleva implícita numerosas ventajas frente a la programación tradicional (también llamada programación procedimental). Su modularidad, la capacidad para encapsular código y la generación de código fácilmente reutilizable mediante la herencia y el polimorfismo, son ventajas lo suficientemente potentes como para justificar su utilización.

7.2.1. Diseño estructurado y diseño orientado a objetos

Se llama software procedimental al software tradicional basado en procedimientos y que generalmente incorpora las características de la programación estructurada. En definitiva, el software procedimental es el software que actualmente está usando un mayor número de personas y aunque las aplicaciones importantes incorporan en su mayoría las técnicas de programación orientada a objetos, éstas no están todavía suficientemente arraigadas en los pequeños programadores.

Para los programadores acostumbrados a la programación procedimental no es sencillo conceptualmente esta forma de programar, pues supone una transformación de la forma de pensar de los diseñadores de aplicaciones. Teóricamente, el cambio de metodología de programación presupone un mayor acercamiento a la forma de razonamiento humano ya que es más fácil hablar de objetos, métodos y clases que de funciones, variables, etc. Pero en la práctica resulta difícil trasladar los algoritmos, que son fácilmente trasladables a procedimientos, a un diseño basado en objetos.

Es frecuente ilustrar las diferentes etapas del diseño de software tradicional como un conjunto de acontecimientos secuenciales. Estos procesos que van desde el establecimiento de los requisitos de la aplicación hasta la instalación final de la misma, constituyen el ciclo de vida del software procedimental. No es el objetivo de este texto ilustrar en detalle el proceso de Ingeniería del software, únicamente presentar a grandes rasgos las etapas que lo constituyen. Las fases que componen ese ciclo de vida son:

1. Análisis.

La etapa de análisis es fundamentalmente una etapa de especificación de la aplicación. Estas especificaciones se obtienen tradicionalmente a través de conversaciones entre los clientes y los diseñadores. El desarrollador se debe reunir con el cliente hasta que consiga tener una idea completa de las necesidades que éste plantee. Del buen entendimiento entre el cliente y el desarrollador depende el éxito o el fracaso de la aplicación. Es en esta etapa cuando se fijan los plazos de entrega y el coste total del proyecto.

2. Diseño.

En la etapa de diseño se pretende estructurar el conjunto de toda la aplicación de manera que sea abordable por un conjunto de desarrolladores coordinados. Quizá las palabras clave de esta etapa sean las de descomposición funcional. Con ella se busca la descomposición del problema en módulos (conjunto de procedimientos independientes que pueden desarrollarse por un único equipo) y se definen los datos para cada módulo. El diseño procedimental se fundamenta en los algoritmos (definidos como procedimientos), su objetivo es la algoritmización del problema, y deja de lado qué tipo de datos deben ser definidos para poder desarrollar la aplicación. Se dice que está más orientado a los programas que a los datos.

3. Codificación.

Los diferentes módulos diseñados en la etapa anterior se transforman en código ejecutable por los diferentes equipos de programadores. El lenguaje de programación, o bien ha sido especificado en los requisitos, o bien, son los propios programadores los que lo deciden en esta etapa. Los equipos escriben las diversas secuencias de código que componen cada uno de los procedimientos.

4. Prueba.

Generalmente, el programador realiza las pruebas de funcionamiento de todos y cada uno de los bloques que constituyen la aplicación. El fundamento de la programación procedimental es aislar los módulos de manera que cada uno de ellos pueda ser probado de forma independiente. Posteriormente se van probando las agregaciones de módulos, para, por último realizar la prueba final donde se verificará si se adapta a los requisitos planteados por el cliente en la fase de análisis. Todos los módulos probados y compilados se ensamblarán para formar un único código ejecutable que se entregará al usuario.

5. Mantenimiento.

Esta etapa puede ser muy costosa o innecesaria, y todo depende de cómo se haya diseñado el software. En esta fase se deberán corregir todos los errores del programador no

corregidos en la fase de prueba. Un mal diseño puede hacer que esta etapa sea interminable.

Aunque no es fácil de observar a primera vista, la disposición secuencial de las etapas y las tareas asociadas a cada una de ellas provocan una serie de problemas que pueden ser solventados mediante el diseño orientado a objetos.

En primer lugar, el desarrollo procedimental comienza por una fase de análisis donde se especifican los requisitos de la aplicación. Esta definición se produce mediante un acuerdo entre los clientes y desarrolladores y debe incluir toda la información y todas las especificaciones de la aplicación, puesto que no habrá más interacción con el cliente hasta la última fase. Cualquier error en este momento será arrastrado durante todo el desarrollo y no se detectará hasta que el usuario pruebe la aplicación. A partir de esta etapa, existe una escasa presencia del cliente en el desarrollo de la aplicación, puesto que en general, éste se ve incapaz de entender las especificaciones que está diseñando el programador o el analista. Además de estos posibles errores, es difícil evaluar desde la fase de análisis la complejidad del sistema, y esto provoca que tanto los costes como los plazos de entrega no se ajusten a lo previamente especificado.

En el caso del diseño orientado a objetos existe una fase previa al diseño donde se modela la aplicación en un lenguaje que permite su comprensión por parte del cliente y al mismo tiempo facilita el proceso de diseño posterior de manera que éste se ajuste al análisis realizado.

En el caso de la programación procedimental, el software diseñado está íntimamente relacionado con la aplicación concreta. Esto hace que los programas tradicionales sean difícilmente adaptables a otras necesidades distintas de la inicial, aunque se ha trabajado mucho en conseguir que los módulos sean independientes y fácilmente intercambiables. En realidad, los diferentes módulos que componen la aplicación están en general bastante acoplados, no llegándose casi nunca a una independencia entre módulos.

La definición de clases en el diseño orientado a objetos permite concentrarse más en el problema que en el algoritmo de resolución. Al hacer clases encapsuladas, estas clases son fácilmente reusables, tanto directamente como a través del mecanismo de la herencia.

En la fase de prueba, si el analista no ha comprendido correctamente el problema, las diferentes pruebas que haga las hará según su criterio, por lo que los errores no serán detectados hasta que en la última etapa el cliente evalúe el producto formado. En el diseño orientado a objetos se mantiene una realimentación entre todas las fases que permite la modificación constante de cada etapa y su ajuste en etapas posteriores.

Por último, en lo que respecta al mantenimiento de la aplicación, en el caso del software procedimental, el mantenimiento puede durar mucho tiempo, recuérdense los casos del efecto 2000 o el efecto euro. Esto hace que no siempre se encargue de esta tarea la misma persona que diseñó el programa, con las consiguientes dificultades que esto acarrea. En el diseño orientado a objetos, el mantenimiento se realiza a nivel de modelo, de manera que el encapsulamiento del software permite modificar el modelo (y por tanto, algunas de las clases) sin que éste tenga un coste excesivo y sin necesidad de contar con los mismos desarrolladores. Según lo comentado, en el diseño orientado las diferentes fases del proceso ya no se sitúan en cascada, sino que lo hacen realimentándose.

Por tanto, el ciclo de vida es repetitivo, esto es, está gobernado por numerosos procesos de realimentación. Cada etapa es revisada continuamente en función de los resultados de las demás. Cada etapa puede describirse de la siguiente manera:

1. Análisis de la aplicación

En líneas generales se puede decir que es en esta etapa donde cliente y desarrollador deben ponerse de acuerdo en la aplicación a realizar. Y si bien el objetivo de la etapa es común con la del software procedimental, es en la forma de lograrlo donde aparecen las diferencias. El desarrollador tiene que hacer uso de la capacidad de abstracción para construir modelos del problema real. Esta abstracción provoca la descomposición del problema en objetos, los cuales representarán las entidades que han de formar parte de la aplicación. Junto a la especificación de los objetos, se deben así mismo describir los métodos que se aplicarán a cada objeto. La elaboración de un modelo del problema, hace que el cliente pueda entender la representación que se ha hecho de sus necesidades y pueda evaluar si ha sido, o no, comprendido por el analista.

2. Diseño

En general, hay que decir que en el diseño orientado a objetos no es fácil establecer una frontera entre las diferentes etapas de desarrollo. La fase de diseño comienza a partir de los objetos obtenidos en la fase de análisis, pero es posible y en general habitual, que se necesite añadir nuevos objetos que serán necesarios para el control interno de la aplicación. Una vez hecha la aclaración de que las fronteras entre las diferentes etapas son difusas, se puede decir que la fase de diseño se caracteriza por el agrupamiento en clases de objetos (tanto de aquellos generados en la etapa de análisis como de los generados durante el diseño) y establecer la jerarquía. En esta fase debemos también definir los elementos de las bibliotecas disponibles y que puedan ser aplicables y reutilizables en nuestra aplicación. Con todo esto, refinamos la estructura del programa. Este refinamiento sigue la filosofía *bottom-up*, es decir, de abajo hacia arriba, ya que se comienza por los objetos, se agrupan en clases y se construye la estructura hasta llegar a la raíz de la jerarquía. Recordar, que los sistemas procedimentales se basaban en la filosofía contraria, la *top-down* (de arriba a abajo).

3. Codificación

Se procede a la creación de código para aquellos métodos nuevos, o bien se modifican aquellos métodos que hayamos sacado de la biblioteca de objetos y que no se adapten en su totalidad a los requisitos.

4. Pruebas

A diferencia de la programación tradicional, en la programación orientada a objetos no es necesario esperar al final de la aplicación para empezar las pruebas. Gracias al encapsulamiento, las distintas clases son totalmente independientes y pueden revisarse por separado a medida que se construyen. Una vez terminado el proceso de construcción, se deben hacer pruebas globales. Es posible que haya que hacer correcciones, reorganizar los objetos, etc., pero esto, dada la independencia entre objetos, es más fácil que en programación tradicional.

7.3. Ejercicios resueltos

7.3.1. Desarrollo de una aplicación gráfica

Enunciado En este primer ejercicio se propone el desarrollo de una aplicación gráfica que permita representar figuras geométricas en el plano: punto, recta, arco, círculo y polígono, tal y como se esbozó en el Apartado 7.1.3. Como se indicó entonces, las figuras estarán agrupadas además en clases intermedias que representen propiedades generales provenientes de su dimensionalidad: figuras de dimensión 0 (con longitud y área nulas, como el punto), que definen una distancia al origen; figuras de dimensión 1 (con área nula, como la recta y el arco), sobre las que se puede calcular una longitud; y figuras de dimensión 2 (el círculo y el polígono), que permiten calcular el valor del área que ocupan sobre el plano. Además, todas las figuras tendrán como propiedades adicionales el poder cargar sus valores de un fichero de texto, y ser dibujadas en una ventana gráfica. Esto último lo harán implementando métodos genéricos definidos en una interfaz, como se sugirió en el Apartado 7.1.3. Sin embargo, como variación a la estructura allí presentada, en la que algunas clases derivadas de *Figura* podrían ser "dibujables" y otras no, según se optara por implementar la clase *ClaseDibujo* o no, aquí impondremos la condición de implementar obligatoriamente la interfaz en la clase primera de la jerarquía, tal y como se indica en la estructura de la Figura 7.11. Esto nos permitirá aprovechar el polimorfismo y hacer que el programa principal declare un array abstracto de figuras, asegurándose de que todas ellas pueden leerse de fichero y ser dibujadas, independientemente de la clase específica final.

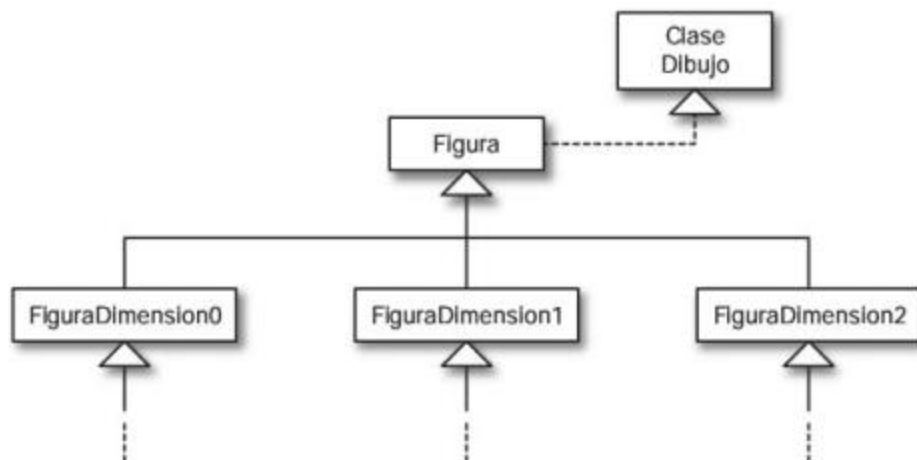


Figura 7.11. Herencia e implementación de interfaz en clase raíz.

