



Recursosinformáticos

Java 11

Los fundamentos del lenguaje Java

(con ejercicios prácticos corregidos)

Descarga



ejemplos de código



ejercicios
y correcciones

Thierry GROUSSARD
Thierry RICHARD



JAVA 11

Los fundamentos del lenguaje Java (con ejercicios prácticos corregidos)

Descargado en: <https://dogramcode.com/bloglibros>

PRESENTACIÓN

Este libro se dirige a todos aquellos informáticos que quieran desarrollar en Java. Tanto si es principiante como si ya tiene experiencia con otro lenguaje, el lector encontrará en este libro todos los fundamentos necesarios para familiarizarse rápidamente con uno de los lenguajes más utilizados en el mundo.

Después de una presentación de la plataforma Java SE, su instalación en Windows y Linux y del entorno de desarrollo utilizado en el marco del libro (Eclipse, MySQL), el lector estudia los fundamentos del lenguaje y la programación orientada a objetos. Entre la riqueza del API Java, se pone el foco en el API de gestión de las fechas, en las nuevas posibilidades ofrecidas por las interfaces, así como sobre la noción de los módulos.

Las expresiones lambda y los API subyacentes como el API `java.util.function` y el API Stream se detallan antes de pasar a estudiar el desarrollo de aplicaciones gráficas con la biblioteca Swing. También se presenta el desarrollo de aplicaciones cliente/servidor con el API JDBC, que garantiza el acceso a las bases de datos relacionales.

El despliegue es una etapa importante en el éxito de una aplicación, el último capítulo presenta la distribución de una aplicación con la solución clásica del archivo de ficheros o el uso de la herramienta `jlink`.

ÍNDICE

▢ Presentación

1. Introducción

2. Historia

2.1 ¿ Por qué Java ?

2.2 Objetivos del diseño de Java

2.3 Auge de Java

3. Las características del lenguaje Java

3.1 Sencillo

3.2 Orientado a objetos

3.3 Distribuido

3.4 Interpretado

3.5 Robusto

Presentación

Índice

Autores

- 3.6 Securizado
- 3.7 Independiente de las arquitecturas
- 3.8 Portable
- 3.9 Eficaz
- 3.10 Multitarea
- 3.11 Dinámico
- 4. La plataforma Java SE
 - 4.1 La máquina virtual de Java (JVM)
 - 4.2 La API Java
 - 4.2.1 Las API básicas
 - 4.2.2 Las API de acceso a los datos y de integración con lo existente
 - 4.2.3 Las API de gestión de la interfaz de las aplicaciones con el usuario
 - 4.3 Las herramientas de despliegue de las aplicaciones
 - 4.4 Las herramientas de ayuda al desarrollo
- 5. Las diferentes implementaciones de la plataforma
- 6. El entorno de desarrollo
 - 6.1 Instalación del JDK en Windows
 - 6.1.1 Instalación de la plataforma OpenJDK
 - 6.1.2 Instalación de la plataforma Oracle JDK
 - 6.1.3 Configuración de las variables de entorno
 - 6.1.4 Probar el correcto funcionamiento
 - 6.2 Instalación del JDK en Linux
 - 6.2.1 Instalación de la plataforma OpenJDK
 - 6.2.2 Instalación de la plataforma Oracle JDK
 - 6.2.3 Configuración de la variable JAVA_HOME
 - 6.2.4 Probar el correcto funcionamiento
 - 6.3 Instalación de Eclipse
 - 6.3.1 Descarga del instalador
 - 6.3.2 Instalación de Eclipse
 - 6.3.3 Añadir un acceso directo a Eclipse en Linux
 - 6.3.4 Creación de un proyecto de Java
 - 6.4 Instalación de MySQL
 - 6.4.1 Windows
 - 6.4.2 Linux
 - 6.4.3 Uso de MySQL Workbench
- 7. La javadoc
 - 7.1 Consulta desde un navegador
 - 7.2 Consulta desde Eclipse
- 8. Primer programa
 - 8.1 Escritura del código
 - 8.2 Compilación y ejecución del código
 - 8.2.1 Teoría
 - 8.2.2 Implementación

8.3 Utilización de Eclipse

9. Conclusión

□ Fundamentos del lenguaje

1. Anatomía de un programa

2. Las variables

2.1 Introducción

2.2 Las ubicaciones

2.3 El nombre

2.4 Los tipos

2.4.1 Presentación

2.4.2 Los tipos por valor

2.4.3 Los tipos por referencia

2.5 La declaración, la inicialización y la asignación

2.5.1 La declaración

2.5.2 La inicialización

2.5.3 La asignación

2.6 Los tipos por valor

2.6.1 Los tipos numéricos enteros

2.6.2 Los tipos numéricos decimales

2.6.3 El uso de valores literales

2.6.4 El tipo carácter

2.6.5 El tipo booleano

2.7 Los tipos por referencia

2.7.1 Introducción

2.7.2 Los arrays

2.7.3 Las cadenas de caracteres

2.7.4 Las fecha y las horas

2.8 Los valores por defecto

2.9 El ámbito de las variables

2.10 El ciclo de vida de las variables

2.11 La conversión de tipos

2.11.1 Presentación

2.11.2 La conversión entre números

2.11.3 La conversión a una cadena de caracteres

2.11.4 La conversión desde una cadena de caracteres

2.12 La inferencia de tipos

3. Las constantes

4. Las enumeraciones

5. Los argumentos de un programa

5.1 Funcionamiento

5.2 Utilización en Eclipse

6. Los operadores

6.1 Los operadores unarios

6.2 Los operadores de asignación

6.3 Los operadores aritméticos

6.4 Los operadores bit a bit

6.4.1 La representación binaria de los enteros

6.4.2 Las operaciones lógicas

6.5 Los operadores de comparación

6.6 El operador de concatenación

6.7 Los operadores lógicos

6.8 Orden de evaluación de los operadores

7. Las estructuras de control

7.1 Presentación

7.2 Estructuras de decisión

7.2.1 Estructura if

7.2.2 Estructura ternaria

7.2.3 Estructura switch

7.3 Las estructuras de bucle

7.3.1 Estructura while

7.3.2 Estructura do ... while

7.3.3 Estructura for

7.4 Interrupción de una estructura de bucle

7.4.1 break

7.4.2 continue

7.4.3 return

8. Ejercicios

8.1 Ejercicio 1

8.2 Ejercicio 2

8.3 Ejercicio 3

8.4 Ejercicio 4

Descargado en: <https://dogramcode.com/bloglibros>

9. Correcciones

9.1 Ejercicio 1

9.2 Ejercicio 2

9.3 Ejercicio 3

9.4 Ejercicio 4

10. Conclusión

□ Programación orientada a objetos

1. Introducción

2. Puesta en práctica con Java

2.1 Contexto

2.2 Creación de una clase

2.2.1 Declaración de la clase

2.2.2 Creación de los campos

2.2.3 Creación de métodos

2.2.4 Creación de las sobrecargas de métodos

2.2.5 Paso de argumentos

2.2.6 Creación de accesores

2.2.7 Creación de los constructores y destructores

2.2.8 Creación de campos y métodos estáticos

2.2.9 Utilización de las anotaciones

2.3 Utilización de una clase

2.3.1 Creación de una instancia

2.3.2 Inicialización de una instancia

2.3.3 Destrucción de una instancia

2.4 Herencia

2.4.1 Creación de una clase hija

2.4.2 this y super

2.4.3 Clases abstractas

2.4.4 Clases finales

2.4.5 Conversión de tipo

2.4.6 La clase Object

2.5 Interfaces

2.5.1 Creación de una interfaz

2.5.2 Utilización de una interfaz

2.5.3 Método por defecto

2.5.4 Método estático

2.6 Clases anidadas

2.6.1 Clases anidadas estáticas

2.6.2 Clases internas (de instancia)

2.6.3 Clases anónimas

3. Los paquetes

3.1 Presentación

3.2 Creación de un paquete

3.3 Utilización e importación de un paquete

3.4 Importación de los métodos estáticos

4. Los módulos

4.1 Instalación

4.2 Presentación

4.3 El JDK es modular

4.4 Utilización de los módulos

4.5 Creación de un nuevo módulo

5. La gestión de los errores

5.1 Los diferentes tipos de error

5.1.1 Los errores de sintaxis

5.1.2 Los errores de ejecución

5.1.3 Los errores de lógica

5.2 La representación orientada a objetos de los errores

5.3 El tratamiento de las excepciones

5.4 Las excepciones asociadas a recursos

5.5 Utilización de las excepciones

5.6 Creación y activación de excepciones

6. Los genéricos

6.1 Presentación

6.2 Clases genéricas

6.2.1 Definición de una clase genérica

6.2.2 Utilización de una clase genérica

6.2.3 Los métodos genéricos

6.3 Los genéricos y la herencia

6.4 Limitaciones de los genéricos

7. Las colecciones

7.1 Presentación

7.2 La clase ArrayList

7.3 La clase HashSet

7.4 La clase LinkedList

7.5 La clase HashMap

7.6 Streams y pipelines

8. Ejercicios

8.1 Ejercicio 1

8.2 Ejercicio 2

8.3 Ejercicio 3

8.4 Ejercicio 4

9. Correcciones

9.1 Ejercicio 1

9.2 Ejercicio 2

9.3 Ejercicio 3

9.4 Ejercicio 4

□ Las expresiones lambda

1. Introducción

2. Funcionamiento

2.1 Las interfaces funcionales

2.2 Los métodos anónimos

2.2.1 Sintaxis general

2.2.2 Declaración de los argumentos

2.2.3 Declaración del cuerpo

2.2.4 Utilización de variables "externas"

2.3 Las referencias de métodos

2.3.1 Método de instancia

2.3.2 Método de clase

2.3.3 Constructor

2.4 La API java.util.function

2.4.1 Presentación de la API

2.4.2 Utilización

3. Manipulación de las colecciones

3.1 La API Stream

3.2 Teoría

3.3 Obtener un Stream

3.3.1 Obtener un Stream genérico

3.3.2 Obtener un Stream numérico

3.4 Utilizar un Stream

3.4.1 Utilizar un Stream genérico

3.4.2 Utilizar un Stream numérico

3.5 La clase Optional<T>

4. Conclusión

□ Aplicaciones gráficas

1. Introducción

1.1 Las bibliotecas gráficas

1.1.1 La biblioteca AWT

1.1.2 La biblioteca Swing

1.2 Construcción de la interfaz gráfica de una aplicación

2. Diseño de una interfaz gráfica

2.1 Las ventanas

2.2 El thread EDT

2.3 La gestión de los eventos

2.4 Aspecto de los componentes

2.5 El posicionamiento de los componentes

2.5.1 FlowLayout

2.5.2 BorderLayout

2.5.3 GridLayout

2.5.4 BoxLayout

2.5.5 GridBagLayout

2.5.6 Sin renderizador

2.6 Los componentes gráficos

2.6.1 La clase JComponent

2.6.2 Visualización de la información

2.6.3 Los componentes de edición de texto

2.6.4 Los componentes de activación de acciones

2.6.5 Los componentes de selección

2.7 Los cuadros de diálogo

2.7.1 El cuadro para introducir información

2.7.2 El cuadro de mensaje

2.7.3 El cuadro de confirmación

2.8 Los tratamientos largos

2.8.1 Delegar los tratamientos a un thread hijo

2.8.2 Actualizar el GUI desde un thread hijo

2.8.3 Utilizar la clase SwingWorker

3. Conclusión

□ Acceso a las bases de datos

1. Principios del funcionamiento de una base de datos

1.1 Terminología

1.2 El lenguaje SQL

1.2.1 Búsqueda de información

1.2.2 Inserción de datos

1.2.3 Actualización de datos

1.2.4 Supresión de datos

2. Acceso a una base de datos desde Java

2.1 Presentación de JDBC

2.2 Carga del driver

2.3 Establecer y manipular la conexión

2.3.1 Establecer la conexión

2.3.2 Manipular la conexión

2.4 Ejecución de instrucciones SQL

2.4.1 Ejecución de instrucciones básicas con el objeto de tipo Statement

2.4.2 Ejecución de instrucciones configuradas con el objeto PreparedStatement

2.4.3 Ejecución de procedimientos almacenados con el objeto CallableStatement

2.5 Utilización de los juegos de registros con la interfaz ResultSet

2.5.1 Posicionamiento en un ResultSet

2.5.2 Lectura de los datos en un ResultSet

2.5.3 Modificación de los datos en un ResultSet

2.5.4 Supresión de datos en un ResultSet

2.5.5 Inserción de datos en un ResultSet

2.6 Gestión de las transacciones

2.6.1 Puesta en marcha de las transacciones

2.6.2 Puntos de salvaguarda

2.6.3 Niveles de aislamiento

□ Despliegue de aplicaciones

1. Archivos de Java

1.1 Presentación

1.2 Manipulación de un archivo

1.2.1 Creación de una aplicación

1.2.2 Creación de un archivo

1.2.3 Visualización del contenido

1.2.4 Extracción

1.2.5 Actualización

1.2.6 Ejecución

1.3 El manifest

1.3.1 Presentación

1.3.2 Creación

1.4 La gestión de las dependencias

1.4.1 En un subdirectorio

1.4.2 Dentro del propio archivo

2. Creación de una aplicación autónoma con jlink

2.1 La creación del archivo de Java

2.2 La creación del módulo de la aplicación

2.3 La búsqueda de las dependencias con jdeps

2.4 La creación de la estructura de árbol con jlink

2.5 La adición de dependencias no modulares

2.6 La prueba de la aplicación

3. Externalización de los argumentos

3.1 Creación del archivo de configuración

3.2 Utilización del archivo de configuración

3.3 Despliegue de la aplicación

4. Conclusión

índice

Descargado en: <https://dogramcode.com/bloglibros>

<https://dogramcode.com/bloglibros>



Dogram

JAVA 11

Los fundamentos del lenguaje Java (con ejercicios prácticos corregidos)

Este libro se dirige a todos aquellos informáticos que quieran **desarrollar en Java**. Tanto si es principiante como si ya tiene experiencia con otro lenguaje, el lector encontrará en este libro **todos los fundamentos necesarios** para familiarizarse rápidamente con uno de los lenguajes más utilizados en el mundo.

Después de una presentación de la **plataforma Java SE**, su instalación en **Windows** y **Linux** y del entorno de desarrollo utilizado en el marco del libro (**Eclipse**, **MySQL**), el lector estudia los **fundamentos del lenguaje** y la **programación orientada a objetos**. Entre la riqueza del API Java, se pone el foco en el API de **gestión de las fechas**, en las nuevas posibilidades ofrecidas por las **interfaces**, así como sobre la noción de los **módulos**.

Las **expresiones lambda** y los API subyacentes como el API **java.util.function** y el **API Stream** se detallan antes de pasar a estudiar el desarrollo **de aplicaciones gráficas** con la **biblioteca Swing**. También se presenta el desarrollo de aplicaciones

cliente/servidor con el **API JDBC**, que garantiza **el acceso a las bases de datos relacionales**.

El despliegue es una etapa importante en el éxito de una aplicación, el último capítulo presenta la **distribución** de una aplicación con la solución clásica del **archivo de ficheros** o el uso de la herramienta **jlink**.

Numerosos ejercicios con sus correcciones le permitirán validar sus conocimientos y poner en práctica, de inmediato, las nociones aprendidas.

Existen elementos complementarios para su descarga en esta página.

Autor(es)

Thierry RICHARD - Thierry GROUSSARD

Después de 8 años como ingeniero en diferentes ESN, **Thierry RICHARD** ahora es formador en la escuela de ENI. Se especializa en técnicas de desarrollo de objetos en entornos .NET y Java. Su experiencia se combina con sus habilidades de enseñanza para proporcionar al lector un libro verdaderamente operativo, sobre el desarrollo de aplicaciones en Java.

Analista y desarrollador durante más de 10 años, **Thierry GROUSSARD** se orientó más adelante a la formación, en particular en el campo del desarrollo de software. Sus conocimientos avanzados de las necesidades de la empresa y sus cualidades pedagógicas hacen que este libro esté especialmente adecuado para el aprendizaje y puesta en práctica del desarrollo de aplicaciones en Java.

Introducción

Cuando los ingenieros de Sun Microsystems desarrollaron el lenguaje Java en 1991, no imaginaron que casi treinta años más tarde todavía sería uno de los lenguajes de programación más usados del mundo. Si bien en su origen fue concebido para desarrollar aplicaciones destinadas a sistemas embebidos, a día de hoy está presente en todos los dominios de la informática. Se revela como uno de los lenguajes más demandados en la mayoría de las ofertas de empleo en el campo del desarrollo de software.

Se trata de un lenguaje cuya sintaxis es simple pero rigurosa. Permite por tanto adquirir rápidamente las buenas prácticas desde el comienzo. Sin duda por este motivo es el lenguaje más utilizado en la enseñanza.

El objetivo de este libro es mostrarle los fundamentos de este lenguaje para permitirle a continuación evolucionar hacia el desarrollo de aplicaciones importantes utilizando numerosas tecnologías disponibles con este lenguaje (Java EE, Java Embedded, etc.).

La lectura de este libro no requiere conocimientos previos en desarrollo. Los capítulos Presentación y Fundamentos del lenguaje le presentan las nociones básicas de cualquier lenguaje informático: las variables, los operadores, las condiciones, los bucles...

Tras haber aprendido estos fundamentos, el capítulo Programación orientada a objetos le presenta los principios y la implementación de la programación orientada a objetos (POO). Las nociones expuestas en este capítulo son capitales para poder abordar a continuación el diseño de aplicaciones gráficas. El capítulo siguiente se dedica a las expresiones lambda, que permiten simplificar la escritura de la programación orientada a objetos en algunas situaciones.

El capítulo Aplicaciones gráficas le permite estudiar el diseño de aplicaciones gráficas autónomas con la biblioteca SWING.

Sus futuras aplicaciones requerirán sin duda procesar información alojada en una base de

datos. El capítulo Acceso a las bases Material para descargar a este tema, le proporcionará una preciosa ayuda para realizar esta tarea correctamente. Se familiarizará con el uso de la API JDBC, que es la tecnología utilizada por Java para la gestión del acceso a una base de datos relacional.

El despliegue es en efecto la última etapa en la construcción de una aplicación, pero es un paso que no debe obviarse. El último capítulo de este libro está dedicado a dos tecnologías de despliegue disponibles, lo que le permitirá simplificar la instalación de sus aplicaciones en los puestos clientes.

Este libro no tiene la vocación de sustituir la documentación proporcionada por Oracle, que debe seguir siendo su referencia a la hora de obtener información tal como la lista de métodos o propiedades presentes en una clase.

Introducción

Java ha existido desde hace más de veinte años. Fue creado en 1995 por la empresa Sun Microsystems. Esta empresa fue comprada por Oracle en 2009. Por lo tanto, Java ahora es propiedad de Oracle.

¿Qué es Java?

- Ante todo, es un lenguaje orientado a objetos. Por lo tanto, el lenguaje Java permite escribir programas.
- En segundo lugar, es una plataforma de desarrollo. Está compuesta por un conjunto de bibliotecas, un conjunto de especificaciones (JSR - *Java Specification Requests*) que describen los diferentes API que forman la plataforma y un conjunto de herramientas para compilar, ejecutar, documentar, etc., sus programas. El conjunto forma el JDK (*Java SE Development Kit*).

En realidad, hay varias plataformas.

La plataforma básica se llama **Java SE** (*Java Standard Edition*), como se indicó anteriormente. Cumple con la mayoría de las necesidades.

La segunda plataforma es la plataforma **Java EE**. Su objetivo es permitir la creación de aplicaciones distribuidas y, en particular, aplicaciones web. Esta plataforma se basa en la plataforma Java SE, pero también en software de terceros, los servidores de aplicaciones.

Desde septiembre de 2017 y la versión Java EE 8, la plataforma se ha vendido a la fundación Eclipse. La plataforma ahora se llama **Jakarta EE** y la primera versión estuvo disponible en 2019.

La última plataforma se llama **Java Embedded**. Estrictamente hablando, no es una plataforma única, sino más bien un conjunto de plataformas. Permiten realizar aplicaciones ligeras que se pueden ejecutar en dispositivos integrados. También se usan en el Internet de

las cosas. Es posible citar de manera más particular las plataformas Java TV y Java Card. Estas son una especie de adaptaciones ligeras de la plataforma Java SE.

- Finalmente, es un entorno de ejecución (**JRE** - *Java Runtime Environment*). Permite ejecutar programas a través de la máquina virtual de Java (**JVM** - *Java Virtual Machine*). Se requiere la instalación de JRE en máquinas que ejecutan un programa desarrollado en Java.

Un desarrollador solo necesita instalar el JDK en su puesto de trabajo porque incorpora el JRE.

Historia

1. ¿Por qué Java?

Bill Joy, ingeniero de Sun Microsystems, y su equipo de investigadores trabajaban en el proyecto «Green», que consistía en desarrollar aplicaciones destinadas a una amplia variedad de periféricos y sistemas embebidos (en particular, teléfonos móviles y televisores interactivos).

Convencidos de las ventajas de la programación orientada a objetos (POO), optaron por desarrollar en C++, que ya había demostrado sus capacidades.

Pero, para este tipo de proyecto, C++ mostró pronto sus lagunas y sus límites. En efecto, aparecieron numerosos problemas de incompatibilidad con las diferentes arquitecturas físicas (procesadores, tamaño de memoria) y los sistemas operativos encontrados, así como también a nivel de la adaptación de la interfaz gráfica de las aplicaciones y de la interconexión entre los diferentes dispositivos.

Debido a las dificultades encontradas con C++, era preferible crear un nuevo lenguaje basado en una nueva plataforma de desarrollo. Dos desarrolladores de Sun, James Gosling y Patrick Naughton, se pusieron manos a la obra.

La creación de este lenguaje y plataforma se inspiró en las interesantes funcionalidades propuestas por otros lenguajes tales como C++, Eiffel, Smalltalk, Objective C, Cedar/Mesa, Ada, Perl. El resultado es una plataforma y un lenguaje idóneos para el desarrollo de aplicaciones seguras, distribuidas y portables en numerosos periféricos y sistemas embebidos interconectados en red, y también en Internet (clientes ligeros), así como en estaciones de trabajo (clientes pesados).

Llamado originalmente C++- (C++ sin sus defectos), más tarde OAK (un nombre ya utilizado en informática) lo bautizó finalmente como Java, palabra de argot que significa

café, debido a las cantidades de café tomadas por los programadores y, en particular, por los diseñadores. Y así, en 1991, nació el lenguaje Java.

2. Objetivos del diseño de Java

Basándose en las necesidades expresadas, se necesitaba un lenguaje y una plataforma sencillos y eficaces, destinados al desarrollo y al despliegue de aplicaciones securizadas, en sistemas heterogéneos en un entorno distribuido, con un consumo de recursos mínimo y que funcionara en cualquier plataforma física y de software.

El diseño de Java aportó una respuesta eficaz a esas necesidades:

- Lenguaje de sintaxis sencilla, orientado a objetos e interpretado, que permite optimizar el tiempo y el ciclo de desarrollo (compilación y ejecución).
- Las aplicaciones son portables sin modificación alguna en numerosas plataformas físicas y sistemas operativos.
- Las aplicaciones son resistentes porque el motor de ejecución de Java se encarga de la gestión de la memoria (*Java Runtime Environment*) y es más fácil escribir programas sin fallos, en comparación con C++, debido a un mecanismo de gestión de errores más evolucionado y estricto.
- Las aplicaciones y, en particular, las aplicaciones gráficas son eficaces debido a la puesta en marcha y a la asunción del funcionamiento de varios procesos ligeros (thread y multithreading).
- El funcionamiento de las aplicaciones está securizado principalmente para la comprobación del bytecode antes de su ejecución.

3. Auge de Java

A pesar de la creación de Java, los desarrollos del proyecto «Green» no tuvieron las repercusiones comerciales esperadas y el proyecto fue apartado.

En aquella época, la emergencia de Internet y de las arquitecturas cliente/servidor heterogéneas y distribuidas aportaron cierta complejidad al desarrollo de las aplicaciones.

Las características de Java resultan por lo tanto muy interesantes para este tipo de aplicaciones.

En efecto:

- Puesto que un programa en Java es poco voluminoso, su descarga desde Internet requiere poco tiempo.
- Un programa en Java es portable y se puede utilizar sin modificaciones en cualquier plataforma (Windows, Macintosh, Unix, Linux...).

Java encuentra así un nuevo campo de aplicación en la red global Internet, así como en las redes locales en una arquitectura intranet y cliente/servidor distribuida.

Para presentar al mundo las posibilidades de Java, dos programadores de Sun, Patrick Naughton y Jonathan Peayne, crearon y presentaron en mayo de 1995, en la feria SunWorld, un navegador web programado en su totalidad con Java, llamado HotJava, que permite ejecutar programas de Java, llamados applets, en páginas HTML. Observe que, desde la versión 1.1, los applets ya no se soportan más por la plataforma Java SE.

En agosto de 1995 la empresa Netscape, muy interesada por las posibilidades de Java, firmó un acuerdo con Sun, lo cual le permitió integrar Java e implementar applets en su navegador web (Netscape Navigator). En enero de 1996, la versión 2 de Netscape llega a los mercados integrando la plataforma Java.

Por lo tanto, fue Internet quien aupó a Java.

Respaldado por este éxito, Sun decide, a partir de noviembre de 1995, promover Java entre los programadores, poniendo a su disposición en su sitio web una plataforma de desarrollo en una versión beta llamada JDK 1.0 (*Java Development Kit*).

Poco después, Sun crea una filial llamada JavaSoft (<http://java.sun.com>), cuyo objetivo es continuar el desarrollo de este lenguaje de programación.

Desde entonces, Java no ha dejado de evolucionar muy regularmente para ofrecer un lenguaje y una plataforma polivalentes y sofisticados. Grandes empresas como Borland/Inprise, IBM, Oracle, por citar algunas, apostaron muy fuerte por Java.

A principios de 2009, IBM realiza una tentativa de compra de Sun. Al no alcanzarse un acuerdo acerca del precio de la transacción, el proyecto de compra fracasa. Poco tiempo después, Oracle realiza a su vez una propuesta de compra que esta vez sí se concreta.

A día de hoy, Java es el principal lenguaje orientado a objetos que se enseña en las escuelas y universidades debido a su rigor y su riqueza funcional.

La comunidad de desarrolladores en Java está compuesta por varios millones de personas y es superior en número a la comunidad de desarrolladores en C++ (a pesar de ser, este último, toda una referencia).

Las características del lenguaje Java

Sun caracteriza a Java como un lenguaje sencillo, orientado a objetos, distribuido, interpretado, robusto, securizado, independiente de las arquitecturas, portable, eficaz, multitarea y dinámico.

Dichas características son el resultado del manual escrito en mayo de 1996 por James Gosling y Henry McGilton y disponible en la dirección siguiente:

<http://www.oracle.com/technetwork/java/langenv-140151.html>

Vamos a explicar detalladamente cada una de estas características.

1. Sencillo

La sintaxis de Java es similar a la de los lenguajes C y C++, pero evita características semánticas que los vuelven complejos, confusos y poco seguros:

- En Java solo existen tres tipos primitivos: los numéricos (enteros y reales), el tipo carácter y el tipo booleano. Todos los tipos numéricos están firmados.
- En Java las tablas y las cadenas de caracteres son objetos, lo que facilita su creación y su manipulación.
- En Java el programador no tiene que preocuparse de la gestión de la memoria. Un sistema llamado «recolector de basura» (*garbage collector*) se encarga de asignar la memoria necesaria a la hora de crear objetos y de liberarla cuando estos ya no se referencian en el dominio del programa (cuando ninguna variable apunta al objeto).
- En Java no existen preprocesadores ni archivos de encabezamiento. Las instrucciones `define` de C se sustituyen por constantes en Java y las instrucciones `typedef` de C lo hacen por clases.
- En C y C++ se definen estructuras y uniones para representar tipos de

datos complejos. En Java, se crean instancias de clases para representar tipos de datos complejos.

- En C++ una clase puede heredar de otras clases, lo que puede generar problemas de ambigüedad. Con el fin de evitar estos problemas, Java solo autoriza la herencia simple, pero aporta un mecanismo de simulación de herencia múltiple mediante la implementación de una o varias interfaces.
- En Java no existe la famosa instrucción `goto`, simplemente porque aporta una complejidad a la lectura de los programas y porque a menudo se puede prescindir de esta instrucción escribiendo un código más limpio. Además, en C y C++ se suele utilizar el `goto` para salir de bucles anidados. En Java se utilizarán las instrucciones `break` y `continue`, que permiten salir de uno o varios niveles de anidamiento.
- En Java no es posible sobrecargar los operadores, para evitar problemas de incomprensión del programa. Se preferirá crear clases con métodos y variables de instancia.
- Y para terminar, en Java no hay punteros, sino referencias a objetos o celdas de una tabla (referenciadas por su índice), simplemente porque la gestión de punteros es fuente de muchos errores en los programas C y C++.

2. Orientado a objetos

Salvo los tipos de datos primitivos, todo en Java es un objeto. Adicionalmente Java se ha provisto de clases incorporadas que encapsulan los tipos primitivos si es necesario (las clases wrappers).

Por lo tanto, Java es un lenguaje de programación orientado a objetos y diseñado según el modelo de otros lenguajes (C++, Eiffel, Smalltalk, Objective C, Cedar/Mesa, Ada, Perl), pero sin sus defectos.

Las ventajas de la programación orientada a objetos son: un mejor dominio de la complejidad (al dividir un problema complejo en una serie de pequeños problemas), una reutilización más sencilla y una mayor facilidad de corrección y de evolución.

Java estándar está dotado de un conjunto de clases que permiten crear y trabajar con todo tipo de objetos (interfaz gráfica, acceso a la red, gestión de entradas/salidas...).

3. Distribuido

Java implementa los protocolos de red estándares, lo que permite desarrollar aplicaciones cliente/servidor en arquitecturas distribuidas, con el fin de invocar tratamientos o recuperar datos de máquinas remotas.

Históricamente, Java estándar cuenta con dos API que permiten crear aplicaciones cliente/servidor distribuidas:

- RMI (*Remote Method Invocation*) permite a los objetos de Java comunicarse entre ellos tanto si se ejecutan en diferentes máquinas virtuales de Java como si lo hacen en diferentes máquinas físicas.
- CORBA (*Common Object Request Broker Architecture*), basado en el trabajo del OMG (<http://www.omg.org>), permite la comunicación entre objetos de Java, C++, Lisp, Python, Smalltalk, COBOL, Ada, que se ejecutan en diferentes máquinas físicas.

Esta última API se retira de la plataforma Java SE 11 como consecuencia de la API que explota la tecnología de los servicios web.

4. Interpretado

Un programa de Java no es ejecutado, sino interpretado por la máquina virtual o JVM (*Java Virtual Machine*). Esto conlleva sus ventajas, en particular el hecho de no tener que recompilar un programa de Java de un sistema a otro porque basta, para cada uno de los sistemas, con tener su propia máquina virtual.

Debido a que Java es un lenguaje interpretado, no es necesario editar los enlaces (obligatorio en C++) antes de ejecutar un programa. En Java, por lo tanto, solo hay dos etapas, la compilación y la ejecución. La máquina virtual se encarga de la operación de edición de enlaces en tiempo de ejecución del programa.

5. Robusto

Java es un lenguaje fuertemente tipado y estricto. Por ejemplo, la declaración de las variables debe ser obligatoriamente explícita en Java.

Se verifica el código (sintaxis, tipos) en el momento de la compilación y también de la ejecución, lo que permite reducir los errores y los problemas de incompatibilidad de versiones.

Además, Java se encarga totalmente de la gestión de los punteros y el programador no tiene manera de acceder a ellos, lo que evita la sobreescritura accidental de datos en memoria y la utilización de datos corruptos.

6. Securizado

Dados los campos de aplicación de Java, es muy importante que haya un mecanismo que vigile la seguridad de las aplicaciones y los sistemas. El motor de ejecución de Java (el JRE) es el encargado de esta tarea.

El JRE se apoya en particular en el archivo de texto `java.policy`, que contiene información relativa a la configuración de la seguridad.

En Java, el JRE es el encargado de gestionar el consumo de memoria de los objetos, y no el compilador, como es el caso en C++.

Puesto que en Java no hay punteros, sino referencias a objetos, el código compilado contiene identificadores sobre los objetos que luego el JRE traduce en direcciones de memoria: esta parte es totalmente opaca para los desarrolladores.

En el momento de la ejecución de un programa de Java, el JRE utiliza un proceso llamado `ClassLoader` que realiza la carga del bytecode (o lenguaje binario intermedio) contenido en las clases Java. A continuación, se analiza el bytecode con el fin de controlar que no se generan ni manipulan punteros en memoria y que tampoco hubo violación de acceso.

Como Java es un lenguaje distribuido, se implementan los principales protocolos de acceso a la red (FTP, HTTP, Telnet...). Se puede, pues, configurar el JRE con el fin de controlar el acceso a la red de sus aplicaciones:

- Prohibir todos los accesos.
- Autorizar el acceso solamente a la máquina anfitriona de donde procede el código de aplicación.
- Autorizar el acceso a máquinas en la red externa (más allá del cortafuegos), en el

caso de que el código de la aplicación también proceda de una máquina anfitriona de la red externa.

- Autorizar todos los accesos. Es la configuración por defecto para las aplicaciones de tipo cliente pesado.

7. Independiente de las arquitecturas

El compilador de Java no produce un código específico para un tipo de arquitectura.

De hecho, el compilador genera un bytecode (lenguaje binario intermedio) que es independiente de cualquier arquitectura, de todo sistema operativo y de todo dispositivo de gestión de la interfaz gráfica de usuario (GUI).

La ventaja de este bytecode reside en su fácil interpretación o transformación dinámica en código nativo para aumentar el rendimiento.

Basta con disponer de la máquina virtual específica de su plataforma para hacer funcionar un programa de Java. Esta última se encarga de traducir el bytecode a código nativo.

8. Portable

Java es portable gracias a que se trata de un lenguaje interpretado.

Además, a diferencia de los lenguajes C y C++, los tipos de datos primitivos (numéricos, carácter y booleano) de Java tienen el mismo tamaño, sea cual sea la plataforma en la cual se ejecuta el código.

Las bibliotecas de clases estándares de Java facilitan la escritura de código fuente que, a continuación, se puede desplegar en diferentes plataformas sin adaptación.

9. Eficaz

Incluso si un programa de Java es interpretado, lo cual es más lento que un programa nativo, Java pone en marcha un proceso de optimización de la interpretación del código, llamado JIT (*Just In Time*) o HotSpot. Este proceso compila el bytecode Java en código nativo en tiempo de ejecución, lo que permite alcanzar el mismo rendimiento que un

programa escrito en lenguaje C o C++.

10. Multitarea

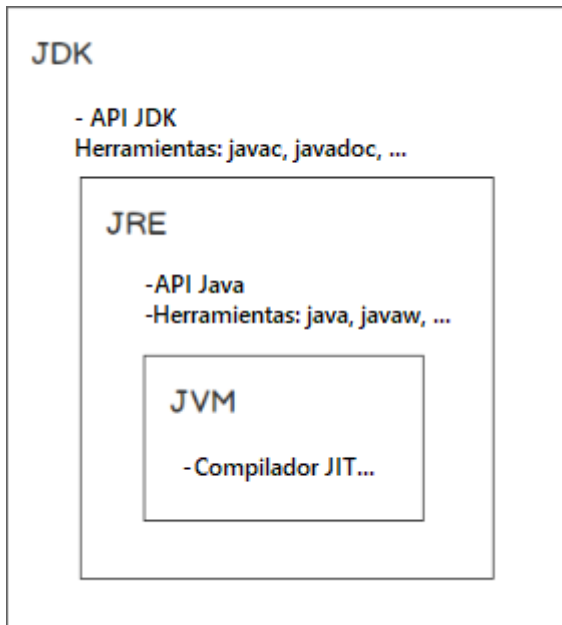
Java permite desarrollar aplicaciones que ponen en marcha la ejecución simultánea de varios hilos (o procesos ligeros). Esto permite efectuar simultáneamente varias tareas con el fin de aumentar la velocidad de las aplicaciones, ya sea compartiendo el tiempo de la CPU o repartiendo las tareas entre varios procesadores.

11. Dinámico

En Java, como dijimos, el programador no tiene que editar los vínculos (obligatorio en C y C++). Por lo tanto, es posible modificar una o varias clases sin tener que efectuar una actualización de estas modificaciones para el conjunto del programa. La comprobación de la existencia de las clases se realiza en tiempo de compilación y la llamada al código de estas clases solo se hace en el momento de la ejecución del programa. Este proceso permite disponer de aplicaciones más ligeras de tamaño en memoria.

La plataforma Java SE

La plataforma Java SE (con sus múltiples aspectos), se puede representar de manera resumida por el siguiente esquema:



La plataforma instalada en el puesto de un desarrollador es el JDK (*Java Development Kit*). Ofrece API específicas, pero sobre todo las herramientas de ayuda al desarrollo, como el compilador `javac`. El JDK incorpora el JRE (*Java Runtime Environment*).

La plataforma instalada en el puesto de destino (en el que se debe ejecutar el programa) es el JRE. Ofrece la API Java y las herramientas necesarias en el entorno de ejecución, como la herramienta `java`, que permite lanzar la ejecución del programa de Java. La JVM (*Java Virtual Machine*) forma parte integrante del JRE. Permite la ejecución efectiva del programa de Java que se pasa como argumento de la herramienta `java`.

Las siguientes secciones presentan más en detalle estos diferentes aspectos de la plataforma.

1. La máquina virtual de Java (JVM)

La máquina virtual es la base de la plataforma Java. Es necesaria para la ejecución de los programas de Java. La JVM está disponible para muchos tipos de ordenadores y de sistemas operativos.

La máquina virtual se encarga:

- De cargar las clases y el bytecode incluidos: cuando un programa invoca la creación de objetos o invoca miembros de una clase, la JVM tiene como misión cargar el bytecode que se va a interpretar.
- De la gestión de la memoria: la JVM se encarga completamente de la gestión de los punteros y, por lo tanto, de cada referencia hecha a un objeto. Este proceso permite también a la JVM encargarse de la liberación automática de la memoria (recolector de basura) en cuanto sale del dominio del programa, es decir, cuando ninguna variable hace referencia a él.
- De la seguridad: es una de las operaciones más complejas realizadas por la JVM. Al cargar el programa, comprueba que no se llama a memoria no inicializada, que no se efectúan conversiones de tipos ilegales y que el programa no manipula punteros de memoria.
- De la interfaz con el código nativo (por ejemplo, código escrito en lenguaje C): la mayoría de las API básicas de Java necesitan código nativo que viene con el JRE con el fin de interactuar con el sistema anfitrión. También se puede utilizar este proceso para acceder a periféricos o a funcionalidades que no se implementan directamente, incluso que no se implementan en absoluto en Java.

El hecho de que Java sea interpretado conlleva ventajas e inconvenientes. Desde siempre, se reprocha a Java ser menos eficaz que los lenguajes nativos, como era el caso sobre todo para aplicaciones con interfaz gráfica de usuario. Con el fin de paliar este problema y perder esta mala imagen injustificada, los desarrolladores de Oracle han trabajado muchísimo en la optimización de la JVM.

Con la versión 1.2, se dispuso de un compilador JIT (*Just In Time*) que permitía optimizar la interpretación del bytecode al modificar su estructura para acercarlo al código nativo. A partir de la versión 1.3, la JVM integra un proceso llamado HotSpot (cliente y servidor) que optimiza aún más la interpretación del código y, de manera general, el rendimiento de la

JVM. HotSpot aporta una ganancia de resultados de entre el 30 % y el 40 % según el tipo de aplicación (se nota especialmente a nivel de las interfaces gráficas).

Descargado en: <https://dogramcode.com/bloglibros>

2. La API Java

La API Java es la API principal e inevitable para desarrollar con el lenguaje Java. La API Java contiene una colección de componentes de software prefabricados que proporcionan numerosas funcionalidades.

La API Java en su versión 11 se organiza en 21 módulos que agrupan más de 220 paquetes, el equivalente a las librerías de C. Cada paquete contiene clases e interfaces prefabricadas y directamente reutilizables. Hay disponibles unas 4400 clases e interfaces. La API del JDK añade módulos adicionales. El grado de esta API es medio.

La API Java se puede descomponer a grandes rasgos, en tres categorías de API:

- Las API básicas,
- Las API de acceso a los datos,
- Las API de gestión de la interfaz de las aplicaciones con el usuario.

Se pueden añadir numerosas extensiones que están disponibles en el sitio Java de Oracle: gestión de imágenes en 3D, de puertos de comunicación del ordenador, de telefonía, de correos electrónicos...

a. Las API básicas

Las API básicas permiten gestionar:

- elementos esenciales, como los objetos, las cadenas de caracteres, los números, las entradas/salidas, las estructuras y colecciones de datos, las propiedades del sistema, la fecha y la hora, y mucho más...;
- la red, con los protocolos estándares tales como FTP, HTTP, UDP, TCP/IP más las URL y la manipulación de los sockets;
- la internacionalización y la adaptación de los programas de Java, al externalizar las cadenas de caracteres contenidas en el código de los archivos de propiedades (.properties). Este proceso permite adaptar el funcionamiento de las aplicaciones en función de entornos dinámicos (nombre de servidor, nombre de usuario,

contraseña...) y adaptar el idioma utilizado en las interfaces gráficas según el contexto regional de la máquina;

- la interfaz con el código nativo, al permitir declarar que la implementación de un método se haga dentro de una función de una DLL, por ejemplo;
- la seguridad, al permitir:
 - cifrar/descifrar los datos (JCE - *Java Cryptography Extension*),
 - poner en marcha una comunicación securizada mediante SSL y TLS (JSSE - *Java Secure Socket Extension*),
 - autenticar y gestionar las autorizaciones de los usuarios en las aplicaciones (JAAS - *Java Authentication and Authorization Service*),
 - intercambiar mensajes con total seguridad entre aplicaciones que se comunican mediante un servidor como Kerberos (GSS-API - *Generic Security Service - Application Program Interface*),
 - crear y validar listas de certificados llamadas Certification Paths (*Java Certification Path API*);
- la creación de componentes de software llamados JavaBeans reutilizables y capaces de comunicarse con otras arquitecturas de componentes tales como ActiveX, OpenDoc, LiveConnect;
- la manipulación de datos XML (*eXtensible Markup Language*) con la ayuda de las API DOM (*Document Object Model*) y SAX (*Simple API for XML*). Las API básicas permiten también aplicar transformaciones XSLT (*eXtensible Stylesheet Language Transformation*) a partir de hojas de estilo XSL sobre datos XML;
- la generación de archivos históricos (logs) que permiten obtener el estado del funcionamiento de las aplicaciones (actividad, errores, bugs...);
- la manipulación de cadenas de caracteres con expresiones regulares;
- los errores de sistema de operación con el mecanismo de excepciones encadenadas;
- las preferencias de usuario o de sistema, al permitir a las aplicaciones almacenar y recuperar datos de configuración en diferentes formatos.

b. Las API de acceso a los datos y de integración con lo existente

Estas API permiten gestionar:

- aplicaciones cliente/servidor en una arquitectura distribuida, al permitir la comunicación en local o por red entre objetos de Java que funcionan en contextos de JVM diferentes, gracias a la API RMI (*Remote Method Invocation*);
- el acceso a casi el 100 % de las bases de datos mediante la API JDBC (*Java DataBase Connectivity*);
- el acceso a los datos almacenados en servicios de directorio del protocolo LDAP (*Lightweight Directory Access Protocol*), como por ejemplo el Active Directory de Windows, mediante la API JNDI (*Java Naming and Directory Interface*).

c. Las API de gestión de la interfaz de las aplicaciones con el usuario

Las API de gestión de la interfaz de usuario permiten gestionar:

- el diseño de interfaces gráficas con la API AWT (*Abstract Window Toolkit*) de antigua generación o la API SWING de última generación. En la actualidad, se prefiere la API Java FX para el desarrollo de interfaces gráficas. No está disponible por defecto con el JDK. Hay que descargarlo en la siguiente dirección: <https://openjfx.io>;
- el sonido, con la manipulación, la lectura y la creación de archivos de sonido de diferentes formatos (*.wav* o *.midi*);
- la grabación de datos en formato texto usando medios distintos al teclado, como, por ejemplo, mecanismos de reconocimiento de voz o de escritura, con la API Input Method Framework;
- las operaciones gráficas de dibujo con la API Java 2D y de manipulación de imágenes con la API Java Image I/O;
- la accesibilidad de las aplicaciones para personas discapacitadas con la API Java Accessibility, que permite interactuar, por ejemplo, con sistemas de reconocimiento de voz o terminales en braille;
- el desplazamiento o traslado de datos durante una operación de arrastrar/soltar (*Drag and Drop*);
- trabajos de impresión de datos en cualquier periférico de impresión.

3. Las herramientas de despliegue de las aplicaciones

Los applets y la tecnología Java Web Start se suspendieron con la llegada de la versión 11.

El despliegue de una aplicación ahora es posible utilizando la herramienta `jlink` que permite crear imágenes autoejecutables, incorporando el programa, las dependencias y las herramientas necesarias para su ejecución. Para obtener más información, consulte el capítulo Despliegue de aplicaciones.

4. Las herramientas de ayuda al desarrollo

La mayoría de las herramientas de ayuda al desarrollo se encuentran en la carpeta `bin` de la carpeta raíz de la instalación de la plataforma.

Las principales herramientas de ayuda al desarrollo permiten:

- compilar (`javac.exe`) el código fuente de archivos `.java` en archivos `.class`;
- generar de forma automática (`javadoc.exe`) la documentación del código fuente (nombre de clase, paquete, jerarquía de herencia, enumeración de las variables y métodos) con el mismo estilo de presentación que la documentación oficial de las API estándares proporcionadas por Sun;
- lanzar la ejecución (`java.exe`) de las aplicaciones autónomas de Java.

También son interesantes otras dos tecnologías, destinadas a integrarse en herramientas de desarrollo de terceros:

- JPDA (*Java Platform Debugger Architecture*): permite integrar una herramienta de depuración dentro del IDE de desarrollo, lo que aporta funcionalidades tales como puntos de interrupción, ejecución paso a paso, la inspección de variables y expresiones...
- JVMPI (*Java Virtual Machine Profiler Interface*): permite efectuar análisis y generar estados relativos al funcionamiento de las aplicaciones (memoria utilizada, objetos creados, número y frecuencia de invocación de los métodos, tiempo de proceso...) con el fin de observar el buen funcionamiento de las aplicaciones y localizar los cuellos de botella.

Las diferentes implementaciones de la plataforma

Hay disponibles dos implementaciones de la plataforma JAVA SE: Oracle JDK (u Oracle Java SE) y OpenJDK. La diferencia radica en la disponibilidad de herramientas avanzadas y soporte específico para la implementación de Oracle JDK a cambio de una suscripción mensual.

Important changes in Oracle JDK 11 License

With JDK 11 Oracle has updated the license terms on which we offer the Oracle JDK. The new [Oracle Technology Network License Agreement for Oracle Java SE](#) is substantially different from the licenses under which previous versions of the JDK were offered. Please review the [new](#) terms carefully before downloading and using this product.

Oracle also offers this software under the [GPL License](http://jdk.java.net/11) on jdk.java.net/11

Ahora es posible usar la plataforma Oracle JDK de forma gratuita solo para crear prototipos, probar y aprender. Para cualquier uso profesional, es necesario activar una suscripción mensual.

La funcionalidad del lenguaje es idéntica. La implementación de Oracle JDK se basa en la implementación de OpenJDK.

El entorno de desarrollo

El propósito de esta sección es describir los pasos principales para configurar un entorno de desarrollo operativo, compuesto por los siguientes elementos en un sistema operativo Windows y Linux (Ubuntu 18.10):

- El JDK (para elegir entre OpenJDK y Oracle JDK),
- Eclipse IDE,
- El servidor de bases de datos MySQL.

1. Instalación del JDK en Windows



a. Instalación de la plataforma OpenJDK

Para instalar la plataforma OpenJDK, siga las siguientes etapas:

- Acceda a la página de descarga <https://jdk.java.net/11>.

Aparece la siguiente pantalla:

Descargado en: <https://dogramcode.com/bloglibros>

jdk.java.net

- GA Releases
- JDK 11
- Early-Access Releases
- JDK 13
- JDK 12
- Jpackage
- OpenJFX
- Panama
- Valhalla
- JMC
- Reference Implementations
- Java SE 11
- Java SE 10
- Java SE 9
- Java SE 8
- Java SE 7
- Feedback
- Report a bug
- Archive

JDK 11.0.2 General-Availability Release

This page provides production-ready open-source builds of the Java Development Kit, version 11.0.2, an implementation of the Java SE 11.0.2 Platform under the GNU General Public License, version 2, with the Classpath Exception.

Commercial builds of JDK 11.0.2 from Oracle under a non-open-source license, for a wider range of platforms, can be found at the Oracle Technology Network.

Documentation

- Features
- Release notes
- API Javadoc
- Tool & command reference

Builds

Linux/x64	tar.gz (sha256)	187513052 bytes
macOS/x64	tar.gz (sha256)	182670822
Windows/x64	zip (sha256)	187383323

- Descargue el archivo adaptado a su SO (por ejemplo, Windows/x64). Observe la ausencia de instalador.
- Descomprima el archivo en el directorio que elija y la instalación termina.

Cuando haya una nueva versión disponible, la descarga de la versión deseada siempre será posible en la siguiente dirección: <https://jdk.java.net/archive>

b. Instalación de la plataforma Oracle JDK

Para instalar la plataforma Oracle JDK, siga estas etapas:

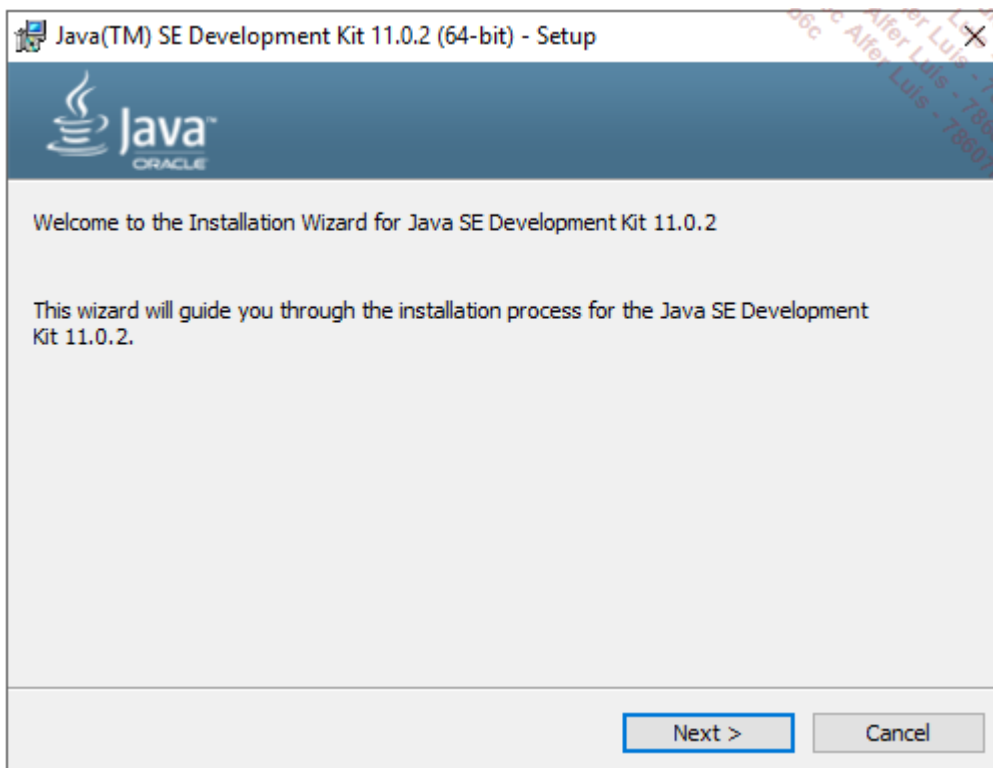
- Acceda a la página de descarga:
<https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html>.

La pantalla siguiente debe aparecer:

- La versión 11.X.X actual está disponible para su descarga. En el momento de escribir este libro, la versión es la 11.0.2.

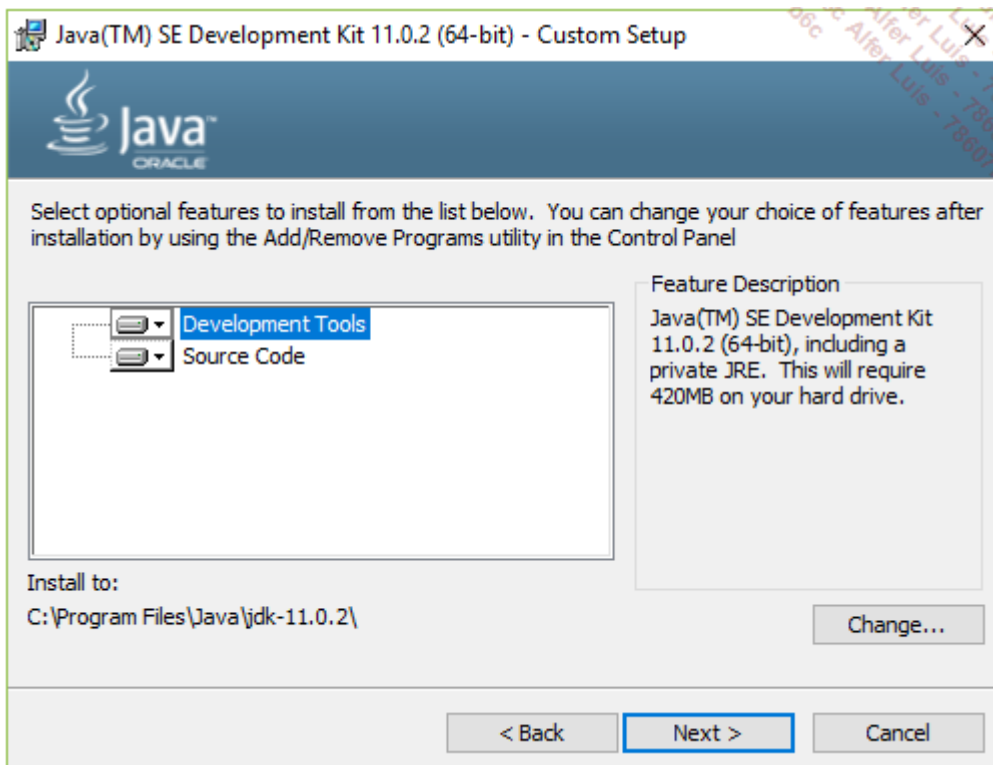
Java SE Development Kit 11.0.2		
You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.		
☒ Accept License Agreement ☐ Decline License Agreement		
Product / File Description	File Size	Download
Linux	147.28 MB	jdk-11.0.2_linux-x64_bin.deb
Linux	154.01 MB	jdk-11.0.2_linux-x64_bin.rpm
Linux	171.32 MB	jdk-11.0.2_linux-x64_bin.tar.gz
macOS	166.13 MB	jdk-11.0.2_osx-x64_bin.dmg
macOS	166.49 MB	jdk-11.0.2_osx-x64_bin.tar.gz
Solaris SPARC	186.78 MB	jdk-11.0.2_solaris-sparcv9_bin.tar.gz
Windows	150.94 MB	jdk-11.0.2_windows-x64_bin.exe
Windows	170.96 MB	jdk-11.0.2_windows-x64_bin.zip

- Acepte los términos de la licencia y lance la descarga de la versión adaptada a su OS. En el marco de Windows, puede seleccionar el instalador llamado `jdk-11.0.2_windows-x64_bin.exe`.
- Ejecute el instalador y pase las etapas siguientes:



Descargado en: <https://dogramcode.com/bloglibros>

- Pulse en **Next**.



- Modifique eventualmente el directorio de instalación. Por defecto, el valor es C:\Program Files\Java\jdk-11.0.2\. A continuación pulse **Next** para arrancar la instalación.



- La visualización de esta pantalla indica el éxito de la instalación. Pulse **Close**. Esta etapa ha terminado.

c. Configuración de las variables de entorno

Para usar herramientas de desarrollo como `javac.exe` o `java.exe` sin la ruta completa al ejecutable, debe agregar el directorio que contiene estos ejecutables a la variable de entorno `PATH`. Este directorio se corresponde con el directorio `\bin` de la instalación del JDK. El directorio de instalación del JDK se puede representar mediante la variable de entorno `JAVA_HOME`. Esta variable se utiliza por un gran número de aplicaciones.

En conclusión, es oportuno:

- Añadir la variable de entorno `JAVA_HOME` poniendo como valor la ruta de acceso al directorio de instalación del JDK. A continuación se muestra un ejemplo:

```
C:\Program Files\Java\jdk-11.0.2
```

- modificar la variable de entorno `PATH` añadiendo al final de esta variable el texto siguiente, que permite el acceso al directorio `\bin` del JDK:

```
;%JAVA_HOME%\bin
```


d. Probar el correcto funcionamiento

Para probar el correcto funcionamiento, es suficiente con ejecutar el comando siguiente:

```
java -version
```

Descargado en: www.detodoprogramacion.org

Si la instalación se corresponde con OpenJDK, el resultado es el siguiente:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.18362.1016]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\>java -version
openjdk version "11.0.2" 2019-01-15
openJDK Runtime Environment 18.9 (build 11.0.2+9)
openJDK 64-Bits Server VM 18.9 (build 11.0.2+9, mixed mode)

C:\>
```

Descargado en: www.detodoprogramacion.org

Si la instalación se corresponde con Oracle JDK, el resultado es el siguiente:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.18362.1016]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\>java -version
java version "11.0.2" 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed mode)

C:\>
```

Descargado en: www.detodoprogramacion.org

- ☐ Es posible instalar varias versiones diferentes en la misma máquina. Este comando permite saber qué versión se utiliza por defecto.

2. Instalación del JDK en Linux

a. Instalación de la plataforma OpenJDK

El procedimiento de instalación de la plataforma OpenJDK se simplifica en el

entorno Linux gracias al uso de la herramienta `apt` (o `apt-get`) para las distribuciones que lo ofrecen.

Los siguientes comandos permiten instalar la versión más reciente de OpenJDK disponible para la distribución Linux.

```
sudo apt update  
sudo apt install default-jdk
```

En Ubuntu 18.10, la versión por defecto es la 11.0.1.

Si no es la versión deseada, es posible instalar una versión específica escribiendo la instrucción siguiente:

```
sudo apt install openjdk-NUMERO_VERSION-jdk
```

Para instalar la versión 8, es suficiente con escribir el siguiente comando:

```
sudo apt install openjdk-8-jdk
```

Si la versión deseada no está disponible por demasiado antigua o demasiado reciente, la solución es descargarla desde el sitio web <https://jdk.java.net>. El procedimiento de instalación es el descrito en la siguiente sección. Por lo tanto, es posible instalar la versión 11.0.2, que es la última versión al escribir estas líneas.

b. Instalación de la plataforma Oracle JDK

Para instalar la plataforma Oracle JDK, siga estas etapas:

- Acceda a la página de descarga
<https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html>. Es la misma que para el entorno Windows.

Debe aparecer la siguiente pantalla:

Java SE Development Kit 11.0.2		
You must accept the Oracle Technology Network License Agreement for Oracle Java SE to download this software.		
☒ Accept License Agreement ☐ Decline License Agreement		
Product / File Description	File Size	Download
Linux	147.28 MB	jdk-11.0.2_linux-x64_bin.deb
Linux	154.01 MB	jdk-11.0.2_linux-x64_bin.rpm
Linux	171.32 MB	jdk-11.0.2_linux-x64_bin.tar.gz
macOS	166.13 MB	jdk-11.0.2_osx-x64_bin.dmg
macOS	166.49 MB	jdk-11.0.2_osx-x64_bin.tar.gz
Solaris SPARC	186.78 MB	jdk-11.0.2_solaris-sparcv9_bin.tar.gz
Windows	150.94 MB	jdk-11.0.2_windows-x64_bin.exe
Windows	170.96 MB	jdk-11.0.2_windows-x64_bin.zip

- Acepte los términos de la licencia e inicie la descarga de la versión adaptada a su SO. En el marco de Ubuntu, seleccione el archivo `jdk-11.0.2_linux-x64_bin.tar.gz`.
- Descomprima el archivo en el directorio de su elección (aquí en el directorio que contiene el archivo):

```
tar xvzf 11.0.2_linux-x64_bin.tar.gz
```

Se crea un directorio llamado `jdk-11.0.2`.

- Eventualmente mueva este directorio al directorio de su elección. A continuación se muestra el comando que debe utilizar para moverlo al directorio `/usr/lib/jvm`:

```
sudo mv jdk-11.0.2 /usr/lib/jvm
```

- Sitúese en el directorio bin del JDK:

```
cd /usr/lib/jvm/jdk-11.0.2/bin
```

- Cambie los permisos sobre los ejecutables que quiera usar (como mínimo, `java` y `javac`):

```
sudo chmod a+x java
sudo chmod a+x javac
```

- Defina los ejecutables `java` y `javac` de esta instalación como referencia usando la herramienta `update-alternatives`:

```
sudo update-alternatives --install /usr/bin/java  
/usr/lib/jvm/jdk-11.0.2/bin/java 1500
```

```
sudo update-alternatives --install /usr/bin/javac  
/usr/lib/jvm/jdk-11.0.2/bin/javac 1500
```

- ☐ El número 1500 es una prioridad asociada con los ejecutables `java` y `javac` de esta versión del JDK. Si se instalan varias versiones del JDK, el SO usará por defecto las herramientas con la más alta prioridad. Para ver las prioridades de estos ejecutables para otras versiones del JDK, ejecute el comando siguiente:

```
sudo update-alternatives --config NOMBRE_DEL_EXECUTABLE
```

- ☐ Este comando también le permite seleccionar manualmente la versión del ejecutable para usar en una lista numerada. En este caso, tenga cuidado con actualizar todos los ejecutables utilizados.

c. Configuración de la variable `JAVA_HOME`

Al igual que en el entorno de Windows, es interesante agregar la variable de entorno `JAVA_HOME` para que algunas herramientas de terceros puedan usarla para su buen funcionamiento. Esta variable simplemente contiene la ruta de acceso al directorio de instalación-dirección del JDK.

A continuación se muestran las operaciones que se deben realizar:

- ☐ Abrir el archivo `/etc/environment` con un editor de texto, como `nano`, por ejemplo:

```
sudo nano /etc/environment
```

- ☐ Añada la línea siguiente:

```
JAVA_HOME="ruta/al/directorio/instalación/java"
```

Ejemplo:

```
JAVA_HOME="/usr/lib/jvm/java-11-openjdk-amd64"
```

- Guarde el archivo.
- Busque el archivo con el comando siguiente:

```
source /etc/environment
```

- Verifique la existencia y el valor de la variable `JAVA_HOME` con el comando siguiente:

```
echo $JAVA_HOME
```

El resultado se debe parecer a esto:

```
ruta/al/directorio/instalación/java
```

Ejemplo:

```
/usr/lib/jvm/java-11-openjdk-amd64
```

- Esta variable se debe actualizar si cambia de versión.

d. Probar el correcto funcionamiento

Para probar el buen funcionamiento, es suficiente con ejecutar el comando siguiente:

```
java -version
```

Si la instalación se corresponde con OpenJDK a través de la herramienta `apt`, el resultado es el siguiente:

```
java@ubuntu-java:~$ java -version
openjdk version "11.0.1" 2018-10-16
OpenJDK Runtime Environment (build 11.0.1+13-Ubuntu-3ubuntu3.18.10.1)
OpenJDK 64-Bit Server VM (build 11.0.1+13-Ubuntu-3ubuntu3.18.10.1, mixed
mode, sharing)
java@ubuntu-java:~$
```

☐ Puede observar que la versión no es la más reciente (11.0.1).

Si la instalación se corresponde a Oracle JDK, el resultado es el siguiente:

```
java@ubuntu-java:~$ java -version
java version "11.0.2" 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed mode)
java@ubuntu-java:~$
```

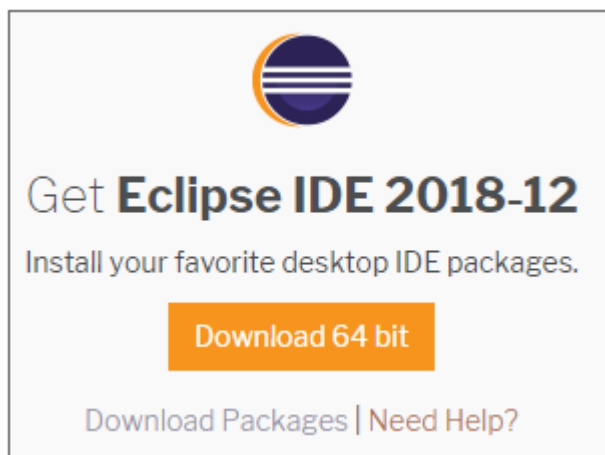
3. Instalación de Eclipse

a. Descarga del instalador

Todo empieza por la descarga del instalador en la dirección siguiente:

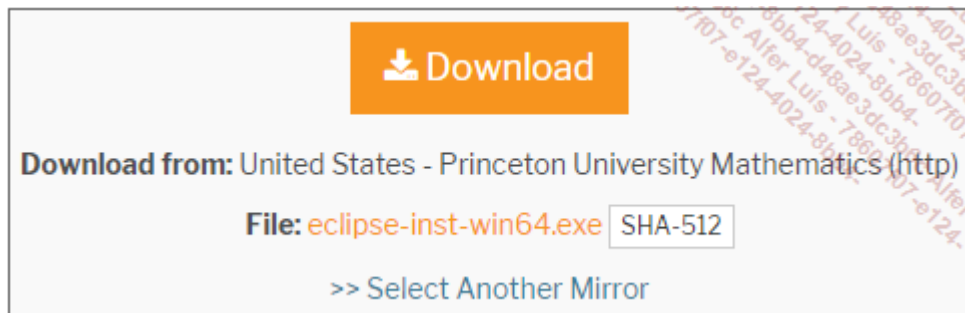
<https://www.eclipse.org/downloads/>

Pulse el enlace siguiente cuando se cargue la página:

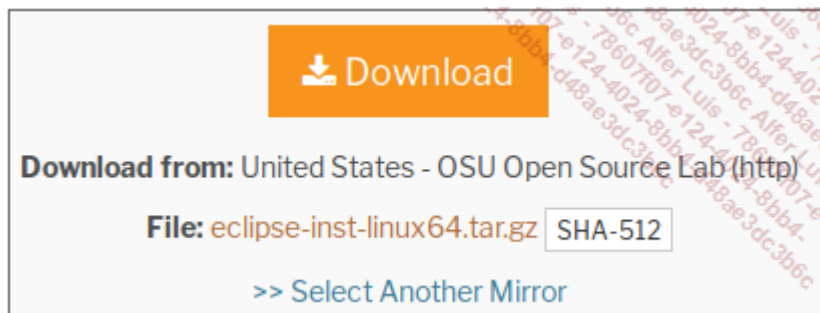


El instalador propuesto está directamente relacionado con el SO de la máquina desde la que accede a la página de descarga.

Desde Windows, se muestra la siguiente página. Ofrece la descarga del instalador.



Desde Linux, se muestra la siguiente página. Ofrece la descarga de un archivo que contiene el instalador.



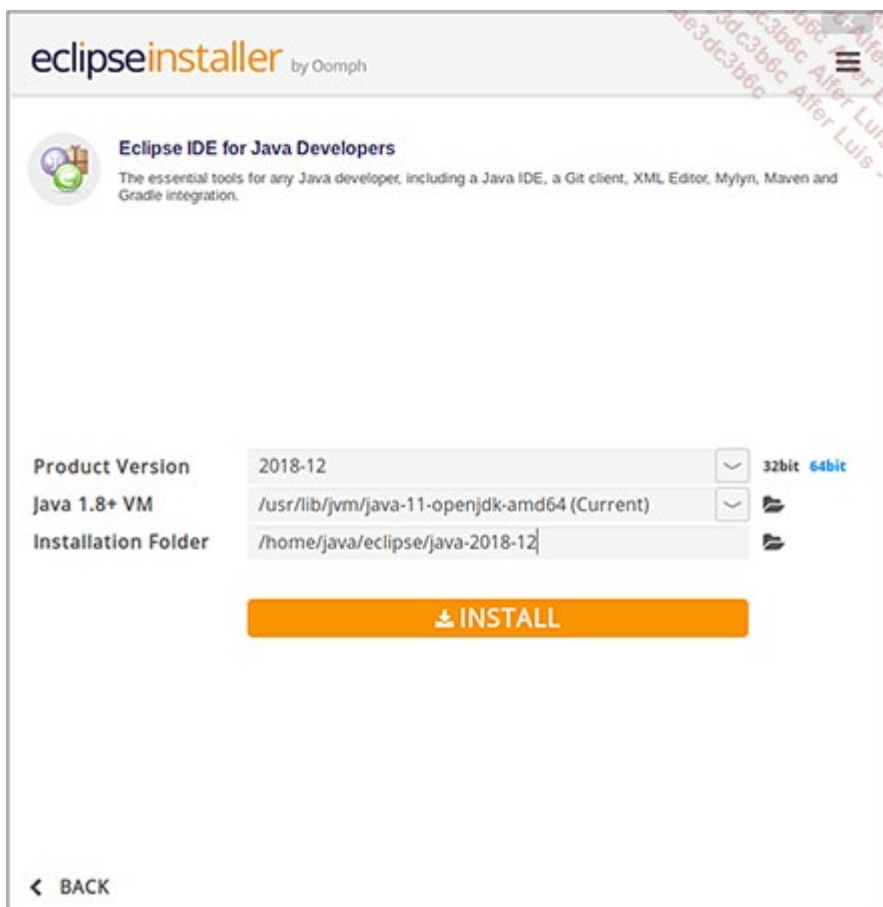
b. Instalación de Eclipse

Cuando el instalador se descarga, es suficiente con descomprimirlo (en el caso de la descarga de un archivo), ejecutarlo y seguir el procedimiento siguiente.

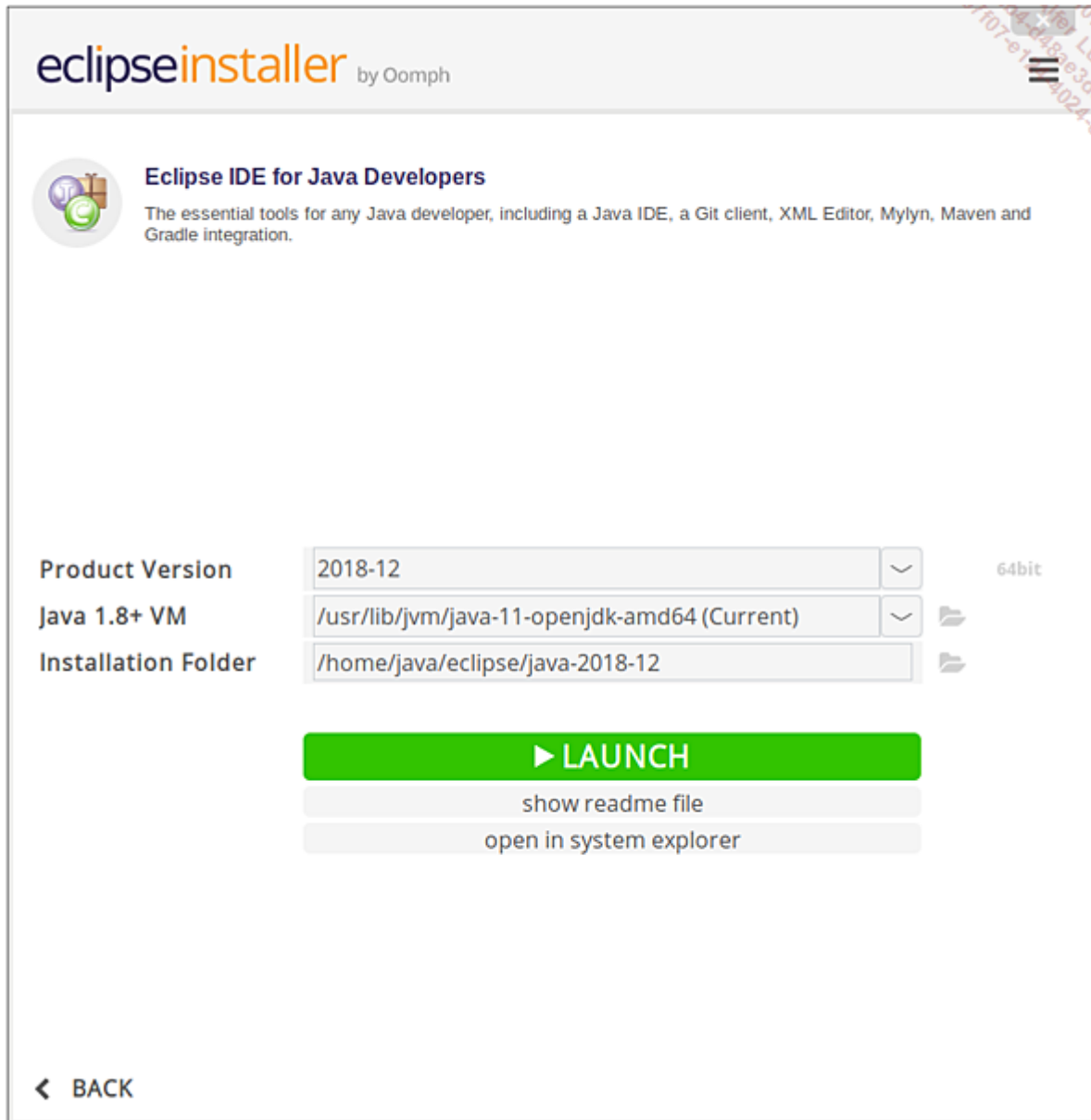
La primera pantalla que se muestra es la siguiente:



- Pulse el primer enlace **Eclipse IDE for Java Developers** para desencadenar la instalación de Eclipse. Aparece la siguiente pantalla:



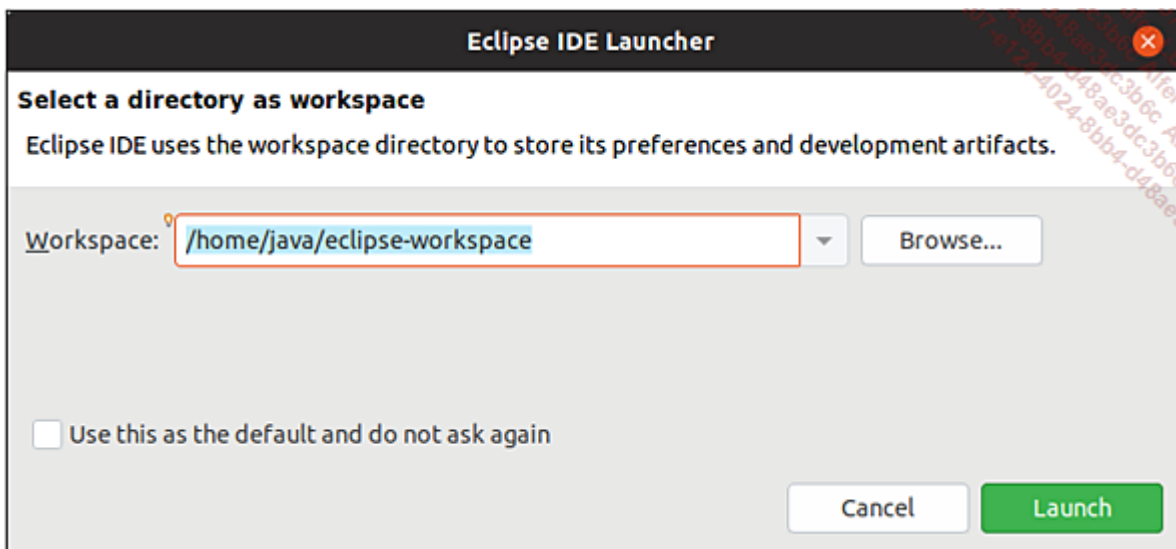
- Modifique eventualmente la versión de Eclipse deseada (**Product Version**), el JDK utilizado (**Java 1.8+ VM**) y el directorio de instalación (**Installation Folder**) antes de pulsar el botón de instalación (**INSTALL**).
- La instalación comienza y puede llevar algunos minutos porque el instalador descarga la versión de Eclipse seleccionada.
- Acepte los certificados propuestos.
- Al finalizar la instalación, se muestra la siguiente pantalla:



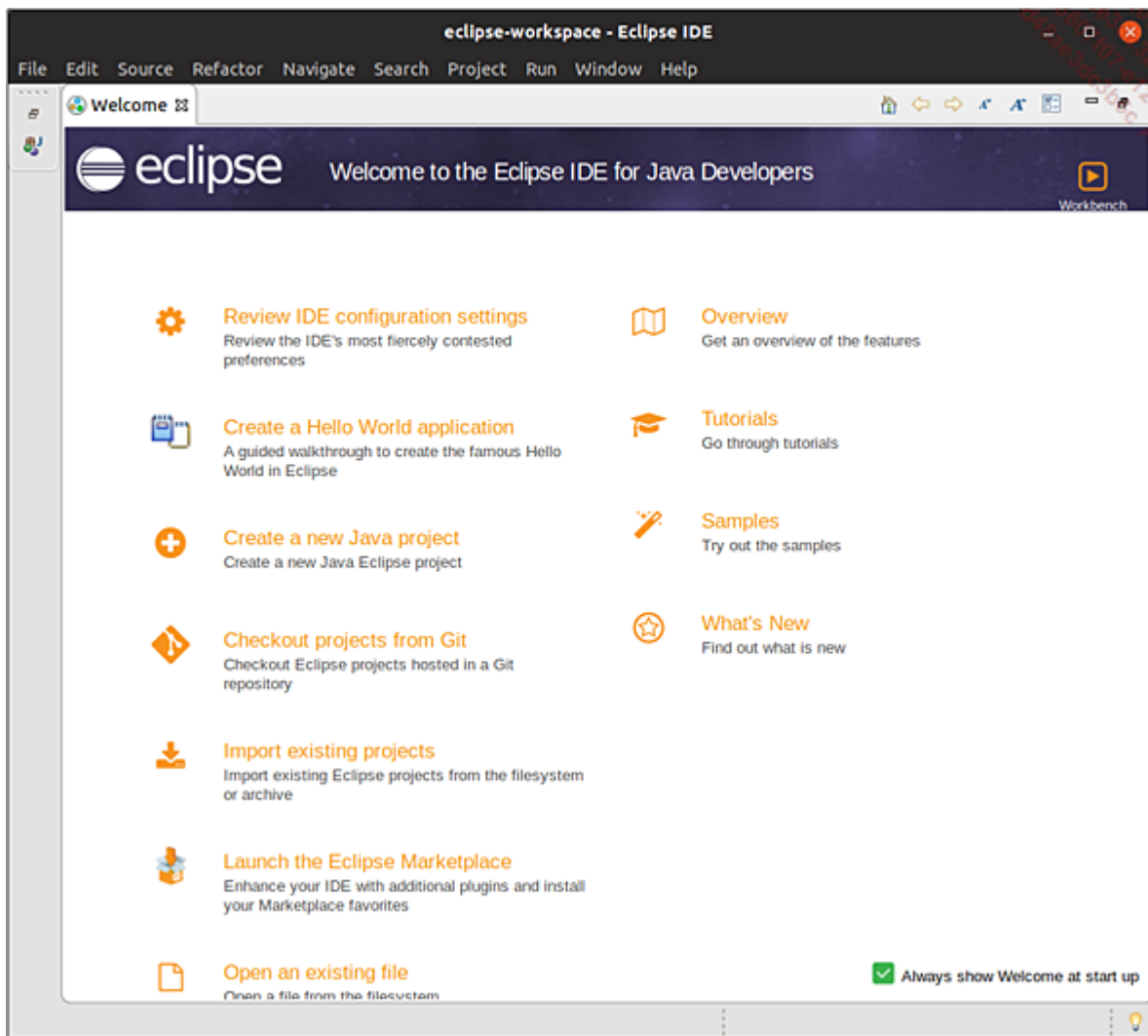
- Pulse el botón **LAUNCH** para lanzar Eclipse. Aparece la siguiente pantalla:



- Pocos segundos después, aparece la siguiente pantalla:



- Indique el directorio de trabajo (**Workspace**) en el que se crearán los proyectos de Java y en el que se encuentran sus recursos de Java. Pulse el botón **Launch** para terminar de arrancar Eclipse.



- Felicidades, Eclipse se ha instalado y está operativo.

c. Añadir un acceso directo a Eclipse en Linux

Para añadir un acceso directo a Eclipse:

- Cree un archivo llamado `eclipse.desktop` en el directorio `/usr/share/applications` con el siguiente contenido:

```
[Desktop Entry]
Name=Eclipse
Type=Application
Exec=DIRECTORIO_INSTALACIÓN_ECLIPSE/eclipse
Terminal=false
Icon=DIRECTORIO_INSTALACIÓN_ECLIPSE/icon.xpm
Comment=Integrated Development Environment
NoDisplay=false
Categories=Development;IDE
```

Name[en]=Eclipse

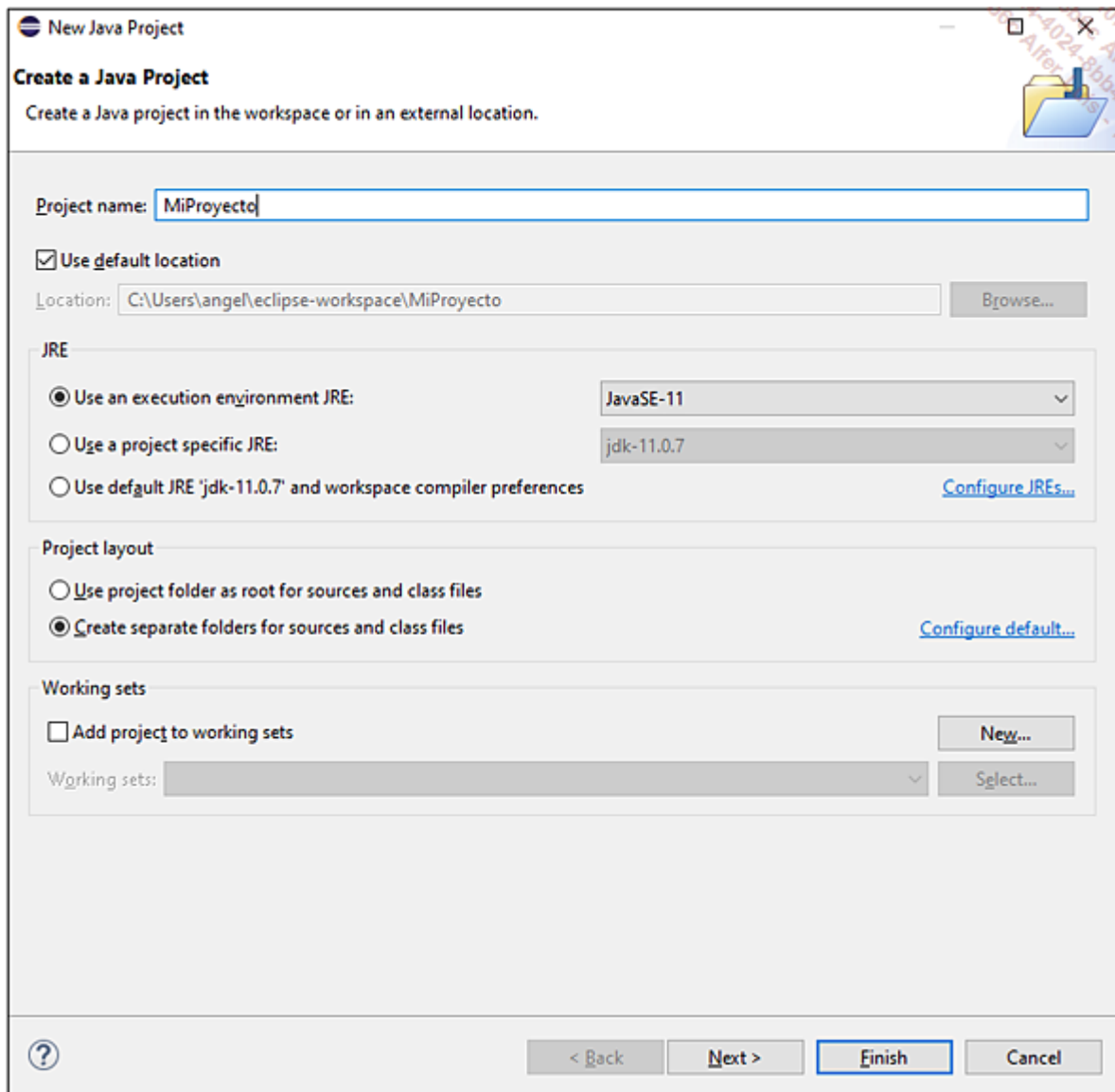
- Sustituya el texto `DIRECTORIO_INSTALACIÓN_ECLIPSE` por el valor correspondiente. En el marco del entorno establecido en este libro, el valor es:

`/home/java/eclipse/java-2018-12/eclipse`

d. Creación de un proyecto de Java

La creación de un proyecto de Java en Eclipse sigue este procedimiento:

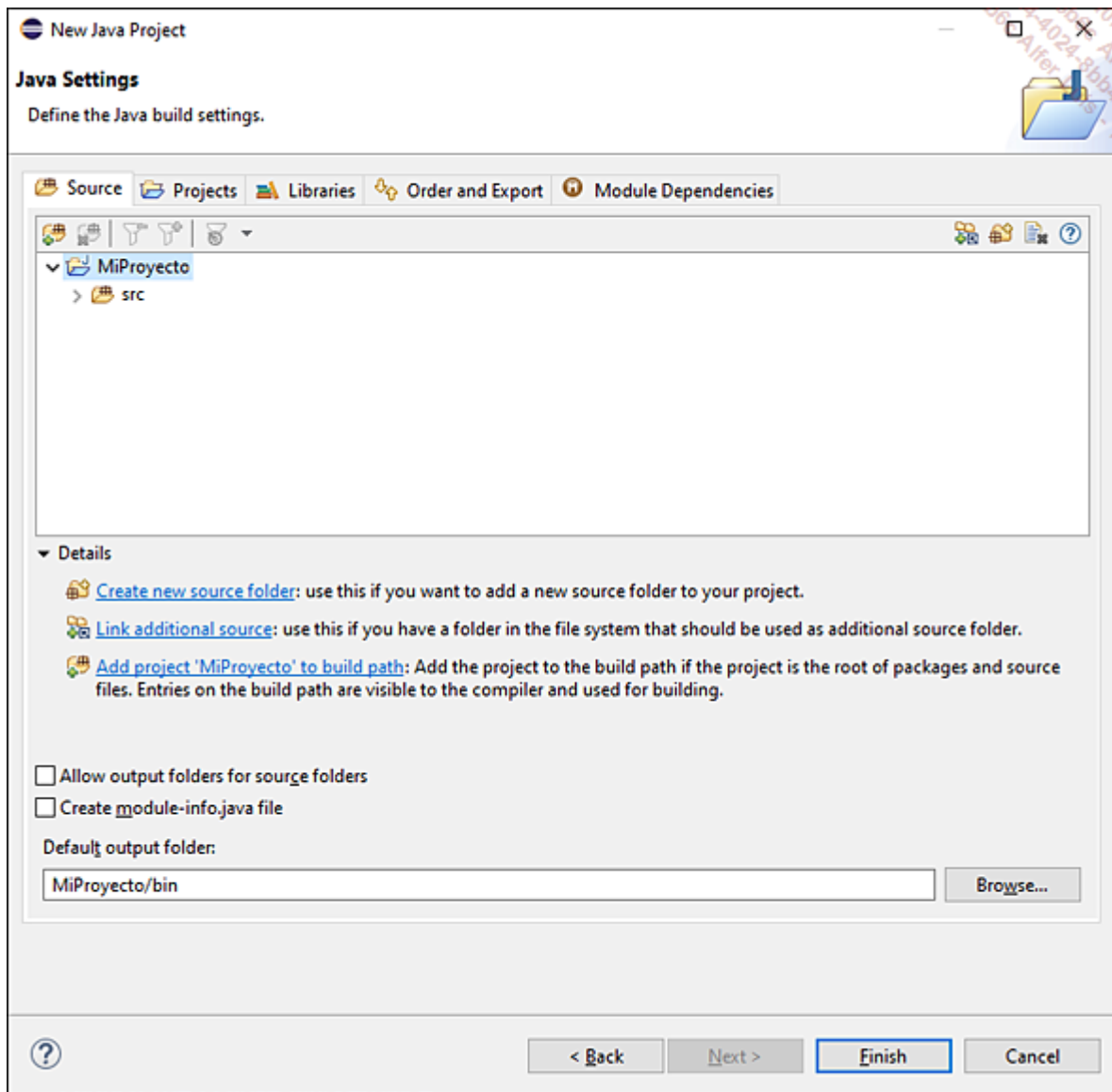
- Pulse el menú **File - New - Java Project** para ver la pantalla de ayuda durante la creación del proyecto.



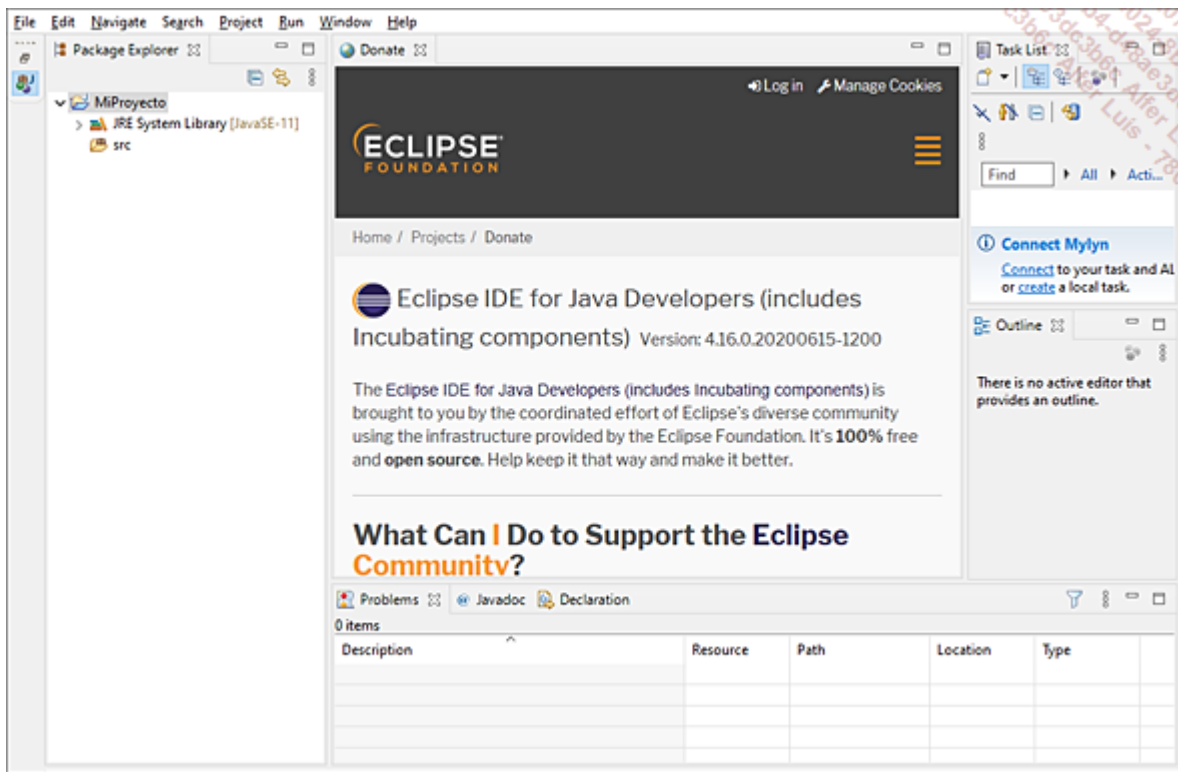
- indique un nombre (**Project name**) para su proyecto y conserve el resto de las opciones con los valores predefinidos. Así el proyecto se creará en su workspace y el JDK

utilizado se corresponderá con el seleccionado durante la instalación de Eclipse. Por supuesto, es posible modificar todos estos valores.

- Pulse el botón **Next** para acceder a las características avanzadas de su proyecto:



- Desmarque la casilla **Create module-info.java file**. Esta noción avanzada se abordará más adelante en el libro.
- Pulse el botón **Finish** y su proyecto se crea:



- Ahora es posible crear su programa de Java en el directorio **src**. El directorio **JRE System Library** hace referencia a las API disponibles para el JDK seleccionado.

Para más información sobre la creación de un primer programa y sobre su ejecución, vaya a la sección Primer programa, un poco más adelante en este capítulo.

4. Instalación de MySQL

a. Windows

El objetivo de esta sección es presentar la instalación del servidor MySQL y MySQL Workbench, una herramienta gráfica que permite administrar el servidor MySQL.

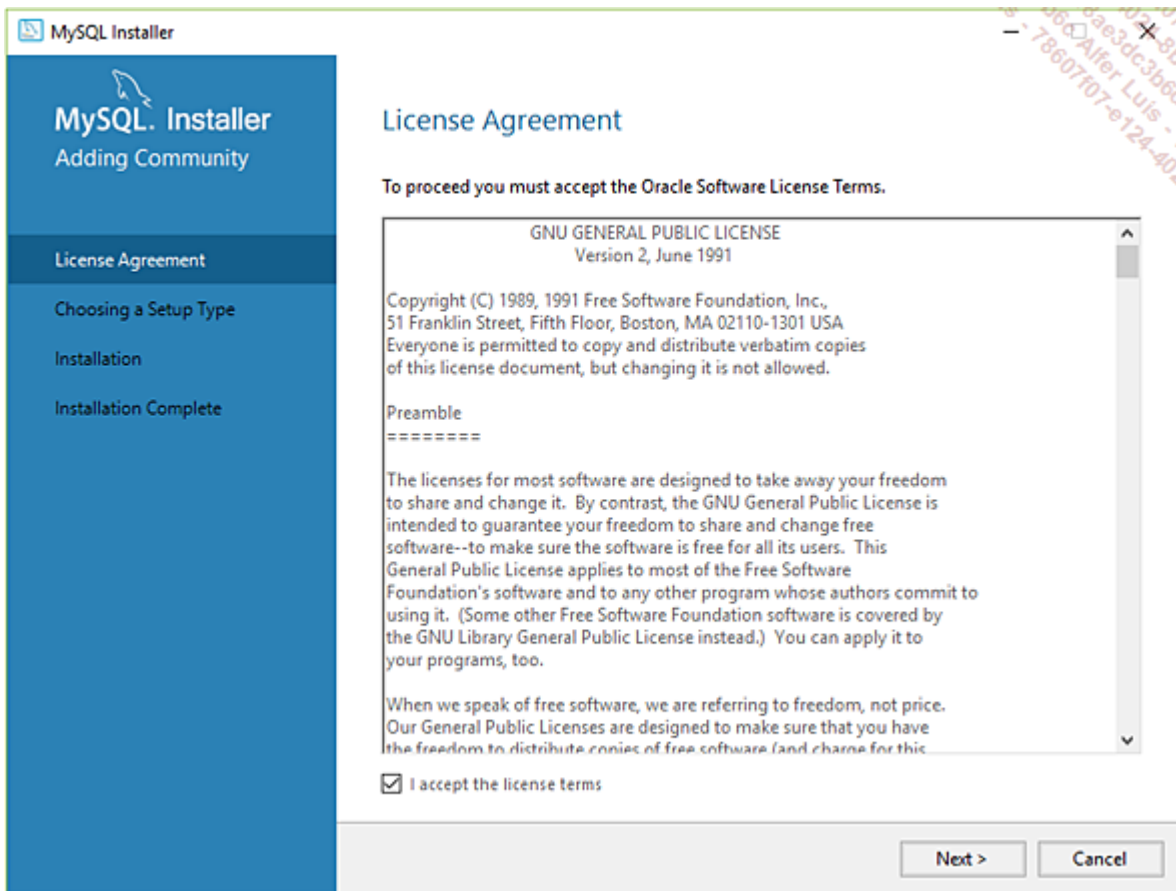
Todo comienza con la descarga del instalador en la siguiente dirección:

<https://dev.mysql.com/downloads/windows/installer/>

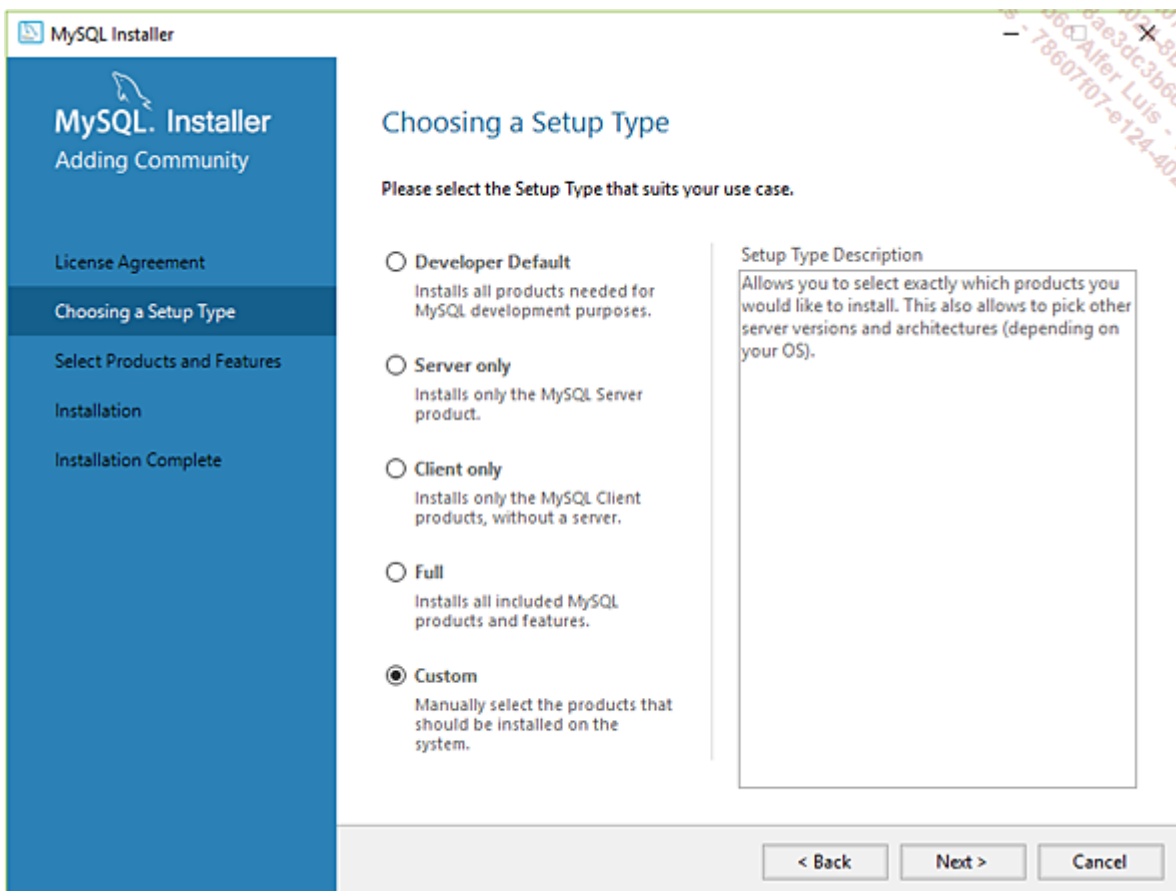
Hay dos instaladores disponibles. El primero requiere una conexión a Internet durante la instalación.