Solución En primer lugar, la clase interfaz, *ClaseDibujo* únicamente define el método *dibujar*, que será implementado por todas las clases finales derivadas de *Figura*. Este método se va a implementar aprovechando la clase de utilidad *Graphics*, que permitirá visualizar diferentes objetos

en ventanas gráficas dentro de applets. Para ello, el método `dibujar` recibirá como parámetro una referencia a un objeto de esta clase:

```
import java.awt.Graphics;
public interface ClaseDibujo
{
    public void dibujar(Graphics dw);
}
```

En cuanto a la jerarquía de las figuras, primero aparece la clase abstracta `Figura`, que define dos métodos abstractos: `leerDatos` y `dibujar`:

```
import java.io.*;
import java.awt.Graphics;
abstract class Figura implements ClaseDibujo
{
    abstract void leerDatos(BufferedReader br);
    abstract public void dibujar (Graphics dw);
}
```

Por tanto, no puede existir ningún objeto de la clase abstracta `Figura`, sino únicamente de clases derivadas de ésta. Como vemos, la clase `Figura` implementa la interfaz definida en `ClaseDibujo` a través de la palabra reservada **`implements`**, que impone la presencia del método `dibujar`, aunque este método es a su vez abstracto. Esto quiere decir que realmente la clase `Figura` lo que hace es obligar a sus clases derivadas a que implementen el método definido en la interfaz `ClaseDibujo`, utilizando para ello las propiedades particulares definidas en cada una de ellas. Por tanto, cada clase no abstracta derivada de `Figura` deberá implementar obligatoriamente los dos métodos abstractos definidos: `dibujar` y `leerDatos`. Naturalmente, como ocurre con la herencia en general, si una clase implementa los métodos de una interfaz, todas las que deriven de ésta ya contendrán al menos esa implementación, teniendo la posibilidad de sobrescribirla con su propia versión. El método `leerDatos` recibe como parámetro un lector de flujo de datos de texto con buffer genérico, tal y como los que se explicaron en el Capítulo 5, que en este ejemplo concreto se materializará como una conexión a un fichero de texto de entrada. Sería válido el mismo método para cualquier otro dispositivo disponible que proporcionara una salida textual en formato Unicode.

Después, en un segundo nivel, tenemos las clases, también abstractas, correspondientes a las figuras de 0, 1 y 2 dimensiones:

```
abstract class FiguraDimension0 extends Figura
{
    abstract double distanciaOrigen();
}

abstract class FiguraDimension1 extends Figura
{
    abstract double longitud();
}
```



```
abstract class FiguraDimension2 extends Figura
{
    abstract double    area();
}
```

Como podemos ver, todas estas clases derivadas (a través de la palabra reservada **extends**) son a su vez abstractas y por tanto no es obligatorio que implementen los métodos abstractos `leerDatos` y `dibujar`, ya que no existirán instancias de esta clase. La tarea de implementar los métodos abstractos definidos en `Figura` se propaga, por tanto, a las clases particulares finales. Sin embargo, a este nivel, estas clases introducen una diferenciación según la dimensionalidad de las figuras, y definen a su vez métodos abstractos específicos (sólo tienen sentido para cada dimensionalidad) que serán implementados por cada clase derivada de cada una de ellas: `distanciaOrigen`, `longitud`, `area`, junto a los métodos abstractos definidos y no implementados en niveles superiores de la jerarquía.

Si pasamos ahora al tercer nivel, se han definido las cinco clases finales requeridas: `Punto`, `Recta`, `Arco`, `Círculo` y `Polígono`, de las que sí existirán objetos concretos (instancias). Estas clases se utilizarán para representar figuras que pueden ser almacenadas en ficheros de textos y después representadas en pantalla a través de un navegador HTML. El diagrama resultante de clases se ilustra en la Figura 7.12.

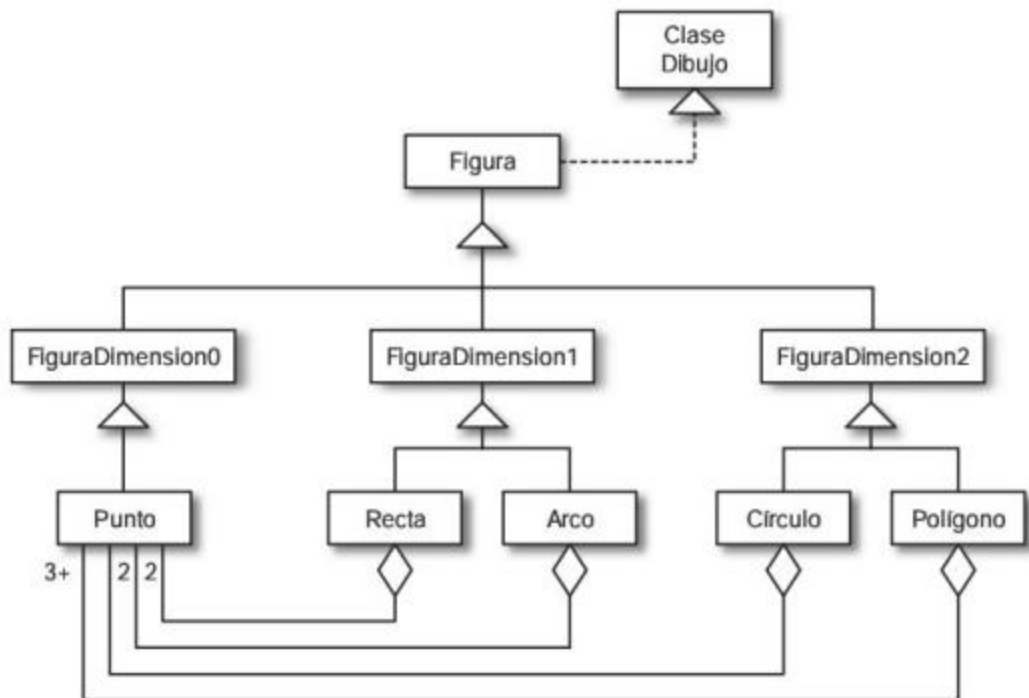


Figura 7.12. Diagrama de clases implementadas.

Obsérvese que además de las relaciones de abstracción, todas las figuras complejas guardan una relación de agregación con la clase `Punto`, que servirá para construir cada una de las figuras de una y dos dimensiones: la recta y el arco contendrán dos puntos en su definición, un punto la clase correspondiente al círculo, y al menos 3 puntos para la clase que representa a los polígonos.

El código de la primera clase, la única de dimensión 0, `Punto`, se presenta a continuación. En primer lugar, al ser la clase `Punto` derivada de la clase `FiguraDimension0`, obligatoriamente implementará el método abstracto de ésta, `distanciaOrigen`, y al ser a su vez derivado de la clase `Figura`, deberá implementar los métodos `leerDatos` y `dibujar`.

Por otro lado, se definen los atributos y métodos específicos de un punto geométrico, que está representado con un par de valores reales con las coordenadas cartesianas. Se han definido tres constructores distintos para la clase `Punto`: un constructor sin parámetros, un constructor a partir de las coordenadas como parámetros, y otro a partir de la referencia a un objeto de tipo punto ya existente. Según el programa en que se utilice esta clase, se empleará el más adecuado. Como se verá después, en este ejemplo concreto el mecanismo implementado será construir objetos de tipo `Punto` con el constructor sin parámetros (por defecto), para a continuación cargar los datos de un fichero de texto a través del método `leerDatos`. Por último se han definido algunos métodos que serán de utilidad para manipular varios objetos de esta misma clase: métodos para determinar las coordenadas o cambiar sus valores, para calcular la distancia a otro punto, etc., y que a su vez serán utilizados en los métodos de clases que contengan objetos de la clase `Punto`.

Obsérvese que para implementar el método heredado `dibujar` se adapta la representación interna de la información de las coordenadas, en variables de tipo **double**, a la interfaz con enteros que ha impuesto el método de la clase **Graphics**, parámetros de tipo `int`, mediante un operador `cast` de conversión explícita de tipos (reducción).

```
import java.awt.Graphics;
import java.io.*;
import java.util.StringTokenizer;
class Punto extends FiguraDimension0{
    private double x;
    private double y;
//Constructores
    //Constructor para inicializar a un valor
    public Punto(){
        x=0;
        y=0;
    }
    //Constructor para inicializar con valores como parametros
    public Punto(double x,double y){
        this.x=x;
        this.y=y;
    }
    //Constructor a partir de otro objeto
```

```

    Punto (Punto otro){
        x=otro.x;
        y=otro.y;
    }
// implementacion de metodos abstractos
public void leerDatos(BufferedReader br)
{
    try
    {
        String linea=br.readLine();
        StringTokenizer sr=new StringTokenizer(linea," ");
        x=Double.valueOf(sr.nextToken()).doubleValue();
        y=Double.valueOf(sr.nextToken()).doubleValue();
        System.out.println("datos leidos punto: x="+x+", y="+y);
    }
    catch (IOException e)
    {
        System.out.println("error lectura en punto");
    }
}
public void dibujar (Graphics dw)
{
    dw.drawOval((int)Math.floor(x), (int)Math.floor(y),1,1);
}
public double distanciaOrigen()
{
    return Math.sqrt(x*x+y*y);
}
//Inspectores
public double corx(){
    return(x);
}
public double cory(){
    return(y);
}
//Cambio datos
public void setx(double x_p){
    x=x_p;
}
public void sety(double y_p){
    y=y_p;
}
//Metodos de utilidad
// Calcula la distancia entre dos puntos
public double distancia(Punto otro){

```

```

        double xdiff, ydiff;
        xdiff=x-otro.x;
        ydiff=y-otro.y;
        return Math.sqrt(xdiff*xdiff+ydiff*ydiff);
    }
    //Si dos puntos son iguales
    boolean iguales(Punto otro){
        return ((x==otro.x) && (y==otro.y));
    }
}

```

En cuanto a las clases derivadas de `FiguraDimension1`, que representan aquellas sobre las que tiene sentido hablar de longitud, a continuación se presenta el código de las clases **Recta** y **Arco**. La clase `recta` contiene dos puntos (relación de agregación). Obsérvese que de nuevo se definen varios constructores con diferentes interfaces. Se implementa el método abstracto de la clase antepasado inmediato, `longitud` en `FiguraDimension1`, aprovechando a su vez funcionalidades de los objetos de tipo `Punto` agregados. Así, se calcula la longitud de la recta (en realidad segmento) utilizando el método de cálculo de distancia contenido en los objetos agregados de clase `Punto`. A continuación, se implementan los métodos abstractos de la clase `Figura`, `leerDatos` y `dibujar`.

```

import java.awt.Graphics;
import java.io.*;
import java.util.StringTokenizer;
import Punto;
class Recta extends FiguraDimension1{
    private Punto P1;
    private Punto P2;
    //Constructores
    //Constructor sin parámetros
    public Recta(){
        P1=new Punto();
        P2=new Punto();
    }
    //Constructor para inicializar con valores como parametros
    public Recta(double x1,double y1, double x2, double y2){
        P1=new Punto(x1, y1);
        P2=new Punto(x2, y2);
    }
    //Constructor a partir de dos puntos
    public Recta (Punto p1, Punto p2){
        P1 = new Punto(p1);
        P2 = new Punto(p2);
    }
    // implementacion de metodos abstractos
    double longitud()
    {