Cuando se descargue el instalador de su elección, haga doble clic en él para iniciar la instalación. El resto de la sección se basa en el instalador que requiere una conexión a Internet. Solo descargará las herramientas elegidas.

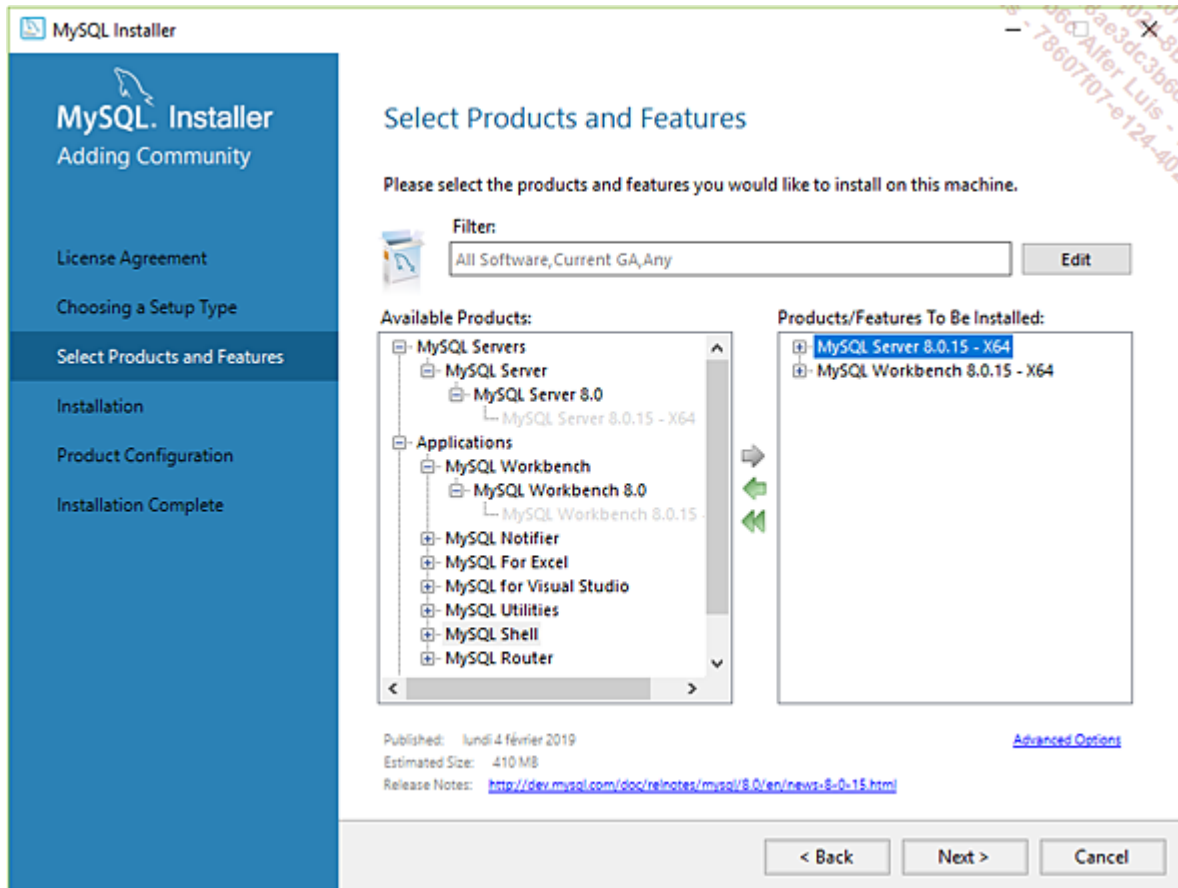
La pantalla de bienvenida de la instalación es la siguiente:



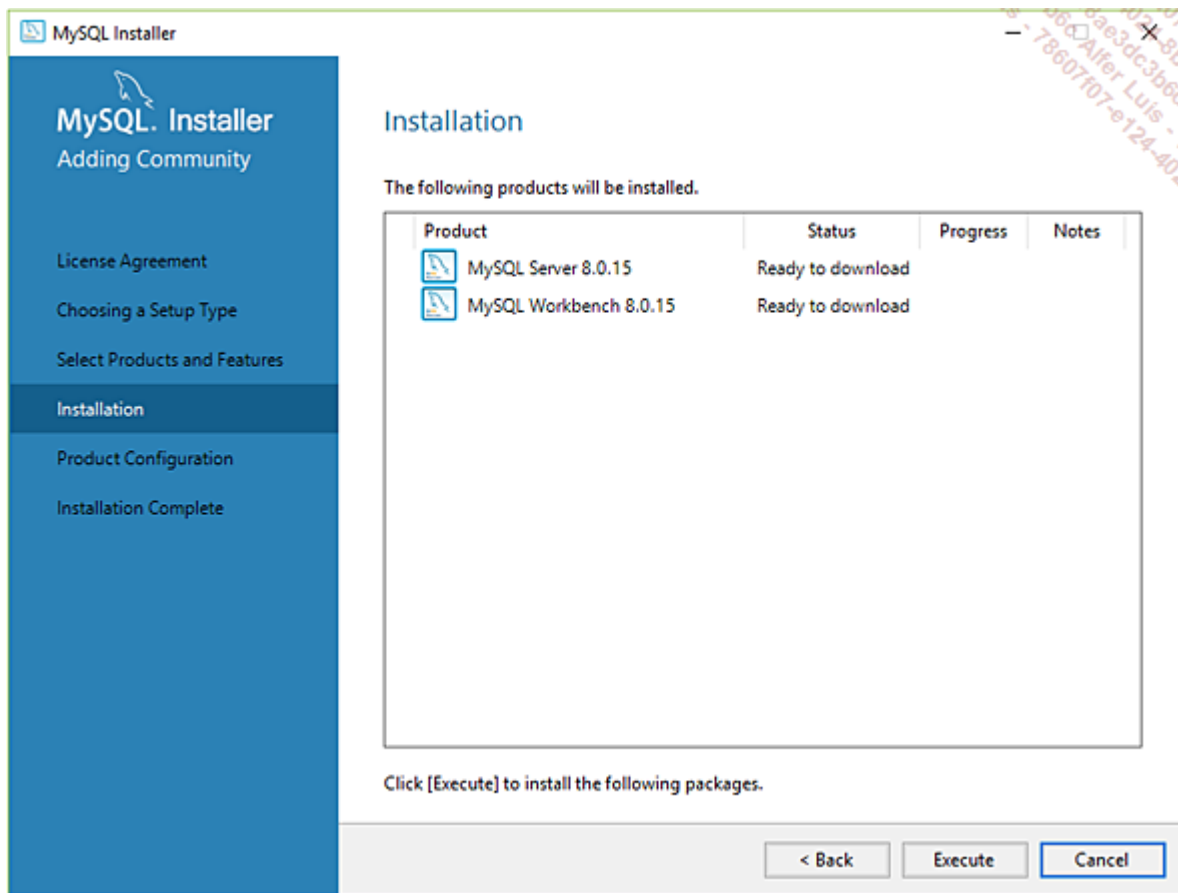
- Marque la casilla para aceptar los términos de la licencia y pulse el botón **Next** para continuar la instalación.



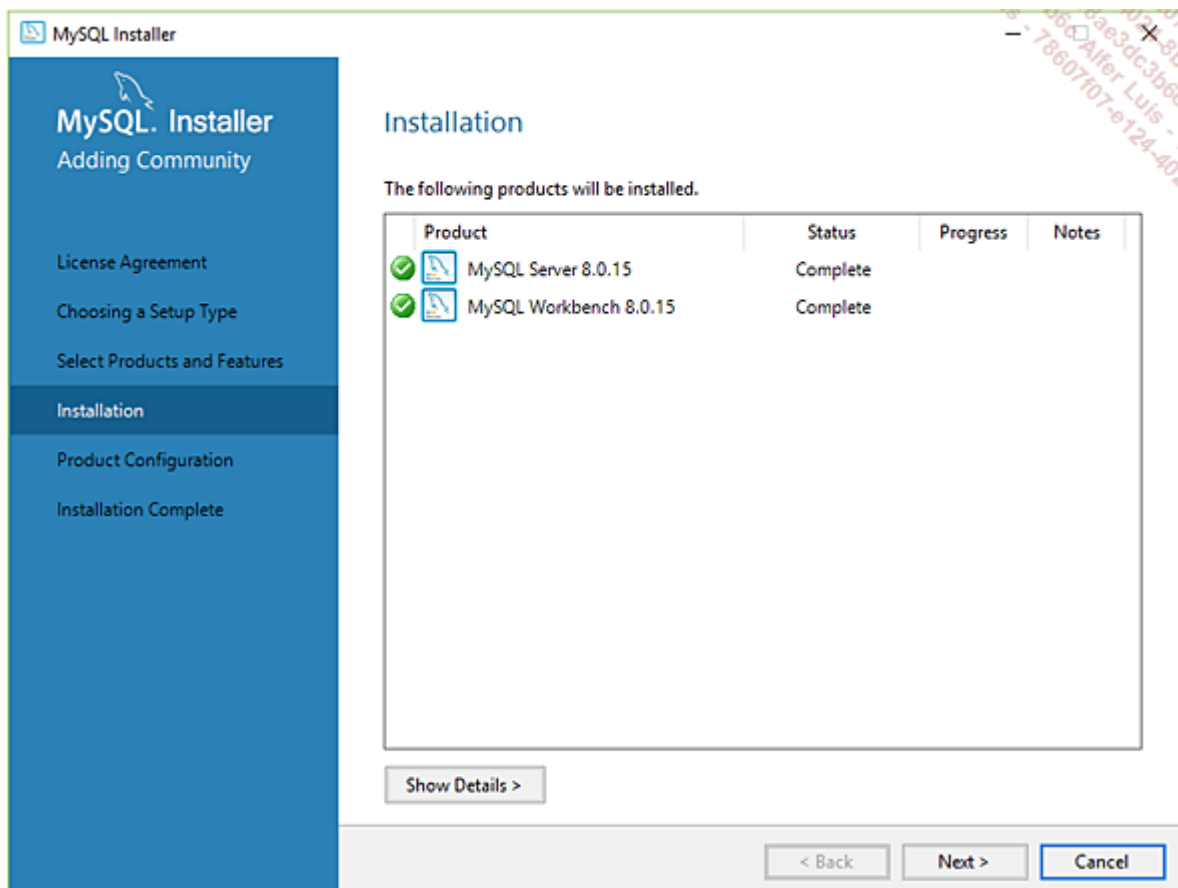
Seleccione una instalación personalizada (**Custom**) para instalar solo las herramientas de su elección.



- Seleccione el servidor **MySQL Server 8.0** y **MySQL Workbench 8.0**. La versión importa poco en el marco de este libro. Pulse el botón **Next** para continuar la instalación.



- Un resumen indica las herramientas que se van a descargar. Pulse el botón **Execute** para que se descarguen e instalen de golpe. Sea paciente durante algunos minutos para que la operación termine.



- Esta pantalla indica el éxito de la instalación. Pulse el botón **Next** para arrancar la configuración de MySQL.
- Van a sucederse varias pantallas. Deje en cada etapa los valores definidos por defecto y pulse el botón **Next** hasta llegar a la siguiente pantalla:

MySQL Installer

MySQL Server 8.0.15

Group Replication

Type and Networking

Authentication Method

Accounts and Roles

Windows Service

Apply Configuration

Accounts and Roles

Root Account Password
Enter the password for the root account. Please remember to store this password in a secure place.

MySQL Root Password:

Repeat Password:

Password strength: **Medium**

MySQL User Accounts
Create MySQL user accounts for your users and applications. Assign a role to the user that consists of a set of privileges.

MySQL User Name	Host	User Role

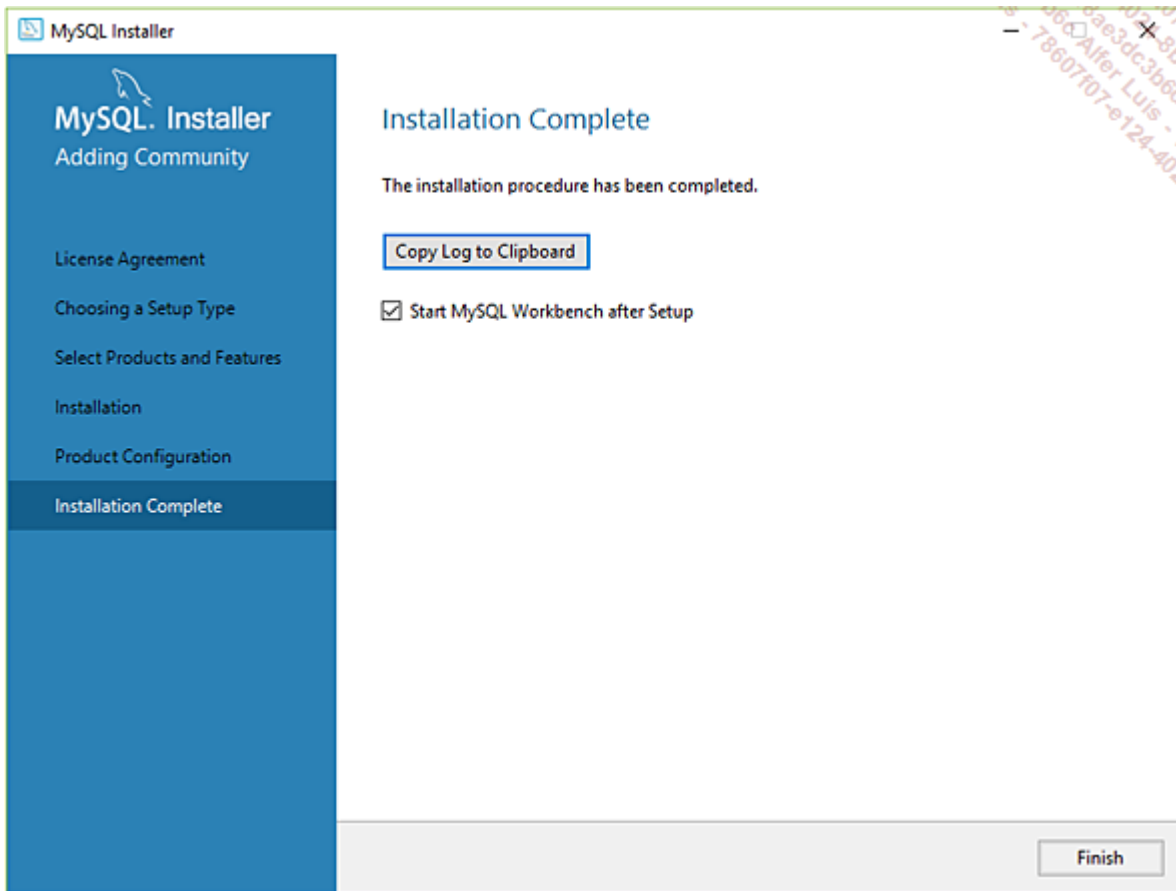
Add User

Edit User

Delete

< Back Next > Cancel

- Defina la contraseña del administrador (**MySQL Root Password**). En el marco de la demostración, se ha definido con el valor Pa\$\$w0rd.
- Continúe pulsando el botón **Next** hasta llegar a esta pantalla:



- ❑ Finalmente, haga clic en **Finish** y la instalación habrá finalizado.
- ❑ Modifique la variable `PATH` para agregar la ruta de acceso correspondiente al directorio que contiene el ejecutable `mysql.exe`. La ruta por defecto para la versión instalada es:

`C:\Program Files\MySQL\MySQL Server 8.0\bin`

Creación de un usuario y una base de datos

Conéctese a la aplicación de consola MySQL como superadministrador ejecutando el siguiente código en su línea de comandos (símbolo del sistema):

```
mysql -u root -p
```

- La opción `-u` permite definir la cuenta que se debe utilizar (`root`).
- La opción `-p` permite indicar que se debe indicar una contraseña para la validación de la instrucción (`Password` en la configuración realizada anteriormente).

En la línea de comandos puede:

- crear un nuevo usuario MySQL (mysqluser/Pa\$\$w0rd),
- crear una nueva base de datos (demo_java),
- dar permisos extendidos a este usuario en la base de datos,

con las instrucciones siguientes:

```
create user 'mysqluser'@'localhost' IDENTIFIED WITH  
mysql_native_password BY 'Pa$$w0rd';
```

```
create database demo_java;  
grant all on demo_java.* TO 'mysqluser'@'localhost';
```

Para usar MySQL Workbench, vaya a la sección Uso de MySQL Workbench, un poco más adelante en el capítulo.

b. Linux

Instalación del servidor MySQL

La instalación del servidor MySQL se hace ejecutando el comando siguiente:

```
sudo apt install mysql-server
```

La cuenta de administrador de Linux es automáticamente la de superadministrador del servidor MySQL. Para más información, consulte la página en la siguiente dirección:

<https://ubunlog.com/mysql-8-base-datos-ubuntu/>

Creación de un usuario y de una base de datos

Conéctese a la aplicación de consola MySQL como superadministrador, ejecutando el siguiente comando en su terminal:

```
sudo mysql
```

En línea de comandos, puede:

- crear un nuevo usuario MySQL (mysqluser/Pa\$\$w0rd),
- crear una nueva base de datos (demo_java),

- dar permisos a este usuario en la base de datos,

con las instrucciones siguientes:

```
create user 'mysqluser'@'localhost' IDENTIFIED WITH  
mysql_native_password BY 'Pa$$w0rd';
```

```
create database demo_java;  
grant all on demo_java.* TO 'mysqluser'@'localhost';
```

Para más información, consulte:

- la documentación de Ubuntu, que presenta los principales comandos de MySQL, en la dirección: <https://ubunlog.com/mysql-base-datos-ubuntu>,
- la documentación oficial de MySQL en la siguiente dirección: <https://dev.mysql.com/doc/refman/5.7/en/>

Instalación de MySQL Workbench

MySQL Workbench es una herramienta gráfica para administrar el servidor MySQL. Se adapta perfectamente durante la fase de desarrollo para manejar una base de datos.

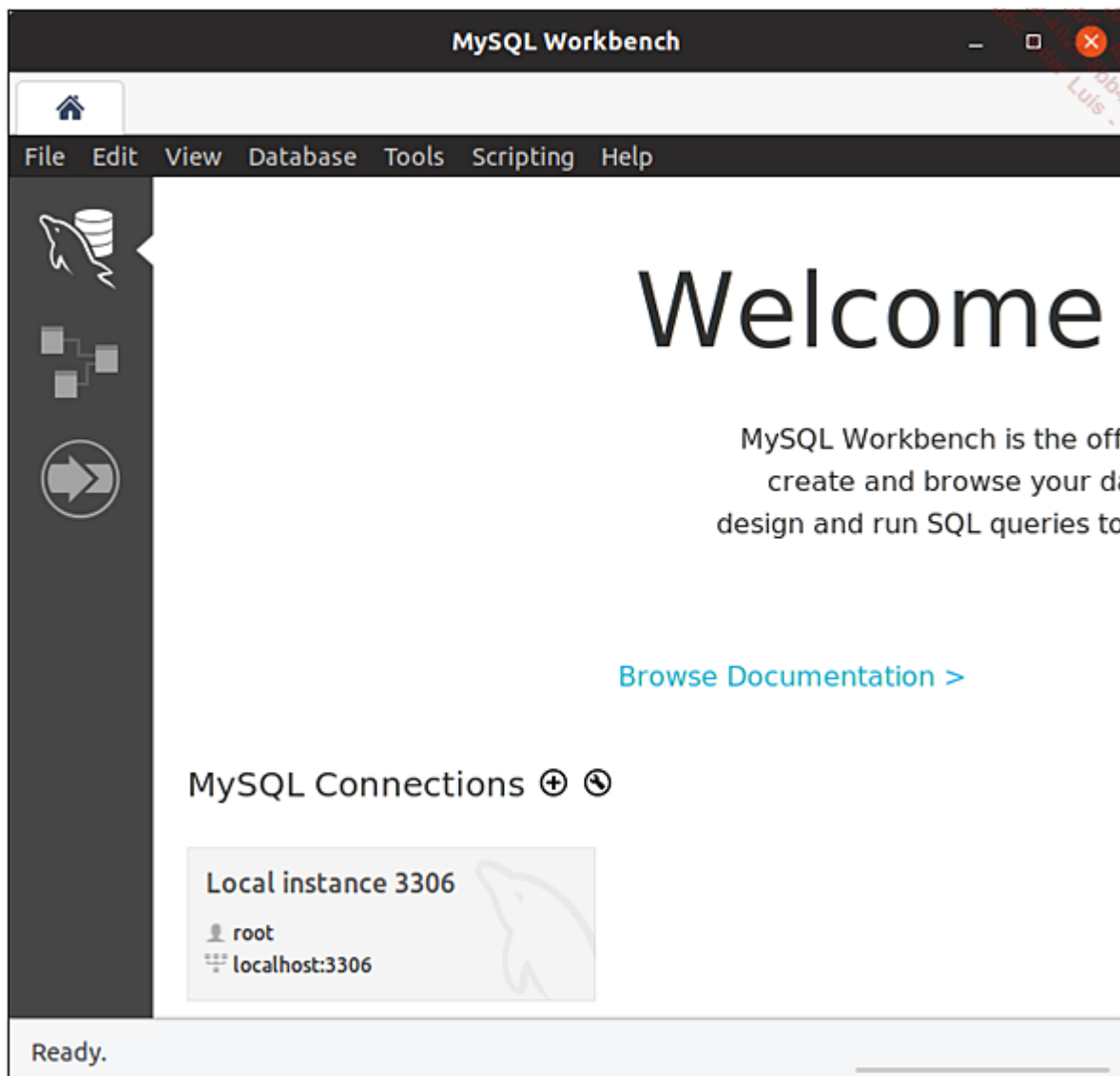
La instalación es posible a través de las siguientes instrucciones:

```
sudo apt install mysql-workbench
```

c. Uso de MySQL Workbench

Para manipular la base de datos `demo_java` a través de MySQL Workbench, debe seguirse este procedimiento:

- Inicie MySQL Workbench para ver la siguiente página de inicio:



- Durante la primera puesta en marcha, pulse el botón + a la derecha del título **MySQL Connections** para crear una nueva conexión al servidor MySQL, utilizando la cuenta `mysqluser` creada anteriormente.

La siguiente pantalla permite definir las características de la conexión:

Setup New Connection

Connection Name: Type a name for the connection

Connection Method: Standard (TCP/IP) Method to use to connect to the RDBMS

Parameters | SSL | Advanced

Hostname: Port: Name or IP address of the server host - and TCP/IP port.

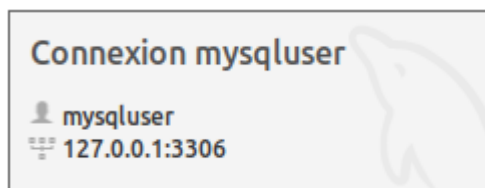
Username: Name of the user to connect with.

Password: Store in Keychain ... Clear The user's password. Will be requested later if it's not set.

Default Schema: The schema to use as default schema. Leave blank to select it later.

Configure Server Management... Test Connection Cancel OK

- Indique un nombre para la conexión (**Connection Name**) y el usuario (**Username**). Termine pulsando el botón **OK**.
- La página de inicio se actualiza y ofrece un nuevo marco para conectarse con la configuración anterior:



- Pulse este marco; aparece la siguiente pantalla:

Connect to MySQL Server

Please enter password for the following service:

Service: Mysql@127.0.0.1:3306

User: mysqluser

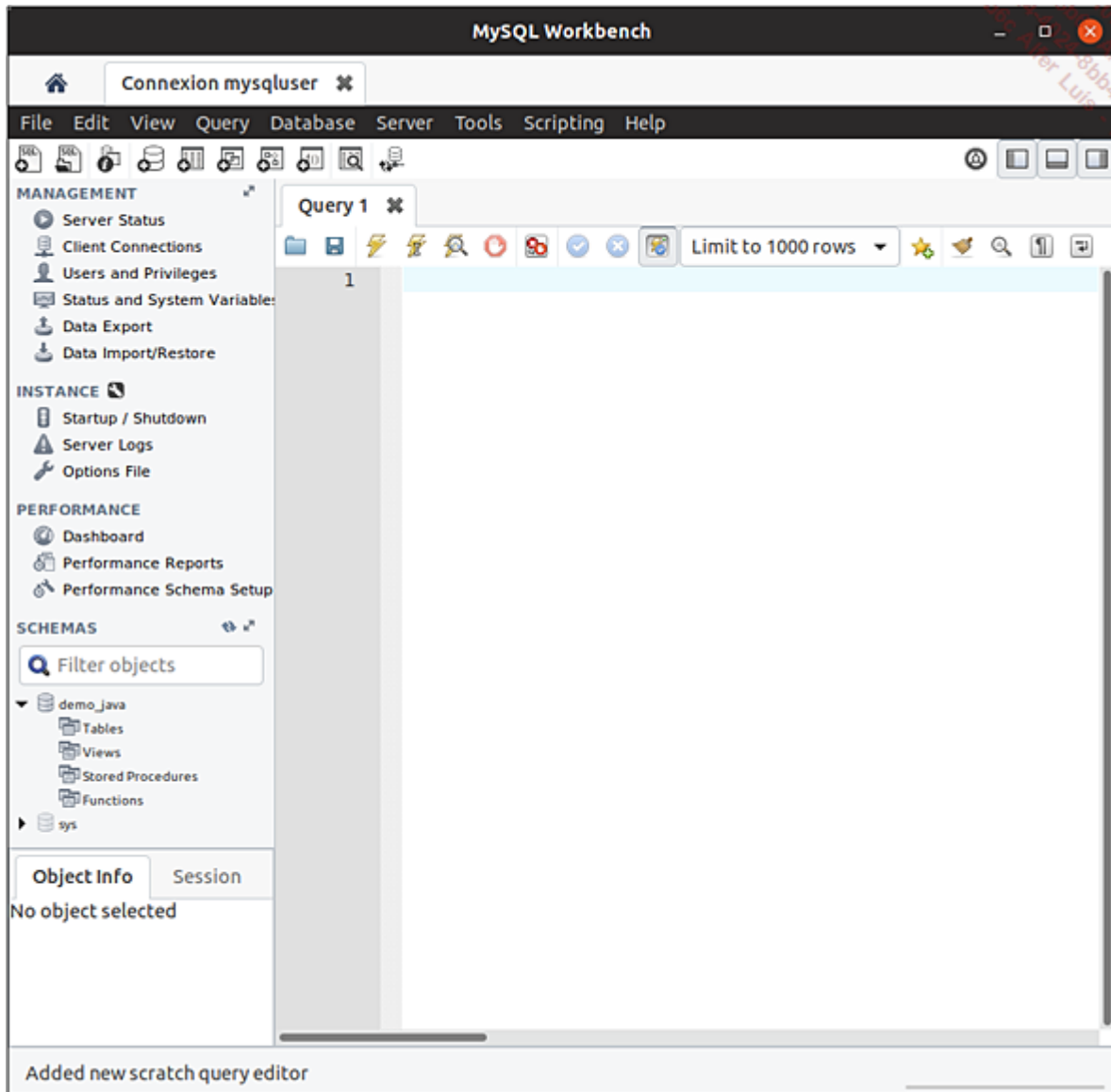
Password:

☐ Save password in keychain

OK Cancel

- Indique la contraseña de la cuenta `mysqluser` (en el área **Password**) y confirme

haciendo clic en el botón **OK**. Ahora está conectado y aparece la siguiente pantalla:



- Observe la presencia de la base de datos `demo_java` en el área de la izquierda. El editor ofrece por defecto un archivo SQL para escribir sus instrucciones.

Esta herramienta se utiliza en el capítulo Acceso a las bases de datos, un poco más adelante en este libro.

La javadoc

La plataforma Java SE viene con contenido muy popular: javadoc.

Javadoc es esencial para cualquier persona que quiera escribir un programa en Java. Proporciona documentación exhaustiva de todas las clases, interfaces, etc., que constituyen la API de Java. Esto permite al desarrollador hallar respuestas a sus dudas.

Es posible consultar la javadoc desde un navegador o desde un EDI como Eclipse.

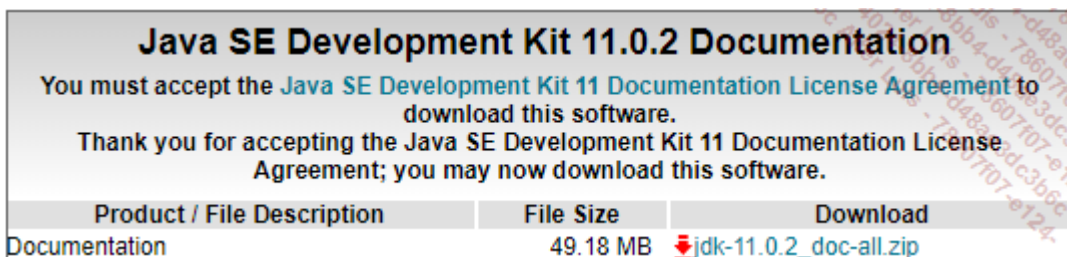
1. Consulta desde un navegador

La javadoc es accesible desde la siguiente dirección:

<https://docs.oracle.com/en/java/javase/11/docs/api>

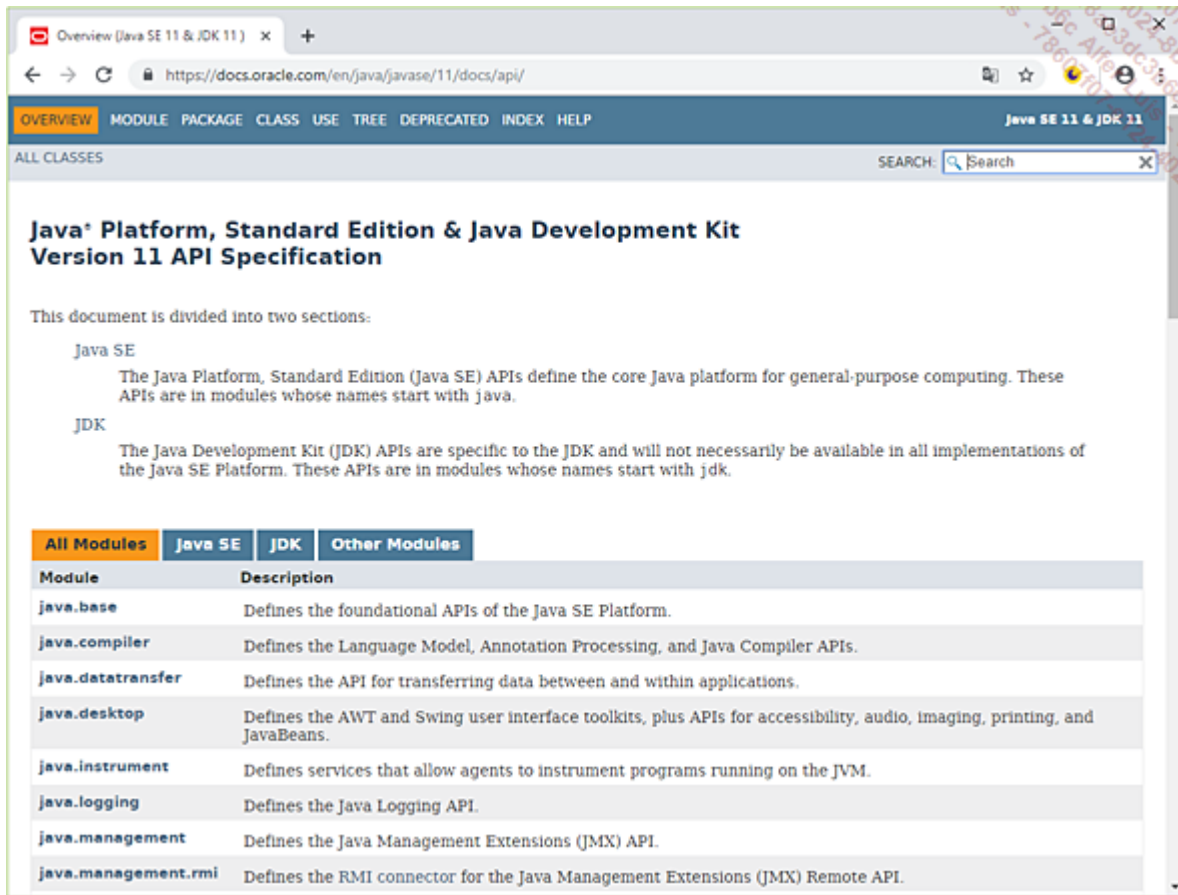
Es posible descargarla en la siguiente dirección para acceder en local:

<https://www.oracle.com/technetwork/java/javase/documentation/jdk11-doc-downloads-5097203.html>.



- ☐ Acepte los términos de la licencia.
- ☐ Descargue el archivo.
- ☐ Descomprímalo en el directorio de su elección.
- ☐ Sitúese en el subdirectorio `/docs/api` y haga doble clic en el archivo `index.html`.

La página de bienvenida es la siguiente:



Esta página lista el conjunto de módulos que componen la plataforma Java SE. Estos módulos se agrupan en dos categorías:

- Los módulos de la plataforma Java SE (que constituyen la API de Java). El número de estos módulos debe comenzar con la palabra `java`. El contenido de este libro se basará en parte de esta API.
- Los módulos adicionales del JDK (que constituyen la API JDK). El número de estos módulos debe comenzar con la palabra `jdk`. El contenido de este libro no abordará esta API.

Para más información, consulte la sección La plataforma Java SE, ya trabajada en este capítulo.

La API Java está compuesta por 21 módulos. La página de inicio de javadoc tiene un enlace a cada uno de estos módulos. El primer módulo de la lista, `java.base`, es el módulo más importante. Contiene la API fundamental de la plataforma Java SE. Al hacer clic en el enlace **java.base**, accede a la descripción de este módulo:

The screenshot shows the Oracle Java SE 11 & JDK 11 API documentation page for the `java.base` module. The page is titled "Module java.base" and describes the foundational APIs of the Java SE Platform. It includes sections for Providers, Tool Guides, and a Module Graph. The "Packages" section is expanded, showing a table of exports.

Module java.base

Defines the foundational APIs of the Java SE Platform.

Providers:
The JDK implementation of this module provides an implementation of the `Jrt` file system provider to enumerate and read the class and resource files in a run-time image. The `Jrt` file system can be created by calling `FileSystems.newFileSystem(URI.create("jrt:/"))`.

Tool Guides:
`java` launcher, `keytool`

Module Graph:

`java.base`

Since:
9

Packages

Exports

Package	Description
<code>java.io</code>	Provides for system input and output through data streams, serialization and the file system.
<code>java.lang</code>	Provides classes that are fundamental to the design of the Java programming language.
<code>java.lang.annotation</code>	Provides library support for the Java programming language annotation facility.

Cada módulo se divide en paquetes. Cada paquete incluye un conjunto de clases, interfaces, etc., de un área funcional específica. Es posible diseñar módulos y paquetes como mecanismos para organizar clases, interfaces, etc.

El módulo `java.base` consta de 51 paquetes. El paquete más icónico es `java.lang`. Contiene el conjunto de clases fundamentales utilizadas en el conjunto de programas escritos en Java. Al hacer clic en el enlace **java.lang**, accede a la descripción de este paquete:

The screenshot shows the Oracle Java SE 11 & JDK 11 API documentation for the `java.lang` package summary. The browser address bar shows the URL `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/package-summary.html`. The navigation bar includes tabs for OVERVIEW, MODULE, PACKAGE (selected), CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. The page title is "Module java.base" and "Package java.lang". The main content area provides a description of the package, stating it provides classes fundamental to the design of the Java programming language. It mentions that `Object` is the root of the class hierarchy and `Class` represents classes at run time. It also discusses wrapper classes (`Boolean`, `Character`, `Integer`, `Long`, `Float`, and `Double`), the `Math` class, and the `ClassLoader`, `Process`, `ProcessBuilder`, `Runtime`, `SecurityManager`, and `System` classes. A section titled "Character Encodings" describes the `java.nio.charset.Charset` class. The "Since" section indicates the package is available since Java 1.0. At the bottom, there is a table with two columns: "Interface" and "Description".

Module java.base

Package java.lang

Provides classes that are fundamental to the design of the Java programming language. The most important classes are `Object`, which is the root of the class hierarchy, and `Class`, instances of which represent classes at run time.

Frequently it is necessary to represent a value of primitive type as if it were an object. The wrapper classes `Boolean`, `Character`, `Integer`, `Long`, `Float`, and `Double` serve this purpose. An object of type `Double`, for example, contains a field whose type is `double`, representing that value in such a way that a reference to it can be stored in a variable of reference type. These classes also provide a number of methods for converting among primitive values, as well as supporting such standard methods as `equals` and `hashCode`. The `Void` class is a non-instantiable class that holds a reference to a `Class` object representing the type `void`.

The class `Math` provides commonly used mathematical functions such as sine, cosine, and square root. The classes `String`, `StringBuffer`, and `StringBuilder` similarly provide commonly used operations on character strings.

Classes `ClassLoader`, `Process`, `ProcessBuilder`, `Runtime`, `SecurityManager`, and `System` provide "system operations" that manage the dynamic loading of classes, creation of external processes, host environment inquiries such as the time of day, and enforcement of security policies.

Class `Throwable` encompasses objects that may be thrown by the `throw` statement. Subclasses of `Throwable` represent errors and exceptions.

Character Encodings

The specification of the `java.nio.charset.Charset` class describes the naming conventions for character encodings as well as the set of standard encodings that must be supported by every implementation of the Java platform.

Since:
1.0

Interface Summary

Interface	Description

Después de una descripción del paquete, se enumera el conjunto de clases, interfaces, etc. Es suficiente hacer clic en el enlace de uno de ellos para obtener una descripción detallada. Al hacer clic en el enlace **String**, se accede a la documentación de la clase `String`:

The screenshot shows the Oracle Java SE 11 & JDK 11 API documentation for the `String` class. The browser address bar shows the URL `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html`. The navigation bar includes tabs for OVERVIEW, MODULE, PACKAGE, CLASS (selected), USE, TREE, DEPRECATED, INDEX, and HELP. The page title is "Module java.base" and "Package java.lang". The main content area provides a description of the `String` class, stating it represents character strings. It mentions that all string literals in Java programs, such as `"abc"`, are implemented as instances of this class. It also discusses the immutability of `String` objects and provides examples of how strings can be used. The "All Implemented Interfaces" section lists `Serializable`, `CharSequence`, and `Comparable<String>`. The "public final class String" section shows that `String` extends `Object` and implements `Serializable`, `Comparable<String>`, and `CharSequence`. The "The String class represents character strings" section explains that all string literals in Java programs, such as `"abc"`, are implemented as instances of this class. The "Strings are constant" section states that their values cannot be changed after they are created. The "Here are some more examples of how strings can be used" section provides code snippets for creating and using `String` objects.

Module java.base

Package java.lang

Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:
`Serializable`, `CharSequence`, `Comparable<String>`

```
public final class String
    extends Object
    implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because `String` objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

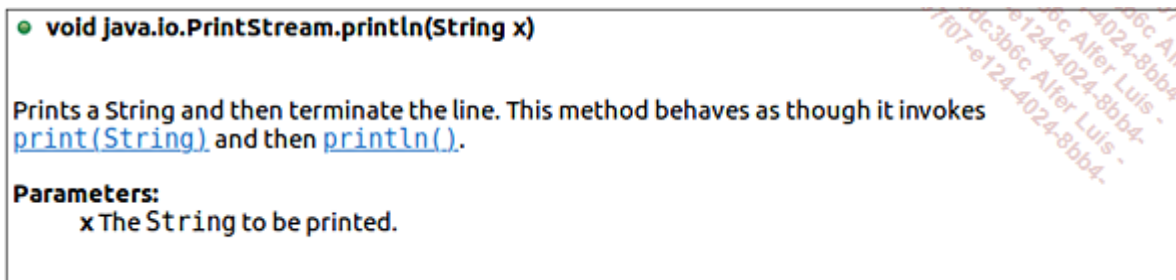
```
System.out.println("abc");
String cde = "cde";
```

Después de una presentación general de la clase que indica principalmente el módulo y el paquete al que pertenecen, se muestra un resumen que presenta el uso de la clase. Este resumen es más o menos consistente según la clase. Luego, se describe el conjunto de elementos que constituyen la clase: variables miembro, constructores, métodos, etc. Por lo tanto, es posible comprender el funcionamiento y el comportamiento de cada método. Es de gran ayuda para escribir un buen programa.

Sin embargo, el uso de javadoc a través de un simple navegador puede ser tedioso. Es importante tener acceso instantáneo a la documentación de un método o clase cuando desee usarlo. Afortunadamente, los EDI como Eclipse han integrado la documentación en el entorno de desarrollo, a través del acceso rápido y automático a la javadoc.

2. Consulta desde Eclipse

Es posible acceder a la documentación de una clase o un método deteniendo el puntero del ratón un instante en el elemento. Esta acción tiene como consecuencia el acceso a la documentación del elemento de la javadoc, en forma de ventana emergente:



El pop-up que acabamos de mostrar presenta en este caso la documentación del método `println` disponible en la clase `PrintStream` del paquete `java.io`.

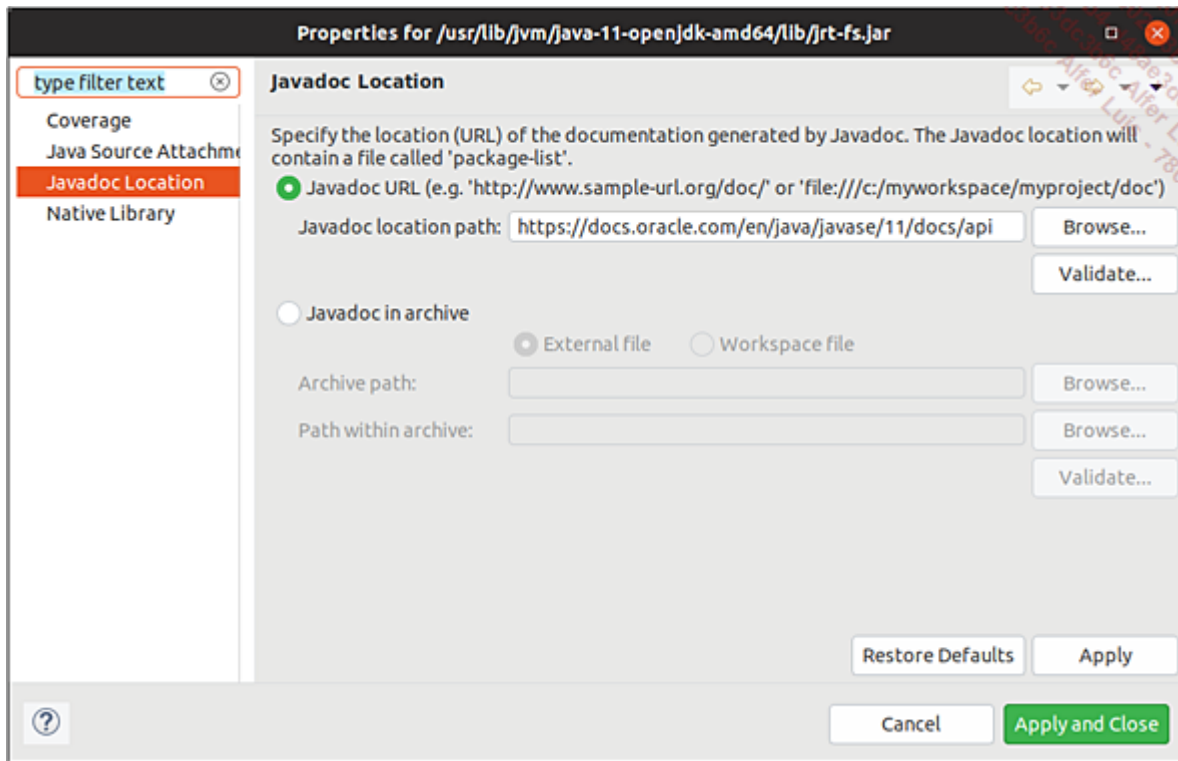
Para que esto funcione, es esencial que la javadoc esté configurada en el EDI. En Eclipse, la configuración se realiza con los módulos de la plataforma Java SE, referenciados en un proyecto Java. Para esto, es suficiente con desplegar el nodo **JRE System Library** del proyecto:



Todos los módulos están representados. La captura de pantalla anterior presenta solo los tres primeros.

A fin de asegurarse de que la configuración de la documentación sea correcta, siga estos pasos para cada uno de los módulos útiles para el programa. Recuerde que el módulo **java.base** es un módulo esencial.

- Visualice las propiedades del módulo haciendo clic derecho en él y seleccionando el menú **Properties**. Se muestra la siguiente pantalla:



- Sitúese en la pestaña **Javadoc Location** y asegúrese de que la ruta de acceso a la javadoc (**Javadoc location path**) esté bien configurada. Si usa la javadoc en línea, el valor debe ser <https://docs.oracle.com/en/java/javase/11/docs/api>. Si ha descargado la javadoc y quiere acceder a ella, pulse el botón **Browse...** para seleccionar los directorios `/docs/api` de su directorio de descompresión de la documentación de Java.
- Pulse el botón **Apply and Close** para finalizar la configuración.

Primer programa

Es tentador crear su primer programa usando Eclipse u otro EDI de su elección. Sin embargo, para comprender completamente los mecanismos de creación de un programa, es necesario llevar a cabo los diferentes pasos de forma manual, antes de delegar este trabajo a una herramienta de terceros.

1. Escritura del código

Un programa de Java es un conjunto de archivos de texto de extensión `.java`. El programa más pequeño contiene un único archivo de Java.



El nombre de un archivo de Java se corresponde con el nombre de la clase de Java que contiene.

El archivo `PrimerPrograma.java` contiene la declaración de una clase llamada `PrimerPrograma`. La estructura de este archivo es la siguiente:

```
public class PrimerPrograma
{
    ...
}
```

 Atención, Java es sensible a la diferencia entre mayúsculas y minúsculas.

Un programa de Java debe disponer de un método particular que se corresponde con el punto de entrada del programa. Este método se llama `main` y tiene obligatoriamente la siguiente estructura:

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

Dentro de este método, es posible escribir el programa.

El programa más clásico consiste en mostrar un mensaje. La instrucción para realizar esta tarea es la siguiente:

```
System.out.println("Hola a todo el mundo");
```

El primer programa va a tener la estructura global siguiente:

```
public class PrimerPrograma
{
    public static void main(String[] args)
    {
        System.out.println("Hola a todo el mundo");
    }
}
```

2. Compilación y ejecución del código

a. Teoría

Para que el programa de Java se pueda ejecutar, debe pasar las etapas de compilación y ejecución.

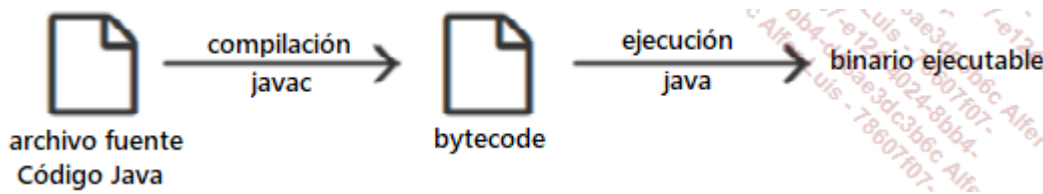
La compilación transforma el código fuente de Java en *bytecode*, un lenguaje intermedio. Este código es un código verificado sintácticamente por el compilador `javac` disponible en la estación de trabajo del desarrollador que instaló el JDK.

Este código no es binario, no es ejecutable directamente por la máquina.

Para eso, debe usar la herramienta `java` disponible en la máquina del desarrollador que instaló el JDK o en la máquina de destino que solo instaló el JRE. La herramienta `java`

tiene la función de iniciar el JVM que carga el *bytecode* y lo compila sobre la marcha (compilación JIT - *Just In Time*) para obtener un binario ejecutable directamente en la máquina.

El siguiente diagrama presenta una vista resumida de estas diferentes etapas.



b. Implementación

La compilación

La compilación se realiza sobre el puesto del desarrollador con ayuda del compilador `javac`.

El comando para compilar un programa compuesto de un único archivo es el siguiente:

```
javac NOMBRE_DEL_ARCHIVO_JAVA
```

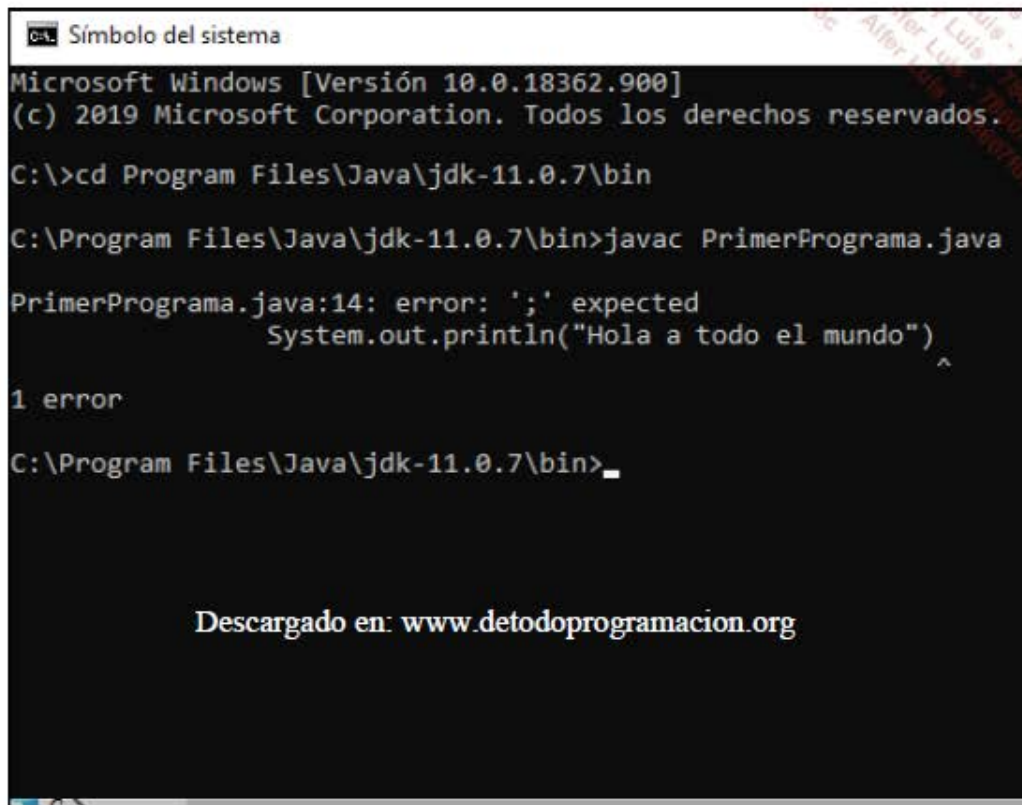
Ejemplo:

```
javac PrimerPrograma.java
```

☐ Esta instrucción funciona si su **línea de comandos** (Windows) o **Terminal** (Linux) está situada en el directorio que contiene el archivo `PrimerPrograma.js`. Si este no es el caso, utilice el comando `cd` para acceder a él.

☐ Solo usaré el término **terminal** a partir de ahora.

En caso de éxito, no se muestra ningún mensaje. En caso de problemas, la información detallada se muestra para ayudarle a resolverlo:

A screenshot of a Windows command prompt window. The title bar reads 'Símbolo del sistema'. The text inside shows the following commands and output:
Microsoft Windows [Versión 10.0.18362.900]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.
C:\>cd Program Files\Java\jdk-11.0.7\bin
C:\Program Files\Java\jdk-11.0.7\bin>javac PrimerPrograma.java
PrimerPrograma.java:14: error: ';' expected
 System.out.println("Hola a todo el mundo")
 ^
1 error
C:\Program Files\Java\jdk-11.0.7\bin>
At the bottom, there is a watermark: 'Descargado en: www.detodoprogramacion.org'.

El ejemplo anterior indica que falta un punto y coma en la línea 14 del archivo `PrimerPrograma.java`.

Cuando el archivo se compila con éxito, la herramienta `javac` genera, por defecto en el mismo directorio, un archivo `PrimerPrograma.class`. Este archivo contiene el *bytecode*, un lenguaje intermedio explotable por la JVM durante la ejecución.

La ejecución

Descargado en: www.detodoprogramacion.org

La ejecución se realiza sobre el puesto del desarrollador o en el puesto de destino, gracias a la herramienta `java`. Solo se usan los archivos `.class` durante la ejecución. No es necesario conservar los archivos fuente (`.java`) en el entorno de ejecución.

El comando es el siguiente si su terminal está situado en el directorio que contiene el archivo `.class`:

```
java NOMBRE_DE_LA_CLASE
```

El nombre de la clase se corresponde con el nombre de la clase que contiene el método `main`.

En el marco del ejemplo actual, el comando es el siguiente:

```
java PrimerPrograma
```

El terminal muestra el resultado siguiente:

```
Hola a todo el mundo
```

La variable CLASSPATH

Para ejecutar el programa sin colocar el terminal en su directorio, es suficiente con crear la variable CLASSPATH, que permite indicar a la herramienta `java` dónde encontrar los archivos para ejecutar.

- En Windows, escriba la siguiente instrucción:

```
set CLASSPATH=DIRECTORIO_DE_LOS_ARCHIVOS_CLASS
```

Si los archivos están en el directorio `C:/Java`, la instrucción es la siguiente:

```
set CLASSPATH=C:\Java
```

Es posible definir varios directorios separándolos por un punto y coma.

```
set CLASSPATH=dir1;di2
```

Para verificar el contenido de la variable, ejecute la instrucción siguiente:

```
echo %CLASSPATH%
```

- En Linux, escriba la instrucción siguiente:

```
export CLASSTPAH=DIRECTORIO_DE _LOS_ARCHIVOS_CLASS
```

Si los archivos están en el directorio `/home/java/Documents/Java`, la instrucción es la siguiente:

```
export CLASSPATH=/home/java/Documents/Java
```

Es posible definir varios directorios separándoles por dos puntos (:).

```
set CLASSPATH=dir1:di2
```

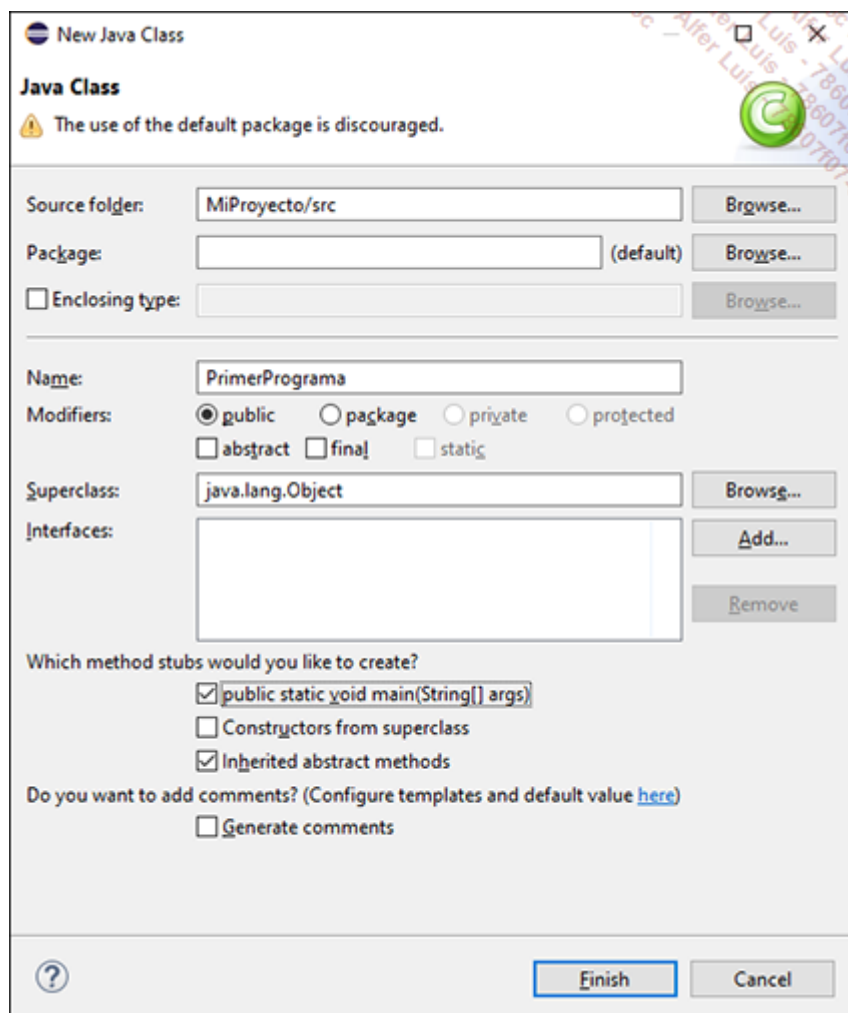
Para verificar el contenido de una variable, ejecute la instrucción siguiente:

```
echo $CLASSPATH
```

3. Utilización de Eclipse

Para escribir un programa a partir de un proyecto de Java creado según las instrucciones de la sección Creación de un proyecto de Java, tratado con anterioridad en este capítulo, siga estas instrucciones:

- Haga un clic con el botón derecho del ratón en el directorio **src** de la vista **Package Explorer**.
- Seleccione el menú **New - Class** para que aparezca la pantalla de ayuda durante la creación de una clase:



- Indique un nombre para su clase en la zona de entrada **Name** y marque la casilla **public static void main(String[] args)** para que el método se cree automáticamente.
- Pulse el botón **Finish** para desencadenar la creación de la clase. Se crea el archivo PrimerPrograma.java con el siguiente contenido:

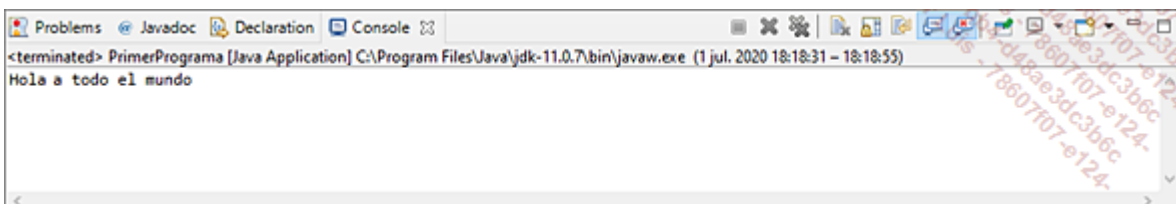
```
public class PrimerPrograma {  
    public static void main(String[] args) {  
        //TODO Auto-generated method stub  
    }  
}
```

El comentario TODO le invita a escribir el contenido del método, indicando que esto es un esqueleto de método autogenerado.

- Cambie el comentario por la línea siguiente:

```
System.out.println("Hola a todo el mundo");
```

- Ejecute el programa realizando un clic con el botón derecho del ratón en el archivo y seleccionando en el menú contextual **Run As - 1 Java Application**.
- El resultado de la ejecución se muestra en la vista **Console**, disponible por defecto en la parte inferior de Eclipse.



Conclusión

Como resultado de la lectura de este capítulo, está listo para abordar con tranquilidad el estudio del lenguaje Java y crear sus primeros programas.

Anatomía de un programa

Estamos acostumbrados a decir que «Todo es un objeto en Java». Esto significa que, para escribir cualquier programa, es necesario crear al menos una clase, cuya estructura debe ser la siguiente:

```
1  public class NombreDeLaClase
2  {
3      //EL CÓDIGO DE SU PROGRAMA
4      public static void main(String[] args)
5      {
6          // EL CÓDIGO DE SU PROGRAMA
7      }
8      // EL CÓDIGO DE SU PROGRAMA
9  }
```

Para obtener más información sobre cómo crear una clase de este tipo, consulte la sección Primer programa del capítulo Presentación.

El punto de entrada de su programa es el método `main`, cuya firma debe ser la presentada en el ejemplo de la línea 4. El código para este método está delimitado por las llaves de apertura y cierre de las líneas 5 y 7.

Este método se incluye en una clase cuya declaración se realiza en la línea 1. El código de clase se encuentra entre las llaves de apertura y cierre de las líneas 2 y 9.

Todos los ejemplos en este capítulo respetarán esta organización.

Las variables

1. Introducción

Permiten almacenar en memoria diferentes valores útiles para su correcto funcionamiento. Antes de usar variables en un programa, necesita saber dónde declararlas, qué nombre y qué tipo asignarles. Este es el objetivo de las siguientes secciones.

2. Las ubicaciones

□ □ Las variables en Java se pueden declarar en cualquier parte del programa. No hay espacios reservados. Dicho esto, solo hay una regla: las variables deben declararse solo entre las llaves que delimitan las clases, interfaces o enumeraciones. No pueden estar fuera de estos elementos.

Se debe declarar obligatoriamente una variable antes de utilizarla en el código. Según la ubicación de su declaración, una variable pertenecerá a una de las categorías siguientes:

- Declarada en el interior de una clase, la variable es una **variable de instancia**. Solo existirá si una instancia de la clase está disponible (dicho de otra manera, un objeto). Cada instancia de clase tendrá su propio ejemplar de la variable.
 - Declarada en el interior de una clase con la palabra clave `static`, una palabra clave del lenguaje, la variable es una **variable de clase**. Se puede acceder a ella directamente por el nombre de la clase y existe en un único ejemplar, independientemente del número de objetos creados a partir de esta clase.
 - Declarada en el interior de un método, la variable es una **variable local**. Solo existe durante la ejecución del método y solo se puede acceder a ella desde el código de esta.
-

- Los **parámetros** de los métodos se pueden considerar como variables locales. La única diferencia reside en la inicialización de la variable efectuada durante la llamada a esta. La estructura del programa permite visualizar la ubicación de sus diferentes categorías de variables:

```
1  public class UbicacionDeLasVariables
2  {
3      //VARIABLES DE INSTANCIA
4      //VARIABLES DE CLASE
5      ... unMetodo(/*PARAMETROS*/)
6      {
7          //VARIABLES LOCALES
8      }
9  }
```

3. El nombre

Debe respetar algunas reglas, incluso si hay un importante grado de libertad para el desarrollador.

- El nombre de una variable empieza obligatoriamente por una letra.
- Puede tener letras, cifras o el carácter de subrayado (_).
- Puede contener un número cualquiera de caracteres (por razones prácticas, es mejor limitarse a un tamaño razonable).
- Se hace una distinción entre minúsculas y mayúsculas (la variable `EDADDELCAPITAN` es diferente a la variable `edaddelcapitan`).
- Las palabras clave del lenguaje no deben utilizarse como nombre de variable.
- Por convenio, el nombre de la variable se escribe con letras minúsculas, salvo la primera letra de cada palabra (a partir de la segunda palabra) si el nombre de la variable incluye varias palabras (`edadDelCapitan`).

4. Los tipos

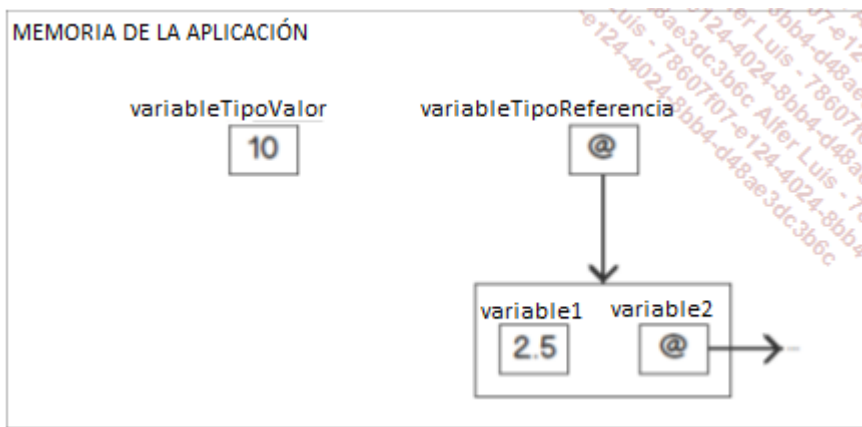
a. Presentación

Al determinar un tipo para una variable, indicamos qué tipo de información vamos a poder almacenar en esta variable y, en consecuencia, las operaciones que podremos efectuar con ella.

Java dispone de dos categorías de tipos de variables:

- Los tipos por valor: la variable contiene realmente la información.
- Los tipos por referencia: la variable contiene la dirección de memoria donde se encuentra la información.

El siguiente diagrama resume este concepto. Representa la memoria de una aplicación y las variables que usa. Es importante dominar este concepto porque tiene impactos que estudiaremos más adelante.



La variable denominada `variableTipoValor` está asociada con un espacio de memoria que contiene directamente el valor. En el ejemplo, el valor es un entero numérico (10).

La variable denominada `variableTipoReferencia` está asociada con un espacio de memoria que contiene una dirección de otro espacio de memoria. Esta dirección se indica con la `@`. En esta dirección hay un espacio de memoria que contiene dos datos: una `variable1` de tipo por valor y una `variable2` de tipo por referencia. Detrás de este mecanismo se esconde la noción de **punteros**, ampliamente utilizados por los desarrolladores de C++ en particular. Es la representación clásica de un objeto.

b. Los tipos por valor

Los tipos por valor representan los tipos de nivel más bajo a partir de los cuales es posible trabajar cualquier información, por compleja que sea. Están representados por los tipos

primitivos.

Java proporciona siete tipos primitivos agrupados en cuatro categorías: números enteros, números decimales, caracteres y booleanos. Con estas cuatro categorías, es posible trabajar cualquier información básica, como:

- La edad de una persona:



- La distancia entre dos ciudades.
- Una letra.
- ...

c. Los tipos por referencia

Cuando la información con la que hay que trabajar es más compleja, una variable de tipo por referencia, es más adecuada. Una variable de tipo por referencia permite agrupar un conjunto de variables que son consistentes entre sí.

Este es el caso, por ejemplo, con una fecha. Una fecha no se puede volver a presentar por un tipo primitivo, ya que se requiere una variable para el año, otra para el mes y una última para el día. Una fecha se representa mediante el uso de tres variables enteras.



Una variable de tipo por referencia simplemente es una variable que hace referencia a un objeto. El próximo capítulo, Programación orientada a objetos, está dedicado a este tema. Sin embargo, los tipos por referencia esenciales se analizan en la siguiente sección de este capítulo (Los tipos por referencia) para abordar la manipulación de arrays, cadenas de caracteres y fechas.

5. La declaración, la inicialización y la asignación

a. La declaración

Ahora que se conocen las categorías de variables, es hora de centrarse en declarar variables. La declaración de una variable debe respetar la siguiente estructura:

```
[modificador] tipo nombreDeLaVariable;
```

Después de posibles modificadores, una variable se declara definiendo su tipo seguido de su nombre. La declaración termina con un punto y coma.

Es posible declarar varias variables del mismo tipo en la misma línea, separando los nombres de las variables con una coma:

```
tipo variable1, variable2, ...;
```

Se muestran algunos ejemplos:

```
int edad;  
Date hoy, ayer;
```

b. La inicialización

La inicialización es el paso por el cual se suministra el primer valor a una variable.

La inicialización se puede llevar a cabo al mismo tiempo que la declaración:

```
[modificador] tipo nombreDeLaVariable = valor;
```

La inicialización se puede realizar después de la declaración de la variable:

```
[modificador] tipo nombreDeLaVariable;  
nombreDeLaVariable = valor;
```

El operador necesario para inicializar una variable, es el =.

valor, en el ejemplo, puede tomar diferentes formas:

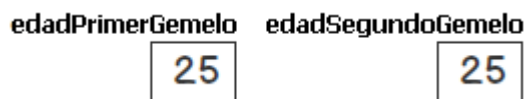
- Un valor literal:

```
edad = 41;
```

- Otra variable:

```
int edadPrimerGemelo = 25;  
int edadSegundoGemelo = primerGemelo;
```

La representación de este ejemplo en memoria es la siguiente:



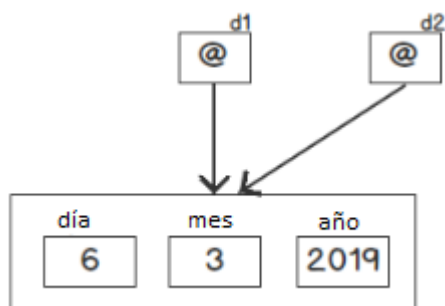
Como puede ver, cada variable gestiona su valor independientemente de la otra. Había una copia del valor de la variable `edadPrimerGemelo` en el espacio de memoria asignado a la variable `edadSegundoGemelo`. Si se cambia la edad de uno de los gemelos, el valor no se transfiere al otro gemelo. Este es el comportamiento que se puede observar para todos los tipos de valor.

No es lo mismo para los tipos por referencia. Considere el siguiente ejemplo:

```
Fecha d1 = new Date(2019,3,6);  
Fecha d2 = d1;
```

Este ejemplo no es real en Java, donde una fecha se crea de una manera más compleja, pero esto no tiene impacto en la demostración actual.

La representación de este caso en la memoria es la siguiente:



Como puede ver, el valor de la variable `d2` se corresponde con la misma dirección que la variable `d1`. Por lo tanto, el comportamiento es el mismo que para las variables de tipo por valor: había una copia de la variable `d1` (en este caso, una dirección) en el espacio de memoria asignado a la variable `d2`. Las consecuencias son, sin embargo, diferentes. De

hecho, las variables `d1` y `d2` están estrechamente relacionadas porque comparten las mismas variables de `dia`, `mes` y `año`.

- Una llamada de función que devuelve un valor:

```
int edad = calcularEdad(fechaDeNacimiento);
```

c. La asignación

La asignación es el paso por el cual cambiamos el valor de una variable. Por lo tanto, la inicialización se puede considerar como una asignación particular.

La asignación de una variable respeta la siguiente declaración:

```
nombreDeLaVariable = valor;
```

Para más detalles, consulte la sección La inicialización.

Ahora es necesario conocer los tipos disponibles.

6. Los tipos por valor

a. Los tipos numéricos enteros

Hay cuatro tipos diferentes para el manejo de números. Cada uno de estos tipos utiliza un espacio de almacenamiento diferente, para manejar valores más grandes o más pequeños. La siguiente tabla presenta estos cuatro tipos, con los valores mínimos y máximos aceptados y el espacio de almacenamiento utilizado.

Tipos enteros firmados			
Tipo	Valor mínimo	Valor máximo	Espacio de almacenamiento
byte	-128	127	8 bits
short	-32768	32767	16 bits

int	-2147483648	2147483647	32 bits
long	-9223372036854775808	9223372036854775807	64 bits

Cuando elija un tipo para sus variables enteras, deberá tener en cuenta los valores mínimos y máximos que piensa almacenar en ella con el fin de optimizar la memoria de la que hacen uso. De hecho, es inútil utilizar un tipo `long` para una variable cuyo valor no superará 50: en este caso, basta con un tipo `byte`. El ahorro de memoria parece insignificante para una variable única, pero se vuelve notable si se utilizan, por ejemplo, tablas de gran dimensión.

Tome el ejemplo de la clase `GestionValores`, que tiene como objetivo agregar valores numéricos completos. Para satisfacer esta necesidad, es posible declarar las dos siguientes variables locales, de tipo `int` y `long`, en el método `main`:

```
1  public class GestionValores
2  {
3      public static void main(String[] args)
4      {
5          int valor;
6          long sumaValor;
7      }
8  }
```

La variable `valor` aceptará sucesivamente valores entre -2147483648 y 2147483647. A continuación, este valor se añadirá a la variable `sumaValor`. Si el análisis funcional indica que la suma puede sobrepasar el límite de un número entero de tipo `int`, es conveniente declarar la variable `sumaValor` con el tipo `long`. Si no es el caso, el tipo `int` puede ser suficiente.

Todos los tipos enteros son firmados, es decir, que permiten generar valores positivos y negativos. No obstante, es posible trabajar con valores enteros no firmados utilizando las clases `Integer` y `Long`. Esto permite extender el valor positivo máximo admisible hasta 4294967296 en un tipo `int` y hasta 18446744073709551616 para un tipo `long`.

b. Los tipos numéricos decimales

Tipos numéricos con signo			
Tipo	Valor mínimopositivo	Valor máximopositivo	Espacio de almacenamiento
float	1.4E-45	3.4028235E38	32 bytes
double	4.9E-324	1.7976931348623157E308	64 bytes

Todos los tipos decimales están firmados y por lo tanto pueden contener valores positivos o negativos.

Las variables de tipo `float` y `double` ocupan el mismo espacio de almacenamiento que las variables de tipo `int` y `long`. Sin embargo, pueden almacenar valores de un rango mucho mayor. Esto es posible sacrificando la precisión. De hecho, estos tipos decimales son solo aproximaciones. El tipo `float` registra un máximo de ocho dígitos significativos, y el tipo `double`, 17 dígitos significativos.

c. El uso de valores literales

Es muy común inicializar variables de tipo numérico con un valor literal:

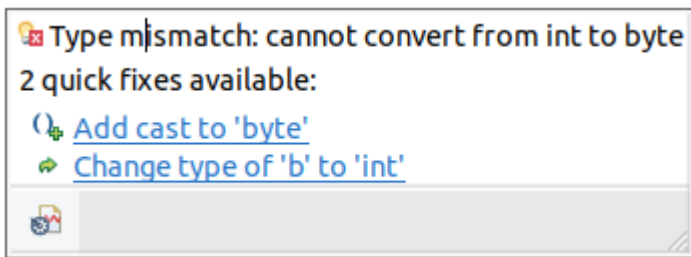
```
int poblacionEspania = 47680000;
double porcentajeMujeres= 52.879;
```

El compilador (`javac`) interpreta los valores numéricos enteros como valores de tipo `int` y los valores numéricos decimales como valores de tipo `double`. En consecuencia, las dos declaraciones anteriores compilan.

En las siguientes declaraciones, se produce un error de compilación:

```
byte b = 153;
```

El compilador indica el siguiente error:



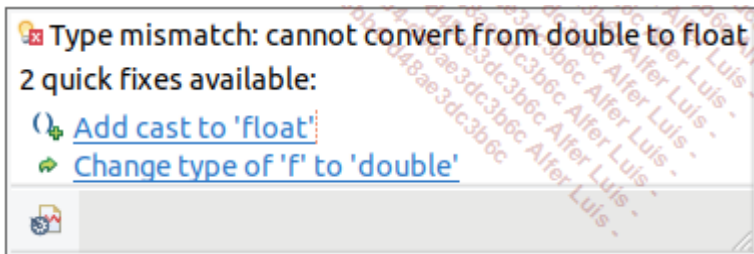
El valor 153 está fuera del rango de los valores aceptables por una variable de tipo `byte`. El compilador indica que no puede convertir este valor de tipo `int` a tipo `byte`. Si el valor establecido está dentro del rango de tipos, el problema desaparece.

El mismo comportamiento se observa para el tipo `short`, cuando intentamos inicializar la variable con un valor fuera del rango aceptable para este tipo.

El mismo error ocurre cuando el desarrollador intenta inicializar una variable de tipo `float` con un valor decimal literal (considerado como un tipo `double`). La diferencia es que el compilador arrojará un error, incluso si el valor está en el rango aceptable del tipo `float`, como en el siguiente ejemplo:

```
float f = 10.1;
```

El compilador informa del siguiente error:



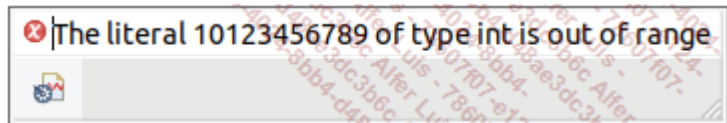
Para corregir este problema, debe decirle al compilador que interprete el valor literal como un valor de tipo `float`. Para realizar esta operación, simplemente agregue el número con la letra `F` (para `float`) en mayúscula o minúscula:

```
float f = 10.1F;
```

El segundo caso en el que es posible encontrar un error se produce cuando el desarrollador desea inicializar una variable de tipo `long` con un valor numérico que excede el rango de valores aceptables para una variable de tipo `int`.

```
long l = 10123456789;
```

Recuerde que el compilador interpreta los valores literales numéricos completos como valores de tipo `int`. El error encontrado con el ejemplo anterior es el siguiente:



Para corregir este problema, debe indicar al compilador que interprete el valor literal como un valor de tipo `long`. Para realizar esta operación, simplemente agregue el número con la letra `L` (para `long`) en mayúscula o minúscula:

```
long l = 10123456789L;
```

Los valores numéricos decimales se pueden expresar con notación decimal o notación científica.

```
float superficie=2356.8f;  
float superficie=2.3568e3f;
```

Puede insertar caracteres en valores numéricos literales para facilitar la lectura. Las siguientes dos sintaxis son equivalentes.

```
long cantidad=1234876567;  
long cantidad=1_234_876_567;
```

d. El tipo carácter

El tipo `char` se utiliza para almacenar un carácter único. Una variable de tipo `char` usa dos bytes para almacenar el código **Unicode** del carácter. En el juego de caracteres Unicode los primeros 128 caracteres son idénticos al juego de carácter **ASCII**; los caracteres siguientes, hasta 255, corresponden a los caracteres especiales del alfabeto latino (por ejemplo, los caracteres acentuados). El resto se utiliza para los símbolos o los caracteres de otros alfabetos.

```
char caracterC = 'c';
```

Los caracteres específicos o los que tienen un significado particular para el lenguaje Java se representan por una secuencia de escape. Se compone del carácter `\` seguido por otro carácter que indica el significado de la secuencia de escape. La tabla siguiente presenta la lista de secuencias de escape y sus significados.

Secuencia	Significado
<code>\t</code>	Tabulación
<code>\b</code>	Retroceso
<code>\n</code>	Salto de línea
<code>\r</code>	Retorno de carro
<code>\f</code>	Salto de página
<code>\'</code>	Comilla simple
<code>\"</code>	Comilla doble
<code>\\</code>	Barra invertida

Los caracteres Unicode no accesibles por teclado se representan también mediante una secuencia de escape compuesta por los caracteres `\u` seguidos por el valor hexadecimal del código Unicode del carácter. El símbolo del euro es, por ejemplo, la secuencia `\u20AC`.

```
char euro1 = '€';  
char euro2 = '\u20AC';
```

Para adquirir conocimientos sobre los caracteres Unicode, puede consultar el recurso Wikipedia que constituye un buen punto de inicio, en la siguiente dirección: https://es.qwe.wiki/wiki/List_of_Unicode_characters

Una cadena de caracteres está representada por la clase `String`. Esta clase maneja una

serie de caracteres. La noción de array y la clase `String` se abordan un poco más adelante en este capítulo, en la sección Los tipos por referencia.

e. El tipo booleano

El tipo `booleano` permite tener una variable que puede presentar dos estados: verdadero/falso, si/no, on/off.

La asignación se hace directamente con los valores `true` o `false`, como en el ejemplo siguiente:

```
boolean disponible,modificable;  
disponible=true;  
modificable=false;
```

Es imposible asignar otro valor a una variable de tipo `booleano`.

7. Los tipos por referencia

a. Introducción

Los arrays, cadenas y variables complejas son variables de tipos por referencia. Los arrays se utilizan para administrar conjuntos de variables (a menudo del mismo tipo). Las cadenas de caracteres se utilizan para administrar conjuntos de caracteres. Las variables complejas también le permiten administrar un conjunto de variables. A menudo, el tipo de variables complejas es una clase. Por ejemplo, una clase de `Persona` le permite caracterizar a una persona agrupando información diferente en tantas variables como sea necesario: nombre (de tipo `String`), apellidos (de tipo `String`) y edad (de tipo `int`).

El siguiente capítulo trata sobre la programación orientada a objetos, con la creación y manipulación de clases. Las siguientes secciones del capítulo actual tratan sobre los tipos por referencia inevitables: los arrays, las cadenas de caracteres y las fechas.

b. Los arrays

Los arrays permiten hacer referencia a un conjunto de variables del mismo tipo con el mismo nombre utilizando un índice para diferenciarlas. Un array puede tener una o varias

dimensiones. El primer elemento de un array siempre tiene como índice el cero. El número de celdas del array se especifica en el momento de su creación.

Por lo tanto, el índice más grande de un array es igual al número de celdas menos una. Después de su creación, las características de un array ya no se pueden modificar (número de celdas, tipo de los elementos almacenados en el array, etc.). La manipulación de un array se divide en tres etapas:

- Declaración de una variable que permite trabajar con el array.
- Creación del array (asignación de memoria).
- Almacenamiento y manipulación de los elementos del array.

Declaración del array

La declaración del array se lleva a cabo de forma similar a la de una variable clásica. Únicamente se deben añadir un par de corchetes `[ ]` después del tipo de datos o del nombre de la variable. Es preferible, para una mejor legibilidad del código, asociar los caracteres `[` y `]` (corchetes) al tipo de datos. La línea siguiente declara una variable de tipo array de enteros.

```
int[] cifraNegocio;
```

Creación del array

Después de la declaración de la variable, hay que crear el array obteniendo memoria para almacenar esos elementos. En este momento, indicamos el tamaño del array. Dado que los arrays se asemejan a objetos, se utilizará el operador `new` para crear una instancia del array. El valor proporcionado por el operador `new` se almacena en la variable declarada previamente.

```
cifraNegocio=new int[12];
```

Esta declaración creará un array con doce celdas numeradas de 0 a 11. El tamaño del array es definitivo; por lo tanto, no es posible ampliar ni reducir un array ya creado.

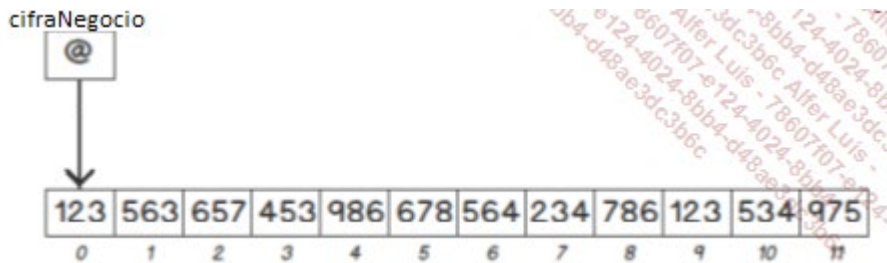
Existe una alternativa a la creación de arrays. Permite la creación e inicialización de su contenido. La sintaxis es la siguiente:

```
int[]
```

```
cifraNegocio={1234,563,657,453,986,678,564,234,786,123,534,975};
```

En este caso, no hace falta precisar un tamaño para el array. El dimensionamiento se hará automáticamente según el número de valores declarados entre las llaves.

A continuación se muestra la representación en memoria que podemos hacer de este array:



La variable `cifraDeNegocio` contiene la dirección en memoria del array. El array está formado por 12 celdas contiguas numeradas de 0 a 11.

Utilización del array

Los elementos de los arrays son accesibles de la misma manera que una variable clásica. Es necesario añadir el índice del elemento que se quiere manipular. El ejemplo siguiente muestra la lectura de una celda del array:

```
int cifraNegocioMesEnero= cifraNegocio [0];
```

El siguiente ejemplo muestra la modificación del array.

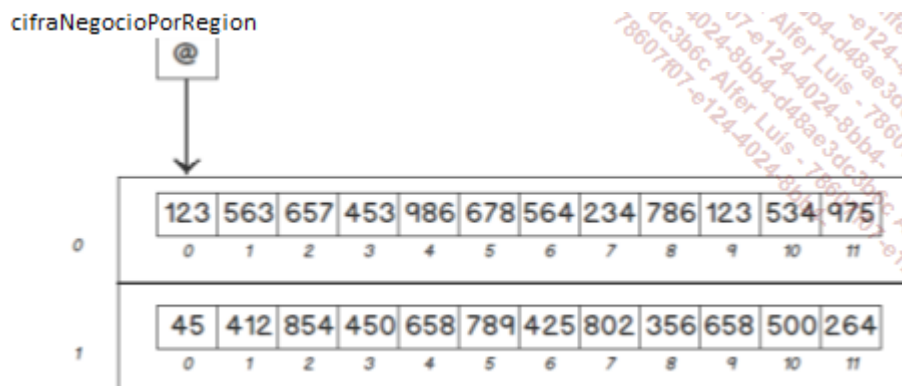
```
CifraNegocio [0]=354;
```

Hay que tener cuidado al manipular un array y no intentar acceder a una celda que no exista, ya que se corre el riesgo de obtener una excepción del tipo `ArrayIndexOutOfBoundsException`.

Arrays de varias dimensiones

Los arrays multidimensionales se usan ampliamente para representar tablas de doble entrada (filas y columnas). Sin embargo, permiten representar estructuras más complejas en tres dimensiones o incluso más. Para representar la lógica de una tabla, es posible utilizar un array tridimensional que se refiera a hojas, filas y columnas.

Los arrays multidimensionales son, de hecho, arrays que contienen otros arrays. La siguiente ilustración muestra un array bidimensional:



Este array representa la cifra de negocio por región y por mes. La primera dimensión contiene dos cuadros para las dos regiones donde opera la empresa. La segunda dimensión contiene doce cajas para la cifra mensual de negocio, producida por la empresa según la región.

Vamos a considerar que Andalucía se corresponde con la primera casilla de la primera dimensión y que Asturias se corresponde con la segunda casilla de la primera dimensión.

Para conocer la cifra de negocio realizada en Andalucía en febrero, la instrucción es la siguiente:

```
int CNAndaluciaFebrero = cifraDeNegocioPorRegion[0][1];
```

[0] representa la primera celda de la primera dimensión (Andalucía).

[1] representa la segunda celda de la segunda dimensión (Febrero).

Para modificar la cifra de negocio realizada en el Asturias en junio, la instrucción es la siguiente:

```
cifraDeNegocioPorRegion[1][5]=1238;
```

[1] representa la segunda celda de la primera dimensión (Asturias).

[5] representa la sexta celda de la segunda dimensión (Junio).

La sintaxis de declaración es similar a la de un array, excepto que se deben especificar tantos pares de corchetes como dimensiones tenga.

```
int[][] matriz;
```

Su creación varía un poco. Se debe mencionar el número de cajas en la primera dimensión. El número de cajas en las otras dimensiones se puede definir más adelante si:

- no sabemos en el momento de la creación el tamaño necesario;
- el tamaño del array en las otras dimensiones varía.

En la mayoría de los casos, el tamaño de las diferentes dimensiones se conoce de antemano y es homogéneo en cada dimensión. La creación del array se realiza entonces así:

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

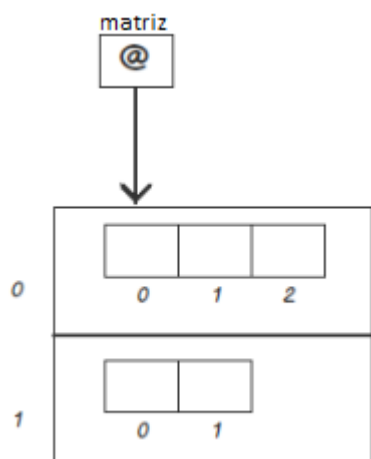
Cuando no se cumplen estas condiciones, es posible diferir la creación de las otras dimensiones. El primer paso es crear la primera dimensión de esta manera:

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

Luego, para cada casilla de la primera dimensión, es posible crear un array correspondiente a la segunda dimensión:

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

La representación siguiente ilustra esta matriz:



La sintaxis que permite inicializar una matriz en el momento de su declaración es un poco más compleja.


```
int[][] matriz={{11,12,13},{21,22,23}};
```

El uso anidado de las llaves permite definir diferentes dimensiones. Los valores se ubican en las celdas de la última dimensión. El ejemplo anterior permite inicializar una matriz con dos dimensiones, con dos líneas y tres columnas.

Manipulaciones habituales con arrays

Cuando se trabaja con arrays, hay ciertas operaciones que se realizan regularmente. Esta sección describe las más corrientes. La mayoría de ellas están disponibles gracias a la clase `java.util.Arrays`, que proporciona numerosos métodos `static` de manipulación de arrays.

Obtener el tamaño de un array: basta con utilizar la propiedad `length` del array para conocer el número de elementos que puede contener. En el caso de un array multidimensional, hay que recordar que se trata de hecho de arrays de arrays. La propiedad `length` indica, por lo tanto, el número de elementos de la primera dimensión.

Para obtener la misma información de las demás dimensiones, hay que utilizar la propiedad `length` de cada celda del array de nivel inferior.

```
int[][] matriz=new int[8][3];
System.out.println("el array contiene " + matriz.length +
    " celdas de " + matriz[0].length +
    " celdas");
```

Mostrar un array: el método `toString` le permite obtener una nueva presentación en forma de una cadena de caracteres del array que se pasa como argumento. Este método devuelve una representación textual del array. El retorno de este método se pasa como un argumento de la función `System.out.println` para tener una pantalla en la consola.

```
System.out.println(Arrays.toString(cifraDeNegocio));
```

La instrucción anterior muestra el resultado siguiente:

```
[123, 234, 453, 534, 563, 564, 657, 678, 786, 975, 986, 1234]
```

El método `deepToString` realiza la misma operación para un array multidimensional.

```
int[][] matriz={{11,12,13},{21,22,23},{31,32,33}};  
System.out.println(Arrays.deepToString(matriz));
```

La instrucción anterior muestra el resultado siguiente:

```
[[11, 12, 13], [21, 22, 23], [31, 32, 33]]
```

Ordenar un array: el método `sort` garantiza la clasificación del array que recibe como argumento. La ordenación se realiza en orden alfabético para los arrays de cadenas de caracteres, y en orden ascendente para los arrays de valores numéricos.

```
Arrays.sort(cifraDeNegocio);  
System.out.println(Arrays.toString(cifraDeNegocio));
```

La instrucción anterior muestra el resultado siguiente:

```
[123,123,234,453,534,563,564,657,678,786,975,986]
```

El método `parallelSort` también ordena el array, pero usando un algoritmo que explota las capacidades de una máquina multiprocesador.

Buscar un elemento en un array: el método `binarySearch` permite efectuar una búsqueda en un array. Recibe como parámetros el array en el cual se hace la búsqueda y el elemento buscado. El valor devuelto corresponde al índice donde se encontró el elemento o un valor negativo si el elemento no ha sido encontrado. Para que este método realice la búsqueda de forma óptima, se debe ordenar el array previamente.

```
Arrays.sort(cifraNegocio);  
System.out.println(Arrays.binarySearch(cifraNegocio, 123));
```

Copiar un array: hay disponibles dos métodos para la copia de arrays.

El método `copyOf` copia un array entero recibido como argumento, con la posibilidad de modificar su tamaño con un segundo argumento.

```
int[] copiaCifraNegocio;
```

```
copiaCifraNegocio=Arrays.copyOf(cifraNegocio, 24);  
System.out.println(Arrays.toString(copiaCifraNegocio));
```

La instrucción anterior muestra el resultado siguiente:

```
[123, 563, 657, 453, 986, 678, 564, 234, 786, 123, 534, 975, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

El array se copió completamente en un array de 24 celdas. Las nuevas celdas se encuentran al final del array y se inicializan con el valor predeterminado de un entero (0).

El método `copyOfRange` realiza la copia de una parte de un array recibido como argumento. El índice de la celda a partir de la que empieza la copia y el número de celdas implicadas forman los otros dos argumentos de esta función.

```
int[] primerTrimestre;  
primerTrimestre=Arrays.copyOfRange(cifraNegocio, 0, 3);  
System.out.println(Arrays.toString(primerTrimestre));
```

La instrucción anterior muestra el resultado siguiente:

```
[123, 563, 657]
```

La copia del array comienza en el índice 0 y afecta a 3 celdas. El tamaño del nuevo array se corresponde exactamente con el tamaño necesario para copiar las celdas seleccionadas por la instrucción.

Rellenar un array: el método `fill` se utiliza para rellenar todas las celdas de un array con el mismo valor. El ejemplo siguiente permite rellenar el array con el valor 2.

```
int[] array = new int[5];  
Arrays.fill(array,2);  
System.out.println(Arrays.toString(array));
```

La instrucción anterior muestra el resultado siguiente:

```
[2,2,2,2,2]
```

c. Las cadenas de caracteres

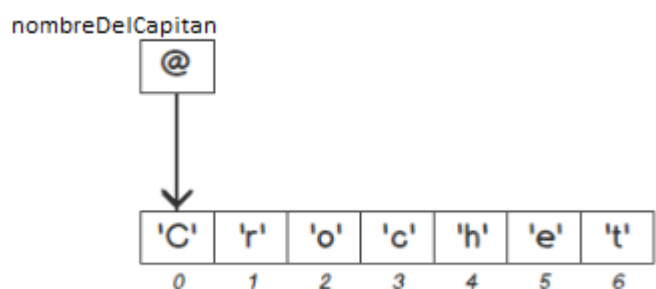
Presentación

Para poder almacenar cadenas de caracteres, debe usar el tipo `String`, que representa una secuencia de cero a n caracteres. Este tipo no es un tipo básico, sino una clase. Sin embargo, para facilitar su uso, se puede utilizar como un tipo básico del lenguaje. Las cadenas de caracteres son invariables porque al asignar un valor a una variable de tipo cadena el espacio se reserva en la memoria para almacenar la cadena.

Si esta variable recibe posteriormente un nuevo valor, se le asigna una nueva ubicación en la memoria. Afortunadamente, este mecanismo es transparente para nosotros y la variable siempre se referirá automáticamente al valor asignado. Con este mecanismo, las cadenas de caracteres pueden tener un tamaño variable. El espacio ocupado en la memoria se ajusta automáticamente de acuerdo con la longitud de la cadena de caracteres. Para asignar una cadena de caracteres a una variable, debe ingresar el contenido de la cadena entre comillas, como en el siguiente ejemplo.

```
String nombreDelCapitan = "Crochet";
```

La representación en memoria de esta variable se puede materializar también como:



Muchos métodos de la clase `String` permiten la manipulación de cadenas de caracteres y se detallan en el resto de este capítulo.

Crear una cadena de caracteres

El método más sencillo para crear una cadena de caracteres consiste en considerar el tipo `String` como un tipo primitivo del lenguaje, y no como un objeto. En este caso, la asignación de un valor a la variable va a provocar la creación de una instancia de la clase `String`. También es posible crear una cadena de caracteres como un objeto utilizando el operador `new` y alguno de los constructores disponibles en la clase `String`.

El ejemplo de código siguiente presenta las dos soluciones.

```
String cadena1="eni";  
String cadena2=new String("eni");
```

Después de su creación, no se puede modificar una cadena de caracteres. La asignación de otro valor a la variable provoca la creación de una nueva instancia de la clase `String`. La clase `String` contiene numerosos métodos que permiten modificar cadenas de caracteres. Al utilizarla, tenemos la sensación de que la función modifica el contenido de la cadena inicial, pero, en realidad, la función devuelve una nueva instancia que contiene el resultado.