```



```

        return P1.distancia(P2);
    }
    public void leerDatos(BufferedReader br)
    {
        double x,y;
        try
        {
            String linea=br.readLine();
            StringTokenizer sr=new StringTokenizer(linea," ");
            x=Double.valueOf(sr.nextToken()).doubleValue();
            y=Double.valueOf(sr.nextToken()).doubleValue();
            P1.setx(x);
            P1.sety(y);
            x=Double.valueOf(sr.nextToken()).doubleValue();
            y=Double.valueOf(sr.nextToken()).doubleValue();
            P2.setx(x);
            P2.sety(y);
        }
        catch (IOException e)
        {
            System.out.println("error lectura en recta");
        }
    }
    public void dibujar (Graphics dw)
    {
        dw.drawLine((int)Math.floor(P1.corx()),
                    (int)Math.floor(P1.cory()), (int)Math.floor(P2.corx()),
                    (int)Math.floor(P2.cory()));
    }
}

```

En cuanto a la clase *Arco*, también contiene dos puntos, que especifican ahora el comienzo del arco y el centro del círculo que contiene a éste, y un atributo de tipo real con la longitud del arco en radianes. De nuevo, se definen varios constructores con diferentes interfaces y se implementan los métodos abstractos heredados *distancia*, *leerDatos* y *dibujar*, considerando las propiedades específicas de los objetos de esta clase.

```

import java.awt.Graphics;
import java.io.*;
import java.util.StringTokenizer;
import Punto;
class Arco extends FiguraDimension1{
    private Punto P1;
    private Punto centro;
    private double angulo;
//Constructores

```

```

//Constructor para inicializara un valor
public Arco(){
    P1=new Punto();
    centro=new Punto();
    angulo=0;
}

//Constructor para inicializar con valores como parametros
public Arco(double x1,double y1, double x2, double y2,double angulo_p){
    P1=new Punto(x1, y1);
    centro=new Punto(x2, y2);
    angulo=angulo_p;
}

//Constructor a partir de dos puntos
public Arco (Punto p1, Punto p2, double angulo_p){
    P1 = new Punto(p1);
    centro = new Punto(p2);
    angulo=angulo_p;
}

// implementacion de metodo abstracto de figuradimension1
double longitud()
{
    double radio_l=centro.distancia(P1);
    return radio_l*angulo;
}

public void leerDatos(BufferedReader br)
{
    double x,y;
    try{
        String linea=br.readLine();
        StringTokenizer sr=new StringTokenizer(linea," ");
        x=Double.valueOf(sr.nextToken()).doubleValue();
        y=Double.valueOf(sr.nextToken()).doubleValue();
        P1.setx(x);
        P1.sety(y);
        x=Double.valueOf(sr.nextToken()).doubleValue();
        y=Double.valueOf(sr.nextToken()).doubleValue();
        centro.setx(x);
        centro.sety(y);
        angulo=Double.valueOf(sr.nextToken()).doubleValue();
    }
    catch (IOException e){
        System.out.println("error lectura en arco");
    }
}

```

```

    }
    public void dibujar (Graphics dw)
    {
        double radio_l=centro.distancia(P1);
        dw.drawArc(
            (int)Math.floor(centro.corx()-radio_l),
            (int)Math.floor(centro.cory()-radio_l),
            (int)Math.floor(2*radio_l),
            (int)Math.floor(2*radio_l),
            (int)Math.floor((180/Math.PI)*
                Math.atan2(P1.cory()-centro.cory(),
                    P1.corx()-centro.corx())),
            (int)Math.floor((180/Math.PI)*angulo) );
    }
}

```

Finalmente, en la jerarquía presentada, en cuanto a las clases derivadas de *FiguraDimension2*, que definen una superficie, a continuación se presenta el código correspondiente a las clases *Circulo* y *PoligonoConvexo*. La clase *Circulo* contiene un objeto de la clase *Punto* y un valor del radio, mientras que la clase *PoligonoConvexo* contiene un array de objetos de la clase *Punto* y el número de vértices. Ambas clases implementan en primer lugar el método abstracto *area*, definido únicamente en la clase antepasado inmediata, *FiguraDimension2*. Obsérvese que la implementación de la clase *Circulo* es sencilla, πr^2 , mientras que la de *PoligonoConvexo* recurre a una descomposición en triángulos y un método privado para calcular el área cubierta por cada uno de los triángulos que pueden extraerse del polígono, compartiendo el primer vértice. De hecho, este procedimiento es correcto por la limitación impuesta de que el polígono sea convexo. En la Figura 7.13. se ilustra gráficamente el pro-

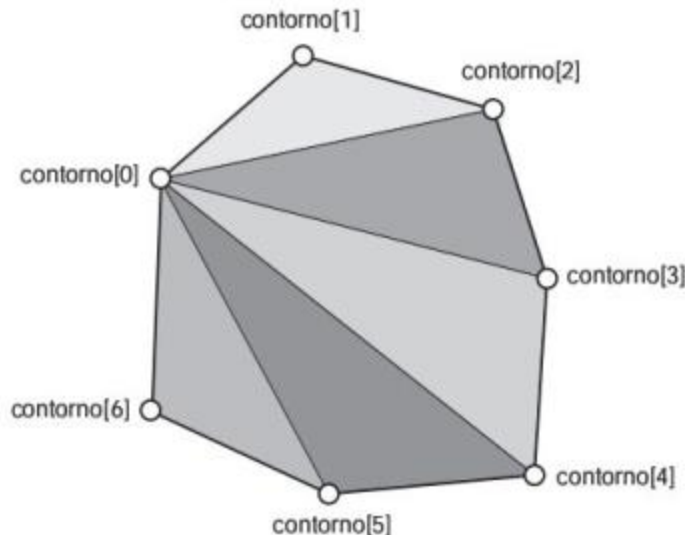


Figura 7.13. Cálculo del área de un polígono convexo.

cedimiento. Finalmente, se implementan los métodos de la clase *Figura*: *leerDatos* y *dibujar* atendiendo a las características específicas de cada clase:

```
import java.awt.Graphics;
import java.io.*;
import java.util.StringTokenizer;
import Punto;
class Circulo extends FiguraDimension2{
    private Punto centro;
    private double radio;
//Constructores
    //Constructor para inicializara un valor
    public Circulo(){
        centro=new Punto();
        radio=0;
    }
    //Constructor para inicializar con valores como parametros
    public Circulo(double x1,double y1, double radio_p){
        centro=new Punto(x1, y1);
        radio=radio_p;
    }
    //Constructor a partir de dos puntos
    public Circulo (Punto p1, double radio_p){
        centro = new Punto(p1);
        radio=radio_p;
    }
// implementacion de metodo abstracto de figuradimension2
    double area()
    {
        return Math.PI*radio*radio;
    }

    public void leerDatos(BufferedReader br)
    {
        double x,y;
        System.out.println("leyendo datos circulo");
        try
        {
            String linea=br.readLine();
            StringTokenizer sr=new StringTokenizer(linea," ");
            x=Double.valueOf(sr.nextToken()).doubleValue();
            y=Double.valueOf(sr.nextToken()).doubleValue();
            centro.setX(x);
            centro.setY(y);
            radio=Double.valueOf(sr.nextToken()).doubleValue();
            System.out.println("datos leidos: x="+x+", y="+y+
                "\n, radio="+radio);
        }
    }
}
```



```

        catch (IOException e)
        {
            System.out.println("error lectura en circulo");
        }
    }
    public void dibujar (Graphics dw)
    {
        dw.drawOval( (int)Math.floor(centro.corx()),
                     (int)Math.floor(centro.cory()),
                     (int)Math.floor(2*radio), (int)Math.floor(2*radio));
    }
}

import java.awt.Graphics;
import java.io.*;
import java.util.StringTokenizer;
import Punto;
class PoligonoConvexo extends FiguraDimension2{
    private Punto[] contorno;
    private int vertices;
    //Constructores
    //Constructor para inicializara un valor
    public PoligonoConvexo(){
    }
    // implementacion de metodo abstracto de figuradimension2
    double area()
    {
        double areaParcial=0;
        for(int n=1;n<vertices-1;n++)
        {
            areaParcial+=
                areaTriangulo(contorno[0],contorno[n],contorno[n+1]);
        }
        return areaParcial;
    }
    public void leerDatos(BufferedReader br)
    {
        double x,y;
        try
        {
            String linea=br.readLine();
            StringTokenizer sr=new StringTokenizer(linea," ");
            vertices=sr.countTokens()/2;
            contorno=new Punto[vertices];
            for(int n=0;n<vertices;n++)
            {

```

```

        x=Double.valueOf(sr.nextToken()).doubleValue();
        y=Double.valueOf(sr.nextToken()).doubleValue();
        contorno[n]=new Punto(x,y);
    }
}
catch (IOException e)
{
    System.out.println("error lectura en poligono");
}
}
public void dibujar (Graphics dw)
{
    int[] coordx=new int[getVertices()];
    int[] coordy=new int[getVertices()];
    Punto[] contorno_l=getContorno();
    for(int n=0;n<getVertices();n++)
    {
        coordx[n]=(int)Math.floor(contorno_l[n].corx());
        coordy[n]=(int)Math.floor(contorno_l[n].cory());
    }
    dw.drawPolygon(coordx, coordy, getVertices());
}
public int getVertices()
{
    return vertices;
}
public Punto[] getContorno()
{
    return contorno;
}

private double areaTriangulo(Punto p1, Punto p2, Punto p3)
{
    //es simplemente la mitad del producto vectorial de
    // las diferencias
    return Math.abs(
        0.5*( (p2.corx()-p1.corx())*(p3.cory()-p1.cory())-
            (p3.corx()-p1.corx())*(p2.cory()-p1.cory()) ) );
}
}

```

Para concluir el ejercicio, a continuación se presenta la clase ejecutable que integrará objetos de las clases dentro de la jerarquía desarrollada a lo largo de este ejemplo. Se trata de un programa genérico que aprovecha el polimorfismo presente en los métodos para leer los datos de las figuras y dibujarlas en pantalla, operaciones que realizará cada objeto en función de sus peculiaridades. Se supone que existe un fichero de texto, "figuras.txt", donde están codificadas una serie de figuras geométricas. Para ello, el formato de fichero es el siguiente: se presenta para cada figura en primer lugar el nombre de ésta, siendo los valores válidos "círculo", "polígono", "arco",

"recta" y "punto", y a continuación, en la siguiente línea, una serie de valores reales de longitud variable, que representan atributos de cada tipo de figura que se va a representar. A continuación, se muestra un ejemplo:

Ejemplo

fichero "figuras.txt":

```

circulo
400 50 37
poligono
40 1 100 1 140 60 70 101 1 60
arco
180 300 240 300 1.57
punto
1 1
punto
400 1
punto
400 200
punto
1 200
recta
1 400 300 300

```

Con el motivo básico de simplificar en lo posible el ejemplo desarrollado, el método dibujar de todas las clases se implementa siempre a través de un objeto de la clase Graphics, recurriendo a métodos básicos definidos en ésta, como drawPolygon, drawOval, drawArc y drawLine. Todos ellos pueden consultarse en la API de Java. La forma más natural de ejecutar estos métodos es a través de del método paint dentro de un applet, en lugar de hacer una aplicación con un método main. Así, este programa no constituye un programa directamente ejecutable (a través de un método main), sino que deriva de la clase Applet, y habrá que invocarlo a través de la función paint. El código de la clase que accede a este fichero y lo muestra en una ventana gráfica, EjemploFiguras se presenta a continuación:

```

import java.io.*;

public class EjemploFiguras extends Applet
{
    public void paint (Graphics g){
        Figura [] conjuntoFiguras=null;
        int numeroFiguras=0;
        try
        {
            // dos pasadas: la primera para saber el tamaño y la segunda
            // para rellenar el array
            FileReader fr=new FileReader("figuras.txt");
            BufferedReader br=new BufferedReader(fr);
            String linea=br.readLine();
            while (linea!=null)

```


que dependerá del tipo de clase concreta contenida en la posición que indica la variable índice. Exactamente lo mismo ocurre con la representación en pantalla: una vez creado el array e inicializado correctamente, se dibujan en la ventana todas las figuras mediante el bucle:

```
for (int n=0;n<conjuntoFiguras.length;n++ )
{
    conjuntoFiguras[n].dibujar(g);
}
```

donde el método `dibujar` es diferente para cada clase concreta final en cada componente del array abstracto, y se decide en tiempo de ejecución. Ambos procesos, lectura y dibujo de figuras, son un ejemplo de uso de polimorfismo para incrementar la flexibilidad y potencia del programa.

Una posibilidad para visualizar este ejemplo es crear el siguiente fichero HTML, `ejemplo.html`:

```
<applet code="EjemploFiguras" width=380 height=150>
</applet>
```

e invocarlo a través del visualizador de applets:

appletviewer ejemplo.html

Otra posibilidad sería cargar simplemente el fichero `"ejemplo.html"` en un navegador web. Para el fichero de figuras, `"figuras.txt"`, la salida será la presentada en la Figura 7.14.

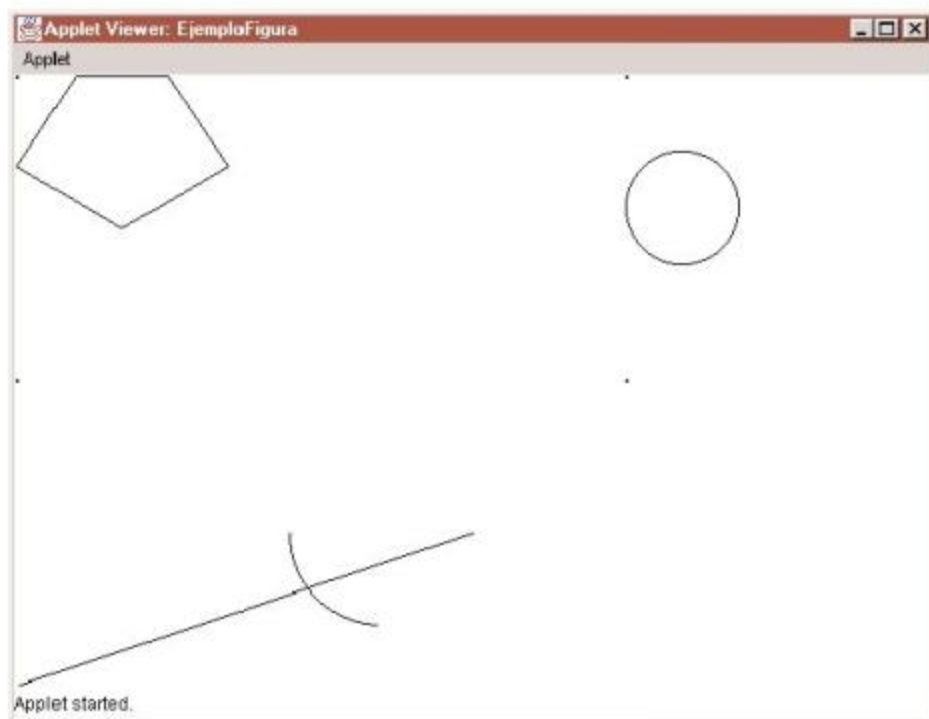


Figura 7.14. Salida del ejemplo presentado en el Ejercicio 7.3.1.

7.3.2. Desarrollo de una aplicación de representación de mapas

Enunciado Se quiere representar de forma simplificada un mapa de ciudad con objetos geométricos sencillos. Para ello se van a utilizar elementos (de una clase genérica, **Elemento**) únicamente de tres tipos posibles: calles, edificios y plazas (que pertenecerán, respectivamente a las clases **Calle**, **Edificio** y **Plaza**), con un identificador asociado. La pieza básica de representación en el mapa serán regiones (clase **Base**) que podrán ser de tipo rectangular o circular (clases **Rectangulo** y **Circulo**). Por simplicidad, los rectángulos siempre estarán alineados con las coordenadas horizontales, XY. Una calle contendrá una o varias regiones, las plazas y los edificios una única región, y estos últimos además tendrán asociada una altura determinada. En la figura siguiente se muestra un ejemplo con 2 plazas, 4 calles y 8 edificios:



Figura 7.15. Ejemplo de representación simplificada de un mapa de una calle.

Se pretende utilizar esta representación, que estará almacenada en un objeto de la clase **Mpa** que contenga tantos objetos de tipo **Elemento** como se necesiten, para implementar una función de localización en el mapa a partir de una posición en el plano, (x,y) , o una posición tridimensional que incluya altura, (x,y,h) . Para ello, la clase **Elemento** incluirá métodos denominados *localizada*, que para una posición dada, (x,y) o (x,y,h) , determinen si ésta se encuentra o no dentro de la región o regiones delimitadas por dicho elemento:

- En el caso de una posición (x,y) , únicamente se deberá comprobar que ésta se encuentra dentro de la base (o bases) que compone el elemento, para lo que se precisa que la clase **Base** a su vez contenga un método *localizada(x,y)*, cuya implementación dependerá del tipo de base (circular o rectangular).
- Para una posición extendida (x,y,h) , además de la condición anterior, deberá verificarse en el caso de calles y plazas que la posición está en el suelo ($h = 0$), o en el caso de edificios que la altura sea inferior a la de éstos.

Se pide:

- Diseño de un diagrama con las clases **Mapa**, **Elemento**, **Plaza**, **Calle**, **Edificio**, **Base**, **Círculo** y **Rectángulo**, indicando el tipo de clase y los tipos de relaciones que se den (jerárquicas o de agregación).
- Implementar en Java el código de las clases **Elemento**, **Plaza**, **Calle**, **Edificio**, **Base**, **Círculo** y **Rectángulo**. En cada clase deberán incluirse los atributos necesarios para representar la información indicada, constructores que inicialicen estos atributos, y los métodos necesarios para que los objetos de la clase **Elemento** puedan determinar si una posición bidimensional o tridimensional está localizada dentro de éste o no. Indicar claramente en qué casos se da sobrecarga y sobreescritura con polimorfismo de métodos.
- Implementar en Java el código de la clase **Mapa**, incluyendo atributos, constructor y los métodos

```
String situar (double x, double y)
```

```
String situar (double x, double y, double h)
```

que devuelvan una línea de texto con los identificadores de los elementos del mapa donde se encuentre la posición indicada.

Solución

- El objeto de la clase Mapa estará compuesto (agregación) de uno o varios objetos de tipo Elemento (clase abstracta), pudiendo ser éstos de tipo Plaza, Calle o Edificio (relaciones jerárquicas). Los objetos de tipo Edificio y Plaza contendrán a su vez (agrega-

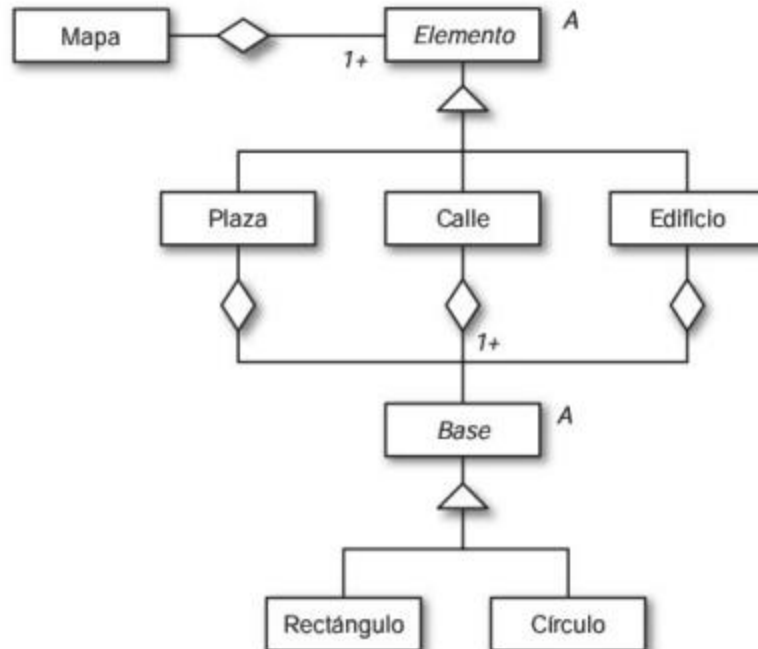


Figura 7.16. Diagrama de clases correspondiente al Ejercicio 7.3.2.

ción) un objeto de tipo Base, mientras que los de la clase Calle pueden contener más de una base. En cuanto a la clase Base es abstracta, pudiendo ser sus derivadas de tipo Rectángulo o Círculo (relaciones jerárquicas).

- b)** La clase abstracta Elemento únicamente contiene como atributo propio el identificador, puesto que el resto de información necesaria para su funcionalidad de localización en el mapa dependerá del tipo específico de elemento que se trate. A su vez, define la interfaz de los métodos sobrecargados localizada que deberán implementar las clases derivadas:

```
abstract class Elemento
{
    private String nombre;
    Elemento (String nombre)
        {this.nombre=nombre;}
    public String devNombre()
        {return nombre;}
    abstract public boolean localizada (double x, double y);
    abstract public boolean localizada (double x, double y, double h);
}
```

En cuanto a la clase Plaza, define un atributo de tipo Base, que le servirá para implementar los métodos localizada. En el caso de que la posición sea tridimensional, cumplirá la condición de pertenecer a la plaza si se encuentra sobre el suelo y a su vez está dentro de los bordes de la base (ya sea rectangular o circular). Si la posición ya está definida sobre el plano, basta con comprobar la segunda condición:

```
class Plaza extends Elemento
{
    private Base base;
    Plaza (String nombre, Base base)
        {
            super(nombre);
            this.base=base;
        }
    public boolean localizada (double x, double y)
        {return base.localizada(x, y);}
    public boolean localizada (double x, double y, double h)
        {
            if (h==0)
                return base.localizada(x,y);
            else return false;
        }
}
```


La clase `Edificio` es similar a la clase `Plaza`, pero define un atributo propio que es la altura del edificio, además del objeto de tipo `Base`, que sirve para delimitar el contorno de la planta del edificio. En el caso de que la posición de entrada sea tridimensional, cumplirá la condición de pertenecer al edificio si su altura está comprendida entre el suelo y la altura del edificio y, asimismo, su proyección horizontal se encuentra dentro la base. Si la posición ya está definida sobre el plano, basta con comprobar la segunda condición:

```
class Edificio extends Elemento
{
    private Base base;
    private double altura;
    Edificio (String nombre, Base base, double altura)
    {
        super(nombre);
        this.base=base;
        this.altura=altura;
    }
    public boolean localizada (double x, double y)
    {return base.localizada(x, y);}
    public boolean localizada (double x, double y, double h)
    {
        if (h<=altura&&h>=0)
            return base.localizada(x,y);
        else return false;
    }
}
```

La particularidad de la clase `Calle` es que está compuesta de un número indeterminado de objetos de tipo `Base`, dependiendo de la forma específica de ésta. Para ello, define un atributo con un array de objetos de tipo `Base`, que se construirá en el momento de instanciar la calle específica. Así mismo, los métodos `localizada` son similares a los de las clases `Plaza` y `Edificio`, pero esta vez, la condición de pertenencia de la proyección horizontal al contorno vendrá dada por la disyunción de pertenecer a alguno de los segmentos que definen la calle. Esto se ha implementado mediante un bucle `for` extendido a todos los elementos del array, y composición de condiciones mediante el operador `OR`:

```
class Calle extends Elemento
{
    private Base[] segmentos;
    Calle (String nombre, Base[] bases)
    {
        super(nombre);
        segmentos=new Base[bases.length];
        for (int k=0;k<bases.length;k++)
            segmentos[k]=bases[k];
    }
}
```

```

public boolean localizada (double x, double y)
{
    boolean esta=false;
    for (int k=0;k<segmentos.length;k++)
        if (segmentos[k].localizada(x,y))
            esta=true;
    return esta;
}

public boolean localizada (double x, double y, double h)
{
    boolean esta=false;
    if (h==0)
        for (int k=0;k<segmentos.length;k++)
            if (segmentos[k].localizada(x,y))
                esta=true;
    return esta;
}
}

```

La clase Base representa los límites de los elementos en el plano horizontal, pero esta funcionalidad depende del tipo de base específico que se use, por lo que no define ningún atributo propio y únicamente especifica el método de localización en el plano.

```

abstract class Base
{
    abstract public boolean localizada(double x, double y);
}

```

La clase Rectangulo representa figuras rectangulares alineadas con los ejes, para lo que únicamente necesita las coordenadas de dos vértices opuestos. La condición de localización de una posición 2D dentro de sus límites se puede expresar de manera sencilla con desigualdades independientes para cada coordenada.

```

class Rectangulo extends Base
{
    private double x1, y1, x2, y2;
    //(x1,y1) es el vertice inferior izqdo y (x2,y2) el sup. dcho.
    Rectangulo(double x1, double y1, double x2, double y2)
    {
        this.x1=x1; this.y1=y1; this.x2=x2; this.y2=y2;
    }
    public boolean localizada (double x, double y)
    {return x>=x1&& x<=x2&& y>=y1&& y<=y2;}
}

```

En cuanto a la clase `Circulo`, se representa mediante las coordenadas del centro y el valor del radio. Una posición 2D se encuentra localizada dentro de sus límites si la distancia al centro es inferior al valor del radio.

```
class Circulo extends Base
{
    private double x, y, r;
    //(x,y) es el centro y r el radio
    Circulo(double x, double y, double r)
    {
        this.x=x; this.y=y; this.r=r;
    }
    public boolean localizada (double x_p, double y_p)
    { return (x-x_p)*(x-x_p)+(y-y_p)*(y-y_p)<=r*r;}
}
```

En las clases implementadas en este ejercicio, aparece sobrecarga de métodos en cada una de las clases `Elemento`, `Plaza`, `Edificio` y `Calle`. En todas ellas, los métodos `localizada(x,y)` y `localizada(x,y,h)` están sobrecargados. En cuanto a la sobreescritura con polimorfismo, las clases `Plaza`, `Edificio` y `Calle` sobreescriben los métodos abstractos `localizada(x,y)` y `localizada(x,y,h)` de la clase `Elemento`, mientras que las clases `Circulo` y `Rectangulo` sobreescriben el método abstracto `localizada(x,y)` de la clase `Base`. Es importante notar que no hay ninguna relación entre los métodos `localizada(x,y)` de la clase `Elemento` y sus derivadas con los métodos del mismo nombre en la clase `Base` y sus derivadas, pues hay una relación de agregación entre estos conjuntos de clases.