Asignar un valor a una cadena

Para asignar un valor a una cadena, es necesario especificarla entre comillas. Se plantea un problema si queremos que el carácter " forme parte de la cadena. Para que no se interprete como carácter de principio o de fin de cadena, hay que protegerlo por una secuencia de escape (usando el carácter `\`), como en el ejemplo siguiente:

```
String cadena;  
cadena=" Dijo: \"¡Basta ya!\";  
System.out.println(cadena);
```

El resultado que se visualiza es el siguiente:

```
Dijo: "¡Basta ya!"
```

Para los ejemplos siguientes, vamos a trabajar con dos cadenas:

```
String cadena1 = "el invierno será lluvioso";  
String cadena2 = "el invierno será frío";
```

Extraer un carácter particular

Para obtener el carácter situado en una posición determinada de una cadena de caracteres, se debe utilizar la función `charAt` proporcionando como argumento el índice del carácter que deseamos obtener. El primer carácter tiene el índice cero, como para un array. Esta

función devuelve un carácter (`char`).

```
System.out.println("el cuarto carácter de la cadena1 es " +  
cadena1.charAt(2));
```

La ejecución de esta instrucción da el siguiente resultado:

```
El cuatro carácter de la cadena 1 es i
```

Obtener la longitud de una cadena

Para determinar la longitud de una cadena, se dispone de la función `length` de la clase `String`.

```
System.out.println("la cadena1 contiene " + cadena1.length() + "  
caracteres");
```

La ejecución de esta instrucción da el siguiente resultado:

```
La cadena 1 contiene 25 caracteres.
```

Descomponer una cadena

La función `substring` de la clase `String` devuelve una sección de una cadena de caracteres, en función de las posiciones inicial y final que se le proporcionan como parámetros. La cadena obtenida empieza por el carácter ubicado en la posición inicial (principio) y termina en el carácter que precede a la posición final.

```
System.out.println("un trozo de la cadena1: " +  
cadena1.substring(3,11));
```

La ejecución de esta instrucción da el siguiente resultado:

```
un trozo de la cadena1: invierno
```

Probar la igualdad de cadenas

Cuando hacemos una comparación entre dos cadenas, tendemos a utilizar el doble igual

(==), como para los tipos por valor (ver la sección Los operadores, un poco más adelante en este capítulo). Este operador funciona correctamente con los tipos primitivos, pero no podemos olvidar que las cadenas de caracteres son de tipo objeto. El operador == verifica la igualdad de las variables (los valores para los tipos por valor y la dirección para los tipos por referencia).

Hay que utilizar los métodos de la clase `String` para efectuar comparaciones de cadenas de caracteres. El método `equals` compara la cadena con la que se pasa como parámetro. Devuelve un booleano igual a `true` si las dos cadenas son idénticas (por valor) y por supuesto un booleano igual a `false` en caso contrario. Esta función distingue entre minúsculas y mayúsculas durante la comparación. La función `equalsIgnoreCase` realiza la misma acción, pero sin tener en cuenta esta distinción.

```
if (cadena1.equals(cadena2))
{
    System.out.println("las dos cadenas son idénticas");
}
else
{
    System.out.println("las dos cadenas son diferentes");
}
```

☐ No nos dejemos engañar por las apariencias. En ciertos casos, el operador == es capaz de realizar una comparación correcta de cadenas de caracteres. El siguiente código funciona correctamente y devuelve el resultado esperado, considerando que ambas cadenas son idénticas.

```
String s1="pepe";
String s2="pepe";
if (s1==s2)
{
    System.out.println("cadenas idénticas");
}
else
{

```

```
        System.out.println("cadenas diferentes");  
    }
```

De hecho, para ahorrar espacio en memoria, Java utiliza en este caso una única instancia de la clase `String` para las variables `s1` y `s2`, pues el contenido de ambas cadenas es idéntico.

Ambas variables `s1` y `s2` hacen referencia a la misma zona de memoria y el operador `==` confirma su igualdad (las direcciones).

Si, por el contrario, utilizamos el siguiente código, que exige explícitamente la creación de una instancia de la clase `String` para cada una de las variables `s1` y `s2`, el operador `==` no es capaz de confirmar la igualdad de las cadenas.

```
String s1=new String("pepe");  
String s2=new String("pepe");  
if(s1==s2)  
{  
    System.out.println("cadenas idénticas");  
}  
else  
{  
    System.out.println("cadenas diferentes");  
}
```

Comparar dos cadenas

Para realizar una comparación, puede utilizar el método `compareTo` de la clase `String` o el método `compareToIgnoreCase` si desea tener en cuenta la diferencia entre mayúsculas y minúsculas. Estos dos métodos esperan como parámetros la cadena que se debe comparar, con la instancia actual en la que se llama al método. Se devuelve el resultado de la comparación bajo la forma de un entero inferior a cero si la cadena es inferior a la recibida como parámetro; igual a cero si las dos cadenas son idénticas, y superior a cero si la cadena es superior a la recibida como parámetro.

```
if (cadena1.compareTo(cadena2)>0)  
{  
    System.out.println("cadena1 es superior a cadena2");  
}
```



```

}
else
if (cadena1.compareTo(cadena2)<0)
{
    System.out.println("cadena1 es inferior a cadena2");
}
else
{
    System.out.println("las dos cadenas son idénticas");
}

```

Las funciones `startsWith` y `endsWith` permiten comprobar si la cadena empieza con la cadena recibida como parámetro o si la cadena termina con la cadena recibida como parámetro. Se puede utilizar la función `endsWith`, por ejemplo, para comprobar la extensión de un nombre de archivo.

```

String nombre="Código.java";

if (nombre.endsWith(".java"))
{
    System.out.println("es un archivo fuente java");
}

```

Suprimir los espacios

La función `trim` permite suprimir los espacios en blanco situados delante del primer carácter significativo y después del último carácter significativo de una cadena.

```

String eni="      eni      ";
System.out.println("longitud de la cadena: " +
                    eni.length());
System.out.println("longitud de la cadena sin espacios: " +
                    eni.trim().length());

```

Cambiar las letras a mayúsculas o minúsculas

Todo a mayúsculas:

```
System.out.println(cadena1.toUpperCase());
```

Todo a minúsculas:

```
System.out.println(cadena1.toLowerCase());
```

Buscar en una cadena

El método `indexOf` de la clase `String` permite buscar una cadena en el interior de otra. El parámetro corresponde a la cadena buscada. La función devuelve un entero que indica la posición donde se encuentra la cadena o `-1` si no se encuentra. Por defecto, la búsqueda empieza al principio de la cadena, salvo si utilizamos otra versión de la función `indexOf` que recibe dos parámetros: el primer parámetro es la cadena buscada, y el segundo, la posición inicial de la búsqueda.

```
String búsqueda;
int posición;
búsqueda = "e";
posición = cadena1.indexOf(búsqueda);
while (posición > 0)
{
    System.out.println("cadena encontrada en la posición " +
                        posición);
    posición = cadena1.indexOf(búsqueda, posición+1);
}
System.out.println("fin de la búsqueda");
```

Obtenemos la visualización siguiente:

```
cadena encontrada en la posición 0
cadena encontrada en la posición 7
cadena encontrada en la posición 13
fin de la búsqueda
```

Sustituir una parte de una cadena

A veces puede ser preferible buscar la presencia de una cadena en el interior de otra, como en el ejemplo anterior, pero también sustituir las porciones de cadenas encontradas. La

función `replace` permite especificar una cadena de sustitución para la cadena buscada. Recibe dos parámetros:

- la cadena buscada,
- la cadena sustituta.

```
String cadena3;  
cadena3= cadena1.replace("invierno", "verano");  
System.out.println(cadena3);
```

Obtenemos lo siguiente:

```
el verano será lluvioso
```

Formatear una cadena

El método `format` de la clase `String` permite evitar largas y complicadas operaciones de conversión y de concatenación. Es un método de clase y no un método de instancia. El primer parámetro que recibe esta función es una cadena de caracteres que indica en qué forma se desea obtener el resultado. Esta cadena contiene uno o varios motivos de formateo representados por el carácter `%` seguido de un carácter específico que indica en qué forma debe presentarse la información. Debe recibir, a continuación, tantos parámetros como motivos de formateo. La cadena devuelta se construye remplazando cada uno de los motivos de formateo por el valor del parámetro correspondiente, y el remplazo se realiza en función del orden de aparición de los motivos. La siguiente tabla presenta los principales motivos de formateo disponibles.

Motivo	Descripción
%b	Insertar un booleano
%s	Insertar una cadena de caracteres
%d	Insertar un número entero
%o	Insertar un número entero en representación octal

%x	Insertar un número entero en representación hexadecimal
%f	Insertar un número decimal
%e	Insertar un número decimal representado en formato científico
%n	Insertar un salto de línea

Aquí se muestra un ejemplo de código usando esta función:

```
boolean b=true;
String s="cadena";
int i=56;
double d=19.6;
System.out.println(String.format("booleano: %b %n" +
    "cadena de caracteres: %s %n" +
    "entero: %d %n" +
    "entero en hexadecimal: %x %n" +
    "entero en octal: %o %n" +
    "decimal: %f %n" +
    "redondeo a primer decimal: %.1f %n" +
    "decimal en formato científico: %e%n",
    b,s,i,i,i,d,d));
```

Observe la secuencia `% . 1 f` para la visualización decimal. Esta instrucción permite indicar el número de cifras significativas deseadas después de la coma.

La ejecución de este código da el siguiente resultado:

```
booleano: true
cadena de caracteres: cadena
entero: 56
entero en hexadecimal: 38
entero en octal: 70
decimal: 5,500000
redondeo a primer decimal: 5,5
decimal en formato científico: 5.500000e+00
```

d. Las fecha y las horas

La gestión de las fecha y hora ha sido, durante mucho tiempo, la bestia negra de los desarrolladores en Java. La clase `java.util.Date` era la referencia para almacenar variables de este tipo. Las clases `Calendar` y `GregorianCalendar` estaban disponibles para responder a las problemáticas de manipulación de fecha y hora. La clase `SimpleDateFormat` permitía la transición entre una variable de tipo `Date` y su representación textual. Se han previsto muchas funcionalidades, aunque su uso pone de relieve, en ocasiones, verdaderos rompecabezas. Es cierto que se trata de un problema complejo. Trabajar en base 60 para los segundos y los minutos y en base 24 para las horas no es sencillo. Aunque el colmo aparece en la gestión de los meses, que no tienen todos el mismo número de días, o incluso peor, dado que algunos meses tienen un número de días diferente en función de los años. Los ordenadores utilizan una técnica diferente al no trabajar directamente con fecha y hora, sino con el número de segundos o milisegundos transcurridos tras una fecha de referencia (generalmente el 1 de enero 1970 a las 0 horas). Esta forma de representación no resulta muy práctica desde un punto de vista humano.

El valor `1552345200000` no nos evoca ninguna fecha; decir, sin embargo, `25/03/2019` es mucho más indicativo. Este es el motivo de que existan numerosas funciones que permiten pasar de un formato a otro.

En la versión 8 de Java, la gestión de las fechas y las horas se ha repensado por completo. Numerosas clases especializadas han hecho su aparición. A continuación se muestra un resumen de estas clases:

Clase	Descripción
<code>LocalDate</code>	Representa una fecha (día, mes, año) sin hora.
<code>LocalDateTime</code>	Representa una fecha y una hora sin tener en cuenta el huso horario.
<code>LocalTime</code>	Representa una hora sin tener en cuenta el huso horario.
<code>OffsetDateTime</code>	Representa una fecha y una hora en formato UTC.

OffsetTime	Representa una hora en formato UTC.
ZonedDateTime	Representa una fecha y una hora con el huso horario correspondiente.
Duration	Representa una duración expresada en horas, minutos y segundos.
Period	Representa una duración expresada en días, meses y años.
MonthDay	Representa un día y un mes, sin año.
YearMonth	Representa un mes y un año, sin día.

Todas estas clases proporcionan una serie de métodos que permiten manipular sus elementos. Estos métodos respetan una convención en su nomenclatura que facilita su uso.

- `of`: devuelve una instancia de la clase inicializada con los distintos valores que se pasan como parámetro.

```
LocalDate navidad;
navidad=LocalDate.of(2019, 12,25);
```

- `from`: permite una conversión entre estos distintos tipos. En caso de conversión hacia un tipo menos complejo, se pierde información.

```
LocalDateTime ahora;
ahora=LocalDateTime.now();
// transformación en LocalDate
// con pérdida de la hora
LocalDate hoy;
hoy=LocalDate.from(ahora);
```

- `parse`: transforma la cadena de caracteres que se pasa como parámetro al tipo correspondiente.

```
LocalTime reloj;  
reloj=LocalTime.parse("22:45:03");
```

- `withXxx`: devuelve una nueva instancia modificando el componente indicado por `Xxx` por el valor que se pasa como parámetro.

```
LocalTime reloj;  
reloj=LocalTime.parse("22:45:03");  
LocalTime nuevaHora;  
nuevaHora=reloj.withHour(9);
```

- `plusXxx` y `minusXxx`: devuelve una nueva instancia de la clase tras agregar o sustraer el número de unidades indicado por el parámetro. `Xxx` indica qué se agrega o se sustrae.

```
LocalDate pascua;  
pascua=LocalDate.of(2014,4,22);  
LocalDate ascension;  
ascension=pascua.plusDays(39);
```

- `atXxx`: combina el objeto recibido como parámetro con el objeto en curso y devuelve el resultado de dicha asociación. Es posible, por ejemplo, combinar un objeto `LocalDate` y un objeto `LocalTime` para obtener un objeto `LocalDateTime`.

```
LocalDate diaPartido;  
diaPartido=LocalDate.of(2019,7,13);
```

```
LocalTime horaPartido;  
horaPartido=LocalTime.of(21,00);
```

```
LocalDateTime inicio;  
inicio=diaPartido.atTime(horaPartido);
```

Formatear una fecha para una visualización adecuada requiere el uso de la clase `DateTimeFormatter`. Es posible construir una instancia de esta clase con el método

de clase de `ofPattern`. Este método espera como argumento una cadena de caracteres que define el formato de visualización. Para visualizar en formato DD/MM/AAAA, el patrón correspondiente es `dd/MM/aaaa`. El formateo es posible usando el método `format`:

```
LocalDate hoy = LocalDate.now();
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy");
System.out.println(dtf.format(hoy));
```

El pequeño ejemplo de código siguiente muestra algunas operaciones sobre las fechas teniendo en cuenta que los días festivos son los sábados y domingos por cada año entre 2019 y 2030.

```
MonthDay[] fiestas;
fiestas=new MonthDay[8];
fiestas[0]=MonthDay.of(1,1);
fiestas[1]=MonthDay.of(5,1);
fiestas[2]=MonthDay.of(8,15);
fiestas[3]=MonthDay.of(10,12);
fiestas[4]=MonthDay.of(11,1);
fiestas[5]=MonthDay.of(12,6);
fiestas[6]=MonthDay.of(12,8);
fiestas[7]=MonthDay.of(12,25);

int numDias;
int año;
LocalDate diaTest;
for (año=2019;año<2030;año++)
{
    numDias=0;
    for(MonthDay test:fiestas)
    {
        diaTest=test.atYear(año);
        if (diaTest.getDayOfWeek()==DayOfWeek.SATURDAY
            ||diaTest.getDayOfWeek()==DayOfWeek.SUNDAY)
        {
            numDias++;
        }
    }
}
```



```

    }
    System.out.println("En " + año + " hay " + numDias
        + " dia(s) festivo(s) sábado o domingo");
}

```

8. Los valores por defecto

No siempre es obligatoria la inicialización de las variables. Por ejemplo, es el caso de las variables de clase que se inicializan con los siguientes valores predeterminados.

Tipo	Valor por defecto
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	\u0000
boolean	false
Los tipos por referencia	null

Sin embargo, las variables locales deben inicializarse antes de ser utilizadas. El compilador también realiza una comprobación cuando encuentra el uso de una variable local y genera un error si la variable no se ha inicializado.

Como recordatorio, la siguiente estructura muestra la localización de las variables :

```

1  public class NombreDeLaClase
2  {
3      //VARIABLES DE INSTANCIA
4      //VARIABLES DE CLASE
5      ... unMetodo(ARGUMENTOS)
5      {
7          //VARIABLES LOCALES
8      }
9  }

```

9. El ámbito de las variables

El ámbito de una variable es la porción de código desde la que podemos manipular esta variable. Depende de dónde se encuentra la declaración.

Esta declaración puede hacerse en:

- el bloque de código de una clase,
- el bloque de código de una función,
- un bloque de código dentro de una función.

Una variable solo se puede usar en el bloque donde se declara y en los sub-bloques. Si el mismo bloque de código se ejecuta varias veces durante la ejecución de la función, en el caso de un ciclo `while` por ejemplo, la variable se creará cada vez que pase por el ciclo. No puede haber dos variables con el mismo nombre con el mismo ámbito.

Sin embargo, tiene la opción de declarar una variable interna a una función o un argumento de una función con el mismo nombre que una variable declarada a nivel de clase. Se implementa en ese caso el principio de localidad. Por lo tanto, la variable local tiene prioridad sobre la variable declarada a nivel de clase. Para acceder a esta variable, es preciso que vaya precedida por la palabra clave `this` (en el caso de una variable de instancia) o por el nombre de la clase (en el caso de una variable de clase).

```

1  public class NombreDeLaClase
2  {
3      int miVariableDeInstancia;

```

```
4      static int miVariableDeClase;
5      ... unMetodo(ARGUMENTOS)
6      {
7          int miVariableLocal;
8          if(expresión booleana)
9          {
10             int miVariableDeBloque;
11          }
12      }
13 }
```

Las variables `miVariableDeInstancia` y `miVariableDeClase` son accesibles en toda la clase (todos los métodos y todos los bloques dentro de métodos). Se puede acceder a la variable `miVariableLocal` desde su declaración hasta el final del método en el que se declara (en el bloque en el que se declara y en todos los bloques dentro del método). Se puede acceder a la variable `miVariableDeBloque` desde su declaración hasta el final del bloque en el que se declara (también es accesible en los subbloques de este bloque).

10. El ciclo de vida de las variables

La vida útil de una variable le permite especificar durante cuánto tiempo estará disponible el contenido de una variable, durante la ejecución de la aplicación.

Para una variable declarada en un método, el tiempo de vida se corresponde con la duración de la ejecución de este. Tan pronto como finaliza la ejecución del método, la variable se elimina de la memoria. Se vuelve a crear durante la próxima llamada al método.

Se puede usar una variable de instancia, siempre que la instancia esté disponible.

Se puede acceder a una variable de clase (declarada con la palabra clave `static`) durante todo el tiempo de funcionamiento de la aplicación.

11. La conversión de tipos

a. Presentación

La conversión de tipos es un mecanismo esencial en un programa informático. Entre la introducción por teclado de un usuario, que es una cadena de caracteres, y el uso de esta entrada, a menudo es necesario realizar una conversión.

Es necesario diferenciar entre la conversión de los tipos de valores, que también cambia el valor, y la conversión de los tipos por referencia, que solo cambia el tipo de la variable. Este punto se abordará en el próximo capítulo sobre programación orientada a objetos.

El resto de este capítulo trata sobre la conversión de tipos primitivos y cadenas de caracteres que, como recordatorio, son inmutables y, por lo tanto, se comportan como tipos por valor.

b. La conversión entre números

La conversión entre números consiste en transformar una variable de un tipo en otro tipo. La conversión se puede hacer a un tipo superior o inferior. Si se utiliza una conversión a un tipo inferior, puede haber una pérdida de información. Por ejemplo, la conversión de un tipo `double` a un tipo `long` hará que, al menos, la parte decimal del valor se pierda. Por esta razón, el compilador requiere en este caso que se indique explícitamente si desea realizar esta operación. Para hacer esto, debe anteponer, al elemento que desea convertir, el tipo que desea obtener, colocándolo entre paréntesis.

```
float superficie;  
superficie=2356.8f;  
//Conversión implícita  
double granSuperficie=superficie;  
//Conversión explícita  
int aproximacion=(int)superficie;
```

La variable `aproximacion` contiene el valor 2356.

La conversión a un tipo superior se hace sin riesgo de pérdida de información y, por lo tanto, se puede realizar directamente mediante una simple asignación.

La siguiente tabla resume las posibles conversiones e indica si deben ser explícitas (😞) o no (😊).

		Tipo de datos que hay que obtener

			byte	short	int	long	float	double	char
Tipo de datos de origen	byte			😊	😊	😊	😊	😊	😊
	short		😞		😊	😊	😊	😊	😊
	int		😞	😞		😊	😊	😊	😊
	long		😞	😞	😞		😊	😊	😞
	float		😞	😞	😞	😞		😊	😞
	double		😞	😞	😞	😞	😞		😞
	char		😞	😊	😊	😊	😊	😊	

Observe la presencia del tipo `char`. Como recordatorio, un `char` se almacena en Unicode con dos bytes.

c. La conversión a una cadena de caracteres

Las funciones de conversión al tipo cadena de caracteres se hace con la clase `String`. El método de clase `valueOf` asegura la conversión de un valor de un tipo básico a una cadena de caracteres.

<code>static String valueOf(boolean b)</code>	Returns the string representation of the boolean argument.
<code>static String valueOf(char c)</code>	Returns the string representation of the char argument.
<code>static String valueOf(char[] data)</code>	Returns the string representation of the char array argument.
<code>static String valueOf(char[] data, int offset, int count)</code>	Returns the string representation of a specific subarray of the char array argument.
<code>static String valueOf(double d)</code>	Returns the string representation of the double argument.
<code>static String valueOf(float f)</code>	Returns the string representation of the float argument.
<code>static String valueOf(int i)</code>	Returns the string representation of the int argument.
<code>static String valueOf(long l)</code>	Returns the string representation of the long argument.
<code>static String valueOf(Object obj)</code>	Returns the string representation of the Object argument.

En ciertas situaciones, el uso de estas funciones es opcional, ya que la conversión se lleva a cabo implícitamente. Este es el caso, por ejemplo, cuando una variable de un tipo básico se concatena con una cadena de caracteres. Ambas versiones del código tienen el mismo efecto.

- Versión 1

```
double precioNeto;
precioNeto=152;
String resumen;
resumen="el importe del pedido es: " +
        precioNeto*1.21;
```

- Versión 2

```
double precioNeto;
precioNeto=152;
String resumen;
resumen="el importe del pedido es: " +
        String.valueOf(precioNeto*1.21);
```

Para obtener más información sobre el tema, puede consultar la sección Las cadenas de caracteres - Formatear una cadena, anteriormente en este capítulo.

d. La conversión desde una cadena de caracteres

A menudo sucede que un valor numérico está disponible en una aplicación en forma de una cadena de caracteres (entrada del usuario, lectura de archivos, etc.).

Para poder ser manipulado por la aplicación, debe convertirse a un tipo numérico. Este tipo de conversión es accesible a través de las clases equivalentes a los tipos básicos. Permiten la manipulación de valores numéricos en forma de objetos. Cada tipo básico tiene su clase asociada.

Tipo básicos	Clase correspondiente
byte	Byte

short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Estas clases se llaman clases *wrappers* porque se usan para «envolver» los tipos básicos de lenguaje en un objeto. Se pueden usar como clases normales, creando una instancia a partir del método de clase `valueOf`; el constructor no se usa desde Java 9. Están disponibles diferentes sobrecargas del método `valueOf`:

```
Integer entero=Integer.valueOf(10);
entero=Integer.valueOf("10");
```

Esta solución se puede evitar gracias al mecanismo llamado *autoboxing* del compilador.

Este mecanismo permite la asignación de un tipo básico del lenguaje a una variable del tipo *wrapper* correspondiente.

```
Integer entero=10;
```

- ☐ Atención, el comportamiento de las instancias de estas clases wrappers es idéntico al tipo `String`. Son inmutables.

El mecanismo inverso, llamado *unboxing*, permite la conversión automática de un tipo *wrapper* a un tipo básico. La variable `entera` del ejemplo anterior se puede asignar a una

variable de tipo `int`.

```
int x=entier;
```

Estas clases proporcionan un método `parseXxx`, que acepta como argumento una cadena de caracteres y devuelve un valor en el tipo básico asociado con la clase.

Clase	Método
Byte	<code>public static byte parseByte(String s)</code>
Short	<code>public static short parseShort(String s)</code>
Integer	<code>public static int parseInt(String s)</code>
Long	<code>public static long parseLong(String s)</code>
Float	<code>public static float parseFloat(String s)</code>
Double	<code>public static double parseDouble(String s)</code>
Boolean	<code>public static boolean parseBoolean(String s)</code>

Para recordar cómo proceder para realizar una conversión, debemos aplicar el siguiente principio: el método que se ha de utilizar se encuentra en la clase correspondiente al tipo de datos que queremos obtener.

12. La inferencia de tipos

La inferencia de tipos es un mecanismo que le deja al compilador determinar el tipo de una variable, basándose en la expresión que permite que se inicialice. Es un mecanismo muy interesante para la programación avanzada en Java, en particular con expresiones lambda. El objetivo principal es simplificar las líneas de código sin molestarse en definir explícitamente el tipo de ciertas variables. Sin embargo, es un mecanismo simple que tiene su lugar en este capítulo que trata las variables y sus tipos.

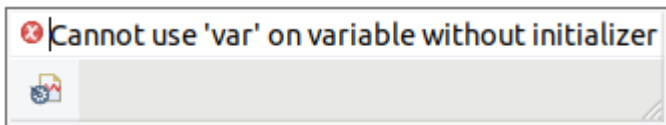
La inferencia de tipos no significa que ya no haya que escribir más. La diferencia es que es el compilador el que define el tipo de la variable.

Para implementar la inferencia de tipos, es necesario declarar la variable con la palabra clave `var`, y ya no con su tipo, como se muestra en el siguiente ejemplo:

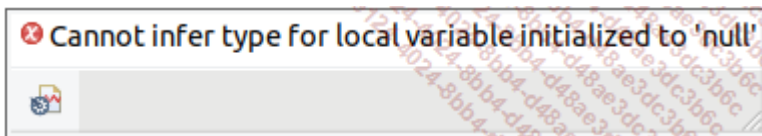
```
var cadena = "Hola, soy una cadena de caracteres";
```

Hay algunas reglas que es preciso seguir para que la inferencia de tipos funcione:

- La variable se debe inicializar en su declaración, para que el compilador pueda determinar su tipo, con objeto de evitar tener el siguiente error:



- La variable no se puede inicializar con `null` porque este valor no se corresponde con ningún tipo, para evitar tener el siguiente error:



- ☐ Para descubrir un uso concreto de la inferencia de tipo, consulte el capítulo Las expresiones lambda.

Las constantes

En una aplicación, con frecuencia sucede que se utilizan valores numéricos o cadenas de caracteres, que no se modificarán durante el funcionamiento de la aplicación. Para facilitar la lectura del código, es recomendable definir estos valores en forma de constantes.

La definición de una constante se realiza agregando la palabra clave `final` antes de la declaración de una variable. Es obligatorio inicializar la constante en el momento de su declaración (este es el único lugar donde es posible hacer una asignación a la constante).

```
double final PORCIVA = 1.21;
```

La constante se puede usar en el código en lugar del valor literal que representa.

```
precioIVA = precioNeto * PORCIVA;
```

Las reglas relativas a la vida útil y el alcance de las constantes son idénticas a las relativas a las variables.

El valor de una constante también se puede calcular a partir de otra constante.

```
final int TOTAL = 100;  
final int MEDIO = TOTAL / 2;
```

Muchas constantes ya están definidas a nivel del lenguaje Java. Se definen como miembros `static` de las muchas clases del lenguaje. Por convención, los nombres de las constantes se escriben completamente en mayúsculas.

Las enumeraciones

Una enumeración nos permitirá definir un conjunto de constantes que están funcionalmente vinculados entre sí. La declaración se realiza de la siguiente manera. :

```
public enum Dia
{
    DOMINGO,
    LUNES,
    MARTES,
    MIERCOLES,
    JUEVES,
    VIERNES,
    SABADO
}
```

El primer valor de la enumeración se inicializa a cero. Las siguientes constantes se inicializan con un incremento de uno. Por lo tanto, la declaración anterior se podría haber escrito :

```
public class Dia
{
    public static final int DOMINGO=0;
    public static final int LUNES=1;
    public static final int MARTES=2;
    public static final int MIERCOLES=3;
    public static final int JUEVES=4;
    public static final int VIERNES=5;
    public static final int SABADO=6;
}
```

Esto es aproximadamente lo que hace el compilador cuando analiza el código de enumeración.

De hecho, la declaración de una enumeración es una declaración de clase «disfrazada». Esta clase hereda implícitamente de la clase `java.lang.Enum`. Los elementos definidos en la enumeración son las únicas instancias posibles de esta clase. Como cualquier clase, puede contener atributos, constructores y métodos. El siguiente ejemplo de código muestra estas posibilidades.

```
public enum Dalton
{
    JOE (1.40, 52),
    WILLIAM (1.68, 72),
    JACK (1.93, 83),
    AVERELL (2.13, 89);

    private final double altura;
    private final double peso;

    private Dalton(double altura, double peso)
    {
        this.altura = altura;
        this.peso = peso;
    }
    public double altura() { return altura; }
    public double peso() { return peso; }

    double imc()
    {

        return peso/(altura+altura);
    }
}
```

El constructor se usa implícitamente para inicializar las constantes para cada uno de los elementos de la enumeración. El constructor de una enumeración se debe declarar como `private`. Varios métodos definidos en la clase base (`java.lang.Enum`)

proporcionan información sobre los elementos de la enumeración. El método `toString` devuelve una cadena de caracteres que representa el nombre de la constante de la enumeración.

```
Dalton d = Dalton.JACK;  
System.out.println(d.toString());
```

El método `valueOf` realiza la operación opuesta proporciona uno de los elementos de la enumeración, cuyo nombre se indica mediante la cadena de caracteres que se pasa como argumento.

```
d = Dalton.valueOf("JOE");  
System.out.println("peso: "+ d.peso());  
System.out.println("altura: "+ d.altura());
```

El método `values` devuelve en forma de array todos los valores posibles de la enumeración.

```
System.out.println("los hermanos Dalton");  
for(Daltons dalton: Dalton.values())  
{  
    System.out.println(dalton.toString());  
}
```

Una vez definida, una enumeración se puede usar como un nuevo tipo de datos. Por lo tanto, puede declarar una variable con su enumeración como tipo.

```
Dia descanso;
```

Es posible usar la variable asignándole uno de los valores definidos en la enumeración.

```
descanso=Dia.LUNES;
```

Cuando hace referencia a un elemento en su enumeración, debe precederlo con el nombre de la enumeración, como en el ejemplo anterior. Asignar a la variable algo diferente a uno de los valores contenidos en la enumeración está prohibido y causa un error de

compilación.

La declaración de una enumeración no puede hacerse en un método. Por otro lado, se puede declarar en una clase, pero en este caso será necesario prefijar el nombre de la enumeración con el nombre de la clase en la que se define durante su uso. Para que la enumeración sea autónoma, es suficiente declararla en su propio archivo.

El ámbito de una enumeración sigue las mismas reglas que el de las variables.

Una variable enumerada se puede usar fácilmente en un `switch ... case`. En este caso, no es necesario que el nombre de la enumeración preceda a sus miembros.

```
switch (descanso)
{
    case LUNES:
    case MARTES:
    case MIERCOLES:
    case JUEVES:
        System.out.println("C\ 'es duro trabajar ");
        break;
    case VIERNES:
        System.out.println("¡Pronto es fin de semana!");
        break;
    case SABADO:
        System.out.println("¡Por fin! ");
        break;
    case DOMINGO:
        System.out.println("Vuelta a empezar");
        break;
}
```

Para más detalles, diríjase a la sección Las estructuras de control, un poco más adelante en este capítulo.

Los argumentos de un programa

1. Funcionamiento

Es posible pasar argumentos a un programa para que adapte su comportamiento, de acuerdo con los valores proporcionados.

Los argumentos suministrados están disponibles en el array de cadenas denominadas `args` en el argumento del método `main`.

Para pasar argumentos durante la ejecución del programa, simplemente ejecute el programa con la herramienta `java` y agregue, después del nombre del programa, los argumentos en la misma línea de comando. El separador es el espacio. Por lo tanto, el siguiente comando inicializa el argumento `args` con un array compuesto por tres celdas:

```
Java MyProgram arg1 arg2 arg3
```

El array `args` contiene las siguientes cadenas: `arg1`, `arg2` y `arg3`.

Si el valor de un argumento contiene un espacio, es suficiente delimitar el valor con comillas:

```
Java MyProgram arg1 arg2 "arg 3"
```

El array `args` en este caso contiene las siguientes cadenas: `arg1`, `arg2` y `arg3`.

Si un argumento contiene comillas, simplemente escape con el carácter `\`:

```
Java MyProgram arg1 arg2 "arg \" 3 \""
```



El array `args` en este caso contiene las siguientes cadenas: `arg1`, `arg2` y `arg "3"`.
Material para descargar

Para obtener más información sobre el manejo de un array, consulte la sección Los arrays, que hemos tratado con anterioridad en este capítulo.

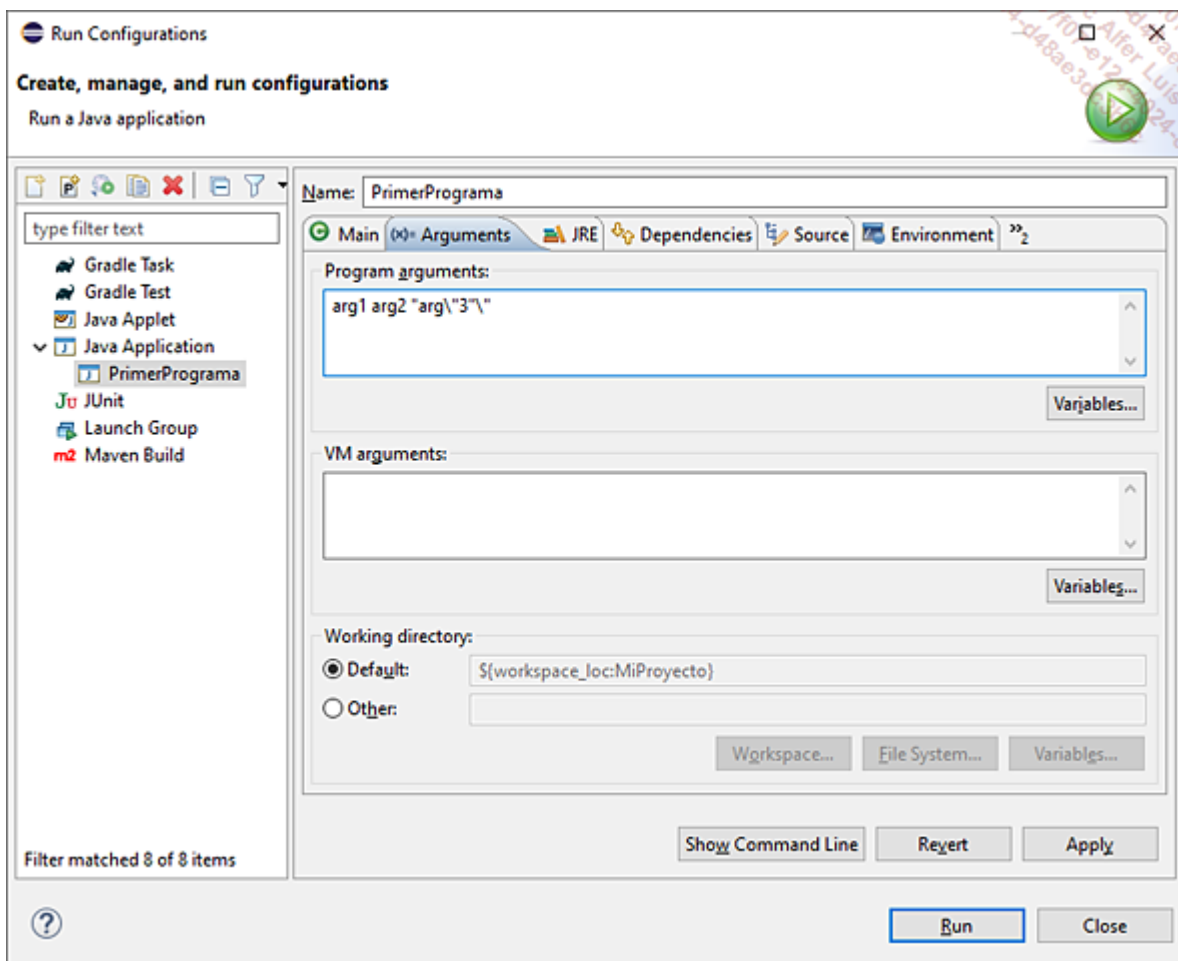
2. Utilización en Eclipse

Estos son los pasos que hay que seguir para definir argumentos en un programa con Eclipse:

- Cree una clase `MiPrograma` con el siguiente código:

```
public class PrimerPrograma {  
    public static void main(String[] args) {  
        for(String arg:args) {  
            System.out.println(arg);  
        }  
    }  
}
```

- Ejecute su programa haciendo clic derecho en la clase y seleccionando el menú **Run as - 1 Java Application**. Esta acción le permite crear fácilmente una configuración de ejecución modificable para agregar argumentos en particular.
- Edite esta configuración de ejecución haciendo clic derecho en la clase y seleccionando el menú **Run As - Run Configurations....** Aparece la siguiente pantalla:



- Ubíquese en el nodo correspondiente a su aplicación en el árbol de la izquierda (**Java Application - PrimerPrograma**).
- Muestre la pestaña **Arguments**.
- Defina los argumentos de su programa (**Program arguments**).
- Pulse el botón **Run** y visualice el resultado en la consola.

Los operadores

Los operadores son palabras claves del lenguaje que permiten ejecutar operaciones en el contenido de ciertos elementos, en general variables, constantes, valores literales o retornos de funciones. La combinación de uno o varios operadores y elementos en los cuales los operadores van a apoyarse se llama una expresión. Estas expresiones se evalúan en tiempo de ejecución en función de los operadores y de los valores asociados.

Java dispone de dos tipos de operadores:

- Los operadores unarios, que trabajan solo en un operando.
- Los operadores binarios, que necesitan dos operandos.

Se pueden distribuir los operadores en siete categorías.

1. Los operadores unarios

Operador	Acción
-	Valor negativo
~	Complemento a uno (inversión de bits)
++	Incremento
--	Decremento
!	Negación

Solo se puede utilizar el operador `!` (exclamación) en variables de tipo `boolean` o en expresiones que producen un tipo `boolean` (comparación).

Los operadores unarios se pueden usar con la notación prefijada. En este caso, el operador se coloca antes del operando. Con la notación posterior, en este caso el operador se coloca después del operando. La posición del operador determina cuándo se aplica el operador a la variable. Si el operador tiene el prefijo, se aplica al operando antes de que se use en la expresión. Con la notación fijada, el operador solo se aplica a la variable después de usarla en la expresión. Esta distinción puede influir en el resultado de una expresión.

La ejecución del siguiente ejemplo muestra 3 porque el incremento se ejecuta después del uso de la variable por la instrucción `println`:

```
int i;  
i = 3;  
System.out.println (i ++);
```

La ejecución del siguiente ejemplo muestra 4 porque el incremento se ejecuta antes del uso de la variable por la instrucción `println`:

```
int i;  
i = 3;  
System.out.println (++ i);
```

Si la variable no se usa en una expresión, ambas versiones conducen al mismo resultado.

Las dos líneas de código son equivalentes :

```
i++;  
++i;
```

2. Los operadores de asignación

El único operador disponible en esta categoría es el operador `=`. Permite asignar un valor a una variable. Se utiliza el mismo operador sea cual sea el tipo de la variable (numérico, cadena de caracteres...).

Se puede combinar este operador con un operador aritmético, lógico o binario.

La sintaxis siguiente:

```
x+=2;
```

equivale a:

```
x=x+2;
```

3. Los operadores aritméticos

Los operadores aritméticos permiten efectuar cálculos en el contenido de las variables. El tipo del resultado de la operación se corresponde con el tipo más grande de los dos operandos.

Operador	Operación efectuada	Ejemplo	Resultado
+	Suma para valores numéricos o concatenación para cadenas	6+4	10
-	Sustracción	12/06/19	6
*	Multiplicación	3*4	12
/	División	25/03/19	8 o 8.333...
%	Módulo (resto entero de la división)	25 % 3	1

Cabe señalar que, en el caso de la división, hay una sutileza:

Si los dos operandos son enteros, la división operada es una división entera. Entonces el resultado de la operación 25/3 será 8.

Por otro lado, si al menos uno de los dos operandos es un número decimal, entonces el

resultado de la misma operación será $8.333 \dots$. La precisión del resultado depende del tipo más grande en la operación. Consulte la sección Los tipos numéricos decimales para obtener más detalles.

4. Los operadores bit a bit

a. La representación binaria de los enteros

Para usar operadores bit a bit, es necesario comprender la representación binaria de los enteros. El mecanismo es el mismo, sea cual sea el tipo (`byte`, `short`, `int`, `long`). Para facilitar la referencia, el resto de esta sección utilizará el tipo `byte` almacenado en solo ocho bits.

Un bit es la unidad de memoria más básica. Puede contener el valor 0 o el valor 1. Hablamos de representación binaria porque solo son posibles dos valores.

Un `byte`, que es una secuencia de ocho bits, está compuesto por ocho 0/1.

A continuación se dan representaciones binarias válidas de valores de tipo `byte`:

35 base 10 =	0	0	1	0	0	0	1	1
12 base 10 =	0	0	0	0	1	1	0	0
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	128	64	32	16	8	4	2	1

Como puede ver, no todo tiene el mismo peso. El bit derecho es el bit de peso débil y el bit izquierdo es el bit de peso fuerte.

Cuando el bit de peso débil es 1, representa 2^0 en la base 10 o 1. Cuando el siguiente bit que va a la derecha es 1, representa 2^1 en la base 10 o 2. Los siguientes bits representan los valores siguientes en base 10: 8, 16, 32, 64, 128.

Para tipos más grandes (`short`, `int`, `long`), simplemente agregue potencias de dos adicionales: 2^8 , $2^9 \dots 2^{63}$.

Entonces, con esta explicación, los valores de un tipo de `byte` pueden ir de 0 a 255 en la base 10 (de 00000000 a 11111111 en la base 2). El rango de valores posibles para un `byte` es de -128 a +127.

De hecho, solo los 7 bits de la derecha permiten definir el valor. Por lo tanto, el valor máximo es $127 (64 + 32 + 16 + 8 + 4 + 2 + 1)$.

El bit izquierdo se usa como el bit de signo. Si el valor es 0, el byte es un entero positivo. Si el valor es 1, el byte es un entero negativo. Siguiendo esta regla básica, el valor mínimo sería -127 ; ahora bien: el valor mínimo real sería -128 .

Las operaciones básicas, como la adición, dejarán de funcionar:

$$\begin{array}{r}
 -3 \text{ base } 10 = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline \end{array} \\
 + 8 \text{ base } 10 = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline \end{array} \\
 \hline
 5 \text{ base } 10! = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline \end{array}
 \end{array}$$

$2^7 \quad 2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$
 $128 \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1$

El siguiente ejemplo muestra un error en la adición. $-3 + 8$ en base 10 da 5. Con la representación binaria actual, tendríamos el valor -11 .

La representación de enteros negativos es, por lo tanto, un poco más compleja. Es necesario llevar a cabo una operación llamada **complemento a 2** en el valor absoluto de un entero para tener la representación binaria del entero negativo correspondiente. La operación es reversible. Si tiene una representación binaria de un entero negativo, para conocer su valor en la base 10, es suficiente llevar a cabo el complemento a 2.

Para representar el valor -3 en base 2, siga estos:

- Comience determinando la representación binaria del valor 3 en 7 bits: 0000011.
- Invierta todos los bits: 1111100.
- Agregue 1: 1111101.
- Por lo tanto, la representación de -3 es 11111101. Se ha agregado el bit de signo.

El complemento consiste en dos operaciones binarias: invertir los bits y sumar 1.

La operación $-3 + 8$ en binario se convierte en la siguiente operación:

$$\begin{array}{r}
 -3 \text{ base } 10 = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline \end{array} \\
 + 8 \text{ base } 10 = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline \end{array} \\
 \hline
 5 \text{ base } 10 = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline \end{array} \\
 \begin{array}{cccccccc}
 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 128 & 64 & 32 & 16 & 8 & 4 & 2 & 1
 \end{array}
 \end{array}$$

Esta representación de los enteros negativos permite resolver este problema.

b. Las operaciones lógicas

Ahora que conoce las representaciones binarias de los enteros, es necesario comprender las operaciones lógicas que es posible hacer en bits. Hay tres operaciones básicas que es preciso conocer: Y lógico, O lógico y O exclusivo..

Y

El Y permite llevar a cabo una operación entre dos bits de acuerdo con la siguiente regla: si los dos bits valen 1, el resultado es 1; de lo contrario, el resultado es 0. Las diferentes posibilidades se resumen en la siguiente tabla:

Y	1	0
1	1	0
0	0	0

O

El O permite que se realice una operación entre dos bits de acuerdo con la siguiente regla: si los dos bits valen 0, entonces el resultado es 0; si no, el resultado es 1. Las diferentes posibilidades se resumen en la siguiente tabla:

O	1	0
1	1	1
0	1	0

O exclusivo

El O exclusivo permite realizar una operación entre dos bits de acuerdo con la siguiente regla: solo si uno de los dos bits vale 1, entonces el resultado es 1; de lo contrario, el resultado es 0. Las diferentes posibilidades se resumen en la siguiente tabla:

O exclusivo	1	0
1	0	1
0	1	0

Los operadores

Estos operadores efectúan operaciones únicamente con enteros (byte, short, integer, long). Trabajan a nivel de bit en las variables que manipulan.

Operador	Operación efectuada	Ejemplo	Resultado
&	Y binario	45 & 22	4
	O binario	99 46	111
^	O exclusivo	99 ^ 46	
>>	Desplazamiento firmado hacia la derecha (división por 2)	26>>1 -26>>1	13 -13
<<	Desplazamiento hacia la izquierda (multiplicación por 2)	26<<1	52
>>>	Desplazamiento no firmado hacia la derecha	26>>>1 -26>>>1	13 21474835635

Las operaciones binarias (Y binarias, O binarias, O exclusivas) se llevan a cabo bit a bit. Esto significa que la operación se realiza en cada bit del mismo peso. A continuación se muestra un ejemplo de operación $45 \ \& \ 22$:

45 base 10 =	0	0	1	0	1	1	0	1
& 22 base 10 =	0	0	0	1	0	1	1	0
4 base 10 =	0	0	0	0	0	1	0	0

La operación de desplazamiento con signo a la derecha consiste en agregar bits a la izquierda correspondientes al valor del bit de signo (el bit de peso más fuerte). Agregamos tantos como el número definido a la derecha de la operación. Agregar bits a la izquierda significa perder otros tantos bits a la derecha. La operación $26 \gg 1$ permite agregar un 0 (valor del bit de signo) a la izquierda y perder un bit a la derecha. La operación $-26 \gg 1$ le permite agregar un 1 (valor del bit de signo) a la izquierda y perder un bit a la derecha.

La operación de desplazamiento con signo a la izquierda consiste en agregar 0 a la derecha. Agregamos tantos como el número definido a la derecha de la operación. Agregar bits a la derecha significa perder tantos bits a la izquierda. La operación $26 \ll 1$ permite agregar un 0 a la derecha y perder un bit a la izquierda.

La operación de desplazamiento a la derecha sin signo consiste en agregar solo 0 a la izquierda. El mecanismo sigue siendo el mismo que el de la operación de desplazamiento a la derecha.

5. Los operadores de comparación

Se utilizan operadores de comparación en las estructuras de control de una aplicación. Devuelven un valor de tipo `boolean` en función del resultado de la comparación efectuada. A continuación, la estructura de control utilizará este valor.

Operador	Prueba efectuada	Ejemplo	Resultado
<code>==</code>	Igualdad	<code>2 == 5</code>	<code>false</code>

<code>!=</code>	Desigualdad	<code>2 != 5</code>	<code>true</code>
<code><</code>	Inferior a	<code>2 < 5</code>	<code>true</code>
<code>></code>	Superior a	<code>2 > 5</code>	<code>false</code>
<code><=</code>	Inferior o igual a	<code>2 <= 5</code>	<code>true</code>
<code>>=</code>	Superior o igual a	<code>2 >= 5</code>	<code>false</code>
<code>instanceof</code>	Comparación del tipo de la variable con el tipo indicado	<code>O1 instanceof Cliente</code>	<code>true</code> si la variable <code>O1</code> hace referencia a un objeto creado a partir de la clase <code>cliente</code> o de una subclase

El operador `instanceof` se comenta con más detalle en el capítulo Programación orientada a objetos.

6. El operador de concatenación

El operador `+` (más), ya utilizado para la suma, se usa también para la concatenación de cadenas de caracteres. El funcionamiento del operador viene determinado por el tipo de los operandos. Si uno de los operandos es del tipo `String`, el operador `+` (más) efectúa una concatenación con, eventualmente, una conversión implícita del otro operando a cadena de caracteres.

Un pequeño inconveniente del operador `+` (más) es que no destaca por su velocidad para las concatenaciones. De hecho, no es realmente el operador la fuente del problema, sino la técnica utilizada por Java para gestionar las cadenas de caracteres (no se pueden modificar después de su creación). Si tenemos que ejecutar varias concatenaciones en una cadena, es preferible utilizar la clase `StringBuilder` (o `StringBuffer` solo si trabaja en entornos multi-thread).

Ejemplo

```
long duracion;
String liebre;
String tortuga="";
long principio, fin;
principio = System.currentTimeMillis();
for (int i = 0; i <= 10000; i++)
{
    tortuga = tortuga + " " + i;
}
fin = System.currentTimeMillis();
duracion = fin-principio;
System.out.println("duración para la tortuga: " + duracion + "ms");
principio = System.currentTimeMillis();
StringBuilder sb = new StringBuilder();
for (int i = 0; i <= 10000; i++)
{
    sb.append(" ");
    sb.append(i);
}
liebre = sb.toString();
fin = System.currentTimeMillis();
duracion = fin-principio;
System.out.println("duración para la liebre: " + duración + "ms");
if (liebre.equals(tortuga))
{
    System.out.println("las dos cadenas son idénticas");
}
```

Resultado de la carrera:

```
duración para la tortuga: 249ms
duración para la liebre: 10ms
las dos cadenas son idénticas
```

¡Este resultado no necesita comentarios! Observe igualmente el número de iteraciones a nivel del bucle. Para algunas concatenaciones sucesivas, el operador + se adapta bien.

7. Los operadores lógicos

Los operadores lógicos permiten combinar las expresiones en estructuras condicionales o estructuras de bucle.

Operador	Operación	Ejemplo	Resultado
&	Y lógico	test1 & test2	true si test1 y test2 valen true
	O lógico	test1 test2	true si test1 o test2 valen true
^	O exclusivo	test1 ^ test2	true si test1 o test2 es true pero no si los dos valen true al mismo tiempo
!	Negación	! test1	Invierte el resultado del test
&&	Y lógico	test1&&	Como & pero test2 solo se evaluará si test1 vale true
	O lógico	test1 test2	Como pero test2 solo se evaluará si test1 vale false

Deberemos tener cuidado con los operadores && (doble ampersand) y || (tubería doble) porque la expresión que probaremos en segundo lugar (test2 en nuestro caso) a veces podrá no ejecutarse. Si esta segunda expresión modifica una variable, esta solo se modificará en los casos siguientes:

- Si test1 vale true en el caso del &&.
- Si test1 vale false en el caso del ||.

☐

Por lo tanto, es preferible evitar llamar a funciones que modifican cierta información en una expresión con operadores lógicos. El interés de los

operadores `&&` y `||` es evitar pruebas innecesarias y son preferibles frente a sus homólogos `&` y `|`.

8. Orden de evaluación de los operadores

Cuando se combinan varios operadores en una expresión, se evalúan en un orden muy preciso. Los incrementos y decrementos prefijados se ejecutan en primer lugar; a continuación, las operaciones aritméticas, las operaciones de comparación, los operadores lógicos y, por último, las asignaciones.

Los operadores aritméticos también tienen entre ellos un orden de evaluación en una expresión. El orden de evaluación es el siguiente:

- Negación (`-`)
- Multiplicación y división (`*`, `/`)
- Módulo (`%`)
- Suma y resta (`+`, `-`), concatenación de cadenas (`+`)

Si se necesita establecer un orden de evaluación diferente, tendremos que ubicar las expresiones con prioridad entre paréntesis, como en el siguiente ejemplo:

```
X= (z * 4) ^ (y * (a + 2));
```

En una expresión podemos utilizar tantos niveles de paréntesis como deseemos. Sin embargo, es importante que la expresión contenga tantos paréntesis de cierre como de apertura, pues en caso contrario el compilador generará un error.

Las estructuras de control

1. Presentación

Las estructuras de control permiten controlar la ejecución de las instrucciones en nuestro código. Java dispone de dos tipos de estructuras:

- Las estructuras de decisión: orientarán la ejecución del código en función de los valores que pueda tomar una expresión de evaluación.
- Las estructuras de bucle: permiten ejecutar una sección de código cierto número de veces, hasta que o mientras una condición se cumpla.



2. Estructuras de decisión

Existen dos soluciones posibles.

a. Estructura if

Se pueden usar hasta cuatro sintaxis para la instrucción if.

```
if (condición)instrucción;
```

Si la condición vale `true`, la instrucción se ejecuta. La condición debe ser una expresión que, una vez evaluada, debe devolver un valor booleano `true` o `false`. Con esta sintaxis, solo la instrucción situada después del `if` será ejecutada si la condición vale `true`. Para ejecutar varias instrucciones en función de una condición, debemos utilizar la sintaxis siguiente.

```
if (condición)
```

```
{  
    instrucción 1;  
    ...  
    instrucción n;  
}
```

En este caso, se ejecutará el grupo de instrucciones situado entre las llaves si la condición vale `true`.

También podemos especificar una o varias instrucciones que se ejecutarán si la condición vale `false` añadiendo un bloque `else`.

```
if (condición)  
{  
    instrucción 1;  
    ...  
    instrucción n;  
}  
else  
{  
    instrucción 1;  
    ...  
    instrucción n;  
}
```

También podemos anidar las condiciones en la sintaxis siguiente.

```
if (condición1)  
{  
    instrucción 1  
    ...  
    instrucción n  
}  
else if (condición 2)  
{  
    instrucción 1  
    ...  
    instrucción n  
}
```

```
}  
else  
{  
    instrucción 1  
    ...  
    instrucción n  
}
```

En este caso, se comprueba la primera condición. Si es verdadera, se ejecuta el bloque de código correspondiente, y si no, se comprueba la siguiente y así sucesivamente. Si no se verifica ninguna condición, se ejecutará el bloque de código definido a continuación del `else`. La instrucción `else` no es obligatoria en esta estructura. Por ello, es posible que no se llegue a ejecutar instrucción alguna si ninguna de las condiciones es verdadera.

b. Estructura ternaria

Esta estructura particular se corresponde con una declaración `if...else`. Solo se puede usar para asignar un valor. Es una estructura concisa que sustituye de manera ventajosa la instrucción `if...else`.

```
String mensaje = condición ? "si true" : "si false";
```

El `?` desencadena la evaluación de la condición. Si la condición vale `true`, entonces es el valor proporcionado después de `?` el que se vuelve a activar; de lo contrario, es el valor proporcionado después de `:` el que se devuelve.

La instrucción equivalente a un `if...else` es la siguiente:

```
if(condición)  
    mensaje="si true";  
else  
    mensaje ="si false";
```

c. Estructura switch

La estructura `switch` permite un funcionamiento equivalente, pero ofrece una mejor legibilidad del código. La sintaxis es la siguiente:


```
switch (expresión)
{
    case valor1:
        instrucción 1
        ...
        instrucción n
        break;
    case valor2:
        instrucción 1
        ...
        instrucción n
        break;
    default:
        instrucción 1
        ...
        instrucción n
}
```

El valor de la expresión se evalúa al principio de la estructura (por el `switch`) y, a continuación, se compara el valor obtenido con el valor especificado en el primer `case`.

Si los dos valores son iguales, entonces el bloque de código adyacente se ejecuta.

En caso contrario, se compara el valor obtenido con el valor del `case` siguiente. Si hay correspondencia, se ejecuta el bloque de código y así sucesivamente hasta el último `case`.

Si no se encuentra ningún valor concordante en los diferentes `case`, se ejecuta el bloque de código especificado por la palabra clave `default`. Cada uno de los bloques de código debe terminarse con la instrucción `break`. Si no es el caso, la ejecución continuará por el bloque de código siguiente hasta encontrar una instrucción `break` o hasta el fin de la estructura `switch`. Esta solución se puede utilizar para poder ejecutar un mismo bloque de código cuando coinciden distintos valores.

El valor que se debe comprobar puede estar contenido en una variable, pero también puede ser el resultado de un cálculo. En tal caso, solo se ejecuta el cálculo una única vez al principio del `switch`. El tipo del valor probado puede ser numérico, entero, carácter, cadena de caracteres o enumeración. Por supuesto, es necesario que el tipo de la variable

comparada corresponda al tipo de los valores en los diferentes `case`.

Si la expresión es de tipo cadena de caracteres, se utiliza el método `equals` para verificar si es igual a los valores de los distintos `case`. La comparación hace por tanto distinción entre mayúsculas y minúsculas.

```
System.out.println("Responda sí, no o cualquier otra cosa:");
BufferedReader br;
br = new BufferedReader (new InputStreamReader(System.in));
String respuesta = "";
respuesta = br.readLine();
switch (respuesta)
{
    case 's':
    case 'S':
        System.out.println("respuesta positiva");
        break;
    case 'n':
    case 'N':
        System.out.println("respuesta negativa");
        break;
    default:
        System.out.println("respuesta errónea");
}
```

3. Las estructuras de bucle

Disponemos de tres estructuras:

```
while (condición)
do ... while (condición)
for
```

Todas tienen como objetivo ejecutar un bloque de código un determinado número de veces en función de una condición.

a. Estructura while

Esta estructura ejecuta un bloque de manera repetitiva mientras la condición sea verdadera.

```
while (condición)
{
    instrucción 1;
    ...
    instrucción n;
}
```

La condición se evalúa antes del primer ciclo. Si el resultado es falso en este momento, el bloque de código no se ejecuta. Después de cada ejecución del bloque de código, la condición vuelve a evaluarse para comprobar si es necesario volver a ejecutar el bloque de código. Se recomienda que la ejecución del bloque de código contenga una o varias instrucciones capaces de hacer evolucionar la condición. Si no es el caso, el bucle se ejecutará sin fin. Bajo ningún concepto se debe poner el carácter ; al final de la línea del `while` porque, en este caso, el bloque de código dejará de asociarse con el bucle.

```
int i=0;
while (i<=10) //No hay ; aquí. El bloque adyacente no está
              //relacionado con el bucle
{
    System.out.println(i);
    i++;
}
```

b. Estructura do ... while

```
do
{
    instrucción 1;
    ...
    instrucción n;
}
while (condición);
```

Esta estructura tiene un funcionamiento idéntico a la anterior. La diferencia reside en el hecho de que la condición se examina después de la ejecución del bloque de código. Nos permite garantizar que el bloque de código se ejecutará al menos una vez, ya que la condición se comprueba por primera vez después de la primera ejecución del bloque de código. Si la condición es verdadera, el bloque se ejecuta de nuevo hasta que la condición sea falsa. Tenga cuidado en no olvidar el punto y coma después del `while`; en caso contrario, el compilador detectará un error de sintaxis.

```
do
{
    System.out.println(i);
    i++;
}
while(i<=10);
```

c. Estructura for

Cuando sabemos el número de iteraciones que debe realizar un bucle, es preferible utilizar la estructura `for`. Para poder utilizar esta instrucción, se debe declarar una variable de contador. Es posible declarar esta variable en la estructura `for` o en el exterior, y en tal caso se debe declarar antes de la estructura `for`.

La sintaxis general es la siguiente:

```
for(inicialización;condición;instrucción de iteración)
{
    instrucción 1;
    ...
    instrucción n;
}
```

La inicialización se ejecuta una única vez durante la entrada en el bucle. La condición se evalúa en el momento de la entrada en bucle y, a continuación, con cada iteración. El resultado de la evaluación de la condición determina si el bloque de código se

ejecuta. Si el resultado es verdadero, se realiza una nueva iteración del bucle. Después de la ejecución del bloque de código, se realiza la de la iteración. A continuación se comprueba de nuevo la condición y así sucesivamente mientras la condición sea verdadera.

El ejemplo muestra dos bucles `for` en acción para visualizar una tabla de multiplicar.

```
int multiplicador;
for(multiplicador=1;multiplicador<=10;multiplicador++)
{
    for (int tabla = 1; tabla <= 10; tabla++)
    {
        System.out.print(table*multiplicador + "\t");
    }
    System.out.println();
}
```

Obtenemos el resultado siguiente:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Otra sintaxis del bucle `for` permite ejecutar un bloque de código para cada elemento contenido en un array o en una instancia de clase al implementar la interfaz `Iterable`. Es el bucle `foreach`. La sintaxis general de esta instrucción es la siguiente:

```
for (tipo elementoDelArray : array)
{
    instrucción 1
    ...
    instrucción n
}
```

```
}
```

No hay noción de contador en esta estructura, ya que efectúa por sí misma las iteraciones en todos los elementos presentes en el array o la colección.

La variable declarada en la estructura (llamada `elementoDelArray` en el ejemplo) sirve para extraer uno a uno los elementos del array o de la colección para que el bloque de código pueda manipularlos. Por supuesto, hace falta que el tipo de la variable sea compatible con el tipo de los elementos almacenados en el array o la colección. Se debe declarar la variable obligatoriamente en la estructura `for` y no en el exterior. Solo se podrá utilizar en el interior de la estructura. No obstante, no tenemos que preocuparnos del número de elementos porque la estructura es capaz de gestionar por sí misma el desplazamiento en el array o la colección. ¡Aquí mostramos un pequeño ejemplo para aclarar la situación!

Con un bucle clásico:

```
String[] array={"rojo","verde","azul","blanco"};
for (int cpt = 0; cpt < array.length; cpt++)
{
    System.out.println(array[cpt]);
}
```

Con el bucle `for` de iteración:

```
String[] array={"rojo","verde","azul","blanco"};
for (String s: array)
{
    System.out.println(s);
}
```

- ☐ El código ubicado en el interior de esta estructura `for` no debe modificar el contenido de la colección porque el elemento recorrido es de solo lectura. En el caso de un array o una colección de objetos, es posible cambiar los valores de las propiedades de estos objetos (la variable que referencia al objeto no se modifica).

Está prohibido agregar o eliminar elementos durante el recorrido de la colección. Este problema no se produce con un array. El tamaño del array, al ser fijo, hace que sea imposible agregar o eliminar elementos. El siguiente código pone de relieve esta limitación cuando se recorre un `ArrayList`. Agregar un elemento a `ArrayList` durante la iteración produce una excepción de tipo `ConcurrentModificationException`.

```
ArrayList<String> lst;  
lst=new ArrayList<String>();  
lst.add("cliente 1");  
lst.add("cliente 2");  
lst.add("cliente 3");  
lst.add("cliente 5");  
  
for(String st:lst)  
{  
    System.out.println(st);  
    if(st.endsWith("3"))  
    {  
        lst.add("cliente 4");  
    }  
}
```

4. Interrupción de una estructura de bucle

Tres instrucciones pueden modificar el funcionamiento normal de las estructuras de bucle.

a. `break`

Si se ubica esta instrucción en el interior del bloque de código de una estructura de bucle, provoca la salida inmediata de este bloque. La ejecución continúa por la instrucción situada después de este bloque. La instrucción `break` suele ejecutarse de manera condicional porque, de lo contrario, las instrucciones posteriores dentro del bucle no llegarían nunca a

procesarse.

En el caso de bucles anidados, es posible utilizar la instrucción `break` asociada a una etiqueta. El siguiente código de ejemplo recorre una tabla de dos dimensiones y se detiene cuando se encuentra el valor 0.

```
int[][] puntos = {
    { 10,10,},
    { 0,10 },
    { 45,24 } };

int x=0,y=0;
boolean encontrado=false;

buscar:
for (x = 0; x < puntos.length; x++)
{
    for (y = 0; y < puntos[x].length;y++)
    {
        if (puntos[x][y] == 0)
        {
            encontrado = true;
            break buscar;
        }
    }
}

if (encontrado)
{
    System.out.println("se ha encontrado el resultado"
        + x + "-" + y);
}
else
{
    System.out.println("búsqueda sin resultados");
}
```

b. continue

Esta instrucción permite interrumpir la ejecución de la iteración actual de un bucle y proseguir con la ejecución de la iteración siguiente después de haber comprobado la condición de salida. Como en el caso de la instrucción `break`, se debe ejecutar de manera condicional y acepta, a su vez, el uso de una etiqueta.

He aquí un ejemplo de código que permite calcular el número de números pares consecutivos a partir de la primera celda en un array ordenado.

```
int[] numeros = new int[20];
Random rd = new Random();
for (int i = 0; i < numeros.length; i++) {
    numeros [i]=rd.nextInt(20);
}
Arrays.sort(numeros);
System.out.println(Arrays.toString(numeros));
int numPar;
for(numPar =0; numPar <numeros.length; numPar ++)
{

    if(numeros[numPar]%2==0)

    {
        continue;
    }
    break;
}
System.out.println("número de pares: " + numPar);
```

c. return

La instrucción `return` se utiliza para salir inmediatamente del método en curso de ejecución y proseguir la ejecución por la instrucción siguiente a la que llamó este método. Si está situada en una estructura de bucle, provoca en primer lugar la interrupción inmediata del bucle y en segundo lugar la salida del método en el cual se encuentra el bucle. Si se utiliza esta instrucción en una función cuyo tipo de retorno es diferente a `void` es obligatorio proporcionar a la instrucción `return` un valor compatible con el tipo de retorno de la función. Esta instrucción se abordará en el capítulo que habla de la

Ejercicios

1. Ejercicio 1

Crear una tabla de diez cadenas de caracteres y, a continuación, rellenar esta tabla con direcciones de correo electrónico. Calcular seguidamente, a partir de la información incluida en la tabla, el segmento de mercado de cada uno de los proveedores de acceso.

Pista: en una dirección de correo electrónico, el proveedor es la parte de la dirección situada después del carácter @.

2. Ejercicio 2

Generar tres números aleatorios comprendidos entre 0 y 1000 y, a continuación, verificar si se ha obtenido dos números pares seguidos de un número impar. Si no fuera el caso, repetir hasta tener la combinación par, par, impar. Mostrar, a continuación, el número de ensayos realizados para obtener dicha combinación.

Pista: la clase `Math Random` ofrece un conjunto de métodos que permiten obtener un número aleatorio. Céntrese en el método siguiente leyendo la javadoc:

```
public int nextInt(int bound)
```

Para utilizar este método, es necesario tener un objeto de tipo `Random`:

```
Random rd = new Random();  
rd.nextInt(...);
```

3. Ejercicio 3

Generar un número aleatorio comprendido entre 0 y 1000. A continuación, pedir al usuario que adivine el número escogido por el ordenador. Para ello, debe introducir un número comprendido entre 0 y 1000. Se compara el número introducido con el calculado por el ordenador y se muestra en la consola «es mayor» o «es menor» en función del caso. Se repite la operación hasta que el usuario encuentra el número correcto. Mostrar, a continuación, el número de intentos necesarios para obtener la respuesta correcta.

Pista: para recuperar los caracteres introducidos por teclado, tenemos a nuestra disposición el flujo `System.in`. Por desgracia, dicho flujo no proporciona más que funciones rudimentarias para recuperar los datos introducidos por el usuario (con una lectura carácter a carácter). Por comodidad, conviene utilizar un objeto de tipo `Scanner`. Podemos disponer de una serie de funciones que permiten recuperar valores enteros, decimales, cadenas de caracteres... Estas funciones se denominan `nextXxx`, donde `Xxx` debe remplazarse por el tipo de datos que se espera obtener; por ejemplo, el método `nextInt` para un valor entero, `nextLine` para una cadena de caracteres, etc.

```
String cadena;  
Scanner sc;  
sc=new Scanner(System.in);  
cadena=sc.nextLine();
```

4. Ejercicio 4

Agregar al ejercicio 3 la representación del tiempo que ha tardado el usuario en adivinar la respuesta correcta.

Pista: preste atención a las clases `OffsetTime` y `Duration`.



Anterior

Las estructuras de control

Siguiente

Correcciones

Correcciones

1. Ejercicio 1

A continuación se muestra una propuesta (y esto es solo una propuesta) para resolver este primer ejercicio. El código se complementa con una gran cantidad de comentarios para comprender completamente el enfoque adoptado. El código fue escrito en un solo método para centrarse en el algoritmo. Una mejora puede consistir en dividir este programa en métodos.

```
import java.util.Arrays;

public class Ejercicio1
{

    static String[] direccion;
    static String[] listaBrutaProveedores;
    static String[] listaProveedoresDistintos;
    static int[] numeroClientes;

    public static void main(String[] args)
    {
        //INICIALIZACIÓN DE LA TABLA
        direccion=new String[10];
        direccion[0]="jpp@sfr.fr";
        direccion[1]="tom@gmail.com";
        direccion[2]="fred@sfr.fr";
        direccion[3]="victor@sfr.fr";
        direccion[4]="chris@sfr.fr";
        direccion[5]="robert@orange.fr";
    }
}
```

Material para descargar

```
direccion[6]="paul@sfr.fr";
direccion[7]="lise@gmail.com";
direccion[8]="thierry@eni.fr";
direccion[9]="marie@eni.fr";
```

```
//RECUPERACIÓN DE LOS PROVEEDORES DE CADA DIRECCIÓN
listaBrutaProveedores =new String[10];
for (int i=0;i<direccion.length;i++)
{
    //Para cada dirección, recuperación de la cadena después
del @:
    listaBrutaProveedores[i]=
direccion[i].substring(direccion[i].indexOf('@')+1);
}
System.out.println("Lista bruta de los proveedores:");
System.out.println(Arrays.toString(listaBrutaProveedores));

//RECUPERACIÓN DE LA LISTA DE LOS PROVEEDORES DISTINTOS Y
NÚMERO CLIENTES
//1-Ordenar el array
Arrays.sort(listaBrutaProveedores);
System.out.println("Lista ordenada de proveedores:");
System.out.println(Arrays.toString(listaBrutaProveedores));
//2-Inicializar el array de los proveedores distintos con una
altura de 1 (porque hay al menos 1 proveedor)
//e inicializar el array del número de clientes por proveedor
listaProveedoresDistintos = new String[1];
numeroClientes = new int[1];
//3-Añadir el primer proveedor y asociarle 1 cliente:
listaProveedoresDistintos[0]=listaBrutaProveedores[0];
numeroClientes[0]=1;
//4-Recorrer el array ordenado de los proveedores para
detectar
otros eventuales proveedores
//El recorrido empieza en el índice 1 porque el de índice 0 se
ha añadido
a la lista de proveedores distintos
for(int i=1;i<listaBrutaProveedores.length;i++)
```

```

    {
        //5-Si el proveedor actual es diferente del último
proveedor distinto añadido
        //Es que es un nuevo proveedor.

if(!listaBrutaProveedores[i].equals(listaProveedoresDistintos
[listaProveedoresDistintos.length-1]))
    {
        //Agrandar el array de los proveedores distintos de una
celda
        //y agrandar el array del número de clientes de una
celda

        listaProveedoresDistintos =
Arrays.copyOf(listaProveedoresDistintos,
listaProveedoresDistintos.length+1);
        numeroClientes = Arrays.copyOf(numeroClientes,
numeroClientes.length+1);
        //Añadir este nuevo proveedor al array y asociarle 1
cliente

        listaProveedoresDistintos[listaProveedoresDistintos.
length-1]=listaBrutaProveedores[i];
        numeroClientes[numeroClientes.length-1]=1;
    }
    else
    {
        //Es un proveedor ya referenciado, es suficiente con
añadir
un cliente
        numeroClientes[numeroClientes.length-1]+=1;
    }
}
System.out.println("Lista de los proveedores distintos:");

System.out.println(Arrays.toString(listaProveedoresDistintos));
System.out.println("Número de clientes para cada proveedor");
System.out.println(Arrays.toString(numeroClientes));

```

```

//VISUALIZACIÓN DE LA PARTE DEL MERCADO DE CADA PROVEEDOR:
System.out.println("Partes del mercado:");
for(int i=0;i<listaProveedoresDistintos.length;i++)
{
    System.out.println(String.format("=> %s \t: %d clientes en
%d
(%.1f%%)"

, listaProveedoresDistintos[i]

, numeroClientes[i]

, direccion.length,

(numeroClientes[i]*100.0/direccion.length)));
}
}
}

```

2. Ejercicio 2

El código de este ejercicio empieza, como el anterior, por las inicializaciones.

El bucle `do...while` permite extraer al azar tres números y comprobar sus valores para saber si se obtiene la combinación par/par/impar o es necesario realizar un nuevo intento.

```

import java.util.Random;

public class Ejercicio2
{
    public static void main(String[] args)
    {
        int numeroIntentos=0;
        int num1,num2,num3;
        Random rd = new Random();
        //El bucle do...while se adapta porque es necesario al menos un

```

```

intento
do
{
    //Ordenación de las 3 variables
    num1=rd.nextInt(1000);
    num2=rd.nextInt(1000);
    num3=rd.nextInt(1000);
    //Incremento del número de intentos
    numeroIntentos++;
    //Mostrar el intento
    System.out.println("número 1:" + num1 + " número 2:" +
num2 + " número 3:" + num3);
    //Volvemos a empezar si num1 es impar o si num2 es impar
o si num3 es par.
    //para esto, se evalúa el resto de la división entera por
2.

    //si el resto es igual a 1, el número es impar
    //si el resto es igual a 0, el número es par
} while(num1 % 2==1 || num2 % 2==1 || num3 % 2==0);

    System.out.println("Resultado obtenido en " + numeroIntentos
+ " intento(s)");
}
}

```

3. Ejercicio 3

El código de este ejercicio empieza, de nuevo, por las inicializaciones.

El bucle `do . . . while` gestiona, a continuación, el valor entero introducido por el usuario y lo compara con el número secreto para, seguidamente, mostrar el mensaje correspondiente al resultado de dicha comparación. Este bucle se repite mientras el número introducido sea diferente al número secreto.

A la salida del bucle, se muestra el resultado.

```

import java.util.Random;

```



```
import java.util.Scanner;

public class Ejercicio3
{
    public static void main(String[] args)
    {
        int numIntentos=0;
        int secreto;
        int numIntento;
        Random rd = new Random();

        secreto=rd.nextInt(1000);
        Scanner sc=new Scanner(System.in);
        System.out.println("Introduzca un entero entre 0 y 1000");
        do
        {
            //recuperación del valor introducido por el usuario
            numIntento=sc.nextInt();
            //incremento del número de intentos
            numIntentos++;
            //comparación con el número secreto
            //visualización de un mensaje si el número es diferente
            if(numIntento<secreto)
            {
                System.out.println("es más ");
            }
            else if(numIntento>secreto)
            {
                System.out.println("es menos ");
            }
            //Se continúa si el numIntento no es igual al número
            secreto
        } while (numIntento!=secreto);
        System.out.println("Bravo, lo ha encontrado en " + numIntentos
+ "
intento (s)");
    }
}
```

4. Ejercicio 4

Para hacer que evolucione la aplicación anterior vamos a agregar, simplemente, la inicialización de la variable **inicio** con la hora en curso antes de comenzar con el bucle. Al finalizar el bucle (y, por tanto, el juego), realizamos la misma acción con la variable **fin**. La duración de la partida es la diferencia entre ambas horas. A continuación el resultado se muestra en segundos.

```
import java.time.Duration;
import java.time.OffsetTime;
import java.util.Random;
import java.util.Scanner;

public class Ejercicio4 {
    public static void main(String[] args)
    {
        int numIntentos=0;
        int secreto;
        int numIntento;
        Random rd = new Random();

        secreto=rd.nextInt(1000);
        Scanner sc=new Scanner(System.in);
        //Guardar el momento concreto del inicio del programa
        OffsetTime inicio = OffsetTime.now();
        System.out.println("Introduzca un entero entre 0 y 1000");
        do
        {
            //recuperación del valor introducido por el usuario
            numIntento=sc.nextInt();
            //incremento del número de intentos
            numIntentos++;
            //comparación con el número secreto
            //mostrar un mensaje si el número es diferente
            if(numIntento<secreto)
            {
```

```
        System.out.println("es más");
    }
    else if(numIntento>secreto)
    {
        System.out.println("es menos ");
    }
    //Se continúa si el numIntento no es igual al
número secreto.
    } while (numIntento!=secreto);
    //Guardar el momento de la finalización del programa
OffsetTime fin = OffsetTime.now();
    //Cálculo de la duración entre el inicio y el fin
Duration duracion = Duration.between(inicio, fin);
    System.out.println("Bravo, lo ha encontrado en " + numIntentos
+
" intento(s)");
    //Visualización del tiempo en segundos
    System.out.println("ha usado " + duracion.getSeconds() +
" segundos");
    }
}
```

Conclusión

Ahora dispone de técnicas básicas para escribir algoritmos (declaración de variables, utilización de operadores disponibles y uso de estructuras de control). Para progresar, es necesario pasar horas descubriendo las funcionalidades que ofrece la API Java. Ahora necesita conocer con rapidez las bases de la programación orientada a objetos para profundizar en sus conocimientos. Este es el objetivo del próximo capítulo.

Introducción

Con Java, la noción de objeto es omnipresente y precisa un mínimo de aprendizaje. En primer lugar, nos centraremos en los principios de la programación orientada a objetos y en su vocabulario asociado, para pasar luego a ponerlo todo en práctica con Java.

En un lenguaje procedural clásico, el funcionamiento de una aplicación está regido por una sucesión de llamadas a diferentes procedimientos y funciones disponibles en el código. Estos procedimientos y funciones son los encargados de procesar los datos de la aplicación representados por las variables de esta última. No existe relación alguna entre los datos y el código que los manipula. Por el contrario, en un lenguaje orientado a objetos, lo que se intenta es agrupar el código. A esta agrupación se le llama clase. Por lo tanto, una aplicación desarrollada con un lenguaje orientado a objetos está formada por numerosas clases que representan los diferentes elementos procesados por la aplicación. Las clases describirán las características de cada uno de los elementos. El ensamblado de estos elementos va a permitir el funcionamiento de la aplicación.

Este principio se utiliza ampliamente en otras disciplinas, además de la informática. En la industria automovilística, por ejemplo, seguramente no existe ningún constructor que disponga de un plano que contenga todas y cada una de las piezas que constituyen un vehículo. Por el contrario, cada subconjunto de un vehículo puede estar representado por un plano específico (el chasis, la caja de cambios, el motor...). Cada subconjunto se va descomponiendo a su vez hasta la pieza elemental (un perno, un pistón, una rueda dentada...). Es el ensamblado de todos estos elementos lo que permite la fabricación de un vehículo.

De hecho, no es el ensamblado de los planos lo que permite la construcción del vehículo, sino el de las piezas fabricadas a partir de ellos. En una aplicación informática, será el ensamblado de los objetos creados a partir de las clases lo que permitirá el funcionamiento de la aplicación. Los dos términos, clase y objeto, suelen confundirse, aunque representan

nociones bien distintas. La clase describe la estructura de un elemento, mientras que el objeto representa un ejemplar creado a partir del modelo de dicha estructura. Tras su creación, un objeto es independiente de otros objetos contruidos a partir de la misma clase. Por ejemplo, después de la fabricación, la puerta de un coche podrá pintarse de un color distinto al de las otras puertas fabricadas con el mismo plano.

Las clases están constituidas por campos y métodos. Los campos representan las características de los objetos. Están identificadas por variables y es posible leer su contenido o asignarles un valor directamente. El robot que va a pintar una puerta cambiará el campo color de dicha puerta. Los métodos representan acciones que un objeto puede efectuar. Se ejecutan mediante la creación de procedimientos o de funciones en una clase.

Esto solo es una faceta de la programación orientada a objetos. Del mismo modo, existen otros tres conceptos fundamentales:

- La encapsulación
- La herencia
- El polimorfismo

La encapsulación consiste en esconder elementos no necesarios para la utilización de un objeto. Esta técnica permite garantizar que el objeto se utilice correctamente. Se trata de un principio muy utilizado en otras disciplinas, aparte de la informática. Regresando al ejemplo de la industria del automóvil, ¿conoce usted cómo funciona la caja de cambios de su vehículo?

Para cambiar de marcha, ¿modificará directamente la posición de los engranajes? Afortunadamente, no. Los constructores tienen previstas soluciones más prácticas para la manipulación de la caja de cambios.

Los elementos visibles de una clase desde su exterior forman la interfaz de clase. En el caso de nuestro coche, la palanca de cambios constituye la interfaz de la caja de cambios. Es a través de ella como se podrá actuar sin riesgo sobre los mecanismos internos de la caja de cambios.

La herencia permite la creación de una nueva clase a partir de otra ya existente. La clase que sirve de modelo se llama clase base, clase madre o superclase. La clase así creada hereda las características de su clase base. También es posible personalizarla añadiendo características adicionales. Las clases creadas a partir de una clase base se denominan

clases derivadas, clases hijas o subclases. Este principio se utiliza también en el mundo industrial. Seguramente, la caja de cambios de su coche incluye cinco marchas. A ciencia cierta, los ingenieros que concibieron esta pieza no partieron de cero. Retomaron el plano de la generación anterior (cuatro marchas) y le añadieron elementos. De la misma manera, los ingenieros que han reflexionado sobre la caja de cambios de seis marchas han tomado como base la versión precedente.

El polimorfismo es otra noción importante en la programación orientada a objetos. Gracias a él, es posible utilizar varias clases de manera intercambiable incluso si el funcionamiento interno de estas clases muestra diferencias. Si usted sabe cambiar la marcha en un coche Peugeot, también sabrá hacerlo en un Renault, y sin embargo los dos tipos de caja de cambios no fueron concebidos de la misma manera.

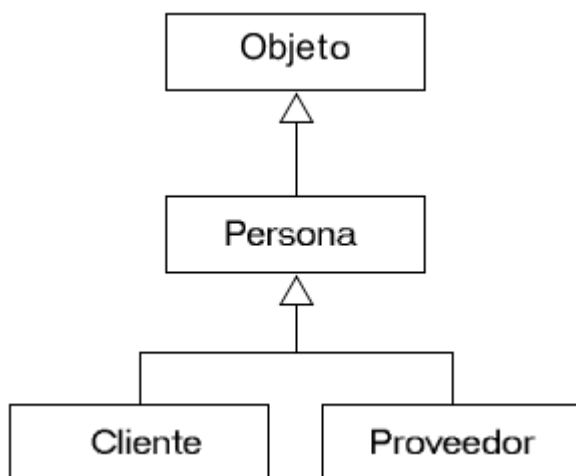
Asociados al polimorfismo existen otros dos conceptos: la sobrecarga y la sustitución de métodos. La sobrecarga se utiliza para diseñar en una clase métodos que compartan el mismo nombre, pero que tengan un número o tipos de parámetros distintos. Se utiliza la sustitución cuando, en una clase derivada, se quiere modificar el funcionamiento de uno de los métodos heredados. El número y el tipo de los parámetros se mantienen idénticos a los definidos en la clase base.

Veamos la puesta en práctica de algunos de los principios fundamentales de la programación orientada a objetos. Dado que el tema de este manual no versa sobre la mecánica automovilística, aplicaremos los conceptos de la orientación a objetos sobre el lenguaje Java.

Puesta en práctica con Java

1. Contexto

En el resto del capítulo, vamos a trabajar con la clase `Persona` y sus subclases. La representación UML (*Unified Modeling Language*) simplificada de esta herencia de clase es la siguiente.



UML es un lenguaje gráfico destinado a la representación de los conceptos de programación orientada a objetos. Para obtener más información sobre este lenguaje, puede consultar el manual UML 2.5 en la colección Recursos Informáticos de Ediciones ENI.

2. Creación de una clase

La creación de una clase consiste en su declaración y la de todos los elementos que la componen.

a. Declaración de la clase

La declaración de una clase se lleva a cabo utilizando la palabra clave `class` seguida del

nombre de la clase y de un bloque de código delimitado por los caracteres { y } (llaves). En este bloque de código se encuentran las declaraciones de variables, que serán los campos de la clase, y las funciones, que serán los métodos de la clase. Se pueden añadir varias palabras clave para modificar las características de la clase. Por lo tanto, la sintaxis de la declaración de una clase es la siguiente.

```
[modificadores] class NombreDeLaClase
    [extends NombreDeLaClaseBasica]
    [implements NombreDeInterfaz1,NombreDeInterfaz2,...]
{
    Código de la clase
}
```

Los signos [y] (corchetes) se utilizan para indicar qué elemento es opcional. No se deben utilizar en el código de declaración de una clase.

Los modificadores permiten determinar la visibilidad de la clase y la manera de utilizarla. A continuación presentamos la lista de los modificadores disponibles:

`public`: indica que la clase puede ser utilizada por cualquier otra clase. Sin este modificador, la clase solo podrán utilizarla clases que formen parte del mismo *package*. Para más información sobre los paquetes, vaya a la sección correspondiente, un poco más adelante en este capítulo.

`abstract`: indica que la clase es abstracta y no puede ser instanciada. Solo se la puede utilizar como clase básica en una relación de herencia. En este tipo de clase solo se suelen definir las declaraciones de métodos y habrá que escribir el contenido de los métodos en las clases derivadas.

`final`: la clase no puede utilizarse como clase de base en una relación de herencia. Dicho de otra manera, no es posible derivarla. Solo puede ser instanciada.

Al ser contradictorio el significado de las palabras clave `abstract` y `final`, su uso simultáneo está prohibido.

Para indicar que su clase recupera las características de otra clase por una relación de herencia, debe utilizar la palabra clave `extends` seguida del nombre de la clase base. También puede implementar en su clase una o varias interfaces utilizando la palabra

implements seguida de la lista de las interfaces implementadas. Se detallarán estas dos nociones más adelante en este capítulo.

El inicio de la declaración de la clase `Persona` es, por lo tanto, el siguiente:

```
public class Persona
{

}
```

Se debe introducir obligatoriamente este código en un archivo con el mismo nombre que la clase (`Persona`) y la extensión `.java`.

b. Creación de los campos

A continuación, nos vamos a interesar en el contenido de la clase. Debemos crear los diferentes campos (o variables miembro) de la clase. Para ello, basta con declarar variables en el interior del bloque de código de la clase e indicar la visibilidad de la variable, su tipo y su nombre.

```
[private | protected | public] tipoDeLaVariable nombreDeLaVariable;
```

Es posible definir variables de clase o constantes usando las palabras clave `static` y `final`. Vaya al capítulo que habla de las bases del lenguaje para más información.

La visibilidad de la variable responde a las reglas siguientes:

`private`: la variable solo es accesible en la clase donde está declarada.

`protected`: la variable es accesible en la clase donde está declarada, en las demás clases que forman parte del mismo paquete y en las clases que heredan de la clase donde esa misma variable está declarada.

`public`: la variable es accesible desde cualquier ubicación.

Si no se proporciona ninguna información relativa a la visibilidad, la variable es accesible desde la clase donde está declarada y desde las demás clases que forman parte del mismo paquete. Cuando definimos la visibilidad de una variable, debemos respetar en lo posible el principio de encapsulación y limitar al máximo la visibilidad de las variables. Lo ideal sería


```
{
```

```
}
```

Java cuenta con los siguientes modificadores de visibilidad:

`private`: indica que el método solo puede utilizarse en la clase donde está definido.

`protected`: indica que solo se puede utilizar el método en la clase donde está definido, en las subclases de esta clase y en las demás clases que forman parte del mismo paquete.

`public`: indica que se puede utilizar el método desde cualquier otra clase.

Si no se utiliza ninguna de estas palabras, entonces la visibilidad se limitará al paquete donde está definida la clase.

Hay disponibles modificadores adicionales.

`static`: indica que el método es un método de clase.

`abstract`: indica que el método es abstracto y que no contiene código. La clase donde está definido también debe ser abstracta.

`final`: indica que el método no puede ser sobrescrito en una subclase.

`native`: indica que el código del método se encuentra en un archivo externo escrito en otro lenguaje.

`synchronized`: indica que el método solo puede ser ejecutado por un único hilo a la vez.

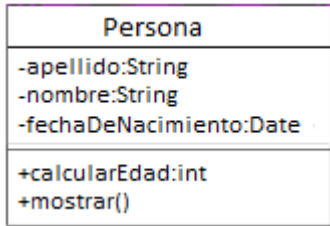
El tipo de retorno puede ser cualquier tipo de dato, tipo por valor o tipo por referencia. Si el método no tiene información para devolver, deberemos usar la palabra clave `void` en sustitución del tipo de retorno.

La lista de los parámetros es idéntica a una lista de declaración de variables. Hay que especificar el tipo y el nombre del parámetro. Si se esperan varios parámetros, hay que separar sus declaraciones con una coma. Incluso si no se espera ningún parámetro, los paréntesis son obligatorios.

La palabra clave `throws` indica la lista de excepciones que este método puede lanzar

durante su ejecución. La gestión de las excepciones se aborda más adelante en este capítulo.

La representación UML de la clase `Persona` con sus dos primeros métodos es la siguiente:



Ambos métodos son públicos (el + delante de los nombres de los métodos indica este carácter público). El primero (`CalcularEdad`) devuelve un número entero de tipo `int`, mientras que el segundo (`mostrar`) no devuelve nada.

La estructura de la clase `Persona` se convierte así en la siguiente:

```
public class Persona
{
    private String apellido;
    private String nombre;
    private LocalDate fechaDeNacimiento;

    public int calcularEdad ()
    {
        if(fechaDeNacimiento!=null)
        {
            return fechaDeNacimiento.until(LocalDate.now())
                                   .getYears();
        }
        return 0;
    }

    public void mostrar()
    {
        System.out.println("nombre: " + nombre);
        System.out.println("apellido: " + apellido);
        System.out.println("edad: " + calcularEdad ());
    }
}
```

```
}  
}
```

- ☐ Para comprender las instrucciones que calculan la edad, lo invito a consultar la javadoc del método `until ()` de la clase `LocalDate`.

d. Creación de las sobrecargas de métodos

En algunos lenguajes de programación, no es posible tener varias funciones con el mismo nombre. El lenguaje Java, como muchos otros lenguajes orientados a objeto, permite superar este problema creando funciones sobrecargadas. Una función sobrecargada tiene el mismo nombre que otra función de la clase, pero presenta una firma diferente. Se toma en cuenta la información siguiente para determinar la firma de una función:

- el nombre de la función,
- el número de parámetros esperados por la función,
- el tipo de los parámetros.

- ☐ El nombre de los argumentos y el tipo de retorno no se tienen en cuenta.

Para poder crear una función sobrecargada, hace falta que al menos uno de sus elementos cambie respecto a una función ya existente. Como el nombre de la función debe seguir siendo el mismo para poder hablar de sobrecarga, solo podemos actuar sobre el número de parámetros o su tipo. Por ejemplo, podemos añadir la función siguiente a la clase `Persona`:

```
public void mostrar(boolean espaniol)  
{  
    if (espaniol)  
    {  
        System.out.println("apellido: " + apellido);  
        System.out.println("nombre: " + nombre);  
        System.out.println("edad: " + calcularEdad());  
    }  
}
```

```

        else
        {
            System.out.println("name: " + apellido);
            System.out.println("first name: " + nombre);
            System.out.println("age: " + calcularEdad());
        }
    }
}

```

En efecto, posee una firma diferente de la primera función `mostrar` que hemos creado, pues espera un parámetro de tipo `boolean`. Si a continuación añadimos la función siguiente, el compilador rechaza la compilación del código.

```

    public void mostrar(boolean mayuscula)
    {
        if (mayuscula)
        {
            System.out.println("apellido: " +
apellido.toUpperCase());
            System.out.println("nombre: " + nombre.toUpperCase());
            System.out.println("edad: " + calcularEdad());
        }
        else
        {
            System.out.println("apellido: " +
apellido.toLowerCase());
            System.out.println("nombre: " + nombre.toLowerCase());
            System.out.println("edad: " + calcularEdad());
        }
    }
}

```

De hecho, determina que dos funciones tienen rigurosamente la misma firma, el mismo nombre, el mismo número de parámetros, incluso el mismo tipo de parámetros. Este ejemplo nos muestra también que el nombre de los parámetros no se tiene en cuenta para determinar la firma de una función.

Es posible diseñar una función que acepte un número variable de parámetros. La primera solución consiste en usar como parámetro un array y comprobar en el código de la función

el tamaño de dicho array para obtener los parámetros. Sin embargo, esta solución requiere la creación de un array en el momento de la llamada a la función. Por lo tanto, no es tan fácil como el uso de una lista de parámetros.

Con objeto de simplificar la llamada a este tipo de función, podemos utilizar la declaración siguiente para indicar que una función espera un número indefinido de parámetros. El parámetro de color es un parámetro **variádico**. Solo puede haber uno en un método y es necesariamente el último parámetro.

```
public void mostrar(String...colores)
{
    {
        if (colores==null)
        {
            System.out.println("ningún color");
            return;
        }
        switch (colores.length)
        {
            case 1:
                System.out.println("un color");
                break;
            case 2:
                System.out.println("dos colores");
                break;
            case 3:
                System.out.println("tres colores");
                break;
            default:
                System.out.println("más de tres colores");
        }
    }
}
```

En el interior del método, el parámetro `colores` se considera como un array (de cadenas de caracteres en nuestro caso).

Por el contrario, durante la llamada a la función, utilizamos una lista de cadenas de

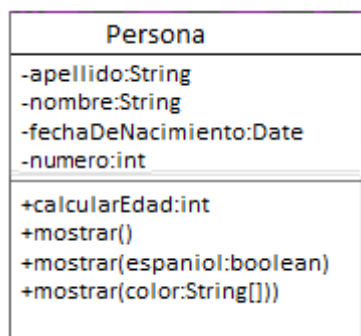
caracteres separadas por comas. Las sintaxis siguientes son perfectamente válidas para la llamada a esta función.

```
p.mostrar("rojo");  
p.mostrar("verde", "azul", "rojo");  
p.mostrar();
```

Solo hay una pequeña anomalía en el momento de la ejecución porque la última llamada de la función `mostrar` no ejecuta la función que acabamos de diseñar, sino la primera versión, que no espera parámetro. En efecto, el compilador no puede adivinar que se trata de la versión que espera un número variable de parámetros la que deseamos ejecutar sin pasarle parámetros. Por lo tanto, para indicarle nuestra intención debemos llamar a la función con la sintaxis siguiente.

```
p.mostrar(null);
```

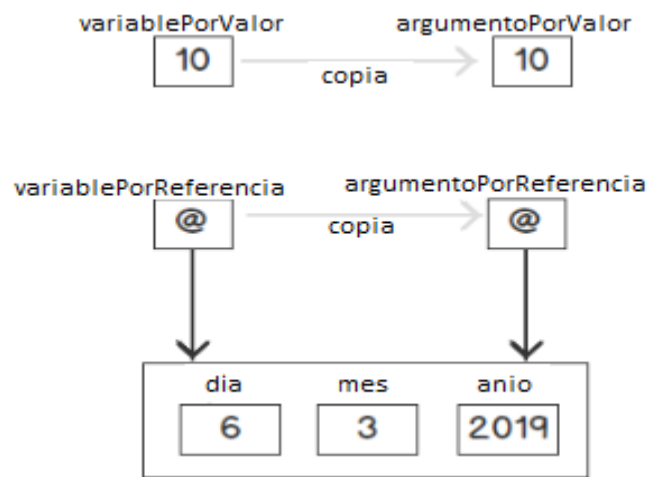
La representación UML de la clase `Persona`, como resultado de este trabajo, es el siguiente:



e. Paso de argumentos

En el momento de la llamada a una función, los parámetros se pasan por copia de valor de la variable, tanto para los tipos por valor del lenguaje (`int`, `float`, `boolean`...) como para los tipos por referencia. Sin embargo, se debe enfatizar un matiz importante.

El paso por copia de valor significa que hay una copia de valor en la variable del argumento correspondiente. El siguiente esquema muestra este principio para las variables de tipo por valor y las variables de tipo por referencia.



Por lo tanto, para una variable de tipo por valor es una copia del valor en sí. Si se cambia el valor en el método, el cambio no se reflejará en la variable que llama, ya que el cambio se realiza en una copia.

Por otro lado, para una variable de tipo por referencia, es una copia de la dirección que hace referencia al objeto en la memoria. El parámetro, en este caso, apunta a la misma área de memoria y, por lo tanto, al mismo objeto. Así, todas las modificaciones realizadas en este objeto en el método serán visibles desde los objetos que llaman.

f. Creación de accesores

La declaración de los atributos con una visibilidad privada es una buena práctica para respetar el principio de encapsulación. Sin embargo, esta solución es algo limitada, ya que solo el código de la clase donde están declarados puede acceder a ellos. Para resolver este problema, debemos definir métodos accesores. Son funciones ordinarias que simplemente tienen como objetivo hacer visibles los campos desde el exterior de la clase. Por convención, las funciones encargadas de asignar un valor a un campo se llaman `set` seguido del nombre del campo; las funciones encargadas de proporcionar el valor del campo se llaman `get` seguido del nombre del campo. Si el campo es de tipo `boolean`, el prefijo `get` se sustituye por el prefijo `is`.

Si un campo debe ser de solo lectura, el accesor `set` no debe estar disponible; si un campo debe ser de solo escritura, entonces se debe omitir la función `get`. Los accesores normalmente se llaman *getters* y *setters*.

Con esta técnica, podemos controlar el uso que se hace de los campos de una clase. Por lo tanto, podemos modificar la clase `Persona` al añadirle algunas reglas de gestión.

- El apellido se debe escribir en mayúsculas.
- El nombre se debe escribir en minúsculas.

```
public class Persona
{
    private String apellido;
    private String nombre;
    private LocalDate fechaDeNacimiento;

    public String getApellido()
    {
        return apellido;
    }

    public void setApellido(String a)
    {
        if(p!=null) {
            {
                apellido = a.toUpperCase(); }
            }
    }

    public String getNombre()
    {
        return nombre;
    }

    public void setNombre(String n)
    {
        if(n!=null)
        {
            nombre = n.toLowerCase(); }
        }
    }
}
```

A continuación, los campos de la clase son accesibles desde el exterior mediante estos métodos.

```
Persona p = new Persona();
p.setApellido("garcía");
p.setNombre("josé");
System.out.println(p.getApellido());
System.out.println(p.getNombre());
```

g. Creación de los constructores y destructores

Los constructores son métodos particulares de una clase por diferentes motivos. El constructor es un método que lleva siempre el mismo nombre que la propia clase. No devuelve ningún tipo, ni siquiera `void`. Se le llama durante la creación de una instancia de clase. Como en el caso de un método clásico, un constructor puede recibir parámetros. El constructor de una clase que no recibe parámetros es designado como el constructor por defecto de la clase. El papel principal del constructor es la inicialización de los campos de una instancia de clase. Como para los demás métodos de una clase, también es posible sobrecargar los constructores. La creación de un constructor por defecto no es obligatoria, ya que el compilador proporciona uno automáticamente. Este constructor efectúa simplemente una llamada al constructor de la superclase que, por supuesto, debe existir. Este es el principio de encadenamiento de constructores que se puede encontrar en otros idiomas. La idea es que el constructor de una clase se use para inicializar las variables miembro de esta clase, pero en ningún caso las variables miembro de las clases principales. Para esto, el constructor de una clase puede llamar a uno de los constructores de su clase padre. Para obtener más información, consulte la sección Herencia, más adelante en este capítulo.

Si su clase contiene un constructor sobrecargado, es interesante añadir explícitamente el constructor por defecto. En efecto, un conjunto de frameworks se basa en la presencia de un constructor como este. Una buena costumbre consiste en crear siempre un constructor por defecto en cada una de sus clases. Añadimos constructores a la clase `Persona`.

```
public Persona()
{
    setApellido("");
    setNombre("");
    fechaDeNacimiento=null;
```

```
}  
public Persona(String a,String n,LocalDate f)  
{  
    setApellido(a);  
    setNombre(n);  
    fechaDeNacimiento=f;  
}
```

Los destructores son otros métodos particulares de una clase. Se invocan implícitamente durante la destrucción de una instancia de clase. Se impone la firma del destructor. Este método debe ser `protected`, no devuelve ningún valor, se llama obligatoriamente `finalize`, no recibe ningún parámetro y es susceptible de producir una excepción de tipo `Throwable`.

Con motivo de esta firma impuesta, solo puede haber un único destructor para una clase y, por lo tanto, la sobrecarga no es posible para los destructores. De este modo, la declaración de un destructor es la siguiente:

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

Veremos con detalle, en la sección Destrucción de una instancia, las circunstancias en las cuales se llama al destructor.

h. Creación de campos y métodos estáticos

Los miembros estáticos son campos o métodos que son accesibles por la propia clase o por cualquier instancia de la clase. En algunos lenguajes, también se habla de miembros compartidos. Son muy útiles cuando es necesario gestionar, en una clase, información que no es específica a una instancia de la clase, sino a la propia clase. A diferencia de los miembros de instancia, para los cuales existe un ejemplar por instancia de la clase, de los miembros estáticos existe un único ejemplar. La modificación del valor de un miembro de instancia solo modifica el valor para esta instancia de clase, mientras que la modificación del valor de un miembro estático modifica el valor para todas las instancias de la clase. Los miembros estáticos son asimilables a variables globales en una aplicación. Se utilizan en el


```
private static int numInstancias;

public String getApellido()
{
    return apellido;
}

public void setApellido(String a)
{
    apellido = a.toUpperCase();
}

public String getNombre()
{
    return nombre;
}

public void setNombre(String n)
{
    nombre = n.toLowerCase();
}

// método de instancia que permite obtener el número de una Persona
public int getNumero()
{
    return numero;
}

// método estático que permite obtener el número de instancias
// creadas
public static int getNumInstancias()
{
    return numInstancias;
}

public Persona()
{
    setApellido("");
    setNombre("");
    fechaDeNacimiento=null;
    // creación de una nueva Persona y por lo tanto
```

```
incrementación del contador
    numInstancias++;
    // asignación a la nueva Persona de su número
    numero=numInstancias;
}
}
```

i. Utilización de las anotaciones

Se utilizan las anotaciones para añadir información adicional a un elemento. Esta información no tiene ningún efecto en el código, pero puede utilizarse en el compilador, en la máquina virtual que se encargará de la ejecución de la aplicación o en ciertas herramientas de desarrollo (como JUnit) o algunos frameworks (Java EE, ORM, Spring). Se pueden aplicar a una clase, a un campo o a un método. Se deben especificar antes del elemento al cual se refieren. Una anotación viene precedida por el símbolo `@` y está seguida del nombre de la anotación.

Por su parte, el compilador reconoce tres tipos de anotaciones que van a permitir la modificación de su comportamiento en tiempo de compilación.

La anotación `@Deprecated` se utiliza para indicar que un método ya no debe utilizarse. Es el caso, por ejemplo, en el que decidimos hacer evolucionar un método y deseamos que no se use más la versión anterior. Esta anotación no cambia el resultado de la compilación, pero añade información adicional al código compilado. Si otra clase intenta usar este método, se genera un aviso en el momento de la compilación del código de esta clase.

Si añadimos esta anotación al método `mostrar` de la clase `Persona`, el código que utiliza la clase `Persona` no debe llamar más a este método, pues generará un aviso en tiempo de compilación.

```
public class Persona
{
    ...

    @Deprecated
    public void mostrar()
    {
```



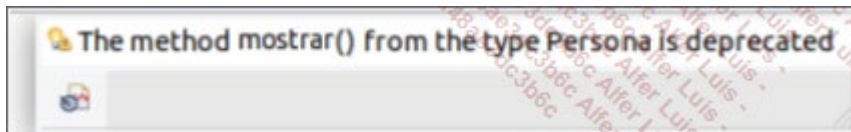
```

        System.out.println("apellido: " + apellido);
        System.out.println("nombre: " + nombre);
        System.out.println("edad: " + calcularEdad());
    }

    ...
}

```

La compilación de una clase que contiene una llamada al método `mostrar` de la clase `Persona` genera un aviso en la línea que contiene esta llamada.



El método se tacha y se muestra un mensaje explícito.

La anotación `@Override` se utiliza para indicar que un método sustituye a otro heredado. Esta anotación no es obligatoria, pero exige al compilador que verifique que la sustitución se realizó correctamente (firma idéntica de los métodos en la clase básica y en la clase actual). Si no es el caso, se activará un error de compilación. El ejemplo siguiente sustituye el método `calcularEdad` en una clase que hereda de la clase `Persona` (más adelante se detallará la puesta en práctica de la herencia).

```

public class Client extends Persona
{
    @Override
    public int calcularEdad()
    {
        ...
        ...
    }
}

```

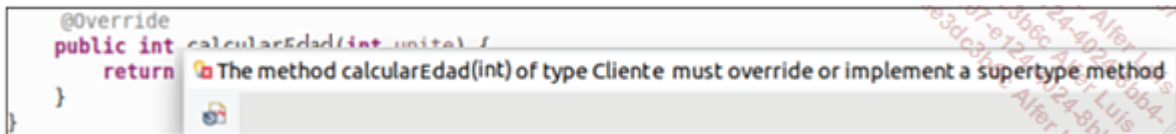
Esta clase se compiló sin problema, ya que el método `calcularEdad` tiene efectivamente la misma firma que la clase `Persona`. Por el contrario, si intentamos compilar el código siguiente:

```

public class Client extends Persona
{
    @Override
    public int calcularEdad(int unidad)
    {
        ...
        ...
    }
}

```

Obtenemos la respuesta siguiente por parte del compilador:



Por supuesto, tiene razón (¡hay que reconocer que siempre lleva razón!); le hemos anunciado nuestra intención de sustituir el método `calcularEdad` y, en realidad, hemos efectuado una sobrecarga, ya que no existe un método en la clase `Persona` con esta firma. Si quitamos la palabra clave `@Override`, el código compila, pero en este caso se trata de una sobrecarga.

La anotación `@SuppressWarnings("...")` indica al compilador que no genere ciertas categorías de avisos. Si por ejemplo deseamos usar discretamente un método marcado como `@Deprecated`, debemos utilizar la anotación siguiente en la función donde se encuentra la llamada a este método.

```

@SuppressWarnings("deprecation")
public static void main(String[] args)
{
    Persona p;
    p=new Persona();
    p.mostrar();
}

```

Este código se compilará correctamente sin ningún aviso, hasta que el método en cuestión

desaparezca completamente de la clase correspondiente. En efecto, hemos de tener en cuenta que la meta de la anotación `@Deprecated` es desaconsejar el uso de un método, con la posible intención de hacerlo desaparecer en una versión posterior de la clase.

3. Utilización de una clase

La utilización de una clase en una aplicación pasa por tres etapas:

- la declaración de una variable que permite acceder al objeto;
- la creación del objeto;
- la inicialización de una instancia.

a. Creación de una instancia

Las variables objeto son variables de tipo referencia. Se distinguen de las variables clásicas por el hecho de que la variable no contiene directamente los datos, sino una referencia de la ubicación en memoria donde se encuentra la información. Al igual que en el caso de las variables de tipos primitivos, las instancias deben declararse antes de su utilización. La declaración se hace de manera idéntica a la de una variable clásica (`int` u otra).

```
Persona p;
```

Después de esta etapa, la variable existe, pero no referencia una ubicación válida. Contiene el valor `null` si es una variable miembro de una clase, o nada si es una variable local a un método. La segunda etapa consiste en crear la instancia de la clase. Se utiliza la palabra clave `new` a este efecto. Recibe como parámetro el nombre de la clase cuya instancia debe crear. El operador `new` realiza una petición para obtener la memoria necesaria para almacenar la instancia de la clase y, a continuación, inicializa la variable con esta dirección de memoria. A continuación, se invoca al constructor de la clase para inicializar la nueva instancia creada.

```
p=new Persona();
```

Es posible combinar las dos operaciones en una única línea.

```
Persona p=new Persona();
```

En este caso, se invoca al constructor por defecto. Para utilizar otro constructor, debemos especificar una lista de parámetros y, según el número y el tipo de los parámetros, el operador `new` llama al constructor correspondiente.

```
Persona pe = new Persona("García",  
                           "José",  
                           LocalDate.of(1975,12,13));
```

b. Inicialización de una instancia

Es posible inicializar los campos de una instancia de clase de varias maneras. La primera consiste en inicializar las variables que forman los campos de la clase en el momento de su declaración.

```
public class Persona  
{  
    private String apellido="nuevoApellido";  
    private String nombre="nuevoNombre";  
    private LocalDate fechaDeNacimiento=LocalDate.now ();  
    ...  
    ...  
}
```

Esta solución, aunque muy sencilla, está bastante limitada, ya que no es posible utilizar estructuras de control tales como un bucle exterior al bloque del código. La solución que nos viene rápidamente a la mente consiste en ubicar el código de inicialización en el interior de un constructor. En efecto, es una muy buena idea y es incluso el objetivo principal del constructor: inicializar las variables de instancia. Como contrapartida, se plantea un problema con los campos estáticos, ya que para ellos no es necesario disponer de una instancia de clase para poder utilizarlos. Por lo tanto, si ubicamos el código encargado de inicializarlos en un constructor, nada nos garantiza que se le haya invocado al menos una vez antes de la utilización de los campos estáticos.

Para resolver este problema, Java proporciona los bloques de inicialización estáticos. Son simples bloques de código precedidos por la palabra clave `static` y delimitados por los

caracteres { y }. Pueden aparecer en cualquier parte del código de la clase y la máquina virtual los ejecuta en el orden en el que aparecen en el código cuando se carga la clase. Podemos, por ejemplo, usar el código siguiente para inicializar un campo estático con un valor aleatorio inferior a 1000.

```
public class Persona
{
    private String apellido="nuevoApellido";
    private String nombre="nuevoNombre";
    private LocalDate fechaDeNacimiento=LocalDate.of(1963,11,29);
    private int numero=0;
    private static int numInstancia;

    static
    {
        Random rd = new Random();
        numInstancias = rd.nextInt(1000);
    }

    ...
    ...
}
```

Es posible obtener el mismo resultado creando una función privada estática y llamándola para inicializar la variable.

```
public class Persona
{
    private String apellido="nuevoApellido";
    private String nombre="nuevoNombre";
    private LocalDate fechaDeNacimiento=LocalDate(1963,11,29);
    private int numero=0;
    private static int numInstancia=inicContador();

    private static int inicContador()
    {
        int cpt=0;
        Random rd = new Random();
```

```
        cpt=numInstancias = rd.nextInt(1000);  
        return cpt;  
    }  
    ...  
    ...  
}
```

Esta solución presenta la ventaja de que permite utilizar la función en otra parte del código de la clase.

Se puede aplicar el mismo principio para la inicialización de los campos de instancia. En este caso, el bloque de código encargado de la inicialización no debe ir precedido de la palabra clave `static`. Este bloque de código se vuelve a copiar implícitamente al principio de cada constructor de la clase durante la compilación. También es posible inicializar un campo de instancia mediante la llamada a una función. En este caso, el desarrollador se encontrará con una sutil restricción: el método no puede sobrescribirse en ninguna subclase. Para ello, hay que declararlo con la palabra clave `final`.

c. Destrucción de una instancia

A veces la gestión de la memoria resulta ser un verdadero rompecabezas para algunos lenguajes de programación. El desarrollador es responsable de la creación de las instancias de clases, pero también de su destrucción con el fin de liberar memoria. Por fortuna, Java se encarga totalmente de la gestión y nos evita esta tediosa tarea.

Es capaz de determinar cuándo no se utiliza un objeto en la aplicación y, a continuación, lo elimina de la memoria. Este mecanismo se llama **Garbage Collector** (recolector de basura). Java considera que se puede suprimir un objeto cuando la aplicación no puede acceder más a él. Esta situación se produce, por ejemplo, al salir de una función, cuando se utiliza una variable local para referenciar al objeto. También puede estar provocada por la asignación del valor `null` a una variable. Para borrar realmente un objeto de la memoria, es necesario que hayan desaparecido todos los medios de acceder a él desde la aplicación. No hay que olvidar que, si un objeto está almacenado en una colección o en un array, estos conservan una referencia hacia el objeto.

Veamos con un poco más de detalle el funcionamiento del Garbage Collector. Existen varios algoritmos para poner en práctica el mecanismo de gestión de memoria. Los

diseñadores de la máquina virtual Java implementan estos mecanismos. Vamos a echarles un vistazo por pura curiosidad:

Mark y Sweep

Con este mecanismo, el Garbage Collector trabaja en dos etapas. Empieza con una exploración de la memoria desde la raíz de la aplicación, el método `main`, y recorre así todos los objetos accesibles desde esta raíz. Cada objeto accesible se marca durante esta exploración (Mark). A continuación, realiza un segundo recorrido durante el cual suprime todos los objetos no marcados, y por lo tanto inaccesibles, y quita las marcas de los objetos que quedan y que puso durante el primer recorrido.

Esta solución rudimentaria presenta ciertos inconvenientes:

- Durante la primera etapa se interrumpe la ejecución de la aplicación.
- Su duración es proporcional a la cantidad de memoria usada por la aplicación.
- Tras varios recorridos, hay riesgo de que la memoria quede fragmentada.

Stop y Copy

Esta solución divide en dos partes idénticas el espacio de memoria disponible para la aplicación en ejecución. En cuanto el Garbage Collector entra en acción, efectúa, como para la solución anterior, una exploración de la memoria desde la raíz de la aplicación. Cuando encuentra un objeto accesible, realiza una copia hacia la segunda zona de memoria y modifica las variables para que referencien esta nueva ubicación. Al final de la exploración, todos los objetos accesibles se han copiado en la segunda zona de memoria. A continuación, es posible borrar totalmente el contenido de la primera zona. Más tarde, se puede repetir el mismo mecanismo con la zona de memoria que se acaba de liberar. Esta solución presenta la ventaja de eliminar la fragmentación de la memoria, ya que los objetos se copian unos tras otros. Por el contrario, un inconveniente importante reside en el desplazamiento frecuente de objetos que tienen un largo ciclo de vida en la aplicación.

Una solución intermedia consiste en repartir los objetos en la memoria según su esperanza de vida o su edad. En ese caso, la mitad de la memoria disponible a veces resulta dividida en tres zonas:

- Una zona para los objetos con un tiempo de vida muy largo y que apenas tienen riesgo de desaparecer durante el funcionamiento de la aplicación.
- Una zona para los objetos creados recientemente.

- Una zona para los objetos más antiguos.

Cuando el Garbage Collector interviene, trata primero la zona reservada a los objetos recientes. Si después de este primer barrido la aplicación dispone de la memoria suficiente, el Garbage Collector detiene su tratamiento; hay que señalar que, durante este primer barrido, puede transferir objetos que lleven tiempo existiendo a la zona reservada a los objetos antiguos. Si la memoria disponible no es suficiente, realiza de nuevo el tratamiento con la zona reservada a los objetos más antiguos.

La eficacia de esta solución reside en una idea cruel: los objetos Java no disponen de una gran esperanza de vida. De este modo, resulta muy fácil encontrar objetos para eliminar entre los creados recientemente.

Entre estas dos soluciones, es difícil decir cuál utiliza más a menudo la máquina virtual, ya que la implementación del Garbage Collector se deja a la libre elección del diseñador de dicha máquina.

Antes de eliminar una instancia de la memoria, el Garbage Collector invoca al destructor de esta instancia.

El último punto por aclarar en relación con el Garbage Collector se refiere a su activación. En realidad, le corresponde a la máquina virtual Java vigilar los recursos de memoria disponibles y provocar la entrada en acción del Garbage Collector cuando los recursos han alcanzado un umbral límite (alrededor del 85 %). Sin embargo, es posible forzar la activación del Garbage Collector invocando el método `gc()` de la clase `System`. Este uso debe resultar excepcional, ya que un uso demasiado frecuente penaliza el rendimiento de la aplicación. Se puede usar justo antes de que la aplicación utilice una cantidad de memoria importante, como, por ejemplo, la creación de un array voluminoso.

El código siguiente permite destacar la acción del Garbage Collector.

```
public class Persona
{
    private String apellido;
    private String nombre;
    private LocalDate fechaDeNacimiento;
    private int numero=0;
    private static int numInstancia;
```


...

```
@Override
protected void finalize() throws Throwable
{
    System.out.print("\u2020");
    super.finalize();
}
}
```

El código unicode \u2020 se corresponde con el siguiente signo:

+

```
public class GestionMemoria
{

    public static void main(String[] args)
                                throws InterruptedException
    {
        double total;
        double resto;
        double porcentaje;

        for (int j=0;j<1000;j++)
        {
            creacionArray();
            total=Runtime.getRuntime().totalMemory();
            resto=Runtime.getRuntime().freeMemory();
            porcentaje=100-(resto/total)*100;
            System.out.println("creacion del " + j + "º
                                array memoria llena a: " + porcentaje + "%" );
            // una pequeña pausa para poder leer los mensajes
            Thread.sleep(1000);
        }
    }
}
```

```

public static void creacionArray()
{
    // creación de un array de 1000 Personas en una variable local
    // al final de esta función los elementos del array ya no están
    // accesibles y se pueden eliminar de la memoria
    Persona[] array;
    array=new Persona[1000];
    for (int i=0;i<1000;i++)
    {
        array[i]=new Persona("García",
                               "José",
                               LocalDate.of(1956,12,13));
    }
}
}

```

El código anterior consume espacio de memoria con la creación de 1000 tablas de 1000 personas. Cuando se agota el espacio en memoria, el recolector de basura lo limpia eliminando todos los objetos inaccesibles.

4. Herencia

a. Creación de una clase hija

La herencia es una funcionalidad potente de un lenguaje orientado a objetos, pero a veces no se utiliza como se debe. Se pueden contemplar dos categorías de relaciones entre dos clases. Podemos tener la relación «es un tipo de» y la relación «se trata de».

La relación de herencia debe utilizarse cuando es posible aplicar la relación «es un tipo de» entre dos clases. Veamos un ejemplo con tres clases: `Persona`, `Cliente`, `Pedido`.

Probamos la relación «es un tipo de» para cada una de las clases.

- Un pedido es un tipo de cliente.
- Un pedido es un tipo de persona.
- Un cliente es un tipo de pedido.

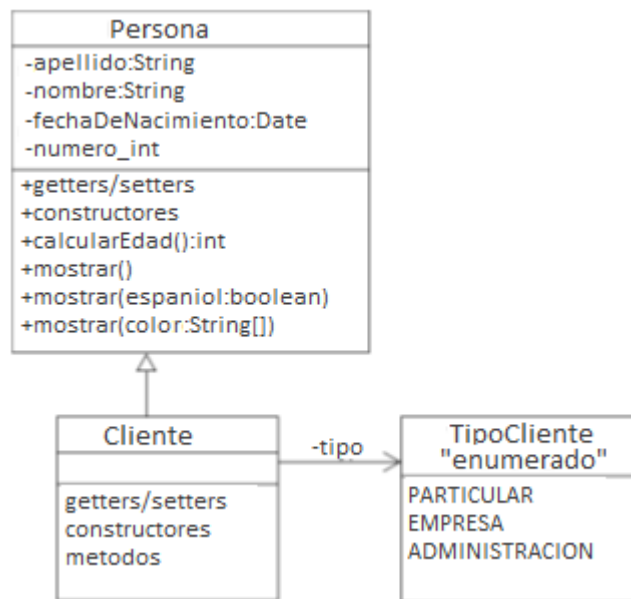
- Un cliente es un tipo de persona.
- Una persona es un tipo de cliente.
- Una persona es un tipo de pedido.

De entre todos estos intentos, solo uno nos resulta lógico: un cliente es un tipo de persona. Por lo tanto, podemos considerar una relación de herencia entre estas dos clases.

La puesta en práctica es muy sencilla a nivel del código, ya que en la declaración de la clase basta con especificar la palabra clave `extends` seguida del nombre de la clase que se desea heredar. Al no aceptar Java la herencia múltiple, solo podemos especificar un único nombre de clase básica. En el interior de esta nueva clase, podemos:

- Utilizar los campos heredados de la clase básica (con la condición, por supuesto, de que su visibilidad lo permita, es decir, que no deben ser privados).
- Añadir nuevos campos.
- Enmascarar un campo heredado declarándolo con el mismo nombre que el usado en la clase base. Se debe utilizar esta técnica con moderación.
- Usar un método heredado siempre que su visibilidad lo permita.
- Sustituir un método heredado al declararlo idéntico (misma firma).
- Sobrecargar un método heredado creándolo con una firma diferente.
- Añadir un nuevo método.
- Añadir uno o varios constructores.

A continuación, presentamos el ejemplo de la creación de la clase `Cliente`; la siguiente podría ser una representación UML simplificada:



Esta clase hereda de la clase `Persona`, a la cual se añade el campo `tipo` y los métodos de acceso correspondientes.

```

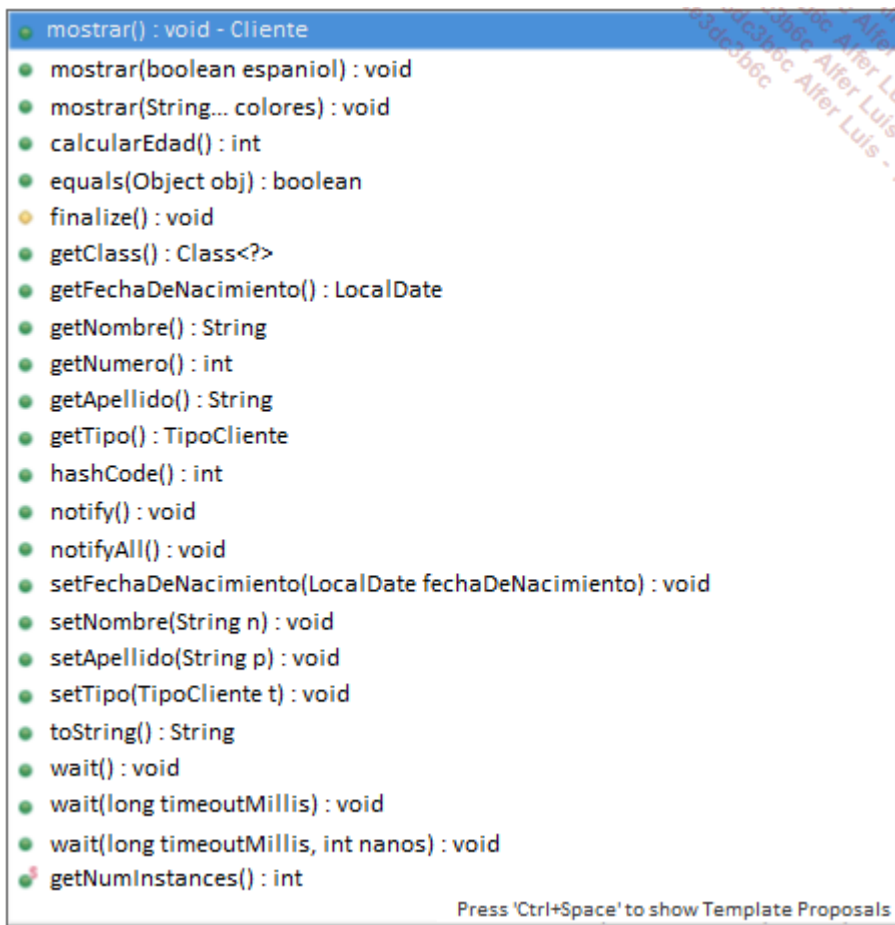
public class Client extends Persona
{
    private TipoCliente tipo;
    public TipoCliente getTipo()
    {
        return tipo;
    }
    public void setTipo(TipoCliente t)
    {
        tipo = t;
    }
}
  
```

Ya se puede utilizar la clase, la cual presenta todas las funcionalidades definidas en la clase `Cliente` más las heredadas de la clase `Persona`.

```

Cliente c;
c=new Cliente();
c.setApellido("ENI");
c.setNombre("");
c.setFechaDeNacimiento(LocalDate.of(1981,05,15));
c.setType(TipoCliente.EMPRESA);
c.mostrar();
  
```

En el autocompletado, Eclipse ofrece los siguientes métodos sobre una variable de tipo `Cliente`:



b. this y super

A estas alturas parece legítimo querer modificar el funcionamiento de algunos métodos heredados para adaptarlos a la clase `Cliente`. Por ejemplo, se puede sustituir el método `mostrar` para tener en cuenta el nuevo campo disponible en la clase.

```
@Override
public void mostrar()
{
    System.out.println("apellido: " + getApellido());
    System.out.println("nombre: " + getNombre());
    System.out.println("edad: " + calcularEdad());
    switch (tipo)
    {
        case PARTICULAR:
            System.out.println("tipo de cliente: Particular");
    }
}
```

```

        break;
    case EMPRESA:
        System.out.println("tipo de cliente: Empresa");
        break;
    case ADMINISTRACION:
        System.out.println("tipo de cliente: Administración");
        break;
    default:
        System.out.println("tipo de cliente: Desconocido");
        break;
    }
}

```

Este código funciona muy bien, pero no respeta uno de los principios de la programación orientada a objetos, a saber; reutilizar al máximo lo que ya existe. En nuestro caso, ya tenemos una sección de código encargada de la visualización del apellido, del nombre y de la edad de una persona (el método `mostrar` de la clase `Persona`). ¿Por qué no volver a utilizarla en el método `mostrar` de la clase `Cliente`, ya que heredamos de ella?

El método se convierte en lo siguiente:

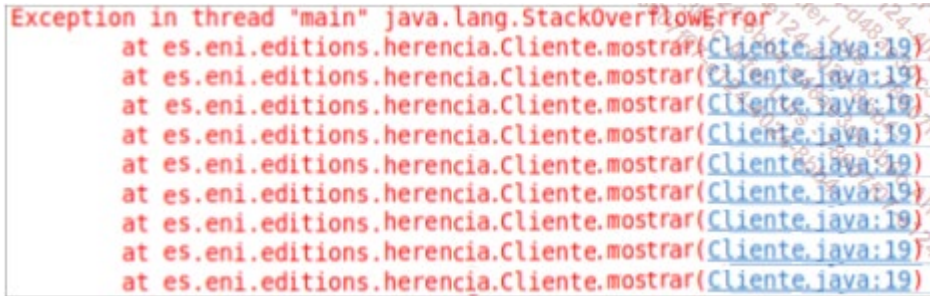
```

@Override
public void mostrar()
{
    mostrar();
    switch (tipo)
    {
    case PARTICULAR:
        System.out.println("tipo de cliente: Particular");
        break;
    case EMPRESA:
        System.out.println("tipo de cliente: Empresa");
        break;
    case ADMINISTRACION:
        System.out.println("tipo de cliente: Administración");
        break;
    default:
        System.out.println("tipo de cliente: Desconocido");
    }
}

```

```
        break;
    }
}
```

¡Desafortunadamente, el resultado no es el esperado!



```
Exception in thread "main" java.lang.StackOverflowError
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
  at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:19)
```

¿Qué ha ocurrido durante la ejecución?

Tras la llamada al método `mostrar` de la clase `Cliente`, la primera línea de código se limita a invocar el método `mostrar` de la clase base (`Persona`). En realidad, la máquina virtual Java ejecuta el primer método `mostrar` que encuentra. Para ello, empieza la búsqueda en la clase desde la cual se ha creado el objeto, en nuestro caso la clase `Cliente`. Así, llama en bucle al método `mostrar` de la clase `Cliente` y, de ahí, el error de desbordamiento de pila que obtenemos (`StackOverflowError`). Para evitar este tipo de problemas, es preciso indicar que el método `mostrar` que debe invocar se encuentra en la clase base, utilizando para ello la palabra clave `super` que apunta al método `mostrar` invocado.

```
@Override
public void mostrar()
{
    super.mostrar();
    switch (tipo)
    {
        case PARTICULAR:
            System.out.println("tipo de cliente: Particular");
            break;
        case EMPRESA:
            System.out.println("tipo de cliente: Empresa");
            break;
        case ADMINISTRACION:
```

```

        System.out.println("tipo de cliente: Administración");
        break;
    }
}

```

Tras realizar esta modificación, todo vuelve al orden y nuestro código muestra:

```

<terminated> TestCliente [Java Application]
nombre: ENI
apellido:
edad: 37
tipo de cliente: Empresa

```

La misma palabra clave se puede utilizar para invocar el método constructor de la clase base. La llamada al constructor de la clase madre debe ser la primera línea de un constructor. Si no se realiza una llamada explícita, se llama implícitamente al constructor predeterminado de la clase principal. Este es el principio de encadenamiento de constructores.

Si no hay un constructor predeterminado en la clase principal, aparece un error de compilación.

Por lo tanto, podemos crear un constructor para la clase `Cliente` que reutilice el constructor de la clase `Persona`.

- ☐ Tras la creación de un constructor sobrecargado, el constructor por defecto, generado automáticamente por el compilador, ya no se encuentra disponible. Por ello, resulta ser una buena costumbre crear siempre un constructor por defecto.

```

public Cliente() {
    super();
}

public Client(String nombre,
              String apellido,
              LocalDate fechaDeNacimiento,
              TipoCliente type) {
    super(nombre, apellido, fechaDeNacimiento);
}

```



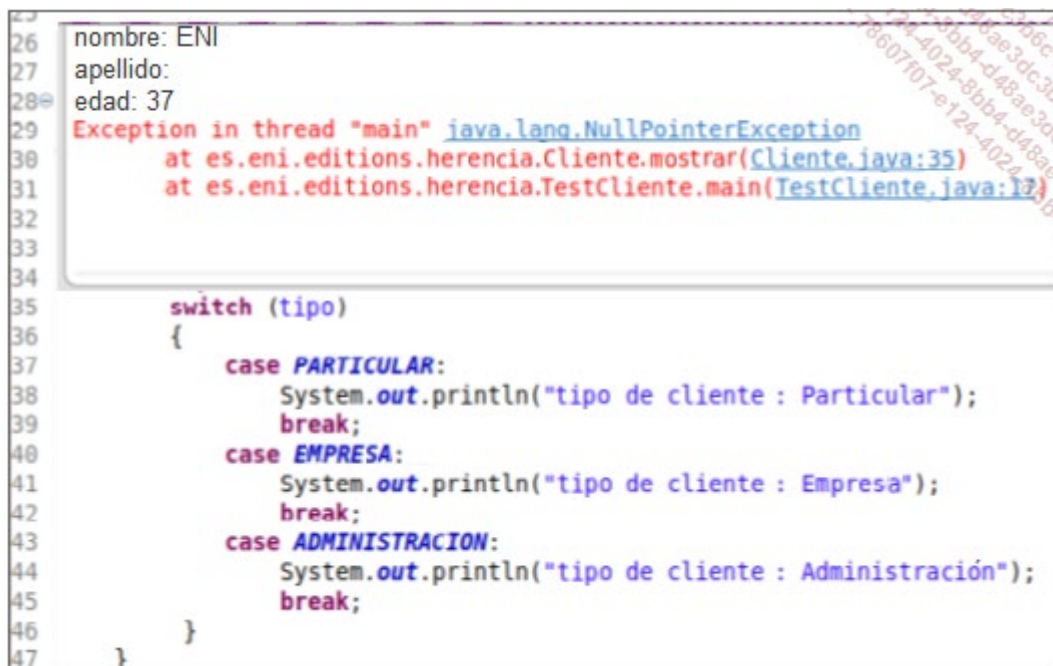
```
    tipo = type;
}
```

Comprobamos que el nuevo constructor funciona con la siguiente prueba.

```
Cliente c2 = new Cliente("ENI",
                        "",
                        LocalDate.of(1981,05,15),
                        TipoCliente.EMPRESA);

c2.mostrar ();
```

Durante la ejecución, el resultado es el siguiente:



The screenshot shows a Java IDE with a console window at the top and a code editor below. The console window displays the following output:

```
26 nombre: ENI
27 apellido:
28 edad: 37
29 Exception in thread "main" java.lang.NullPointerException
30   at es.eni.editions.herencia.Cliente.mostrar(Cliente.java:35)
31   at es.eni.editions.herencia.TestCliente.main(TestCliente.java:17)
```

The code editor shows a switch statement for the 'tipo' variable:

```
35 switch (tipo)
36 {
37     case PARTICULAR:
38         System.out.println("tipo de cliente : Particular");
39         break;
40     case EMPRESA:
41         System.out.println("tipo de cliente : Empresa");
42         break;
43     case ADMINISTRACION:
44         System.out.println("tipo de cliente : Administración");
45         break;
46 }
47 }
```

Se ha tenido en cuenta la información, salvo el tipo de cliente, que no se ha inicializado. Veamos más de cerca el código del constructor. Descubrimos que un parámetro del constructor tiene el mismo nombre que un campo de la clase. Cuando escribimos la línea `type=type`, el compilador considera que deseamos asignar al parámetro `type` el valor contenido en el parámetro `type`. No se trata de nada ilegal, pero no es en absoluto lo que deseamos hacer. Debemos indicar que la asignación se debe efectuar al campo de la clase. Para ello, debemos prefijar su nombre con la palabra clave `this`.

El constructor se convierte entonces en:

```
public Cliente(String apellido,
               String nombre,
```

```
        LocalDate fechaDeNacimiento,  
        TipoCliente type) {  
    super(apellido,nombre,fechaDeNacimiento);  
    this.type=type;  
}
```

c. Clases abstractas

Las clases abstractas son clases que solo sirven como clase base en una relación de herencia. Resulta imposible crear una instancia de una clase abstracta. Son esencialmente un modelo para la creación de clases, todas las cuales deben tener unas mínimas características idénticas. Pueden contener campos, propiedades, constructores y métodos como una clase ordinaria. Para que una clase sea abstracta, debemos utilizar la palabra clave `abstract` en el momento de su declaración.

Esta técnica facilita la evolución de la aplicación, ya que, si una nueva funcionalidad debe estar disponible en las clases derivadas, basta con añadir esta funcionalidad a la clase base. También es posible no proporcionar implementación para algunos métodos de una clase abstracta y así dejar al desarrollador de la clase la responsabilidad de crear la implementación en la clase derivada. También se deben declarar estos métodos con la palabra clave `abstract`. Por lo tanto, en este caso, no hay bloque de código correspondiente a este método y se debe terminar su declaración con un punto y coma.

Una clase abstracta puede contar con métodos no abstractos, pero si contiene un método abstracto obligatoriamente debe ser abstracta para evitar encontrarse con objetos para los que no hay disponible ninguna implementación del método abstracto. La clase que hereda de la clase abstracta deberá implementar los métodos declarados como abstractos en su clase base o ser a su vez abstracta. Si implementa un método abstracto, no puede reducir la visibilidad declarada en la clase abstracta. A continuación, veamos un ejemplo de clase abstracta:

```
public abstract class SerVivo  
{  
    private double tamaño;  
    private double peso;  
    public double getTamaño()  
    {
```

```

        return tamaño;
    }
    public void setTamaño(double tamaño)
    {
        this.tamaño = tamaño;
    }
    public double getPeso()
    {
        return peso;
    }
    public void setPeso(double peso)
    {
        this.peso = peso;
    }
    // se deberá implementar este método en las clases derivadas
    protected abstract void desplazarse();
}

```

d. Clases finales

Las clases finales son clases ordinarias que pueden ser instanciadas, pero que no pueden utilizarse como clase base en una relación de herencia. Deben declararse con la palabra clave `final`. Es el caso de varias clases del lenguaje Java, como por ejemplo la clase `String`. A continuación presentamos su declaración extraída de la documentación de Java.

```

public final class String
extends Object

```

También se puede declarar un método con la palabra clave `final` para impedir su redefinición en una subclase.

El significado de las palabras clave `abstract` y `final` es algo contradictorio, y está prohibido utilizar ambas palabras clave de manera simultánea. El compilador detecta dicha situación y genera un error.

```

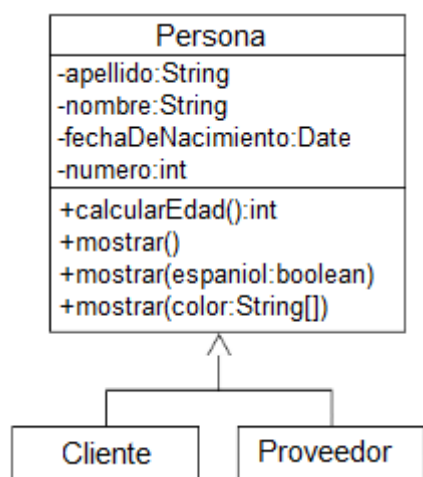
❌ The class SerVivo can be either abstract or final, not both

```

e. Conversión de tipo

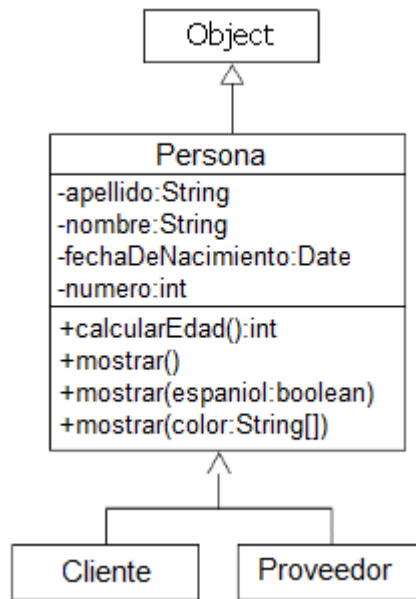
Para entender correctamente las operaciones de conversiones de tipo, hay que tener en mente que, una vez creado en memoria, un objeto no cambiará de tipo hasta el final de su vida. Así, si creamos una instancia de la clase `Persona`, dicha instancia siempre será una instancia de la clase `Persona`. Con este preámbulo, nos preguntamos qué significado real tiene el término «conversión». En realidad, la conversión no interviene en el objeto en sí, sino en la manera de manipularlo. Va a actuar sobre el tipo de la variable utilizada para acceder al objeto.

La relación de herencia entre clases es el elemento fundamental que permite el uso de conversiones de tipo. Hemos visto que una clase podía recuperar las características de otra clase por este medio. Es decir, que recupera automáticamente las características de su clase base. Por ejemplo, podemos tener la jerarquía de clases siguiente.



La clase `Cliente` es una evolución de la clase `Persona` de la misma manera que la clase `Proveedor` es una evolución de la clase `Persona`. Todas las características asociadas a una instancia de la clase `Persona` estarán también disponibles en una instancia de la clase `Cliente` o en una instancia de la clase `Proveedor`.

De hecho, nuestro esquema no está del todo completo y debería tener más bien la forma siguiente:



Nunca se debe olvidar que una clase base hereda de manera implícita de la clase `Object`. Por lo tanto, podemos decir que cualquier instancia de clase tendrá como mínimo las características de una instancia de la clase `Object`.

Tras esta puesta en escena, veamos con un juego lo que podemos hacer o no con las conversiones.

Tomando como partida el código siguiente:

```

Object o;
Persona p;
Cliente c;
Proveedor pr;
  
```

Debemos indicar qué líneas, entre las siguientes, son las que dan problema.

<code>pr=new Proveedor();</code>		
<code>o=pr;</code>	☐ funciona	☐ no funciona
<code>p=pr;</code>	☐ funciona	☐ no funciona
<code>c=pr;</code>	☐ funciona	☐ no funciona
<code>c=new Cliente();</code>		
<code>o=c;</code>	☐ funciona	☐ no funciona
<code>p=c;</code>	☐ funciona	☐ no funciona
<code>pr=c;</code>	☐ funciona	☐ no funciona
<code>p=new Persona();</code>		

<code>o=p;</code>	☐ funciona	☐ no funciona
<code>c=p;</code>	☐ funciona	☐ no funciona
<code>pr=p;</code>	☐ funciona	☐ no funciona
<code>o=new Object();</code>		
<code>p=o;</code>	☐ funciona	☐ no funciona
<code>pr=o;</code>	☐ funciona	☐ no funciona
<code>c=o;</code>	☐ funciona	☐ no funciona

Aquí está la solución:

<code>pr=new Proveedor();</code>		
<code>o=pr;</code>	☒ funciona	☐ no funciona
<code>p=pr;</code>	☒ funciona	☐ no funciona
<code>c=pr;</code>	☐ funciona	☒ no funciona
<code>c=new Cliente();</code>		
<code>o=c;</code>	☒ funciona	☐ no funciona
<code>p=c;</code>	☒ funciona	☐ no funciona
<code>pr=c;</code>	☐ funciona	☒ no funciona
<code>p=new Persona();</code>		
<code>o=p;</code>	☒ funciona	☐ no funciona
<code>c=p;</code>	☐ funciona	☒ no funciona
<code>pr=p;</code>	☐ funciona	☒ no funciona
<code>o=new Object();</code>		
<code>p=o;</code>	☐ funciona	☒ no funciona
<code>pr=o;</code>	☐ funciona	☒ no funciona
<code>c=o;</code>	☐ funciona	☒ no funciona

Podemos deducir la regla siguiente de este resultado: con una variable de tipo X, podemos referenciar sin problemas una instancia de clase de tipo X, pero también una instancia de cualquier subclase de la clase X. Así, una variable de tipo `Object` puede utilizarse para referenciar una instancia de cualquier clase.

Para proseguir con nuestros experimentos con las conversiones, hagamos un segundo ejercicio con las mismas consignas que el anterior.

```

public static void main(String args[])
{
    ...
    o=new Object();
    p=new Persona();
    c=new Cliente();
    pr=new Proveedor();

    prueba(o);           ☐ funciona           ☐ no funciona
    prueba(p);           ☐ funciona           ☐ no funciona
    prueba(pr);          ☐ funciona           ☐ no funciona
    prueba(c);           ☐ funciona           ☐ no funciona
    ...
}

```

```

public static void prueba(Object obj)
{
    Object o;
    Persona p;
    Cliente c;
    Proveedor pr;

    o=obj;               ☐ funciona           ☐ no funciona
    p=obj;               ☐ funciona           ☐ no funciona
    c=obj;               ☐ funciona           ☐ no funciona
    pr=obj;              ☐ funciona           ☐ no funciona
}

```

Aquí está la solución de este segundo ejercicio:

```

public static void main(String args[])
{
    ...
    o=new Object();
    p=new Persona();
    c=new Cliente();

```

```

    pr=new Proveedor();

    prueba(o);           ■ funciona       □ no funciona
    prueba(p);           ■ funciona       □ no funciona
    prueba(pr);          ■ funciona       □ no funciona
    prueba(c);           ■ funciona       □ no funciona
    ...
}

public void prueba(Object obj)
{
    Object o;
    Persona p;
    Cliente c;
    Proveedor pr;

    o=obj;               ■ funciona       □ no funciona
    p=obj;               □ funciona       ■ no funciona
    c=obj;               □ funciona       ■ no funciona
    pr=obj;              □ funciona       ■ no funciona
}

```

Las reglas que hay que seguir para resolver este problema son las mismas que para el ejercicio anterior. En el momento de la llamada a la función `prueba`, el paso del parámetro equivale a una asignación a una variable de tipo `Object`. Por lo tanto, podemos llamar a esta función proporcionándole una instancia de cualquier clase.

En el interior de la función `prueba` el problema se invierte. El parámetro `obj` referencia una instancia de clase, pero el compilador no puede saber de qué clase se trata. Por eso, acepta únicamente la asignación `o=obj`; sin riesgo ya que las dos variables son del mismo tipo. Sin embargo, es posible resolver este problema realizando una operación de tipado dinámico. Este tipo de operación se lleva a cabo simplemente haciendo preceder la variable con el nombre de la clase hacia la cual queremos realizar el tipado dinámico. El nombre de la clase debe estar ubicado entre los caracteres (y) (paréntesis).

```

public static void prueba(Objeto obj)
{

```



```
Object o;  
Persona p;  
Cliente c;  
Proveedor pr;  
  
o=obj;  
p=(Persona)obj;  
c=(Cliente)obj;  
pr=(Proveedor)obj;  
}
```

No obtenemos más errores en tiempo de compilación porque hemos pedido al compilador que «confíe» en nosotros. En cambio, en tiempo de ejecución, obtenemos una excepción del tipo `ClassCastException`. Esta excepción la provoca la máquina virtual cuando descubre que le hemos mentado al intentar hacerle creer que la variable `obj` hace referencia a una instancia de una determinada clase, cuando no es el caso. Debemos tener más prudencia con nuestras operaciones de tipado dinámico y comprobar a qué hace realmente referencia la variable `obj`. El operador `instanceof` permite efectuar esta comprobación. El operador devuelve `true` si la variable referencia un objeto del tipo evaluado o de un subtipo del tipo evaluado.

Imaginemos la siguiente situación:

```
Object o = new Cliente();
```

Las instrucciones siguientes devuelven `true`:

```
o instanceof Cliente  
o instanceof Persona  
o instanceof Object
```

La instrucción siguiente devuelve `false`:

```
o instanceof Proveedor
```

Por lo tanto, la función `intento` se puede escribir de la siguiente manera:

```

public static void prueba(Object obj)
{
    Object o;
    Persona p;
    Cliente c;
    Proveedor pr;

    o=obj;
    if (obj instanceof Persona)
    {
        p=(Persona)obj;
    }
    if (obj instanceof Cliente)
    {
        c=(Cliente)obj;
    }
    if (obj instanceof Proveedor)
    {
        pr=(Proveedor)obj;
    }
}

```

Conviene utilizar este operador para comprobar la viabilidad de una operación de tipado dinámico.

f. La clase Object

La clase `Object` está directa o indirectamente presente en la jerarquía de todas las clases de una aplicación. Los métodos definidos en la clase `Object` están, por lo tanto, disponibles para cualquier clase. El inconveniente de estos métodos es que no hacen nada útil o su funcionamiento no está adaptado a las diferentes clases de la aplicación. Por este motivo es preciso sustituirlos, a menudo en las clases de la aplicación, para poder utilizarlos de manera eficaz. Por lo tanto es importante conocer a fondo su utilidad para adaptar su funcionamiento. Vamos a estudiar en detalle los más utilizados.

`hashCode`

Este método permite obtener la dirección de memoria donde se almacena una instancia de

clase. De por sí, no es muy útil, salvo para algunos experimentos que haremos en el párrafo siguiente.

clone

El método `clone` puede reivindicar el título de «fotocopiador» de objetos. Mediante este método podemos obtener una copia compulsada de un objeto presente en la memoria. Para que esté disponible, es indispensable que la clase desde la que se ha creado el objeto que se desea copiar implemente la interfaz `Cloneable`. Esta interfaz exige la creación de un método `clone` en la clase. Este método es a menudo muy sencillo, ya que basta con llamar al método `clone` de la clase `Object`.

Esta versión por defecto se limita a efectuar una copia de la zona de memoria correspondiente a la instancia de clase que se ha de duplicar. Si la instancia que se ha de duplicar contiene referencias hacia otros objetos, estos últimos no se duplicarán, sino que se compartirán por el original y la copia. Si este funcionamiento no se adapta a la aplicación, hay que diseñar el método `clone` para que efectúe también una copia de los objetos referenciados.

Para ilustrar este mecanismo, vamos a trabajar con la clase `Cliente`, a la cual vamos a asociar una clase `Pedido`, que a su vez está asociada a una clase `LineasDeComando`.



Funcionalmente, puede ser útil poder duplicar un pedido para hacer un nuevo pedido idéntico. Si realizamos una copia de un pedido, es interesante conservar la referencia hacia el cliente pero, en cambio, es preciso duplicar las líneas del pedido. Debemos diseñar el método `clone` de la clase `Pedido` para que duplique también la instancia de la clase `LineasDePedido` referenciada. A continuación presentamos un extracto de código de estas clases.

```
public class Pedido implements Cloneable
{
    Cliente elCliente;
    LineasDePedido lasLineas;
```

```

public Pedido()
{
    super();
    lasLíneas=new LineasDePedido();

}
public Object clone() throws CloneNotSupportedException
{
    Pedido ped;
    // creación de una copia del pedido
    ped=(Pedido)super.clone();
    // duplicación de las líneas del pedido
    ped.lasLíneas=(LineasDePedido)lasLíneas.clone();
    return ped;

}
public Cliente getElCliente()
{
    return elCliente;
}
public void setElCliente(Cliente elCliente)
{
    this.elCliente = elCliente;
}
public LineasDePedido getLasLíneas()
{
    return lasLíneas;
}
public void setLasLíneas(LineasDePedido lasLíneas)
{
    this.lasLíneas = lasLíneas;
}
}

/*****/
public class LineasDePedido implements Cloneable
{
    public Object clone() throws CloneNotSupportedException

```

```

    {
        return super.clone();
    }
}

```

A continuación, podemos crear instancias de nuestras clases y comprobar su correcto funcionamiento.

```

public class TestClone {

    public static void main(String[] args)
        throws CloneNotSupportedException
    {
        Cliente c;
        c=new Cliente("ENI",
            "",
            LocalDate.of(1981,05,15),
            TipoCliente.EMPRESA);

        Pedido ped1,ped2;
        // creación e inicialización de un pedido
        cmd1=new Pedido();
        cmd1.setElCliente(c);
        System.out.println("hashCode del pedido: " +
            cmd1.hashCode());
        System.out.println("hashCode del Cliente: " +
            cmd1.getElCliente().hashCode());
        System.out.println("hashCode de las líneas: " +
            cmd1.getLasLíneas().hashCode());
        cmd2=(Pedido)cmd1.clone();
        System.out.println("hashCode de la copia: " +ped2.hashCode());
        System.out.println("hashCode del Cliente de la copia: " +
            ped2.getElCliente().hashCode());
        System.out.println("hashCode de las líneas de la copia:" +
            ped2.getLasLíneas().hashCode());
    }
}

```

Observe la gestión de la excepción `CloneNotSupportedException` a nivel de la firma del método `main`. Para más información sobre las excepciones, vaya a la sección dedicada a este tema un poco más adelante en este capítulo.

La ejecución de este código da:

```
<terminated> TestClone [Java Application] /usr/lib/  
hashCode del pedido: 617901222  
hashCode del Cliente: 1159190947  
hashCode de las líneas : 925858445  
hashCode de la copia: 798154996  
hashCode del Cliente de la copia: 1159190947  
hashCode de las líneas de la copia: 681842940
```

Observando el `hashCode` de dos pedidos distintos, es posible constatar que son dos objetos diferentes que hacen referencia al mismo cliente.

`equals`

Se utiliza el método `equals` para comparar dos instancias. El código siguiente permite verificar la igualdad de dos clientes.

```
public class TestEquals {  
    public static void main(String[] args) {  
        Cliente c1,c2;  
        c1=new Cliente("ENI",  
            "",  
            LocalDate.of(1981,05,15),  
            TipoCliente.EMPRESA);  
        c2=new Cliente("ENI",  
            "",  
            LocalDate.of(1981,05,15),  
            TipoCliente.EMPRESA);  
        if (c1.equals(c2))  
        {  
            System.out.println("los dos clientes son idénticos");  
        }  
        else  
        {  
            System.out.println("los dos clientes son diferentes");  
        }  
    }  
}
```

A pesar de las apariencias, la ejecución de este código nos muestra el resultado siguiente:

```
los dos clientes son diferentes
```

La implementación de este método realiza una comparación de las referencias para determinar si hay igualdad entre ambos objetos. Si queremos tener un criterio de comparación diferente, debemos sustituir el método `equals` en la clase `Cliente` utilizando nuestros propios criterios de comparación.

```
@Override
public boolean equals(Object obj)
{
    Cliente c;
    // verificación si obj es null o referencia una instancia
    // de otra clase
    if (obj ==null || obj.getClass() !=this.getClass())
    {
        return false;
    }
    else
    {
        c=(Cliente)obj;
        // verificación de los criterios de igualdad sobre
        // - el apellido
        // - el nombre
        // - la fecha de nacimiento
        // - el tipo de cliente
        if (c.getApellido().equals(this.getApellido())&
            c.getNombre().equals(this.getNombre()) &
            c.getFechaDeNacimiento().
                equals(this.getFechaDeNacimiento()) &
            c.getTipo().equals(this.getType()) )
        {
            return true;
        }
        else
        {

```

```

    }
    }
    return false;
}

```

Dos objetos considerados idénticos por el método `equals` deben tener el mismo `hashCode`. Por lo tanto, es necesario sustituir este método en caso de sustitución del método `equals` para mantener esta consistencia.

A continuación se muestra una posible implementación del método hashCode:

```
public int hashCode()  
{  
    return this.getApellido().hashCode()+  
           this.getNombre().hashCode()+  
           this.getFechaDeNacimiento().hashCode()+  
           this.getTipo().hashCode();  
}
```

getClass

Este método proporciona una instancia de la clase `Class` que contiene las características de la clase desde la cual se creó el objeto. Podemos, por ejemplo, obtener el nombre de la clase, los métodos disponibles, los campos, etc. Como medida de seguridad, no se puede sustituir este método.

A continuación, presentamos una función que muestra algo de información de la clase del objeto que se le ha pasado como parámetro.

```
public static void infoClase(Object o)
{
    Class c;
    c=o.getClass();
    System.out.println("nombre de la clase: " + c.getName());
    System.out.println("está en el paquete: " +
        c.getPackage().getName());
    System.out.println("hereda de la clase: " +
        c.getSuperclass().getName());
}
```



```

System.out.println("posee los campos: ");
for (int i=0;i<c.getFields().length;i++)
{
    System.out.print("\t" + c.getFields()[i].getName());
    System.out.println(" de tipo:" +
        c.getFields()[i].getType().getName());
}
System.out.println("posee los métodos: ");
for (int i=0;i<c.getMethods().length;i++)
{
    System.out.print("\t" + c.getMethods()[i].getName());
    System.out.print(" que espera como parámetro (");
    for (int j=0;j<c.getMethods()[i]
        .getParameterTypes()
        .length;j++)
    {
        System.out.print(c.getMethods()[i]
            .getParameterTypes()[j]+
                " ");
    }
    System.out.println(")");
}
}

```

Este método es uno de los puntos de entrada de la API de *Reflection*. Esta API es muy útil para el establecimiento de frameworks como ORM (*Object Relational Mapping*) o frameworks que usan la inyección de dependencias como Spring.

toString

A diferencia de lo visto hasta ahora, este método debería estar sobrecargado casi siempre. Permite obtener la representación de un objeto bajo la forma de una cadena de caracteres. Por defecto, la implementación de este método en la clase `Object` devuelve el nombre de la clase seguido del `hashCode` de la instancia. Por supuesto, se aconseja una representación más elocuente. Debería construirse a partir de valores contenidos en los campos del objeto. Una posible versión del método `toString` para la clase `Persona` podría ser, por ejemplo, la siguiente.

```

@Override
public String toString()
{
    String cadena;
    cadena="apellido: " + this.getApellido()+ "\r\n";
    cadena=cadena + "nombre: " + this.getNombre();
    return cadena;
}

```

La llamada al método `toString` está a veces implícita cuando se pasa a una función un objeto como parámetro.

```

public class TestToString {
    public static void main(String[] args) {
        Cliente c;
        c=new Cliente("ENI",
            "",
            LocalDate.of(1981,05,15),
            TipoCliente.EMPRSA);
        //El método println recibe un objeto como argumento
        //comprueba que no es nulo el objeto antes de llamar
        //al método toString de este objeto
        System.out.println(c);
        //El método println siguiente muestra la cadena
        //de caracteres que se pasa como argumento
        System.out.println(c.toString());
    }
}

```

5. Interfaces

Hemos visto que podemos obligar a una clase a implementar un método declarándolo con la palabra clave `abstract`. Si varias clases deben implementar el mismo método, es posible utilizar una interfaz. El uso de una interfaz es más conveniente que la clase abstracta, si las clases deben implementar el mismo método, no tienen clases madre comunes. Como las clases, las interfaces permiten definir un conjunto de constantes y

métodos. Por lo general, una interfaz contiene únicamente firmas de métodos, siendo en cierto modo similares a los métodos abstractos definidos en una clase abstracta. La interfaz constituye un contrato firmado por la clase. Al declarar que la clase implementa una interfaz, se compromete a proporcionar todo lo definido en la interfaz. Conviene tener precaución cuando se utilizan interfaces muy a menudo y se modifica alguna, pues se corre el riesgo de tener que recodificar todas las clases que implementen dicha interfaz.

A partir de la versión 8 del lenguaje Java, esta restricción puede evitarse creando métodos por defecto en la interfaz (consulte la sección Método por defecto).

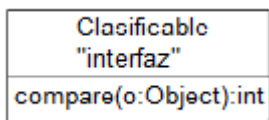
a. Creación de una interfaz

Para poder utilizar una interfaz hay que definirla previamente. La declaración es similar a la de una clase, pero se utiliza la palabra clave `interface` en lugar de la palabra clave `class`.

También se puede utilizar la palabra clave `extends` para introducir una relación de herencia en la interfaz. A diferencia de las clases, las interfaces autorizan la herencia múltiple. En este caso, los nombres de las interfaces heredadas están separados por comas después de la palabra clave `extends`. Conviene ser cauto con esta posibilidad, ya que las clases que implementaron esta interfaz tendrán que proporcionar todos los métodos definidos en la jerarquía de la interfaz.

Vamos a crear nuestra primera interfaz. Esta va a imponer la presencia de una función `compare` que recibe un objeto como parámetro. Durante la definición de una interfaz se recomienda proporcionar, como comentarios, una descripción del trabajo que deberá realizar cada método, así como de los resultados que deberá facilitar.

La representación UML de esta interfaz es la siguiente:



```
// se deberá implementar esta interfaz con las clases
// para las cuales se considera una clasificación de las instancias
public interface Clasificable
{
    // este método se podrá llamar para comparar la instancia actual
```

```

// con la recibida como parámetro
// el método devuelve un entero cuyo valor depende
// de las reglas siguientes
// 1 si la instancia actual es superior a la recibida
// como parámetro
// 0 si las dos instancias son iguales
// -1 si la instancia actual es inferior a la recibida como parámetro
// -99 si la comparación es imposible

    int compare(Object o);
    public static final int INFERIOR=-1;
    public static final int IGUAL=0;
    public static final int SUPERIOR=1;
    public static final int ERROR=-99;
}

```

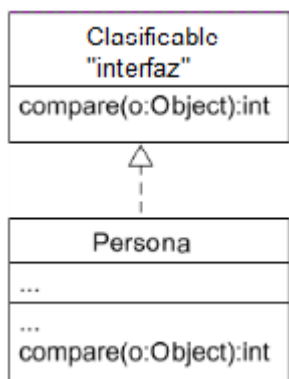
Este ejemplo se inspira en una interfaz existente de la API Java: `Comparable`.

b. Utilización de una interfaz

Pero ¿qué criterio vamos a usar para decir que un objeto es superior a otro?

En la descripción de nuestra interfaz, ¡no tenemos que preocuparnos! Dejamos al usuario, que va a definir una clase que implemente la interfaz, que se encargue de definir los criterios de comparación.

La representación UML para indicar que una clase implementa una interfaz es la siguiente:



La clase `Persona` se encarga de implementar la interfaz `Clasificable` y respetar el rol del método `compare` tal y como se describe en la javadoc.

Por ejemplo: en la clase `Persona`, se podría implementar la interfaz `Clasificable` de la manera siguiente, eligiendo comparar dos instancias de la clase `Persona` en el nombre:

```
public class Persona
implements Clasificable
{
    ...
    public int compare(Object o)
    {
        Persona p;
        if (o instanceof Persona)
        {
            p=(Persona)o;
        }
        else
        {
            return Clasificable.ERROR;
        }
        if (getApellido().compareTo(p.getApellido())<0)
        {
            return Clasificable.INFERIOR;
        }
        if (getApellido().compareTo(p.getApellido())>0)
        {
            return Clasificable.SUPERIOR;
        }
        return Clasificable.IGUAL;
    }
    ...
}
```

Hay dos modificaciones llamativas en la clase:

- El hecho de que implemente la interfaz `Clasificable`.
- La implementación real de la función `compare`.

En esta función, la comparación se hará basándose en el apellido de los clientes. Muy bien,

pero ¿para qué sirve?

A menudo es necesario clasificar elementos en una aplicación. Se proponen dos soluciones:

- Crear una función de clasificación específica para cada tipo de elemento que se quiere clasificar.
- Crear una función de clasificación genérica de manera que los elementos que se utilizan sean clasificables mediante esta función.

Las interfaces nos van a ayudar a poner en práctica esta segunda solución. Para poder ordenar elementos, sea cual sea el método utilizado para ello, necesitaremos comparar dos elementos. Para asegurarnos de que nuestra rutina de clasificación funciona de manera correcta, es preciso que los elementos que va a ordenar pueden compararse los unos con los otros.

Solo lo podemos garantizar si todos nuestros elementos implementan la interfaz `Clasificable`. Por lo tanto, vamos a exigirlo en la declaración de nuestra rutina de clasificación.

```
public static Clasificable[] clasificar(Clasificable[] array)
{
}
```

Así definida, la función será capaz de ordenar cualquier tipo de arrays siempre que sus elementos implementen la interfaz `Clasificable`. Devolverá el array clasificado. Por lo tanto, podemos escribir el código siguiente y utilizar el método `compare` sin riesgo.

```
public static Clasificable[] clasificar(Clasificable[] array)
{
    int i,j;
    Clasificable c;
    Clasificable[] arrayOrdenado;
    arrayOrdenado = Arrays.copyOf(array, array.length);
    for (i=0;i< arrayOrdenado.length;i++)
    {
        for( j = i + 1; j<arrayOrdenado.length;j++)
        {
            if (arrayOrdenado[j].compare(arrayOrdenado[i])==
```

```

                                Clasificable.INFERIOR)
        {
            c = arrayOrdenado[j];
            arrayOrdenado[j] = arrayOrdenado[i];
            arrayOrdenado[i] = c;
        }
        else if (arrayOrdenado[j].compare(arrayOrdenado[i])==
                                Clasificable.ERROR)
        {
            return null;
        }
    }
}
return arrayOrdenado;
}

```

Para comprobar esta función, vamos a crear e intentar ordenar algunos clientes, para luego visualizar sus apellidos.

```

public static void main(String[] args)
{
    Persona [] arrayNoOrdenado = new Persona[5];
    arrayNoOrdenado[0] = new Persona("PEPE2",
                                    "apellido2",
                                    LocalDate.of(1922,2,15));
    arrayNoOrdenado[1] = new Persona("PEPE1",
                                    "apellido 1",
                                    LocalDate.of(1911,1,15));
    arrayNoOrdenado[2] = new Persona("PEPE5",
                                    "apellido 5",
                                    LocalDate.of(1955,05,15));
    arrayNoOrdenado[3] = new Persona("PEPE3",
                                    "apellido 3",
                                    LocalDate.of(1933,03,15));
    arrayNoOrdenado[4] = new Persona("PEPE4",
                                    "apellido 4 ",
                                    LocalDate.of(1944,04,15));

    Persona[] ArrayOrdenado;
}

```

```

ArrayOrdenado=(Persona[])ordenar(arrayNoOrdenado);
for (int i=0;i<ArrayOrdenado.length;i++)
{
    System.out.println(ArrayOrdenado[i]);
}
}

```

La ejecución de este programa da el resultado siguiente:

```

<terminated> TestClassable[Java Application] /usr/lib/jvm/java-11-openjdk
nombre: PEPE1, apellido: apellido1, fecha de nacimiento: 15-01-1911
nombre: PEPE2, apellido: apellido2, fecha de nacimiento: 15-02-1922
nombre: PEPE3, apellido: apellido3, fecha de nacimiento: 15-03-1933
nombre: PEPE4, apellido: apellido4, fecha de nacimiento: 15-04-1944
nombre: PEPE5, apellido: apellido5, fecha de nacimiento: 15-05-1955

```

La lista de nuestros clientes se ordena alfabéticamente según su apellido.

No es posible proporcionar a la función de ordenación un array de objetos que no implementa la interfaz `Clasificable`. El compilador muestra un error de compilación en este caso. El siguiente ejemplo resalta este comportamiento si se proporciona un array de `String`:

```

6 String[] arrayDeCadenas = {"Hola","todo","el","mundo"};
7 ordenar(arrayDeCadenas);
8
The method ordenar(Classable[]) in the type TestClassable is not applicable for the arguments (String[])

```

Este error se produce cuando se invoca la función `ordenar`. Los elementos del array, que hemos pasado como parámetro, no implementan la interfaz `Clasificable` y, por lo tanto, el compilador no está seguro de que contengan una función `compare`.

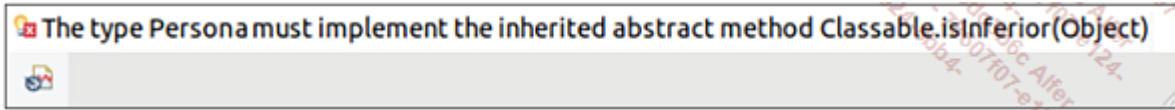
Destaquemos que, incluso si existe una función `compare` correcta en la clase `String`, no será más utilizable porque la clase `String` no implementa la interfaz `Clasificable`.

c. Método por defecto

Ahora que el código funciona correctamente, es posible optimizar el algoritmo de ordenación utilizado. Para realizar dicha mejora, necesitamos que los objetos que queremos ordenar provean dos métodos suplementarios. Podemos, entonces, agregar a la interfaz `Clasificable` la firma de estos dos métodos.


```
boolean isInferior(Object o);  
boolean isSuperior(Object o);
```

Agregar estos dos métodos provoca, de inmediato, un problema en tiempo de compilación de la clase `Persona` y en todas aquellas que implementan la antigua versión de la interfaz.



Si se han implementado muchas clases con la antigua versión de la interfaz, va a ser necesario realizar un importante trabajo de modificación para que respeten la nueva versión de la interfaz.

Para evitar este largo y tedioso trabajo, es posible definir los dos métodos que se agregan a la interfaz con la palabra clave `default`. Es preciso, también, proveer una implementación por defecto para ambos métodos. Deben, por tanto, incluir un bloque de código delimitado por llaves. Pero ¿qué podríamos incluir en este bloque de código dado que no sabemos sobre qué interfaz se va a aplicar? Poco importa, este bloque de código es importante por su presencia y no por su contenido. Permite, simplemente, a las clases que implementan la antigua versión de la interfaz ser compatibles con la nueva versión sin tener que proporcionar de forma obligatoria una implementación de estos nuevos métodos.

Por lo tanto, la estructura de la interfaz puede tener la siguiente forma:

```
public interface Clasificable  
{  
  
    int compare(Object o);  
    default boolean isInferior(Object o)  
    {  
        return false;  
    }  
    default boolean isSuperior(Object o)  
    {  
        return false;  
    }  
    public static final int INFERIOR=-1;
```

```
public static final int IGUAL=0;
public static final int SUPERIOR=1;
public static final int ERROR=-99;

}
```

¿Cuál es la diferencia entre una interfaz con métodos por defecto y una clase abstracta? Se basa en el hecho de que el método predeterminado es heredado por clases que no necesariamente tienen clases primarias comunes. Por lo tanto, podemos considerar en cierta medida que las interfaces con métodos predeterminados nos permiten configurar una herencia múltiple.

Si alguna clase no implementa todos los métodos de la interfaz, hereda los métodos definidos por defecto en la interfaz. Siempre existe la posibilidad de que provean su propia implementación de dichos métodos. Con esta nueva definición de la interfaz, la clase `Persona` no requiere modificarse. Podemos, no obstante, crear otras clases que implementen completamente la interfaz proveyendo todos los métodos de la interfaz.

```
public class Coche implements Clasificable
{
    private String matricula;
    private String marca;
    private String modelo;
    private int potencia;

    ...
    public int getPotencia()
    {
        return potencia;
    }
    /**
     * Comparación de la potencia del coche
     */
    @Override
    public int compare(Object o)
    {
        Coche c;
```

```

        if (o instanceof Coche)
        {
            c=(Coche)o;
        }
        else
        {
            return Clasificable.ERROR;
        }
        if (this.getPotencia()<c.getPotencia())
        {
            return Clasificable.INFERIOR;
        }
        if (this.getPotencia()>c.getPotencia())
        {
            return Clasificable.SUPERIOR;
        }

        return Clasificable.IGUAL;
    }

    /**
     * @devuelve true si la potencia del coche es inferior
     */

    @Override
    public boolean isInferior(Object o)
    {
        return this.compare(o)==Clasificable.INFERIOR;
    }

    /**
     * @devuelve true si la potencia del coche es superior
     */

    @Override
    public boolean isSuperior(Object o)
    {
        return this.compare(o)==Clasificable.SUPERIOR;
    }

}

```

d. Método estático

Es posible agregar métodos estáticos en una interfaz. Para obtener más información sobre métodos estáticos, consulte la sección Creación de campos y métodos estáticos, anteriormente en el capítulo.

6. Clases anidadas

La mayoría de las clases que utilizaremos en una aplicación se definirán en su propio archivo de código fuente. Sin embargo, Java ofrece la posibilidad de declarar una clase en el interior de otra, o incluso en el interior de un método. Esta técnica permite definir una clase únicamente en un contexto donde sea realmente útil. Se designa a las clases anidadas con el término de clases asistentes. Según la ubicación de su declaración, tienen acceso bien a los demás miembros de la clase en la cual están declaradas (incluso los miembros privados) o bien solamente a las variables locales de los métodos.

a. Clases anidadas estáticas

Como cualquier elemento declarado en una clase (campo o método), una clase anidada puede declararse con la palabra clave `static`. En este caso, se la somete a las mismas reglas que las impuestas por esta palabra clave a los demás elementos de una clase.

- No puede emplear los campos y métodos estáticos de su clase contenedora.
- Puede utilizarse (instanciarse) sin que exista una instancia de su clase contenedora.
- Puede utilizar los miembros de instancia de su clase contenedora únicamente mediante una instancia de dicha clase.

Aquí tenemos un ejemplo sencillo de clase anidada estática.

```
public class Externa {  
    static final double TASA=1.2;  
    static class Interna  
    {  
        public double calculoNeto(double precio)  
        {  
            return precio*Externa.TASA;  
        }  
    }  
}
```

```
    }  
  }  
}
```

Se puede utilizar de manera muy parecida a cualquier otra clase. La única obligación reside en el nombre utilizado para hacer referencia a ella en el código, que debe ir precedido del nombre de la clase contendora si se debe crear una instancia de la clase interna desde el exterior de la clase contenedora.

```
public class Test {  
    public static void main(String[] args) {  
        Externa.Interna objeto =new Externa.Interna();  
        System.out.println(objeto.calculoNeto(100));  
    }  
}
```

Esto es bastante raro, porque la clase anidada estática es a menudo privada y, por lo tanto, las instancias se crean dentro de la clase contenedora. En este caso, la instrucción es bastante clásica:

```
Interna interna = new Interna();
```

b. Clases internas (de instancia)

Las clases anidadas se conocen también con el nombre de clases internas (de instancia). Se las somete a las mismas reglas que a cualquier otro elemento (de instancia) declarado en una clase.

- Pueden tener acceso a todos los miembros de la clase en la cual están declaradas, incluso los miembros privados.
- Solo pueden emplearse si hay disponible alguna instancia de la clase contenedora.

La declaración es muy sencilla y similar a la anterior, salvo en la palabra clave `static`, que desaparece.

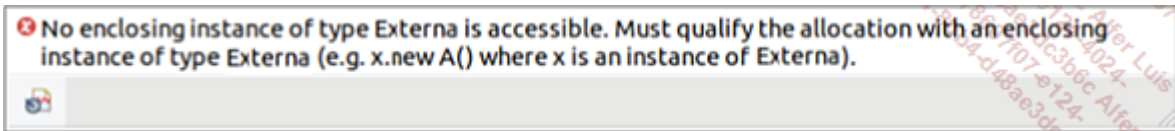
```
public class Externa {  
    final double TASA=1.2;
```

```

class Externa
{
    public double calculoNeto(double precio)
    {
        return precio*TASA;
    }
}

```

En contraposición, si intentamos utilizarla con el mismo código que la versión anterior, encontramos problemas en tiempo de compilación.



En efecto, debemos disponer obligatoriamente de una instancia de la clase contenedora para que la clase interna esté disponible. Por lo tanto, debemos previamente instanciar la clase `Externa` y, a continuación, pedirle que nos proporcione una instancia de la clase `Interna`. De este modo, la sintaxis correcta es la siguiente:

```

public class Test {
    public static void main(String[] args) {
        Externa e;
        e=new Externa();
        Externa.Interna ci;
        ci=e.new Interna();
        System.out.println(ci.calculoNeto(100));
    }
}

```

Al igual que sucede con las clases anidadas estáticas, esto es bastante raro, ya que las clases internas a menudo son privadas y, por lo tanto, se crean instancias dentro de una instancia de la clase contenedora.

c. Clases anónimas

Una clase anónima es una clase interna para la cual no se ha definido ningún nombre. Las

clases anónimas están perfectamente adaptadas para realizar una operación que necesita un objeto, pero que no justifica la creación de una clase normal. Es el caso, por ejemplo, de clases sencillas o de aquellas que solo se utilizan en un único método. Este mecanismo se emplea a menudo para gestionar las acciones del usuario en una aplicación en modo gráfico.

Para ilustrar el uso de las clases internas anónimas, vamos a retomar la función de clasificación del array creada anteriormente y hacerla más universal. En su versión actual, exige que los elementos del array implementen la interfaz `Clasificable` para que pueda compararlos de dos en dos.

El objetivo es modificar ligeramente la función para que, en el momento de la llamada, se le pueda proporcionar, además del array que hay que clasificar, la función que deberá usar para efectuar las comparaciones. La solución estándar para transmitir una función es proporcionar una instancia de clase que contiene la función a transmitir. Por medida de seguridad, podemos exigir que esta instancia esté creada desde una clase que implemente la interfaz, para garantizar la existencia de la función. A continuación, presentamos el código de esta nueva interfaz:

```
public interface Comparador
{
    // se podrá llamar a este método para comparar los dos objetos
    // recibidos como parámetros
    // el método devuelve un entero cuyo valor depende
    // de las reglas siguientes
    * @return
    // 1 si la instancia o1 es superior a o2
    // 0 si las dos instancias son iguales
    // -1 si la instancia o1 es inferior a o2
    // -99 si la comparación es imposible
    int compare(Object o1, Object o2);

    public static final int INFERIOR=-1;
    public static final int IGUAL=0;
    public static final int SUPERIOR=1;
    public static final int ERROR=-99;
}
```

Este ejemplo se inspira en una interfaz existente en la API de Java: `Comparator`.

Una interfaz así, que no contiene más que la definición de único método, se llama interfaz funcional.

Ahora es posible revisar la función de clasificación teniendo en cuenta esta mejora.

```
public static Object[] clasificación(Object[] array,
Comparator clasificador)
{
    int i,j;
    Object c;
    Object[] arrayClasificación;
    arrayClasificación=Arrays.copyOf(array,array.length);
    for (i=0;i< arrayClasificación.length;i++)
    {
        for( j = i + 1; j<arrayClasificación.length;j++)
        {
            // utiliza la función compare del objeto recibido como
parámetro
            // para comparar el contenido de dos celdas del array
            if
(clasificador.compare(arrayClasificación[j],arrayClasificación[i])==
Comparator.INFERIOR)
            {
                c = arrayClasificación[j];
                arrayClasificación[j] = arrayClasificación[i];
                arrayClasificación[i] = c;
            }
            else if
(clasificador.compare(arrayClasificación[j],arrayClasificación[i])==
Comparator.ERROR)
            {
                return null;
            }
        }
    }
    return arrayClasificación;
}
```



```
}
```

El array que se ha de clasificar ya no tiene que implementar necesariamente la interfaz `Clasificable` porque la metodología ha cambiado: ahora es necesario proporcionar un objeto adicional de tipo `Comparador` que lleve el mecanismo de clasificación. Esta solución es más flexible que la anterior porque ahora es posible ordenar la misma tabla con múltiples mecanismos de comparación, al proporcionar un objeto de tipo `Comparador` adecuado para la clasificación deseada.

Para utilizar esta nueva función de clasificación, debemos proporcionarle dos parámetros:

- El array que se ha de clasificar.
- Una instancia de clase que implemente la interfaz `Comparador`.

Para facilitar esta instancia de clase, tenemos varias soluciones:

- Crear una instancia de una clase «normal» que implemente la interfaz.
- Crear una instancia de una clase interna que implemente la interfaz.
- Crear una instancia de una clase interna anónima que implemente la interfaz.

Esta última es la solución que vamos a utilizar. De hecho, para una clase dada, es posible querer n ordenaciones diferentes. Imagine la visualización de un array de datos, cuyas diferentes columnas queremos ordenar de forma ascendente y descendente. Así es difícil imaginar la creación de una clase, es decir, un archivo de Java, para cada una de estas ordenaciones. El número de archivos crecería enormemente y el uso es a menudo muy puntual y limitado a una funcionalidad específica. Este es uno de los límites del desarrollo de objetos. Por lo tanto, es preferible, en este caso, crear clases anónimas.

A primera vista, la sintaxis para crear una instancia de una clase interna anónima es bastante extraña.

```
new "nombre de la clase base o de la interfaz implementada" ()
{
    // declaración de los campos
    int i;
    float j;
    // declaración de los métodos
    public int método1(..., ...)
```

```
{  
    } // llave de cierre del bloque de código del método  
} // llave de cierre del bloque de código de la clase
```

Como para crear una instancia de clase normal, utilizamos el operador `new` seguido del nombre de la clase. De hecho, no indicamos directamente el nombre de la clase de la cual deseamos obtener una instancia, sino el nombre de la clase base o de la interfaz implementada por esta clase (¡la clase no tiene nombre, ya que es anónima!). A continuación, facilitamos un bloque de código delimitado por llaves que corresponden al contenido de la clase, como en la definición de una clase normal. Destaquemos que, si la clase anónima implementa una interfaz, este bloque de código debe proporcionar obligatoriamente todos los métodos exigidos por la interfaz.

He aquí la puesta en práctica de estos principios para la llamada de nuestra función de clasificación con el apellido de la persona como criterio de clasificación.

```
public static void main(String[] args) {  
    Persona[] tab = new Persona[5];  
    tab[0] = new Persona("juanito2",  
        "nombre2",  
        LocalDate.of(1922,2,15));  
    tab[1] = new Persona("juanito1",  
        "nombre1",  
        LocalDate.of(1911,1,15));  
    tab[2] = new Persona("juanito5",  
        "nombre5",  
        LocalDate.of(1955,05,15));  
    tab[3] = new Persona("juanito3",  
        "nombre3",  
        LocalDate.of(1933,03,15));  
    tab[4] = new Persona("juanito4",  
        "nombre4",  
        LocalDate.of(1944,04,15));  
  
    // creación de una instancia de clase que implementa la  
    interfaz  
    // Comparador  
    new Comparador()
```

```

// a continuación el código de la clase
{
    // como lo exige la interfaz, aquí está el método compare
    public int compare(Object o1, Object o2)
    {
        Persona p1,p2;
        if (o1 instanceof Persona & o2 instanceof Persona)
        {
            p1=(Persona)o1;
            p2=(Persona)o2;
        }
        else
        {
            return Comparador.ERROR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())<0)
        {
            return Comparador.INFERIOR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())>0)
        {
            return Comparador.SUPERIOR;
        }
        return Comparador.IGUAL;
    } // llave de cierre del método compare
} // llave de cierre de la clase
); // fin de la llamada de la función de clasificación

// visualización del array clasificado
for (int i=0;i<tabClasificación.length;i++)
{
    System.out.println(tabClasificación[i]);
}

}

```

Si queremos utilizar otra clasificación con un criterio distinto, debemos invocar sencillamente la función `clasificación` con una nueva instancia de una clase interna

anónima que implemente de manera diferente la interfaz `Comparador`. Es posible, por ejemplo, ordenar las personas por su edad si la función `compare` de la clase anónima que implementa la interfaz `Comparador` se corresponde con el siguiente código:

```
public int compare(Object o1, Object o2) {
    Persona p1, p2;
    if (o1 instanceof Persona & o2 instanceof Persona) {
        p1 = (Persona) o1;
        p2 = (Persona) o2;
    } else {
        return Comparador.ERROR;
    }
    if (p1.calcularEdad () < p2.calcularEdad ()) {
        return Comparador.INFERIOR;
    }
    if (p1.calcularEdad () > p2.calcularEdad ()) {
        return Comparador.SUPERIOR;
    }

    return Comparador.IGUAL;
}
```

Durante la compilación de este código se generan archivos `.class` para cada clase utilizada en el código. El compilador genera automáticamente un nombre para el archivo de cada clase anónima. Para ello, emplea el nombre de la clase contenedora, al cual añade el sufijo `$` seguido de un valor numérico. Por lo tanto, la compilación de este código va a generar los archivos siguientes:

- `TestComparador.class`: el archivo que corresponde a la clase `TestComparador` que contiene los métodos `main` y `ordena`.
- `TestComparador$1.class`: el archivo que corresponde a la primera clase interna anónima (comparación sobre el nombre).
- `TestComparador$2.class`: el archivo que corresponde a la segunda clase interna anónima (comparación sobre la edad).

El único inconveniente de las clases internas anónimas reside en el hecho de que son de un único uso. Si desea utilizarla varias veces, tendrá que crear en este caso una clase con un nombre definido, ya que el nombre asignado a la clase interna anónima por el compilador no es accesible desde el código (¡ya que es anónima!).

- ☐ La solución a este problema es usar las referencias de métodos en lugar y clases anónimas. Para aprender a usarlas, vaya al capítulo Las expresiones lambda.

Los paquetes

1. Presentación

La meta principal de los paquetes es organizar y ordenar las clases y las interfaces de una aplicación. El principio es el mismo que en la vida corriente. Si disponemos de un gran armario en casa, es obvio que la instalación de estanterías en este armario facilitará la organización y la búsqueda de los objetos guardados en relación con un almacenamiento en desorden.

La organización de las clases en paquetes conlleva las ventajas siguientes:

- ☐● Facilidad para encontrar y utilizar de nuevo un elemento.
- Limitación de los riesgos de conflictos de nombres.
- Creación de una nueva visibilidad (la visibilidad `package`) además de las estándares (`private`, `protected`, `public`). Cuando un elemento (una clase, un campo, un método) no tiene modificadores de visibilidad, entonces este elemento solo es visible por los elementos presentes en el mismo paquete. Hay una sutileza para los elementos protegidos (campos, métodos). Por supuesto, son accesibles en las clases derivadas, pero también en las clases del mismo paquete.

2. Creación de un paquete

Lo primero que hay que hacer cuando se quiere crear un paquete es darle un nombre. Hay que elegirlo con cuidado para permitir que el paquete cumpla totalmente con su papel. Debe ser único y representativo de los elementos almacenados en su interior.

Para asegurar la unicidad del nombre de un paquete, por convención se utiliza el nombre de dominio de la empresa invirtiendo el orden de los elementos como primera parte para el

nombre del paquete.

Por ejemplo, si el nombre del dominio es `eni.es`, la primera parte del nombre del paquete será `es.eni`. Si el nombre del dominio contiene caracteres prohibidos para los nombres de paquetes, se sustituyen por el carácter `_` (guión bajo). De este modo, para el nombre de dominio `eni-escuela.es`, la primera parte del nombre del paquete será `es.eni_escuela`. Esta primera parte del nombre permite garantizar la unicidad de los elementos respecto al exterior de la empresa. La parte siguiente del nombre del paquete asegurará la unicidad de los elementos en el interior de la aplicación. Hay que elegir nombres claros y representativos de los elementos del paquete. Por ejemplo, en una aplicación podemos tener dos clases `Cliente`, una que represente a una persona que hizo un pedido a nuestra empresa y otra que represente a un cliente en el sentido informático del término (cliente - servidor). Podríamos emplear los siguientes nombres de paquete para albergar estas dos clases. `es.eni_escuela.conta` para la clase que representa al cliente físico y `es.eni_escuela.red` para la clase que representa un cliente de software. Por convención, los nombres de los paquetes se escriben en minúsculas. Conviene indicar el nombre elegido en el archivo de código fuente de cada clase precedido de la palabra clave `package`.

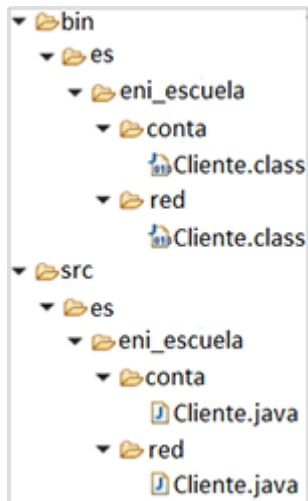
La declaración del paquete se hace obligatoriamente en la primera línea del archivo.

Todos los elementos alojados en este archivo de código fuente formarán parte de este paquete. Si no se indica ninguna información relativa al paquete en el archivo de código fuente, entonces se considerará que los elementos definidos en este archivo forman parte del paquete por defecto, que no tiene ningún nombre. Conviene ser prudente con este paquete por defecto, ya que las clases definidas en él no son accesibles desde un paquete con nombre.

Entre otros, este es el motivo por el que los diseñadores de Java recomiendan ubicar siempre una clase en un paquete con nombre.

El uso de los paquetes impone también una organización específica a la hora de grabar en disco los archivos de código fuente y los archivos compilados. Las carpetas donde se graban estos archivos deben respetar los nombres utilizados para los paquetes. Cada componente del nombre de paquete debe corresponder a una subcarpeta. Los archivos de código fuente y los compilados pueden almacenarse en distintas ramas de árbol.

Por ejemplo, podemos tener la organización siguiente:



El respeto a este árbol es imperativo. Si el archivo de una clase que pertenece a un paquete no está ubicado en la carpeta correcta, la máquina virtual, encargada de ejecutar la aplicación, será incapaz de localizar esta clase. Para localizar una clase, la máquina virtual utiliza la variable de entorno `CLASSPATH` para construir la ruta de acceso al archivo de esta clase.

Concatena la ruta contenida en esta variable con la ruta correspondiente al paquete donde se encuentra la clase. Si se indican varias carpetas en la variable `CLASSPATH`, deben ir separadas por puntos y comas en un entorno Windows y por comas en un entorno Unix. La carpeta en curso forma parte por defecto de la ruta de búsqueda.

3. Utilización e importación de un paquete

Cuando se utiliza una clase que forma parte de un paquete en una aplicación, se debe usar el nombre completo de la clase (nombre del paquete + nombre de la clase). Esto provoca la escritura de líneas de código relativamente largas y difíciles de leer.

```
es.eni_escuela.conta.Cliente c=new es.eni_escuela.conta.Cliente();
```

Java propone la instrucción `import` para indicar al compilador que algunas clases pueden utilizarse directamente a partir de su nombre sin usar el nombre de paquete. Se debe ubicar esta instrucción al principio del archivo, justo después de una eventual declaración de paquete, pero antes de la definición de la clase. La línea de código anterior se puede acortar muchísimo con la sintaxis siguiente:


```
import es.eni_escuela.conta.Cliente;

...

Cliente c=new Cliente();
```

En contraposición, si se utilizan muchas clases del mismo paquete, la lista de las importaciones se va a volver voluminosa. En este caso, es más lógico importar el paquete completo con el carácter comodín * (asterisco).

El código se convierte en:

```
import es.eni_escuela.conta.*;

...

Cliente c=new Cliente();
```

Con la posibilidad de emplear todas las clases del paquete

`es.eni_escuela.conta` sencillamente haciendo uso del nombre de la clase. Esta solución puede suponer un problema si se importan dos paquetes que contienen clases con nombres idénticos. En este caso, para cada una de estas clases habrá que utilizar su nombre completo.

```
package es.eni_escuela.test;

import es.eni_escuela.conta.Cliente;

public class TestClientPackage {
    public static void main(String[] args) {
        Client clienteContabilidad = new Cliente();
        es.eni_escuela.red.Cliente clienteRed =
            new es.eni_escuela.red.Cliente();
    }
}
```

Esta solución tampoco permite la importación de varios paquetes a la vez. A veces las apariencias engañan: la estructura de paquetes no es jerárquica y, a pesar de lo que podría hacer pensar la estructura de los archivos en el disco, el paquete `es.eni_escuela.conta` no está incluido en el paquete `es.eni_escuela`. La importación `import es.eni_escuela.*`; solo permite el acceso a las clases de este paquete, pero en absoluto a las clases del paquete `es.eni_escuela.conta`. Si este fuera el caso, es fácil imaginar los problemas que podrían conllevar las importaciones siguientes:

```
import com.*;
import es.*;
```

4. Importación de los métodos estáticos

La importación puede también simplificar la escritura del código cuando se utilizan clases que contienen numerosos métodos `static`. Cada llamada a estos métodos debe prefijarse con el nombre de la clase. Esto tiene como consecuencia una peor legibilidad del código.

```
public class Distance
{
    public static double dist(double lat1, double lon1,
                              double lat2, double lon2)
    {
        double deltaLat;
        double abscurv;
        deltaLat = lon2 - lon1;
        abscurv = Math.acos((Math.sin(lat1) * Math.sin(lat2)) +
                             (Math.cos(lat1) * Math.cos(lat2) *
                              Math.cos(deltaLat)));
        return abscurv * 6371598;
    }
}
```

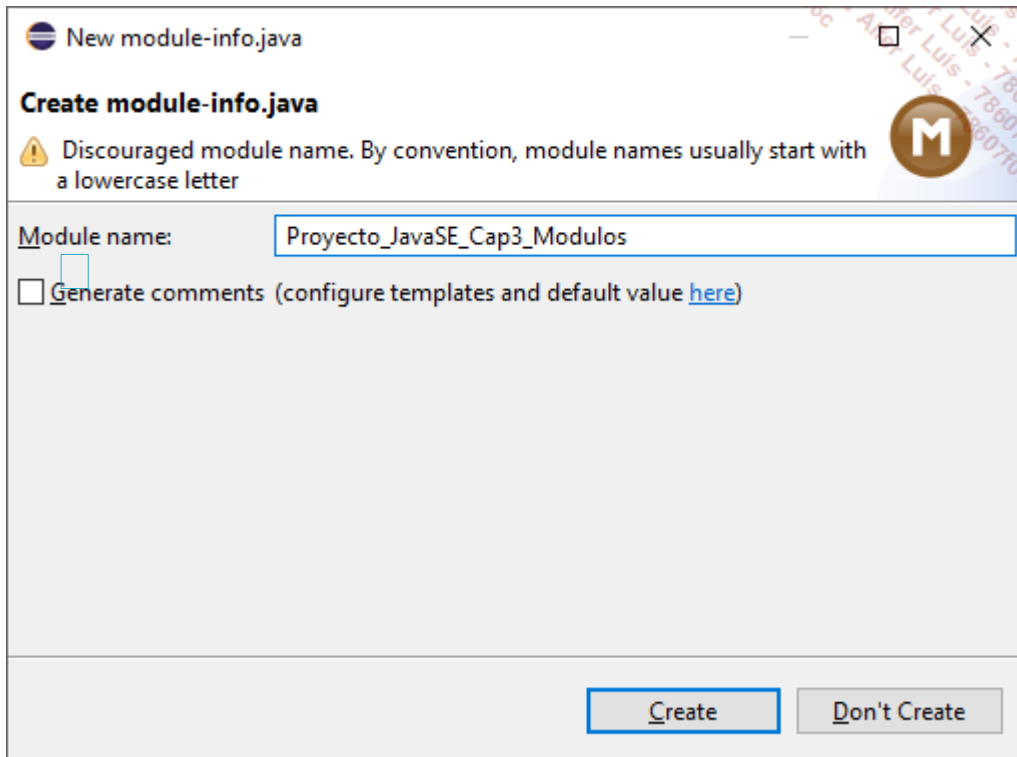
Es posible simplificar este código realizando una importación `static` de la clase `Math` que permita utilizar estos miembros `static` sin prefijarlos por el nombre de la clase.

```
import static java.lang.Math.*;
public class Distance
{
    public static double dist(double lat1,double lon1,
                               double lat2, double lon2)
    {
        double deltaLat;
        double abscurv;
        deltaLat =lon2-lon1;
        abscurv=acos((sin(lat1)*sin(lat2))+
                     (cos(lat1)*cos(lat2)*
                      cos(deltaLat)));
        return abscurv;/*6371598;
    }
}
```

Los módulos

1. Instalación

Al crear un proyecto de Java con Eclipse, aparece la siguiente ventana para solicitar la creación de un archivo llamado **module-info.java**:



Este paso está vinculado a la aparición de la noción de módulo con Java 9. Haga clic en **Create** y se crea un archivo `module-info.java` en el directorio raíz, que contiene las fuentes. Su contenido es el siguiente:

```
module Proyecto_JavaSE_Cap3_Modulos {  
}
```



2. Presentación

Desde Java 9, la API de Java se ha modularizado. El objetivo es múltiple. La principal ventaja de la modularización de la API es la ganancia en el rendimiento durante la ejecución del programa. De hecho, la modularización permite que la máquina virtual cargue solo los módulos necesarios para el correcto funcionamiento de los programas que ejecuta.

Esto puede parecer superfluo con la capacidad de cálculo de las máquinas actuales, pero, si miramos el Internet de las cosas (IOT - *Internet Of Things*), esto tiene sentido.

Otros intereses que se pueden identificar son:

- Una plataforma modular es más fácil de mantener y, por lo tanto, escalable.
- Gestión más precisa de los elementos disponibles. De hecho, hasta ahora, tan pronto como era necesaria una clase más allá del paquete en el que se declaraba, era preciso declararla pública. La consecuencia fue que estaba disponible en todos los niveles. La noción de módulo permite hacer una clase «privada» para el proyecto en el que se declara.