- c) Finalmente, la clase `Mapa` implementa la lista de elementos mediante un array. El constructor da tamaño a ese array en el momento de crear el objeto. Los métodos `situar(x,y)` y `situar(x,y,h)`, únicamente se valen de los métodos `localizada(x,y)` y `localizada(x,y,h)` que obligatoriamente implementarán cada uno de los elementos. Como una posición puede estar a la vez en varios elementos (si sus bases están solapadas), estos métodos se sirven del operador de concatenación de la clase `String` para ir componiendo el resultado final, que contendrá los identificadores de cada uno de los elementos a los que pertenece la posición de test. En caso de no estar en ninguna, devolverá un objeto `String` vacío, en el sentido de que no contiene ningún carácter, aunque sí existe el objeto. Otra posibilidad sería devolver simplemente una referencia nula.

```
class Mapa
{
    private Elemento[] elementos;
    Mapa (Elemento[] elementos_p)
    {
        elementos=new Elemento[elementos_p.length];
    }
}
```

```

        for (int k=0;k<elementos_p.length;k++)
            elementos[k]=elementos_p[k];
    }
    public String situar (double x, double y)
    {
        String resultado="";
        for (int k=0;k<elementos.length;k++)
            if (elementos[k].localizada(x,y))
                resultado+=elementos[k].devNombre()+" ";
        return resultado;
    }

    public String situar (double x, double y, double h)
    {
        String resultado="";
        for (int k=0;k<elementos.length;k++)
            if (elementos[k].localizada(x,y,h))
                resultado+=elementos[k].devNombre()+" ";
        return resultado;
    }
}

```

7.3.3. Desarrollo de un juego basado en personajes

Enunciado Se desea diseñar un juego de rol llamado "La Batalla de las Almas". El juego está formado por una clase principal denominada *Batalla* donde se crean los diferentes participantes del juego y se produce el desarrollo del mismo. El principal objetivo del juego consiste en disputarse el alma de un conjunto de seres humanos, por tanto, existirán diferentes elementos (Ángeles y Demonios) que podrán combatir para tratar de salvar o condenar cada una de las almas puestas en juego. Los diferentes atributos y funciones básicas de cada uno de los elementos que pueden aparecer en el juego son:

Elemento	Descripción	Atributos	Funciones
Ser	Todo en el juego deriva de este ente, no dispone de ningún atributo ni funcionalidad en particular.	—	Ser();
Incorpóreo	Todos aquellos seres que no puedan tener una realidad física en la tierra.	double fe double maldad double bondad	double engañar (Humano h); double gular (Humano h); int luchar (Humano h); double probar (Humano h); double proteger (Humano h); double seducir (Humano h); double tentar (Humano h);

Elemento	Descripción	Atributos	Funciones
Físico	Seres que pueden ser creados en nuestro mundo.	—	—
Espiritual	Interfaz que implementan aquellos seres físicos/incorpóreos y que les permite rezar a Dios.	—	boolean rezar();
Dios	Ser de tipo incorpóreo.	int num numAngeles int num numDemonios	
DiosCristiano	Es un tipo de Dios, en particular el profesado por la religión del mismo nombre.	—	boolean esBuenHombre(Humano h); DiosCristiano(int numAngeles, int numDemonio);
Angeles	Ser incorpóreo y espiritual que se enfrentará a un Demonio para tratar de salvar el alma en juego.	double fe fe : 0..1000 double maldad maldad : 0..1000 double bondad bondad : 0..10	Angeles(); boolean rezar();
Demonios	Ser incorpóreo y espiritual que se enfrentará a un Ángel para tratar de condenar el alma en juego.	double fe fe : 0..1000 double maldad maldad : 0..1000 double bondad bondad : 0..10	Demonios(); boolean rezar();
Humano	Seres físicos y espirituales.	double inteligencia inteligencia : 0..1000 double fe fe : 0..100 double maldad maldad : 0..100 double bondad bondad : 0..100 double alma alma : 0..1000	Humano(); void conflictoMoral(Angeles a, Demonios d); double golpear(Incorporeo s, int habilidad); boolean rezar();

Se pide:

- Diseñar el diagrama de clases que permite representar a todos los elementos involucrados en el juego, indicar las diferentes características de estas clases (abstracta, interfaz, etc.). Incluir en el diseño la clase *Batalla* y su relación con las anteriores.
- Implementar los constructores de Ángeles, Demonios y Humanos, teniendo en cuenta que en el constructor debe generarse aleatoriamente el valor de sus atributos (en el rango indicado).

- c) Implementar el método de rezar para Ángeles, Demonios y Humanos (en los primeros dos casos, simplemente se muestra que el ser habla o no habla con Dios). En el caso de un ser humano, reza si su fe es superior a 50 puntos o su bondad supera los 60.
- d) Implementar el método de lucha para Ángeles y Demonios. Únicamente consistirá en generar aleatoriamente un valor entre 0 y 5 que determinará cual de sus habilidades se aplicará sobre el humano.
- e) Implementar los métodos de combate de Ángeles y Demonios (engañar, guiar, proteger, tentar, seducir, probar) someten a cada Humano a una prueba, pudiendo éste pasarla o no, para ello, se calcula el resultado de someterle a cada una de esas pruebas en función de las características que definen a cada Humano, en la siguiente tabla aparecen las relaciones matemáticas entre las características del Humano y cómo le afectan cada una de las pruebas:

Habilidad	Ángeles	Demonios
engañar	return0;	$\left \frac{fe_{humano}}{fe_{angel}} \right * \left(\frac{ bondad - maldad }{fe_{angel}} \right)$
seducir	return0;	$\sqrt{\frac{ -bondad^2 + maldad^2 - inteligencia }{ fe^2 - bondad^2 }}$
tentar	return0;	$\left \frac{(-fe + maldad) / inteligencia}{\sqrt{ bondad^2 - maldad^2 _{demonio}}} \right $
guiar	$\sqrt{\frac{ bondad^2 - maldad^2 - inteligencia }{ fe^2 - maldad^2 }}$	return0;
probar	$\left \frac{(fe - maldad^2 / inteligencia)}{\sqrt{ bondad^2 - maldad^2 _{angel}}} \right $	return0;
proteger	$\frac{fe_{humano}}{fe_{angel}} * \left(\frac{ bondad - maldad }{fe_{angel}} \right)$	return0;

- f) Implementar los métodos **void conflictomoral(Angeles a, Demonios d);** y **double golpear(Incorporeo s, int Habilidad);** El primer método selecciona una de las habilidades del Ángel o del Demonio mediante el método `luchar()`; una vez hecho esto llama al método `golpear` para el Ángel y para el Demo-

nio, gana el que mayor puntuación obtenga. Si gana el Ángel, la bondad del Humano aumenta y su maldad disminuye. Si gana el Demonio la bondad del Humano disminuye y aumenta su maldad. El método `golpear` selecciona la habilidad que corresponde al método `luchar` (0=engañar, 1=guiar,... 5=proteger). Cuando es aplicada la habilidad correspondiente, el Humano tratará de rezar, si reza su fe aumenta en un punto, en caso contrario disminuye en la misma cantidad.

- g) Implementar el método `main()` de la clase `Batalla.java`, donde se debe leer como argumentos, el número de Humanos, el número de Ángeles, el número de Demonios y finalmente el número de turnos. Ejemplo: **java Batalla 100 5 5 30**.

En este método se construirán los diferentes elementos, así como un objeto de tipo `DiosCristiano`, y se implementará la estructura que permita ejecutar los diferentes turnos donde para cada hombre (y en cada turno) se le someterá a un conflicto moral, en función del cual su bondad o maldad aumentará o disminuirá.

- Solución** a) La clase `Batalla` se compone de una serie de objetos de la clase `Humano`, `Angel` y `Demonio`, así como de un objeto de tipo `DiosCristiano`. La clase `Humano` deriva de la clase `Fisico`, mientras `Angel` y `Demonio` derivan de `Incorporeo`. De esta clase deriva también la clase abstracta `Dios`, de la cual deriva la clase final `DiosCristiano`. Las

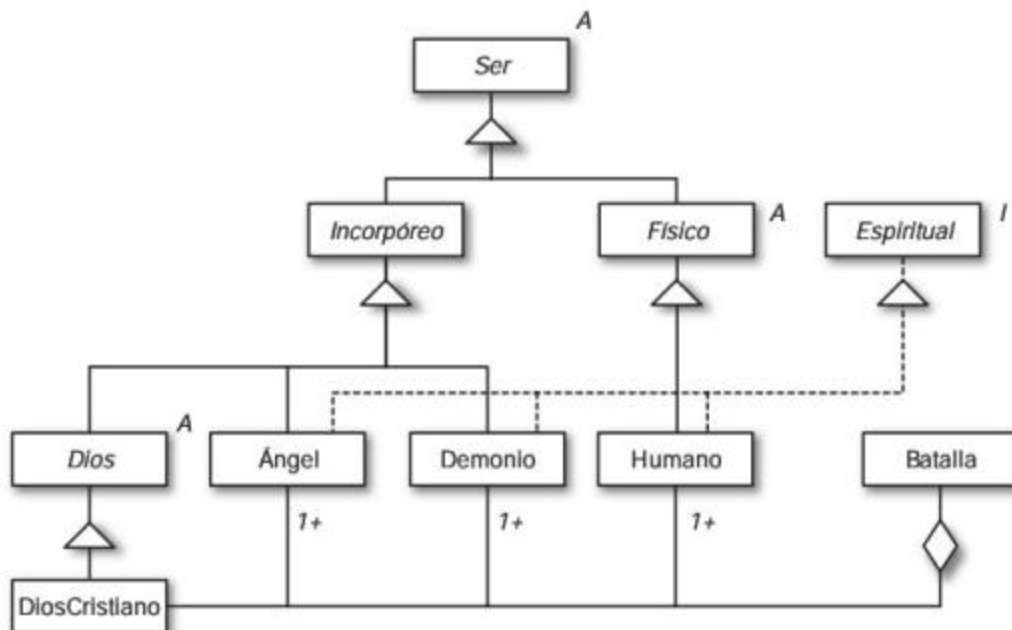


Figura 7.17. Diagrama de clases correspondientes al Ejercicio 7.3.3.

clases abstractas *Incorporeo* y *Fisico* derivan de la clase abstracta *Ser*, y, por último, las clases *Angel*, *Demonio* y *Humano* implementan la interfaz *Espiritual*:

- b)** Las clases *Angel*, *Demonio* y *Humano* construyen sus objetos con valores aleatorios de los atributos, según los intervalos pedidos, utilizando la función de generador aleatorio en la clase utilidad *Math*.

```
public Angel() { //construyendo un Angel
    this.fe = Math.random() * 1000; //creo en Dios
    this.bondad = Math.random() * 1000;
    this.maldad = Math.random() * 10;
}

public Demonio() { //construyendo un Demonio
    this.fe = Math.random() * 1000; //creo en el Angel Caído
    this.bondad = Math.random() * 10;
    this.maldad = Math.random() * 1000;
}

public Humano() { //constructor de humanos
    //no todas las almas valen lo mismo
    alma = Math.random()*1000;
    inteligencia = Math.random()*1000;
    fe = Math.random()*100;
    bondad = Math.random()*100;
    maldad = Math.random()*100;
}
```

- c)** El método *rezar* devuelve valor falso para el *Demonio*, cierto para el *Ángel*, y un valor dependiente de los atributos de *fe* y *bondad* para los *Humanos*:

```
// ANGELES
public boolean rezar() {
    System.out.println("soy un angel y rezo a Dios pidiendo ayuda");
    return true;
}

// DEMONIOS
public boolean rezar() {
    System.out.println("soy un demonio y no rezo");
    return false;
}

// seres HUMANOS
public boolean rezar() {
    // el humano solo reza a veces.....
    // si reza Dios le ayuda, en caso contrario no....
    if (this.fe >= 50 || this.bondad >= 60)
```



```

        return true;
    else
        return false;
}

```

- d) El método `luchar`, especificado en la clase abstracta `Incorporeo`, se implementa en las clases `Angel` y `Demonio` según lo especificado.

```

public int luchar(Humano h) {
    // el angel decide cual de sus habilidades aplicar....
    // de forma Aleatoria
    int habilidad = (int)(Math.random()*6);
    if (habilidad == 6)
        habilidad = 5;
    return habilidad;
}

public int luchar(Humano h) {
    // el demonio decide cual de sus habilidades aplicar....
    // de forma Aleatoria
    int habilidad = (int)(Math.random()*6);
    if (habilidad == 6)
        habilidad = 5;
    return habilidad;
}

```

Sin embargo, obsérvese que, dado que no hay ninguna diferencia entre ambas implementaciones, sería una solución más elegante incluir esta implementación directamente en la clase padre de ambas, en `Incorporeo`.

- e) Los métodos pedidos se implementan a continuación, sin más que codificar las distintas funciones matemáticas especificadas. Obsérvese que en este caso sí hay diferencias de implementación de los métodos especificados en la clase abstracta, siendo un ejemplo de utilización de polimorfismo.

```

// ANGELES
public double engañar(Humano h) {
    //los angeles no engañan
    return 0;
}

public double guiar(Humano h) {
    double valorH, valorA;
    valorH = Math.sqrt( Math.abs(h.dameBondad() * h.dameBondad() -
    h.dameMaldad()*h.dameMaldad()-h.dameInteligencia()));
    valorA = Math.sqrt(Math.abs(this.dameFe() * this.dameFe()-

```

```

        this.dameMaldad() * this.dameMaldad());
        return valorH/valorA;
    }
    public double probar(Humano h) {
        double valorH, valorA;
        double valorD;
        valorD = Math.sqrt( Math.abs( this.dameBondad() *
        this.dameBondad()-this.dameMaldad() * this.dameMaldad()));
        valorH = (h.dameFe()-h.dameMaldad())/h.dameInteligencia();
        return Math.abs(valorH/valorD);
    }
    public double proteger(Humano h) {
        double valorFe, valorH;
        valorFe = Math.abs(h.dameFe()/this.fe);
        valorH = Math.abs(h.dameBondad()-h.dameMaldad());
        return valorFe * (valorH/this.fe);
    }
    public double seducir(Humano h) { //un angel no seduce a los humanos
        return 0;
    }
    public double tentar(Humano h) { // un angel no tienta a los humanos
        return 0;
    }
    // DEMONIOS
    public double probar(Humano h) { // un demonio no prueba al humano
        return 0;
    }
    public double proteger(Humano h) { //un demonio no protege
        return 0;
    }
    public double seducir(Humano h) {
        double valorH, valorA;
        valorH = Math.sqrt(Math.abs(-h.dameBondad() *
        h.dameBondad()+h.dameMaldad() *
        h.dameMaldad()-h.dameInteligencia()));
        valorA = Math.sqrt(Math.abs( this.dameFe() *
        this.dameFe()-this.dameBondad() * this.dameBondad()));
        return valorH/valorA;
    }
    public double tentar(Humano h) {
        double valorH, valorA;
        double valorD;
        valorD = Math.sqrt( Math.abs( this.dameBondad() *
        this.dameBondad()-this.dameMaldad() * this.dameMaldad()));

```

```

        valorH = (-h.dameFe()+h.dameMaldad())/h.dameInteligencia();
        return Math.abs(valorH/valorD);
    }
    public double engañar(Humano h) {
        valorFe = Math.abs(h.dameFe()/this.fe);
        valorH = Math.abs(-h.dameBondad()+h.dameMaldad());
        return valorFe * (valorH/this.fe);
    }
    public double guiar(Humano h) { // los Demonios no guian.....
        return 0;
    }
}

```

- El método `conflictoMoral` del `Humano` simplemente invoca a los métodos *luchar* de los objetos argumentos (`Angel` y `Demonio`), a los que envía a su vez como argumento una referencia a su propia instancia, `this`. A continuación, compara las puntuaciones y aplica la lógica pedida para modificar sus atributos.

```

public void conflictoMoral(Angeles a, Demonios d) {
    //se introduce un angel y un demonio.....
    //el resultado de la batalla hace incrementar o disminuir las
    // características: bondad y maldad
    double golpeAngel=0;
    double golpeDemonio=0;
    int HabilidadAngel, HabilidadDemonio;

    //selecciono un metodo de lucha
    HabilidadAngel = a.luchar(this);
    HabilidadDemonio = d.luchar(this);

    //el angel y el demonio lo aplican sobre el humano
    golpeAngel = this.golpear(a,HabilidadAngel);
    golpeDemonio = this.golpear(d,HabilidadDemonio);

    //ganador
    if (golpeAngel >= golpeDemonio){
        this.bondad = this.bondad + (golpeAngel/golpeDemonio);
        this.maldad = this.maldad - (golpeAngel/golpeDemonio);
    }else{
        this.bondad = this.bondad - (golpeAngel/golpeDemonio);
        this.maldad = this.maldad + (golpeAngel/golpeDemonio);
    }
}
}

```

El método `golpear` del `Humano` selecciona la habilidad correspondiente al objeto `Incorporeo` (obsérvese que no necesita saberse el tipo específico en este caso, simplemente llama a sus métodos abstractos utilizando el polimorfismo).

```
public double golpear(Incorporeo s, int hab) {
    // se ejecuta la habilidad del correspondiente angel o demonio
    double valor=0;
    switch (hab){
        case 0:
            valor = s.enganar(this);
        case 1:
            valor = s.guiar(this);
        case 2:
            valor = s.probar(this);
        case 3:
            valor = s.proteger(this);
        case 4:
            valor = s.seducir(this);
        case 5:
            valor = s.tentar(this);
    }
    if (this.rezar())
        this.fe = this.fe+1;
    else
        this.fe = this.fe-1;
    return valor;
}
```

- g Por último, la clase `Batalla` implementa el método `main` para definir el escenario de juego y efectuar la partida. Se crean primero los objetos que intervienen como elementos del juego: un `DiosCristiano` y arrays de objetos de tipo `Humano`, `Angel` y `Demonio`, con tantos elementos como especificados por línea de argumento (para lo que se usan los métodos de extracción de datos de la clase de utilidad `Integer`). A continuación, cada turno de la batalla consiste en seleccionar los Ángeles y Demonios que afectarán a todos los Humanos que intervengan (mediante bucles `for`). Finalmente, se presentan los resultados del juego:

```
public static void main(String[] args) throws IOException{
    //argumentos: numero de humanos, angeles y demonios
    int numHumanos = Integer.parseInt(args[0]);
    int numAngeles = Integer.parseInt(args[1]);
    int numDemonios = Integer.parseInt(args[2]);

    //numero de turnos
    int numTurnos = Integer.parseInt(args[3]);
    System.out.println("Turnos de Batalla: "+numTurnos);
}
```



```
// DIOS
DiosCristiano unico = new DiosCristiano(numAngeles,numDemonios);

// HUMANOS
Humano[] humanidad = new Humano[numHumanos];
crearHumanos(humanidad,numHumanos);

//ANGELES Y DEMONIOS
Angeles[] angeles = new Angeles[unico.numeroAngeles];
Demonios[] demonios = new Demonios[unico.numeroDemonios];

crearAngeles(angeles,unico.numeroAngeles);
crearDemonios(demonios,unico.numeroDemonios);
int luchadorAngel, luchadorDemonio;

// BATALLA
for (int i=0; i < numTurnos; i++){
    //escogo un Angel y un demonio
    luchadorAngel = selecciona(angeles);
    luchadorDemonio = selecciona(demonios);
    // LUCHAR
    for (int j=0; j < humanidad.length; j++){

        humanidad[j].conflictoMoral(angeles[luchadorAngel],
                                    demonios[luchadorDemonio]);
    }
}
//RESULTADO
visualizarResultados(unico,humanidad);
}
```

7.3.4. Desarrollo de una aplicación de simulación

Enunciado

En este ejercicio se propone la simulación de un sistema de colas genérico y la estimación de sus parámetros estadísticos. Este sistema estará compuesto por una cola de usuarios en espera, de longitud arbitraria, y un servidor, como se indica en la Figura 7.18. Su funcionamiento es el siguiente: cada vez que llega un usuario al sistema, si el servidor está desocupado, el servidor comienza a ser atendido; en cambio, si existen otros usuarios en el sistema, el recién llegado se incorpora a la cola y permanecerá esperando hasta que todos los que llegaron antes que él hayan sido servidos (la cola sigue, por tanto, una disciplina conocida como *FIFO: First In First Out*). En todo caso, el tiempo que está siendo servido un usuario, una vez que ha llegado al servidor, es un valor aleatorio y uniformemente distribuido dentro del intervalo (T_{s1}, T_{s2}) . En cuanto a la llegada de usuarios, se simulará suponiendo que el intervalo de tiempo que pasa entre dos llega-



Figura 7.18. Sistema de colas de un servidor.

das sucesivas es también aleatorio y uniforme en el intervalo $(0, T_u)$, de tal manera que en promedio llega un usuario cada $T_u/2$ unidades de tiempo.

El objeto de la simulación será determinar el tiempo medio de espera en cola que permanece un usuario, así como el porcentaje de tiempo de ocupación del servidor, en función de los parámetros de entrada T_{s1} , T_{s2} y T_u . La simulación se llevará a cabo mediante eventos discretos, definiéndose dos tipos de eventos posibles: llegadas al sistema y salidas del sistema, cuya generación dependerá de la situación en que se encuentra el sistema (longitud de la cola) cada vez que llega o sale un usuario. Los objetos que se implementarán para llevar a cabo esta simulación son los siguientes:

Usuario: este objeto únicamente precisa como atributo el tiempo de llegada al sistema, que se utilizará para calcular después cuánto tiempo ha permanecido en cola.

Sistema: este objeto es el que implementará el sistema simulado, conteniendo la cola de usuarios y las variables de estado necesarias para estimar los estadísticos pedidos. Contendrá los siguientes elementos:

- Una lista de objetos de tipo *Usuario*, existiendo métodos para añadir y extraer usuarios del sistema.
- Se dispondrá de una variable con el tiempo total que el sistema ha estado ocupado, así como una variable para determinar el instante de última activación. Ambas variables se actualizarán con métodos para "activar" y "desactivar" el servidor.
- Para estimar el tiempo medio que un usuario espera en cola, hay que sumar los tiempos que esperan todos los usuarios y después dividir por el número total de usuarios que pasaron por el sistema. Para ello se definirán sendas variables en el sistema que representen el número de usuarios que han pasado por el sistema y la suma de tiempos totales de espera, que se actualizarán según avance la simulación.

Evento: cada evento, que puede ser de tipo "Llegada" o "Salida", es un objeto que almacena su tiempo de futura atención, y a su vez implementa dos métodos que se utilizarán para realizar la simulación. En primer lugar, un método de modificación del sistema *modificarSistema* (*Sistema s*), que cambiará el sistema de acuerdo con el tipo de evento concreto (llegada o salida), así como del estado actual del sistema, del modo indicado en la tabla siguiente:

	Modificar sistema
Llegada	— Crear un nuevo usuario y añadirlo al sistema. — Si el sistema estaba inactivo, (no hay otros usuarios en el instante de la llegada), se modifican las variables con el estado de ocupación del sistema y el instante de última activación.
Salida	— Extraer el usuario más antiguo del sistema. — Actualizar las variables para medir el tiempo medio de espera. — Si no hay otro usuario a continuación del que sale, el sistema se queda inactivo, por lo que se modifican las variables con el estado de activación y el tiempo total activo.

GeneradorEventos este objeto contiene el tiempo de simulación actual, los parámetros de generación de llegadas y salidas (T_{s1} , T_{s2} y T_u) y los eventos pendientes de atención: la siguiente llegada y la siguiente salida del sistema. Como interfaz pública, contendrá un método de activación de evento, `activarSiguiente`, que selecciona el evento siguiente a procesar, modifica el sistema con él, y a continuación, determina los eventos que aparecen a continuación, dependiendo del estado concreto del sistema y del tipo de evento, tal y como se indica en la tabla siguiente:

	Siguientes eventos
Llegada	— La siguiente llegada aparecerá en un instante aleatorio posterior a la actual entre 0 y T_u unidades. En caso de que este instante sea superior al de simulación, no se genera. — Si ningún usuario está en el sistema en el momento de llegada, se generará el siguiente evento de salida, correspondiente al usuario que acaba de llegar y no ha tenido que esperar.
Salida	— En el caso de haber más usuarios pendientes en el sistema en el momento de la salida, se genera la salida del siguiente usuario, que aparecerá una vez transcurrido su tiempo de servicio, entre T_{s1} y T_{s2} unidades después del instante actual.

Solución

En la Figura 7.19, se indica el diagrama de clases del sistema implementado. El generador de eventos contiene un objeto de tipo `Llegada` y otro de tipo `Salida`, y a su vez tiene una referencia al objeto `Sistema` para afectarle a través de los eventos simulados.

La clase `Usuario` únicamente tiene como atributo privado el tiempo de llegada, que se asigna en el momento de la creación, y un método para acceder a su valor:

```
class Usuario
{
    private double tiempoLlegada;
    public Usuario(double t)
    {
        tiempoLlegada=t;
    }
}
```

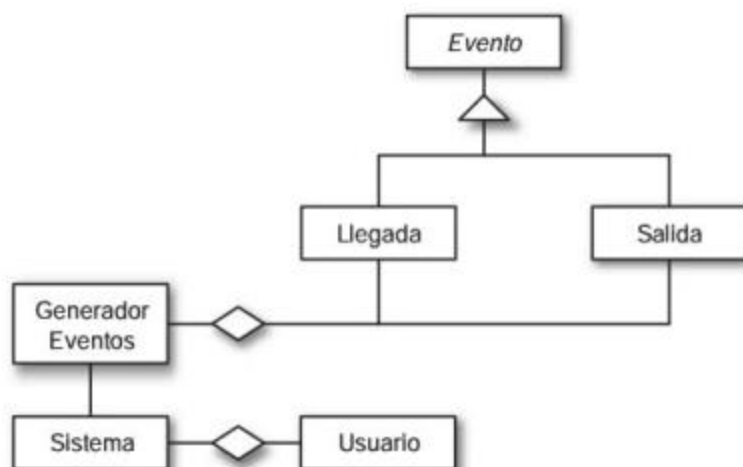



Figura 7.19. Diagrama de clases correspondiente al Ejercicio 7.3.4.

```

public double daTiempo()
{
    return tiempoLlegada;
}

```

La clase *Sistema* aparece a continuación. La lista de usuarios se implementa mediante un array. El usuario más antiguo será el situado en la posición 0 del array y el más reciente ocupará el índice mayor. Obsérvese que el método de meter usuario simplemente crea uno nuevo y lo añade por la derecha, mientras que para extraer se precisa mover un paso a la izquierda a todos los usuarios. Para actualizar la ocupación, basta con ir sumando los intervalos de tiempo durante los que éste está ocupado, lo que se realiza con el método *desactivar*, que calcula el tiempo de ocupación desde la última activación y lo añade al acumulador. Cada vez que cambie el estado de ocupación (pasa a desocupado el servidor cuando hay 0 usuarios en la lista y viceversa), se actualizará esta variable y el acumulador de tiempo de ocupación. En cuanto al tiempo medio de espera en cola, obsérvese que cada vez que un usuario va a salir del sistema (evento de salida), es cuando puede calcularse el tiempo que ha pasado en espera el usuario en la posición siguiente: es la diferencia entre el instante actual y el instante en el que llegó. En caso particular de que al salir un usuario del sistema no haya ningún otro esperando, no se acumulará nada, lo que equivale a decir que el siguiente usuario que llegue experimentará un tiempo de espera en cola de valor 0. Por tanto, cada vez que se saca un usuario se incrementa el acumulador con la diferencia entre el instante actual y el de llegada de ese usuario, y al final se dividirá el valor total acumulado por el número de usuarios atendidos.

```

class Sistema
{
    int numeroUsuarios=0;
    Usuario [] usuariosPendientes=new Usuario[100];
}

```



```
private double tiempoSimulacion=0.0;
    private double tiempoCambioActivo=0.0;
    private double tiempoOcupacion=0.0;

    private double tiempoMedioEnCola=0.0;
    private int usuariosAtendidos=0;

    public int numeroUsuarios()
    {return numeroUsuarios;}

    public void cambiaTiempoSimulacion(double t)
    {tiempoSimulacion=t;}

    public double daTiempoSimulacion()
    {return tiempoSimulacion;}

    public double daTiempoOcupacion()
    {return tiempoOcupacion;}

    public double daTiempoMedioEnCola()
    {return tiempoMedioEnCola/usuariosAtendidos;}

    public double daUsuariosAtendidos()
    {return usuariosAtendidos;}

    public void meterUsuario(double t)
    {
        usuariosPendientes[numeroUsuarios++]=new Usuario(t);
    }

    public void sacarUsuario(double t)
    {
        // sale el tiempo del usuario en posicion 0
        numeroUsuarios--;
        usuariosAtendidos++;
        // todas las posiciones se desplazan a la izquierda uno
        for (int i=0;i<numeroUsuarios;i++)
        {
            usuariosPendientes[i]=usuariosPendientes [i+1];
        }
        //actualizacion de variable de tiempo medio de espera
        //en cola
    }
```

```

        if (numeroUsuarios>0)
        {
            tiempoMedioEnCola+=t-usuariosPendientes[0].daTiempo();
        }
    }
    public void activar (double t)
    {
        tiempoCambioActivo=t;
    }
    public void desactivar (double t)
    {
        tiempoOcupacion+=t-tiempoCambioActivo;
    }
}

```

Las clases Evento, Llegada, Salida están a continuación. El método abstracto `modificarSistema` se implementa por cada clase derivada e invoca las tareas de meter o sacar usuarios en el sistema, que a su vez actualizarán sus valores que permiten determinar los parámetros estadísticos analizados.

```

abstract class Evento
{
    private double tiempo;

    Evento(double t)
    {tiempo=t;}

    public double daTiempo()
    {return tiempo;}

    abstract public void modificarSistema(Sistema s);
}

class Llegada extends Evento
{
    Llegada(double t)
    {super(t);}

    public void modificarSistema(Sistema s)
    {
        s.cambiaTiempoSimulacion(this.daTiempo());
        if (s.numeroUsuarios()==0)
        {

```

```

        s.activar(this.daTiempo());
    }
    s.meterUsuario(this.daTiempo());
}
}

class Salida extends Evento
{
    Salida(double t)
    {super(t);}

    public void modificarSistema(Sistema s)
    {
        s.cambiaTiempoSimulacion(this.daTiempo());
        s.sacarUsuario(this.daTiempo());
        if (s.numeroUsuarios()==0)
        {
            s.desactivar(this.daTiempo());
        }
    }
}

```

La clase `GeneradorEventos` implementa los dos eventos de interés con dos referencias a objetos de tipo evento, que tendrán un valor nulo en caso de no existir alguno (por ejemplo, se conoce el próximo instante de llegada de usuario al sistema pero no el de salida, por estar éste vacío). El método de activación en primer lugar determina qué evento se debe activar: si existen los dos, el de tiempo menor, y si no, el que esté pendiente. Es un error activar un evento si no hay más pendientes, para lo cual deberá invocarse antes al método que comprueba si hay eventos pendientes de activación o no, `masEventos()`. A continuación, se generan los siguientes eventos en función del tipo y del estado del sistema, tal y como se ha especificado. En caso de que no se genere la llegada o la salida siguiente, se ponen las referencias correspondientes a `null`. El método `arranca()` se utiliza para inicializar, creando la primera llegada al sistema.

```

class GeneradorEventos
{
    Evento siguienteLlegada=null, siguienteSalida=null;
    double tS1, tS2, tU, tMaximo;
    Sistema sistema;
    boolean masEventos=false;

    GeneradorEventos (Sistema s, double tMaximo, double tS1, double tS2, double tU)
    {
        this.tMaximo=tMaximo;
    }
}

```

```

        this.tS1=tS1;
        this.tS2=tS2;
        this.tU=tU;
        this.sistema=s;
    }

    public boolean masEventos()
    {return siguienteLlegada!=null||siguienteSalida!=null;}

    public void arranca()
    {
        siguienteLlegada=new Llegada(Math.random()*tU);
    }

    public void activarSiguiente()
    {
        boolean tocaSalida=false;
        if (siguienteLlegada==null&&siguienteSalida!=null)
            tocaSalida=true;
        else if (siguienteLlegada!=null&&siguienteSalida==null)
            tocaSalida=false;
        else if (siguienteLlegada!=null&&siguienteSalida!=null)
        {
            if (siguienteSalida.daTiempo()<=siguienteLlegada.daTiempo())
                tocaSalida=true;
            else
                tocaSalida=false;
        }
        else
        {
            System.out.println("\fError: activar evento inexistente!");
        }
    }

    Evento siguiente;

    if (tocaSalida)
    {
        siguiente=siguienteSalida;
        siguiente.modificarSistema(sistema);
        // si quedan usuarios, generar la siguiente salida
        if (sistema.numeroUsuarios()>0)

```



```

        {
            siguienteSalida=new Salida(
siguiente.daTiempo()+tS1+Math.random()*(tS2-tS1) );
        }
        else
            siguienteSalida=null;
    }
else
    {
        siguiente=siguienteLlegada;
        siguiente.modificarSistema(sistema);

        // si el tiempo del siguiente es mayor que el total,
        // no hay mas llegadas
        double tproxima=siguiente.daTiempo()+Math.random()*tU;

        if (tproxima<tMaximo)
        {
            siguienteLlegada=new Llegada(tproxima);
        }
        else
            siguienteLlegada=null;

        // si no habia usuarios, puede generar la siguiente salida
        if (sistema.numeroUsuarios()==1)
        {
            siguienteSalida=new Salida(
                siguiente.daTiempo()+tS1+Math.random()*(tS2-tS1));
        }
    }
}
}

```

Finalmente, la clase principal lee de la consola los parámetros de simulación, inicializa el sistema y el generador y arranca la simulación, activando eventos mientras haya lugar. Por último, muestra los resultados en pantalla:

```

public class SistemaColas{
    public static void main (String args[]) throws IOException
    {
        BufferedReader br=new BufferedReader(new
            InputStreamReader(System.in));
    }
}

```

```

        System.out.println("Introducir tiempo total: ");
        String  valor=br.readLine();
        double  tMax=Double.parseDouble(valor);

        System.out.println("Introducir tiempo max entre llegadas: ");
        valor=br.readLine();
        double  tUsuario=Double.parseDouble(valor);

        System.out.println("Introducir tiempo minimo servicio: ");
        valor=br.readLine();
        double  tServ1=Double.parseDouble(valor);

        System.out.println("Introducir tiempo maximo servicio: ");
        valor=br.readLine();
        double  tServ2=Double.parseDouble(valor);

        Sistema miSistema = new Sistema();
        GeneradorEventos ge=new GeneradorEventos(miSistema, tMax, tServ1,
            tServ2, tUsuario);
        ge.arranca();

        do {
            ge.activarSiguiente();
        }while (ge.masEventos());

        System.out.println("usuarios atendidos: "+miSistema.daUsuariosAtendidos());
        System.out.println("tiempo medio espera: "+miSistema.daTiempoMedioEnCola());
        System.out.println("porcentaje ocupacion="+100*(miSistema.daTiempoOcupacion()/
            miSistema.daTiempoSimulacion()) );
    }
}