3. El JDK es modular

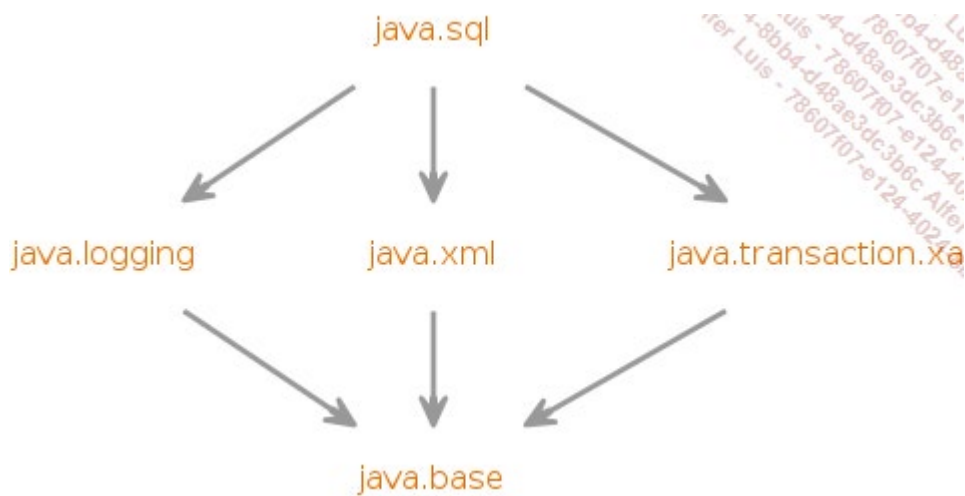
El JDK en sí es modular. Por lo tanto, los miles de clases e interfaces que lo componen se agrupan en módulos. Un módulo es un conjunto de paquetes. La API Java SE está dividida en 22 módulos. El JDK proporciona 37 módulos adicionales.

El primer módulo contiene todas las clases básicas necesarias para el correcto funcionamiento de cualquier programa. Este módulo se llama `java.base`.

Se compone de 51 paquetes. Puede consultar la javadoc en la siguiente dirección: <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html>

Hay una dependencia entre los módulos. Por ejemplo, para que el módulo `java.sql` esté operativo, debe usar los módulos `java.logging`, `java.xml` y `java.transaction.xa`, que requieren el módulo `java.base`.

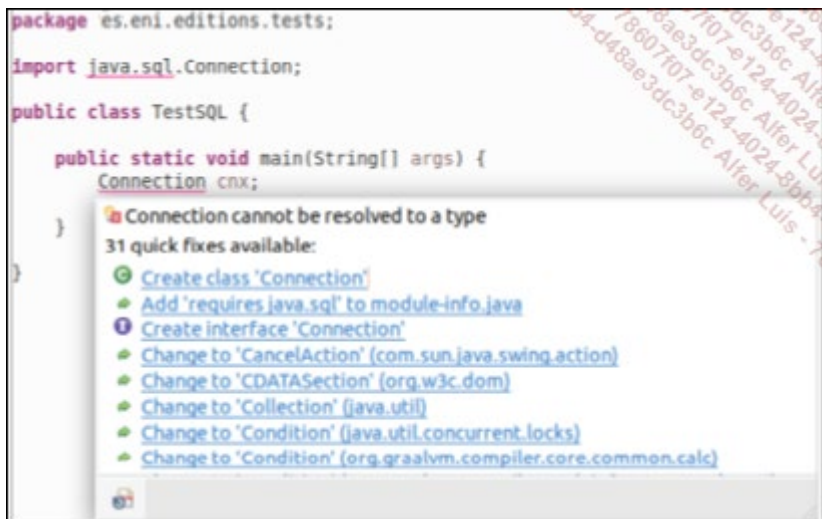
El siguiente diagrama, tomado de la javadoc, ilustra esto:



4. Utilización de los módulos

La utilización de una clase de un módulo necesita su declaración en el archivo `module-info.java` si este archivo existe.

Imagine un programa que necesita conectarse a una base de datos. Para esto, es necesario usar la API JDBC (un capítulo está dedicado a esta API, un poco más adelante en el libro). La interfaz `Connection` (del paquete `java.sql`) es esencial. Usarla en un programa causa el siguiente error:



Resulta que la interfaz no se puede resolver como un tipo, pero existe. El error se produce porque, de acuerdo con el archivo `module-info.java`, esta interfaz no se cargará durante la ejecución del programa, ya no se indica que se utilizará. Para resolver el problema, simplemente declare el uso del módulo que contiene esta interfaz en el archivo `module-info.java`. Es posible hacer clic en el enlace **Add 'requires java.sql' to**

module-info.java o escribir manualmente las siguientes instrucciones en este archivo:

```
module Proyecto_JavaSE_Cap3_Modulos {  
    requires java.sql;  
}
```

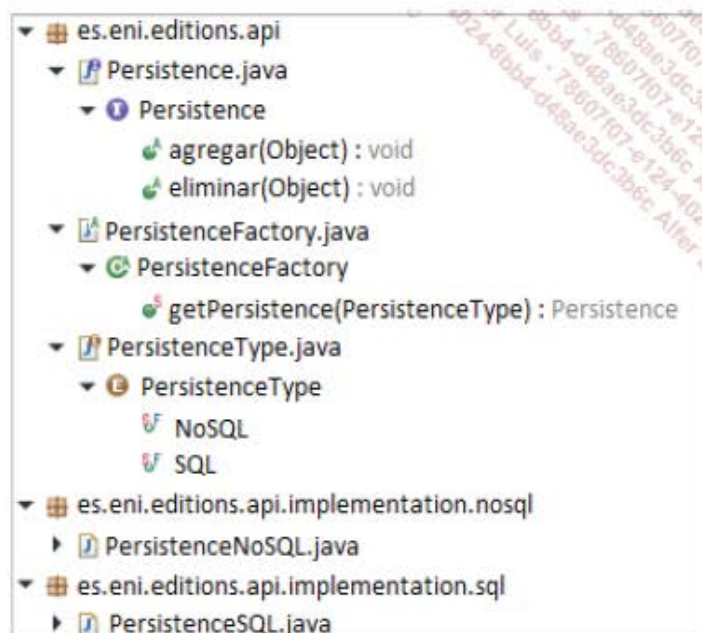
Descargado en: www.detodoprogramacion.org

La palabra clave `requires` le permite enumerar todos los módulos necesarios para el correcto funcionamiento del programa. No es útil declarar el uso del módulo `java.base` porque se lleva a cabo implícitamente.

5. Creación de un nuevo módulo

La creación de una nueva aplicación se puede organizar en varios proyectos. Por ejemplo, es posible crear un proyecto dedicado a la capa de persistencia. Esta capa de persistencia a menudo se puede construir para trabajar con diferentes sistemas de respaldo. La organización clásica que se usa con frecuencia es el patrón **Factory** o uno de sus derivados.

La siguiente organización se basa en este patrón:

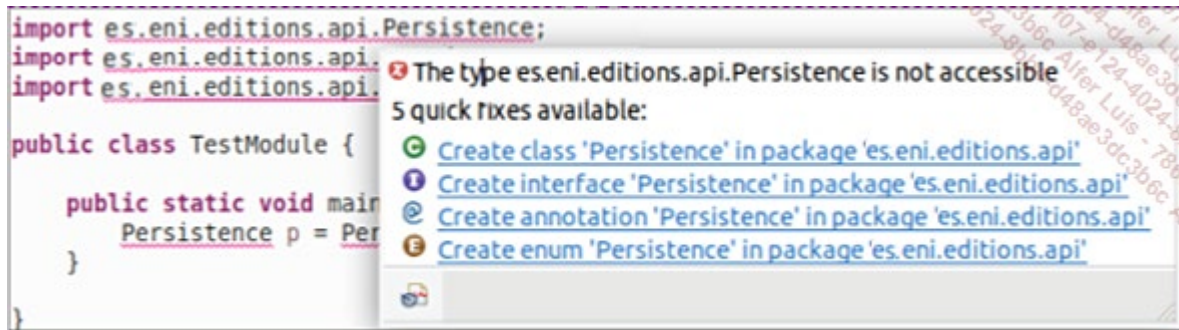


Descargado en: www.detodoprogramacion.org

El paquete `es.eni.editions.api` contiene la interfaz `Persistence`, la factory `PersistenceFactory` y una enumeración `PersistenceType`, que se utiliza como discriminador para seleccionar el mecanismo de persistencia apropiado. Solo estos elementos son útiles para usar en el resto de las capas de la aplicación. Las clases que constituyen las implementaciones (`sql` y `nosql`) no tienen que ser conocidas por las otras

capas, pero las clases son públicas porque deben ser conocidas por la clase `PersistenceFactory`. Si no se utilizara el concepto de módulo, las demás capas tendrían acceso a todos los elementos.

Sin embargo, como el proyecto contiene un archivo `module-info.java`, la escritura del código siguiente no funciona:



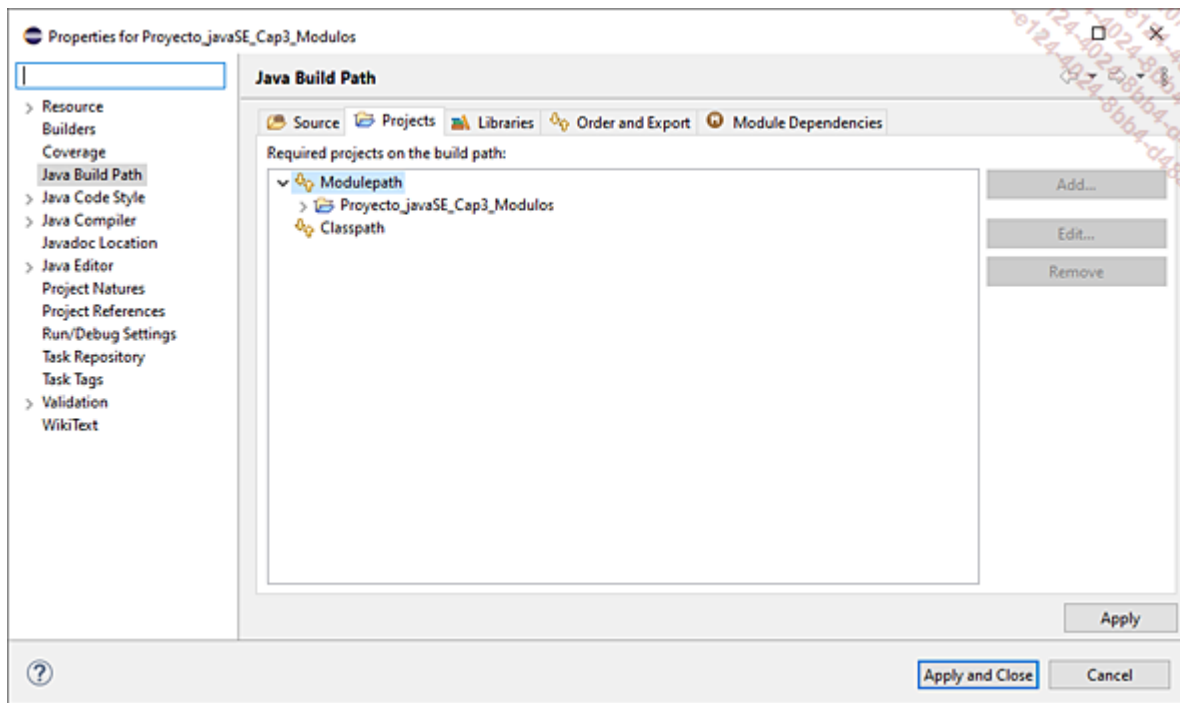
Por defecto, como puede ver en la captura anterior, no se puede acceder a ningún elemento. Simplemente modifique el archivo `module-info.java` para indicar lo que desea exportar y, por lo tanto, hacerlo accesible:

```
module Proyecto_JavaSE_Cap3_Modulos {  
    requires java.sql;  
    exports es.eni.editions.api;  
}
```

La palabra clave `exports` permite indicar los paquetes que queremos hacer accesibles.

☐ Para hacer referencia a un proyecto en otro proyecto, debe realizar las siguientes operaciones:

- ☐ Haga clic con el botón derecho del ratón en el proyecto que debe referenciar a otro proyecto y pulse en el menú **Build Path - Configure Build Path...**
- ☐ Vaya a la pestaña **Projects**.



- Haga clic en el botón **Add...** y seleccione el proyecto o los proyectos que hay que referenciar.

La gestión de los errores

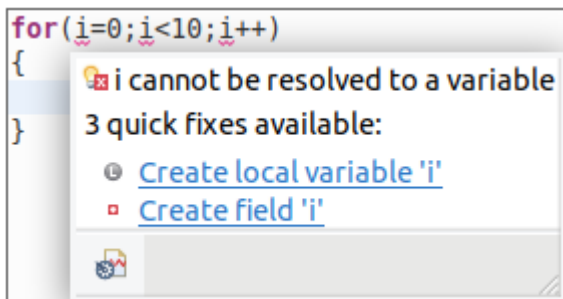
¡La vida de un desarrollador no es del todo fácil! Los errores son una de las fuentes principales de estrés. De hecho, si nos fijamos bien, podemos clasificar estos errores que nos arruinan la vida en tres categorías. Veremos cada una de ellas, así como las soluciones existentes para resolverlos.

1. Los diferentes tipos de error

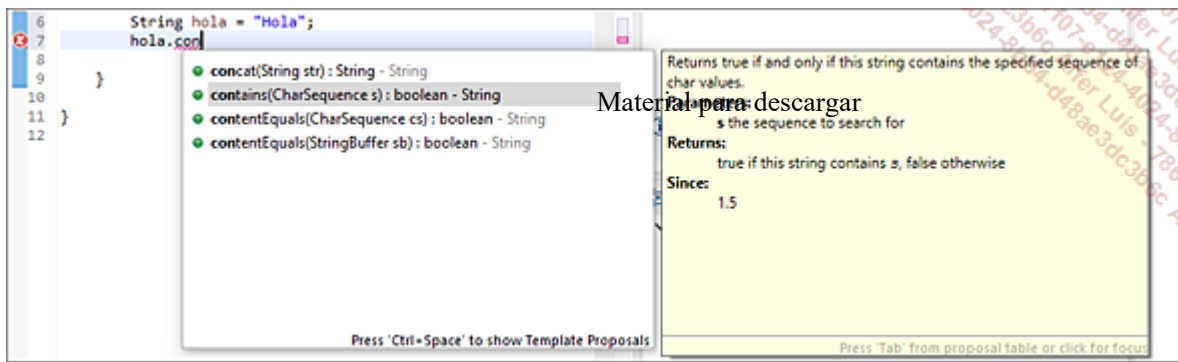
a. Los errores de sintaxis

Este tipo de errores se produce en tiempo de compilación cuando el desarrollador comete un error tipográfico. Son habituales con las herramientas de desarrollo, en las cuales el editor de código y el compilador son dos entidades separadas, y menos frecuentes en entornos de desarrollo integrado (Eclipse, NetBeans, Jbuilder...). La mayoría de estos entornos proporcionan un análisis sintáctico al mismo tiempo que se escribe el código.

Si se detecta un error de sintaxis, el entorno propone soluciones que permiten corregir este error.



Además, las «faltas de ortografía» en los nombres de campos o métodos se eliminan fácilmente mediante las funcionalidades disponibles en estos entornos.



b. Los errores de ejecución

Estos errores aparecen después de la compilación, cuando lanzamos la ejecución de la aplicación. La sintaxis del código es correcta, pero el entorno de la aplicación no permite ejecutar alguna instrucción empleada en la aplicación. Esto sucede, por ejemplo, cuando intenta abrir un archivo que no existe en el disco de su máquina. Sin duda, obtendrá un mensaje de este tipo.

```
Exception in thread "main" java.io.FileNotFoundException: ArchivoDesconocido.txt (No hay ningún archivo o carpeta de este tipo)
at java.base/java.io.FileInputStream.open0(Native Method)
at java.base/java.io.FileInputStream.open(FileStream.java:219)
at java.base/java.io.FileInputStream.<init>(FileStream.java:157)
at java.base/java.io.FileInputStream.<init>(FileStream.java:112)
at es.eni.editions.herencia.TestCliente.main(TestCliente.java:25)
```

¡No es un mensaje muy simpático para el usuario!

Afortunadamente, Java permite que nos recuperemos de este tipo de errores y evita, así, la visualización de este mensaje preocupante. Vamos a detallar esto más adelante en este capítulo.

c. Los errores de lógica

Son los peores enemigos de los desarrolladores. Todo se compila sin problema, todo se ejecuta sin errores, y sin embargo ¡¡¡no funciona como estaba previsto!!!

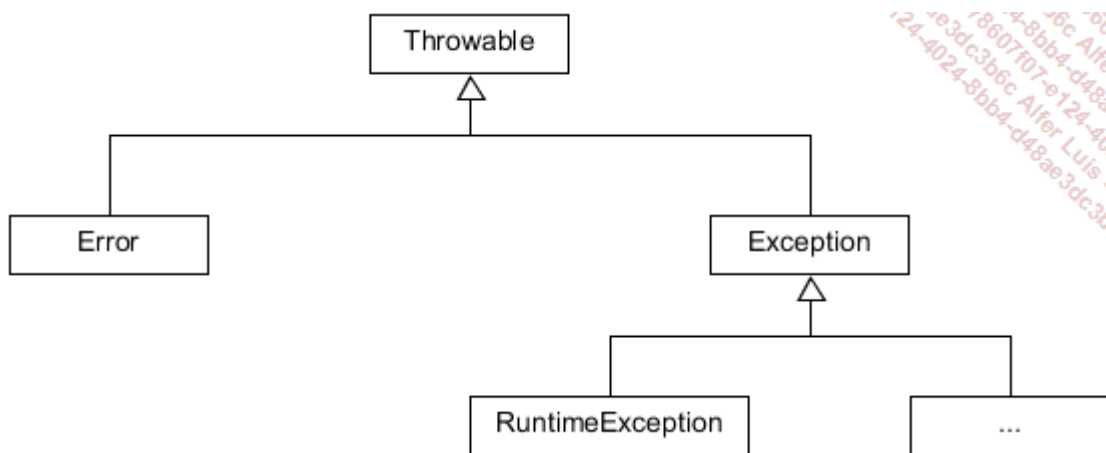
En este caso, hay que revisar la lógica de funcionamiento de la aplicación. Las herramientas de depuración permiten seguir el desarrollo de la aplicación, situar puntos de interrupción, visualizar el contenido de las variables, etc.

Estas herramientas no sustituyen, sin embargo, una buena dosis de reflexión (y a veces algunas pastillas de aspirina).

2. La representación orientada a objetos de los errores

Cuando se produce un error durante la ejecución de un programa, se crea un objeto para representar el error que se acaba de producirse. Este objeto contiene numerosa información relativa al error ocurrido en la aplicación, así como el estado de la aplicación en el momento de la aparición del error. A continuación, se transmite este objeto a la máquina virtual. Esto activa la excepción. Entonces, la máquina virtual debe buscar una solución para resolverla. Para ello, explora los diferentes métodos llamados para alcanzar la ubicación donde se produjo el error. En estos distintos métodos, la máquina busca un gestor de excepciones capaz de tratar el problema. La búsqueda empieza con el método en el cual se activó el error y, a continuación, sube hasta el método `main` de la aplicación, si es necesario. Cuando se localiza un gestor de excepciones adecuado, se le transmite el objeto para que se encargue de su tratamiento. Si la búsqueda no da resultado, la aplicación se detiene.

Las excepciones se suelen clasificar en tres categorías representadas en el diagrama de clases siguiente:



- Las excepciones verificadas corresponden a una situación anormal durante el funcionamiento de la aplicación. Esta situación suele estar relacionada con algún elemento externo a la aplicación, como por ejemplo una conexión hacia una base de datos o la lectura de un archivo. Estas excepciones se representan mediante instancias de la clase `Exception` o una de sus subclases. Deben tratarse, obligatoriamente, dentro de un bloque `try catch` o propagarse al código que la invoca mediante la palabra clave `throws` declarada en la firma de la función.
- Los errores corresponden a condiciones excepcionales exteriores a la aplicación que esta última no puede prever. Estas excepciones se representan mediante una instancia

de la clase `Error` o una de sus subclases. Estas excepciones no tienen por qué tratarse. Cabe precisar que, cuando se producen, el funcionamiento de la aplicación se ve, por lo general, comprometido. Si se gestionan, la mayor parte del tiempo el único procesamiento consiste en mostrar un mensaje algo menos alarmante al usuario del que le propone por defecto la máquina virtual de Java. En prácticamente todos los casos es inevitable que se detenga la aplicación.

- Los errores relacionados con un uso incorrecto de alguna funcionalidad del lenguaje, o un error de lógica en el diseño de la aplicación. El error más frecuente que podrá encontrar al principio con Java será sin duda la excepción `NullPointerException`, que se produce cuando se intenta utilizar una variable no inicializada. Estas excepciones se representan mediante una instancia de la clase `RuntimeException` o una de sus subclases. Estas excepciones se pueden procesar dentro de bloques `try catch`. No es necesario hacerlo cada vez.

Por ejemplo, para una excepción de tipo `NullPointerException` es preferible analizar el código y modificarlo a fin de evitar que aparezcan. Para otras excepciones, es necesario leer la javadoc para comprender por qué ocurre y determinar si es apropiado administrarlo.

3. El tratamiento de las excepciones

La gestión de las excepciones ofrece la posibilidad de proteger un bloque de código contra las excepciones que podrían producirse en él. El código «peligroso» debe ubicarse dentro un bloque `try`. Si se produce una excepción dentro de este bloque de código, el bloque o los bloques de código `catch` son examinados. Si alguno es capaz de tratar la excepción, se ejecuta el código correspondiente; en caso contrario, la excepción se propaga hacia el método que llama para eventualmente ser recuperada por un bloque `try` de mayor nivel. Una instrucción `finally` permite marcar un grupo de instrucciones que se ejecutarán, ya sea a la salida del bloque `try` si no se produjo ninguna excepción, o a la salida de un bloque `catch` si se produjo alguna una excepción.

Por lo tanto, la sintaxis general es la siguiente:

`try`

```

{
    ...
    Instrucciones peligrosas
    ...
}
catch (Excepción1A | Excepción1B e1)
{
    ...
    código ejecutado si se produce una excepción de tipo
    Excepción1A o Excepción1B
    ...
}
catch (Excepción2 e2)
{
    ...
    código ejecutado si se produce una excepción de tipo Excepción2
    ...
}
finally
{
    ...
    código ejecutado en cualquier caso después de la salida del bloque
try
    o de un bloque catch
    ...
}

```

Esta estructura tiene un funcionamiento muy similar al `switch case` ya estudiado.

Es necesario indicar para cada bloque `catch` el tipo de excepción que debe gestionar.

```

public static void leerFichero(String nombre)
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    try
    {

```

```

        fichero=new FileInputStream(nombre);
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    br=new BufferedReader(new InputStreamReader(fichero));
    try
    {
        línea=br.readLine();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
    while (línea!=null)
    {
        System.out.println(línea);
        try
        {
            línea=br.readLine();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}

```

En el ejemplo anterior, cada instrucción susceptible de producir una excepción está protegida por su propio bloque `try`. Esta solución presenta la ventaja de ser extremadamente precisa para la gestión de las excepciones en detrimento de la legibilidad del código. Una solución más sencilla consiste en agrupar varias instrucciones en un mismo bloque `try`. Podríamos codificar el ejemplo anterior de la manera siguiente:

```

public static void leerFichero(String nombre)
{
    FileInputStream fichero=null;

```



```

BufferedReader br=null;
String línea=null;
try
{
    fichero=new FileInputStream(nombre);
    br=new BufferedReader(new InputStreamReader(fichero));
    línea=br.readLine();
    while (línea!=null)
    {
        System.out.println(línea);
        línea=br.readLine();
    }
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
catch (IOException e)
{
    e.printStackTrace();
}
}

```

El código es más legible; sin embargo, perdemos precisión ya que resulta complicado determinar qué instrucción provocó la excepción.

También hay que tener cuidado con el orden de los bloques `catch` y organizarlos siempre desde el más preciso hasta el más general. Las excepciones, al ser clases, pueden tener relaciones de herencia. Si se prevé un bloque `catch` para gestionar un tipo particular de excepción, este puede también gestionar todos los tipos de excepciones que heredan de ella. Es el caso en nuestro ejemplo, ya que la clase `FileNotFoundException` hereda de la clase `IOException`. El compilador detectará la situación y generará un error si modificamos nuestro código de la manera siguiente (inversión de los bloques `catch`):

```

public static void leerFichero(String nombre)
{
    FileInputStream fichero=null;

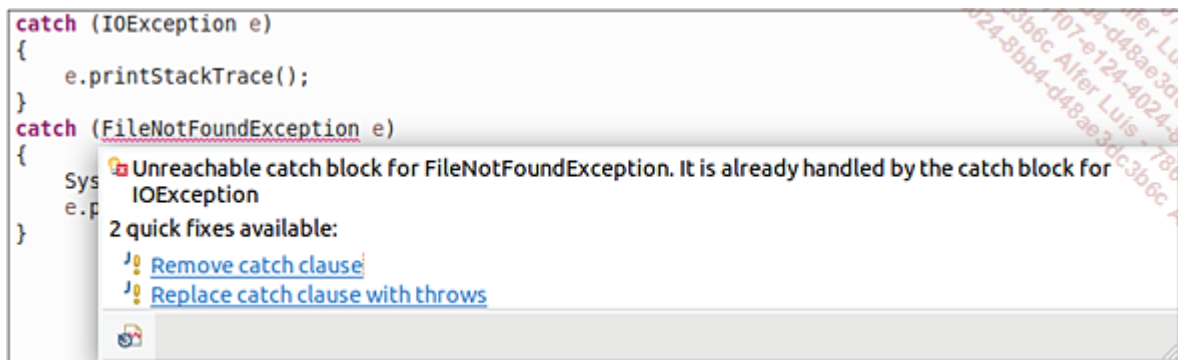
```

```

BufferedReader br=null;
String línea=null;
try
{
    fichero=new FileInputStream(nombre);
    br=new BufferedReader(new InputStreamReader(fichero));
    línea=br.readLine();
    while (línea!=null)
    {
        System.out.println(línea);
        línea=br.readLine();
    }
}
catch (IOException e)
{
    e.printStackTrace();
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
}

```

El mensaje que se muestra es el siguiente:



El código puede ser todavía más preciso si se indica que un mismo bloque `catch` debe gestionar varios tipos de excepciones. Los distintos tipos de excepciones que puede procesar un bloque `catch` deben indicarse en la declaración separados mediante el carácter `|`.

```

public void leerFichero2(String nombre)
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    double suma=0;

    try
    {
        fichero=new FileInputStream(nombre);
        br=new BufferedReader(new InputStreamReader(fichero));
        línea=br.readLine();
        while (línea!=null)
        {
            System.out.println(línea);
            línea=br.readLine();
            suma=suma+Double.parseDouble(línea);
        }
        System.out.println("total:"+suma);
    }
    catch (IOException | NumberFormatException e)
    {
        e.printStackTrace();
    }
}

```

4. Las excepciones asociadas a recursos

Numerosas aplicaciones necesitan con frecuencia acceder a recursos externos. Los archivos y las bases de datos son sin duda los ejemplos más comunes. El uso de estos recursos comienza por una operación de apertura, después viene la explotación del recurso y se finaliza con el cierre del recurso. A menudo, los métodos que permiten explotar estos recursos son susceptibles de provocar numerosas excepciones y de hecho se sitúan en una estructura de tipo `try-catch`. También puede verse aquí el cierre del recurso al final de la ejecución de las instrucciones que contiene. Los recursos deben declararse e instanciarse entre paréntesis tras la palabra clave `try`. Si el bloque `try` contiene varias

declaraciones, estas deben estar separadas por un punto y coma.

Al finalizar la ejecución del bloque `try`, el método `close` se invoca sobre cada recurso declarado a nivel de la palabra clave `try`. Esta llamada se realiza siempre antes de ejecutar un bloque `catch` o del bloque `finally`. Para asegurar que este mecanismo funciona, las clases correspondientes a los recursos utilizados deben implementar la interfaz `Closeable` o `AutoCloseable`. Ambas interfaces exigen la existencia del método `close` en la clase del recurso utilizado en el bloque `try`.

En el ejemplo que aparece a continuación, el objeto `BufferedReader` se cierra automáticamente tras la ejecución del bloque `try`.

```
String response="";
try (BufferedReader br=new BufferedReader(new
                                   InputStreamReader(System.in)))
{
    while (!respuesta.equals("fin"))
    {
        ...
        ...
        response=br.readLine();
    }
}
catch(IOException e)
{
    e.printStackTrace();
}
```

5. Utilización de las excepciones

El código de cada bloque `catch` puede obtener más información sobre la excepción que debe tratar utilizando los métodos disponibles en la clase correspondiente a la excepción. Los métodos siguientes son los más útiles para obtener información adicional sobre la excepción.

- `getMessage`: permite obtener el mensaje de error asociado a la excepción.

- `getCause`: permite obtener la excepción inicial si se utiliza la traza de la excepción.
- `getStackTrace`: permite obtener una pila de `StackTraceElement` de la cual cada elemento representa uno de los métodos invocados hasta el método donde se trata la excepción. Para cada uno de ellos, obtenemos la información siguiente:
 - El nombre de la clase donde se encuentra el método: `getClassName`.
 - El nombre del archivo donde se encuentra esta clase: `getFileName`.
 - El número de línea donde se activó la excepción: `getLineNumber`.
 - El nombre del método: `getMethodName`.

Se puede utilizar esta información para generar archivos históricos de funcionamiento de la aplicación. He aquí un ejemplo que permite grabar esta información en un archivo de texto.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.GregorianCalendar;

public class LecturaFichero
{
    public static void main(String args[])
    {
        try
        {
            leerFichero("informe.txt");
        }
        catch (NoFuncionaExcepcion e)
        {
            FileWriter log;
            BufferedWriter br;
```

```

try
{
    log=new FileWriter("histórico.txt",true);
    br=new BufferedWriter(log);
    br.write("----->"+
            new GregorianCalendar()
            .getTime()+" <-----\r\n");
    br.write("error: " + e.getMessage()+"\r\n");
    for (int i=0;i<e.getStackTrace().length;i++)
    {
        br.write("en el fichero "+
                e.getStackTrace()[i].getFileName());
        br.write(" a la línea "+
                e.getStackTrace()[i].getLineNumber());
        br.write(" en el método "+
                e.getStackTrace()[i].getMethodName());
        br.write(" de la clase "+
                e.getStackTrace()[i].getClassName()+
                "\r\n");
    }
    br.close();
    log.close();
}
catch (IOException ex)
{
    System.out.println("error en la aplicación");
}
}

public static void leerFichero(String nombre)
    throws NoFuncionaExcepcion
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    try
    {

```

```

        fichero=new FileInputStream(nombre);
        br=new BufferedReader(new InputStreamReader(fichero));
        línea=br.readLine();
        while (línea!=null)
        {
            System.out.println(línea);
            línea=br.readLine();
        }
    }
    catch (FileNotFoundException e)
    {
        throw new NoFuncionaExcepcion("el fichero no existe",e);
    }
    catch (IOException e)
    {
        throw new NoFuncionaExcepcion("error de lectura del
fichero",e);
    }
}
}

```

Para más información sobre la excepción `NoFuncionaExcepcion`, vaya a la sección siguiente.

6. Creación y activación de excepciones

Las excepciones son, ante todo, clases; por lo tanto, es posible crear nuestras propias excepciones al heredar de una de las numerosas clases de excepción ya disponibles. Para respetar las convenciones, se aconseja terminar el nombre de la clase con el término `Excepcion`. Podemos escribir, por ejemplo, el código siguiente:

```

public class NoFuncionaExcepcion extends Exception
{
    public NoFuncionaExcepcion()
    {
        super();
    }
}

```

```

public NoFuncionaExcepcion(String message)
{
    super(message);
}
public NoFuncionaExcepcion(Throwable cause)
{
    super(cause);
}
public NoFuncionaExcepcion(String message, Throwable cause)
{
    super(message, cause);
}
}

```

- ☐ Se desaconseja encarecidamente la sobrecarga de los constructores de la clase base para conservar la coherencia entre las clases de excepción.

Esta clase puede utilizarse, a continuación, para generar una excepción personalizada. Para activar una excepción, hay que crear previamente una instancia de la clase correspondiente y, seguidamente, generar la excepción mediante la palabra clave `throw`. La generación de una excepción en una función con la palabra clave `throw` provoca la salida inmediata de la función. El siguiente código genera una excepción personalizada en los bloques `catch`.

```

public static void leerFichero2(String nombre) throws
NoFuncionaExcepcion
{
    FileInputStream fichero=null;
    BufferedReader br=null;
    String línea=null;
    try
    {
        fichero=new FileInputStream(nombre);
        br=new BufferedReader(new InputStreamReader(fichero));
        línea=br.readLine();
        while (línea!=null)

```



```

        {
            System.out.println(línea);
            línea=br.readLine();
        }
    }
    catch (FileNotFoundException e)
    {
        throw new NoFuncionaExcepcion("el fichero no existe",e);
    }
    catch (IOException e)
    {
        throw new NoFuncionaExcepcion("error de lectura del fichero",e);
    }
}

```

Cuando una función es susceptible de generar una excepción, debería indicarse en la firma de esta función con la palabra clave `throws` seguida de la lista de las excepciones que puede generar, salvo si la excepción deriva de la clase `RuntimeException`. Cuando, más tarde, se use esta función en otra, deberá tener en cuenta obligatoriamente esta o estas posibles excepciones. Tendrá que gestionar la excepción con un bloque `try . . . catch` o propagarla añadiendo la palabra clave `throws` a la declaración de la función. Sin embargo, hay que ser prudente y no propagar las excepciones más allá del método `main`, ya que, en este caso, es la máquina virtual de Java quien las recupera y detiene la aplicación de forma brusca.

Los genéricos

1. Presentación

Los tipos genéricos son elementos de un programa que se adaptan automáticamente para lograr la misma funcionalidad sobre diferentes tipos de datos. Cuando crea un elemento genérico, no necesita diseñar una versión diferente para cada tipo de datos con los que desea lograr la funcionalidad. Para hacer una analogía con un objeto actual, tomaremos el ejemplo de un destornillador. Dependiendo del tipo de tornillo que tenga que usar, puede tomar un destornillador específico para este tipo de tornillo (plano, en cruz, etc.). Una técnica utilizada frecuentemente por un técnico experimentado es adquirir un destornillador universal con múltiples puntas. Dependiendo del tipo de tornillo, elige la que mejor se adapte. El resultado final es el mismo que si tuviera una variedad de destornilladores diferentes: puede atornillar y desatornillar.

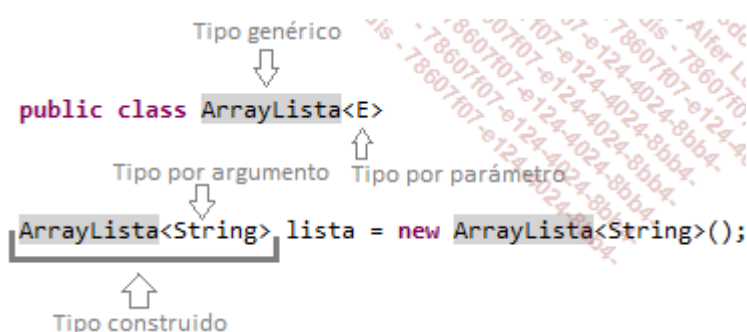
Cuando usa un tipo genérico, lo configura con un tipo de datos. Esto permite que el código se adapte automáticamente y realice la misma acción independientemente del tipo de datos. Una alternativa podría ser el uso del tipo universal `Object`. El uso de tipos genéricos tiene varias ventajas respecto a esta solución:

- Requiere la verificación de los tipos de datos en el momento de la compilación y evita las verificaciones inevitables que deben hacerse manualmente con el uso del tipo `Object`.
- Evita las operaciones de conversión del tipo `Object` a un tipo más específico y viceversa.
- La escritura del código se facilita en ciertos entornos de desarrollo, con la visualización automática de todos los miembros disponibles para un tipo particular de datos.

- Promueve la escritura de algoritmos que son independientes de los tipos de datos.

Sin embargo, los tipos genéricos pueden imponer ciertas restricciones sobre el tipo de datos utilizados. Por ejemplo, pueden obligar a que el tipo utilizado implemente una o más interfaces o derive de una clase particular. Es importante comprender algunos términos utilizados con los genéricos.

- Tipo genérico: esta es la definición de una clase, interfaz o función para la que se especifica al menos un tipo de datos durante su declaración.
- Tipo de parámetro: este es el espacio reservado para el tipo de parámetro en la declaración del tipo genérico.
- Tipo de argumento: este es el tipo de datos que sustituye al tipo de parámetro durante la construcción de un tipo a partir de un tipo genérico.
- Restricciones: son las condiciones que usted impone, que limitan el tipo de argumento que puede proporcionar.
- Tipo construido: es la clase, interfaz o función construida a partir de un tipo genérico para el que ha especificado tipos de argumento.



2. Clases genéricas

Una clase que espera un tipo de parámetro se llama clase genérica. Puede generar una clase construida proporcionando a la clase genérica un tipo de argumento para cada uno de sus tipos de parámetro.

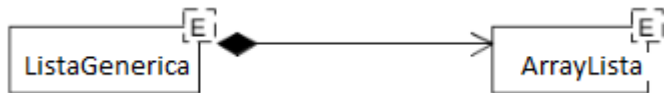
a. Definición de una clase genérica

Puede definir una clase genérica que proporcione las mismas funcionalidades en diferentes tipos de datos. Para hacer esto, debe proporcionar uno o más tipos de parámetros en la definición de la clase. Tomemos como ejemplo una clase capaz de administrar una lista de

elementos con las siguientes funcionalidades.

- Agregar un elemento.
- Eliminar un elemento.
- Moverse al primer elemento.
- Moverse al último elemento.
- Pasar al siguiente elemento.
- Moverse al elemento anterior.
- Obtener el número de elementos.

Este ejemplo se basa en una clase genérica de la API de Java: `ArrayList<T>`. El objetivo es crear una clase que simplifique el uso de esta clase genérica. Por lo tanto, será genérica en sí misma. Aquí está el diagrama de clase para esta asociación:



La sección sobre las colecciones, un poco más adelante en este capítulo, abordará con mayor profundidad el uso de la clase `ArrayList<T>`.

Primero debemos definir la clase como una clase ordinaria.

```
public class ListaGenerica
{
}
```

La transformación de esta clase en una clase genérica se realiza agregando un tipo de parámetro inmediatamente después del nombre de la clase.

```
public class ListaGenerica <E>
{
}
```

Si se requiere más de un tipo de parámetro, deben estar separados por comas en la declaración. Por convención, los tipos de parámetros están representados por un solo carácter en mayúscula, pero este no es siempre el caso.

Si el código de la clase debe realizar operaciones diferentes a las asignaciones, debe

agregar restricciones en el tipo de parámetro. Para hacer esto, agregue la palabra clave `extends`, seguida de la restricción. La restricción puede estar constituida por una clase específica de la que debe heredar el tipo de argumento, o por una o más interfaces que debe implementar. Si se deben aplicar varias restricciones, están separadas por el carácter `&`. En este caso, debe especificar al comienzo de la lista la restricción vinculada a una clase y luego las vinculadas a las interfaces. A continuación hay algunos ejemplos para ilustrar esto.

Clase genérica que requiere que el tipo de argumento herede de la clase `Persona`:

```
public class ListaGenerica <E extends Persona>
{
}
```

Clase genérica que requiere que el tipo de argumento implemente la interfaz `Clasificable`:

```
public class ListaGenerica <E extends Clasificable >
{
}
```

- ☐ Es la misma palabra clave `extends` que se utiliza para una restricción vinculada a una clase o una interfaz.

Clase genérica que requiere que el tipo de argumento herede de la clase `Persona` e implemente las interfaces `Clasificable` y `Cloneable`:

```
public class ListaGenerica <E extends Persona & Clasificable
                                & cloneable>
{
}
```

Si no se especifica ninguna restricción, las únicas operaciones autorizadas serán las admitidas por el tipo `Object`.

En el código de clase, cada miembro que debe ser del tipo argumento debe definirse con el tipo de parámetro, E en nuestro caso. Ahora veamos el código completo de clase.

```
public class ListaGenerica <E>
{
    // para almacenar los elementos de la lista
    private ArrayList<E> lista;
    // puntero de posición en la lista
    private int posicion;
    // número de elementos de la lista
    private int numElementos;
    // constructor con un argumento que permite dimensionar
    // la lista
    public ListaGenerica (int tamaño)
    {
        lista=new ArrayList<E>(tamaño);
    }
    public void agregar(E elemento)
    {
        lista.add(elemento);
        numElementos = numElementos + 1;
    }
    public void insert(E elemento,int indice)
    {
        // se verifica si el índice no es superior al número
        // de elementos o si el índice no es inferior a 0
        if (indice >= numElementos || indice < 0)
        {
            return;
        }
        lista.add(indice,elemento);
        // actualizamos el número de elementos
        numElementos = numElementos + 1;
    }

    public void cambia(E elemento,int indice)
    {
        // se verifica si el índice no es superior al número
```

```

        // de elementos o si el índice no es inferior a 0
        if (indice >= numElementos || indice < 0)
        {
            return;
        }
        lista.set(indice, elemento);
    }
    public void elimina(int indice)
    {
        int i;
        // se verifica si el índice no es superior al número
        // de elementos o si el índice no es inferior a 0
        if (indice >= numElementos || indice < 0)
        {
            return;
        }
        lista.remove(indice);
        // se actualiza el número de elementos
        numElementos = numElementos - 1;
    }
    public E getElement(int j)
    {
        return lista.get(j);
    }
    public int getNumElementos()
    {
        return numElementos;
    }
    public E primero () throws Exception
    {
        if (numElementos == 0)
        {
            throw new Exception("lista vacía");
        }
        // se desplaza el puntero al primer elemento
        posicion = 0;
        return lista.get(0);
    }

```

```
public E ultimo() throws Exception
{
    if (numElementos == 0)
    {
        throw new Exception("lista vacía");
    }
    // movemos el puntero al último elemento
    posicion = numElementos - 1;
    return lista.get(posicion);
}

public E siguiente() throws Exception
{
    if (numElementos == 0)
    {
        throw new Exception("lista vacía");
    }
    // se comprueba si no estamos al final de la lista
    if (posicion == numElementos - 1)
    {
        throw new Exception("no hay elemento siguiente ");
    }
    // movemos el puntero al elemento siguiente
    posicion = posicion + 1;
    return lista.get(posicion);
}

public E anterior() throws Exception
{
    if (numElementos == 0)
    {
        throw new Exception("lista vacía");
    }
    // se comprueba si no estamos en el primer elemento
    if (posicion == 0)
    {
        throw new Exception("no hay elemento anterior");
    }
    // nos movemos al elemento anterior
    posicion = position - 1;
}
```



```
        return lista.get(position);  
    }  
}
```

b. Utilización de una clase genérica

Para poder usar una clase genérica, primero debe generar un tipo construido proporcionando un tipo de argumento para cada uno de sus tipos de parámetros. Luego puede instanciar este tipo por uno de los constructores disponibles. Utilizaremos la clase previamente diseñada para trabajar con una lista de cadenas.

```
ListaGenerica <String> lista = new ListaGenerica <String> (5);
```

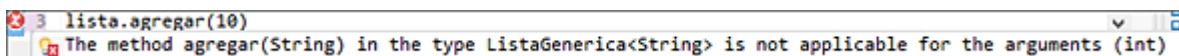
Esta sintaxis se puede simplificar permitiendo que el compilador determine por sí mismo el argumento de tipo que se usará al llamar al constructor. Solo necesita omitir el tipo de argumento entre los caracteres < y >. Es inferencia de tipos.

```
ListaGenerica <String> lista = new ListaGenerica <> (5);
```

Esta declaración crea una instancia de una lista de cinco cadenas. Los métodos de la clase están entonces disponibles.

```
list.agregar("primero");  
list.agregar("segundo");
```

Por supuesto, el compilador verifica que estamos usando nuestra clase correctamente, en particular comprobando los tipos de datos que le confiamos. Si estamos tratando de proporcionar un número entero, aquí está el mensaje que recibimos:



El compilador reemplazó el tipo genérico (E) con el tipo de argumento (String) en la firma de métodos.

He aquí el código de una pequeña aplicación que permite probar el correcto funcionamiento de esta clase genérica:

```
public class PruebaListaGenerica {
```

```

static ListaGenerica<String> lista =
                                new ListaGenerica<String>(5);
public static void main(String[] args) {
    lista.agregar("primero");
    lista.agregar("segundo");
    lista.agregar("tercero");
    lista.agregar("cuarto");
    lista.agregar("quinto");
    menu();
}

public static void menu() {
    char opcion = '\0';
    System.out.println("p (primero) < (anterior)
                        >(siguiente) d (ultimo) f (fin)");
    while (opcion != 'f') {
        try {
            BufferedReader br;
            br = new BufferedReader(
                    new InputStreamReader(System.in));
            opcion = br.readLine().charAt(0);
            switch (opcion) {
                case 'p':
                    System.out.println("el primero: "
                                         + lista.primero ());
                    break;
                case '<':
                    System.out.println("el anterior: "
                                         + lista. anterior ());
                    break;
                case '>':
                    System.out.println("el siguiente: "
                                         + lista.siguiente());
                    break;
                case 'd':
                    System.out.println("el último: "
                                         + lista.ultimo());
                    break;
            }
        }
    }
}

```

```

        }
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
    System.out.println("p (primero) < (anterior)
        >(siguiente) d (ultimo) f (fin)");
}
}
}

```

También podemos verificar que nuestra clase funcione sin problemas si le pedimos que trabaje con personas. Solo existe la declaración de la lista para modificar y la inicialización de su contenido. El método `menu` permanece sin cambios.

```

static ListaGenerica<Persona> lista =
    new ListaGenerica<Persona>(5);

public static void main(String[] args) {

    lista.agregar(
        new Persona("toto2", "apellido2", LocalDate.of(1922, 2, 15)));

    lista.agregar(
        new Persona("toto1", "apellido1", LocalDate.of(1911, 1, 15)));

    lista.agregar(
        new Persona("toto5", "apellido5", LocalDate.of(1955, 5, 15)));

    lista.agregar(
        new Persona("toto3", "apellido3", LocalDate.of(1933, 3, 15)));

    lista.agregar(
        new Persona("toto4", "apellido4", LocalDate.of(1944, 4, 15)));

    menu();
}

```

c. Los métodos genéricos

Una función genérica es un método definido con al menos un tipo de parámetro. Esto permite que el código de llamada especifique el tipo de datos que necesita cada vez que se llama a la función. Sin embargo, dicho método puede usarse sin especificar información para el tipo de argumento. En este caso, el compilador intenta determinar el tipo en función de los argumentos que se pasan al método. Para ilustrar la creación de funciones genéricas, vamos a transformar la función de clasificación en una versión genérica.

```
public static <T extends Clasificable> void clasificar(ListaGenerica<T>
lista)
    throws Exception
{
    int i, j;
    T c;
    for (i = 0; i < lista.getNumElementos() - 1; i++) {
        for (j = i + 1; j < lista.getNumElementos(); j++) {
            if (lista.getElement(j).compare(
                lista.getElement(i)) == Clasificable.INFERIOR)
            {
                c = lista.getElement(j);
                lista.sustituir(lista.getElement(i), j);
                lista.sustituir(c, i);
            } else if (lista.getElement(j).compare(
                lista.getElement(i)) == Clasificable.ERROR)
            {
                throw new Exception(
                    "error durante la ordenación");
            }
        }
    }
}
```

En cuanto a la creación de una clase genérica, el tipo de parámetro se especifica entre los caracteres < y >. Si las restricciones se deben aplicar a los tipos de argumentos, se deben especificar con las mismas reglas que para las clases genéricas.

Llamar a una función genérica es similar a llamar a una función normal, excepto que podemos especificar tipos de argumentos para cada uno de los tipos de parámetros. Esto no es obligatorio; el compilador puede inferir tipos de argumentos en el momento de la compilación.

Por lo tanto, la llamada a la función se puede realizar con las dos sintaxis siguientes:

```
TestListaGenerica.<Persona>ordenar(lista);
```

o

```
TestListaGenerica.ordenar (lista);
```

3. Los genéricos y la herencia

El uso combinado de genéricos y herencia a veces puede causar cierta preocupación para el desarrollador de una aplicación. En la sección dedicada a la herencia, determinamos que con una variable de cierto tipo podríamos hacer referencia a una instancia de clase de este tipo, pero también a una instancia de clase de cualquiera de sus subtipos. Por lo tanto, el siguiente código es perfectamente legal y funciona de forma correcta.

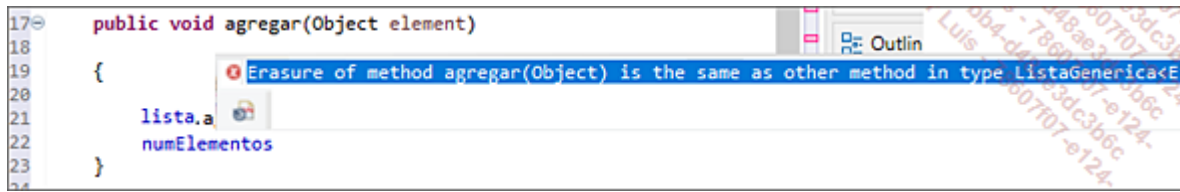
```
Persona p;  
Cliente c;  
c=new Cliente();  
p=c;  
p.setApellido("Dupont");  
p.setNombre("paul");
```

Intentemos la misma experiencia con los genéricos comprobando el código siguiente.

```
ListaGenerica<Cliente> listaCliente =  
    new ListaGenerica<Cliente>(5);  
ListaGenerica<Persona> listaPersona = listaCliente;
```

De hecho, es legítimo pensar que, dado que podemos asignar a una variable de tipo `Persona` una referencia a una instancia de una de sus subclases (`Cliente`), la misma

operación es ciertamente posible con un `ListaGenerica<Persona>` y una `ListaGenerica<Cliente>`. Sin embargo, el compilador no ve las cosas de la misma manera.



De hecho, esta limitación está vinculada al mecanismo de borrado de tipo (*type erasure*) utilizado por el compilador. Su objetivo principal es hacer que el código compilado sea compatible con versiones anteriores de máquinas virtuales de Java.

De hecho, durante la compilación de una clase genérica, el compilador reemplaza el tipo de parámetro por el tipo `Object` o por un tipo correspondiente a la restricción colocada en el tipo de parámetro. El compilador procesará el código de la clase `ListaGenerica` de la siguiente manera.

```
import java.util.ArrayList;
public class ListaGenerica <T>
{
    // para almacenar los elementos de la lista
    private ArrayList<T Object> lista;
    // puntero de posición en la lista
    private int posicion;
    // número de elementos de la lista
    private int numElementos;
    // constructor con un argumento que permite
    // dimensionar la lista
    public ListaGenerica (int tamanioa)
    {
        lista=new ArrayList<T Object>(tamanio);
    }
    public void agregar(T Object elemento)
    {
        lista.add(elemento);
        numElementos = numElementos + 1;
    }
    public void insert(T Object elemento,int indice)
```

```

{
    // se comprueba si el índice no es superior al número
    // de elementos o si el índice no es inferior a 0
    if (index >= numElementos || indice < 0)
    {
        return;
    }
    lista.add(indice, elemento);
    // se actualiza el número de elementos
    numElementos = numElementos + 1;
}

public T Object primero () throws Exception
{
    if (numElementos == 0)
    {
        throw new Exception("lista vacía");
    }
    // se mueve el puntero al primer elemento
    posicion = 0;
    return lista.get(0);
}

public T Object ultimo() throws Exception
{
    if (numElementos == 0)
    {
        throw new Exception("lista vacía");
    }
    // se mueve el puntero al último elemento
    posicion = numElementos - 1;
    return lista.get(posicion);
}

...
...
}

```

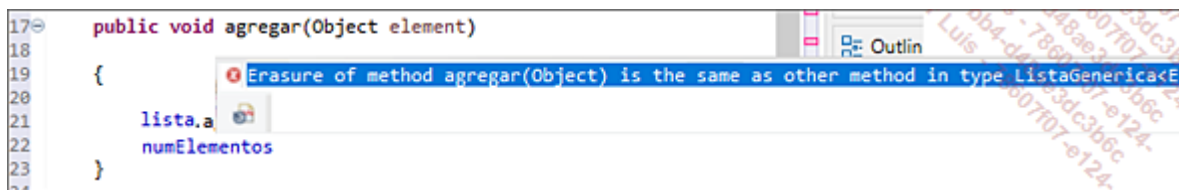
Para convencernos, probamos el siguiente código en la clase ListaGenerica.

```

public void agregar(T elemento)
{
    lista.add(elemento);
    numElementos = numElementos + 1;
}
public void agregar(Object elemento)
{
    lista.add(elemento);
    numElementos = numElementos + 1;
}

```

Durante la compilación, tenemos un mensaje de error que indica que el método `agregar (Object)` se define dos veces en la clase:



Otra operación realizada por el compilador es efectuar una conversión para los valores devueltos por las funciones de la clase genérica. El siguiente código:

```

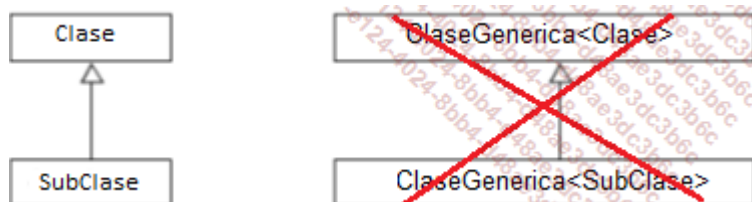
static ListaGenerica<Persona> lista =
                                new ListaGenerica<Persona>(5);
public static void main(String[] args) throws Exception
{
    lista.agregar(
        new Persona("toto2", "apellido2", LocalDate.of(1922, 2, 15)));
    lista.agregar(
        new Persona("toto1", "apellido1", LocalDate.of(1911, 1, 15)));
    lista.agregar(
        new Persona("toto5", "apellido5", LocalDate.of(1955, 5, 15)));
    lista.agregar(
        new Persona("toto3", "apellido3", LocalDate.of(1933, 3, 15)));
    lista.agregar(
        new Persona("toto4", "apellido4", LocalDate.of(1944, 4, 15)));
    Persona p;
    p=lista.getElement(0);
}

```


se compilará de esta manera:

```
static ListaGenerica<Persona> lista =  
                                new ListaGenerica<Persona>(5);  
public static void main(String[] args) throws Exception  
{  
    lista.agregar(  
        new Persona("toto2", "apellido2", LocalDate.of(1922, 2, 15)));  
    lista.agregar(  
        new Persona("toto1", "apellido1", LocalDate.of(1911, 1, 15)));  
    lista.agregar(  
        new Persona("toto5", "apellido5", LocalDate.of(1955, 5, 15)));  
    lista.agregar(  
        new Persona("toto3", "apellido3", LocalDate.of(1933, 3, 15)));  
    lista.agregar(  
        new Persona("toto4", "apellido4", LocalDate.of(1944, 4, 15)));  
    Persona p;  
    p=(Persona)lista.getElement(0);  
}
```

Por lo tanto, lo que debemos retener de estos experimentos es que una clase genérica construida teniendo como tipo de parámetro una clase arbitraria no es la subclase de una clase genérica construida con un tipo que es el supertipo de esta clase arbitraria.



Esta limitación a veces puede ser problemática. Si queremos crear una función que acepte como parámetro una `ListaGenerica` construida a partir de cualquier tipo de datos, nuestra primera intuición es escribir el siguiente código:

```
public static void mostrar(ListaGenerica<Object> lista)  
{  
  
}
```

Por desgracia, nuestra primera intuición no es buena porque, con esta sintaxis, la función de visualización acepta como parámetro solo instancias de `ListaGenerica` de `Object`.

Para representar el hecho de que la función de visualización acepta como parámetro una instancia `ListaGenerica` construida a partir de cualquier tipo, tenemos que usar el comodín `?` en la definición de la función de visualización. Por lo tanto, toma la siguiente forma:

```
public static void mostrar(ListaGenerica<?> lista)
{
}
}
```

Con esta sintaxis, podemos utilizar la función de visualización usando como parámetro una instancia de `ListaGenerica` construida a partir de cualquier clase. Por otro lado, en el código de función solo podremos usar los métodos de la clase `Object` en los elementos presentes en la lista. Podemos agregar restricciones al tipo de argumento de la función poniendo, a continuación del carácter `?`, una restricción definida con la misma sintaxis que para una clase genérica. Para que la función de visualización acepte como parámetro una `ListaGenerica` de `Persona` y cualquiera de sus subclases, debemos usar la siguiente sintaxis:

```
public static void mostrar(
    ListaGenerica<? extends Persona> lista)
{
    lista.primer();
    for (int i=0;i<lista.getNumElementos();i++)
    {
        System.out.println(lista.getElement(i).getNombre());
        System.out.println(lista.getElement(i).getApellido());
        System.out.println("-----");
    }
}
```

Tenga en cuenta que, en este caso, las instancias presentes en la lista serán al menos instancias de la clase `Persona` y se pueden usar como tales de forma segura dentro de la

función.

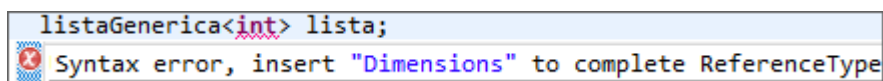
4. Limitaciones de los genéricos

El uso de tipos genéricos impone algunas restricciones. Estas limitaciones están relacionadas con el tipo de argumento utilizado y el uso que se hace de él.

- El tipo de argumento es necesariamente un tipo por referencia. Por ejemplo, es imposible crear una instancia de `ListaGenerica` de tipo `int`.

```
ListaGenerica<int> listaEnteros = new ListaGenerica<int>();
```

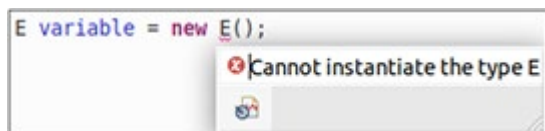
Se muestra el siguiente error:



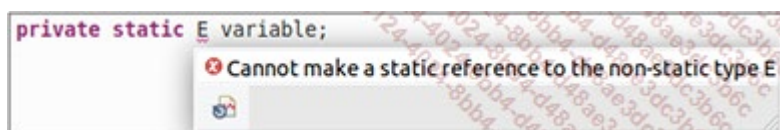
Esta limitación se puede superar mediante el uso de las clases de contenedor correspondientes a tipos simples en su lugar. En nuestro caso, el uso de la clase `Integer` resuelve el problema. Los mecanismos de autoboxing y unboxing hacen que el uso de tipos *wrapper* sea transparente en lugar de tipos simples.

```
ListaGenerica<Integer> lista;
```

- El tipo de parámetro no se puede usar para la creación de una instancia de clase dentro de un tipo genérico. Por ejemplo, el siguiente código está prohibido dentro de la clase `ListaGenerica`:



- El tipo de parámetro tampoco se puede utilizar para declarar variables de clase (`static`) dentro de un tipo genérico.



Como la variable es única para todas las instancias de la clase `ListaGenerica`, su tipo no puede cambiar dependiendo del tipo de argumento utilizado para crear cada una de las instancias. Por ejemplo, con la siguiente línea de código, la variable de prueba sería de tipo `String`.

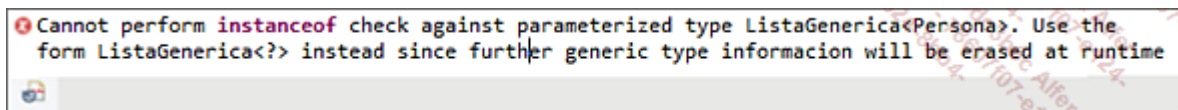
```
ListaGenerica<String> listaCadena=new ListaGenerica<String>(5);
```

Si se crea otra instancia de la clase `ListaGenerica` con un tipo de argumento diferente, la variable de clase tendría que cambiar el tipo, lo que por supuesto es imposible.

- El operador `instanceof` no se puede usar con tipos genéricos. Dado que el mecanismo de borrado de tipo elimina todos los rastros del tipo de argumento, después de la compilación, la siguiente prueba no podía funcionar.

```
if(lista instanceof ListaGenerica<Persona>)  
{  
}
```

Se muestra el siguiente error:



El compilador ofrece la siguiente solución:

```
if(lista instanceof ListaGenerica<?>)  
{  
}
```

Esto nos permite asegurarnos de que la variable `lista` sea del tipo `ListaGenerica`, pero sin dominar el tipo de argumento. Para conocer el tipo de argumento, es necesario manipular la API de *reflection* que permite introspectar el contenido de los objetos. Esto está más allá del alcance de este libro.

Las colecciones

1. Presentación

Las aplicaciones necesitan, con frecuencia, manipular una gran cantidad de información. Existen numerosas estructuras que facilitan la gestión de dicha información. Se agrupan bajo el término «colección». Como ocurre en la vida, existen distintos tipos de colecciones y de coleccionadores. Puede haber personas que lo guardan todo pero que no tienen una organización particular para clasificarlo; otras que se especializan en la colección de ciertos tipos de objetos, o incluso los maníacos que toman todas las precauciones posibles para poder encontrar rápidamente un objeto...



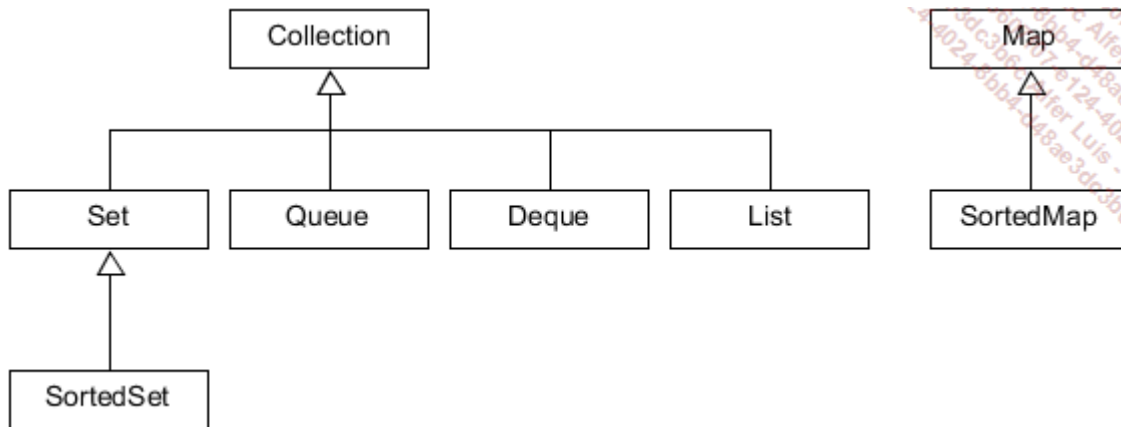
El lenguaje Java proporciona distintas clases que permiten implementar varios modelos de gestión.

La primera solución para gestionar un conjunto de elementos consiste en utilizar tablas. Esta solución se ha descrito en la sección Los arrays del capítulo Fundamentos del lenguaje. Si bien son sencillas de implementar, esta solución no es demasiado flexible. Su principal defecto es el carácter fijo del tamaño de la tabla. Si no hay espacio para almacenar elementos suplementarios, hay que crear una nueva tabla, más grande, y transferirle el contenido de la tabla anterior. Esta solución, muy pesada de implementar, consume además muchos recursos.

El lenguaje Java proporciona una vasta paleta de interfaces y de clases que permiten gestionar fácilmente conjuntos de elementos. Las interfaces describen las funcionalidades disponibles, mientras que las clases implementan y proveen realmente dichas funcionalidades. En función del modelo de gestión que se desee tener sobre los elementos, se utilizará la clase mejor adaptada. No obstante, las mismas funcionalidades básicas deben estar accesibles sea cual sea el modelo de gestión. Para asegurar la presencia

de estas funcionalidades, indispensables en todas las clases, se han agrupado en la interfaz `Collection`, que se utiliza como interfaz básica.

La jerarquía de interfaces se presenta en el siguiente diagrama.



Todas estas interfaces son genéricas, con el objetivo de gestionar conjuntos compuestos por no importa qué tipos de elementos, evitando así las molestas operaciones de cambio de tipo.

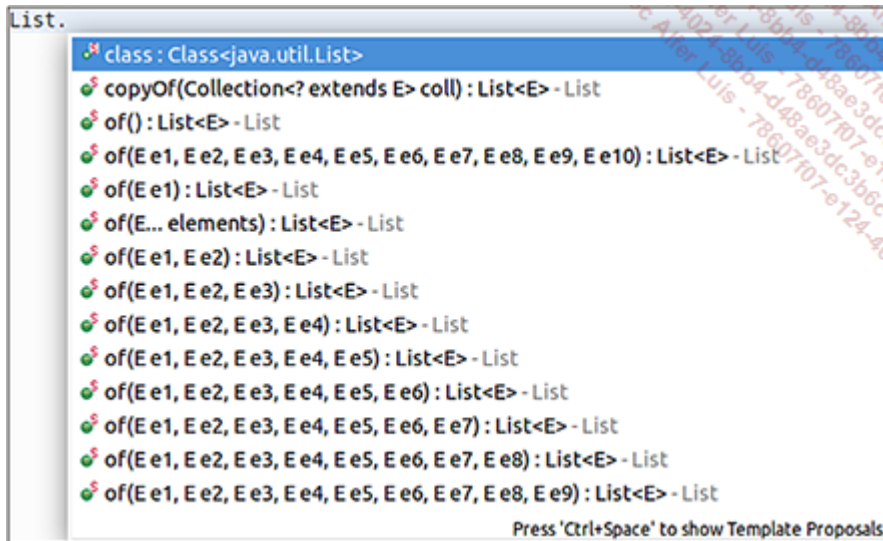
- La interfaz `Collection` es la interfaz básica que proporciona todas las características comunes a los diferentes tipos de colecciones.
- La interfaz `Set` permite administrar colecciones de elementos separados. La interfaz `SortedSet` le permite mantener el orden en los elementos que componen la colección.
- La interfaz `Queue` permite almacenar una lista de elementos antes de que pueda manipularlos. Los elementos generalmente se organizan en FIFO (*First In First Out*). El primer elemento almacenado en la `Queue` será el primer elemento leído. Pero esto no es una obligación.
- La interfaz `Deque` permite, al igual que la interfaz `Queue`, almacenar una lista de elementos antes de que puedan ser manipulados. La diferencia radica en el hecho de que los elementos se pueden organizar en FIFO, pero también en LIFO (*Last In First Out*). Es posible agregar y eliminar un elemento al principio o al final de la lista.
- La interfaz `List` permite administrar colecciones ordenadas de elementos. A diferencia de la interfaz `Set`, acepta elementos duplicados.

Al lado de este grupo de interfaces hay otro grupo con las interfaces `Map` y `SortedMap`. Estas dos interfaces le permiten administrar colecciones de elementos de tipo clave/valor.

Por cada una de estas interfaces existen una o varias clases asociadas: son las que nos van a interesar.

Como dato informativo, desde Java 9, estas interfaces tienen métodos estáticos que permiten obtener colecciones que no se pueden modificar. Estas son colecciones en las que es imposible agregar, eliminar o reemplazar un elemento. Sin embargo, es posible modificar los elementos.

Estos son los métodos disponibles en la interfaz de `List`.



En este libro no se abordarán todas la clases. Para obtener más información, puede consultar el tutorial oficial disponible en la siguiente dirección:

<https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html>

2. La clase `ArrayList`

Esta clase es una implementación de la interfaz `List`. Permite gestionar elementos que se han implementado de manera casi similar a la disponible con una tabla, con un aspecto más dinámico. Tras la creación de la instancia de la clase `ArrayList`, es posible agregar elementos mediante los métodos `add` y `addAll`. Por defecto, ambos métodos agregan los elementos a continuación de los ya existentes. Existe una versión sobrecargada de ambos métodos que permite seleccionar el lugar donde se ubican. En dicha versión, es preciso indicar la posición donde debe realizarse la inserción.

La primera posición tiene índice 0. Si ya existen elementos, simplemente se desplazan.

El acceso a los elementos de la lista se realiza mediante el método `get`, al que se indica simplemente el índice del elemento. A la inversa, el método `set` permite situar un elemento en la posición indicada. No se realiza, en este caso, una inserción, sino que se reemplaza el elemento que se encuentra en dicha posición.

La eliminación de un elemento se realiza mediante el método `remove`, al que se indica o bien el índice o bien el elemento que se desea suprimir. En ambos casos los elementos siguientes se desplazan un paso (no hay jamás una «franja vacía» en un `ArrayList`). Las tres funciones `indexOf`, `lastIndexOf` y `contains` realizan la búsqueda de un elemento en el `ArrayList`. Las dos primeras devuelven el índice donde se ha encontrado el elemento o `-1` si no se ha encontrado ningún elemento. La función `indexOf` comienza buscando desde el primer elemento de la lista; la función `lastIndexOf` comienza la búsqueda por el final de la lista. La función `contains` indica simplemente si el elemento está presente en la lista, sin especificar su posición, si existe.

Es posible recorrer los elementos de la lista con un bucle `for` o utilizando el objeto `Iterator` provisto por el método `iterator` de la clase `ArrayList`. Con ambas soluciones, el recorrido se realiza únicamente desde el primero hasta el último elemento de la lista; además, el contenido de la lista no puede modificarse durante su recorrido. El objeto `ListIterator`, devuelto por el método `listIterator`, permite trabajar con una mayor flexibilidad, pues autoriza desplazamientos en la lista en ambos sentidos y acepta también modificaciones de la lista en curso durante su recorrido.

El siguiente código de ejemplo ilustra las distintas funcionalidades.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class TestArrayList {
    public static void main(String[] args)
    {
        // Declaración de dos variables
```



```

// del tipo de la interfaz deseada
List<Persona> lista1;
List<Persona> lista2;
// creación de las dos instancias
lista1=new ArrayList<Persona>();
lista2=new ArrayList<Persona>();

// creación de las personas para completar la lista
Persona p1,p2,p3,p4,p5;
p1 = new Persona("Wayne", "John",LocalDate.of(1907,5,26));
p2 = new Persona("McQueen", "Steve",LocalDate.of(1930,3,24));
p3 = new Persona("Lennon", "John",LocalDate.of(1940,10,9));
p4 = new Persona("Gibson", "Mel",LocalDate.of(1956,1,3));
p5 = new Persona("Willis", "Bruce",LocalDate.of(1955,3,19));

// agregar cuatro personas a la lista
lista1.add(p1);
lista1.add(p3);
lista1.add(p4);
lista1.add(p5);

// incluir una persona entre p1 y p3
// en la posición 1 de la lista
lista1.add(1, p2);

// agregar contenido de una lista a otra lista
// ambas listas contienen ahora los mismos objetos.
// ¡¡¡ no confundir con lista2=lista1; !!!
lista2.addAll(lista1);

// mostrar el número de elementos de la lista
System.out.println("Hay " + lista1.size() + " persona(s) en la
lista");

System.out.println("Aquí están las personas de la lista 1
(foreach):");
// recorrer la primera lista con el bucle foreach
for (Persona persona : lista1) {

```

```

        System.out.println(persona.getApellido());
    }

    System.out.println("Aquí están las personas de la lista 1
                        (for):");
    // recorrer la primera lista con el bucle for
    for (int index = 0; index < lista1.size(); index++) {
        System.out.println(liste1.get(index).getApellido ());
    }

    System.out.println("Aquí están las personas de la lista 1
                        (Iterator):");
    // recorrer la primera lista de inicio a fin
    Iterator<Persona> it;
    it=lista1.iterator();
    Persona p;
    // mientras haya elementos

    while (it.hasNext()){
        // recuperar el elemento en curso
        p=it.next();
        System.out.println(p.getApellido());
    }

    System.out.println("Aquí están las personas de la lista 1
                        (ListIterator):");

    // recorrer la primera lista desde el final hasta el inicio
    // recuperar un ListIterator posicionado tras
    // el último elemento (el número de elementos de la lista)

    ListIterator<Persona> lit;
    lit=lista1.listIterator(lista1.size());
    // mientras haya elementos
    while (lit.hasPrevious()) {
        // recuperar el elemento en curso
        // volviendo atrás en la lista

```

```

        p=lit.previous();
        System.out.println(p.getApellido());
    }

    // remplazar un elemento de la lista
    lista1.set(2,new Persona("Grant",
"Cary",LocalDate.of(1904,1,18)));

    // mostrar el elemento situado en la tercera posición de la
lista
    System.out.println(lista1.get(2).getApellido());
    // buscar un elemento en la lista
    int posicion;
    posicion=lista1.indexOf(p4);
    if(posicion==-1)
        System.out.println("no se ha encontrado en la lista");
    else
        System.out.println(lista1.get(posicion).getApellido());

    // buscar un elemento inexistente en la lista.
    // John Lennon se ha remplazado por Cary Grant
    // La búsqueda comienza por el final de la lista

    posicion=lista1.lastIndexOf(p3);
    if(posicion==-1)
        System.out.println("no se ha encontrado en la lista");
    else
        System.out.println(lista1.get(posicion).getApellido());
    }
}

```

3. La clase HashSet

Esta clase es una implementación de la interfaz `Set`. Su funcionamiento es prácticamente idéntico al de la clase `ArrayList`. La principal diferencia está ligada a la gestión de duplicados.

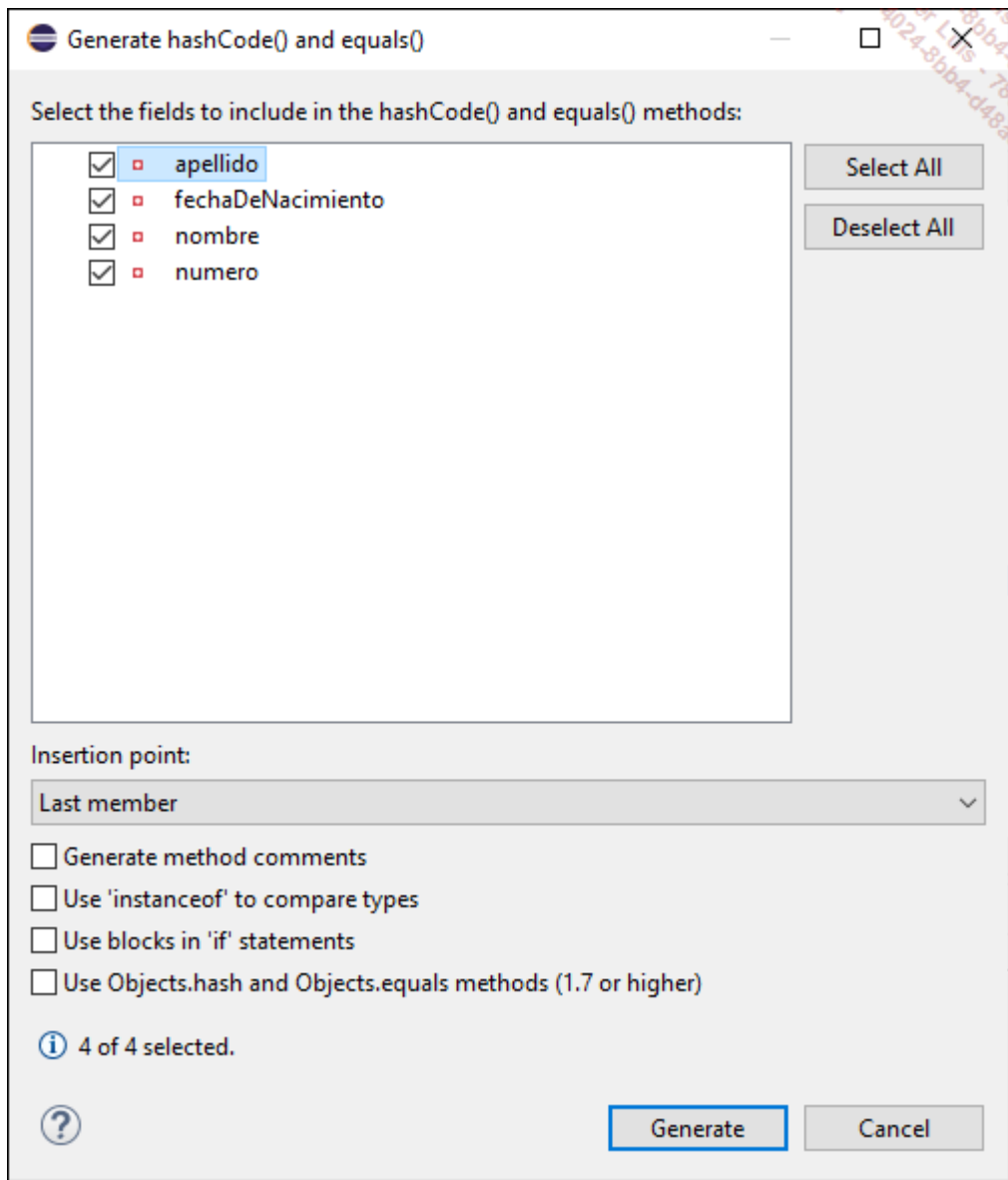
A diferencia de un `ArrayList`, un `HashSet` no permite almacenar dos elementos idénticos. Cuando se agrega un elemento a un `HashSet` con el método `add`, este verifica que no exista un elemento idéntico comparándolo con los elementos que ya están presentes en la lista gracias al método `equals` del elemento. Si se agrega el elemento, el método `add` devuelve el valor booleano `true`, o `false` en caso contrario. Esto nos obliga a redefinir el método `hashCode` de todas las clases para que nos permitan almacenar instancias en un `HashSet`.

Para mantener la coherencia, es indispensable también redefinir el método `hashCode` de la clase que debe agrupar el mismo valor para los objetos considerados iguales, según los criterios del método `equals`.

Para poder almacenar en un `HashSet` instancias de la clase `Persona`, debemos agregar estos dos métodos.

Eclipse ofrece facilidades para agregar estos métodos. Simplemente, haga clic derecho en el archivo fuente de la clase `Persona` y seleccione el siguiente menú: **Source - Generate hashCode() and equals()...**

Aparece el siguiente cuadro de diálogo:



Es suficiente con marcar las variables miembro que se deben evaluar para garantizar que dos objetos de tipo `Persona` sean idénticos. En el presente caso, dado que las cuatro variables miembro están marcadas, dos objetos de tipo `Persona` se considerarán idénticos si, y solo si, todas estas variables son idénticas. Simplemente haga clic en el botón **Generate** para crear automáticamente estos dos métodos.

Ahora es posible probar las funcionalidades de la clase `HashSet`. Son prácticamente las mismas que las de la clase `ArrayList`, salvo por pequeños detalles. En efecto, en un `HashSet` es imposible acceder a un elemento particular, puesto que no existe la noción de índice.

```
public class TestHashSet {  
    public static void main(String[] args)  
    {  
        HashSet<Persona> hash1;
```

```
HashSet<Persona> hash2;  
// creación de las dos instancias  
hash1=new HashSet<Persona>();  
hash2=new HashSet<Persona>();  
  
// creación de las personas para completar el HashSet  
Persona p1,p2,p3,p4,p5;  
p1 = new Persona("Wayne", "John",LocalDate.of(1907,5,26));  
p2 = new Persona("McQueen", "Steve",LocalDate.of(1930,3,24));  
p3 = new Persona("Lennon", "John",LocalDate.of(1940,10,9));  
p4 = new Persona("Gibson", "Mel",LocalDate.of(1956,1,3));  
p5 = new Persona("Willis", "Bruce",LocalDate.of(1955,3,19));  
  
// agregar cuatro personas al HashSet  
hash1.add(p1);  
hash1.add(p3);  
hash1.add(p4);  
hash1.add(p5);  
  
// agregar el contenido de un HashSet a otro HashSet  
// ambos HashSet contienen ahora los mismos  
// objetos.  
// ¡¡¡ no confundir con hash2=hash1; !!!  
hash2.addAll(hash1);  
  
// mostrar el número de elementos del HashSet  
System.out.println("Hay " + hash1.size() + " persona(s)  
en el HashSet");  
  
System.out.println("Aquí están las personas de la lista 1  
(foreach):");  
// recorrido de la primera lista con el bucle foreach  
for (Persona persona : hash1) {  
    System.out.println(persona.getApellido());  
}  
  
System.out.println("Aquí están las personas de la lista 1
```

```

(Iterator):");
    // recorrer el primer HashSet de inicio a fin
    Iterator<Persona> it;
    it=hash1.iterator();
    // mientras queden elementos en el HashSet
    Persona p;
    while (it.hasNext())
    {
        // recuperar el elemento en curso
        p=it.next();
        System.out.println(p.getApellido());
    }
}
}

```

A pesar de esta restricción, los `HashSet` son particularmente eficaces para realizar operaciones sobre conjuntos, como uniones e intersecciones. Antes de probar estas operaciones con `HashSet`, veamos sus definiciones teóricas.

Supongamos que tenemos a nuestra disposición dos conjuntos que contienen personas. Uno contiene cantantes, el otro contiene actores.

- **Noción de subconjunto:** se considera que un conjunto es un subconjunto de otro si todos sus elementos están contenidos en él. Por ejemplo, podemos decir que cantantes es un subconjunto de actores si todos los cantantes son, a su vez, actores.
- **Noción de unión:** la unión de dos conjuntos es el conjunto formado por todos los elementos contenidos en cada uno de ambos conjuntos. Podemos, por ejemplo, definir un conjunto llamado artistas que sea la unión de actores y cantantes.
- **Noción de intersección:** la intersección de dos conjuntos está formada por los elementos contenidos a la vez en el primer y el segundo conjuntos. En nuestro caso contiene aquellas personas que son, a la vez, actores y cantantes.
- **Noción de diferencia:** la diferencia entre dos conjuntos está formada por todos los elementos presentes en uno pero no en el otro. En nuestro caso, representa aquellas personas que son únicamente cantantes y aquellas que son únicamente actores.

Existen cuatro funciones de la clase `HashSet` que permiten traducir muy fácilmente estas nociones. Se trata de la función `containsAll` para la noción de

subconjunto, la función `addAll` para la unión, la función `retainAll` para la intersección y la función `removeAll` para la diferencia.

Estas funciones modifican el contenido del `HashSet` sobre el que se invocan, por lo que resulta prudente realizar una copia y trabajar sobre ella.

El ejemplo de código siguiente ilustra las distintas nociones.

```
import java.time.LocalDate;
import java.util.HashSet;
import java.util.Iterator;

public class TestHashSetConjunto {
    public static void main(String[] args)
    {
        HashSet<Persona> actores;
        HashSet<Persona> cantantes;
        actores=new HashSet<Persona>();
        cantantes=new HashSet<Persona>();

        // creación de las personas para completar la lista
        Persona p1,p2,p3,p4,p5;
        p1 = new Persona("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Persona("McQueen", "Steve",LocalDate.of(1930,3,24));
        p3 = new Persona("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Persona("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Persona("Willis", "Bruce",LocalDate.of(1955,3,19));

        actores.add(p1);
        actores.add(p2);
        actores.add(p4);
        actores.add(p5);

        cantantes.add(p1);
        cantantes.add(p3);

        // comprueba si los cantantes son también actores
        if (actores.containsAll(cantantes))
            System.out.println("todos los cantantes son también
```



```

actores");
else
    System.out.println("algunos cantantes no son actores");
System.out.println("***** los artistas *****");
// creación de un HashSet artistas que contenga a cantantes
// y actores
HashSet<Persona> artistas;
artistas=new HashSet<Persona>(cantantes);
artistas.addAll(actores);
// recorrer el primer HashSet de artistas
Iterator<Persona> it;
it=artistas.iterator();
// mientras queden elementos en el HashSet
Persona p;
while (it.hasNext()) {
    // recuperar el elemento en curso
    p=it.next();
    System.out.println(p.getApellido());
}
System.out.println("***** cantantes y actores
*****");
// creación de un HashSet de personas que son
// cantantes y actores

HashSet<Persona> act_cant;
act_cant=new HashSet<Persona>(cantantes);
act_cant.retainAll(actores);
it=act_cant.iterator();
// mientras queden elementos en el HashSet
while (it.hasNext()) {
    // recuperar el elemento en curso
    p=it.next();
    System.out.println(p.getApellido());
}
System.out.println("***** cantantes únicamente
*****");
// creación de un HashSet de personas
// únicamente cantantes

```

```

HashSet<Persona> unicamenteCantantes;
unicamenteCantantes=new HashSet<Persona>(cantantes);
unicamenteCantantes.removeAll(actores);
for(Persona pe:unicamenteCantantes) {
    System.out.println(pe.getApellido());
}

System.out.println("***** actores unicamente
*****");
// creación de un HashSet de personas
// unicamente actores
HashSet<Persona> unicamenteActores;
unicamenteActores=new HashSet<Persona>(actores);
unicamenteActores.removeAll(cantantes);
for(Persona pe:unicamenteActores) {
    System.out.println(pe.getApellido());
}
}
}

```

4. La clase LinkedList

La clase `LinkedList` es muy completa, pues implementa las interfaces `Iterable`, `Collection`, `Deque`, `List` y `Queue`. Gracias a la implementación de las interfaces `Collection`, `List` e `Iterable`, posee un funcionamiento idéntico al de la clase `ArrayList`. La implementación de las interfaces `Deque` y `Queue` aporta algunas funcionalidades suplementarias que vamos a estudiar.

Se recomienda utilizar esta clase si necesita acceder a los elementos en el mismo orden en que se han almacenado o en orden inverso. Estos mecanismos se llaman *First in-First out* (FIFO) o *Last in-First out* (LIFO).

Es posible agregar un elemento mediante los métodos `addFirst` o `addLast` para agregar un elemento al inicio de la lista o al final de la lista.

Para obtener un elemento presente en la lista, existen dos opciones posibles:

- Obtener el elemento dejándolo en la lista, en cuyo caso se utilizan los métodos `peekFirst` o `peekLast`.
- Obtener el elemento retirándolo de la lista, en cuyo caso se utilizan los métodos `pollFirst` o `pollLast`.

```
import java.time.LocalDate;
import java.util.LinkedList;

public class TestLinkedList
{
    public static void main(String[] args)
    {
        LinkedList<Persona> ll;
        ll=new LinkedList<Persona>();
        // creación de personas para completar el HashSet
        Persona p1,p2,p3,p4,p5;
        p1 = new Persona("Wayne", "John",LocalDate.of(1907,5,26));
        p2 = new Persona("McQueen", "Steve",LocalDate.of(1930,3,24));
        p3 = new Persona("Lennon", "John",LocalDate.of(1940,10,9));
        p4 = new Persona("Gibson", "Mel",LocalDate.of(1956,1,3));
        p5 = new Persona("Willis", "Bruce",LocalDate.of(1955,3,19));
        // agregar elementos a la lista
        ll.addFirst(p1);
        ll.addFirst(p2);
        ll.addFirst(p3);
        ll.addFirst(p4);
        ll.addFirst(p5);
        Persona p=null;
        // extracción y eliminación de elementos
        // de la lista comenzando por el más antiguo
        do
        {
            p=ll.pollLast();
            if (p!=null)
                System.out.println(p.getApellido());
        } while(p!=null);
    }
}
```

```
        ll.clear();  
    }  
}
```

5. La clase HashMap

La clase `HashMap` implementa la interfaz `Map`. Le permite manipular pares clave/valor. Es un tipo de colección que difiere significativamente de otros tipos de colecciones. Esta clase es útil si necesita manipular elementos basados en una clave. La clave y el valor pueden ser de cualquier tipo.

El siguiente ejemplo propone un objeto que permite la manipulación de pares `Integer/String`, siendo la clave del tipo `Integer` y el valor del tipo `String`. El método `put` hace posible agregar un par clave/valor. La clave es única; si ya se utiliza, el método `put` simplemente sobrescribe el par clave/valor ya presente. El método `remove` elimina una entrada. Espera un parámetro correspondiente a la clave que se ha de eliminar. El método `get` le permite obtener el valor vinculado a la clave pasada como parámetro.

Los métodos `containsKey` y `containsValue` permiten verificar respectivamente la presencia de una clave y la presencia de un valor utilizando su método `equals`.

Un conjunto de métodos le permite examinar las claves (`keySet`), los valores (`values`) o los pares clave/valor (`entrySet`).

```
import java.util.HashMap;  
import java.util.Map;  
import java.util.Map.Entry;  
  
public class TestHashMap {  
  
    public static void main(String[] args) {  
        Map<Integer, String> ciudades = new HashMap<Integer, String>();  
        ciudades.put(35, "Valencia");  
        ciudades.put(44, "Santander");  
        ciudades.put(56, "Alicante");  
        ciudades.put(29, "Fin");  
    }  
}
```

```
//Corrección por eliminación:
ciudades.put(29, "Oviedo");
ciudades.put(22, "León");
//Eliminación de una entrada
ciudades.remove(22);

String dep35 = ciudades.get(35);
System.out.println("El nombre de la ciudad 35 es: " + dep35);

System.out.println("¿Se hace referencia a la ciudad 29?");
//Buscar un elemento por la clave
if(ciudades.containsKey(29))
{
    System.out.println("sí");
}
else
{
    System.out.println("no");
}

System.out.println("¿Vigo es una ciudad?");
//Buscar un elemento por el valor
if(ciudades.containsValue("Vigo"))
{
    System.out.println("sí");
}
else
{
    System.out.println("no");
}

System.out.println("Lista de claves");
//Recorrer la colección de claves (es un Set porque cada clave
es única)
for(int clave:ciudades.keySet())
{
    System.out.println("-"+clave);
}
```

```
System.out.println("Lista de valores");
//Recorrer la colección de valores
for(String valor:ciudades.values())
{
    System.out.println("-"+valor);
}

System.out.println("Lista de las entradas clave/valor");
//Recorrer la colección de clave/valor (es un Set porque cada
clave
es única)
    for(Entry<Integer, String> entrada:ciudades.entrySet())
    {
        System.out.println(entrada.getKey()+"="+entrada.getValue());
    }
}
```

6. Streams y pipelines

Esta breve introducción a las colecciones está lejos de ser exhaustiva. Muchas funcionalidades adicionales se simplifican o pasan a estar disponibles, gracias a la llegada de expresiones lambda y de la API de Streams y Pipelines. Para obtener más información, consulte el próximo capítulo, Las expresiones lambda. Dentro de este marco se abordará la clasificación de las listas.

Ejercicios

1. Ejercicio 1

Crear una clase que represente un artículo de una tienda de venta por correspondencia. Un artículo se caracteriza por su referencia, su nombre y su precio. Crear, a continuación, un método `main` que permita probar el correcto funcionamiento de la clase anterior.

2. Ejercicio 2

Agregar las dos clases `Libro` y `Dvd` que heredan de la clase `Articulo`.

Un libro posee un número ISBN, contiene cierto número de páginas y lo ha escrito un autor; un DVD tiene cierta duración y lo ha producido un realizador.

Agregar los atributos necesarios a las clases `Libro` y `Dvd` para obtener el nombre del autor o del realizador. Probar, a continuación, el funcionamiento de estas dos nuevas clases.

3. Ejercicio 3

Modificar las clases `Libro` y `Dvd` para tener disponible información relativa al autor o el realizador:

- su apellido,
- su nombre,
- su fecha de nacimiento.

Pista: los autores y realizadores son personas.

4. Ejercicio 4

Modificar el código anterior para poder obtener rápidamente la lista de artículos relacionados con un autor o un realizador.

Correcciones

1. Ejercicio 1

La clase `Articulo` es relativamente simple. Las líneas 5 a 7 contienen la declaración de atributos que representan la información que vamos a almacenar en las instancias de la clase. Se declaran con la visibilidad `private` para garantizar que solo el código de la clase pueda acceder a ellas. A continuación se declaran los constructores (líneas 9 a 29). Las distintas versiones permiten crear instancias en función de la información disponible. Para evitar la duplicación de código, los constructores pueden invocarse mutuamente (líneas 16, 21, 27).

Si bien es posible utilizar directamente los atributos en los constructores, es preferible usar los métodos `setXXX` para acceder a ellos, de cara a aprovechar las eventuales verificaciones realizadas sobre los valores que se aplican a dichos atributos.

Para hacer que los atributos sean accesibles desde el exterior de la clase, cada uno debe disponer de los métodos `getXXX` y `setXXX`, que permiten obtener el valor del atributo o asignarle uno nuevo.

El método `toString` permite obtener una representación en forma de cadena de caracteres de la instancia de la clase.

El método `main` crea, a continuación, dos instancias de la clase `Articulo`, bien utilizando el constructor por defecto e inicializando a continuación los atributos uno a uno (líneas 9 a 12), o bien utilizando el constructor correspondiente a la lista de atributos disponibles (línea 14). Las dos instancias se muestran, a continuación, por la llamada de la función `prueba`, que permite definir el formato de representación o bien utilizar el método `toString`.

En este caso, el formato lo impone la clase Artículo.

Material para descargar

```
1. package es.eni.editions.ejercicio1;
2.
3. public class Artículo
4. {
5.     private int referencia;
6.     private String designacion;
7.     private double precio;
8.
9.     public Artículo()
10.    {
11.        super();
12.    }
13.
14.    public Artículo(int referencia)
15.    {
16.        this();
17.        setReferencia(referencia);
18.    }
19.    public Artículo(int referencia,String designacion)
20.    {
21.        this(referencia);
22.        setDesignacion(designacion);
23.    }
24.
25.    public Artículo(int referencia,String designacion, double
precio)
26.    {
27.        this(referencia,designacion);
28.        setPrecio(precio);
29.    }
30.
31.    public int getReferencia()
32.    {
33.        return referencia;
34.    }
```

```

35.
36.     public void setReferencia(int referencia)
37.     {
38.         this.referencia = referencia;
39.     }
40.
41.     public String getDesignacion()
42.     {
43.         return designacion;
44.     }
45.
46.     public void setDesignacion(String designacion)
47.     {
48.         this.designacion = designacion;
49.     }
50.
51.     public double getPrecio()
52.     {
53.         return precio;
54.     }
55.
56.     public void setPrecio(double precio)
57.     {
58.         this.precio = precio;
59.     }
60.
61.     public String toString()
62.     {
63.         return getReferencia() + " " + getDesignacion() +
64.         " " + getPrecio();
65.     }
66.     }
67.
/***** Clase Principal *****/
1. package es.eni.editions.ejercicio1;
2.
3.

```

```

4. public class Principal
5. {
6.     public static void main(String[] args)
7.     {
8.         Artículo a1,a2;
9.         a1=new Artículo();
10.        a1.setReferencia(100);
11.        a1.setNombre("Mortadelo y Filemón");
12.        a1.setPrecio(8.5);
13.
14.        a2=new Artículo(110,"El escarabajo de oro",8.5);
15.
16.        test(a1);
17.        System.out.println(a1.toString());
18.
19.        test(a2);
20.        System.out.println(a2.toString());
21.
22.    }
23.
24.    public static void test(Artículo a)
25.    {
26.        System.out.println("referencia: " + a.getReferencia());
27.        System.out.println("designacion: " +
28.            a.getDesignacion());
29.        System.out.println("precio: " + a.getPrecio() + " €");
30.    }

```

2. Ejercicio 2

Las dos clases `Libro` y `Dvd` heredan de la clase `Artículo` gracias a la palabra clave `extends`, que se agrega a continuación de sus definiciones. La estructura de ambas clases es similar a la de la clase `Artículo`. Se agregan los atributos correspondientes exigidos por las clases `Libro` y `Dvd`. Los demás atributos y métodos se heredan de la clase `Artículo`. La llamada a los métodos de la clase `Artículo` se realiza mediante

la palabra clave super.

```

/***** Artículo *****/
1. package es.eni.editions.ejercicio2;
2.
3. public class Artículo
4. {
5.     private int referencia;
6.     private String designacion;
7.     private double precio;
8.
9.     public Artículo()
10.    {
11.        super();
12.    }
13.
14.    public Artículo(int referencia)
15.    {
16.        this();
17.        setReferencia(referencia);
18.    }
19.    public Artículo(int referencia,String designacion)
20.    {
21.        this(referencia);
22.        setDesignación(designacion);
23.    }
24.
25.    public Artículo(int referencia,String designacion,
26.        double precio)
27.    {
28.        this(referencia,designacion);
29.        setPrecio(precio);
30.    }
31.    public int getReferencia()
32.    {
33.        return referencia;
34.    }

```

```

35.
36.     public void setReferencia(int referencia)
37.     {
38.         this.referencia = referencia;
39.     }
40.
41.     public String getDesignacion()
42.     {
43.         return designacion;
44.     }
45.
46.     public void setDesignacion(String designacion)
47.     {
48.         this.designacion = designacion;
49.     }
50.
51.     public double getPrecio() {
52.         return precio;
53.     }
54.
55.     public void setPrecio(double precio)
56.     {
57.         this.precio = precio;
58.     }
59.
60.     public String toString()
61.     {
62.         return getReferencia() + " " + getDesignacion() +
63.             " " + getPrecio();
64.     }
65. }
66.
67. /***** Libro *****/
68. 1. package es.eni.editions.ejercicio2;
69. 2.
70. 3. public class Libro extends Artículo
71. 4. {

```

```
5.     private String isbn;
6.     private int numPaginas;
7.     private String autor;
8.
9.     public Libro()
10.    {
11.        super();
12.    }
13.
14.     public Libro(int referencia,String designacion,
15.        double precio,String isbn,int numPaginas,String autor)
16.    {
17.        super(referencia,designacion,precio);
18.        setIsbn(isbn);
19.        setNumPaginas(numPaginas);
20.        setAutor(autor);
21.    }
22.
23.     public String getIsbn() {
24.         return isbn;
25.     }
26.
27.     public void setIsbn(String isbn) {
28.         this.isbn = isbn;
29.     }
30.
31.     public int getNumPaginas() {
32.         return numPaginas;
33.     }
34.
35.     public void setNumPaginas(int numPaginas) {
36.         this.numPaginas = numPaginas;
37.     }
38.
39.     public String getAutor() {
40.         return autor;
41.     }
```

```

42.         public void setAutor(String autor) {
43.             this.autor = autor;
44.         }
45.         public String toString()
46.         {
47.             return super.toString() + " " + getNumPaginas() +
               " " + getAutor();
48.         }
49.
50.     }

```

/***** Dvd *****/

```

1. package es.eni.editions.ejercicio2;
2.
3. import java.time.Duration;
4.
5.
6. public class Dvd extends Artículo
7. {
8.     private Duration duracion;
9.     private String realizador;
10.
11.     public Dvd()
12.     {
13.         super();
14.     }
15.
16.     public Dvd(int referencia,String nombre,double
       precio,Duration duracion,String realizador)
17.     {
18.         super(referencia,nombre,precio);
19.
20.         setDuracion(duracion);
21.         setRealizador(realizador);
22.     }
23.
24.     public Duration getDuracion()
25.     {

```



```

26.         return duracion;
27.     }
28.
29.     public void setDuracion(Duration duracion)
30.     {
31.         this.duracion = duracion;
32.     }
33.
34.     public String getRealizador()
35.     {
36.         return realizador;
37.     }
38.
39.     public void setRealizador(String realizador)
40.     {
41.         this.realizador = realizador;
42.     }
43.
44.     public String toString()
45.     {
46.         return super.toString() + " " +
47.             getDuracion().toMinutes() + " " + getRealizador();
48.     }

```

/****** Clase Principal *****/

```

1. package es.eni.editions.ejercicio2;
2.
3. import java.time.Duration;
4.
5. public class Principal
6. {
7.     public static void main(String[] args)
8.     {
9.         Libro l;
10.        Dvd d;
11.        l=new Libro();

```

```

12.         l.setReferencia(100);
13.         l.setNombre("El sueño de la razón");
14.         l.setPrecio(18.00);
15.         l.setNumPaginas(352);
16.         l.setAutor("Aguilera");
17.         System.out.println(l.toString());
18.
19.         d=new Dvd();
20.         d.setReferencia(110);
21.         d.setNombre("La escopeta nacional");
22.         d.setPrecio(24.62);
23.         d.setDuracion(Duration.ofMinutes(91));
24.         d.setRealizador("Berlanga");
25.         System.out.println(d.toString());
26.
27.     }
28. }

```

3. Ejercicio 3

Para agregar información suplementaria relativa al autor o al realizador, es preferible crear una clase `Persona` que agrupe apellido, nombre y fecha de nacimiento. Esta clase se utiliza, a continuación, para remplazar al tipo `String` de los atributos `autor` y `realizador` de las clases `Libro` y `Dvd`. Es preciso modificar la firma de los constructores y los métodos `getXXX` y `setXXX` para tener en cuenta dicha modificación.

```

/***** Artículo *****/

1. package es.eni.editions.ejercicio3;
2.
3. public class Artículo
4. {
5.     private int referencia;
6.     private String designacion;
7.     private double precio;
8.

```

```
9.     public Artículo()
10.    {
11.        super();
12.    }
13.
14.    public Artículo(int referencia)
15.    {
16.        this();
17.        setReferencia(referencia);
18.    }
19.    public Artículo(int referencia,String designacion)
20.    {
21.        this(referencia);
22.        setDesignacion(designacion);
23.    }
24.
25.    public Artículo(int referencia,String designacion,double
precio)
26.    {
27.        this(referencia,designacion);
28.        setPrecio(precio);
29.    }
30.
31.    public int getReferencia()
32.    {
33.        return referencia;
34.    }
35.
36.    public void setReferencia(int referencia)
37.    {
38.        this.referencia = referencia;
39.    }
40.
41.    public String getDesignacion()
42.    {
43.        return designacion;
44.    }
45.
```

```

46.     public void setDesignacion(String designacion)
47.     {
48.         this.designacion = designacion;
49.     }
50.
51.     public double getPrecio() {
52.         return precio;
53.     }
54.
55.     public void setPrecio(double precio)
56.     {
57.         this.precio = precio;
58.     }
59.
60.     public String toString()
61.     {
62.         return getReferencia() + " " + getDesignacion() +
           " " + getPrecio();
63.     }
64.
65. }

```

```

/***** Libro *****/

```

```

1. package es.eni.editions.ejercicio3;
2.
3. public class Libro extends Artículo
4. {
5.     private String isbn;
6.     private int numPaginas;
7.     private Persona autor;
8.
9.     public Libro()
10.    {
11.        super();
12.    }
13.
14.     public Libro(int referencia,String designacion,double
        precio,String isbn,int numPaginas,Persona autor)

```

```
15.     {
16.         super(referencia,designacion,precio);
17.         setIsbn(isbn);
18.         setNumPaginas(numPaginas);
19.         setAutor(autor);
20.     }
21.
22.     public String getIsbn() {
23.         return isbn;
24.     }
25.
26.     public void setIsbn(String isbn) {
27.         this.isbn = isbn;
28.     }
29.
30.     public int getNumPaginas() {
31.         return numPaginas;
32.     }
33.
34.     public void setNumPaginas(int numPaginas) {
35.         this.numPaginas = numPaginas;
36.     }
37.
38.     public Persona getAutor() {
39.         return autor;
40.     }
41.
42.     public void setAutor(Persona autor) {
43.         this.autor = autor;
44.     }
45.     public String toString()
46.     {
47.         return super.toString() + " " + getNumPaginas() + "
         " + getAutor();
48.     }
49.
50. }
```

/***** Dvd *****/

```
1. package es.eni.editions.ejercicio3;
2.
3. import java.time.Duration;
4.
5.
6. public class Dvd extends Artículo
7. {
8.     private Duration duracion;
9.     private Persona realizador;
10.
11.     public Dvd()
12.     {
13.         super();
14.     }
15.
16.     public Dvd(int referencia,String designacion,double
17.         precio,Duration duracion,Persona realizador)
18.     {
19.         super(referencia,designación,precio);
20.
21.         setDuracion(duracion);
22.         setRealizador(realizador);
23.
24.     public Duration getDuracion()
25.     {
26.         return duracion;
27.     }
28.
29.     public void setDuracion(Duration duracion)
30.     {
31.         this.duracion = duracion;
32.     }
33.
34.     public Persona getRealizador()
35.     {
```

```

36.         return realizador;
37.     }
38.
39.     public void setRealizador(Persona realizador)
40.     {
41.         this.realizador = realizador;
42.     }
43.
44.     public String toString()
45.     {
46.         return super.toString() + "
            " + getDuracion().toMinutes() + " " + getRealizador();
47.     }
48. }

```

/****** Persona *****/

```

1. package es.eni.editions.ejercicio3;
2.
3. import java.time.LocalDate;
4.
5. public class Persona
6. {
7.     private String apellido;
8.     private String nombre;
9.     private LocalDate fechaDeNacimiento;
10.
11.     public Persona()
12.     {
13.         super();
14.     }
15.
16.     public Persona(String n,String p,LocalDate d)
17.     {
18.         this.apellido=n;
19.         this.nombre=p;
20.         this.fechaDeNacimiento=d;
21.     }

```

```
22.
23.     public String getApellido()
24.     {
25.         return apellido;
26.     }
27.
28.     public void setApellido(String apellido)
29.     {
30.         this.apellido = apellido;
31.     }
32.
33.     public String getNombre()
34.     {
35.         return nombre;
36.     }
37.
38.     public void setNombre(String nombre)
39.     {
40.         this.nombre = nombre;
41.     }
42.
43.     public LocalDate getFechaDeNacimiento()
44.     {
45.         return fechaDeNacimiento;
46.     }
47.
48.     public void setFechaDeNacimiento(LocalDate fechaDeNacimiento)
49.     {
50.         this.fechaDeNacimiento = fechaDeNacimiento;
51.     }
52.
53.     public String toString() {
54.         return nombre + " " + apellido;
55.     }
56.
57. }
```

```
/****** Clase Principal *****/
```



```
1. package es.eni.editions.ejercicio3;
2.
3. import java.time.Duration;
4. import java.time.LocalDate;
5.
6.
7.
8. public class Principal
9. {
10.     public static void main(String[] args)
11.     {
12.         Libro l;
13.         Dvd d;
14.         l=new Libro();
15.         l.setReferencia(100);
16.         l.setNombre("El sueño de la razón");
17.         l.setPrecio(18.00);
18.         l.setNumPaginas(352);
19.         l.setAutor(new
                Persona("Aguilera", "Juan
Miguel", LocalDate.of(1960,8,7)));
20.         testLibro(l);
21.         System.out.println(l.toString());
22.
23.         d=new Dvd();
24.         d.setReferencia(110);
25.         d.setNombre("la escopeta nacional");
26.         d.setPrecio(24.62);
27.         d.setDuracion(Duration.ofMinutes(91));
28.         d.setRealizador(new
                Persona("Berlanga", "Luis
García", LocalDate.of(1921,6,21)));
29.         testDvd(d);
30.         System.out.println(d.toString());
31.
32.     }
33.
```

```

34.     public static void test(Articulo a)
35.     {
36.         System.out.println("referencia: " +
37.                               a.getReferencia());
38.         System.out.println("designación: " +
39.                               a.getDesignacion());
40.         System.out.println("precio: " + a.getPrecio() + " €");
41.     }
42.
43.     public static void testLibro(Libro l)
44.     {
45.         test(l);
46.         System.out.println("número de páginas: " +
47.                               l.getNumPaginas());
48.         System.out.println("autor: " +
49.                               l.getAutor().toString());
50.     }
51.
52.     public static void testDvd(Dvd d)
53.     {
54.         test(d);
55.         System.out.println("duración: " +
56.                               d.getDuracion().toMinutes() + " minutos");
57.         System.out.println("realizador: " +
58.                               d.getRealizador().toString());
59.     }
60. }

```

4. Ejercicio 4

En el ejercicio anterior hemos implementado un vínculo entre un DVD o un libro y el autor o realizador correspondiente. A menudo, resulta interesante poder disponer de un vínculo bidireccional. En nuestro caso, esto nos va a permitir obtener fácilmente la lista de obras de un autor o realizador. Basta con agregar a la clase `Persona` un atributo de tipo `ArrayList`. Este permite asociar a un autor o realizador la lista de todas sus obras. La instancia del objeto `ArrayList` se crea en el constructor y permite almacenar obras de

esta persona. Los métodos `setAutor` y `setRealizador` de las clases `Libro` y `Dvd` se modifican también. Tras la asociación, se verifica si el `Dvd` o el `Libro` ya están en la lista de obras de dicha persona. Si no fuera el caso, se agrega el `Dvd` o el `Libro` a la lista.

```

/***** Artículo *****/

1. package es.eni.editions.ejercicio4;
2.
3. public class Artículo
4. {
5.     private int referencia;
6.     private String designacion;
7.     private double precio;
8.
9.     public Artículo()
10.    {
11.        super();
12.    }
13.
14.     public Artículo(int referencia)
15.    {
16.        this();
17.        setReferencia(referencia);
18.    }
19.     public Artículo(int referencia,String designacion)
20.    {
21.        this(referencia);
22.        setDesignacion(designacion);
23.    }
24.
25.     public Artículo(int referencia,String designacion,
26.        double precio)
27.    {
28.        this(referencia,designacion);
29.        setPrecio(precio);
30.    }

```

```
31.     public int getReferencia()
32.     {
33.         return referencia;
34.     }
35.
36.     public void setReferencia(int referencia)
37.     {
38.         this.referencia = referencia;
39.     }
40.
41.     public String getDesignacion()
42.     {
43.         return designacion;
44.     }
45.
46.     public void setDesignacion(String designacion)
47.     {
48.         this.designacion = designacion;
49.     }
50.
51.     public double getPrecio() {
52.         return precio;
53.     }
54.
55.     public void setPrecio(double precio)
56.     {
57.         this.precio = precio;
58.     }
59.
60.     public String toString()
61.     {
62.         return getReferencia() + " " + getDesignacion() + "
        " + getPrecio();
63.     }
64.
65. }
```

```
/****** Libro *****/
```

```
1. package es.eni.editions.ejercicio4;
2.
3. import java.util.ArrayList;
4.
5.
6.
7. public class Libro extends Articulo
8. {
9.     private String isbn;
10.     private int numPaginas;
11.     private Persona autor;
12.
13.     public Libro()
14.     {
15.         super();
16.     }
17.
18.     public Libro(int referencia,String designacion,double
19.         precio,String isbn,int numPaginas,Persona autor)
20.     {
21.         super(referencia,designacion,precio);
22.         setIsbn(isbn);
23.         setNumPaginas(numPaginas);
24.         setAutor(autor);
25.     }
26.     public String getIsbn() {
27.         return isbn;
28.     }
29.
30.     public void setIsbn(String isbn) {
31.         this.isbn = isbn;
32.     }
33.
34.     public int getNumPaginas() {
35.         return numPaginas;
36.     }
```

```

37.
38.     public void setNumPaginas(int numPaginas) {
39.         this.numPaginas = numPaginas;
40.     }
41.
42.     public Persona getAutor() {
43.         return autor;
44.     }
45.
46.     public void setAutor(Persona autor) {
47.         //si el libro pertenece a otro autor
48.         //se retira de la lista de este antiguo autor
49.
50.         if(this.autor != null)
51.         {
52.             this.autor.getObras ().remove(this);
53.         }
54.         this.autor = autor;
55.
56.         ArrayList<Articulo> lst;
57.         lst=autor.getObras();
58.         if (!lst.contains(this))
59.         {
60.             lst.add(this);
61.         }
62.     }
63.     public String toString()
64.     {
65.         return super.toString() + " " + getNumPaginas() + "
66.         " + getAutor();
67.     }
68. }
69.
70. /***** Dvd *****/

```

```

1. package es.eni.editions.ejercicio4;
2.

```

```
3. import java.time.Duration;
4. import java.util.ArrayList;
5.
6.
7. public class Dvd extends Artículo
8. {
9.     private Duration duracion;
10.    private Persona realizador;
11.
12.    public Dvd()
13.    {
14.        super();
15.    }
16.
17.    public Dvd(int referencia,String designacion,double
    precio,Duration duracion,Persona realizador)
18.    {
19.        super(referencia,designacion,precio);
20.
21.        setDuracion(duracion);
22.        setRealizador(realizador);
23.    }
24.
25.    public Duration getDuracion()
26.    {
27.        return duracion;
28.    }
29.
30.    public void setDuracion(Duration duracion)
31.    {
32.        this.duracion = duracion;
33.    }
34.
35.    public Persona getRealizador ()
36.    {
37.        return realizador;
38.    }
39.
```

```

40.     public void setRealizador(Persona realizador)
41.     {
42.         //si el libro pertenece a otro realizador
43.         //entonces se elimina de la lista de este realizador
antiguo
44.         if(this.realizador!=null)
45.         {
46.             this.realizador.getObras ().remove(this);
47.         }
48.         this.realizador = realizador;
49.         ArrayList<Articulo> lst;
50.         lst=realizador.getObras();
51.         if (!lst.contains(this))
52.         {
53.             lst.add(this);
54.         }
55.     }
56.
57.     public String toString()
58.     {
59.         return super.toString() + " " +
            getDuracion().toMinutes() + " " + getRealizador();
60.     }
61. }

```

/***** Persona *****/

```

1. package es.eni.editions.ejercicio4;
2.
3. import java.time.LocalDate;
4. import java.util.ArrayList;
5.
6. public class Persona
7. {
8.     private String apellido;
9.     private String nombre;
10.    private LocalDate fechaDeNacimiento;
11.    private ArrayList<Articulo> obras;

```



```
12.
13.     public Persona()
14.     {
15.         super();
16.         obras=new ArrayList<> ();
17.     }
18.
19.     public Persona(String n,String p,LocalDate d)
20.     {
21.         this();
22.         this.apellido=n;
23.         this.nombre=p;
24.         this.fechaDeNacimiento=d;
25.     }
26.
27.     public String getApellido()
28.     {
29.         return apellido;
30.     }
31.
32.     public void setApellido(String apellido)
33.     {
34.         this.apellido = apellido;
35.     }
36.
37.     public String getNombre()
38.     {
39.         return nombre;
40.     }
41.
42.     public void setNombre(String nombre)
43.     {
44.         this.nombre = nombre;
45.     }
46.
47.     public LocalDate getFechaDeNacimiento()
48.     {
49.         return fechaDeNacimiento;
```

```

50.         }
51.
52.         public void setFechaDeNacimiento(LocalDate
fechaDeNacimiento)
53.         {
54.             this.fechaDeNacimiento = fechaDeNacimiento;
55.         }
56.
57.         public ArrayList<Articulo> getObras()
58.         {
59.             return obras;
60.         }
61.
62.         public String toString() {
63.             return nombre + " " + apellido;
64.         }
65.     }
/***** Principal *****/

```

```

1.  package es.eni.editions.ejercicio4;
2.
3.  import java.time.Duration;
4.  import java.time.LocalDate;
5.
6.  public class Principal {
7.      public static void main(String[] args) {
8.          Libro l;
9.          Dvd d;
10.         l = new Libro ();
11.         l.setReferencia(100);
12.         l.setTitulo("El cangrejo de pinza dorada");
13.         l.setPrecio(8.5);
14.         l.setNumPaginas (86);
15.         l.setAutor(new Persona ("Hergé", "Georges",
LocalDate.of(1907, 05, 22)));
16.         testLibro (l);
17.         System.out.println(l.toString());
18.

```

```
19.      d = new Dvd();
20.      d.setReferencia(110);
21.      d.setTitulo ("Sopa de repollo ");
22.      d.setPrecio (19.50);
23.      d.setDuracion (Duration.ofMinutes(98));
24.      Persona lucBesson = new Persona("Besson", "luc",
        LocalDate.of(1959, 03, 18));
25.      d.setRealizador(lucBesson);
26.      testDvd(d);
27.      System.out.println(d.toString());
28.      System.out.println("Película de Luc Besson:");
29.      for(Article a: lucBesson.getObras ())
30.      {
31.          System.out.println(a);
32.      }
33.
34.      //Cambio del realizador porque está erróneo.
35.      d.setRealizador(new Persona ("Girault", "jean",
        LocalDate.of(1924, 05, 9)));
36.      testDvd(d);
37.      System.out.println(d.toString());
38.      System.out.println("Película de Luc Besson:");
39.      if(lucBesson.getObras ().size()>0)
40.      {
41.          for(Articulo a: lucBesson.getObras ())
42.          {
43.              System.out.println(a);
44.          }
45.      }
46.      else
47.      {
48.          System.out.println("Ningún obra ");
49.      }
50.}
51.
52. public static void test(Articulo a) {
53.     System.out.println("Referencia : " + a.getReferencia());
54.     System.out.println("título : " + a.getTitulo ());
```

```
56.         System.out.println("precio : " + a.getPrecio() + " €");
57.     }
58.
59.     public static void testLibro (Libro l) {
60.         test(l);
61.         System.out.println("número de páginas: " +
62.         l.getNumPaginas());
63.         System.out.println("autor: " + l.getAutor().toString());
64.     }
65.
66.     public static void testDvd(Dvd d) {
67.         test(d);
68.         System.out.println("duración: " + d.getDuracion
69.         ().toMinutes() +
70.         " minutos");
71.         System.out.println("realizador: " +
72.         d.getRealizador().toString());
73.     }
74. }
```

Introducción

En el ecosistema de Java, las expresiones lambda han sido largamente esperadas desde hace mucho tiempo. Aparecieron con Java 8. Java se ha puesto al día con ciertos lenguajes competitivos, como C #. Pero ¿qué son las expresiones lambda? Son simplemente funciones que se pueden pasar como un parámetro de otra función. Esto simplifica el código en ciertas situaciones donde el desarrollo de objetos ofrece soluciones detalladas basadas en clases anónimas en particular.

Funcionamiento

1. Las interfaces funcionales

El funcionamiento de las expresiones lambda se basa en el funcionamiento de las interfaces. De hecho, un parámetro de método que espera una expresión lambda es simplemente un parámetro del tipo de una interfaz. Sin embargo, **esta interfaz** debe respetar una regla importante: debe **definir la firma de un solo método abstracto**.

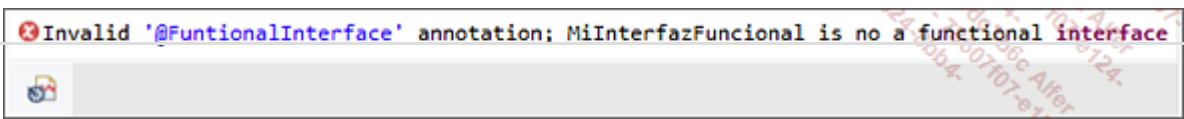
En este caso, la interfaz se convierte en una interfaz funcional. Siempre es posible tener métodos adicionales si no son abstractos (métodos predeterminados, métodos estáticos).

Por ejemplo, he aquí el código simplificado para la interfaz

`java.util.Comparator<T>`. Su función es proporcionar un mecanismo para comparar objetos del mismo tipo:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}
```

Una interfaz funcional es una interfaz como las demás en su estructura. La diferencia es la presencia de la anotación `@FunctionalInterface`, que también es opcional, pero que permite al compilador realizar las comprobaciones de coherencia necesarias. Siempre que no tenga exactamente un único método abstracto en la interfaz, el compilador le indicará el siguiente mensaje:

A screenshot of a Java IDE showing a red error message. The message reads: "Invalid '@FunctionalInterface' annotation; MiInterfazFuncional is not a functional interface". The text "MiInterfazFuncional" is highlighted in blue, and "functional interface" is highlighted in red. There is a small icon of a person in the bottom left corner of the error box.

Invalid '@FunctionalInterface' annotation; MiInterfazFuncional is not a functional interface

Java ofrece un conjunto de interfaces funcionales que responden a la gran mayoría de las situaciones. Por lo tanto, lo más probable es que no necesite crearlas usted mismo, o al menos no con frecuencia. Están disponibles en el paquete `java.util.function`. Una sección dedicada le permitirá explorar el contenido de este paquete.

Un ejemplo concreto del uso de interfaces funcionales es el método de clasificación de la interfaz `List <T>`. Este método permite ordenar los elementos de la lista de acuerdo con un criterio de comparación proporcionado como un parámetro del método, en la forma de una implementación de la interfaz `Comparator <T>` que, le recuerdo, permite comparar dos objetos del mismo tipo.

Aquí está la firma del método `sort`:

```
void java.util.List.sort (Comparator <? super String> c)
```

El método `sort` recibe como argumento de tipo `Comparator <T>`. Como es una interfaz funcional, es posible pasar como parámetro:

- Una instancia de clase que implementa esta interfaz:

```
List <String> lista = new ArrayList <String> (
    List.of ("hola", "todo", "el", "mundo"));

lista.sort (nuevo Comparador <String> () {
    @ Override
    public int compare (String s1, String s2) {
        return s1.compareTo (s2);
    }
});
```

Para obtener más información sobre clases anónimas, consulte la sección homónima en el capítulo de Programación orientada a objetos.

- Una expresión lambda (o método anónimo), que respeta la firma del método abstracto único de esta interfaz:

```
lista.sort ((s1, s2) -> s1.compareTo (s2));
```

Para obtener más información sobre las reglas de sintaxis, consulte la siguiente sección, Los métodos anónimos.

- Una referencia a una variable del tipo de la interfaz:

```
Comparador <String> comparador = (s1, s2) -> s1.compareTo (s2);  
lista.sort,(comparador);
```

- Una referencia a un método existente, que respeta la firma del método abstracto único de esta interfaz:

```
private static int compare(String s1, String s2)  
{  
    return s1.compareTo(s2);  
}
```

Para pasar como parámetro una referencia a este método, la escritura es la siguiente:

```
lista.sort (MiClase::compare);
```

Para obtener más información sobre las reglas de sintaxis, consulte la sección Las referencias de métodos, más adelante en este capítulo.

Estos primeros ejemplos muestran que una expresión lambda ofrece una sintaxis mucho más concisa que una clase anónima. La referencia del método también permite que el procesamiento se factorice en un método reutilizable.

2. Los métodos anónimos

Existen diferentes sintaxis para escribir métodos anónimos, en función de diferentes criterios:

- El número de parámetros,
- El tipo de método: función o procedimiento,
- El número de instrucciones para escribir en el método anónimo.

a. Sintaxis general

Una expresión lambda tiene la siguiente forma general:

```
parametro -> cuerpo
```

Una expresión lambda debe declarar los parámetros antes de declarar el cuerpo del método. Estos dos bloques están necesariamente separados por una flecha `->`.

b. Declaración de los argumentos

Si no hay ningún argumento, la declaración se escribe de la siguiente manera:

```
() -> cuerpo
```

El par de paréntesis indica la ausencia de un parámetro.

Si hay uno o más parámetros, la declaración se puede escribir de la siguiente manera:

```
(TipoArgumento nombreArgumento, ...) -> cuerpo
```

Esto se hace exactamente igual que para un método convencional.

Los métodos anónimos, sin embargo, ofrecen algunas facilidades:

- No es obligatorio definir el tipo de parámetros. El compilador implementa la inferencia de tipos para adivinar el tipo de los parámetros. Por lo tanto, es posible simplificar la escritura anterior eliminando los tipos:

```
(nombreArgumento, ...) -> cuerpo
```

- Si hay un único parámetro, incluso es posible eliminar los paréntesis para la declaración de los parámetros:

```
nombreArgumento -> body
```

En este caso, no es posible definir explícitamente el tipo del parámetro.

c. Declaración del cuerpo

Después de declarar los parámetros, es necesario declarar el cuerpo del método anónimo. Para esto, es posible utilizar la misma organización que un método convencional, mediante el uso de llaves ({ }) como delimitadores:

```
... -> {  
    instrucción;  
    instrucción;  
    [return ...;]  
}
```

Al igual que con los parámetros, los métodos anónimos ofrecen algunas facilidades para escribir el cuerpo del método.

- Si el método contiene solo una instrucción (que suele ser el caso en un método anónimo), entonces es posible eliminar las llaves y el punto y coma.

```
... -> la única instrucción sin punto y coma
```

- Si el método contiene solo una instrucción cuyo resultado debe devolverse, no es útil usar la palabra clave `return`:

```
... -> la única instrucción que devuelve un valor sin return  
y punto y coma
```

d. Utilización de variables "externas"

Una expresión lambda puede acceder a variables «externas», es decir, variables declaradas fuera de la expresión lambda. Simplemente es necesario que la variable se declare `final` o que en realidad sea `final`. Una variable es efectivamente `final` si no se declara `final` pero se usa como si fuera `final`, en la expresión lambda (es decir, solo se usa en la lectura).

3. Las referencias de métodos

La segunda forma de explotar las interfaces funcionales es usar referencias de métodos. Es posible utilizar, en lugar de una expresión lambda, una referencia a un método cuya firma

respetar la firma del método abstracto único de la interfaz funcional. La forma de referenciar puede evolucionar de acuerdo con las diferentes situaciones descritas en las siguientes secciones.

a. Método de instancia

Si el método que se ha de referenciar es un método de instancia, la sintaxis es la siguiente:

```
nombreDeLaVariable::nombreDelMetodo
```

Si el método que se ha de referenciar está en la instancia con la que se está trabajando en ese momento, la sintaxis se convierte en la siguiente:

```
this::nombreDelMetodo
```

Los dos puntos (:) permiten separar el nombre del método de su «propietario».

También permiten indicar al compilador que no se trata de una llamada al método que se desea realizar, sino simplemente una referencia. Tenga en cuenta asimismo que no se proporcionan parámetros; los paréntesis están ausentes.

b. Método de clase

Si el método que se ha de referenciar es un método de clase (dicho de otra manera, estático), la sintaxis difiere un poco:

```
NombreDeLaClase::nombreDelMetodo
```

c. Constructor

Si el método que se ha de referenciar es un constructor, la sintaxis es la siguiente:

```
NombreDeLaClase::new
```

Para una matriz, será necesario usar la siguiente sintaxis:

```
NombreDeLaClase[]::new
```

Para una clase genérica, habría que utilizar la sintaxis siguiente:

4. La API `java.util.function`

a. Presentación de la API

La API `java.util.function` contiene 43 interfaces funcionales y, por lo tanto, el mismo número de firmas de métodos abstractos. La mayoría de estos métodos son genéricos, por lo que se adaptan según el contexto en el que desea utilizarlos. La javadoc está disponible en la siguiente dirección:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html>

Estas 43 interfaces respetan una convención de nomenclatura. El sufijo permite determinar la función general de la interfaz:

- `XxxConsumer`: las interfaces `XxxConsumer` permiten determinar la firma de un procedimiento. Dependiendo de la interfaz, el procedimiento acepta uno o dos parámetros. Por supuesto, no devuelve nada. Este tipo de interfaz permite realizar una acción. El método abstracto de estas interfaces siempre se llama `accept`.
- `XxxFunction`: las interfaces `XxxFunction` permiten determinar la firma de una función. Dependiendo de la interfaz, la función acepta uno o dos parámetros y devuelve un valor. Este tipo de interfaz permite realizar una transformación. El método abstracto de estas interfaces siempre se llama `apply`.
- `XxxOperator`: las interfaces `XxxOperator` permiten determinar la firma de una función cuyos tipos de entrada y salida son los mismos. Estas son especializaciones de ciertas interfaces del grupo `XxxFunction`.
- `XxxPredicate`: las interfaces `XxxPredicate` permiten determinar la firma de una función, que recibe uno o dos parámetros como entrada y devuelve un valor booleano. Este tipo de interfaz permite realizar una prueba. Además, el método abstracto de estas interfaces siempre se llama `test`.
- `XxxSupplier`: las interfaces `XxxSupplier` permiten determinar la firma de una función que no recibe parámetros y devuelve un valor. Representan a los getters.

El método abstracto de estas interfaces comienza con `get`.

Todas estas interfaces cubren una gran parte de las necesidades. Un conjunto de interfaces adicionales están disponible en el resto de los paquetes. Como puede ver, rara vez necesitará crear su propia interfaz funcional.

b. Utilización

Las siguientes secciones presentan diferentes casos de uso de expresiones lambda.

Procedimiento de un argumento

Vamos a tomar como ejemplo la visualización de una lista en la consola. El código clásico equivale a realizar un bucle *foreach*, como en el siguiente ejemplo:

```
public class PruebaLista {  
  
    public static void main(String[] args) {  
        List<String> lista = new ArrayList<String>(  
            List.of("hola",  
                    "todo",  
                    "el ",  
                    "mundo"));  
  
        System.out.println("Visualización clásica:");  
        for(String element : lista)  
        {  
            System.out.print(element+" ");  
        }  
    }  
}
```

Al estudiar con mayor precisión el tipo `List<T>`, es posible descubrir el método `foreach`. Este método, como su nombre indica, le permite realizar una acción para cada uno de los elementos de la lista. Recibe como argumento un objeto de tipo `Consumer<String>`. La interfaz `Consumer<String>`, como se explicó en la sección anterior, le permite definir un procedimiento para realizar una acción. El método abstracto de esta interfaz recibe un argumento genérico, en este caso de tipo `String`

(void accept (T t)). Es posible concluir que se llamará a este método para todos los elementos que componen la lista. Aquí hay diferentes ejemplos de uso de este método utilizando las diferentes sintaxis posibles, detalladas anteriormente:

```
System.out.println("Visualización con una expresión lambda:");

System.out.println("Ejemplo 1:");
//el argumento element recibe sucesivamente el valor
//de cada elemento de la lista.
//La expresión lambda respeta
//la firma de la interfaz Consumer<T> :
//void accept(T t);
lista.forEach((String element)->
    {
        System.out.print(element+" ");
    });

System.out.println();
System.out.println("Ejemplo 2:");
//Sintaxis más condensada como una única instrucción
lista.forEach(s->System.out.print(s+" "));

System.out.println();
System.out.println("Ejemplo 3:");
//Definición de una variable del tipo de la interfaz:
Consumer<String> visualizador = s->System.out.print(s+" ");
lista.forEach(visualizador);

System.out.println();
System.out.println("Ejemplo 4:");
//Uso de una referencia de método
lista.forEach(PruebaLista::visualizadorStatic);
```

El último ejemplo necesita la presencia del método `visualizadorStatic` en la clase `PruebaLista`:

```
//Definición de un método que respeta el método void accept(T t)
//de la interfaz Consumer<T>
```

```
private static void visualizadorStatic(String s) {  
    System.out.print(s+" ");  
}
```

Función de un argumento

Vamos a tomar como ejemplo la eliminación de elementos de una lista según un criterio. Comencemos nuevamente desde el ejemplo anterior y escribamos un programa que permita eliminar los elementos de la lista que contienen la letra o. Al llegar, solo debe permanecer el elemento el; los elementos hola, todo y mundo se eliminarán.

El código estándar para escribir podría basarse en un bucle `for` de la siguiente manera:

```
System.out.println("Eliminación de los elementos con un o");  
for(int i=lista.size()-1;i>=0;i--)  
{  
    if(lista.get(i).contains("o"))  
    {  
        lista.remove(i);  
    }  
}  
lista.forEach(s->System.out.println(s));
```

Es posible simplificar este código utilizando el método `removeIf`, con un objeto de tipo `Predicate <String>` como parámetro. La interfaz `Predicate<T>`, como se explicó en la sección anterior, le permite definir una función para evaluar una condición y devolver un valor en forma de un valor booleano. El método abstracto de esta interfaz recibe un argumento genérico, en este caso de tipo `String`, y devuelve un valor booleano (`boolean test(T t)`). Es posible concluir que se llamará a este método para todos los elementos que componen la lista. Si el valor devuelto es `true`, el elemento se eliminará de la lista; si el valor devuelto es `false`, el elemento se mantendrá en la lista.

```
System.out.println();  
System.out.println("Eliminación con una expresión lambda");  
  
System.out.println();  
System.out.println("Ejemplo 1:");
```

```
lista.removeIf((String s)->{return s.contains("o");});
lista.forEach(s->System.out.println(s));

System.out.println();
System.out.println("Ejemplo 2:");
//Sintaxis más condensada como una única instrucción
lista.removeIf(s->s.contains("o"));
lista.forEach(s->System.out.println(s));
```

Función de varios argumentos

Tomemos como ejemplo la ordenación de una lista con el método `sort`. Este método espera un objeto de tipo `Comparador<T>`. Es una interfaz funcional, pero no forma parte de la API `java.util.function`. Esto no es un problema. Es posible usarla de la misma manera.

La interfaz `Comparator<T>` tiene un método abstracto `int compare(T o1, T o2)`. Este método devuelve un entero que permite indicar si `o1` es mayor que `o2` (entero positivo), si `o1` es menor que `o2` (entero negativo) o si `o1` es igual a `o2` (entero 0).

El método `sort` utiliza un algoritmo de ordenación. Solo le falta la forma de comparar dos elementos entre sí. Este es el objetivo de la interfaz `Comparator<T>`.

Para ordenar la lista alfabéticamente, puede escribir el siguiente código:

```
System.out.println();
System.out.println("Ejemplo 1:");
lista.sort((String s1, String s2)->{return s1.compareTo(s2);});
lista.forEach(s->System.out.println(s));

System.out.println();
System.out.println("Ejemplo 2:");
//Sintaxis más condensada como una única instrucción
lista.sort((s1, s2)->s1.compareTo(s2));
lista.forEach(s->System.out.println(s));
```


Manipulación de las colecciones

1. La API Stream

La aparición de expresiones lambda ha permitido la creación o el enriquecimiento de un cierto número de API. La más característica es la API *Stream*, que permite la manipulación de flujos de elementos. Esta API simplifica el manejo de colecciones. El desarrollador se concentra en las operaciones que se llevarán a cabo, sin preocuparse por cómo recorrer la colección.

Esta sección tiene como objetivo presentar el funcionamiento de esta API, para ser autónomo al final de este capítulo.

2. Teoría

En general, las colecciones se utilizan con objeto de almacenar información para que se pueda recuperar o realizar operaciones con ella. La técnica clásica para realizar operaciones en los elementos contenidos en una colección consiste en hacer un bucle *for* o usar el iterador asociado con la colección. Esta solución puede ser reemplazada ventajosamente por el uso de *streams* y *pipelines*. Para ilustrar el funcionamiento de estos dos elementos, podemos hacer el paralelismo con el funcionamiento de una fábrica.

Nuestra fábrica recibe la materia prima, que pasa a través de diferentes unidades de procesamiento, para obtener el producto terminado como salida. En nuestro caso, los datos presentes en la colección representan la materia prima. Estos se transportan a través de objetos que generalmente implementan la interfaz `Stream`. Las unidades de procesamiento están representadas por los pipelines. Estos pueden producir un resultado directamente explotable (el producto terminado) u otro objeto `Stream`, que alimentará un

nuevo pipeline.

Todas las clases que implementan la interfaz `Collection` son capaces de proporcionar un objeto de tipo `Stream`. Este objeto es el responsable de la iteración sobre los elementos presentes en la colección.

Los pipelines son un poco más complejos, ya que están formados por varios elementos.

El primero representa la fuente de la cual el pipeline obtendrá la información sobre la que llevará a cabo el procesamiento o los procesamientos. Por supuesto, es un objeto `Stream` que será la fuente de un pipeline. Las operaciones están representadas por los diferentes métodos definidos en la interfaz `Stream`. Estos métodos se corresponden con los diferentes tipos de procesamiento que se pueden realizar sobre los datos. Por defecto, estos métodos no hacen nada. Su comportamiento se debe adaptar según el tipo de objeto que se ha de tratar. Para esto, generalmente se usa una expresión lambda o referencia de método. Estos métodos pueden generar el resultado final, otro objeto `Stream` compuesto por objetos del mismo tipo que los objetos originales u otro objeto `Stream` que contenga otro tipo de datos.

Las siguientes secciones presentan cómo usar estos objetos de tipo `Stream`, sin ser exhaustivos en todas las funcionalidades disponibles. La javadoc de la API *Stream* está disponible en la siguiente dirección:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html>

3. Obtener un Stream

a. Obtener un Stream genérico

Para usar esta API, primero debe obtener un objeto de tipo `Stream<T>` de una colección. Esto es posible gracias al método `stream` (o `parallelStream` para permitir el procesamiento paralelo) de la interfaz `Collection<T>`.

El siguiente ejemplo muestra cómo obtener un objeto de tipo `Stream <Persona>`:

```
List<Persona> listaPersonas = new ArrayList<Persona>(List.of(
    new Persona("Wayne",
```

```

        "John",
        LocalDate.of(1907, 5, 26)),

    new Persona("McQueen",
        "Steeve",
        LocalDate.of(1930, 3, 24)),

    new Persona("Lennon",
        "John",
        LocalDate.of(1940, 10, 9)),

    new Persona("Gibson",
        "Mel",
        LocalDate.of(1956, 1, 3)),
    new Persona("Willis",
        "Bruce",
        LocalDate.of(1955, 3, 19)))
);
Stream<Persona> streamPersonas = listaPersonas.stream();

```

En ausencia de una colección disponible directamente, es posible crear un objeto de tipo `Stream` a través de un *builder* (*Design Pattern Builder*) del `Stream<T>.Builder`:

```

Stream<String> streamStrings = Stream.<String>builder()
    .add("Hola")
    .add("todo")
    .add("el")
    .add("mundo")
    .build();

```

Para obtener un *builder*, simplemente use el método estático y genérico `builder` de la interfaz `Stream`. Cuando el *builder* esté disponible, solo use el método `add` tantas veces como sea necesario para alimentar el futuro `Stream`, antes de construirlo con el método `build`.

Para que estas llamadas sucesivas funcionen, el método `add` devuelve el objeto `Stream<T>.builder`.

b. Obtener un Stream numérico

El siguiente ejemplo muestra la obtención de un objeto de tipo `Stream<Integer>`:

```
List<Integer> listaEnteros = List.of(1, 15, 13, 5, 48, 6, 89, 25, 46, 35);  
Stream<Integer> streamEnteros = listaEnteros.stream();
```

Al recuperar un `Stream` numérico, el objetivo suele ser realizar operaciones matemáticas en todos los elementos. Por lo tanto, es preferible obtener un `Stream` especializado como `IntStream`, `LongStream` o `DoubleStream`. Estas tres clases proporcionan un conjunto de métodos matemáticos muy útiles.

Para obtener un `Stream` numérico, debe pasar por:

- Un *builder* (`IntStream.Builder`, `LongStream.Builder`, `DoubleStream.Builder`).

```
IntStream streamNumerosEnteros = IntStream.builder().add(1)  
                                                    .add(15)  
                                                    .build();
```

- Una secuencia genérica en la que se llama al método `mapToInt`, `mapToLong` o `mapToDouble`. Estos métodos esperan respectivamente un parámetro de tipo `ToIntFunction<T>`, `ToLongFunction<T>` y `ToDoubleFunction<T>`. Estas funciones esperan un parámetro del mismo tipo que los elementos de la lista y devuelven respectivamente un `int`, un `long` y un `double`.

```
List<Integer> listaEnteros = List.of(1, 15, 13, 5, 48, 6, 89, 25, 46, 35);  
IntStream streamNumerosEnteros =  
    listaEnteros.stream().mapToInt(i->i);
```

Puede parecer complicado obtener un objeto de tipo `Stream<Integer>` desde un objeto de tipo `List<Integer>`, pero este mecanismo simplifica enormemente las

cosas tan pronto como se desea recuperar información numérica a partir de objetos complejos. Imagine que queremos calcular la edad promedio de las personas en la lista `listaPersona`, declarada anteriormente. Para hacer esto, primero debemos recuperar todas las edades. Es posible hacerlo en forma de un objeto de tipo `IntStream`:

```
IntStream streamEdadesPersonas =  
    listaPersonas.stream (). mapToInt (p-> p.calcularEdad ());
```

El método `mapToInt` toma como parámetro una expresión lambda, que devuelve la edad de la persona que se itera. Este simple ejemplo muestra la potencia de los métodos `mapToX`.

Esta sección ha presentado las principales formas de obtener un `Stream`. Leer la javadoc le permitirá descubrir métodos adicionales para obtener un `Stream`.

Ahora es el momento de descubrir las posibilidades de cálculos y manipulaciones que ofrecen estos objetos.

4. Utilizar un Stream

a. Utilizar un Stream genérico

Filtrar

Es posible filtrar la colección utilizando el método `filter`. El método `collect`, por su parte, transforma el `Stream` devuelto por el método `filter` en `List<String>`, para finalizar el procesamiento. Las llamadas se realizan sucesivamente como se muestra en el siguiente ejemplo:

```
List<String> palabras5CaracteresMin =  
    streamStrings.filter(s->s.length()>=5)  
                .collect(Collectors.toList());
```

El método de `filter` espera como argumento un objeto de tipo `Predicate<T>`, que permite definir la regla del filtro y devuelve un objeto de tipo `Stream<T>`, con los elementos que respetan este filtro. A medida que el método devuelve un `Stream<T>`, es

posible realizar otra operación con calma. El método `collect` espera como argumento un objeto de tipo `Collector<String>` y devuelve un objeto adaptado de acuerdo con el `Collector` definido. En el ejemplo, el `Collector` elegido se usa para generar una `List<String>`. El método `collect` es muy interesante y permite, según el parámetro proporcionado, finalizar un procesamiento haciendo posible, en particular, transformar el `Stream` final en una lista (`Collectors.toList()`), en un conjunto (`Collectors.toSet()`) o en *mapa* (`Collectors.toMap()`).

- ❑ Después de un conjunto de tratamientos, el stream original ya no es utilizable. Se produce un error si intentamos reutilizarlo:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
    at java.base/java.util.stream.ReferencePipeline.reduce(ReferencePipeline.java:558)
    at Proyecto_JavaSE_Cap4/es.eni.editions.lambdas.TestStream.main(TestStream.java:52)
```

Ordenar

Es posible enriquecer el ejemplo anterior ordenando el resultado. Para esto, está disponible el método `sorted`. Devuelve un `Stream<T>` ordenado de forma natural (orden alfabético para cadenas). Hay una sobrecarga esperando un objeto de tipo `Comparator<T>`, como el método `sort` de una `List<T>`. El siguiente ejemplo enriquece el ejemplo anterior con la implementación de una clasificación natural antes de la constitución de la lista resultante:

```
List<String> palabras5CaracteresMin =
    streamStrings.filter(s->s.length()>=5)
                .sorted()
                .collect(Collectors.toList());
```

Otras operaciones disponibles

El objetivo no es hacer una presentación exhaustiva de los métodos disponibles. Aquí hay una muestra representativa de métodos útiles:

- Los métodos `allMatch`, `anyMatch` y `noneMatch` permiten verificar respectivamente si todos los elementos del `Stream` al menos un elemento del `Stream` o ningún elemento del `Stream`, respetan el predicado que se pasa como

argumento.

- El método `count` le permite saber la cantidad de elementos.
- El método `distinct` permite obtener un `Stream` que contiene solo los elementos diferentes del `Stream` original, de acuerdo con el método `equals` de los objetos contenidos en la `Stream`.
- El método `limit` trunca la secuencia original para no exceder el tamaño proporcionado como parámetro.
- Los métodos `max` y `min` permiten determinar el elemento más grande y el más pequeño respectivamente, según el parámetro `Comparador<T>` que se pasa como argumento. El valor devuelto es un `Optional<T>` porque es posible que no se devuelva ningún elemento si no existe. Una sección dedicada a esta clase está disponible un poco más adelante en el capítulo.
- Ejemplos avanzados con un objeto de tipo `Stream<Persona>`:

```
// la función allMatch devuelve true si todos los elementos
// de la lista cumplen la condición expresada
// en la expresión lambda -
```

```
if(listaPersonas.stream().allMatch(
    p->p.getFechaDeNacimiento ().getYear()>1945))
    System.out.println("todas las personas nacieron
                        después de 1945");
```

```
else
    System.out.println("algunas personas nacieron
                        antes de 1945");
```

```
// filtro de personas nacidas en el mes de marzo
// este filtro genera un nuevo Stream
// forEach recorre este nuevo Stream y
// ejecuta la expresión lambda para cada elemento
```

```
listaPersonas.stream().filter(
    p->p.getFechaDeNacimiento ().getMesValue()==3)
    .forEach(
    p->System.out.println(p.getNombre()));
```

```

// búsqueda de la persona con más edad de la lista
// la expresión lambda representa la implementación
// de la interfaz Comparator

System.out.println(listaPersonas.stream().max(
    (pe1,pe2)-> pe1.calcularEdad ()-pe2.calcularEdad ())
    .get()
    .getNombre());

// cálculo de la edad media de las personas presentes en la
// lista mapToLong genera un nuevo Stream de tipo long
// obtenido a partir de la referencia de función
// average calcula la media de este nuevo flujo
// getAsDouble la tranforma en tipo double
double edadMedia = listaPersonas.stream().mapToLong(
    p->p.calcularEdad ())
    .average()
    .getAsDouble();

System.out.println("edad media de las personas de la lista: " +
    edadMedia + " años");

```

Los métodos destacables en este nuevo ejemplo son:

- El método `get`. Este método está disponible en el `Optional<T>` que devuelve el método `max`. Da acceso al elemento eventualmente devuelto por el método `max`.
- El método `mapToLong` le permite generar un `LongStream` a partir de un `Stream<T>`. En el ejemplo, le permite generar un `LongStream` a partir de las edades de las diferentes personas en `Stream`. Existen métodos equivalentes para los tipos `Integer` (`mapToInt`) y `Double` (`mapToDouble`).

La siguiente sección presenta el uso de los streams numéricos.

El mapeo

La función `map` es una función más general que las funciones `mapToXxx` presentadas anteriormente. Le permiten generar un nuevo `Stream` con nuevos objetos de

un Stream existente. El siguiente ejemplo propone crear una colección de clientes a partir de una colección de personas:

```
List<Cliente> listaClientes =
    listaPersonas.stream().map(
        p->new Cliente(p.getNombre(),
            p.getApellido(),
            p.getFechaDeNacimiento()))
    .collect(Collectors.toList());
```

b. Utilizar un Stream numérico

Los streams numéricos ofrecen métodos complementarios útiles en la manipulación de una colección de números. A continuación se muestran varios ejemplos de aplicación:

```
System.out.println("Suma de números");
int suma = listaEnteros.stream().mapToInt(i->i)
    .sum();
System.out.println("La suma es de " + suma);

System.out.println("Media de los números");
OptionalDouble media = listaEnteros.stream().mapToInt(i->i)
    .average();
System.out.println("La media es de " + media);

System.out.println("El valor máximo es:");
OptionalInt optMax = listaEnteros.stream().mapToInt(i->i)
    .max();

if(optMax.isPresent())
{
    System.out.println(optMax.getAsInt());
}
else
{
    System.out.println("No hay valor máximo ");
}
System.out.println("Encontrar el valor máximo o dar un valor
por defecto:");
```

```
int valMax = listaEnteros.stream().mapToInt(i->i)
                                .max()
                                .orElse(123);
```

Los métodos de `sum`, `average` y `max` le permiten conocer la suma, el promedio de los valores y el valor máximo. El método `average` devuelve un objeto de tipo `OptionalDouble` y el método `max` devuelve un objeto de tipo `OptionalInt` porque, en ausencia de valores en el `Stream`, no hay resultado para estas dos operaciones. Los objetos de tipo `optionalXxx` permiten que el resultado se maneje de manera segura. En el ejemplo, el método `isPresent` hace posible verificar la presencia de un valor, el método `getAsXxx` permite recuperar el resultado y el método `orElse` hace posible definir un valor predeterminado. La siguiente sección trata de la clase genérica `optional<T>`, pero el funcionamiento de las clases `OptionalInt`, `OptionalLong` y `OptionalDouble` esencialmente es el mismo.

5. La clase `Optional<T>`

La clase `optional<T>` se usa en la API `Stream` como el resultado final de varias operaciones. De hecho, la manipulación de un `Stream` a veces provoca la selección o creación de un solo elemento: cuando se busca el elemento más grande, el elemento más pequeño, el promedio, etc. Puede ser que estos tratamientos no devuelvan ningún resultado.

Para proteger el programa (... y al desarrollador) de un `NullPointerException`, los resultados se encapsulan en un objeto de tipo `optional<T>`. Por lo tanto, este objeto proporciona un conjunto de métodos para manejar de manera segura el resultado:

- Los métodos `isPresent` e `isEmpty` permiten determinar la presencia de un resultado.
- El método `get` proporciona el resultado. Este método se debe utilizar si se ha verificado la presencia de un resultado con uno de los dos métodos anteriores.
- El método `orElse` le permite obtener el resultado sin verificar su presencia. Si no hay resultado, se devuelve el valor proporcionado como parámetro del método

`orElse`. Hay un conjunto de variantes disponibles para este método:

- El método `orElseThrow` permite devolver el resultado o generar una excepción de tipo `NoSuchElementException` en ausencia de resultado.
- El método `orElseGet` le permite devolver el resultado o ejecutar la expresión lambda, que se pasa como parámetro a este método. Esta expresión lambda es del tipo `Supplier<T>`. Entonces se devuelve el resultado de esta expresión lambda.
- El método `ifPresent` le permite ejecutar la expresión lambda que se pasa como argumento si hay un resultado. La expresión lambda es de tipo `Consumer<T>`. Este método se puede utilizar, por ejemplo, para formatear el elemento antes de su lectura.

Conclusión

Este capítulo le ha presentado el poder de las expresiones lambda y el valor de algunas API que giran en torno a ellas. Por supuesto, esto es solo una breve descripción. Puede descubrir la riqueza de estas API a medida que las necesite.

Introducción

Hasta ahora, todos los ejemplos de código que hemos realizado funcionan exclusivamente en modo texto. La información se visualiza en una consola y también se informa desde dicha consola. La sencillez de este modo de funcionamiento supone una ventaja innegable para el aprendizaje de un lenguaje. Sin embargo, la mayoría de los usuarios de las futuras aplicaciones seguramente esperan disponer de una interfaz un poco menos austera que una pantalla en modo texto. En este capítulo vamos a estudiar cómo funcionan las interfaces gráficas con Java. Se dará cuenta enseguida de que el diseño de interfaces gráficas en Java no es tan sencillo y requiere escribir muchas líneas de código. En la práctica, contará con varias herramientas de desarrollo, capaces de encargarse de la generación de una gran parte de este código según el diseño gráfico de la aplicación que esté dibujando. Sin embargo, es importante entender correctamente los principios de funcionamiento de este código para intervenir en él y eventualmente optimizarlo. En este capítulo, no emplearemos ninguna herramienta específica.

1. Las bibliotecas gráficas

El lenguaje Java propone dos bibliotecas dedicadas al diseño de interfaces gráficas: la biblioteca AWT y la biblioteca SWING. Los fundamentos de uso son casi idénticos en ambas bibliotecas. El uso simultáneo de las dos bibliotecas en una misma aplicación puede provocar problemas de funcionamiento y por ello debería evitarse. Hay una tercera biblioteca más reciente llamada Java FX. Sin embargo, ya no forma parte de la plataforma estándar Java SE. Por lo tanto, este capítulo no cubrirá esta biblioteca.

a. La biblioteca AWT

Esta biblioteca es la primera disponible para el desarrollo de interfaces gráficas. Contiene

una multitud de clases e interfaces que permiten la definición y la gestión de interfaces gráficas. En realidad, esta biblioteca ^{Material para descargar} utiliza las funcionalidades gráficas del sistema operativo. Por lo tanto, no es el código presente en esta biblioteca el que asegura el resultado gráfico de los diferentes componentes. Este código hace de intermediario con el sistema operativo. Su uso ahorra bastante recursos, pero presenta varios inconvenientes:

- Al estar relacionado el aspecto visual de cada componente con la representación que el sistema operativo hace de él, puede resultar delicado desarrollar una aplicación que tenga una apariencia coherente en todos los sistemas. El tamaño y la posición de los diferentes componentes son los dos elementos que se ven principalmente afectados por este problema.
- Para que esta biblioteca sea compatible con todos los sistemas operativos, los componentes que contiene están limitados a los más corrientes (botones, zonas de texto, listas...).

b. La biblioteca Swing

Esta biblioteca se diseñó para resolver las principales carencias de la biblioteca AWT. Se obtuvo esta mejora al escribir completamente la biblioteca en Java sin apenas recurrir a los servicios del sistema operativo. Únicamente algunos elementos gráficos (ventanas y cuadros de diálogo) siguen relacionados con el sistema operativo. Para los demás componentes, es el código de la biblioteca Swing el encargado de determinar completamente su aspecto y su comportamiento.

La biblioteca Swing contiene por lo tanto una cantidad impresionante de clases que sirven para redefinir los componentes gráficos. Sin embargo, no debemos pensar que la biblioteca Swing convierte la biblioteca AWT en algo completamente obsoleto. De hecho, Swing recupera muchos de los elementos de la biblioteca AWT. En el resto del capítulo, emplearemos esencialmente esta biblioteca.

2. Construcción de la interfaz gráfica de una aplicación

El diseño de la interfaz gráfica de una aplicación se fundamenta ante todo en crear instancias de las clases que representan los diferentes elementos necesarios, modificar las características de estas instancias de clase, agruparlas y prever el código de gestión de los diferentes eventos que pueden intervenir durante el funcionamiento de la aplicación. Así,

una aplicación gráfica está compuesta por una multitud de elementos superpuestos o anidados. Entre estos elementos, uno toma un papel preponderante en la aplicación. Se le suele llamar contenedor de primer nivel. Es el encargado de interactuar con el sistema operativo y agrupar a los demás elementos. Este contenedor de primer nivel no suele contener directamente los componentes gráficos, sino otros contenedores en los cuales están ubicados los componentes gráficos. Para facilitar la disposición de estos elementos entre sí, vamos a utilizar la ayuda de un renderizador. Esta superposición de elementos podría asimilarse a un árbol, en su raíz principal tenemos el contenedor de primer nivel y sus diferentes ramas están constituidas por los demás contenedores. Las hojas del árbol corresponden a los componentes gráficos.

Dado que el contenedor de primer nivel es el elemento indispensable de cualquier aplicación gráfica, empezaremos estudiando con detalle sus características y utilización, para luego pasar a comprender los principales componentes gráficos.

Diseño de una interfaz gráfica

Hemos visto un poco más arriba que cualquier aplicación gráfica se compone de, al menos, un contenedor de primer nivel. La biblioteca Swing dispone de tres clases que permiten llevar a cabo este papel:

`JApplet`: representa una ventana gráfica incluida en una página html para que un navegador se haga cargo de ella. Esta clase ya no es útil en la plataforma Java 11 porque la tecnología de applets ha sido abandonada.

`JWindow`: representa la ventana gráfica más rudimentaria que pueda existir. No dispone de ninguna barra de título, ningún menú de sistema, ningún borde: en realidad es un mero rectángulo. Esta clase se utiliza rara vez, excepto para la visualización de una pantalla de inicio en el momento del arranque de una aplicación (*splash screen*).

`JFrame`: representa una ventana gráfica completa y plenamente funcional. Dispone de una barra de título, de un menú de sistema y de un borde. Puede fácilmente contener un menú y, por supuesto, es el elemento que vamos a emplear en la mayoría de los casos.

1. Las ventanas

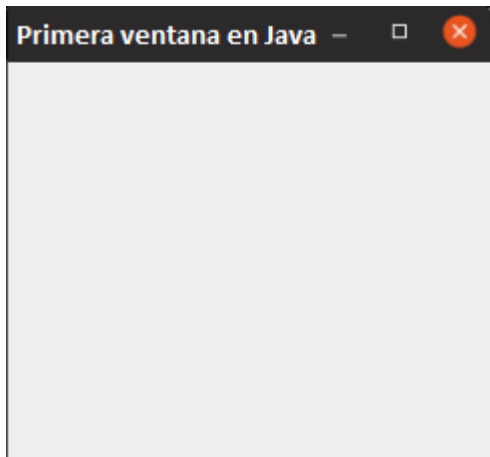
La clase `JFrame` es un elemento indispensable en cualquier aplicación gráfica. Como en el caso de una clase normal, debemos crear una instancia, modificar eventualmente las propiedades y utilizar los métodos. A continuación se muestra el código de la primera aplicación gráfica.

```
package es.eni.editions.ihml;
import javax.swing.JFrame;
public class Main {
    public static void main(String[] args)
```




```
{  
    JFrame ventana;  
    // creación de la instancia de la clase JFrame  
    ventana=new JFrame();  
    // modificación de la posición y del  
    // tamaño de la ventana  
    ventana.setBounds(0,0,300,400);  
    // modificación del título de la ventana  
    ventana.setTitle("Primera ventana en Java");  
    // visualización de la ventana  
    ventana.setVisible(true);  
}
```

y el resultado de su ejecución:



Para que este programa compile y se ejecute con el archivo `module-info.java` en su proyecto, es necesario agregar la referencia al módulo que contiene el paquete `java.swing`. Por lo tanto, la instrucción para agregar es la siguiente:

```
requires java.desktop;
```

Volviendo al programa, cabe admitir que es fácil de usar y muy eficaz. De hecho, es tan eficaz que no se puede parar la aplicación. En efecto, incluso si el usuario cierra la ventana, este cierre no provoca la supresión de la instancia de `JFrame` de la memoria. La única solución para detener la aplicación es apagar la máquina virtual Java con la combinación de teclas [Ctrl] C. Ante esto, se recomienda proporcionar otra solución para detener más fácilmente la ejecución de la aplicación, es decir, junto con el cierre de la ventana.

Una primera solución consiste en gestionar los eventos que se producen en el momento del cierre de la ventana y, en uno de ellos, provocar la detención de la aplicación. Se estudiará esta solución en la sección dedicada a la gestión de los eventos.

La segunda solución utiliza comportamientos predefinidos para el cierre de la ventana. Estos comportamientos están determinados por el método `setDefaultCloseOperation`. Se definen varias constantes para determinar la acción emprendida al cierre de la ventana.

`DISPOSE_ON_CLOSE`: esta opción provoca la detención de la aplicación en el momento del cierre de la última ventana controlada por la máquina virtual.

`DO_NOTHING_ON_CLOSE`: con esta opción, no ocurre nada cuando el usuario pide el cierre de la ventana. En este caso, es obligatorio gestionar los eventos para que la acción del usuario tenga algún efecto sobre la ventana o la aplicación.

`EXIT_ON_CLOSE`: esta opción provoca la detención de la aplicación incluso si otras ventanas siguen visibles.

`HIDE_ON_CLOSE`: con esta opción la ventana simplemente queda oculta como consecuencia de una llamada a su método `setVisible(false)`.

La clase `JFrame` se encuentra al final de una jerarquía de clases bastante importante e implementa numerosas interfaces. Por este motivo, dispone de varios métodos y atributos.

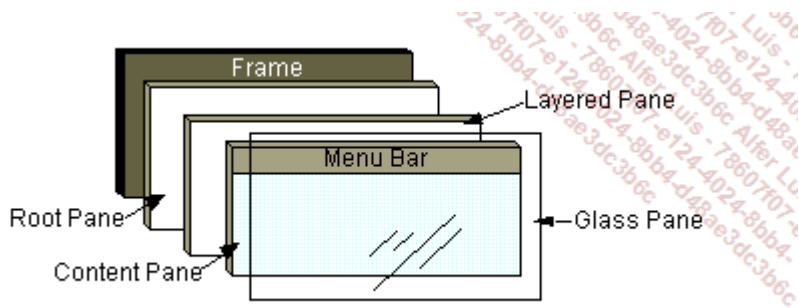


La meta de este libro no es retomar toda la documentación del JDK, de modo que no recorreremos todos los métodos disponibles, sino sencillamente los más utilizados según las necesidades. Sin embargo, puede resultar interesante revisar la documentación antes de realizar el diseño de un método para determinar si lo que queremos diseñar no ha sido ya previsto por los diseñadores de Java. La javadoc del paquete `java.swing` está

disponible en la dirección siguiente:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/package-summary.html>

Ahora que somos capaces de visualizar una ventana, el grueso del trabajo va a consistir en añadirle un contenido. Antes de poder añadir algo a una ventana, conviene entender bien su estructura, que resulta relativamente compleja. Un objeto `JFrame` se compone de varios elementos superpuestos, cada uno con un papel muy específico en la gestión de la ventana.



El elemento `RootPane` corresponde al contenedor de los otros tres elementos. El elemento `LayeredPane` es el responsable de la gestión de la posición de los elementos tanto en los ejes X e Y como en el eje Z, lo que permite la superposición de diferentes elementos. El elemento `ContentPane` es el contenedor básico de todos los elementos añadidos en la ventana. A él vamos a confiarle, por esta razón, los diferentes componentes de la interfaz de la aplicación. Por encima del `ContentPane` se superpone el `GlassPane`, como es posible hacer con un cristal sobre una foto. De hecho, presenta muchas similitudes con el cristal.

- Es transparente por defecto. -
- Lo dibujado en el `GlassPane` esconde los demás elementos.
- Es capaz de interceptar los eventos relacionados con el ratón antes de que estos hayan alcanzado los demás componentes.

De todos estos elementos, es sin duda el `ContentPane` el que vamos a utilizar con mayor frecuencia. Podemos acceder a él a través del método `getContentPane` de la clase `JFrame`. Desde el punto de vista técnico, es posible ubicar componentes directamente en el objeto `ContentPane`, aunque es una práctica que Oracle desaconseja. Se prefiere intercalar un contenedor intermedio que contenga los componentes y ubicarlo en el `ContentPane`. Para ello, se suele utilizar el componente `JPanel`.

Según esto, el escenario clásico de diseño de una interfaz gráfica consiste en crear los diferentes componentes y, a continuación, ubicarlos en un contenedor y, por último, situar este contenedor en el `ContentPane` de la ventana. El ejemplo siguiente lo pone en práctica creando una interfaz usuario compuesta por tres botones.

```
package es.eni.editions.ihm2;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Main {

    public static void main(String[] args)

    {
        // creación de la ventana
        JFrame ventana;
        ventana=new JFrame();
        ventana.setTitle("Primera ventana en Java");
        ventana.setBounds(0,0,300,100);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
        b3=new JButton("Azul");
        // creación del contenedor intermedio
        JPanel panel;
        panel=new JPanel();
        // se agregan los botones en el contenedor intermedio
        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        // se agrega el contenedor intermedio en el ContentPane
        ventana.getContentPane().add(panel);
        // visualización de la ventana
```

```
        ventana.setVisible(true);  
    }  
}
```

Al ejecutarse, este código muestra la ventana siguiente:



La siguiente etapa de nuestro análisis nos va a permitir determinar lo que debe hacer la aplicación cuando el usuario haga clic en alguno de los botones. Pero antes de eso, es necesario comprender cómo se ejecuta una aplicación swing especificando el rol de los thread y más particularmente el thread EDT (*Event Dispatching Thread*).

2. El thread EDT

Cuando se ejecuta una aplicación Java sin una interfaz gráfica, se ejecutan al menos dos threads:

- El thread principal (el *thread main*), cuya función es ejecutar el método principal del programa. Es el punto de entrada obligatorio para ejecutar una aplicación Java SE estándar.
- El thread recolector de basura (*garbage collector*) para limpiar la memoria cuando está ocupada más allá de cierto umbral.

Un thread es simplemente una unidad de ejecución para un programa. Un programa puede estar compuesto por uno o más thread. La ventaja de tener múltiples threads es que puede ejecutar múltiples tareas en paralelo. Tome el ejemplo de un navegador: es muy común hacer una solicitud a un sitio y, mientras espera su descarga y visualización, consultar otra página, ver un vídeo, etc. Todo esto es posible gracias a uso de diferentes threads.

En una máquina multiprocesador, los diferentes threads se pueden ejecutar en paralelo, explotando los diferentes procesadores. En una máquina con un solo procesador, es la frecuencia de transición entre los diferentes threads lo que permite al usuario pensar que las tareas se ejecutan en paralelo. De hecho, el procesador divide su tiempo entre los diferentes threads. Solo ejecuta un thread a la vez, pausando otros threads durante este tiempo. Por lo

tanto, para ejecutar un thread completo, el procesador puede hacer esto varias veces pausando el thread repetidamente para ejecutar los otros threads.

Para ejecutar una aplicación Java con una interfaz gráfica, entra en juego un tercer thread: el **thread EDT** (*Event Dispatching Thread*). Este thread es responsable de administrar la interfaz gráfica y manejar los eventos (como hacer clic en un botón).

En el ejemplo anterior, la interfaz gráfica se creó en el método `main` y, por lo tanto, en el **thread main**. Así, es el thread `main` el «propietario» de los botones presentes en la interfaz gráfica. Si el usuario hace clic en él, se le pide al thread EDT que reaccione a esta acción. Para hacer esto, el thread EDT a menudo necesita acceder a los gráficos. En este caso, es necesaria una operación interthreads. Y aquí es donde pueden surgir los problemas. De hecho, los componentes gráficos no son *thread safe*. Esto significa que acceder desde otro thread posiblemente podría causar un problema.

Para evitar este tipo de situación, es necesario asegurarse de que solo el thread EDT manipule la interfaz gráfica, ya sea para la creación de la interfaz o para la gestión de eventos. Por lo tanto, es necesario que, desde el thread principal que ejecuta el método `main`, se pueda pedir al thread EDT que construya la interfaz gráfica de la aplicación. Esto es posible utilizando el método `invokeLater` de la clase `SwingUtilities`. Este método espera en el parámetro un objeto de tipo `Runnable` que es una interfaz funcional que contiene un único método abstracto cuya firma es la siguiente: `void run()`. Este objeto (o expresión lambda) simplemente define lo que debe ser ejecutado por el thread EDT. El siguiente código realiza una modificación esencial en el ejemplo anterior para garantizar el correcto funcionamiento de la aplicación en todas las situaciones

```
package es.eni.editions.ihm3;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingUtilities;

public class Principal {
    private JFrame ventana;
    //Constructor llamado por el thread EDT - www.detodoprogramacion.org
    Principal (String[] args) {
```

```

        // Construcción del IHM
        // creación de la ventana
        this.ventana=new JFrame();
        this.ventana.setTitle("Primera ventana en Java");
        this.ventana.setBounds(0,0,300,100);
        this.ventana.setDefaultCloseOperation
(JFrame.EXIT_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
        b3=new JButton("Azul");
        // creación del contenedor intermedio
        JPanel panel;
        panel=new JPanel();
        // agregar los botones en el contenedor intermedio
        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        // agregar el contenedor intermedio al ContentPane
        this.ventana.getContentPane().add(panel);
    }

    //Método llamado por el thread EDT
    public void show() {
        // Mostrar el IHM
        this.ventana.setVisible(true);
    }

    //Punto de entrada ejecutado por el thread main
    public static void main(final String[] args) {
        // Programa una tarea para el thread EDT:
        // Creación y visualización de la interfaz gráfica
        /*SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new Principale(args).mostrar();
            }
        });*/
    }

```

```
SwingUtilities.invokeLater(()->
    new Principal(args).mostrar());
}
```

Por supuesto, el resultado de la ejecución es el mismo.

3. La gestión de los eventos

Todos los sistemas operativos que emplean una interfaz gráfica deben vigilar permanentemente los diferentes periféricos de introducción de datos para detectar las acciones del usuario y transmitir las a las diferentes aplicaciones. Para cada acción del usuario, se crea un evento. A continuación, se envían estos eventos a cada aplicación, que determinará si le aplica el evento y determina lo que desea realizar como respuesta. La manera de gestionar estos eventos difiere según los lenguajes. En algunos casos, cada componente dispone de una sección de código predefinida asociada automáticamente a cada tipo de evento. En este caso, el papel del desarrollador consiste en personalizar las diferentes secciones de código asociadas a los eventos. En otros lenguajes, el sistema ubica los eventos en una fila y le corresponde al desarrollador vigilar esta fila para determinar qué componente está afectado por el evento y provocar la ejecución de la sección de código correspondiente. El planteamiento empleado por Java es una técnica intermedia. Java se encarga de determinar qué evento acaba de producirse y sobre qué elemento. El desarrollador es responsable de configurar la sección de código que va a tratar el evento. Desde un punto de vista más técnico, el elemento origen del evento se denomina receptor de evento, y el elemento que contiene la sección de código encargada de gestionar el evento se denomina receptor de evento. Las fuentes de eventos gestionan, para cada evento que pueden activar, una lista que les permite saber qué receptores deben ser avisados si el evento se produce. Por supuesto, las fuentes de eventos y los receptores de eventos son objetos. Es necesario prever qué receptores van a gestionar los eventos que les va a transmitir la fuente de eventos.

Para garantizar esto, a cada tipo de evento le corresponde una interfaz que debe implementar un objeto si quiere ser candidato a gestionar dicho evento. Para evitar la proliferación de interfaces (ya muy numerosas), los eventos se agrupan en categorías. El nombre de estas interfaces siempre respeta la convención siguiente:

La primera parte del nombre representa la categoría de eventos que los objetos que implementan esta interfaz pueden gestionar. El nombre siempre termina en `Listener`.

Por ejemplo, tenemos la interfaz `MouseEventListener`, que corresponde a los eventos activados por los movimientos del ratón, o la interfaz `ActionListener`, que corresponde a un clic en un botón. En cada una de estas interfaces encontramos las firmas de los diferentes métodos asociados a cada evento.

```
public interface MouseEventListener
extends EventListener
{
    void mouseDragged(MouseEvent e);
    void mouseMoved(MouseEvent e);
}
```

Cada uno de estos métodos recibe como argumento un objeto que representa el propio evento. Este objeto se crea automáticamente en el momento de la activación del evento; a continuación, se pasa como argumento al método encargado de gestionar el evento en el receptor de eventos. En general, contiene información adicional relativa al evento y es específico para cada tipo de evento.

Es preciso crear clases que implementen estas interfaces. Desde este punto de vista, tenemos una multitud de posibilidades:

- Crear una clase «normal» que implemente la interfaz.
- Implementar la interfaz en una clase ya existente.
- Crear una clase interna que implemente la interfaz.
- Crear una clase interna anónima que implemente la interfaz.
- Crear eventualmente una expresión lambda si la interfaz es una interfaz funcional.

En algunos casos, quizá sea necesario no gestionar todos los eventos presentes en la interfaz. Sin embargo, es obligatorio escribir todos los métodos exigidos por la interfaz incluso si varios de ellos no contienen ningún código. Esto puede perjudicar la legibilidad del código. Para paliar este problema, Java proporciona para casi cada interfaz `XxxListener`, una clase abstracta correspondiente que implementa la interfaz y que contiene los métodos exigidos por esta. Estos métodos no contienen código alguno, ya que el tratamiento de cada evento debe ser específico a cada aplicación. Estas clases emplean la

misma nomenclatura que las interfaces, excepto que se sustituye `Listener` por `Adapter`. Tenemos por ejemplo la clase `MouseMotionAdapter`, que implementa la interfaz `MouseMotionListener`. Se pueden utilizar estas clases de varias maneras:

- Creando una clase «normal» que herede de una de estas clases.
- Creando una clase interna que herede de una de estas clases.
- Creando una clase interna anónima que herede de una de estas clases.

El uso de una clase interna anónima es la solución que más se utiliza, con el pequeño inconveniente de que se obtiene una sintaxis difícil de leer si uno no está acostumbrado.

Para aclarar todo esto vamos a ilustrar cada una de estas posibilidades con un pequeño ejemplo que nos va a permitir terminar correctamente la aplicación en el momento del cierre de la ventana principal al invocar al método `System.exit(0)`. Esta solución permite realizar verificaciones antes de detener la aplicación (copia de seguridad, mostrar un mensaje de confirmación, desconectar al usuario...). Es preciso, en este caso, modificar la propiedad `DefaultCloseOperation` de la ventana con el valor `DO_NOTHING_ON_CLOSE` para que no tenga una acción por defecto.

Se debe invocar este método durante la detección del cierre de la ventana. Para ello, debemos gestionar los eventos relacionados con la ventana y, en particular, el evento `windowClosing`, que se produce cuando el usuario cierra la ventana mediante el menú del sistema. La interfaz `WindowListener` está perfectamente adaptada para este tipo de trabajo.

La base de nuestro trabajo se compone de las dos clases siguientes:

```
package es.eni.editions.ihm4;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla
extends JFrame
{
    public Pantalla()
    {
```

```

        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
        b3=new JButton("Azul");
        // creación del contenedor intermedio
        JPanel panel;
        panel=new JPanel();
        // se agregan los botones en el contenedor intermedio
        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        // se agrega el contenedor en el ContentPane
        getContentPane().add(panel);
    }
    public void mostrar()
    {
        this.setVisible(true);
    }
}

package es.eni.editions.ihm4;

import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->new Pantalla().mostrar());
    }
}

```

Tenga en cuenta que la interfaz gráfica está definida por una clase derivada de la clase `JFrame`. Por lo tanto, está completamente separada de la clase que contiene el método

main.

Si ejecutamos este código, la ventana aparece, pero ya no es posible cerrarla y aún menos detener la aplicación. Veamos ahora cómo remediar este problema con las diferentes soluciones mencionadas más arriba.

Usar una clase «normal» que implementa la interfaz

```
package es.eni.editions.ihm5;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

public class EscuchadorVentana implements WindowListener {

    public void windowActivated(WindowEvent arg0)
    {
    }
    public void windowClosed(WindowEvent arg0)
    {
    }
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0)
    {
    }
    public void windowDeiconified(WindowEvent arg0)
    {
    }
    public void windowIconified(WindowEvent arg0)
    {
    }
    public void windowOpened(WindowEvent arg0)
    {
    }
}
```

```
package es.eni.editions.ihm5;

import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->
        {
            Pantalla pantalla=new Pantalla();
            // creación de una instancia de la clase encargada
            // de gestionar los eventos
            EscuchadorVentana ev =new EscuchadorVentana();
            // referencia a esta instancia de clase
            // como receptor de eventos para la ventana
            pantalla.addWindowListener(ev);
            // visualización de la pantalla
            pantalla.mostrar();
        });
    }
}
```

Implementar la interfaz en una clase ya existente

En esta solución, vamos a delegar a la clase que representa la ventana la tarea de gestionar sus propios eventos al hacerle implementar la interfaz `WindowListener`.

```
package es.eni.editions.ihm6;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

```
public class Pantalla extends JFrame
                                implements WindowListener

{
    public Pantalla()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
        b3=new JButton("Azul");
        // creación del contenedor intermedio
        JPanel panel;
        panel=new JPanel();
        // se agregan los botones en el contenedor intermedio
        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        // se agrega el contenedor intermedio en el ContentPane
        getContentPane().add(panel);
        // referencia a la propia ventana
        // como receptor de sus propios eventos

        addWindowListener(this);
    }
    public void mostrar()
    {
        this.setVisible(true);
    }
    public void windowActivated(WindowEvent arg0)
    {
    }
    public void windowClosed(WindowEvent arg0)
    {
    }
}
```

```

    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0)
    {
    }
    public void windowDeiconified(WindowEvent arg0)
    {
    }
    public void windowIconified(WindowEvent arg0)
    {
    }
    public void windowOpened(WindowEvent arg0)
    {
    }
}

package es.eni.editions.ihm6;

import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->new Pantalla().mostrar());
    }
}

```

Con esta solución, el código se centraliza en una única clase. Si hay que gestionar varios eventos, esta clase va a contener una cantidad excesiva de métodos.

Crear una clase interna que implemente la interfaz

Esta solución es una mezcla de las dos anteriores, ya que tenemos una clase específica para la gestión de los eventos, pero definida en el interior de la clase que corresponde a la ventana.

```
package es.eni.editions.ihm7;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    public Pantalla()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
        b3=new JButton("Azul");
        // creación del contenedor intermedio
        JPanel panel;
        panel=new JPanel();
        // se agregan los botones en el contenedor intermedio
        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        // se agrega el contenedor intermedio
        // en el ContentPane
        getContentPane().add(panel);
        // creación de una instancia de la clase encargada
        // de gestionar los eventos
        EscuchadorVentana ev;
        ev=new EscuchadorVentana();
        // referencia a esta instancia de clase
        // como receptor de eventos para la ventana
    }
}
```



```
        addWindowListener(ev);
    }

    public void mostrar()
    {
        this.setVisible(true);
    }

    public class EscuchadorVentana implements WindowListener
    {
        public void windowActivated(WindowEvent arg0)
        {
        }
        public void windowClosed(WindowEvent arg0)
        {
        }
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
        public void windowDeactivated(WindowEvent arg0)
        {
        }
        public void windowDeiconified(WindowEvent arg0)
        {
        }
        public void windowIconified(WindowEvent arg0)
        {
        }
        public void windowOpened(WindowEvent arg0)
        {
        }
    }
}

package es.eni.editions.ihm7;

import javax.swing.SwingUtilities;
```

```

public class Main {

    public static void main(String[] args)

    {
        SwingUtilities.invokeLater(()->new Pantalla().mostrar());
    }
}

```

Con esta solución se reparte la responsabilidad entre varias clases, pero, a cambio, multiplicamos el número de clases.

Crear una clase interna anónima que implemente la interfaz

Esta solución es una ligera variante de la anterior, ya que seguimos teniendo una clase específica encargada de la gestión de los eventos, pero declarada en el momento de su instanciación.

```

package es.eni.editions.ihm8;

import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    public Pantalla()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
    }
}

```

```
b3=new JButton("Azul");
// creación del contenedor intermedio
JPanel panel;
panel=new JPanel();
// se agregan los botones en el contenedor intermedio
panel.add(b1);
panel.add(b2);
panel.add(b3);
// se agrega el contenedor intermedio
// en el ContentPane
getContentPane().add(panel);
// creación de una instancia de una clase anónima
// encargada de gestionar los eventos
addWindowListener(new WindowListener()
// principio de la definición de la clase
{
    public void windowActivated(WindowEvent arg0)
    {
    }
    public void windowClosed(WindowEvent arg0)
    {
    }
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
    public void windowDeactivated(WindowEvent arg0)
    {
    }
    public void windowDeiconified(WindowEvent arg0)
    {
    }
    public void windowIconified(WindowEvent arg0)
    {
    }
    public void windowOpened(WindowEvent arg0)
    {
    }
}
```

```

        } // fin de la definición de la clase
    ); // fin de la llamada del método addWindowListener
} // fin del constructor
public void mostrar()
{
    this.setVisible(true);
}
} // fin de la clase Pantalla

package es.eni.editions.ihm8;

import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->new Pantalla().mostrar());
    }
}

```

El único inconveniente que presenta esta solución reside en la relativa complejidad de su sintaxis. Los comentarios entre las diferentes líneas ofrecen una ayuda valiosa para no perderse entre llaves y paréntesis.

También existe una problemática general que se le puede reprochar a todas estas soluciones: para un único método realmente útil, tenemos que escribir siete.

Para evitar este código inútil, podemos trabajar con una clase que implemente ya la interfaz correcta y volver a definir únicamente los métodos que nos interesan.

Crear una clase «normal» que herede de una clase XxxAdapter

```

package es.eni.editions.ihm9;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

public class EscuchadorVentana extends WindowAdapter

```

```

{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
}

package es.eni.editions.ihm9;

import javax.swing.SwingUtilities;
public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->
        {
            Pantalla pantalla=new Pantalla();
            // creación de una instancia de la clase encargada
            // de gestionar los eventos - www.detodoprogramacion.org
            EscuchadorVentana ev=new EscuchadorVentana();
            // referencia a esta instancia de clase
            // como receptor de eventos para la ventana
            pantalla.addWindowListener(ev);
            // visualización de la pantalla
            pantalla.mostrar();
        });
    }
}

```

Crear una clase interna que herede de una clase XxxAdapter

```

package es.eni.editions.ihm10;

import java.awt.event.WindowEvent;
import java.awt.event.WindowAdapter;

```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    public Pantalla()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        JButton b1,b2,b3;
        b1=new JButton("Rojo");
        b2=new JButton("Verde");
        b3=new JButton("Azul");
        // creación del contenedor intermedio
        JPanel panel;
        panel=new JPanel();
        // se agregan los botones en el contenedor intermedio
        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        // se agrega el contenedor intermedio
        // en el ContentPane
        getContentPane().add(panel);
        // creación de una instancia de la clase encargada
        // de gestionar los eventos
        EscuchadorVentana ev;
        ev=new EscuchadorVentana();
        // como receptor de eventos para la ventana

        addWindowListener(ev);
    }
    public void mostrar()
    {
        this.setVisible(true);
    }
}
```

```

    }

    public class EscuchadorVentana extends WindowAdapter
    {
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
    }
}

package es.eni.editions.ihm10;

import javax.swing.SwingUtilities;

public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->new Pantalla().mostrar());
    }
}

```

Crear una clase interna anónima que herede de una clase XxxAdapter

```

package es.eni.editions.ihm11;

import java.awt.event.WindowEvent;
import java.awt.event.WindowAdapter;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    public Pantalla()

```

```
{
    setTitle("Primera ventana en Java");
    setBounds(0,0,300,100);
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    // creación de los tres botones
    JButton b1,b2,b3;
    b1=new JButton("Rojo");
    b2=new JButton("Verde");
    b3=new JButton("Azul");
    // creación del contenedor intermedio
    JPanel panel;
    panel=new JPanel();
    // se agregan los botones en el contenedor intermedio
    panel.add(b1);
    panel.add(b2);
    panel.add(b3);
    // se agrega el contenedor intermedio
    // en el ContentPane
    getContentPane().add(panel);
    // creación de una instancia de una clase anónima
    // encargada de gestionar los eventos
    addWindowListener(new WindowAdapter()
    // principio de la definición de la clase
    {
        public void windowClosing(WindowEvent arg0)
        {
            System.exit(0);
        }
    } // fin de la definición de la clase
    ); // fin de la llamada al método addWindowListener
} // fin del constructor

public void mostrar()
{
    this.setVisible(true);
}

} // fin de la clase Pantalla
```



```
package es.eni.editions.ihm11;

import javax.swing.SwingUtilities

public class Main {

    public static void main(String[] args)
    {
        SwingUtilities.invokeLater(()->new Pantalla().mostrar());
    }
}
```

Por supuesto, esta solución es la más económica en número de líneas; además, es utilizada por numerosas herramientas de desarrollo que generan automáticamente código. La relativa complejidad del código puede intimidar ligeramente cuando uno no está acostumbrado a ello.

Hasta ahora, tenemos una fuente de eventos y un receptor para esta fuente de eventos. En algunos casos, podemos estar en la situación de tener varias fuentes de eventos y desear utilizar el mismo receptor o tener una fuente de evento y avisar a varios receptores. La situación clásica en la cual tenemos varias fuentes de eventos y un único receptor se da cuando proporcionamos al usuario varias soluciones para lanzar la ejecución de una misma acción (menú y barra de herramientas o botones).

Sea cual sea el medio utilizado para lanzar la acción, el código que se ha de ejecutar sigue siendo el mismo. En este supuesto, podemos emplear el mismo receptor para las dos fuentes de eventos. Para ilustrar esto, vamos a añadir un menú a la aplicación y hacer que el uso del menú o de uno de los botones ejecute la misma acción modificando el color de fondo correspondiente al botón o al menú usado. Como debemos utilizar el mismo receptor para dos fuentes de eventos, es preferible usar una clase interna para la creación del receptor.

A continuación, presentamos el código correspondiente.

```
package es.eni.editions.ihm12;

import java.awt.Color;
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    JPanel panel;
    public Pantalla ()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        JButton btnRojo,btnVerde,btnAzul;
        btnRojo=new JButton("Rojo");
        btnVerde=new JButton("Verde");
        btnAzul=new JButton("Azul");
        // creación de los tres receptores de eventos
        EscuchadorRojo escR;
        EscuchadorVerde escV;
        EscuchadorAzul escA;
        escR=new EscuchadorRojo();
        escV=new EscuchadorVerde();
        escA=new EscuchadorAzul();
        // asociación de cada receptor con su botón
        btnRojo.addActionListener(escR);
        btnVerde.addActionListener(escV);
        btnAzul.addActionListener(escA);
        // Creación del menú
        JMenuBar barraMenu;
        barraMenu=new JMenuBar();
```

```

JMenu mnuColores;
mnuColores=new JMenu("Colores");
barraMenu.add(mnuColores);
JMenuItem mnuRojo,mnuVerde,mnuAzul;
mnuRojo=new JMenuItem("Rojo");
mnuVerde=new JMenuItem("Verde");
mnuAzul=new JMenuItem("Azul");
mnuColores.add(mnuRojo);
mnuColores.add(mnuVerde);
mnuColores.add(mnuAzul);
// asociación de cada receptor con su elemento de menú
// (los mismos que para los botones)
mnuRojo.addActionListener(escR);
mnuVerde.addActionListener(escV);
mnuAzul.addActionListener(escA);
// se agrega el menú a la ventana
setJMenuBar(barraMenu);
// creación del contenedor intermedio
panel=new JPanel();
// se agregan los botones en el contenedor intermedio
panel.add(btnRojo);
panel.add(btnVerde);
panel.add(btnAzul);
// se agrega el contenedor intermedio
// en el ContentPane
getContentPane().add(panel);
// creación de una instancia de una clase anónima
// encargada de gestionar los eventos de la ventana
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}
public void mostrar()

```

```

    {
        this.setVisible(true);
    }
public class EscuchadorRojo implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        panel.setBackground(Color.RED);
    }
}
public class EscuchadorVerde implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        panel.setBackground(Color.GREEN);
    }
}
public class EscuchadorAzul implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        panel.setBackground(Color.BLUE);
    }
}
}

```

En este código, tenemos nuestras tres clases receptoras de eventos, que son muy similares. Con un pequeño truco vamos a poder simplificar el código para obtener una única clase receptora de eventos para los tres botones. Se invocará el mismo método `actionPerformed` con un clic en cualquiera de los botones. La elección de la acción que se ha de ejecutar se realizará en el interior de este método. Para ello, vamos a utilizar el parámetro `ActionEvent` facilitado al propio método. Permite obtener una referencia sobre el objeto que origina el evento a través del método `getSource`. A continuación se presenta el código simplificado:

```
package es.eni.editions.ihm13;
```

```
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    JPanel panel;
    JButton btnRojo, btnVerde, btnAzul;
    JMenuItem mnuRojo, mnuVerde, mnuAzul;
    public Pantalla ()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones
        btnRojo=new JButton("Rojo");
        btnVerde=new JButton("Verde");
        btnAzul=new JButton("Azul");
        // creación de un solo receptor de eventos
        ReceptorColor ec;
        ec=new ReceptorColor();
        // asociación de cada receptor de eventos con su botón
        btnRojo.addActionListener(ec);
        btnVerde.addActionListener(ec);
        btnAzul.addActionListener(ec);
        // Creación del menú
        JMenuBar barraMenu;
        barraMenu=new JMenuBar();
```

```
JMenu mnuColores;
mnuColores=new JMenu("Colores");
barraMenu.add(mnuColores);

mnuRojo=new JMenuItem("Rojo");
mnuVerde=new JMenuItem("Verde");
mnuAzul=new JMenuItem("Azul");
mnuColores.add(mnuRojo);
mnuColores.add(mnuVerde);
mnuColores.add(mnuAzul);
// asociación de cada receptor de eventos con su elemento
// de menú (el mismo que para los botones)
mnuRojo.addActionListener(ec);
mnuVerde.addActionListener(ec);
mnuAzul.addActionListener(ec);
// se agrega el menú en la ventana
setJMenuBar(barraMenu);
// creación del contenedor intermedio
panel=new JPanel();
// se agregan los botones en el contenedor intermedio
panel.add(btnRojo);
panel.add(btnVerde);
panel.add(btnAzul);
// se agrega el contenedor intermedio en el ContentPane
getContentPane().add(panel);
// creación de una instancia de una clase anónima
// encargada de gestionar los eventos
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}

public void mostrar()
{
```

```

        this.setVisible(true);
    }
    public class ReceptorColor implements ActionListener
    {
        public void actionPerformed(ActionEvent arg0)
        {
            if (arg0.getSource()==btnRojo | arg0.getSource()==mnuRojo)
            {
                panel.setBackground(Color.RED);
            }
            if (arg0.getSource()==btnVerde | arg0.getSource()==mnuVerde)
            {
                panel.setBackground(Color.GREEN);
            }
            if (arg0.getSource()==btnAzul | arg0.getSource()==mnuAzul)
            {
                panel.setBackground(Color.BLUE);
            }
        }
    }
}

```

Hay que señalar que para que funcione esta solución, los objetos fuente de eventos deben ser accesibles desde la clase receptor de eventos. Por lo tanto, la declaración de los botones y de los elementos de menú se realiza dentro de la propia clase y no dentro del constructor, como era el caso en la versión anterior. Esta solución es posible únicamente si la clase receptor de eventos es una clase interna. En el caso de que la clase receptor de eventos sea independiente de la clase donde se crean los objetos fuente de eventos, hay que revisar el código del método `actionPerformed`. El parámetro `ActionEvent` del método `actionPerformed` nos proporciona otra solución para resolver este problema. A través del método `getActionCommand` tenemos acceso a una cadena de caracteres que representa al objeto fuente de eventos.

Por defecto, esta cadena de caracteres corresponde al título del componente que produce el evento, pero se la puede modificar con el método `setActionCommand` de cada componente. De hecho, se recomienda esta práctica, ya que nos permite obtener un código

idéntico para una aplicación que funciona en varios idiomas. A continuación mostramos las modificaciones correspondientes.

```
package es.eni.editions.ihm14;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    JPanel panel;
    JButton btnRojo, btnVerde, btnAzul;
    JMenuItem mnuRojo, mnuVerde, mnuAzul;

    public Pantalla ()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,300,100);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        // creación de los tres botones y asociación de un
        ActionCommand
        btnRojo=new JButton("Rojo");
        btnRojo.setActionCommand("red");
        btnVerde=new JButton("Verde");
        btnVerde.setActionCommand("green");
        btnAzul=new JButton("Azul");
        btnAzul.setActionCommand("blue");
    }
}
```



```
// creación de un solo receptor de eventos
ReceptorColor ec;
ec=new ReceptorColor();
// se asocia cada receptor con su botón
btnRojo.addActionListener(ec);
btnVerde.addActionListener(ec);
btnAzul.addActionListener(ec);
// Creación del menú
JMenuBar barraMenu;
barraMenu=new JMenuBar();
JMenu mnuColores;
mnuColores=new JMenu("Colores");
barraMenu.add(mnuColores);
// Creación de los menús y asociaciones de un ActionCommand
mnuRojo=new JMenuItem("Rojo");
mnuRojo.setActionCommand("red");
mnuVerde=new JMenuItem("Verde");
mnuVerde.setActionCommand("green");
mnuAzul=new JMenuItem("Azul");
mnuAzul.setActionCommand("blue");
mnuColores.add(mnuRojo);
mnuColores.add(mnuVerde);
mnuColores.add(mnuAzul);
// se asocia cada receptor con su elemento de menú
// (el mismo que para los botones)
mnuRojo.addActionListener(ec);
mnuVerde.addActionListener(ec);
mnuAzul.addActionListener(ec);
// se agrega el menú en la ventana
setJMenuBar(barraMenu);
// creación del contenedor intermedio
panel=new JPanel();
// se agregan los botones en el contenedor intermedio
panel.add(btnRojo);
panel.add(btnVerde);
panel.add(btnAzul);
// se agrega el contenedor intermedio en el ContentPane
getContentPane().add(panel);
```

```
// creación de una instancia de una clase anónima
// encargada de gestionar los eventos
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}
public void mostrar()
{
    this.setVisible(true);
}
public class ReceptorColor implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String comando;
        comando=arg0.getActionCommand();
        if (comando.equals("red"))
        {
            panel.setBackground(Color.RED);
        }
        if (comando.equals("green"))
        {
            panel.setBackground(Color.GREEN);
        }
        if (comando.equals("blue"))
        {
            panel.setBackground(Color.BLUE);
        }
    }
}
}
```

Destacaremos que, en esta solución, la declaración de los botones y de los elementos de menú pueden integrarse en el constructor, ya que no son necesarios a nivel de clase.