```

Referencias bibliográficas

- Arnow, D.; Weiss, G., *Introducción a la programación con Java™*, Pearson Educación, S.A. (Addison Wesley), Madrid, 2000.
- Brassard, G., *et al.*, *Algorítmica, concepción y análisis*, Ed. Masson, 1990.
- Muñoz Caro, C.; Niño Ramos, A.; Vizcaíno Barceló, A., *Introducción a la programación con orientación a objetos*, Pearson Educación, S.A., Madrid, 2002.
- Eckel, B., *Piensa en Java*, Pearson Educación, S.A., Madrid, 2002.
- Horstmann, C., *Big Java*, John Wiley & Sons, 2002.
- Hoare, C. A. R., "Quicksort", *Comp. J.*, 5, núm. 1, pp. 10-15, 1962.
- Morgan, M., *Descubre Java™ 1.2*, Prentice Hall, 1998.
- Naughton, P.; Schildt, H., *Java. Manual de referencia*, Mc-Graw Hill-Interamericana de España, S. A. U., 1997.
- Williams, J. W. J., Heapsort (Algoritmo 232), *Comm. ACM*, 7, núm. 6, pp. 347-348, 1964.
- Wirth, N., *Algoritmos + estructuras de datos = Programas*, Ediciones del Castillo, Madrid, 1980.
- Wu, C. T., *Introducción a la programación orientada a objetos con Java*, Mc-Graw Hill-Interamericana de España, S. A. U., 2001.

Índice analítico

A

- abstracción, 262, 265
- abstract, 283, 340, 358
- acceso, 264, 267, 268
 - aleatorio, 193
 - secuencial, 193
- amistoso (*friendly*), 268
- análisis, 360, 362
- Applet, 83, 86, 375
- Appletviewer, 83, 377
- árbol jerárquico, 352
- archivo, 191
- Array, 17, 18, 20, 22, 47, 49
 - de una dimensión, 18
 - polimórfico, 344
- Arrays
 - de objetos, 273, 309
 - multidimensionales, 22
 - no ordenados, 104
 - ordenados, 105
- Atributos, 39, 41, 261, 264, 265

B

- Backtracking, 167
- boolean, 10, 11
- break, 32, 37, 53, 54
- Bubble Sort, 106
- BufferedInputStream, 197, 228, 233
- BufferedOutputStream, 197, 233
- BufferedReader, 196, 207, 226
- BufferedWriter, 196
- Burbuja, 106, 125, 126, 127
- Búsqueda, 101, 102, 112
 - binaria, 103, 114, 116, 117, 133
 - secuencial, 102, 112
- byte, 10, 14, 16
- bytecode, 3, 5, 6

C

- C++, 1, 4, 30
- Caso base, 162, 163, 164
- Casting, 13
- catch, 64, 66, 69, 217
- char, 10, 11, 14, 16
- clase
 - abstracta, 283, 339, 340, 356
 - base, 281, 283
 - derivada, 281, 283
 - interfaz, 273, 358, 363
- Clases, 261, 265, 291
 - instanciables, 356, 358
- class, 266, 291
- cola, 273, 274
- commands, 264
- Comparaciones, 106, 270, 306, 307
- Compilación, 3, 12, 14, 18
- Compilador javac, 5, 271, 307
- Complejidad, 102, 103
- Constantes, 7, 11, 26, 33
 - de clase
 - de objeto, 271, 307
- constructor, 268, 283, 300, 302
- continue, 38
- Control de flujo
 - bucles, 34
 - sentencias condicionales, 31
- Conversión
 - automática, 14
 - de tipos, 13
 - con herencia, 277, 366
 - explícita, 15, 277, 366
- criba, 110
- Cubos, 17, 20, 22, 47, 49

D

- DataInputStream, 197, 204, 227
- DataOutputStream, 197, 204, 228

- diagrama
 - de agregación, 354, 355
 - de herencia, 256, 257, 352
 - de relaciones, 354
- dibujo de figuras, ejemplo, 363
- diseño
 - de jerarquías de clases, ejemplo, 363, 379, 384
 - orientado a objetos, 359
- double, 10, 11, 16
- do-while, 37
- drawArc, 371, 375
- drawLine, 369, 375
- drawOval, 367, 375
- drawPolygon, 374, 375

E

- E/S, 38, 201
- Eficiencia, 103, 111
- encapsulamiento, 262
- Entorno
 - de desarrollo, 2, 3, 4
 - de ejecución, 5, 39
- Entrada/salida, 38, 201
- Entrada estándar, 40
- Envoltorios, 9, 10, 44
- eof (*end of file*), 194, 209
- Error, 60
- especificaciones, 360
- especificador, 267, 285, 286
- eventos de simulación, ejemplo, 393, 395
- excepción, 59
- excepciones propias, 77
- Exception, 60, 64
- extends, 283, 332

F

- Factorial, 162
- Fibonacci, 165
- fichero html, 95
- ficheros
 - binarios, 196
 - de texto, 196, 207
- figuras geométricas, ejemplo, 363
- File, 191, 197
- FileInputStream, 196, 202, 230

- FileOutputStream, 196, 202, 230
- FileReader, 197, 226
- FileWriter, 197, 227
- final, 12, 271, 272, 286
- finally, 64, 71
- float, 38, 39
- Flujos, 38, 39, 200, 201, 203
- for, 35, 37

G

- Graphics (clase), 366, 375

H

- Heap Sort, 108, 143, 147, 148, 149
- herencia, 279, 281, 331, 352
 - múltiple, 272, 357
- URLConnection, 97

I

- if-else, 31, 32
- implements, 358, 364
- InetAddress, 92
- InputStream, 200
- InputStreamReader, 197
- Inserción, 104, 105, 119, 121, 122, 124
 - directa, 107, 130, 131, 133
- instanceOf, 354
- instancia, 266, 268, 301
- int, 7, 10, 14, 16
- interface, 272, 356, 363
- interfaces
 - gráficas, 79, 82
 - vs herencia, 358, 364
- Internet, 2, 3
- IOException, 64, 217

J

- J2SE, 3
- Java 1.0, 2
- Java 1.1, 2
- Java 2.2, 2, 3, 10
- java.net, 91

jerarquía, 351, 363
JRE, 5, 39
juego de personajes, ejemplo, 384
JVM, 3, 4, 5, 6

L

Lenguaje de programación, 1, 6
lista de objetos, 273, 311, 316
listas
 doblemente enlazadas, 277
 simples, 277, 322
Literales, 11
long, 10, 11, 16

M

mapas (representación y localización), ejemplo, 378
Máquina virtual Java, 3, 4, 5, 6
Matriz, 17, 20, 22, 47, 49
memoria dinámica, 276, 316
mensajes a objetos, 262, 264
Métodos, 5, 10, 39, 264, 270, 288
 avanzados, 107, 136, 268
 de acceso, 268
 de clase, 271, 308
 directos, 106, 125
 heredados, 285, 338
mezcla
 directa de ficheros, 221, 234
 natural de ficheros, 223, 253
miembros heredados
Montículo, 108, 110
Movimientos, 106
multiplicidad en relaciones, 354

N

new, 18, 266, 268, 273, 287
Notación asintótica, 102
números aleatorios, 302, 304, 388, 395

O

ObjectInputStream, 197, 216, 229
ObjectOutputStream, 197, 216, 229

objeto
 antepasado, 353
 descendiente, 353
Objetos, 9, 13, 16, 42, 265, 343, 359
 agregados, 353, 298
Ocho reinas, 180, 181
Operadores
 A nivel de bit, 27
 Aritméticos, 25
 Asignación, 29
 Lógicos, 26
 Precedencia, 30
 Relacionales, 26
 Ternario, 29
orden de llamada en constructores derivados,
 284
Ordenación, 105, 106, 125, 136
 externa, 106, 221
 interna, 105, 221
 rápida, 107, 136, 138
OutputStream, 200
OutputStreamWriter, 197

P

Partición, 108, 136
Paso de parámetros
 Por referencia, 43, 44
 Por valor, 43
Peor caso, 103
persistencia, 212
pila, 166, 273, 313
Pila, 313
Pivote, 108
polimorfismo, 281, 286, 343
PrintStream, 196
PrintWriter, 196, 207, 227
private, 268, 294, 295
procedimental, 280, 359
Programación orientada a objetos, 6, 13, 279,
 359
Promoción en expresiones, 16
protected, 285, 333
pruebas de programa, 360, 362
public, 4, 267, 268
puntero lectura/escritura, 194

Q

queries, 264
Quick Sort, 107, 136, 137, 138

R

RandomAccessFile, 209, 231
Reales, 6, 10, 11, 41, 43
Recursividad, 161
redefinición, 297
referencias, 267, 273, 281
Reglas
 de compatibilidad
 de conversión, 15
 de promoción, 16
relaciones
 de agregación, 353
 de herencia, 354
 jerárquicas, 351
 semánticas, 351
reutilización
 de clases, 279, 362
 de código, 263, 362

S

salida
 de errores, 39
 estándar, 39, 226
Salto del caballo, 176
SDK (Software Development Kit), 2, 3, 4
Selección directa, 107, 133, 134
serializable (interfaz), 212, 230
serialización, 212
short, 7, 10, 14, 16
signatura, 264
simulación, ejemplo, 393
Sistemas operativos
 Linux, 3, 5, 6, 55
 Solaris, 3, 5, 6
 Windows, 3, 5, 6
sobrecarga, 269, 305
sobrescribir métodos, 285
socket, 91

static, 271, 307
streams, 38, 39, 200, 201, 203
String, 5, 11, 16, 17
StringTokenizer
 super, 284, 286
switch, 31, 32, 33, 37

T

Tablas, 17, 20, 22, 47, 49
this, 270, 300, 301
Throwable, 60
Tipos
 básicos, 6, 9, 17, 43
 compatibles, 14, 33
 de datos
 Caracteres, 7, 8
 Coma flotante, 6, 7
 Enteros, 6, 7
 Lógicos, 7
Torres de Hanoi, 173
throw, 73
throws, 75, 217
try, 64, 69, 217

U

URL, 93
URLConnection, 95

V

variables
 Ámbito, 12, 13, 35, 43
 Declaración, 12
 globales, 271
 Tiempo de vida, 12, 13, 35
Versiones de Java, 3
void, 10
Vuelta atrás, 167

W

while, 34, 35, 37
World Wide Web, 2



Programación, Algoritmos y Ejercicios Resueltos en JAVA

Camacho • Valls • García • Molina • Bueno

Este libro está orientado a aquellas personas que están comenzando en el mundo de la programación, o a aquellas personas que, disponiendo de conocimientos de otros lenguajes de programación, desean dar el salto a la programación en Java™.

El libro consta de siete capítulos, en cada uno de los cuales puede encontrarse un conjunto de ejercicios que varían en su grado de complejidad:

- **Ejercicios simples**, utilizados para mostrar características que se consideran esenciales.
- **Ejercicios propuestos**, se encuentran al final de cada capítulo y permitirán aplicar de forma práctica los conceptos teóricos estudiados en dicho capítulo.
- **Propuestas de prácticas**, conjunto de ejercicios que pueden ser utilizados como ejercicios de laboratorio.
- **Ejercicios de examen**, conjunto de problemas que han sido utilizados en exámenes de diferentes asignaturas de programación.

El objetivo principal de este libro es enseñar a programar mediante el uso de ejercicios prácticos resueltos y que pueden ser consultados *a posteriori* por el lector. Presenta una visión aplicada de las principales técnicas de programación, desde las principales técnicas algorítmicas clásicas, hasta los diseños basados en jerarquías de clases; todas aparecen orientadas a la resolución de ejercicios prácticos.



www.pearsoneducacion.com

ISBN 978-84-832-2924-8



9 788483 229248