La última etapa de nuestro maratón en los eventos nos va a permitir disponer de varios receptores de eventos para una misma fuente de eventos y, en caso necesario, suprimir un receptor existente. Para ello, vamos a crear una nueva clase receptor que nos va a permitir mostrar en la consola la fecha y la hora del evento y el objeto fuente del evento.

```
package es.eni.editions.ihm15;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.swing.AbstractButton;
import javax.swing.JButton;
import javax.swing.JMenuItem;

public class ConsoleLog implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String mensaje;
        SimpleDateFormat sdf;
        sdf=new SimpleDateFormat("dd/MM/yyyy hh:mm:ss");
        mensaje=sdf.format(new Fecha());
        mensaje=mensaje + " clic en el ";
        if (e.getSource() instanceof JButton)
        {
            mensaje=mensaje+ "botón ";
        }
        if (e.getSource() instanceof JMenuItem)
        {
            mensaje=mensaje+ "menu ";
        }
        mensaje=mensaje + ((AbstractButton)e.getSource()).getText();
        System.out.println(mensaje);
    }
}
```

```
}  
}
```

Agregamos a continuación, en nuestra aplicación, una casilla que nos permite seleccionar si se visualizan los eventos en la consola. Según la configuración de esta opción, añadimos con el método `addActionListener`, o suprimimos con el método `removeActionListener`, un receptor de eventos a los botones y menús. Estos dos métodos reciben como argumento la instancia del receptor de eventos que se desea agregar o eliminar.

```
package es.eni.editions.ihm15;  
  
import java.awt.Color;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
import java.awt.event.WindowAdapter;  
import java.awt.event.WindowEvent;  
import javax.swing.JButton;  
import javax.swing.JCheckBox;  
import javax.swing.JFrame;  
import javax.swing.JMenu;  
import javax.swing.JMenuBar;  
import javax.swing.JMenuItem;  
import javax.swing.JPanel;  
  
public class Pantalla extends JFrame  
{  
    JPanel panel;  
    JButton btnRojo, btnVerde, btnAzul;  
    JMenuItem mnuRojo, mnuVerde, mnuAzul;  
    ConsoleLog lg;  
    public Pantalla ()  
    {  
        setTitle("Primera ventana en Java");  
        setBounds(0,0,300,130);  
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);  
        // creación de los tres botones
```

```
btnRojo=new JButton("Rojo");
btnRojo.setActionCommand("red");
btnVerde=new JButton("Verde");
btnVerde.setActionCommand("green");
btnAzul=new JButton("Azul");
btnAzul.setActionCommand("blue");
// creación de un solo receptor de eventos
ReceptorColor ec;
ec=new ReceptorColor();
// se asocia cada receptor con su botón
btnRojo.addActionListener(ec);
btnVerde.addActionListener(ec);
btnAzul.addActionListener(ec);
// creación de la opción a marcar
JCheckBox chkLog;
chkLog=new JCheckBox("log en consola");
// se agrega un receptor de eventos a la opción a marcar
chkLog.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        JCheckBox chk;
        chk=(JCheckBox)arg0.getSource();
        if (chk.isSelected()) //Si la casilla está marcada
        {
            // se agrega un receptor adicional
            // a los botones y menús
            lg=new ConsoleLog();
            btnAzul.addActionListener(lg);
            btnRojo.addActionListener(lg);
            btnVerde.addActionListener(lg);
            mnuAzul.addActionListener(lg);
            mnuRojo.addActionListener(lg);
            mnuVerde.addActionListener(lg);
        }
        else
        {
```

```

        // se elimina el receptor adicional
        // de los botones y menús
        btnAzul.removeActionListener(lg);
        btnRojo.removeActionListener(lg);
        btnVerde.removeActionListener(lg);
        mnuAzul.removeActionListener(lg);
        mnuRojo.removeActionListener(lg);
        mnuVerde.removeActionListener(lg);
    }

}

});
// Creación del menú
JMenuBar barraMenu;
barraMenu=new JMenuBar();
JMenu mnuColores;
mnuColores=new JMenu("Colores");
barraMenu.add(mnuColores);
mnuRojo=new JMenuItem("Rojo");
mnuRojo.setActionCommand("red");
mnuVerde=new JMenuItem("Verde");
mnuVerde.setActionCommand("green");
mnuAzul=new JMenuItem("Azul");
mnuAzul.setActionCommand("blue");
mnuColores.add(mnuRojo);
mnuColores.add(mnuVerde);
mnuColores.add(mnuAzul);
// se asocia cada receptor de eventos con su elemento
// de menú (el mismo que para los botones)
mnuRojo.addActionListener(ec);
mnuVerde.addActionListener(ec);
mnuAzul.addActionListener(ec);
// se agrega el menú a la ventana
setJMenuBar(barraMenu);
// creación del contenedor intermedio
panel=new JPanel();
// se agregan los botones al contenedor intermedio
panel.add(btnRojo);

```

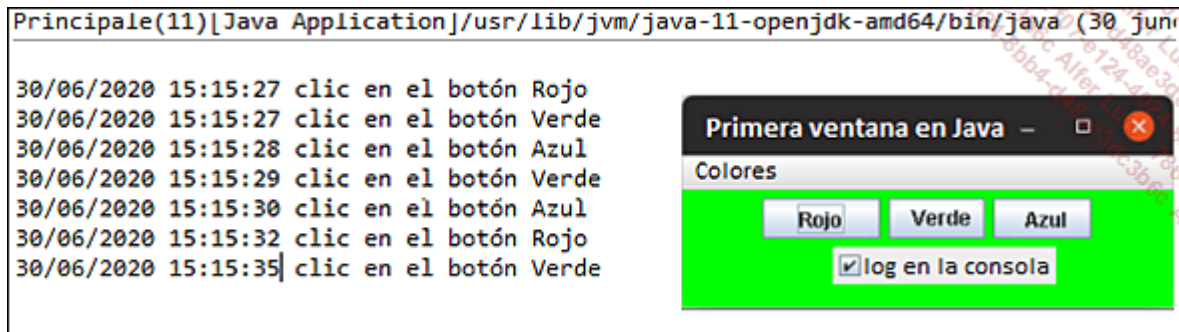
```
panel.add(btnVerde);
panel.add(btnAzul);
panel.add(chkLog);
// se añade el contenedor intermedio
// en el ContentPane
getContentPane().add(panel);
// creación de una instancia de una clase anónima
// encargada de gestionar los eventos
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
}
public void mostrar()
{
    this.setVisible(true);
}
public class ReceptorColor implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String comando;
        comando=arg0.getActionCommand();
        if (comando.equals("red"))
        {
            panel.setBackground(Color.RED);
        }
        if (comando.equals("green"))
        {
            panel.setBackground(Color.GREEN);
        }
        if (comando.equals("blue"))
        {
            panel.setBackground(Color.BLUE);
        }
    }
}
```

```

    }
}
}
}

```

En tiempo de ejecución y en función de la selección, vemos aparecer los mensajes siguientes en la consola.



Cabe destacar que, independientemente de la visualización de estos mensajes, el color de la ventana cambia siempre al usar estos botones o menús.

Todos estos principios de gestión son idénticos sean cuales sean los eventos y los objetos que los activan. El único problema que podemos encontrar al principio es saber cuáles son los eventos disponibles para un objeto particular. A este nivel, la documentación Java es la única que le puede ayudar. Un pequeño truco consiste en buscar en la clase relativa al objeto correspondiente, los métodos llamados `addXxxxListener` y, a continuación, desde este método, subir hasta la interfaz correspondiente y descubrir, gracias a los métodos definidos en esta interfaz, los diferentes eventos posibles. El análisis del tipo de argumento recibido en los diferentes métodos le permitirá saber qué información adicional está disponible para un evento particular.

Por ejemplo, si localizamos el método `addMouseListener`.

addMouseListener

```
public void addMouseListener(MouseListener l)
```

Adds the specified mouse listener to receive mouse events from this component. If listener `l` is null, no exception is thrown and no action is performed.

Refer to [AWT Threading Issues](#) for details on AWT's threading model.

Parameters:

`l` - the mouse listener

Since:

1.1

See Also:

`MouseEvent`, `MouseListener`, `removeMouseListener(java.awt.event.MouseListener)`, `getMouseListeners()`

Ya puede determinar que este método va a permitir trabajar con eventos relacionados con el ratón. Para obtener más información relativa a los diferentes eventos, debemos consultar la interfaz `MouseListener`.

All Methods	Instance Methods	Abstract Methods	
Modifier and Type	Method	Description	
void	<code>mouseClicked(MouseEvent e)</code>	Invoked when the mouse button has been clicked (pressed and released) on a component.	
void	<code>mouseEntered(MouseEvent e)</code>	Invoked when the mouse enters a component.	
void	<code>mouseExited(MouseEvent e)</code>	Invoked when the mouse exits a component.	
void	<code>mousePressed(MouseEvent e)</code>	Invoked when a mouse button has been pressed on a component.	
void	<code>mouseReleased(MouseEvent e)</code>	Invoked when a mouse button has been released on a component.	

La documentación de esta interfaz muestra que están previstos cinco métodos y, por lo tanto, que es posible gestionar cinco tipos de eventos y en qué circunstancias se producen. Para saber qué información estará disponible cuando se produzca alguno de estos eventos, debe consultar la clase correspondiente al tipo de los argumentos recibidos por estos métodos (`MouseEvent`).

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description	
int	getButton()	Returns which, if any, of the mouse buttons has changed state.	
int	getClickCount()	Returns the number of mouse clicks associated with this event.	
Point	getLocationOnScreen()	Returns the absolute x, y position of the event.	
static String	getMouseModifiersText (int modifiers)	Returns a String instance describing the modifier keys and mouse buttons that were down during the event, such as "Shift", or "Ctrl+Shift".	
Point	getPoint()	Returns the x,y position of the event relative to the source component.	
int	getX()	Returns the horizontal x position of the event relative to the source component.	
int	getXOnScreen()	Returns the absolute horizontal x position of the event.	
int	getY()	Returns the vertical y position of the event relative to the source component.	
int	getYOnScreen()	Returns the absolute vertical y position of the event.	
boolean	isPopupTrigger()	Returns whether or not this mouse event is the popup menu trigger event for the platform.	
String	 paramString()	Returns a parameter string identifying this event.	
void	translatePoint (int x, int y)	Translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets.	

Sin duda, pensará que todo esto es muy engorroso, pero se dará cuenta pronto de que casi siempre se utilizan los mismos eventos y los llegará a conocer de memoria en poco tiempo.

4. Aspecto de los componentes

Para una aplicación Java susceptible de ejecutarse en cualquier plataforma, el aspecto de los componentes debe adaptarse a cada plataforma. Java provee una solución con el mecanismo de *look and feel*. Cada componente gráfico está de hecho formado por dos clases. Una gestiona el aspecto funcional del componente, mientras que la segunda afecta únicamente al aspecto visual de este. Esta última se adapta en función de la plataforma sobre la que se ejecuta la aplicación. La clase `UIManager` es responsable del funcionamiento de este mecanismo. Cuando se crea un componente gráfico, el constructor del componente utiliza la clase `UIManager` para obtener una instancia de la clase encargada de gestionar el aspecto gráfico del componente. La clase `UIManager` debe estar informada del *look and feel* que debe utilizar.

Es posible indicar el *look and feel* que debe utilizarse de tres formas diferentes:

- Mediante programación: utilice el método `setLookAndFeel` de la clase `UIManager` pasando como parámetro el nombre completo de la clase que implementa el *look and feel* deseado.

```
UIManager.setLookAndFeel("javax.swing.plaf.nimbus.  
NimbusLookAndFeel");
```

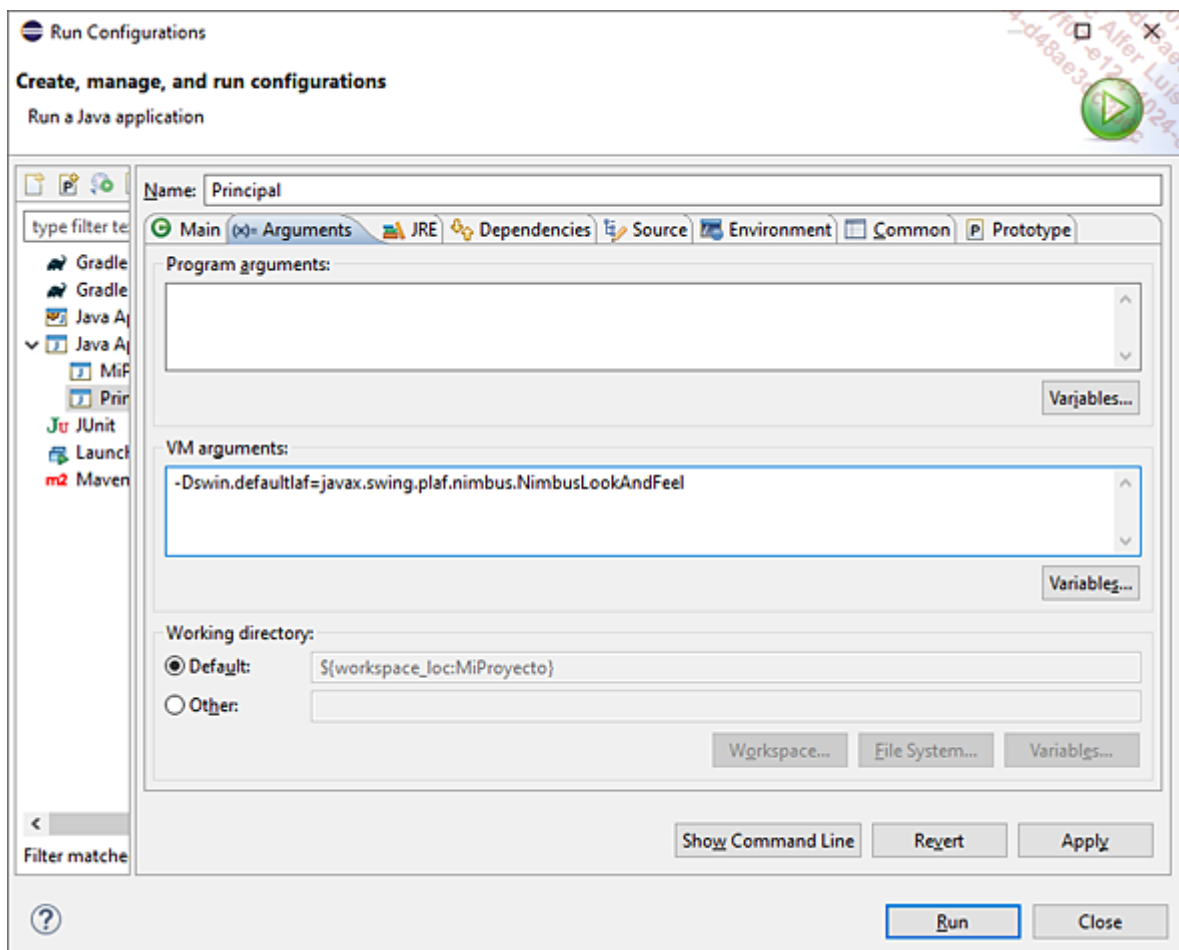
Es aconsejable realizar la modificación del *look and feel* al comienzo de la aplicación antes incluso de la creación del primer componente gráfico.

- Mediante un parámetro de la línea de comandos utilizada para ejecutar la aplicación.

```
java -Dswing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel  
applicationDemo
```

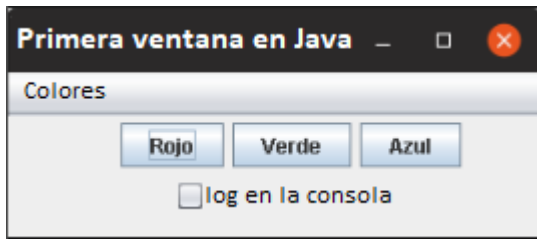
El procedimiento que hay que realizar en Eclipse para obtener el mismo resultado es el siguiente:

- Abra el cuadro de diálogo utilizado para configurar la ejecución de su programa haciendo clic en el menú **Run - Run Configurations**. Aparece la siguiente pantalla:



- Seleccione su configuración a la derecha y vaya a la pestaña **Arguments**.

- Añada el argumento mencionado con anterioridad en la zona **VM arguments**.
- Aplique sus modificaciones y ejecute el programa; el aspecto gráfico de la aplicación ahora es el siguiente:



- Modificando un archivo de configuración.

También puede utilizar un archivo de configuración `swing.properties` para definir el *look and feel* por defecto que aplicará la máquina virtual Java durante la ejecución de la aplicación gráfica. Este archivo debe estar situado en la carpeta `lib` de la máquina virtual. Debe contener la siguiente línea:

```
swing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel
```

- Los siguientes ejemplos utilizan el look and feel nimbus disponible desde la versión 7 de Java. Se distingue de otros look and feel por su modo de gestión gráfica. Las versiones anteriores utilizaban mapas de bits para la representación gráfica de los componentes. Esta nueva versión utiliza un modo gráfico vectorial que proporciona una representación gráfica más precisa que el modo de mapa de bits.

5. El posicionamiento de los componentes

Si ya ha utilizado otro lenguaje de programación que permite desarrollar interfaces gráficas, una cosa le debe de haber parecido extraña en los ejemplos de código que hemos utilizado.

- En ninguna parte hemos indicado el tamaño y la posición de los componentes usados.
- Si redimensiona la ventana de la aplicación, los componentes cambian de sitio.

Y sin embargo, no se ha previsto ninguna línea de código para efectuar este tratamiento. Este pequeño milagro está relacionado con un concepto muy práctico de java: los

renderizadores (*layout manager*). En realidad, cuando confiamos componentes a un contenedor, este delega a su renderizador la tarea de organizar la disposición de los componentes en su superficie. Hay varios tipos de renderizador; cada uno de ellos dispone de una estrategia diferente a la hora de organizar los componentes. Cada tipo de contenedor dispone de un renderizador por defecto, diferente según el tipo de contenedor. Si este renderizador por defecto no le conviene, puede sustituirlo por otro a través de la llamada al método `setLayout`, pasándole el nuevo renderizador que debe utilizar este contenedor.

Para poder trabajar y organizar la disposición de los componentes, el renderizador debe conocer el tamaño de cada componente. La mejor manera de obtener esta información es dirigirse directamente al propio componente. Para ello, el renderizador pregunta a cada componente invocando el método `getPreferredSize`. Este método calcula el tamaño del componente según su contenido; por ejemplo, la longitud del título para un botón.

Para cortocircuitar este cálculo, puede fijar un tamaño por defecto para cada componente con el método `setPreferredSize`.

El éxito del diseño de una interfaz de usuario supone, por tanto, conocer a fondo cómo funcionan los diferentes renderizadores. Vamos a estudiar las características de los que más se utilizan.

a. FlowLayout

Este renderizador es seguramente el más fácil de utilizar. Está asociado por defecto a un componente `JPanel`. Su estrategia de organización de los componentes consiste en ubicarlos uno tras otro en una línea hasta que no haya más sitio en esta línea. Después de rellenar la primera línea, se ubican los componentes siguientes en una nueva línea, y así sucesivamente. Es el orden en que se añaden los componentes en el contenedor el que determina sus posiciones en las diferentes líneas. Cada componente está separado de manera horizontal de su vecino por un espacio de cinco píxeles predeterminado y cada línea de componentes está separada de su vecina por un espacio de cinco píxeles también por defecto. Cuando este renderizador organiza los componentes, los alinea por defecto en el centro del contenedor. Se pueden modificar todos estos parámetros para cada renderizador `FlowLayout`. Esto se puede hacer desde la creación del `FlowLayout` indicando en el constructor la información correspondiente. Existen tres constructores para esta clase. El primero no recibe argumento alguno y crea un `FlowLayout` con las características por defecto descritas más arriba. El segundo recibe como argumento una de

las constantes siguientes, que permiten especificar el tipo de alineación.

- `FlowLayout.CENTER`: cada línea de componente está centrada en el contenedor (valor por defecto).
- `FlowLayout.LEFT`: cada línea de componente está alineada a la izquierda del contenedor.
- `FlowLayout.RIGHT`: cada línea de componente está alineada a la derecha del contenedor.

El último constructor disponible recibe como parámetros dos enteros, además de la constante que indica la alineación. Estos dos enteros indican el espaciado horizontal y vertical entre los componentes. La línea de código siguiente crea un `FlowLayout` que alinea las líneas de componentes en la izquierda del contenedor, deja un espacio horizontal de cincuenta píxeles y un espacio vertical de veinte píxeles entre los componentes.

```
FlowLayout fl=new FlowLayout(FlowLayout.LEFT,50,20);
```

Si en lugar de crear su propio `FlowLayout` opta por utilizar el proporcionado por defecto con un `Jpanel`, puede intervenir en estos parámetros de funcionamiento con los métodos siguientes:

- `setAlignment`: para especificar la alineación de los componentes.
- `setHgap`: para especificar el espaciado horizontal entre los componentes.
- `setVgap`: para especificar el espaciado vertical entre los componentes.

En este caso, es necesario obtener una referencia sobre el `FlowLayout` asociado al `JPanel` empleando el método `getLayout` de este. En este supuesto, se hace obligatorio realizar una operación de tipado dinámico para poder utilizar los métodos de la clase `FlowLayout` sobre la referencia obtenida.

De hecho, el método `getLayout` devuelve una variable de tipo `LayoutManager`, el supertipo de todos los tipos de administradores de presentaciones.

```
//Definición de las características del LayoutManager
LayoutManager lm = panel.getLayout();
if(lm instanceof FlowLayout)
{
    FlowLayout fl = (FlowLayout) lm;
```

```
f1.setAlignment(FlowLayout.LEFT);  
f1.setHgap(50);  
f1.setVgap(20);
```

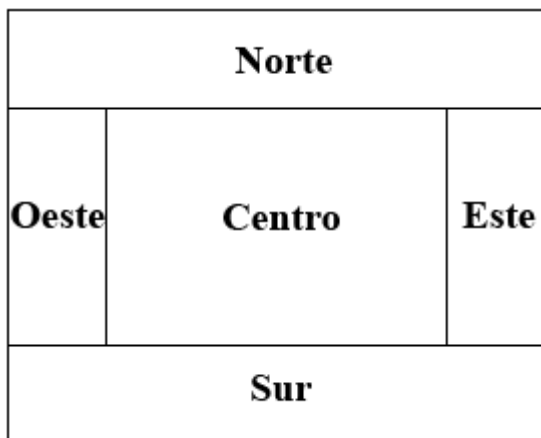


Este ejemplo está disponible en el paquete `es.eni.editions.ihm16`.

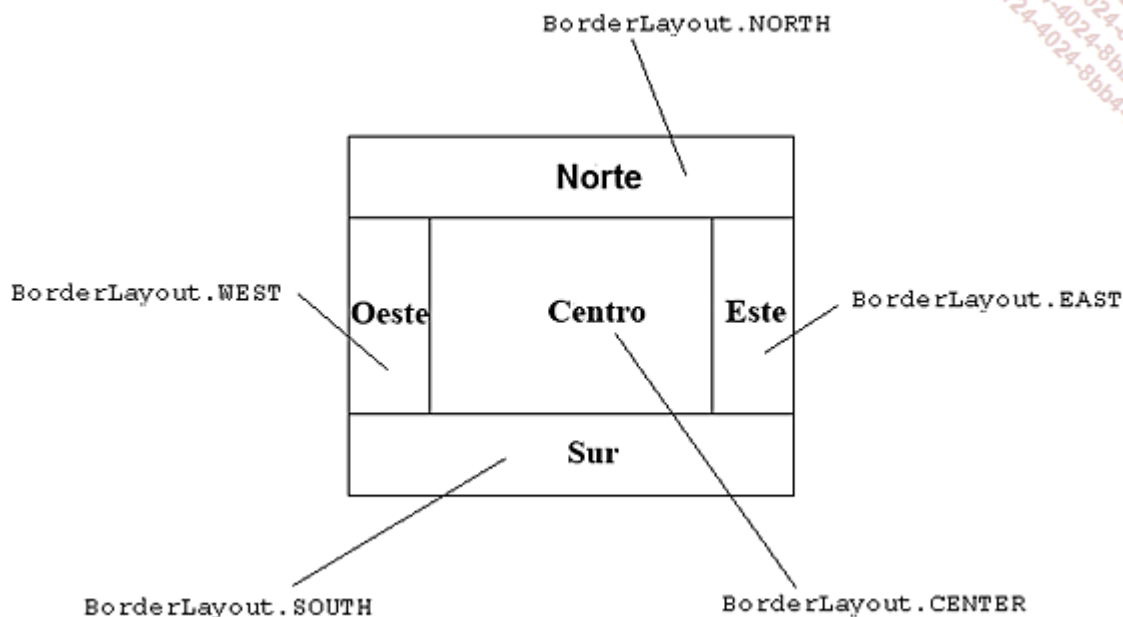
Redimensione la ventana para observar el comportamiento.

b. BorderLayout

Este renderizador organiza en cinco zonas la superficie que se le confía, según el esquema siguiente.



Cuando se confía un componente a un `BorderLayout`, conviene indicar en qué zona se le debe ubicar. Se proporciona esta indicación como segundo argumento del método `add`; el primer argumento siempre es el componente que se debe insertar en el contenedor. Las siguientes constantes están disponibles para identificar una zona del `BorderLayout`.



Si no se indica ninguna información para identificar una zona cuando se agrega un componente, este se ubica sistemáticamente en la zona **Centro** (`BorderLayout.CENTER`).

Al `BorderLayout` no le gusta el vacío, por lo que redimensiona automáticamente todos los componentes que se le confían para ocupar todo el espacio disponible. Cuando se redimensiona el contenedor, los componentes ubicados en los bordes no cambian de anchura para las zonas Oeste y Este, ni de altura para las zonas Norte y Sur. Por lo tanto, se gana o se pierde espacio en la zona Centro. Cada zona del `BorderLayout` solo puede contener un único componente. Si, a pesar de todo, se añaden varios componentes a una misma zona de un `BorderLayout`, este solo se encargará del último añadido y los demás no serán visibles.

Por esta limitación, se suele utilizar el `BorderLayout` para posicionar contenedores en relación con otros elementos, más que para posicionar componentes aislados.

El `BorderLayout` es el renderizador por defecto del elemento `ContentPane` de una `JFrame`. El uso clásico consiste en colocar en la zona **Norte** la barra o las barras de herramientas, en la zona **Sur** la barra de estado de la aplicación y en la zona **Centro** el documento sobre el que debe trabajar el usuario. Como para el `FlowLayout`, es posible indicar al `BorderLayout` que debe reservar un espacio vertical u horizontal entre las distintas zonas. Esto se puede llevar a cabo en el momento de la creación del `BorderLayout` indicando el espacio horizontal y vertical en la llamada al constructor o,

si utilizamos un BorderLayout existente, invocando los métodos setHgap y setVgap del BorderLayout. Para ilustrar todo ello, vamos a retomar la interfaz de nuestra aplicación ubicando los botones en la zona **Norte**, la opción marcable en la zona **Sur** y modificando el color de la zona **Centro** en el momento del clic en los botones o menús.

```
package es.eni.editions.ihm17;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.LayoutManager;
import java.awt.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{
    JPanel panelBotones;
    JPanel panelChk;
    JPanel panelColor;
    JButton btnRojo,btnVerde,btnAzul;
    JMenuItem mnuRojo,mnuVerde,mnuAzul;
    ConsoleLog lg;
    public Pantalla()
    {
        setTitle("Primera ventana en Java");
        setBounds(0,0,400,200);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

```
// creación de los tres botones
```

```
btnRojo=new JButton("Rojo");
```

```
btnRojo.setActionCommand("red");
```

```
btnVerde=new JButton("Verde");
```

```
btnVerde.setActionCommand("green");
```

```
btnAzul=new JButton("Azul");
```

```
btnAzul.setActionCommand("blue");
```

```
// creación de un solo receptor de eventos
```

```
ReceptorColor ec;
```

```
ec=new ReceptorColor();
```

```
// se asocia cada receptor de eventos con su botón
```

```
btnRojo.addActionListener(ec);
```

```
btnVerde.addActionListener(ec);
```

```
btnAzul.addActionListener(ec);
```

```
// creación de la opción a marcar
```

```
JCheckBox chkLog;
```

```
chkLog=new JCheckBox("log en consola");
```

```
// se agrega un receptor de eventos a la opción a marcar
```

```
chkLog.addActionListener(new ActionListener()
```

```
{
```

```
    public void actionPerformed(ActionEvent arg0)
```

```
    {
```

```
        JCheckBox chk;
```

```
        chk=(JCheckBox)arg0.getSource();
```

```
        if (chk.isSelected())
```

```
        {
```

```
            // se agrega un receptor adicional
```

```
            // a los botones y menús
```

```
            lg=new ConsoleLog();
```

```
            btnAzul.addActionListener(lg);
```

```
            btnRojo.addActionListener(lg);
```

```
            btnVerde.addActionListener(lg);
```

```
            mnuAzul.addActionListener(lg);
```

```
            mnuRojo.addActionListener(lg);
```

```
            mnuVerde.addActionListener(lg);
```

```
        }
```

```
    else
```

```

        {
            // se elimina un receptor adicional
            // de los botones y menús
            btnAzul.removeActionListener(lg);
            btnRojo.removeActionListener(lg);
            btnVerde.removeActionListener(lg);
            mnuAzul.removeActionListener(lg);
            mnuRojo.removeActionListener(lg);
            mnuVerde.removeActionListener(lg);
        }

    }
});
// Creación del menú
JMenuBar barraMenu;
barraMenu=new JMenuBar();
JMenu mnuColores;
mnuColores=new JMenu("Colores");
barraMenu.add(mnuColores);
mnuRojo=new JMenuItem("Rojo");
mnuRojo.setActionCommand("red");
mnuVerde=new JMenuItem("Verde");
mnuVerde.setActionCommand("green");
mnuAzul=new JMenuItem("Azul");
mnuAzul.setActionCommand("blue");
mnuColores.add(mnuRojo);
mnuColores.add(mnuVerde);
mnuColores.add(mnuAzul);
// se asocia cada receptor con su elemento de menú
// (el mismo que para los botones)
mnuRojo.addActionListener(ec);
mnuVerde.addActionListener(ec);
mnuAzul.addActionListener(ec);
// se agrega el menú en la ventana
setJMenuBar(barraMenu);
// creación del contenedor intermedio
panelBotones=new JPanel();
// se agregan los botones en el contenedor intermedio

```

```

panelBotones.add(btnRojo);
panelBotones.add(btnVerde);
panelBotones.add(btnAzul);
// se agrega el contenedor intermedio en el ContentPane
// zona norte
getContentPane().add(panelBotones, BorderLayout.NORTH);
// creación del contenedor para la opción a marcar
panelChk=new JPanel();
panelChk.add(chkLog);
// se agrega el contenedor en la zona sur
getContentPane().add(panelChk, BorderLayout.SOUTH);
// creación del contenedor para visualización del color
panelColor=new JPanel();
// se agrega el contenedor en la zona centro
getContentPane().add(panelColor, BorderLayout.CENTER);
// creación de una instancia de una clase anónima
// encargada de gestionar los eventos
addWindowListener(new WindowAdapter()
{
    public void windowClosing(WindowEvent arg0)
    {
        System.exit(0);
    }
});
LayoutManager lm = panelBotones.getLayout();
if(lm instanceof FlowLayout)
{
    FlowLayout fl = (FlowLayout) lm;
    fl.setAlignment(FlowLayout.LEFT);
    fl.setHgap(50);
    fl.setVgap(20);
}
}

public void mostrar() {
    this.setVisible(true);
}

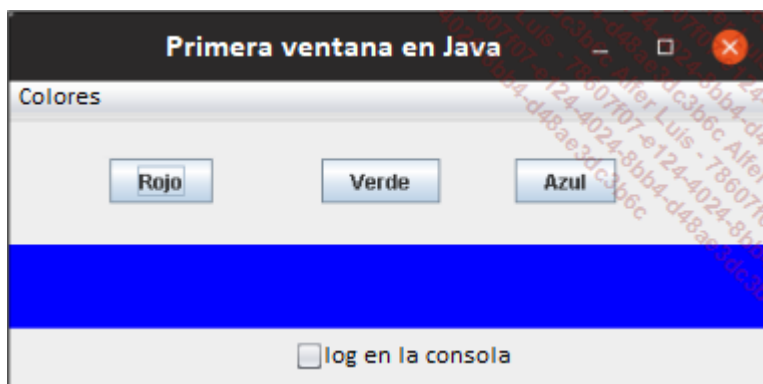
```

```

public class ReceptorColor implements ActionListener
{
    public void actionPerformed(ActionEvent arg0)
    {
        String comando;
        comando=arg0.getActionCommand();
        if (comando.equals("red"))
        {
            panelColor.setBackground(Color.RED);
        }
        if (comando.equals("green"))
        {
            panelColor.setBackground(Color.GREEN);
        }
        if (comando.equals("blue"))
        {
            panelColor.setBackground(Color.BLUE);
        }
    }
}
}

```

La ventana de la aplicación presenta ahora la disposición siguiente:



c. GridLayout

Este renderizador organiza la superficie que se le confía mediante una malla invisible. Se debe indicar el número de filas y columnas en la llamada al constructor. Cuando este renderizador ubica dichos componentes, empieza por rellenar todas las casillas de la

primera fila y, a continuación, sigue con la fila siguiente y así sucesivamente. Todas las casillas tienen un tamaño idéntico y por lo tanto su contenido se redimensiona para ocupar todo el espacio disponible en cada casilla. Por defecto, se colocan los componentes uno al lado del otro. Es posible insertar un espacio vertical u horizontal entre cada componente especificando dos argumentos adicionales en la llamada al constructor. Estos dos argumentos indican el espaciado horizontal y el espaciado vertical. Estos espaciados también pueden modificarse gracias a los métodos `setHgap` y `setVgap`.

Si se insertan más componentes que casillas existentes en el `GridLayout`, este creará casillas adicionales aumentando el número de columnas pero respetando el número de filas de origen. Por ejemplo, con el código siguiente prevemos una malla de 2x2 casillas, pero añadimos seis componentes.

```
package es.eni.editions.ihm18;

import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{

    public Pantalla() {
        setTitle("Primera ventana en Java");
        setBounds(0, 0, 400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel malla;
        malla=new JPanel();
        GridLayout grl;
        grl=new GridLayout(2,2);
        malla.setLayout(grl);
        malla.add(new JButton("1"));
        malla.add(new JButton("2"));
        malla.add(new JButton("3"));
        malla.add(new JButton("4"));
```

```

        malla.add(new JButton("5"));
        malla.add(new JButton("6"));

        getContentPane().add(malla);
    }

    public void mostrar() {
        this.setVisible(true);
    }
}

```

Los componentes están colocados según el esquema siguiente:



Si deseamos que el `GridLayout` respete el número de columnas, debemos indicar que no importa el número de filas y, para ello, debemos especificar el valor 0 para este argumento en el constructor. En este caso, el renderizador añade filas adicionales para los componentes en exceso.

```

GridLayout grl;
grl=new GridLayout(0,2);

```



d. BoxLayout

Se utiliza este renderizador cuando se necesita organizar componentes en una fila o una columna. Aunque se pueda obtener el mismo resultado con un `GridLayout` de una única fila o de una única columna, este renderizador ofrece más funcionalidades. El precio que se debe pagar a cambio de estas funcionalidades adicionales es el de una relativa complejidad de utilización. El constructor de la clase `BoxLayout` recibe como primer argumento el contenedor cuyo contenido va a gestionar, y como segundo argumento, una constante que informa de si este contenedor gestiona una fila (`BoxLayout.X_AXIS`) o una columna (`BoxLayout.Y_AXIS`).

Aunque proporcionemos una referencia al contenedor en el momento de la creación del renderizador, resulta obligatorio asociar el renderizador al contenedor mediante el método `setLayout`, como ya hemos hecho con otros renderizadores. Por lo tanto, se implementa una relación bidireccional.

Estudiamos ahora la estrategia de funcionamiento de este renderizador. Veamos el ejemplo de un renderizador horizontal.

- El renderizador busca el componente más alto.
- Intenta ampliar los demás componentes hasta esa altura.
- Si un componente no puede alcanzar esa altura, se alinea verticalmente sobre los demás componentes según el valor devuelto por el método `SetAlignmentY` del componente.
 - 0 para una alineación por encima de los demás componentes.
 - 1 para una alineación por debajo de los demás componentes.
 - 0.5 para que esté centrado respecto a los demás componentes.
- Se suman las anchuras preferidas de cada componente.
- Si la suma es inferior a la anchura del contenedor, se amplían los componentes hasta su anchura máxima.
- Si la suma es superior a la anchura del contenedor, se reducen los componentes hasta su anchura mínima. Si no resulta suficiente, no se visualizarán los últimos componentes.
- A continuación, se organizan los componentes en el contenedor de izquierda a derecha sin espacio de separación.

El funcionamiento es muy similar para un renderizador vertical.

```
package es.eni.editions.ihm19;

import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{

    public Pantalla() {
        setTitle("Primera ventana en Java");
        setBounds(0, 0, 500, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel línea;
        línea=new JPanel();
        BoxLayout bl;
        bl=new BoxLayout(línea,BoxLayout.X_AXIS);
        línea.setLayout(bl);
        JButton b1,b2,b3,b4,b5;
        // creación de un botón con alineación en la parte superior
        b1=new JButton("pequeno");
        b1.setAlignmentY(0);
        línea.add(b1);
        // creación de un botón con alineación en la parte inferior
        b2=new JButton("    medio    ");
        b2.setAlignmentY(1);
        línea.add(b2);
        // uso de html para el título del botón
        b3=new JButton("<html>muy<BR>alto</html>");
        línea.add(b3);
        b4=new JButton("        muy ancho        ");
        línea.add(b4);
        b5=new JButton("<html>muy alto<br>y<br>muy ancho</html>");
```

```

        línea.add(b5);
        getContentPane().add(línea);

    }

    public void mostrar() {
        this.setVisible(true);
    }

}

```

Este código permite obtener la interfaz usuario siguiente.



Por defecto no hay espacio entre los componentes gestionados por un `BoxLayout` y el aspecto final no es gran cosa. Para poder añadir un espacio entre los componentes, debemos insertar componentes de reserva de espacio, que pueden ser de tres tipos:

Strut: este elemento se utiliza para añadir un espacio fijo entre dos componentes. Un `Strut` puede ser horizontal o vertical según el tipo de `BoxLayout` al cual se destina. Estos dos tipos de elementos se crean mediante los métodos estáticos `createVerticalStrut` y `createHorizontalStrut` de la clase `Box`. Ambos métodos reciben como argumento el número de píxeles para la anchura o la altura.

```

// creación de un botón con alineación en la parte superior
b1=new JButton("pequeño");
b1.setAlignmentY(0);
línea.add(b1);
línea.add(Box.createHorizontalStrut(10));
// creación de un botón con alineación en la parte inferior
b2=new JButton(" medio ");

```

```
b2.setAlignmentY(1);  
línea.add(b2);
```

RigidArea: este elemento tiene un funcionamiento similar al de un `Strut` al separar los componentes entre sí. También requiere una altura mínima para el contenedor. En caso de que un componente contenido en el contenedor tenga una altura superior a esa altura mínima, el componente será quien imponga la altura del contenedor. En caso contrario, será el componente más alto el que impondrá la altura. Como para un `Strut`, el método estático `createRigidArea` de la clase `Box` nos permite la creación de un `RigidArea`.

Este método recibe como parámetro un objeto `Dimension` que especifica el espaciado entre los componentes y la altura mínima del contenedor.

```
// creación de un botón con alineación en la parte superior  
b1=new JButton("pequeño");  
b1.setAlignmentY(0);  
línea.add(b1);  
línea.add(Box.createRigidArea(new Dimension(50,150)));  
// creación de un botón con alineación en la parte inferior  
b2=new JButton(" medio ");  
b2.setAlignmentY(1);  
línea.add(b2);
```

Glue: la meta de este elemento sigue siendo la de separar los componentes, pero, esta vez, el espacio entre los componentes no tiene un tamaño fijo. Se puede comparar el objeto `Glue` con un muelle que se coloca entre dos componentes y que los aleja siempre lo más posible. Para obtener un funcionamiento correcto del objeto `Glue`, los componentes deben tener un tamaño fijo o un tamaño máximo fijado. Como los demás elementos de separación, se crea a partir del método estático `createGlue` de la clase `Box`. El siguiente código es una adaptación de los ejemplos anteriores, con la instalación de un componente de tipo `Glue` entre el primer y el segundo botón.

```
package es.eni.editions.ihm20;  
  
import java.awt.Dimension;
```

```
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Pantalla extends JFrame

{

    public Pantalla() {
        setTitle("Primera ventana en Java");
        setBounds(0, 0, 500, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel línea;
        línea=new JPanel();
        BoxLayout bl;
        bl=new BoxLayout(línea,BoxLayout.X_AXIS);
        línea.setLayout(bl);
        JButton b1,b2,b3,b4,b5;
        // creación de un botón con alineación en la parte superior
        b1=new JButton("pequeño");
        b1.setAlignmentY(0);
        b1.setMaximumSize(new Dimension(50,20));
        línea.add(b1);

        // Creación de un separador de tamaño variable
        línea.add(Box.createGlue());

        // creación de un botón con alineación en la parte inferior
        b2=new JButton(" medio ");
        b2.setAlignmentY(1);
        b2.setMaximumSize(new Dimension(50,20));
        línea.add(b2);
        // utilización de html para la etiqueta del botón
        b3=new JButton("<html>muy<BR>alto</html>");
        b3.setMaximumSize(new Dimension(50, 100));
```

```

        línea.add(b3);
        b4=new JButton("    my    largo    ");
        b4.setMaximumSize(new Dimension(100, 20));
        línea.add(b4);
        b5=new JButton("<html>muy alto<br>y<br>my largo</html>");
        b5.setMaximum Size(new Dimension(100, 100));
        línea.add(b5);
        getContentPane().add(línea);

    }
    public void mostrar() {
        this.setVisible(true);
    }
}

```

Ejecute este código y redimensione la ventana para observar el nuevo comportamiento.

e. GridBagLayout

Puede considerar el `GridBagLayout` como el «súperrenderizador» tanto desde el punto de vista de las posibilidades que ofrece como de la complejidad de utilización. Funciona básicamente como un `GridLayout`, ubicando los componentes en el interior de una malla, pero aquí acaba la comparación. En un `GridBagLayout` las filas y las columnas tienen tamaños variables, las celdas adyacentes se pueden fusionar para acoger los componentes más grandes. Los componentes no ocupan obligatoriamente toda la superficie de sus celdas y es posible modificar su posición en el interior de la celda. La especificidad de este renderizador se sitúa principalmente en el método que permite agregar componentes. Este método recibe como parámetro, por supuesto, el componente que se ha de añadir, pero también un objeto `GridBagConstraints`, que indica la manera de colocar el componente en el contenedor.

Por lo tanto, son las características de este objeto las que van a permitir el posicionamiento del componente por el `GridBagLayout`. Veamos las características principales:

- **gridx**: número de la columna a partir de la esquina superior izquierda del componente.

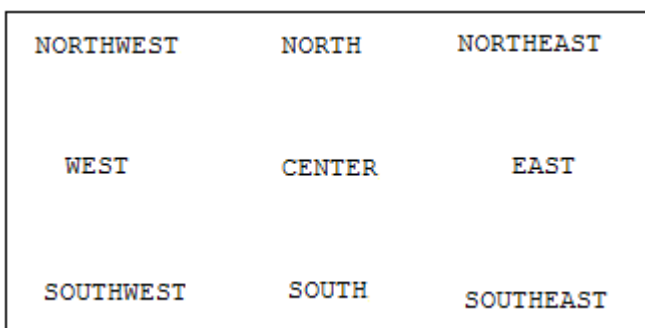
gridy: número de la fila a partir de la esquina superior izquierda del componente.

Para estas dos propiedades, la numeración empieza por cero.

- **gridwidth**: número de columnas ocupadas por el componente.
- **gridheight**: número de filas ocupadas por el componente.
- **weightx**: indica cómo se reparte el espacio adicional disponible en anchura cuando se redimensiona el contenedor. El reparto se hace repartiendo este valor. Si el valor es igual a cero, no se redimensiona. Si es idéntico para todos los componentes, se reparte el espacio de manera proporcional.
- **weighty**: esta propiedad tiene el mismo papel que la anterior, pero en el eje vertical.
- **fill**: se utiliza esta propiedad cuando la zona asignada a un componente es superior a su tamaño. Determina si se redimensiona el componente y la manera de hacerlo.

Puede adoptar los siguientes valores constantes:

- **NONE**: no se redimensiona el componente.
- **HORIZONTAL**: se redimensiona el componente en anchura y no se cambia su altura.
- **VERTICAL**: se redimensiona el componente en altura y no se cambia su anchura.
- **BOTH**: se redimensiona el componente en anchura y altura para rellenar completamente la superficie disponible.
- **anchor**: indica la manera en que está posicionado el componente en el espacio disponible si no se redimensiona. Puede adoptar los siguientes valores constantes.



Se puede usar un mismo objeto `GridBagConstraints` para organizar varios componentes en un `GridBagLayout`. En este caso, no hay que olvidar inicializar correctamente los diferentes campos en el momento de la inserción de cada componente.

A título de ejemplo, presentamos a continuación la interfaz de una aplicación sencilla y el código correspondiente.



```
package es.eni.editions.ihm21;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class Pantalla extends JFrame

{
    JPanel panel;
    JCheckBox chkNegrita,chkItalica;
    JLabel lblTamaño,lblEjemplo;
    JComboBox<String> cboTamaño;
    JList<String> lstFuentes;
```

```
JScrollPane desplaFuentes;
```

```
public Pantalla()  
{  
    setTitle("elección de una fuente");  
    setBounds(0,0,400,400);  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    // creación de los componentes  
  
    panel=new JPanel();  
    chkNegrita=new JCheckBox("Negrita");  
    chkItalica=new JCheckBox("Itálica");  
    lblTamaño=new JLabel("Tamaño");  
    lblEjemplo=new JLabel("Prueba de fuente de caracteres");  
    cboTamaño=new JComboBox(new String[]  
{ "10", "12", "14", "16", "18", "20" });  
    lstFuentes=new JList(new String[] { "Arial", "Courier",  
    "Letter", "Helvetica", "Times Roman", "Symbol", "Antique" });  
    desplaFuentes=new JScrollPane(lstFuentes);  
  
    GridBagLayout gbl;  
    gbl=new GridBagLayout();  
    panel.setLayout(gbl);  
  
    GridBagConstraints gbc;  
    gbc=new GridBagConstraints();  
    // posición en la casilla 0,0  
    gbc.gridx=0;  
    gbc.gridy=0;  
    // en una columna de anchura  
    gbc.gridwidth=1;  
    // y en tres líneas en altura  
    gbc.gridheight=3;  
    // ponderación en caso de ampliación del contenedor  
    gbc.weightx=100;  
    gbc.weighty=100;  
    // se redimensiona el componente para ocupar  
    // todo el espacio disponible en su contenedor
```



```
gbc.fill=GridBagConstraints.BOTH;
panel.add(desplaFuentes,gbc);
// posición en la casilla 1,0
gbc.gridx=1;
gbc.gridy=0;
// en dos columnas de anchura
gbc.gridwidth=2;
// y en una fila de altura
gbc.gridheight=1;
// ponderación en caso de ampliación del contenedor
gbc.weightx=100;
gbc.weighty=100;
// no se redimensiona el componente para ocupar
// todo el espacio disponible en su contenedor
gbc.fill=GridBagConstraints.NONE;
panel.add(chkNegrita,gbc);
// posición en la casilla 1,1
gbc.gridx=1;
gbc.gridy=1;
// en dos columnas de anchura
gbc.gridwidth=2;
// y en una fila de altura
gbc.gridheight=1;
// ponderación en caso de ampliación del contenedor
gbc.weightx=100;
gbc.weighty=100;
panel.add(chkItalica,gbc);
// posición en la casilla 1,2
gbc.gridx=1;
gbc.gridy=2;
// en una columna de anchura
gbc.gridwidth=1;
// y en una fila de altura
gbc.gridheight=1;
// ponderación en caso de ampliación del contenedor
gbc.weightx=100;
gbc.weighty=100;
panel.add(lblTamaño,gbc);
```

```

        // posición en la casilla 2,2
        gbc.gridx=2;
        gbc.gridy=2;
        // en una columna de anchura
        gbc.gridwidth=1;
        // y en una fila de altura
        gbc.gridheight=1;
        // ponderación en caso de ampliación del contenedor
        gbc.weightx=100;
        gbc.weighty=100;
        panel.add(cboTamaño,gbc);
        // posición en la casilla 0,3
        gbc.gridx=0;
        gbc.gridy=3;
        // en tres columnas de anchura
        gbc.gridwidth=3;
        // y en una fila de altura
        gbc.gridheight=1;
        // ponderación en caso de ampliación del contenedor
        gbc.weightx=100;
        gbc.weighty=100;
        panel.add(lblEjemplo,gbc);
        getContentPane().add(panel);
    }
}

```

f. Sin renderizador

Si no le convence la utilidad de un renderizador, tiene la posibilidad de gestionar personalmente el tamaño y la posición de los componentes de la interfaz de usuario. Primero debe indicar que renuncia a emplear un renderizador llamando al método `setLayout` del contenedor y pasándole el valor `null`. A continuación, puede añadir los componentes al contenedor con el método `add` de este. Y, para terminar, debe posicionar y dimensionar los componentes. Estas dos operaciones pueden realizarse en dos etapas mediante los métodos `setLocation` y `setSize` o en una sola con el método `setBounds`.

```
package es.eni.editions.ihm22;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;

public class Pantalla extends JFrame

{
    JPanel panel;
    JCheckBox chkNegrita,chkItalica;
    JLabel lblTamaño,lblEjemplo;
    JComboBox<String> cboTamaño;
    JList<String> lstFuentes;
    JScrollPane desplaFuentes;

    public Pantalla()
    {
        setTitle("elección de una fuente");
        setBounds(0,0,400,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los componentes
        panel=new JPanel();
        chkNegrita=new JCheckBox("Negrita");
        chkItalica=new JCheckBox("Itálica");
        lblTamano=new JLabel("Tamaño");
        lblEjemplo=new JLabel("Prueba de fuente de caracteres");
        cboTamano=new JComboBox(new String[]{"10","12","14",
"16","18","20"});
        lstFuentes=new JList
            (new String[]{"Arial","Courier","Letter",
"Helvetica","Times Roman","Symbol", "Antique"});
```

```

        desplaFuentes=new JScrollPane(lstFuentes);

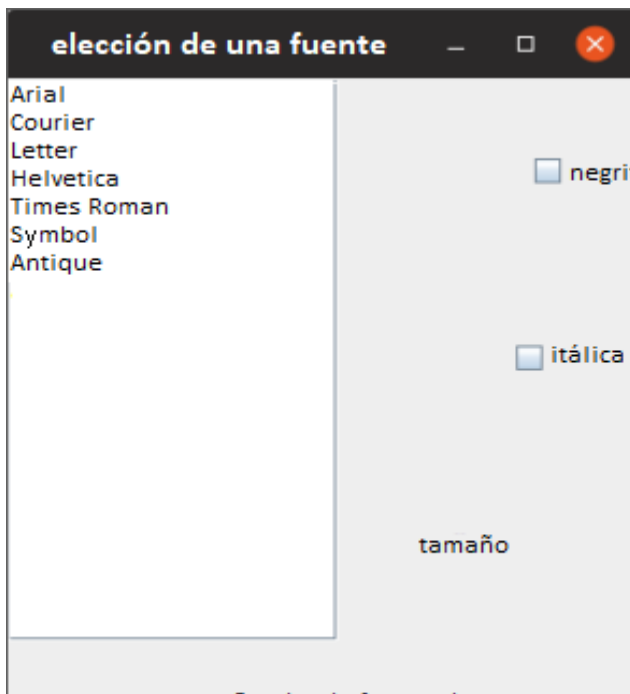
        // se agregan los elementos en el contenedor
        panel.setLayout(null);
        panel.add(desplaFuentes);
        panel.add(chkNegrita);
        panel.add(chkItalica);
        panel.add(lblTamaño);
        panel.add(cboTamaño);
        panel.add(lblEjemplo);

        // posición de los componentes.
        desplaFuentes.setBounds(24, 29, 156, 255);
        chkNegrita.setBounds(258, 78, 170, 25);
        chkItalica.setBounds(261, 139, 167, 46);
        lblTamaño.setBounds(215, 211, 106, 24);
        cboTamaño.setBounds(354, 208, 107, 32);
        lblEjemplo.setBounds(17, 309, 459, 28);
        getContentPane().add(panel);
    }
    public void mostrar() {
        this.setVisible(true);
    }
}

```

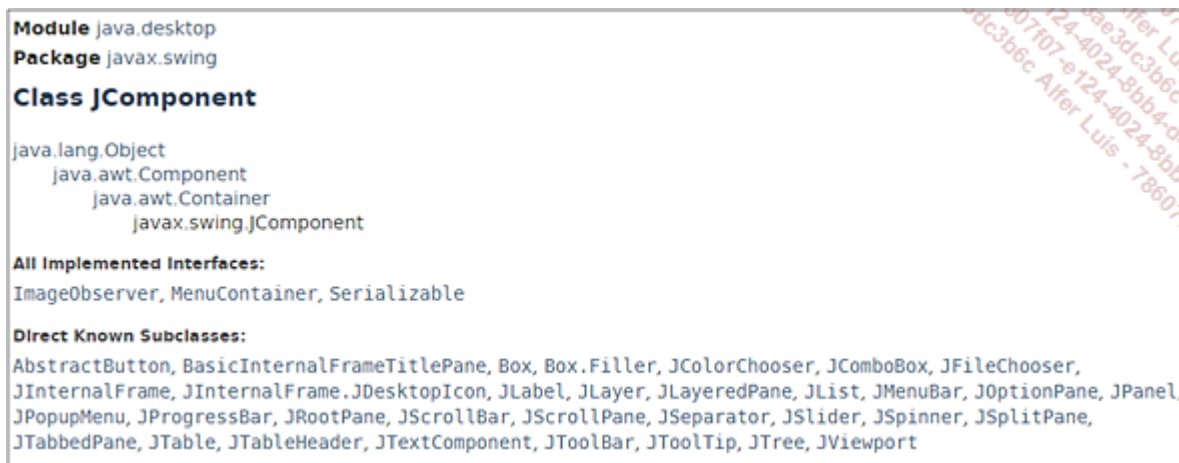
Nuestra aplicación presenta el mismo aspecto que si empleara un renderizador.

En cambio, si se redimensiona la ventana, los componentes conservan sus tamaños y posiciones y la interfaz puede presentar un aspecto incoherente.



6. Los componentes gráficos

Cada componente utilizable con Java está representado por una clase desde la cual vamos a poder crear instancias para diseñar la interfaz de la aplicación. La mayoría de los componentes Swing proceden de la clase `JComponent` y por este motivo heredan una buena cantidad de propiedades y métodos de esta clase o de las clases situadas por encima en la jerarquía de herencia.



Por lo tanto, vamos a estudiar los elementos más útiles de la clase `JComponent`.

a. La clase `JComponent`

Dimensiones y posición

Una multitud de métodos intervienen en la gestión de la posición y el tamaño de los componentes. Hay dos categorías de información disponibles para el tamaño de los componentes:

- Se accede al tamaño actual por el método `getSize`. Este último obtiene el tamaño del componente en el momento de la llamada al método. Si un renderizador se encarga del componente, es susceptible de modificar el tamaño del componente y, por lo tanto, la llamada de este método puede devolver un resultado diferente tras cada llamada. También es posible modificar el tamaño mediante el método `setSize`. Los efectos de este método pueden tener poca duración, ya que si un renderizador se encarga del componente puede redibujar en cualquier momento dicho componente con un tamaño diferente.
- El tamaño favorito es accesible por el método `getPreferredSize`. Por defecto este tamaño se calcula en función del contenido del componente. Los renderizadores suelen utilizar este método sobre todo a nivel interno. Para evitar que el tamaño del componente se recalcule según su contenido, puede emplear el método `setPreferredSize`. La información facilitada por este método la utiliza, a continuación, el método `getPreferredSize` cuando el contenedor necesita información sobre el tamaño del componente.

La posición de un componente en el interior de su contenedor se puede obtener o definir con los métodos `getLocation` o `setLocation`. Como para el método `setSize`, el renderizador puede cancelar en cualquier momento el efecto del método `setLocation` si debe volver a dibujar los componentes.

Los métodos `getBounds` y `setBounds` permiten combinar la modificación del tamaño y de la posición del componente. Estos métodos solo tienen realmente interés cuando el interior del contenedor asume por completo la presentación de los componentes, sin pasar por los servicios de un renderizador.

Apariencia de los componentes

Es posible modificar el color de fondo del componente mediante el método `setBackground`, mientras que el color del texto del componente se modifica mediante la propiedad `setForeground`. Estos métodos reciben como parámetro un objeto `Color`. Se puede obtener este objeto `Color` con las constantes definidas en la clase `Color`.

```
lstFuentes.setBackground(Color.red);  
lstFuentes.setForeground(Color.GREEN);
```

La clase `Color` propone también numerosos constructores que permiten obtener un color particular al efectuar una mezcla de los colores básicos.

```
lstFuentes.setBackground(new Color(23,67,89));  
lstFuentes.setForeground(new Color(167,86,23));
```

Se puede modificar la fuente con el método `setFont` del componente. Para la ocasión, podemos crear una nueva instancia de la clase `Font` y asignarla al componente. Para ello, utilizaremos uno de los constructores de la clase `Font` indicando el nombre de la fuente, su estilo y su tamaño.

```
lstFuentes.setFont(new Font("Serif",Font.BOLD,24));
```

Estas dos propiedades se aplican sobre el conjunto del texto que se muestra en el componente. Estos ejemplos están disponibles en el código del paquete `es.eni.editions.ihm23`.

Es posible utilizar varios colores y tipos de letra para formatear el texto del componente mediante etiquetas html. Basta, simplemente, con incluir el texto del componente entre las etiquetas `<html>` y `</html>`. En su interior, puede utilizar cualquier etiqueta de formato html. El siguiente ejemplo modifica el color del texto y muestra en itálica la etiqueta del botón.

```
// creación de un botón  
JButton b1;  
b1=new JButton("<html><div style=\"color:red;font-style:  
italic\">Rojo</div><html>") ;  
");
```

El método `setCursor` permite elegir la apariencia del cursor cuando se pasa el ratón sobre el componente. Existen varios cursores predefinidos, que son accesibles creando una instancia de la clase `Cursor` con una de las constantes predefinidas.

```
lstFuentes.setCursor(new Cursor(Cursor.HAND_CURSOR));
```

La detección de la entrada y salida del ratón sobre el componente y la modificación del cursor como consecuencia de ello la gestiona automáticamente el propio componente.

Comportamiento de los componentes

Se puede ocultar cualquiera de los componentes ubicados en un contenedor invocando el método `setVisible` con el argumento `false` o desactivarlo mediante el método `setEnabled` con el argumento `false`. En este caso, el componente sigue visible, pero aparece con un aspecto gris para indicar al usuario que está inactivo de momento.

```
chkNegrita.setVisible(false);  
chkItalica.setEnabled(false);
```

Los componentes que se encuentren ocultos o deshabilitados no podrán recibir el foco de la aplicación. Se puede comprobar llamando al método `isFocusable`, que devuelve un booleano. También se puede verificar si un componente tiene actualmente el foco utilizando el método `isFocusOwner`. Se puede colocar el foco sobre un componente sin la intervención del usuario, llamando al método `requestFocus` del componente.

Para manejar el paso del foco de un componente al otro, puede aprovechar dos eventos:

- `focusGained` indica que un componente particular recibió el foco.
- `focusLost` indica que un componente perdió el foco.

Por ejemplo, para visualizar correctamente que un componente tiene el foco, se puede emplear el código siguiente que modifica el color del texto cuando el componente recibe o pierde el foco:

```
lstFuentes.addFocusListener(new FocusListener()  
{  
  
    public void focusGained(FocusEvent arg0)  
    {  
        lstFuentes.setForeground(Color.RED);  
    }  
});
```



```

    public void focusLost(FocusEvent arg0)
    {
        lstFuentes.setForeground(Color.BLACK);
    }

});

```

Estos ejemplos están disponibles en el código del paquete `es.eni.editions.ihm23`.

b. Visualización de la información

El componente `JLabel`

Se utiliza el componente `JLabel` para visualizar en un formulario un texto que no podrá modificar el usuario. Sirve básicamente para facilitar en una leyenda los controles disponibles (por ejemplo: zonas de texto, listas desplegables...). En este caso, también puede proporcionar un acceso rápido desde el teclado para alcanzar ese componente.

El texto que muestra el componente se indica en el momento de su creación o mediante el método `setText`. Por defecto, el componente `JLabel` no dispone de borde. Se puede añadir uno invocando el método `setBorder` y pasándole el borde que se quiere emplear.

```

JLabel leyenda;
leyenda=new JLabel("nombre");
leyenda.setBorder(BorderFactory.createEtchedBorder());

```

También puede indicar la posición del texto en el componente mediante los métodos `setHorizontalAlignment` y `setVerticalAlignment` especificando una de las constantes predefinidas.

```

SwingConstants.RIGHT: alineación a la derecha
SwingConstants.CENTER: alineación en el centro
SwingConstants.LEFT: alineación a la izquierda
SwingConstants.TOP: alineación en la parte superior
SwingConstants.BOTTOM: alineación en la parte inferior

```

Estos métodos no tienen efecto visible alguno si el contenedor dimensiona

automáticamente el componente, ya que en este caso su tamaño se ajusta automáticamente a su contenido. Para que estos métodos tengan un efecto visible, deberá indicar un tamaño favorito para el componente.

```
package es.eni.editions.ihm24;

import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;

public class Pantalla extends JFrame

{
    JPanel panel;

    public Pantalla() {
        setTitle("Presentación JLabel");
        setBounds(0, 0, 220, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los componentes

        panel = new JPanel();

        JLabel leyenda=new JLabel("nombre");
        leyenda.setBorder(BorderFactory.createEtchedBorder());
        leyenda.setPreferredSize(new Dimension(200,50));
        leyenda.setHorizontalAlignment(SwingConstants.LEFT);
        leyenda.setVerticalAlignment(SwingConstants.TOP);

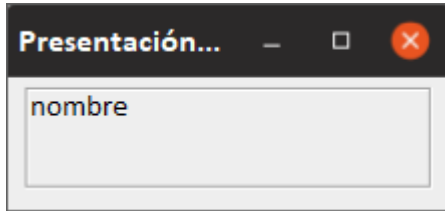
        panel.add(leyenda);

        getContentPane().add(panel);
    }
}
```

```

        public void mostrar() {
            this.setVisible(true);
        }
    }
}

```



Los componentes `JLabel` también pueden mostrar imágenes. Podemos indicar la imagen que se quiere visualizar mediante el método `setIcon`. Por supuesto, la imagen debe existir previamente. Si también se visualiza algo de texto en el componente, se puede determinar su posición en relación con la imagen mediante los métodos `setHorizontalTextPosition` y `setVerticalTextPosition`, y también el espacio entre el texto y la imagen con el método `setIconTextGap`.

```

package es.eni.editions.ihm25;

import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;

public class Pantalla extends JFrame

{
    JPanel pano;

    public Pantalla() {
        setTitle("Presentación JLabel");
        setBounds(0, 0, 300, 300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los componentes
    }
}

```

```

panel = new JPanel();

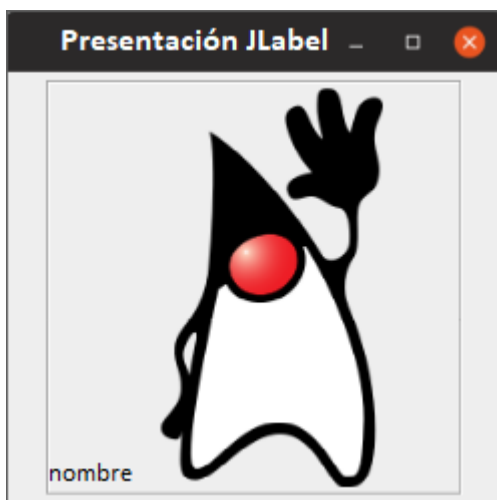
JLabel leyenda=new JLabel("nombre");
leyenda.setBorder(BorderFactory.createEtchedBorder());
leyenda.setPreferredSize(new Dimension(250,250));
//La instrucción getClass().getResource() permite
//buscar un recurso en el mismo paquete
//que la clase que se está ejecutando
leyenda.setIcon(new
ImageIcon(getClass().getResource("duke.png")));
leyenda.setHorizontalTextPosition(SwingConstants.LEFT);
leyenda.setVerticalTextPosition(SwingConstants.BOTTOM);
leyenda.setIconTextGap(40);

pano.add(leyenda);

getContentPane().add(panel);
}

public void mostrar() {
    this.setVisible(true);
}
}

```



Hemos indicado que es posible utilizar el componente JLabel como atajo de teclado para otro componente. Para ello, se deben tomar dos precauciones:

- Indicar, con el método `setDisplayedMnemonic`, el carácter utilizado como

atajo.

- Indicar al `JLabel` para qué componente juega el papel de leyenda utilizando el método `setLabelFor`.

El siguiente ejemplo muestra un formulario para introducir un nombre y apellido. Puede acceder a las áreas de entrada con los accesos directos `[Alt] n` (para el nombre) y `[Alt] a` (para el apellido).

```
package es.eni.editions.ihm26;

import java.awt.Dimension;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

public class Pantalla extends JFrame

{
    JPanel panel;

    public Pantalla() {
        setTitle("Presentación JLabel");
        setBounds(0, 0, 400, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los componentes
        panel = new JPanel();

        JLabel lNombre=new JLabel("Nombre:");
        JTextField tfNombre=new JTextField();
        tfNombre.setPreferredSize(new Dimension(100,25));
        //Acceso directo para acceder a la zona de escritura:
        lNombre.setDisplayedMnemonic('n');
        lNombre.setLabelFor(tfNombre);
```

```

        JLabel lApellido=new JLabel("Apellido:");
        JTextField tfApellido=new JTextField();
        tfApellido.setPreferredSize(new Dimension(100,25));
        //Acceso directo para acceder a la zona de escritura:
        lApellido.setDisplayedMnemonic('p');
        lApellido.setLabelFor(tfApellido);

        pano.add(lNombre);
        pano.add(tfNombre);
        pano.add(lApellido);
        pano.add(tfApellido);

        getContentPane().add(panel);
    }

    public void mostrar() {
        this.setVisible(true);
    }
}

```

El componente JProgressBar

Se utiliza este componente para informar al usuario del progreso de una acción iniciada en la aplicación. Esta información se muestra como una zona rectangular, que estará más o menos llena según el estado de progreso de la acción ejecutada.

La posición de la barra de progreso está controlada por el método `setValue`. Este método recibe un argumento que permite establecer el progreso entre los dos extremos indicados por los métodos `setMinimum` y `setMaximum`.

También se puede visualizar una leyenda incrustada en la barra de progreso. Se indica el texto de la leyenda con el método `setString` y se controla su visualización con el método `setStringPainted`.

Si resulta imposible evaluar el progreso de una operación y por lo tanto obtener un valor para facilitárselo a la barra de progreso, es posible configurar la barra en un estado indeterminado con el método `setIndeterminate`. En este caso, se muestra un cursor

que se desplaza entre los dos extremos.

El ejemplo siguiente presenta un reloj original que permite visualizar la hora con tres JProgressBar:

```
package es.eni.editions.ihm27;

import java.time.LocalDateTime;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JProgressBar;

public class Pantalla extends JFrame
{
    JPanel panel;
    JProgressBar pgbHora,pgbMinutos,pgbSegundo,pgbDespla;

    public Pantalla()
    {
        setTitle("Presentación JLabel");
        setBounds(0,0,400,100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los componentes
        pgbHora =new JProgressBar();
        pgbMinutos=new JProgressBar();
        pgbSegundo=new JProgressBar();
        pgbDespla=new JProgressBar();
        pgbHora.setMinimum(0);
        pgbHora.setMaximum(23);
        pgbMinutos.setMinimum(0);
        pgbMinutos.setMaximum(59);
        pgbSegundo.setMinimum(0);
        pgbSegundo.setMaximum(59);
        pgbHora.setString("hora");
        pgbHora.setStringPainted(true);
        pgbMinutos.setString("minuto");
        pgbMinutos.setStringPainted(true);
```

```

pgbSegundo.setString("segundo");
pgbSegundo.setStringPainted(true);
pgbDespla.setString("pasa el tiempo");
pgbDespla.setStringPainted(true);
pgbDespla.setIndeterminate(true);
panel=new JPanel();
panel.add(pgbHora);
panel.add(pgbMinutos);
panel.add(pgbSegundo);
panel.add(pgbDespla);
getContentPane().add(panel);
Thread th;
th = new Thread() {
    public void run() {
        while (true) {
            LocalTime d;
            d = LocalTime.now();
            pgbHora.setValue(d.getHour());
            pgbMinutos.setValue(d.getMinute());
            pgbSegundo.setValue(d.getSecond());
            try {
                sleep(500);
            }
            catch (InterruptedException e) {
            }
        }
    }
};
th.start();
}

public void mostrar() {
    this.setVisible(true);
}
}

```




Tenga en cuenta que, en el código, la presencia de un thread hijo permite que el reloj funcione. Este thread ejecuta un bucle infinito para actualizar la interfaz gráfica con el paso del tiempo. Es necesario realizar este procesamiento en un thread hijo; de lo contrario, no se realiza ninguna visualización. De hecho, el thread EDT procesa las tareas que se realizarán secuencialmente. Hasta que se complete una tarea, no realiza el resto de las tareas. Si el bucle infinito se ejecutó en el thread EDT, este thread solo haría esta tarea y, por lo tanto, no realizaría las otras, como actualizar la interfaz gráfica.

Tenga en cuenta también que en el thread hijo se realizan operaciones para actualizar las diferentes barras de progreso. Es posible que esto cause excepciones, ya que estas son operaciones de interthread. Sería más razonable delegar la actualización de la GUI al thread EDT. La sección sobre operaciones largas, al final de este capítulo, cubre este tema.

c. Los componentes de edición de texto

JTextField

Se utiliza el componente `JTextField` para permitir al usuario la introducción de datos. Este componente solo permite la introducción de texto en una única línea. Este componente también es capaz de gestionar la selección de texto y las operaciones con el Portapapeles. Hay distintos métodos disponibles para trabajar con este componente. Por defecto, adapta automáticamente su tamaño a su contenido, lo que puede provocar una revisualización permanente del componente. Para resolver este problema, es preferible especificar un tamaño para el componente utilizando su método `setPreferredSize` o indicando el número de columnas deseado para la visualización del texto en este componente. Esto no limita para nada el número de caracteres que se pueden introducir, sino únicamente la anchura del componente. Es posible modificar o recuperar el texto mostrado en el componente mediante los métodos `setText` o `getText`. El conjunto de ejemplos que se presentan a partir de ahora están disponibles en el paquete `es.eni.editions.ihm28`.

La gestión de la selección del texto la hace automáticamente el componente. El método

`getSelectedText` permite recuperar la cadena de caracteres actualmente seleccionada en el control. Los métodos `getSelectionStart` y `getSelectionEnd` indican respectivamente el carácter de principio de la selección (el primer carácter tiene el índice 0) y el último carácter de la selección.

También se puede efectuar la selección de texto con el método `select`, indicando el carácter de principio y de fin de la selección.

Se puede seleccionar la totalidad del texto con el método `selectAll`. Por ejemplo, se puede forzar la selección de todo el texto cuando el componente recibe el foco.

```
JTextField txt=new JTextField(10);
txt.addFocusListener(new FocusListener()
{
    //Selección del texto en el foco
    public void focusGained(FocusEvent e)
    {
        txt.selectAll();
    }
    //Deselección del texto durante la pérdida del foco
    public void focusLost(FocusEvent e)
    {
        txt.setSelectionStart(0);
        txt.setSelectionEnd(0);
    }
});
```

Para la gestión del Portapapeles, el componente `JTextField` gestiona automáticamente los atajos de teclado del sistema para las operaciones de copiar, cortar y pegar. Sin embargo, tenemos la posibilidad de invocar los métodos `copy`, `cut` y `paste` para gestionar las operaciones de copiar, cortar y pegar de otra manera, como por ejemplo mediante un menú de la aplicación o un menú contextual, tal y como se aprecia en el ejemplo siguiente:

```
JTextField txt=new JTextField(10);

// Implementación de un menú para gestionar los copiar/cortar/pegar.
JPopupMenu mnu = new JPopupMenu();
```

```

JMenuItem mnuCopiar;
JMenuItem mnuCortar;
JMenuItem mnuPegar;
mnuCopiar = new JMenuItem("Copiar");
mnuCortar = new JMenuItem("Cortar");
mnuPegar = new JMenuItem("Pegar");
mnuCopiar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //Todo copiar
        txt.selectAll();
        txt.copy();
    }
});
mnuCortar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        //Todo cortar
        txt.selectAll();
        txt.cut();
    }
});
mnuPegar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        txt.paste();
    }
});
mnu.add(mnuCopiar);
mnu.add(mnuCortar);
mnu.add(mnuPegar);
//Hacer que aparezca el menú con un clic derecho en la zona de texto
txt.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        if (e.getButton() == MouseEvent.BUTTON3) {
            mnu.show((Component) e.getSource(), e.getX(), e.getY());
        }
    }
});

```

Sin embargo, las operaciones cortar y pegar no serán posibles si el componente

`JTextField` está configurado en solo lectura con el método `setEditable(false)`, en cuyo caso, resulta imposible que el usuario modifique el texto.

Ya que todo el mundo se puede equivocar, es posible provocar la cancelación de la última modificación de texto efectuada en el control. Para ello, debemos incorporar la ayuda de un objeto `UndoManager`. Efectivamente, este último se encargará de la cancelación de la última modificación gracias a su método `undo`. Este método se invoca mediante una opción del menú contextual o mediante el atajo de teclado `[Ctrl] Z`. Solo existe un nivel de «Undo», ¡No es posible volver al texto introducido hace más de dos horas!

```
// creación del UndoManager
UndoManager udm;
udm=new UndoManager();
// asociación con el JTextField
txt.getDocument().addUndoableEditListener(udm);
// se agrega un receptor de eventos para interceptar el ctrl Z
txt.addKeyListener(new KeyAdapter()
{
    public void keyPressed(KeyEvent e)
    {
        if (e.getKeyCode() == (int) 'Z' && e.isControlDown())
        {
            udm.undo();
        }
    }
});
```

El método `getKeyCode` se usa para encontrar el código ASCII de la letra mayúscula, correspondiente a la tecla presionada. El método `isControlDown` le permite saber si se presiona la tecla `[Ctrl]`. La combinación de las dos pruebas verifica que se ha usado la combinación `[Ctrl] Z`.

JPasswordField

Este componente es en esencia un `JTextField` normal ligeramente modificado para especializarse en la introducción de contraseñas. Las dos diferencias principales realmente útiles son el método `setEchoChar`, que permite indicar qué carácter se utiliza como carácter de sustitución durante la visualización, y el método `getPassword`, que permite

obtener la contraseña introducida por el usuario.

JTextArea

Este componente ofrece funcionalidades más avanzadas que un mero `JTextField`. La primera mejora importante aportada por este componente reside en su capacidad de gestionar la introducción de datos en varias filas. Por este motivo, en el momento de la construcción de un objeto `JTextArea`, debemos especificar el número de filas y de columnas que va contener. Sólo se usan estos dos datos para el dimensionamiento del componente y no influyen en la cantidad de texto que puede contener el componente. En cambio, este componente no gestiona por sí mismo el desplazamiento del texto que se introduce. Para añadirle esta funcionalidad, debe estar ubicado en un contenedor de tipo `JScrollPane`, que se encargará del desplazamiento del contenido del `JTextArea`.

Si no se utiliza esta solución, se puede configurar el componente `JTextArea` para dividir automáticamente una fila en el momento de su visualización. El método `setLineWrap` permite activar este modo de funcionamiento. Por defecto el recorte se produce en el último carácter visible de una fila con el riesgo de ver que algunas palabras se muestran en dos filas. Para evitar este problema, se puede configurar el componente `JTextArea` para efectuar su recorte en los espacios que separan las palabras con el método `setWrapStyleWord`. Los saltos de línea añadidos por el componente no forman parte del texto y se utilizan únicamente para su visualización.

```
package es.eni;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
```

```
package es.eni.editions.ihm29;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class Pantalla extends JFrame
{
    JPanel panel;
    JTextArea txt;
    JCheckBox chkWrap,chkWrapWord;
    JScrollPane defil;

    public Pantalla()
    {
        setTitle("Presentación JTextArea");
        setBounds(0,0,600,300);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // creación de los componentes
        panel=new JPanel();
        txt=new JTextArea(10,40);
        despla=new JScrollPane(txt);
        panel.add(despla);
        chkWrap=new JCheckBox("saltos de línea automáticos");
        chkWrapWord=new JCheckBox("saltos de línea entre dos
palabras");
        chkWrap.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                txt.setLineWrap(chkWrap.isSelected());
            }
        });
        chkWrapWord.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                txt.setWrapStyleWord(chkWrapWord.isSelected());
            }
        });
    }
}
```

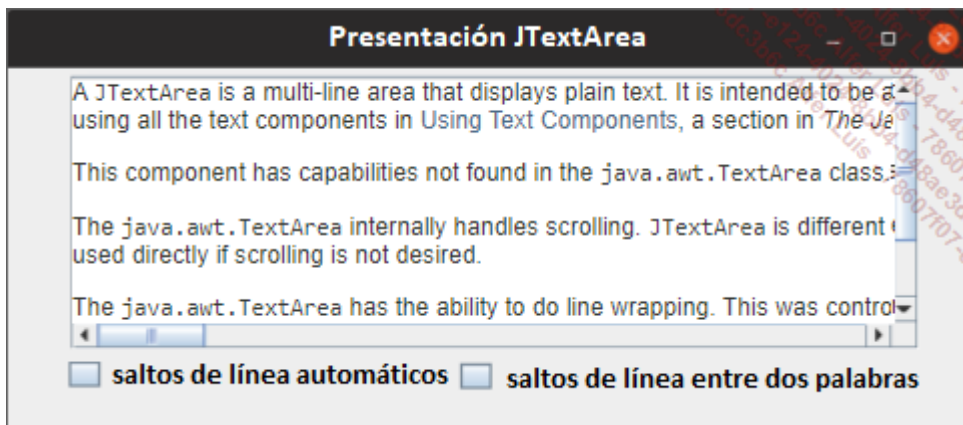
```

    }
    });
    panel.add(chkWrap);
    panel.add(chkWrapWord);
    getContentPane().add(panel);
}

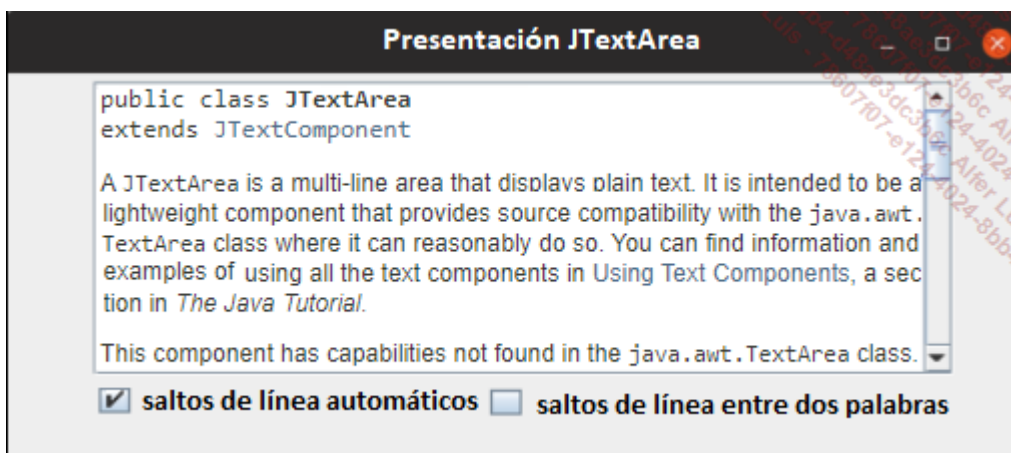
public void mostrar() {
    this.setVisible(true);
}
}

```

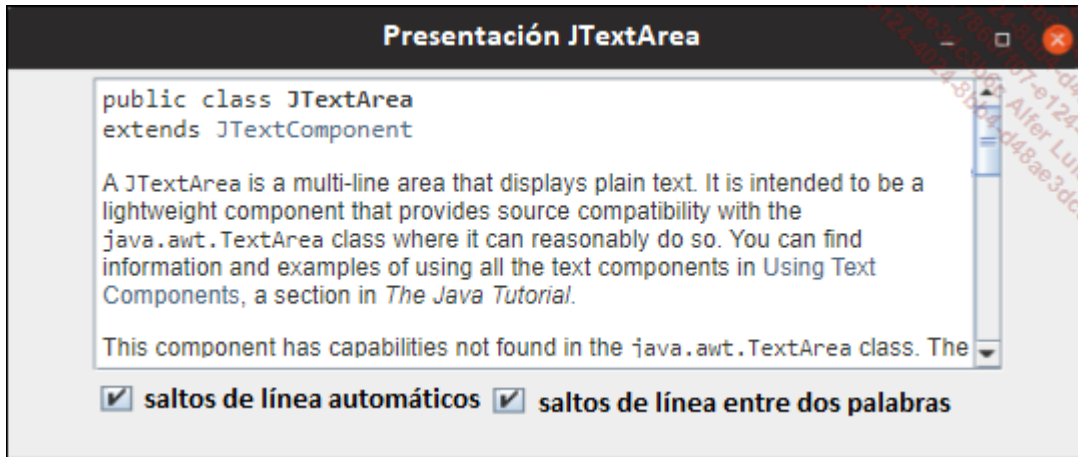
Con las opciones por defecto, las líneas demasiado largas no son visibles en su totalidad y las barras de desplazamiento se activan automáticamente aunque sean necesarias.



Con la opción `LineWrap`, se cortan las líneas automáticamente a la anchura del componente. Ya no hace falta la barra de desplazamiento horizontal. En cambio, eventualmente es necesaria una barra de desplazamiento vertical porque el número de líneas es superior.



Para mejorar la legibilidad del texto, es posible recortar palabras enteras con la opción `WrapStyleWord`.



La gestión del texto contenido en este componente se ve facilitada gracias a varios métodos específicos.

- `append(String cadena)`: añade la cadena de caracteres que se pasa como argumento al texto ya presente en el componente.
- `insert(String cadena, int position)`: inserta la cadena de caracteres que se pasa como primer argumento a la posición indicada por el segundo argumento.
- `replaceRange(String cadena, int principio, int fin)`: reemplaza la porción de texto comprendida entre el valor facilitado por el segundo argumento y el proporcionado por el tercer argumento por la cadena que se pasa como primer argumento.

d. Los componentes de activación de acciones

JButton

Se utiliza principalmente el componente `JButton` en una aplicación para lanzar la ejecución de una acción. Esta acción puede ser la ejecución de una sección de código o el cierre de un cuadro de diálogo. Como para los controles estudiados hasta ahora, el título del botón se puede modificar con el método `setText` del componente.

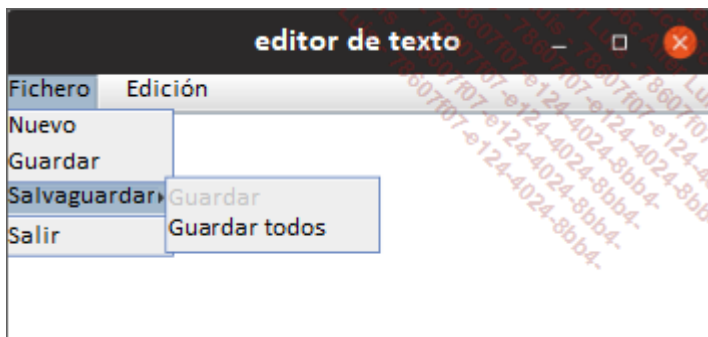
Este componente puede también contener una imagen. Se gestiona dicha imagen de la misma manera que para el componente `JLabel`. Se suele asociar casi siempre este componente a un receptor de eventos que implementa la interfaz `ActionListener` con

el fin de gestionar los clics en el botón.

JMenuBar, JMenu, JMenuItem, JPopupMenu, JSeparator

Este conjunto de componentes va a permitir gestionar los menús de aplicación o los menús contextuales. El componente JMenuBar constituye el contenedor que va a acoger los menús representados por JMenu, que, a su vez, van a contener elementos de menús representados por JMenuItem. El componente JMenuBar se comporta como un contenedor para los JMenu. Los JMenu también se van a comportar como un contenedor para JMenuItem u otros JMenu. Por lo tanto, el diseño de un menú va a consistir en crear instancias de estas diferentes clases y a continuación asociarlas entre sí. Para terminar, el JMenuBar obtenido se ubica en su contenedor, que no es otro que el JFrame de la aplicación. Naturalmente también hace falta pensar en tratar los eventos producidos por sus elementos. En general, solo se asocia los JMenuItem a receptores de eventos. Se comportan como los JButton. De hecho, tienen un pariente común, ya que los dos heredan de la clase AbstractButton. El diseño de un menú es muy complejo. La única dificultad reside en la cantidad de código relativamente importante necesaria.

Para ilustrar esto, a continuación mostramos el código de un editor de texto rudimentario que posee los menús siguientes:



```
package es.eni.editions.ihm30;
```

```
import java.awt.BorderLayout
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSeparator;
import javax.swing.JTextArea;

public class Pantalla extends JFrame
{
    JPanel panel;
    JTextArea txt;
    JScrollPane defil;
    JMenuBar barra;
    JMenu mnuFichero,mnuEdición,mnuSalvaguardar;
    JMenuItem mnuNuevo,mnuAbrir,mnuGuardar,mnuGuardarComo,
mnuSalir;
    JMenuItem mnuCopiar,mnuCortar,mnuPegar;
    File fichero;

    public Pantalla()
    {
        setTitle("editor de texto");
        setBounds(0,0,600,400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

```
// creación de los componentes
panel=new JPanel();
panel.setLayout(new BorderLayout());
txt=new JTextArea();
despla=new JScrollPane(txt);
panel.add(despla,BorderLayout.CENTER);
getContentPane().add(panel);

// creación de los componentes de los menús
barra=new JMenuBar();
mnuFichero=new JMenu("Fichero");
mnuEdición=new JMenu("Edición");
mnuSalvaguardar=new JMenu("Salvaguardar");
mnuNuevo=new JMenuItem("Nuevo");
mnuAbrir=new JMenuItem("Abrir");
mnuGuardar=new JMenuItem("Guardar");
mnuGuardar.setEnabled(false);
mnuGuardarComo=new JMenuItem("Guardar como");
mnuCopiar=new JMenuItem("Copiar");
mnuCortar=new JMenuItem("Cortar");
mnuPegar=new JMenuItem("Pegar");
mnuSalir=new JMenuItem("Salir");

// asociación de los elementos
barra.add(mnuFichero);
barra.add(mnuEdición);
mnuFichero.add(mnuNuevo);
mnuFichero.add(mnuAbrir);
mnuFichero.add(mnuSalvaguardar);
mnuSalvaguardar.add(mnuGuardar);
mnuSalvaguardar.add(mnuGuardarComo);
mnuFichero.add(new JSeparator());
mnuFichero.add(mnuSalir);
mnuEdición.add(mnuCopiar);
mnuEdición.add(mnuCortar);
mnuEdición.add(mnuPegar);

// asociación del menú con la JFrame
setJMenuBar(barra);

// los receptores de eventos asociados a los diferentes
// menús
```

```

mnuNuevo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        fichero=null;
        txt.setText("");
        mnuGuardar.setEnabled(false);
    }

});
mnuAbrir.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        JFileChooser dlg;
        dlg=new JFileChooser();
        dlg.showDialog(null,"Abrir");
        fichero=dlg.getSelectedFile();
        FileInputStream in;
        try
        {
            in=new FileInputStream(fichero);
            BufferedReader br;
            br=new BufferedReader(new
InputStreamReader(in));
            String línea;
            txt.setText("");
            while ((línea=br.readLine())!=null)
            {
                txt.append(línea + "\r\n");
            }
            br.close();
            mnuGuardar.setEnabled(true);
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
    }
}

```

```
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
});
mnuSalir.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});
mnuCopiar.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.copy();
    }
});
mnuCortar.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.cut();
    }
});
mnuPegar.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.paste();
    }
});
mnuGuardar.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
```

```

        try
        {
            PrintWriter pw;
            pw=new PrintWriter(fichero);
            pw.write(txt.getText());
            pw.close();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
    }
});
mnuGuardarComo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent arg0)
    {
        try
        {
            JFileChooser dlg;
            dlg=new JFileChooser();
            dlg.showDialog(null,"guardar como");
            fichero=dlg.getSelectedFile();
            PrintWriter pw;
            pw=new PrintWriter(fichero);
            pw.write(txt.getText());
            pw.close();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
    }
});
}

public void mostrar() {
    this.setVisible(true);
}

```

```
}  
}
```

JToolBar

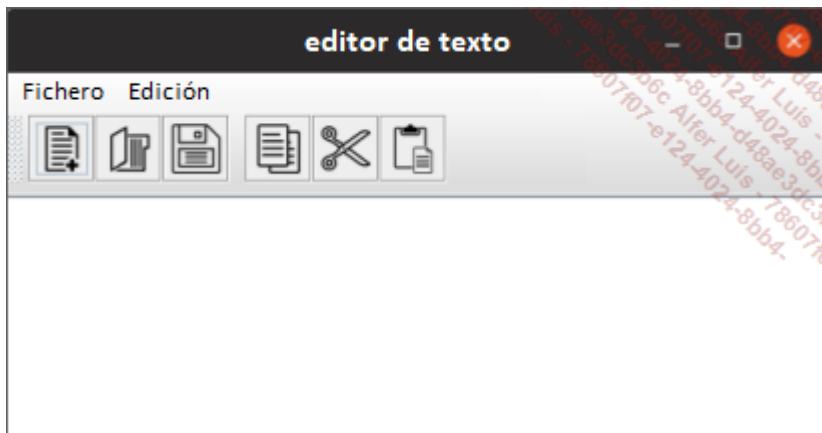
Este componente sirve de contenedor para los elementos que constituyen una barra de herramientas. Se añaden elementos al componente `JToolBar` como para cualquier otro contenedor. Se suelen colocar en este componente unos `JButton` sin título, pero que muestran imágenes. Facilitan el acceso a alguna funcionalidad ya disponible, generalmente, a través de algún elemento de menú. En este sentido, suelen compartir el mismo receptor de eventos. Este componente presenta la particularidad de que puede desplazarlo el usuario hacia cualquiera de los bordes de la ventana, e incluso en algunos casos al exterior de la ventana. Para que este mecanismo funcione correctamente, el componente `JToolBar` debe ubicarse dentro de un contenedor que utilice un `BorderLayout` como renderizador. En este caso, se le ubica en alguno de los bordes (`NORTH`, `SOUTH`, `WEST`, `EAST`) y debe ser el único colocado en esta zona. Se puede desactivar esta funcionalidad invocando el método `setFloatable(false)`. La barra de herramientas se queda entonces en su posición de origen. El código siguiente añade una barra de herramientas al editor de texto de la sección anterior. Este ejemplo está disponible en el paquete `es.eni.editions.ihm31`.

```
JToolBar tlbr;  
tlbr=new JToolBar();  
JButton btnNuevo,btnAbrir,btnGuardar;  
JButton btnCopiar,btnCortar,btnPegar;  
// creación de los botones  
btnNuevo=new JButton(new  
ImageIcon(getClass().getResource("new.png")));  
btnAbrir=new JButton(new  
ImageIcon(getClass().getResource("open.png")));  
btnGuardar=new JButton(new  
ImageIcon(getClass().getResource("save.png")));  
btnCopiar=new JButton(new  
ImageIcon(getClass().getResource("copy.png")));  
btnPegar=new JButton(new  
ImageIcon(getClass().getResource("paste.png")));  
btnCortar=new JButton(new
```

```

ImageIcon(getClass().getResource("cut.png"))));
    // se agregan los botones a la barra de herramientas
    tlbr.add(btnNuevo);
    tlbr.add(btnAbrir);
    tlbr.add(btnGuardar);
    tlbr.addSeparator();
    tlbr.add(btnCopiar);
    tlbr.add(btnCortar);
    tlbr.add(btnPegar);
    // se agrega la barra de herramientas en su contenedor
    panel.add(tlbr, BorderLayout.NORTH);
    // reutilización de receptores de eventos ya asociados a los menús
    btnNuevo.addActionListener(mnuNuevo.getActionListeners()[0]);
    btnAbrir.addActionListener(mnuAbrir.getActionListeners()[0]);
    btnGuardar.addActionListener(mnuGuardar.
    getActionListeners()[0]);
    btnCopiar.addActionListener(mnuCopiar.getActionListeners()[0]);
    btnCortar.addActionListener(mnuCortar.getActionListeners()[0]);
    btnPegar.addActionListener(mnuPegar.getActionListeners()[0]);

```



e. Los componentes de selección

JCheckBox

Se utiliza el componente `JCheckBox` para proponer al usuario diferentes opciones, entre las cuales podrá elegir ninguna, una o varias. El componente `JCheckBox` presenta dos estados: marcado, cuando la opción está seleccionada, o no marcado, cuando la opción no está seleccionada. Es posible verificar o modificar el estado de la opción marcapable mediante los métodos `isSelected` y `setSelected`. Como otros muchos

componentes, puede tener un título y una imagen. Sin embargo, la gestión de la imagen difiere un poco de los `JLabel`, ya que viene a remplazar el dibujo de la opción marcabable. En este caso, para poder distinguir entre una `JCheckBox` marcada y otra no marcada, debe asociar una segunda imagen correspondiente al estado marcado. Se indica esta segunda imagen con el método `setSelectedIcon`.

El código siguiente, que puede encontrar en el paquete `es.eni.editions`
`.ihm32`, permite elegir el estilo de la fuente empleada en nuestro editor de texto.

```
...
JPanel opciones;
GridLayout gl;
opciones=new JPanel();
gl=new GridLayout(2,1);
opciones.setLayout(gl);
chkNegrita=new JCheckBox("Negrita");
chkItalica=new JCheckBox("Itálica");
opciones.add(chkNegrita);
opciones.add(chkItalica);
chkNegrita.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changeFuente();
    }
});
chkItalica.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changeFuente();
    }
});

panel.add(opciones,BorderLayout.SOUTH);
...

public void changeFuente()
```

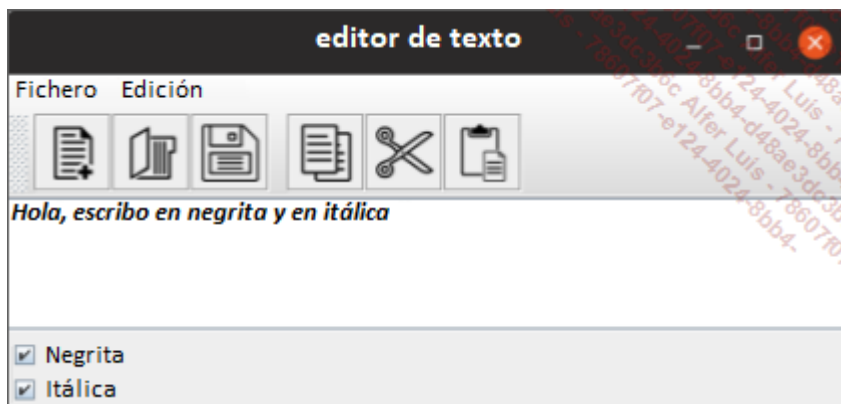
```

{
    int atributos;
    atributos=0;
    if (chkNegrita.isSelected())
    {
        atributos=atributos+Font.BOLD;
    }
    if (chkItalica.isSelected())
    {
        atributos=atributos+Font.ITALIC;
    }

    Font fuente;
    fuente=new
Font(txt.getFont().getName(),atributos,txt.getFont().getSize());
    txt.setFont(fuente);
}

```

Ahora, la interfaz gráfica tiene el siguiente aspecto:



JRadioButton

El componente `JRadioButton` permite también proponer al usuario diferentes opciones entre las cuales solo podrá seleccionar una. Como su nombre indica, este control funciona como los botones que permiten seleccionar una emisora en una radio (¡no puede escuchar tres emisoras a la vez!). Las características de este componente son idénticas a las disponibles en el componente `JCheckBox`.

Para poder funcionar correctamente, los componentes `JRadioButton` se deben agrupar, como es lógico. La clase `ButtonGroup` proporciona esta funcionalidad. Funciona de

modo que, entre todos los `JRadioButton` que se le han confiado, mediante su método `add`, solo uno podrá estar seleccionado. Para agrupar físicamente unos `JRadioButton`, estos deben estar ubicados en un contenedor, como, por ejemplo, un `JPanel`. Para visualizar correctamente esta agrupación, se suele añadir un borde en el `JPanel` y, de manera opcional, una leyenda. Es lo que vamos a hacer añadiendo dos conjuntos de botones que permiten elegir el color del texto y el color de fondo del editor. Puede encontrar el siguiente código en el paquete `es.eni.editions.ihm33`.

```
JRadioButton optFondoRojo,optFondoVerde,optFondoAzul;
JRadioButton optRojo,optVerde,optAzul;
JPanel color,colorFondo;
ButtonGroup grpColor,grpColorFondo;
// creación de los botones
optRojo=new JRadioButton("Rojo");
optVerde=new JRadioButton("Verde");
optAzul=new JRadioButton("Azul");
optFondoRojo=new JRadioButton("Rojo");
optFondoVerde=new JRadioButton("Verde");
optFondoAzul=new JRadioButton("Azul");
// agrupación lógica de los botones
grpColor=new ButtonGroup();
grpColor.add(optRojo);
grpColor.add(optVerde);
grpColor.add(optAzul);
grpColorFondo=new ButtonGroup();
grpColorFondo.add(optFondoRojo);
grpColorFondo.add(optFondoVerde);
grpColorFondo.add(optFondoAzul);
panel.add(color, BorderLayout.WEST);
// agrupación física de los botones
color=new JPanel();
color.setLayout(new GridLayout(0,1));
color.add(optRojo);
color.add(optVerde);
color.add(optAzul);
colorFondo=new JPanel();
colorFondo.setLayout(new GridLayout(0,1));
```

```
colorFondo.add(optFondoRojo);
colorFondo.add(optFondoVerde);
colorFondo.add(optFondoAzul);
panel.add(colorFondo, BorderLayout.EAST);
// se agrega un borde con título
color.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"Texto"));
colorFond.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"Fondo"));
// referencia a los receptores de eventos
optAzul.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setForeground(Color.BLUE);
    }
});
optVerde.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setForeground(Color.GREEN);
    }
});
optRojo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setForeground(Color.RED);
    }
});
optFondoAzul.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setBackground(Color.BLUE);
    }
});
```

```

optFondoRojo.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setBackground(Color.RED);
    }
});
optFondoVerde.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        txt.setBackground(Color.GREEN);
    }
});

```

La interfaz gráfica ahora tiene el siguiente aspecto:



JList

El componente `JList` propone al usuario una lista de opciones en la cual podrá seleccionar una o varias de ellas. El componente gestiona automáticamente la visualización de los elementos; sin embargo, no es capaz de asegurar su desplazamiento. Este componente debe ubicarse dentro de un contenedor que pueda encargarse de esta funcionalidad, como por ejemplo un `JScrollPane`. Se pueden proporcionar los elementos visualizados en la lista en el momento de la creación del componente pasando como parámetro al constructor un array de objetos o un `Vector`. Los métodos `toString` de cada uno de los objetos serán usados por el componente `JList` para poder

mostrar el elemento. Con esta solución, la lista se crea en solo lectura y no es posible añadir otros elementos después. Para disponer de una lista en la cual sea posible añadir otros elementos, hay que emplear un objeto de tipo `DefaultListModel` al que se le va a confiar la tarea de gestionar los elementos de la lista. Este objeto posee numerosos métodos, similares a los métodos disponibles para un `Vector`, que permiten gestionar los elementos. Si deseamos usar una lista dinámica, debemos pasar al constructor de la clase `JList` un objeto de este tipo.

Se puede diseñar la lista para autorizar diferentes tipos de selección. La elección del tipo de selección se realiza con el método `setSelectionMode`, al cual se le pasa como argumento una de las constantes siguientes:

- `ListSelectionMode.SINGLE_SELECTION`: solo se puede seleccionar un único elemento de la lista a la vez.
- `ListSelectionMode.SINGLE_INTERVAL_SELECTION`: se pueden seleccionar varios elementos, pero deben ir seguidos en la lista.
- `ListSelectionMode.MULTIPLE_INTERVAL_SELECTION`: se pueden seleccionar varios elementos en la lista y no tienen que ser consecutivos.

La recuperación del elemento o de los elementos seleccionados puede efectuarse de dos maneras diferentes. Podemos obtener el índice o los índices de los elementos seleccionados con los métodos `getSelectedIndex` o `getSelectedIndices`. Estos métodos devuelven un entero o un array de enteros que representan el índice o los índices de los elementos seleccionados. El objeto o los objetos seleccionados están disponibles mediante los métodos `getSelectedValue` y `getSelectedValues`. De manera similar, estos dos métodos devuelven el objeto seleccionado o un array que contiene los objetos seleccionados. Es posible comprobar la presencia de un elemento seleccionado en la lista mediante el método `isSelected`. Se puede cancelar la selección con el método `clearSelection`. La modificación de la selección en la lista activa un evento de tipo `ValueChanged`. Es posible gestionar este evento asociando a la lista un receptor de eventos de tipo `ListSelectionListener` con el método `addListSelectionListener`. Hay que tener cuidado con este evento, ya que se produce varias veces seguidas cuando se selecciona un elemento. El primer evento corresponde a la deselección del elemento anterior, y el segundo, a la selección del elemento actual. Por lo tanto, es este último el que hay que tener en cuenta. Para ello, el

argumento `ListSelectionEvent` disponible con este evento dispone del método `getValueIsAdjusting`, que permite saber si la selección terminó o si se encuentra en un estado transitorio. Para probar el funcionamiento, vamos a añadir a nuestro editor de texto una lista que permite seleccionar una fuente de caracteres. Puede encontrar el siguiente código en el paquete `es.eni.editions.ihm34`.

```
JList<String> fuentes;
...
...
...
JScrollPane desplaFuentes;

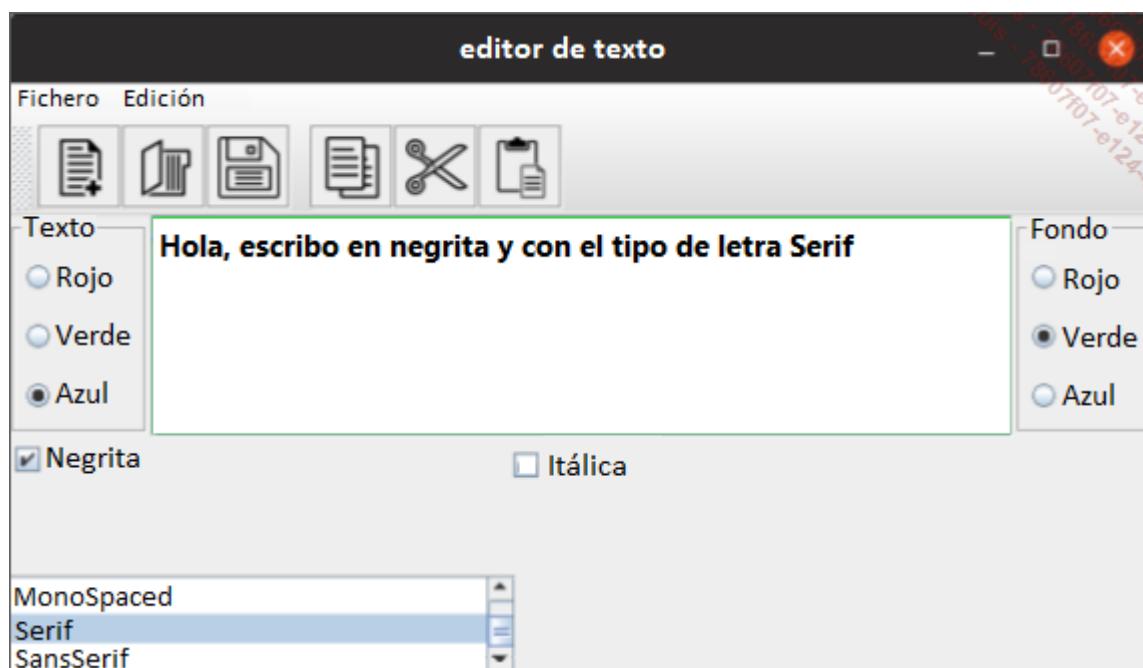
String[] nombresFuentes={"Dialog","DialogInput","Monospaced","Serif",
"SansSerif"};
fuentes=new JList<String> (nombresFuentes);
fuentes.setSelectedIndex(0);
desplaFuentes=new JScrollPane(fuentes);
desplaFuentes.setPreferredSize(new Dimension(100,60));
opciones.add(desplaFuentes);
fuentes.addListSelectionListener(new ListSelectionListener()
{
    public void valueChanged(ListSelectionEvent e)
    {
        if (!e.getValueIsAdjusting())
        {
            changeFuente();
        }
    }
});
...
...
...
public void changeFuente()
{
    int atributos;
    atributos=0;
    if (chkNegrita.isSelected())
    {
```

```

        atributos=atributos+Font.BOLD;
    }
    if (chkItalica.isSelected())
    {
        atributos+=Font.ITALIC;
    }
    Font fuente;
    fuente=new Font(fuentes.getSelectedValue().toString(),
    atributos,txt.getFont().getSize());
    txt.setFont(fuente);
}

```

La interfaz gráfica ahora tiene el aspecto siguiente:



JCombobox

Se puede asimilar este componente a la asociación de otros tres componentes:

- un JTextField
- un JButton
- un JList

El usuario puede elegir un elemento en la lista que aparece cuando hace clic en el botón o bien introducir directamente en la zona de texto la información esperada si esta no está disponible en la lista. Este control presenta dos modos de funcionamiento. Podemos

autorizar la selección de un único elemento en la lista al configurar la zona de texto en modo de solo lectura con el método `setEditable(false)`, o bien autorizar la selección en la lista, o la introducción en la zona de texto con el método `setEditable(true)`. La creación de una instancia de este componente sigue las mismas pautas que la creación de un `JList`. La recuperación del elemento seleccionado se efectúa de manera similar a un `JList`. Hay que tener cuidado, sin embargo, ya que no hay selecciones múltiples en este tipo de componente.

La detección de la selección de un nuevo elemento se puede hacer gestionando el evento `actionPerformed`. Este evento se activa en el momento de la selección de un elemento en la lista o cuando el usuario valida la introducción con la tecla [Intro]. Terminamos el editor de texto proponiendo al usuario la selección de un tamaño de fuente de caracteres. Puede encontrar el siguiente código en el `paquetes.eni.editions.ihm35`.

```
JComboBox<Integer> cboTamaño;
...
...
Integer tamaños[]={ "10", "12", "14", "16", "20" };
cboTamaño=new JComboBox(tamaños);
cboTamaño.setEditable(true);
opciones.add(cboTamaño);
cboTamaño.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        changeFuente();
    }
});
...
...
...
public void changeFuente()
{
    int atributos;
```

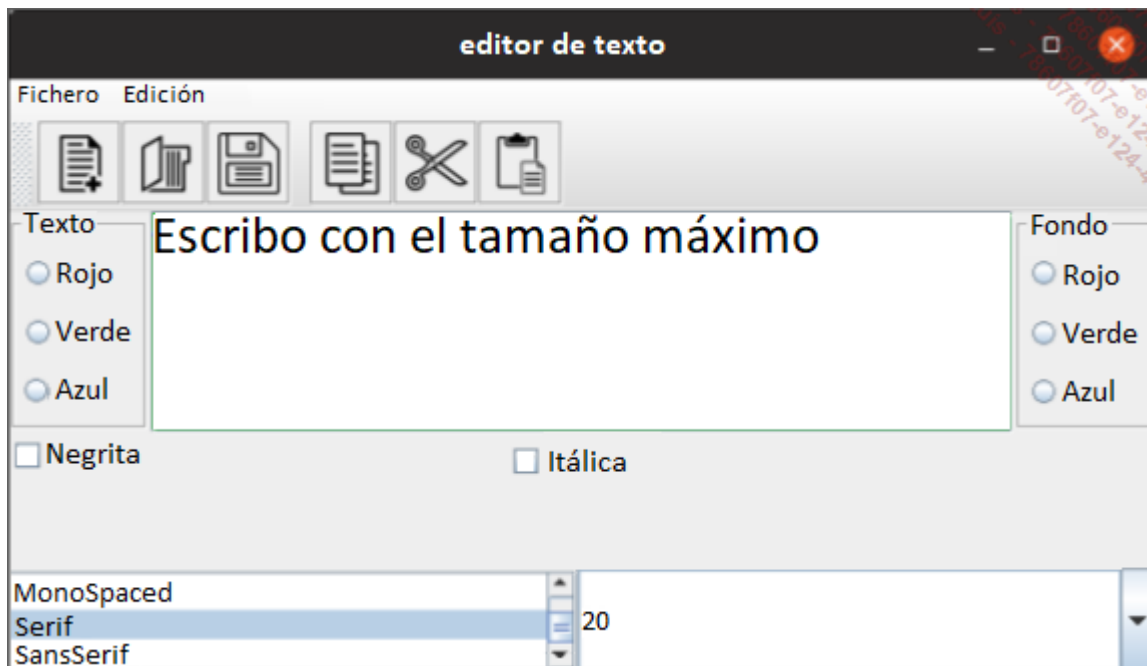
```

    atributos=0;
    if (chkNegrita.isSelected())
    {
        atributos=atributos+Font.BOLD;
    }
    if (chkItalica.isSelected())
    {
        atributos=atributos+Font.ITALIC;
    }

    Font fuente;
    fuente=new Font(fuentes.getSelectedValue().toString(),
    atributos,cboTamaño.getItemAt(cboTamano.getSelectedIndex()));
    txt.setFont(fuente);
}

```

La interfaz gráfica ahora tiene el siguiente aspecto:



7. Los cuadros de diálogo

Los cuadros de diálogo son ventanas que tienen una función especial en la aplicación. Se utilizan, por lo general, para solicitar al usuario que introduzca información, presentarle un mensaje o hacerle una pregunta. Para asegurarse de que el usuario ha procesado el cuadro

de diálogo antes de continuar con la aplicación, este se muestra en modo modal, es decir, el resto de la aplicación se bloquea mientras se está mostrando el cuadro de diálogo. Para evitar tener que recrear cada vez un nuevo cuadro de diálogo, podemos acceder a varios cuadros de diálogo predefinidos.

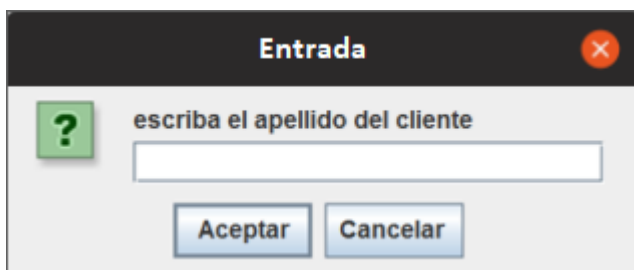
Están disponibles mediante métodos `static` que se hallan en la clase `JOptionPane`. Puede encontrar los próximos ejemplos de código en el paquete `es.eni.editions.ihm36`.

a. El cuadro para introducir información

El cuadro para introducir información permite solicitar al usuario que introduzca una cadena de caracteres. Esta funcionalidad está disponible mediante la función `showInputDialog` de la clase `JOptionPane`. Tenemos a nuestra disposición varias versiones de esta función, que nos permiten configurar la representación del cuadro de diálogo. La más sencilla recibe como parámetro una cadena de caracteres que representa el mensaje que muestra el cuadro de diálogo. Generalmente, este mensaje indica la naturaleza de la información que debe introducir el usuario. El siguiente código:

```
String apellido;  
apellido=JOptionPane.showInputDialog("escriba el apellido  
del cliente");
```

muestra el siguiente cuadro de diálogo:



Una versión más completa permite escoger:

- el modo de visualización: ventana autónoma o ventana interna de un componente. En este caso, se le pasa como primer parámetro el componente afectado, o el valor `null` si la ventana es autónoma.
- el mensaje que se muestra en el cuadro de diálogo. Es el segundo parámetro que se utiliza para este propósito.

- el título del cuadro de diálogo como tercer parámetro.
- el tipo de icono que se visualizará en el cuadro de diálogo. Una serie de constantes definidas en la clase permiten realizar dicha selección.

Constante	Icono
ERROR_MESSAGE	
INFORMATION_MESSAGE	
WARNING_MESSAGE	
QUESTION_MESSAGE	
PLAIN_MESSAGE	Ningún icono

He aquí un ejemplo de uso de esta versión más evolucionada y el resultado correspondiente.

```
String apellido;
apellido=JOptionPane.showInputDialog(null,"escriba el apellido
del cliente",
"búsqueda de cliente",JOptionPane.WARNING_MESSAGE);
```



Existe otra versión que tiene un funcionamiento ligeramente distinto. No proporciona una zona de libre introducción de texto, sino una lista de objetos entre los que el usuario debe realizar su elección. La lista representa los objetos invocando al método `toString` de cada uno de ellos. Recibe como parámetros:

- El modo de visualización: ventana autónoma o ventana interna a un componente. En este caso, se le pasa como primer parámetro el componente afectado o el valor `null` si la ventana es autónoma.
- El mensaje que muestra el cuadro de diálogo. El segundo parámetro se utiliza con este propósito.
- El título del cuadro de diálogo como tercer parámetro.
- El tipo de icono que se visualizará en el cuadro de diálogo. Una serie de constantes definidas en la clase permiten realizar dicha selección.
- Un icono que permite remplazar al icono por defecto o `null` para conservar el icono por defecto.
- Una tabla de objetos que forman la lista de opciones que se presentan al usuario.
- El objeto que representa la opción por defecto seleccionada en la visualización del cuadro de diálogo.

Esta función devuelve el objeto seleccionado o `null` si se utiliza el botón **Cancelar** para cerrar el cuadro de diálogo.

El siguiente ejemplo permite seleccionar una persona entre las propuestas en la lista.

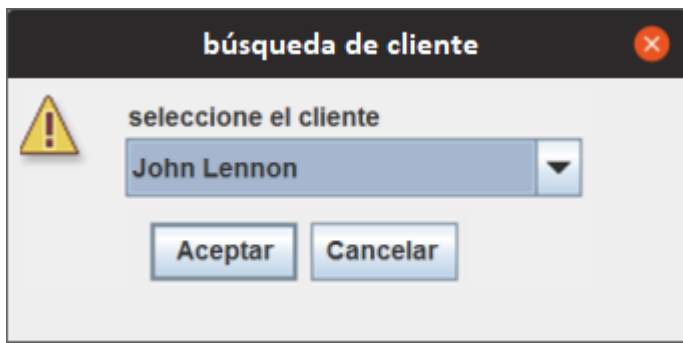
```

Persona[] seleccion;
seleccion=new Persona[5];
seleccion[0] = new Persona("Wayne", "John",LocalDate.of(1907,5,26));
seleccion[1] = new Persona("McQueen",
"Steve",LocalDate.of(1930,3,24));
seleccion[2] = new Persona("Lennon", "John",LocalDate.of(1940,10,9));
seleccion[3] = new Persona("Gibson", "Mel",LocalDate.of(1956,1,3));
seleccion[4] = new Persona("Willis", "Bruce",LocalDate.of(1955,3,19));

Persona seleccionada;

seleccionada=(Persona)JOptionPane.showInputDialog(null,
"seleccionar el cliente","búsqueda de cliente",
JOptionPane.WARNING_MESSAGE,null,seleccion,seleccion[1]);

```



b. El cuadro de mensaje

El cuadro de mensaje permite mostrar una información al usuario; está disponible mediante la función `showMessageDialog`.

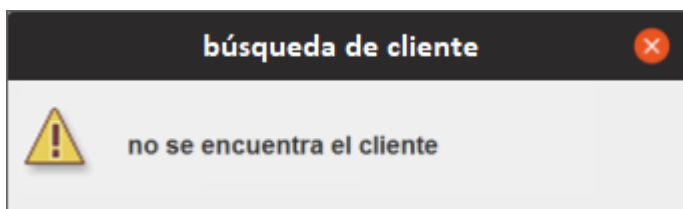
Esta función recibe los siguientes parámetros:

- El modo de visualización: como ventana autónoma o interna a un componente. En este caso, se pasa como primer parámetro el componente afectado o el valor `null` si la ventana es autónoma.
- El mensaje que se muestra en el cuadro de mensaje. El segundo parámetro se utiliza con este propósito.
- El título del cuadro de diálogo como tercer parámetro.
- El tipo de icono que se muestra en el cuadro de mensaje. Una serie de constantes definidas en la clase permiten realizar dicha selección. Estas constantes son idénticas a las que se utilizan en el cuadro de diálogo para introducir información.

El siguiente código:

```
JOptionPane.showMessageDialog(null,"no se encuentra el cliente",  
"búsqueda de cliente ",JOptionPane.WARNING_MESSAGE);
```

muestra el siguiente cuadro de mensaje:



c. El cuadro de confirmación

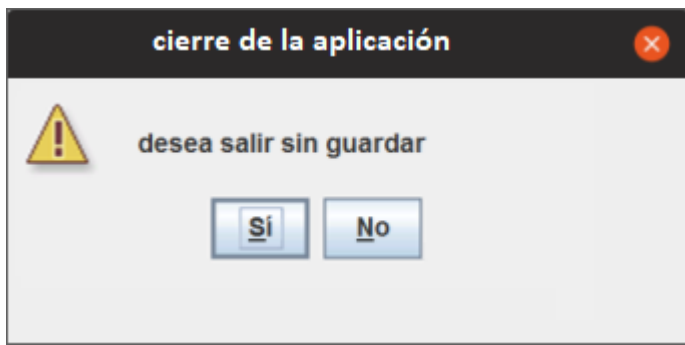
Este cuadro de diálogo se utiliza para realizar una pregunta al usuario y permitirle responder mediante uno de los botones que proporciona. Los botones disponibles están predefinidos por el cuadro de confirmación. La función `showConfirmDialog` provoca la apertura del cuadro de confirmación. Devuelve un valor entero que permite identificar el botón utilizado para cerrar el cuadro de diálogo y, en consecuencia, obtener la respuesta del usuario.

Recibe los siguientes parámetros:

- El modo de visualización: como ventana autónoma o interna a un componente. En este caso, se pasa como primer parámetro el componente afectado o el valor `null` si la ventana es autónoma.
- El mensaje que se muestra en el cuadro de confirmación. El segundo parámetro se utiliza con este propósito.
- El título del cuadro de diálogo como tercer parámetro.
- La combinación de botones que se ha de mostrar. Las constantes `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` u `OK_CANCEL_OPTION` permiten configurar este parámetro.
- El tipo de icono que se muestra en el cuadro de mensaje. Una serie de constantes definidas en la clase permiten realizar dicha selección. Estas constantes son idénticas a las que se utilizan en el cuadro de diálogo para introducir información.

Esta función devuelve un valor entero que se compara, a continuación, con las siguientes constantes `YES_OPTION`, `OK_OPTION`, `NO_OPTION`, `CANCEL_OPTION` o `CLOSE_OPTION` para determinar la respuesta por parte del usuario.

```
int respuesta;  
respuesta=JOptionPane.showConfirmDialog(null,"desea salir  
sin guardar",  
    "cierre de la aplicación", JOptionPane.YES_NO_OPTION,  
    JOptionPane.WARNING_MESSAGE);
```



8. Los tratamientos largos

Esta última sección presenta un elemento básico que es necesario entender antes de comenzar a crear aplicaciones más complejas con la API Swing.

Cuando el usuario realiza una acción, a veces pueden llevar mucho tiempo las operaciones necesarias para procesar la respuesta. La causa puede ser múltiple:

- Un cálculo complejo,
- La manipulación de una gran cantidad de datos,
- Conexión a un servidor remoto, etc.

Por defecto, el procesamiento es compatible con el thread EDT. Recuerde que el thread EDT es responsable de administrar la visualización de la aplicación y de recibir eventos relacionados con las acciones del usuario. Este thread procesa secuencialmente las tareas que se han de realizar. Entre estas, hay tareas de refresco de GUI que ocurren continuamente (cambiar el tamaño de la pantalla, entrada de datos en un área, cambiar la apariencia de un componente cuando se pasa el ratón, etc.). Pero también hay tareas correspondientes a las acciones del usuario que solicita el procesamiento de negocio (validación de un formulario, selección de un criterio de una lista desplegable, etc.). Mientras procesa estas tareas, las tareas de actualización de la GUI quedan en espera. ¡Este es el problema!

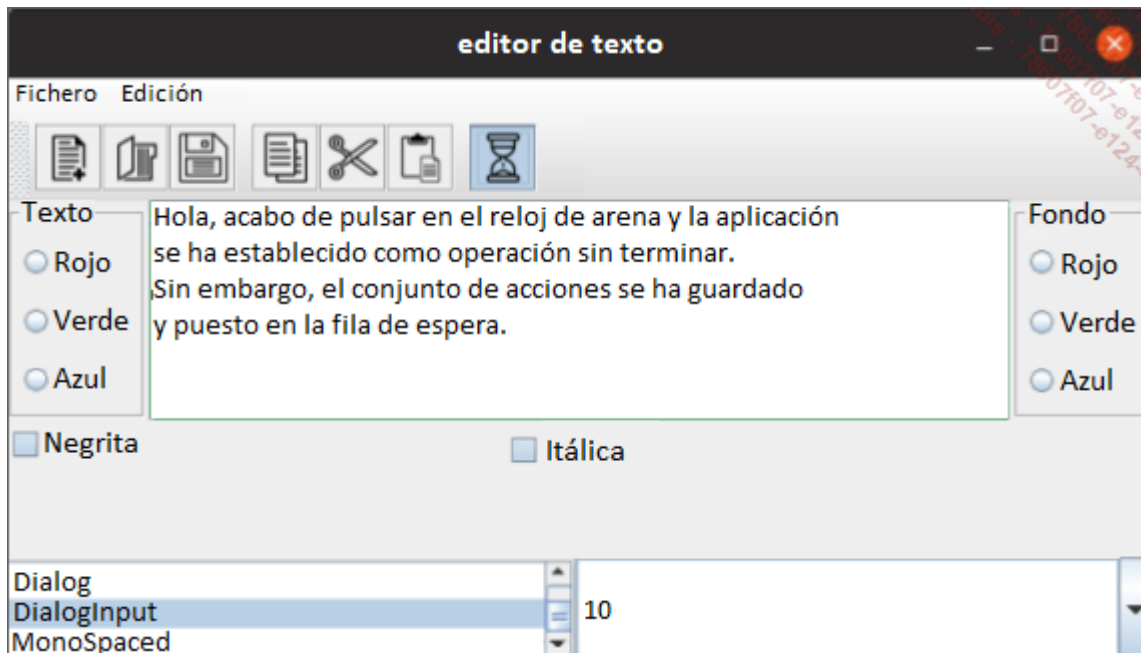
El usuario se encuentra frente a una aplicación que parece estar congelada porque ya no responde a sus acciones. No hay nada. Las tareas que se han de realizar se colocan en una cola y se procesarán cuando se completen las tareas anteriores. Pero esto no es aceptable para el usuario. Se debe encontrar una solución para que la aplicación siempre permanezca fluida a los ojos del usuario.

La solución consiste en delegar en un thread hijo la realización de las tareas que puede

llevar mucho tiempo procesar. El thread EDT solo tiene que crear este thread hijo y decirle cuál es el trabajo que debe realizar. Despojando de este modo, el hilo EDT está disponible para continuar actualizando la GUI. Por lo tanto, el usuario está frente a una aplicación fluida.

a. Delegar los tratamientos a un thread hijo

El siguiente ejemplo resalta el problema de los tratamientos largos. Está disponible en el paquete `es.eni.editions.ihm37`. La aplicación de edición de texto ha sido modificada. Tiene un botón de acción adicional representado por un reloj de arena que provoca un procesamiento largo al hacer clic. Este largo procesamiento es simplemente la realización de 200000 concatenaciones de la letra «s». Este es el aspecto de la aplicación cuando se hace clic en el botón:



Observe el nuevo botón representado por un reloj de arena. El usuario acaba de hacer clic en él y el botón permanece presionado durante el tratamiento. Durante este tiempo, la aplicación ya no reacciona a las acciones del usuario. Si continúa introduciendo texto o hace clic en un elemento de la GUI, no hay reacción visible. Cuando finaliza el procesamiento, todas las operaciones realizadas a ciegas se suceden muy rápidamente.

Este es el código que se corresponde con el tratamiento largo desencadenado con el clic del botón:

```
public void lanzarTratamientoLong(ActionEvent actionEvent)
{
```

```
String cadena = "s";
for(int i=0;i<200000;i++)
{
    cadena+="s";
}
}
```

Simplemente ejecute este código en un thread hijo para mantener la aplicación funcionando de manera fluida. La modificación necesaria es la siguiente:

```
public void lanzarTratamientoLargo (ActionEvent actionEvent) {
    Thread threadHijo = new Thread(() -> {
        String cadena = "s";
        for (int i = 0; i < 200000; i++) {
            cadena += "s";
        }
    });
    threadHijo.start();
}
```

La creación de un thread consiste en crear un objeto de tipo `Thread`. El constructor utilizado en el ejemplo espera como argumento un objeto de tipo `Runnable`. Este tipo es una interfaz funcional que ofrece un único método abstracto `void run()`. Este método debe contener el código que se ejecutará en el thread hijo. Para iniciar la ejecución del thread hijo, es necesario llamar al método `start()` del thread.

b. Actualizar el GUI desde un thread hijo

Cuando un thread hijo completa la tarea que se le asigna, a menudo es útil notificarlo al usuario. Existe una gran tentación de modificar la GUI en el thread hijo manipulando sus componentes gráficos. Sin embargo, esto obvia el hecho de que los componentes solo deben ser modificados por el thread EDT, bajo pena de provocar una posible excepción. Por lo tanto, es necesario pedirle al thread EDT del thread hijo que actualice la GUI. Para hacer esto, es posible usar el método `invokeLater` de la clase `SwingUtilities`.

El siguiente ejemplo está disponible en el paquete `es.eni.editions.ihm38`.

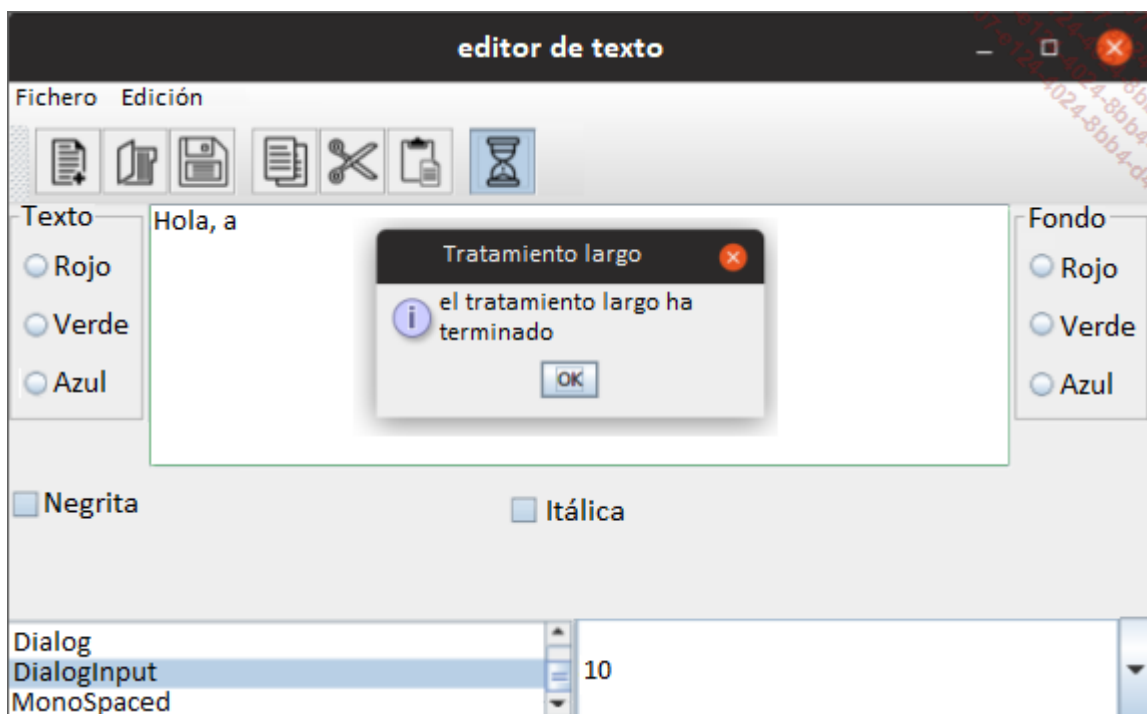
```

private void lanzarTratamientoLargo(ActionEvent actionEvent) {
    Thread threadHijo = new Thread(() -> {
        String cadena = "s";
        for (int i = 0; i < 200000; i++) {
            cadena += "s";
        }
        SwingUtilities.invokeLater(()->notificarFinTratamientoLargo
    ));
    });
    threadHijo.start();
}

private void notificarFinTratamientoLargo ()
{
    JOptionPane.showMessageDialog(null, "el tratamiento largo ha
terminado",
        "Tratamiento largo", JOptionPane.INFORMATION_MESSAGE);
}

```

Cuando termina el tratamiento, la visualización es la siguiente:



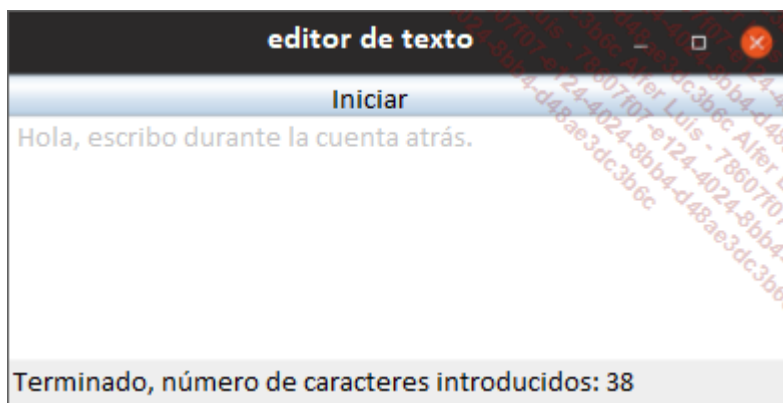
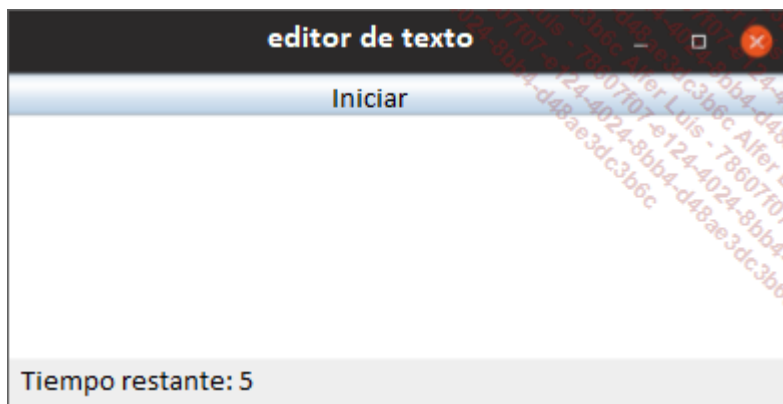
Esta solución funciona. Pero puede volverse rápidamente complejo el manejo de threads. La API Swing proporciona una clase llamada `SwingWorker` que simplifica la gestión del procesamiento largo al ocultar la manipulación de threads. Este es el objetivo de la

siguiente sección.

c. Utilizar la clase `SwingWorker`

La clase `SwingWorker` permite manejar los threads hijos de forma transparente y, por lo tanto, facilitar la gestión de los procesamientos largos al evitar operaciones demasiado complejas.

El resto de la sección se basa en un ejemplo correspondiente a la configuración de una cuenta regresiva. Durante la cuenta regresiva, el usuario podrá escribir texto en un área de entrada, mientras puede consultar el tiempo restante. Por lo tanto, la entrada y la cuenta regresiva son dos procesos que se deben poder llevar a cabo en paralelo. Cuando se completa la cuenta regresiva, el área de entrada se desactiva y se muestra el número de caracteres ingresados.



La cuenta hacia atrás es una operación larga, por lo que la utilización de un `SwingWorker` se adapta muy bien.

Versión 1

El código del ejemplo siguiente está disponible en el paquete `es.eni.editions.ihm39`.

La clase `SwingWorker` es una clase abstracta. Para explotarla, es suficiente con crear una clase que herede de ella. En el ejemplo, la clase `CuentaRegresivaSwingWorker` se crea de la siguiente manera:

```
package es.eni.editions.ihm39;

import javax.swing.SwingWorker;

public class CuentaRegresivaSwingWorker extends SwingWorker<String,
Integer> {

    //Método ejecutado en un thread hijo
    @Override
    protected String doInBackground() throws Exception {
        for(int i=10;i>=0;i--)
        {
            System.out.println(i);
            this.setProgress(i);
            Thread.sleep(1000);
        }
        return "terminado";
    }
}
```

Esta clase hereda de la clase `SwingWorker<String, Integer>`. Es una clase genérica que espera dos tipos de parámetros:

- El primero se corresponde con el tipo de resultado final, cuando termina el tratamiento.
- El segundo se corresponde con el tipo de resultado intermedio si es necesario.

En el contexto del ejemplo, el resultado final se corresponde con la cadena de caracteres «terminado» y el resultado intermedio se corresponde con un número que indica el tiempo restante (la cuenta regresiva).

La clase `CuentaRegresivaSwingWorker` implementa el método `doInBackground`. Este método se ejecuta en un thread hijo. No es necesario que el

desarrollador cree el thread él mismo. El método devuelve una cadena de caracteres al final del procesamiento, correspondiente al tipo de parámetro definido cuando se creó la clase. El método se conforma con realizar un procesamiento largo que consiste en un bucle, que emite un número (de 10 a 0) cada segundo a través del método `setProgress`. Este método espera como argumento un valor de tipo `int`, que indica el progreso del tratamiento. En nuestro ejemplo, es posible usar este atributo para indicar el tiempo restante (el tipo de este atributo no se corresponde con el segundo tipo de parámetro en la clase. Este tipo de parámetro solo se usará en la versión 2 del ejemplo presentado un poco más adelante).

Este procesamiento largo se ejecuta al hacer clic en el botón **Inicio** de la aplicación. La suscripción, realizada en la clase `Pantalla`, se efectúa utilizando una referencia de método:

```
//Desencadenar el tratamiento largo con el clic del botón:  
boton.addActionListener(this::iniciarTratamientoLargo);
```

El método `iniciarTratamientoLargo` tiene el siguiente código:

```
private void iniciarTratamientoLargo(ActionEvent e)  
{  
    textArea.setEnabled(true);  
    textArea.setText("");  
    CuentaRegresivaSwingWorker CuentaRegresiva =  
        new CuentaRegresivaSwingWorker();  
    CuentaRegresiva.addPropertyChangeListener(  
        (PropertyChangeEvent pce)->  
        {  
            System.out.println(pce.getPropertyName());  
            if("progress".equals(pce.getPropertyName()))  
            {  
                label.setText("Tiempo restante: " +  
                    CuentaRegresiva.getProgress());  
            }  
            else if("state".equals(pce.getPropertyName()))  
            {  
                if(CuentaRegresiva.getState()==StateValue.DONE)
```

```

        {
            try {
                label.setText(CuentaRegresiva.get() +
                    ", número de caracteres escritos: " +
                    textArea.getText().length());
                textArea.setEnabled(false);
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
});
CuentaRegresiva.execute();
}

```

El método `iniciarTratamientoLargo` crea un objeto de tipo `CuentaRegresivaSwingWorker` antes de iniciar su ejecución a través del método `execute`. Puede observar el método `addChangeListener`. Le permite suscribirse (a través de una expresión lambda en el ejemplo) a los cambios en los valores de los atributos de la clase `CuentaRegresivaSwing Worker`. Este método verifica el nombre de la propiedad cuyo valor ha cambiado. Si el valor es «`progress`», significa que el valor de la propiedad del mismo nombre ha cambiado (llamando al método `setProgress` en el método `doInBackground`).

El texto de la etiqueta variable se modifica para mostrar el nuevo tiempo restante. Si el valor es «`state`», significa que el valor de la propiedad del mismo nombre ha cambiado. Este valor cambia dos veces: cuando `SwingWorker` se ejecuta a través del método de ejecución (`StateValue.STARTED`) y cuando se completa el procesamiento (`StateValue.DONE`). Si el valor es `StateValue.DONE`, entonces es posible obtener el retorno del método `doInBackground` llamando al método `get` del `SwingWorker`.

Versión 2

La sección que muestra el uso de `SwingWorker` es una ligera variación de la primera versión. El código está disponible en el paquete `es.eni.editions .ihm40`.

La gestión de la modificación del GUI ya no se realizará más a través del método `addChangeListener`. El trabajo se delega a la clase `CuentaRegresivaSwingWorker`. El código del método `iniciarTratamientoLargo` de la clase `Pantalla` se simplifica:

```
private void iniciarTratamientoLargo(ActionEvent e)
{
    textArea.setEnabled(true);
    textArea.setText("");
    CuentaRegresivaSwingWorker CuentaRegresiva =
new CuentaRegresivaSwingWorker(label, textArea);
    CuentaRegresiva.execute();
}
```

Los objetos del GUI (`label` y `textArea`), se ofrecen al objeto `CuentaRegresiva` para dejarle hacer las actualizaciones del GUI.

Por lo tanto, el código de la clase `CuentaRegresivaSwingWorker` ha evolucionado y ahora se parece a este:

```
package es.eni.editions.ihm40;

import java.util.List;
import javax.swing.JLabel;
import javax.swing.JTextArea;
import javax.swing.SwingWorker;

public class CuentaRegresivaSwingWorker extends SwingWorker<String,
Integer> {

    private JLabel label;
    private JTextArea textArea;

    public CuentaRegresivaSwingWorker(JLabel label, JTextArea textArea)
    {
        this.label=label;
        this.textArea=textArea;
    }
}
```



```

}

//Método ejecutado en un thread hijo
@Override
protected String doInBackground() throws Exception {
    for(int i=10;i>=0;i--)
    {
        System.out.println(i);
        this.publish(i);
        Thread.sleep(1000);
    }
    return "terminado";
}

//Ejecutado en el thread EDT
@Override
protected void process(List<Integer> chunks) {
    label.setText("Tiempo restante: " +
chunks.get(chunks.size()-1));
}

//Ejecute en el thread EDT
@Override
protected void done() {
    try {
        label.setText(this.get() +
            ", número de caracteres introducidos: " +
            textArea.getText().length());
        textArea.setEnabled(false);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

Puede observar la presencia de dos métodos adicionales (`process` y `done`), ejecutados en el thread EDT. Por lo tanto, pueden actualizar la GUI a través de objetos `label` y

`textArea`, suministrados a través del constructor.

Se llama al método `process` cada vez que se publica información utilizando el método `publish`. Observe la modificación en el método `doInBackground`; la llamada al método `setProgress` (realizada en la versión 1) se sustituye por la llamada al método `publish`. Llamar a este método en el thread hijo desencadena el método `process` en el thread EDT. Por lo tanto, es posible, en el método `process`, actualizar el GUI durante el transcurso del tratamiento prolongado.

El tipo de parámetro del método `publish` se corresponde con el segundo tipo de parámetro de la clase (`Integer` en el ejemplo). Por lo tanto, este método es más adaptable que el método `setProgress`, que necesariamente requiere un número entero. La firma del método `process` es una lista de este segundo tipo de parámetro.

Dependiendo de la ocupación del procesador, se pueden omitir algunas llamadas al método `process`. Sin embargo, los valores se guardan y se envían en la próxima llamada. En el contexto del ejemplo, la actualización de la GUI se realiza con el último valor de la lista.

El método `done` se llama automáticamente después de que finalice la ejecución del método `doInBackground`. El retorno del método `doInBackground` es accesible a través del método `get`.

Conclusión

Este capítulo le permite comprender una cierta cantidad de conceptos y componentes relacionados con la creación de interfaces gráficas con la API Swing. Para ir más allá, comience consultando la documentación oficial disponible en la siguiente dirección:

<https://docs.oracle.com/javase/tutorial/uiswing/>

Principios del funcionamiento de una base de datos

Las bases de datos relacionales se han convertido en elementos ineludibles de la mayoría de las aplicaciones. Sustituyen al uso de archivos gestionados por el propio desarrollador. Esta aportación permite un aumento de productividad importante durante el desarrollo y una mejora significativa de las posibilidades de las aplicaciones. Facilitan también compartir información entre usuarios. Para poder utilizar una base de datos, es necesario conocer un mínimo de vocabulario relacionado con esta tecnología.

1. Terminología

En el contexto de las bases de datos, se utilizan frecuentemente los términos siguientes:

- **Base de datos relacional:** una base de datos relacional es un tipo de base de datos que utiliza tablas para el almacenamiento de datos. Emplea valores procedentes de dos tablas para asociar los datos de una tabla con los datos de otra tabla. En general, en una base de datos relacional, se almacena la información una única vez.
- **Tabla:** una tabla es un componente de una base de datos que almacena los datos en registros (filas) y en campos (columnas). Se suele agrupar la información por categoría a nivel de una tabla. Por ejemplo, tendremos la tabla de los Clientes, de los Productos o de los Pedidos.
- **Registro:** el registro es el conjunto de datos relativos a un elemento de una tabla. Los registros son los equivalentes al nivel lógico de las filas de una tabla. Por ejemplo, un registro de la tabla Clientes contiene las características de un cliente particular.
- **Campo:** un registro se compone de varios campos. Cada campo de un registro contiene una única información en el registro. Por ejemplo, un registro Cliente puede



contener los campos CódigoCliente, Apellido, Nombre...

- **Clave primaria:** se utiliza una clave primaria para identificar unívocamente una fila de una tabla. La clave primaria es un campo o una combinación de campos cuyo valor es único en la tabla. Por ejemplo, el campo CódigoCliente es la clave primaria de la tabla Cliente. No pueden existir dos clientes con el mismo código.
- **Clave foránea:** una clave foránea representa uno o varios campos de una tabla que hacen referencia a los campos de la clave primaria de otra tabla. Las claves foráneas indican cómo se relacionan las tablas entre sí.
- **Relación:** una relación es una asociación establecida entre campos comunes en dos tablas. Una relación puede ser de uno a uno, uno a varios o varios a varios. Gracias a las relaciones, los resultados de peticiones pueden contener datos procedentes de varias tablas. Una relación de uno a varios entre la tabla Cliente y la tabla Pedido permite a una petición devolver todos los pedidos correspondientes a un cliente.

2. El lenguaje SQL

Antes de poder escribir una aplicación de Java que utiliza datos, es conveniente familiarizarse con el lenguaje SQL (*Structured Query Language*). Este lenguaje permite dialogar con la base de datos. Existen diferentes versiones del lenguaje SQL según la base de datos utilizada. Sin embargo, SQL dispone también de una sintaxis elemental normalizada independiente de cualquier base de datos.

a. Búsqueda de información

El lenguaje SQL permite especificar los registros que se han de extraer, así como el orden en el cual deseamos extraerlos. Se puede crear una instrucción SQL que extrae datos de varias tablas simultáneamente o se puede crear una instrucción que extrae solamente un registro específico. Se utiliza la instrucción **SELECT** para devolver unos campos específicos de una o de varias tablas de la base de datos.

La instrucción siguiente devuelve la lista de los apellidos y nombres de todos los registros de la tabla Cliente.

```
SELECT Apellido, Nombre FROM Cliente
```

Se emplea el símbolo * (asterisco) en lugar de la lista de los campos de los cuales desea el valor. En este caso, se devuelve el conjunto de campos. No es una práctica recomendada en producción.

```
SELECT * FROM Cliente
```

Se puede limitar el número de registros seleccionados utilizando uno o varios campos para filtrar el resultado de la petición. SQL cuenta con varias cláusulas para ejecutar este filtro.

Cláusula WHERE

Esta cláusula permite especificar la lista de las condiciones que deberán cumplir los registros para formar parte de los resultados devueltos. El ejemplo siguiente permite encontrar todos los clientes que viven en Cuenca.

```
SELECT * FROM Cliente WHERE Ciudad='Cuenca'
```

La sintaxis de la cláusula requiere el uso de comillas simples para delimitar las cadenas de caracteres.

Cláusula WHERE ... IN

Podemos utilizar la cláusula WHERE . . . IN para devolver todos los registros que responden a una lista de criterios. Por ejemplo, puede buscar todos los clientes que vivan en Francia o España.

```
SELECT * FROM Cliente WHERE País IN ('Francia','España')
```

Cláusula WHERE ... BETWEEN

También puede devolver una selección de los registros que se sitúan entre dos criterios especificados. La petición siguiente permite recuperar la lista de los pedidos pasados en el mes de noviembre de 2018.

```
SELECT * from Pedidos WHERE FechaPedido BETWEEN '01/11/18' AND  
'30/11/18'
```

Se recomienda el uso del formato internacional yyyy-MM-DDTHH:mm:ss[.sss] para garantizar un funcionamiento adecuado, independientemente del servidor instalado y del

idioma utilizado.

Por lo tanto, el ejemplo anterior se puede escribir de la siguiente manera:

```
SELECT * from Pedidos WHERE FechaPedido BETWEEN '2018-11-01' AND  
'2018-11-30'
```

Cláusula WHERE ... LIKE

Se puede utilizar la cláusula WHERE . . . LIKE para devolver todos los registros para los cuales existe una condición particular en un campo dado. Por ejemplo, la sintaxis siguiente selecciona todos los clientes cuyo nombre empieza por la letra d minúscula:

```
SELECT * FROM Cliente WHERE Apellido LIKE 'd%'
```

En esta instrucción, el símbolo % se utiliza para remplazar una secuencia de caracteres cualquiera.

cláusula ORDER BY ...

Puede utilizar la cláusula ORDER BY para devolver los registros en un orden en particular. La opción ASC especifica orden ascendente; la opción DESC, descendente. Se pueden especificar varios campos como criterio de clasificación. Se analizan de izquierda a derecha. En caso de igualdad en el valor de un campo, se utiliza el campo siguiente.

```
SELECT * FROM Cliente ORDER BY Apellido DESC, Nombre ASC
```

Esta instrucción devuelve los clientes clasificados por orden descendente en el apellido y, en caso de igualdad, por orden ascendente en el nombre.

b. Inserción de datos

La creación de los registros en una tabla se hace con el comando INSERT INTO. Se ha de indicar la tabla en la cual deseamos insertar una línea, la lista de los campos en los cuales deseamos especificar un valor y, por último, la lista de los valores correspondientes.

La sintaxis completa es, por lo tanto, la siguiente:

```
INSERT INTO Cliente (códigoCliente,apellido,nombre) VALUES
```

```
(1000, 'García', 'Pedro')
```

En el momento de la inserción de este nuevo cliente, solo se insertará en la tabla el código, el apellido y el nombre. Los demás campos tomarán el valor NULL. Si no se indica la lista de campos, la instrucción INSERT exige que se especifique un valor para cada campo de la tabla. Por lo tanto, es necesario utilizar la palabra clave NULL para indicar que de un campo particular no hay información. Si la tabla Cliente se compone de cinco campos (CódigoCliente,Apellido,Nombre,Dirección,País) se puede escribir la instrucción anterior con la sintaxis siguiente:

```
INSERT INTO Cliente VALUES (1000, 'García', 'Pedro', NULL, NULL)
```

Como puede apreciar en este caso, las dos palabras clave NULL son obligatorias para los campos Dirección y País.

c. Actualización de datos

La modificación de los campos para registros existentes se realiza con la instrucción UPDATE. Esta instrucción puede actualizar varios campos de varios registros de una tabla a partir de las expresiones que se le proporcionan. Se debe facilitar el nombre de la tabla que hay que actualizar, así como el valor que hay que asignar a los diferentes campos. La lista se indica con la palabra clave SET seguida de la asignación del nuevo valor a los campos. Si deseamos que las modificaciones solo afecten a un conjunto limitado de registros, debemos especificar la cláusula WHERE con el fin de limitar el alcance de la actualización. Si no se indica cláusula WHERE alguna, la modificación se hará en el conjunto de los registros de la tabla.

Por ejemplo, para modificar la dirección de un cliente particular, se emplea la instrucción siguiente:

```
UPDATE Cliente SET Dirección='Calle de Madrid, 4 16000 Cuenca' WHERE  
CódigoCliente=1000
```

Si la modificación ha de abarcar el conjunto completo de registros de la tabla, la cláusula WHERE es inútil. Por ejemplo, si queremos aumentar el precio unitario de todos los artículos, podemos usar la instrucción siguiente:

`UPDATE Catalogo SET PrecioUnitario=PrecioUnitario*1.1`

d. Supresión de datos

La instrucción `DELETE FROM` permite suprimir uno o varios registros de una tabla. Debemos proporcionar como mínimo el nombre de la tabla en la cual se hará la supresión. Si no se indica nada más, en este caso se suprimirán todas las filas de la tabla. En general, se añade una cláusula `WHERE` para limitar el alcance de la supresión. El comando siguiente borra todos los registros de la tabla `Cliente`:

```
DELETE FROM Cliente
```

El comando siguiente es menos radical, dado que solo suprime un registro particular:

```
DELETE FROM Cliente WHERE CódigoCliente=1000
```

Por supuesto, el lenguaje SQL es mucho más completo y no se limita a estas cinco instrucciones. Sin embargo, son suficientes para la manipulación de datos desde Java. Si desea profundizar en el aprendizaje del lenguaje SQL, le aconsejo consultar cualquiera de los libros disponibles en esta misma colección que abordan el tema de manera más extensa.

Acceso a una base de datos desde Java

Cuando se desea manipular una base de datos a partir de un lenguaje de programación, se dispone de dos soluciones:

- Comunicarse directamente con la base de datos.
- Utilizar una capa de software que asegure el diálogo con la base de datos.

La primera solución comporta varios requisitos.

- Debe dominar perfectamente la programación de red.
- También debe conocer en detalle el protocolo utilizado por la base de datos.
- Este tipo de desarrollo suele ser muy largo y lleno de trabas. Por ejemplo, ¿puede acceder a las especificaciones del protocolo?
- Deberá empezar de nuevo todo su trabajo si cambia de tipo de base de datos ya que por supuesto los protocolos no son compatibles de una base de datos a otra. O incluso peor, de una versión a otra de una misma base de datos.

Por supuesto, la segunda solución es preferible y es la que los diseñadores de Java han elegido. Para ello, desarrollaron la API JDBC (*Java DataBase Connectivity*) para el acceso a una base de datos. Precisamente, la API JDBC se compone de dos partes. La primera parte, contenida en el paquete `java.sql`, se compone esencialmente de interfaces. Estas interfaces están implementadas por los drivers JDBC. Estos drivers (o puentes) no están desarrollados por Oracle sino, en general, por la empresa diseñadora de la base de datos. Efectivamente, es el propio fabricante quien domina mejor la técnica para comunicarse con su base de datos. Existen cuatro tipos de drivers JDBC con características y rendimiento diferentes.

-
- Tipo 1: Driver JDBC-ODBC

Este tipo de puente no es específico de una base de datos concreta, sino que traduce simplemente las instrucciones jdbc en instrucciones ODBC. A continuación, es el driver odbc quien asegura la comunicación con la base de datos. Esta solución solo ofrece resultados mediocres porque dependen del número de capas de software utilizadas. Las funcionalidades jdbc también están limitadas por las de la capa ODBC. Este tipo de controlador no está disponible a partir de la versión 8 de Java. Oracle recomienda utilizar el controlador específico de la base de datos con la que desee trabajar. Está, generalmente, disponible en el sitio del fabricante de la base de datos.

- Tipo 2: Driver nativo

Este tipo de puente no está escrito completamente en Java. La parte de este driver escrita en Java efectúa sencillamente llamadas hacia funciones del driver nativo. Se efectúan estas llamadas mediante la API JNI (*Java Native Interface*). De la misma forma que para los puentes de tipo 1, hace falta una traducción entre el código Java y la base de datos. Sin embargo, este tipo sigue siendo más eficaz que los drivers JDBC-ODBC.

- Tipo 3: Driver que utiliza un servidor intermediario

Este puente de protocolo de red, desarrollado completamente en Java, no se comunica directamente con la base de datos, sino que se dirige a un servidor intermediario. El diálogo entre el driver y el servidor intermediario es estándar sea cual sea el tipo de base de datos. A continuación, es este servidor intermediario quien transmite las peticiones utilizando un protocolo específico para la base de datos.

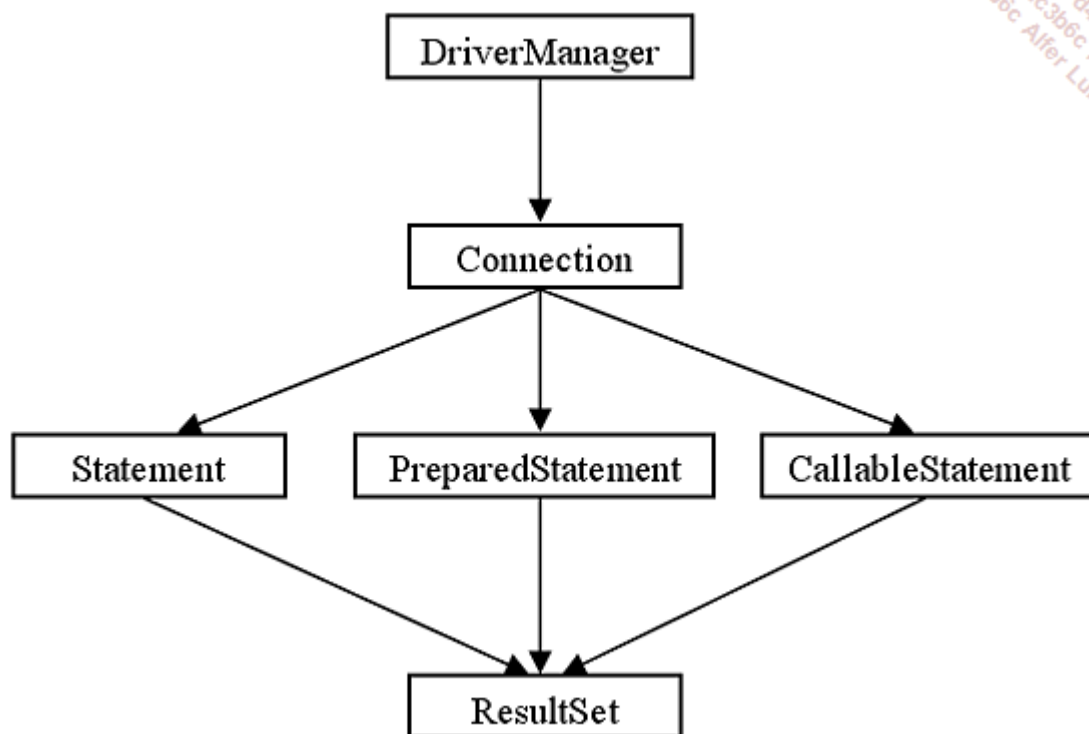
- Tipo 4: Driver completamente escrito en Java

Este tipo de driver representa la solución ideal, ya que no hay intermediario alguno. El driver transmite directamente las peticiones a la base de datos utilizando el protocolo propio de la base de datos. La mayoría de los puentes actuales son de este tipo.

1. Presentación de JDBC

La API JDBC proporciona un conjunto de clases y sobre todo de interfaces que permiten la manipulación de una base de datos. Estos elementos representan todo lo necesario para

acceder a los datos desde una aplicación de Java. El esquema siguiente retoma el camino lógico desde el driver hasta los datos.



La clase `DriverManager` es nuestro punto de partida. Es ella quien asegura el vínculo con el driver. A través de ella, podemos obtener una conexión hacia la base de datos. Está representada por una instancia de clase que implementa la interfaz `Connection`. A continuación, se utiliza esta conexión para transmitir instrucciones a la base de datos. Las peticiones simples se ejecutan mediante la interfaz `Statement`; las peticiones con parámetros, con la interfaz `PreparedStatement`, y los procedimientos almacenados, con la interfaz `CallableStatement`.

Los eventuales registros seleccionados por la instrucción SQL se recogen y devuelven en forma de un elemento `ResultSet`. Vamos a detallar estas diferentes etapas en los secciones siguientes.

2. Carga del driver

La primera etapa indispensable es obtener el driver JDBC adaptado a su base de datos. En general, este controlador está disponible para su descarga en el sitio del diseñador de la base de datos. Para nuestros ejemplos, utilizaremos el controlador facilitado por Oracle para acceder a un servidor de base de datos MySQL. Es posible descargarlo en la dirección

URL siguiente: <https://dev.mysql.com/downloads/connector/j/>

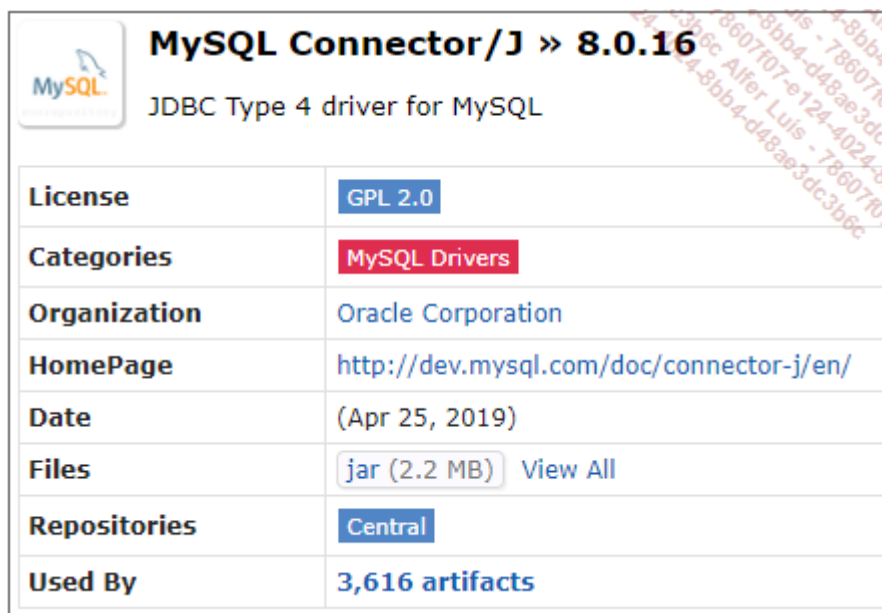
Se necesita la versión 8.0 del driver (llamado *Connector/J*) para utilizar MySQL 5.7.

MySQL Connector/J 8.0 is highly recommended for use with MySQL Server 8.0, 5.7, 5.6, and 5.5. Please upgrade to MySQL Connector/J 8.0.

Es suficiente con realizar la descarga adaptada en función de su sistema operativo.

También es posible descargar de manera más sencilla este driver en el store Maven, en la siguiente dirección: <https://mvnrepository.com/artifact/mysql/mysql-connector-java/8.0.16>

Se muestra la siguiente pantalla. Permite la descarga directa del driver, pulsando en **jar (2.2MB)** o usando un administrador de dependencias, como Maven o Gradle.

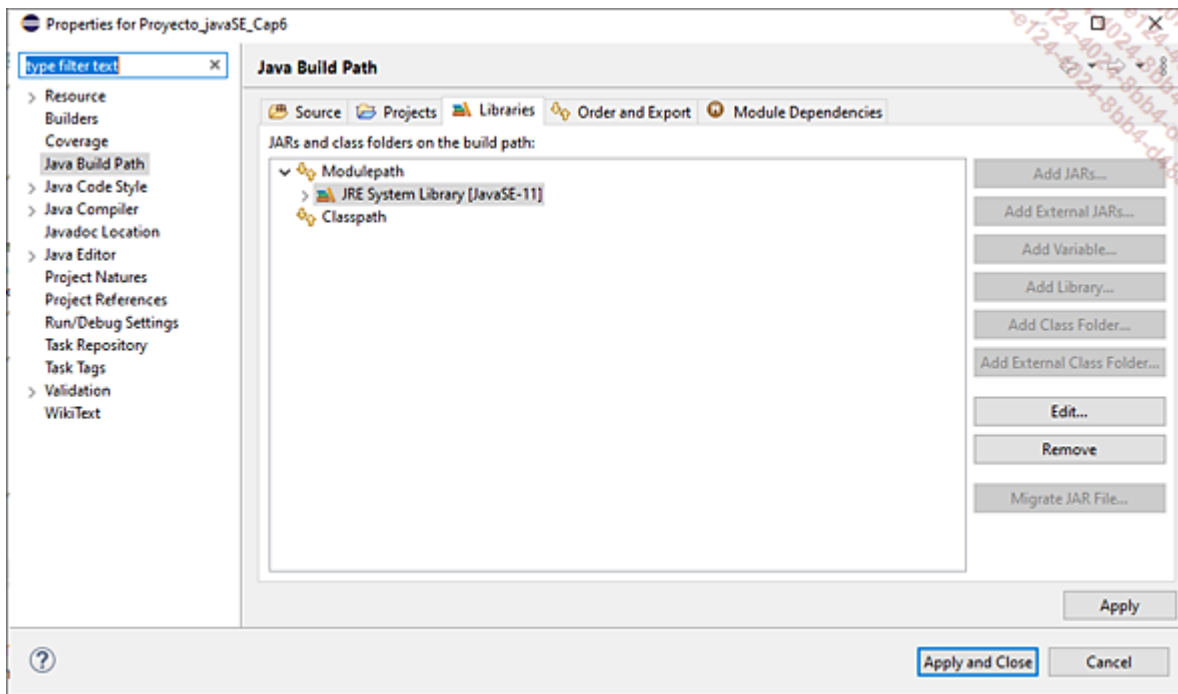


License	GPL 2.0
Categories	MySQL Drivers
Organization	Oracle Corporation
HomePage	http://dev.mysql.com/doc/connector-j/en/
Date	(Apr 25, 2019)
Files	jar (2.2 MB) View All
Repositories	Central
Used By	3,616 artifacts

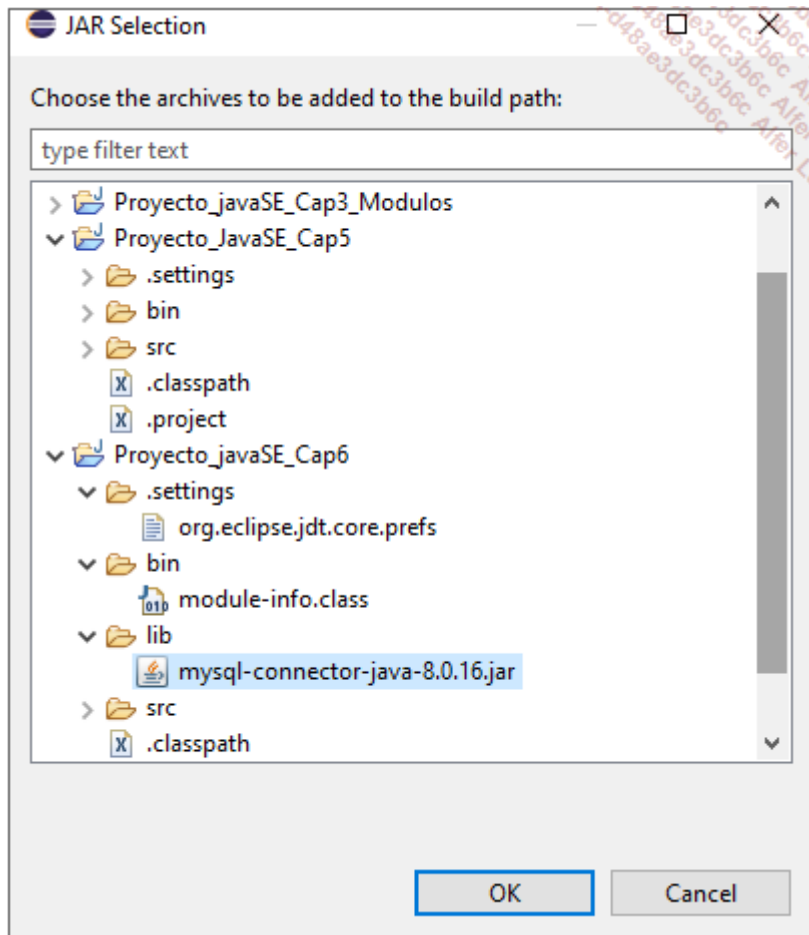
El archivo jar obtenido (**mysql-connector-java-8.0.16.jar**) contiene las clases desarrolladas por Oracle que implementan las diferentes interfaces JDBC. Por supuesto, este archivo deberá estar accesible en el momento de la compilación y de la ejecución de la aplicación.

Para ello, siga estas etapas:

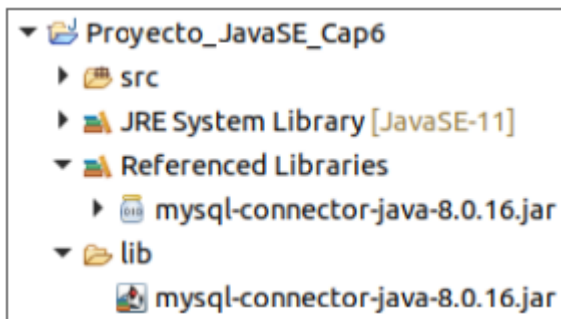
- ❑ Copiar el fichero **mysql-connector-java-8.0.16.jar** en un directorio llamado **lib** de su proyecto.
- ❑ Haga clic con el botón derecho del ratón en su proyecto y seleccione el menú **Build Path - Configure Build Path....**
- ❑ Vaya a la pestaña **Libraries** para ver la siguiente pantalla:



- Pulse el botón **Add JARs....** Aparece la siguiente pantalla:



- Seleccione el archivo **mysql-connector-java-8.0.16.jar** en su proyecto y pulse el botón **OK**. La estructura de su proyecto se debe parecer a la siguiente:



El directorio **lib** contiene físicamente el controlador y este se referencia en el proyecto (**Referenced Libraries**). El driver se utiliza para el desarrollo y estará disponible en el classpath durante la ejecución.

El driver se carga automáticamente durante la ejecución del proyecto desde la versión 4.0 de la API JDBC (disponible desde Java SE 6).

Anteriormente, el piloto tenía que cargarse con el método `forName` de la clase `Class`. Este método recibe como parámetro una cadena de caracteres que contiene el nombre del driver. Llegados a este punto, es indispensable consultar la documentación del driver para obtener el nombre de la clase. En nuestro caso, esta clase tiene el nombre siguiente:

```
com.mysql.cj.jdbc.Driver
```

El registro del driver se crea, por lo tanto, con la instrucción siguiente:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Las cadenas de caracteres que se pasan como parámetros representan nombres de clases; por lo tanto, son sensibles a mayúsculas y minúsculas. De hecho, se debe proteger la instrucción `forName` con un bloque `try catch`, ya que es susceptible de producir una excepción de tipo `ClassNotFoundException`.

3. Establecer y manipular la conexión

Una vez descargado e instanciado, el driver es capaz de facilitar una conexión hacia el servidor de base de datos.

a. Establecer la conexión

El método `getConnection` de la clase `DriverManager` se encarga de esta tarea y propone tres soluciones para establecer la conexión. Cada una de las versiones de este método recibe como parámetro una cadena de caracteres que representa una URL que contiene los datos necesarios para establecer la conexión. El contenido de esta cadena de caracteres es específico para cada driver y hay que consultar de nuevo la documentación para obtener la sintaxis adecuada. Sin embargo, el principio de esta cadena de caracteres es estándar y presenta la forma siguiente `jdbc:nombreDelProtocolo`. El nombre del protocolo es propio para cada driver y gracias a este nombre el método `getConnection` es capaz identificar el driver correcto. El resto de la cadena es específica de cada driver. En general, contiene los datos que permiten identificar el servidor y la base de datos en este servidor, hacia la cual debe establecerse la conexión. Para el driver de MySQL, la sintaxis básica es la siguiente:

```
jdbc:mysql://direccionIP/nombreDeLaBaseDeDatos?user=nombreUsuario&
password=contraseñaUsuario;
```

En caso de éxito, el método `getConnection` devuelve una instancia de una clase que implementa la interfaz `Connection`. En algunos casos, es conveniente emplear la segunda versión del método `getConnection`, ya que permite indicar el nombre y la contraseña utilizados para establecer la conexión como parámetros separados de la URL de conexión.

He aquí un ejemplo que permite establecer una conexión con una base de datos. Este ejemplo supone que tiene un servidor MySQL instalado en la máquina de desarrollo con una base de datos llamada **demo_java** y una cuenta de usuario **mysqluser** identificada por la contraseña **Pa\$\$w0rd**.

```
package es.eni.editions.jdbc.connection;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestConnection {

    public static void main(String[] args) {
        String url="jdbc:mysql://localhost/demo_java";
```



```

        String user="mysqluser";
        String password="Pa$$w0rd";
        try {
            Connection cnx = DriverManager.getConnection(url,
user, password);
            System.out.println("Estado de la conexión:");

            System.out.println(cnx.isClosed()?"cerrado":"abierto");
        } catch (SQLException e) {
            System.out.println("Un error se ha producido
durante la
conexión a la base de datos");
            e.printStackTrace();
        }
    }
}

```

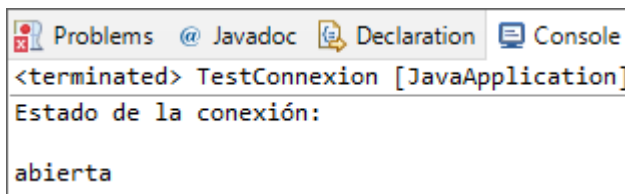
Para que este código se compile, es necesario cargar el módulo `java.sql` porque no se carga de manera predeterminada. Para hacer esto, simplemente modifique el archivo `module-info.java` de este modo:

```

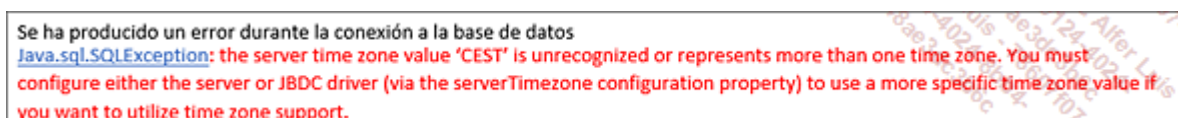
module ProjetJavaSE_Capitulo6 {
    requires java.sql;
}

```

La ejecución de este programa muestra el siguiente mensaje:



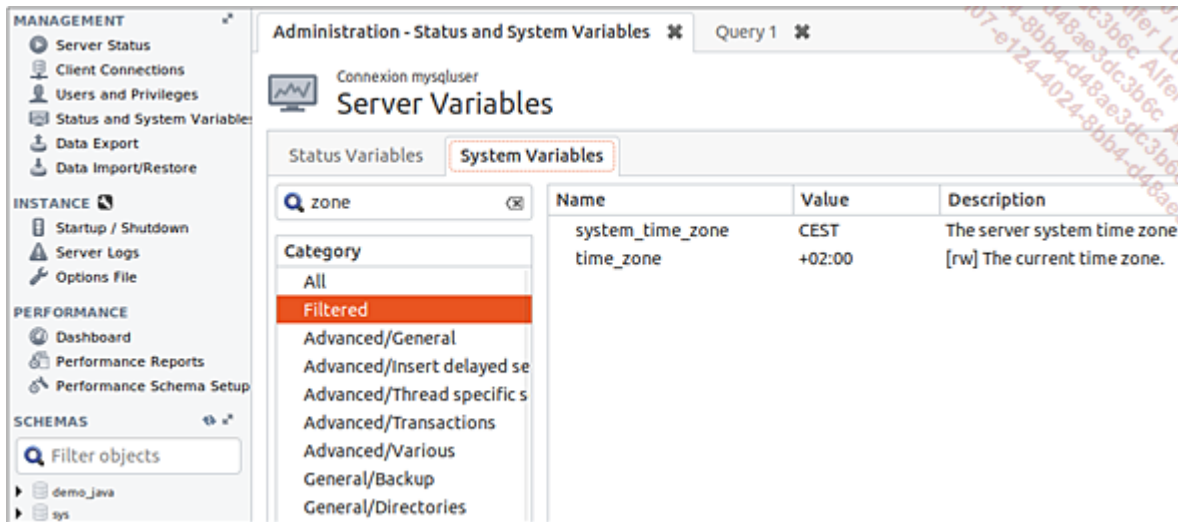
Dependiendo de la configuración de su máquina, puede recibir el siguiente mensaje de error:



Esto significa que hay un problema para determinar qué zona horaria usar. Es posible corregir el problema agregando un parámetro a la cadena de conexión:

```
jdbc:mysql://localhost/demo_java?serverTimezone=UTC
```

La otra posibilidad es cambiar la configuración del servidor MySQL. Para hacer esto, vaya a la pantalla **Server Variables** del servidor desde MySQL Workbench haciendo clic en el menú **MANAGEMENT - Status and System Variables**. Encuentre la propiedad **time_zone** y ajústela con el valor deseado (+02: 00 en el ejemplo):



b. Manipular la conexión

Una conexión queda abierta desde el momento en que se crea. Por lo tanto, no existe un método específico que permita abrir una conexión. En cambio, se puede cerrar una conexión llamando al método `close`. Es importante hacer esto cuando la conexión, ya no sea útil para liberar los recursos comprometidos. Después del cierre de una conexión, ya no es posible usarla y debe volver a crearse para poder utilizarla de nuevo.

La interfaz `Connection` era la interfaz `AutoCloseable`. Por lo tanto, no es necesario usar el método `close` si el desarrollador hace uso de la instrucción `try with resources`. El siguiente ejemplo muestra el uso de una instrucción como esta:

```
package es.eni.editions.jdbc.connection;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```

public class TestConnectionTryWithResources {

    public static void main(String[] args) {
        String url = "jdbc:mysql:
//localhost/demo_java?serverTimezone=UTC";
        String user = "mysqluser";
        String password = "Pa$$w0rd";
        //Utilización de la instrucción try with resources
        try(Connection cnx = DriverManager.getConnection(url,
user, password)) {
            System.out.println("Estado de la conexión:");
            System.out.println(cnx.isClosed() ? "cerrada" :
"abierta");
        } //cierre automático de la conexión
        catch (SQLException e) {
            System.out.println("Un error se ha producido
durante
la conexión a la base de datos");
            e.printStackTrace();
        }
    }
}

```

La función `isClosed` permite comprobar si una conexión está cerrada. Si esta función devuelve un booleano igual a `false`, se puede pensar con razón que la conexión está abierta y que permite dialogar, por lo tanto, con la base de datos. En realidad, esto no es siempre así. La conexión puede estar en un estado intermedio: no está cerrada, pero no puede utilizarse para transferir instrucciones hacia el servidor. Para comprobar la disponibilidad de la conexión, puede utilizar el método `isValid`. Este método comprueba realmente la disponibilidad de la conexión intentando enviar una instrucción SQL y verificando que obtiene una respuesta de parte del servidor. Este método no está implementado en todos los drivers; en ese caso, produce una excepción del tipo `java.lang.UnsupportedOperationException` o un error del tipo `java.lang.AbstractMethodError`. Este es el código que muestra el uso de este método:

```

public static void pruebaEsValida(Connection cnx) {

```

```

        boolean esValida;
        try {
            esValida = cnx.isValid(0);
            if (esValida) {
                System.out.println("La conexión es
válida");
            } else {
                System.out.println("La conexión no es
válida");
            }
        } catch (Exception e) {
            System.out.println("el método isValid no se soporta
por
el controlador");
            e.printStackTrace();
        }
    }
}

```

Si únicamente se efectúan operaciones de lectura en la base de datos, podemos optimizar la comunicación precisando que la conexión está en solo lectura. El método `setReadOnly` permite modificar esta configuración de la conexión. A continuación, se puede comprobar el estado usando el método `isReadOnly`. Para algunos drivers, esta funcionalidad no está implementada y el método `setReadOnly` no surte ningún efecto. La función siguiente verifica si esta funcionalidad está disponible para la conexión que recibe como parámetro.

```

public static void testSoloLectura(Connection cnx)
{
    boolean estado;
    try
    {
        estado = cnx.isReadOnly();
        cnx.setReadOnly(!estado);
        if (cnx.isReadOnly() != estado)
        {
            System.out.println("este driver se encarga
del modo solo lectura");
        }
    }
}

```

```

    }
    else
    {
        System.out.println("este driver no se encarga
del modo solo lectura");
    }
    cnx.setReadOnly(estado);
}
catch (SQLException e)
{
    e.printStackTrace();
}
}

```

Durante la ejecución de una instrucción SQL, el servidor puede detectar problemas y, por lo tanto, generar advertencias. Es posible recuperar estas advertencias mediante el método `getWarnings` de la conexión. Este método devuelve un objeto `SQLWarning` representativo del problema encontrado por el servidor. Si el servidor encuentra varios problemas, genera varios objetos `SQLWarning` encadenados entre sí. El método `getNextWarning` permite obtener el elemento siguiente o `null` si se ha alcanzado el final de la lista. Se puede vaciar la lista con el método `clearWarnings`. La función siguiente visualiza todos los avisos recibidos por la conexión.

```

public static void visualizaciónWarnings(Connection cnx)
{
    SQLWarning aviso;
    try
    {
        aviso=cnx.getWarnings();
        if (aviso==null)
        {
            System.out.println("no hay avisos");
        }
        else
        {
            while (aviso!=null)
            {

```

```

        System.out.println(avisos.getMessage());
        System.out.println(avisos.getSQLState());
        System.out.println(avisos.getErrorCode());
        avisos=avisos.getNextWarning();
    }
}
cnx.clearWarnings();
}
catch (SQLException e)
{
    e.printStackTrace();
}
}

```

En general, en el momento de la creación de la cadena de conexión, uno de sus parámetros determina el nombre de la base de datos con la que queremos establecer una conexión. La modificación del nombre de la base de datos a la que estamos conectados se lleva a cabo con el método `setCatalog`, al cual hay que proporcionar el nombre de otra base de datos presente en el mismo servidor. Por supuesto, hace falta que la cuenta con la que se abrió la sesión disponga de permisos de acceso suficientes para esta base de datos. Cabe señalar que con este método cambiamos de base de datos, pero la conexión se sigue refiriendo al mismo servidor. No hay ningún medio para cambiar de servidor sin crear una nueva conexión.

El código siguiente:

```

public static void cambiarDeBaseDeDatos (Connection cnx) {
    try {
        System.out.println("base actual: " + cnx.getCatalog());
        System.out.println("carga de la base de datos");
        cnx.setCatalog("demo_java_2");
        System.out.println("base actual: " + cnx.getCatalog());
    } catch (SQLException e) {
        System.out.println("Problema en la manipulación
de las bases de datos en el servidor");
        e.printStackTrace();
    }
}
}

```

muestra el siguiente resultado:

```
base actual: demo_java  
cambio de base de datos  
base actual: demo_java_2
```

Por supuesto, en primer lugar hay que haber creado una nueva base de datos llamada demo_java_2.

Es posible también obtener la estructura de la base de datos mediante el método getMetaData. Este método devuelve un objeto de tipo DataBaseMetaData que facilita numerosa información relativa a la estructura de la base de datos. La función siguiente muestra un análisis rápido de esta información.

```
public static void infosBase(Connection cnx)
{
    ResultSet rs;
    DatabaseMetaData dbmd;
    try
    {
        dbmd=cnx.getMetaData();
        System.out.println("tipo de base: " +
dbmd.getDatabaseProductName());
        System.out.println("versión: " +
dbmd.getDatabaseProductVersion());
        System.out.println("nombre del driver: " +
dbmd.getDriverName());
        System.out.println("versión del driver: " +
dbmd.getDriverVersion());
        System.out.println("nombre del usuario: " +
dbmd.getUserName());
        System.out.println("url de conexión: " + dbmd.getURL());
        rs=dbmd.getTablas(null,null,"%",null);
        System.out.println("estructura de la base");
        System.out.println("base\ttesquema\tnombre tabla\t
tipo tabla");
        while(rs.next())
```

```

{
    for (int i = 1; i <=4 ; i++)
    {
        System.out.print(rs.getString(i)+"\t");
    }
    System.out.println();
}
rs.close();
rs=dbmd.getProcedures(null,null,"%");
System.out.println("los procedimientos almacenados");
System.out.println("base\tesquema\tnombre procedimiento");
while(rs.next())
{
    for (int i = 1; i <=3 ; i++)
    {
        System.out.print(rs.getString(i)+"\t");
    }
    System.out.println();
}
rs.close();
}
catch (SQLException e)
{
    e.printStackTrace();
}
}

```

Puede observar en este ejemplo el uso de un objeto de tipo `ResultSet`. Le permite leer los conjuntos de resultados obtenidos en el servidor. Más adelante en este capítulo dedicamos una sección a este tema..

Todos los métodos presentados hasta ahora pueden ser de ayuda algún día, pero la meta principal de una conexión es permitir la ejecución de instrucciones SQL. Son los objetos de tipo `Connection` los que van a facilitar los objetos necesarios para la ejecución de estas instrucciones. Se pueden ejecutar tres tipos de instrucciones SQL:

- Las peticiones simples.
- Las peticiones precompiladas.

- Los procedimientos almacenados.

A cada uno de estos tipos le corresponde una clase de la API JDBC:

- `Statement` para las peticiones simples.
- `PreparedStatement` para las peticiones precompiladas.
- `CallableStatement` para los procedimientos almacenados.

Para terminar, se puede obtener una instancia de estas clases a partir de un método diferente de un objeto de tipo `Connection`:

- `createStatement` para los objetos de tipo `Statement`.
- `prepareStatement` para los objetos de tipo `PreparedStatement`.
- `prepareCall` para los objetos de tipo `CallableStatement`.

En la siguiente sección se aborda en detalle el uso de estos métodos.

4. Ejecución de instrucciones SQL

Antes de la ejecución de una instrucción SQL, debemos elegir el tipo de objeto JDBC más apropiado. Las secciones siguientes describen los tres tipos de objetos disponibles y su utilización.

a. Ejecución de instrucciones básicas con el objeto de tipo `Statement`

Un objeto de este tipo se obtiene mediante el método `createStatement` de la conexión. JDBC cuenta con dos versiones de este método; la primera no recibe ningún parámetro. En este caso, si se utiliza el objeto `Statement` para ejecutar una instrucción SQL que genera un juego de registros (`select`), este estará en solo lectura y con un desplazamiento de los datos hacia delante únicamente.

Los datos presentes en este juego de registros no podrán modificarse y solo se podrá hacer el recorrido del juego de registros desde el primero hacia el último. La segunda versión permite elegir las características del juego de registros generado. Recibe dos parámetros. El primero determina el tipo del juego de registros.

Se definen las constantes siguientes de la clase `ResultSet`:

- `ResultSet.TYPE_FORWARD_ONLY`: el juego de registros será de desplazamiento hacia delante únicamente.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`: el juego de registros podrá recorrerse en ambos sentidos, pero será insensible a los cambios realizados en la base de datos por otros usuarios.
- `ResultSet.TYPE_SCROLL_SENSITIVE`: el juego de registros podrá recorrerse en ambos sentidos y será sensible a los cambios realizados en la base de datos por otros usuarios.

El segundo argumento determina las posibilidades de modificación de los datos contenidos en el juego de los registros. Las dos constantes siguientes están definidas:

- `ResultSet.CONCUR_READ_ONLY`: los registros son de solo lectura.
- `ResultSet.CONCUR_UPDATABLE`: los registros pueden modificarse dentro del juego de registros.

Este objeto es el más elemental que permite la ejecución de instrucciones SQL. Puede encargarse de la ejecución de cualquier instrucción SQL. Por lo tanto, puede ejecutar tanto instrucciones DDL (*Data Definition Language*) como instrucciones DML (*Data Manipulation Language*). Solo hace falta elegir en este objeto el método mejor adaptado para la ejecución del código SQL. Debería seleccionar un método u otro en función del tipo de resultado que devuelve la instrucción SQL. Hay cuatro métodos disponibles:

- `public boolean execute(String sql)`: este método permite ejecutar cualquier instrucción SQL. El `boolean` devuelto por este método indica si se ha generado algún juego de registros (`true`) o si simplemente la instrucción modificó registros en la base de datos (`false`). Si se genera un juego de registros, puede obtenerse mediante el método `getResultSet`. Si ha habido registros modificados en la base de datos, el método `getUpdateCount` devuelve el número de registros modificados. La función siguiente ilustra el uso de estos métodos.

```
package es.eni.editions.jdbc.consulta;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestStatement {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            Statement stm =
cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY)) {

            BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
            System.out.println("introducir su
instrucción SQL:");
            String petición = br.readLine();
            boolean resultado = stm.ejecuta(petición);
            //Si es verdadero, ha habido un resultado
            if (resultado) {
                System.out.println("su instrucción ha
generado
un juego de registros");

                //Recuperación del juego de resultados
                ResultSet rs = stm.getResultSet();
                //Posicionar en el último registro
                rs.last();
                //Lectura del número de línea
                System.out.println("contiene " +
rs.getRow() + " registros");
                //si es falso, puede que haya una actualización
            } else {
                System.out.println("su instrucción
ha modificado registros en la base");
                System.out.println("número de registros
modificados:" + stm.getUpdateCount());
            }
        }
    }
}
```

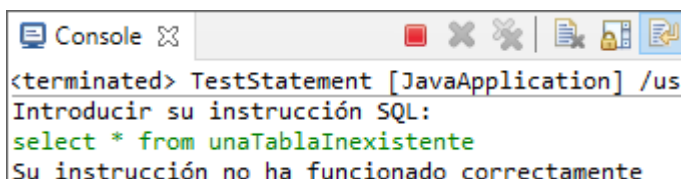
```

    } catch (SQLException e) {
        System.out.println("su instrucción no ha funcionado
correctamente");
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static Connection getConnection() throws SQLException{
    String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

Ejecutar este código con una consulta con errores produce el siguiente resultado:

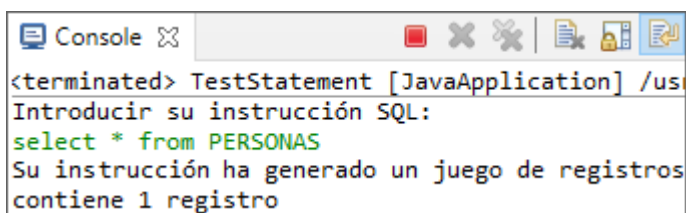


```

<terminated> TestStatement [JavaApplication] /us
Introducir su instrucción SQL:
select * from unaTablaInexistente
Su instrucción no ha funcionado correctamente

```

La ejecución de este código con una consulta de tipo `select` provoca el siguiente resultado:



```

<terminated> TestStatement [JavaApplication] /us
Introducir su instrucción SQL:
select * from PERSONAS
Su instrucción ha generado un juego de registros
contiene 1 registro

```

Para el ejemplo, se ha creado una tabla `PERSONAS` con el siguiente script:

```

use demo_java;
create table PERSONAS(
    id int auto_increment primary key,
    nombre varchar(30) not null,

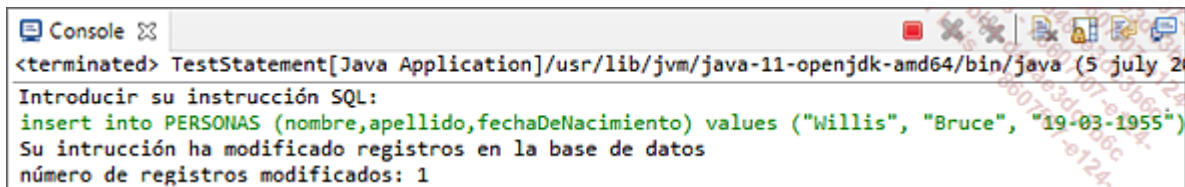
```

```
    apellido varchar(30) not null,  
    fechaDenacimiento date  
);
```

Se ha añadido un registro:

```
insert into PERSONAS(nombre,apellido,fechaDenacimiento)  
values("McQueen", "Steeve", "1930-03-24")
```

Para terminar, la ejecución de este código con una consulta de inserción provoca el resultado siguiente:



```
Console  
<terminated> TestStatement[Java Application]/usr/lib/jvm/java-11-openjdk-amd64/bin/java (5 july 21)  
Introducir su instrucción SQL:  
insert into PERSONAS (nombre,apellido,fechaDeNacimiento) values ("Willis", "Bruce", "19-03-1955")  
Su intrucción ha modificado registros en la base de datos  
número de registros modificados: 1
```

- `public ResultSet executeQuery(String petición):` este método fue diseñado especialmente para la ejecución de instrucciones `select`. El juego de los registros está disponible directamente mediante el valor devuelto por la función.
- `public int executeUpdate(String petición):` este método se adapta perfectamente a la ejecución de instrucciones que modifican el contenido de la base de datos, como son las instrucciones `insert`, `update`, `delete`. El valor entero devuelto por esta función indica el número de registros afectados por la modificación.
- `public int[] executeBatch():` este método permite ejecutar un conjunto de instrucciones SQL por lote. El lote de instrucciones que se va a ejecutar debe prepararse previamente mediante los métodos `addBatch` y `clearBatch`. El primero recibe como parámetro una cadena de caracteres que representa una instrucción SQL que hay que añadir al lote. El segundo permite reinicializar el lote de instrucciones. No es posible suprimir una instrucción particular del lote. Conviene no añadir al lote ninguna instrucción SQL que genere un juego de resultados, ya que en este caso el método `updateBatch` produce una excepción de tipo `BatchUpdateException`. Esta función devuelve una matriz de enteros que permite obtener información sobre la ejecución de cada una de las peticiones del lote.

Cada celda de la matriz contiene un valor entero que representa el resultado de la ejecución de la petición correspondiente en el lote. Un valor superior o igual a 0 indica un funcionamiento correcto de la instrucción y representa el número de registros modificados. Un valor igual a la constante `Statement.EXECUTE_FAILED` indica que la ejecución de la instrucción falló. En este caso, algunos drivers detienen la ejecución del lote, mientras que otros siguen con la instrucción siguiente del lote. Un valor igual a la constante `Statement.SUCCESS_NO_INFO` indica que la instrucción ha sido ejecutada correctamente, pero que no se puede determinar el número de registros modificados. Este método es muy práctico para ejecutar modificaciones en varias tablas relacionadas. Este podría ser, por ejemplo, el caso en una aplicación de gestión comercial con una tabla para las solicitudes y otra para las líneas de detalle de cada solicitud. La supresión de un comando debe causar la supresión de todas las líneas correspondientes. La función siguiente le permite introducir varias instrucciones SQL y ejecutarlas en lote.

```
package es.eni.editions.jdbc.consulta;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestStatementBatch {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            Statement stm = cnx.createStatement
(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY)) {

            BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
            System.out.println("introducir sus instrucciones
```

```

SQL y después run para ejecutar el lote:");
        String peticion = br.readLine();
        while (!peticion.equalsIgnoreCase("run")) {
            stm.addBatch(peticion);
            peticion = br.readLine();
        }
        System.out.println("ejecución del lote
de instrucciones");
        int[] resultados = stm.executeBatch();
        for (int i = 0; i < resultados.length; i++) {
            switch (resultados[i]) {
                case Statement.EXECUTE_FAILED:
                    System.out.println("la ejecución
de la instrucción " + i + " ha fallado");
                    break;
                case Statement.SUCCESS_NO_INFO:
                    System.out.println("la ejecución
de la instrucción " + i + " ha terminado con éxito");
                    System.out.println("el número
de registros modificados es desconocido");
                    break;
                default:
                    System.out.println("la ejecución
de la instrucción " + i + " ha terminado con éxito");
                    System.out.println("ha modificado "
+
resultados[i] + " registro");
                    break;
            }
        }

        } catch (SQLException e) {
            System.out.println("su lote no ha funcionado
correctamente ");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

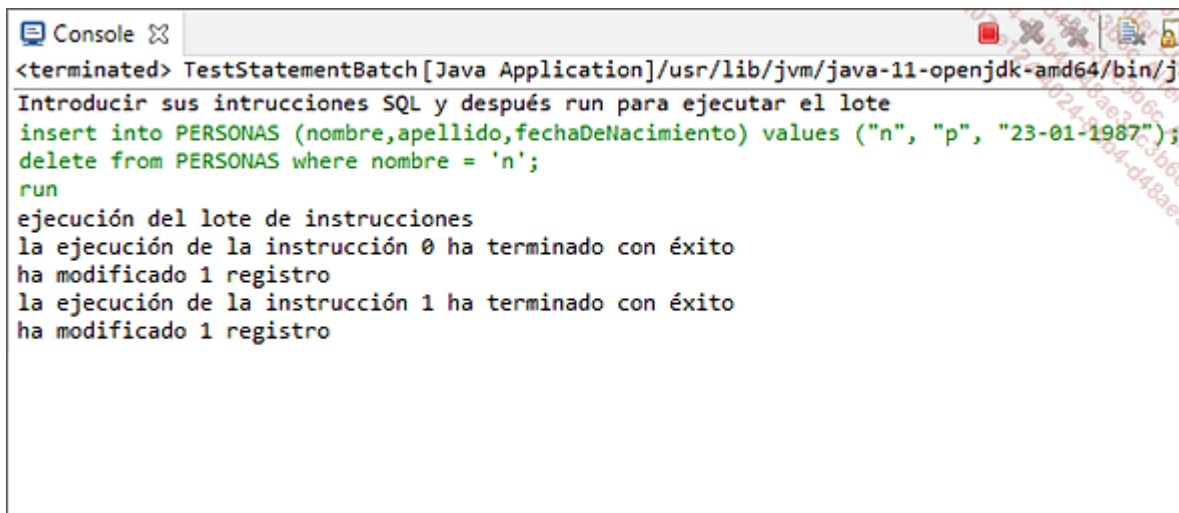
```

```

    private static Connection getConnection() throws SQLException {
        String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
        String user = "mysqluser";
        String password = "Pa$$w0rd";
        return DriverManager.getConnection(url, user, password);
    }
}

```

La ejecución de código con instrucciones SQL válidas da el siguiente resultado:



```

<terminated> TestStatementBatch[Java Application]/usr/lib/jvm/java-11-openjdk-amd64/bin/java
Introducir sus instrucciones SQL y después run para ejecutar el lote
insert into PERSONAS (nombre,apellido,fechaDeNacimiento) values ("n", "p", "23-01-1987");
delete from PERSONAS where nombre = 'n';
run
ejecución del lote de instrucciones
la ejecución de la instrucción 0 ha terminado con éxito
ha modificado 1 registro
la ejecución de la instrucción 1 ha terminado con éxito
ha modificado 1 registro

```

Otros métodos van a intervenir en el comportamiento del objeto `Statement`:

- `public void setQueryTimeout(int duracion)`: este método indica la duración máxima asignada para la ejecución de una instrucción SQL antes de producir una excepción.
- `public void close()`: cuando un objeto `Statement` ya no es útil en una aplicación, es preferible cerrarlo explícitamente invocando este método. Esto provoca la liberación de todos los recursos usados por este objeto. El cierre de un objeto `Statement` provoca también el cierre del juego de registros asociado (`ResultSet`). Al igual que la interfaz `Connection`, la interfaz `Statement` hereda de la interfaz `AutoCloseable`. Por lo tanto, es posible (como muestran los ejemplos) inicializar este tipo de objeto en una instrucción *try with resources*, para provocar su cierre automático al final del bloque `try`.

- `public void setMaxRows(int número):` este método limita el número de líneas de los juegos de registros generados por este objeto `Statement`. Si una instrucción SQL genera un juego de registros que contiene más líneas, las líneas de más simplemente se ignoran (sin más información).
- `public void setMaxFieldSize(int tamaño):` este método limita el tamaño de algunos tipos de campos en el juego de registros. Los tipos de campos afectados son los campos de una base de datos que pueden tener un tamaño variable, como por ejemplo los campos compuestos por cadenas de caracteres o datos binarios. Los datos de más simplemente se ignoran. Los campos afectados pueden ser de hecho inutilizables en la aplicación.
- `public void setFetchSize(int numLíneas):` cuando una instrucción SQL genera un juego de registros, los datos correspondientes se transfieren desde el servidor de base de datos hacia la memoria de la aplicación de Java. Este traslado se efectúa por bloques según el uso de los datos. Este método indica al driver el número de líneas de cada bloque transferido desde la base de datos hacia la aplicación.
- `public boolean getMoreResults():` si se utiliza el método `execute` para ejecutar varias instrucciones SQL, por ejemplo dos instrucciones `select`, en este caso, se generan dos juegos de registros. El primero se obtiene con el método `getResultSet`. El segundo solo será accesible después de haber llamado el método `getMoreResults`. Esta función permite desplazar el puntero sobre el resultado siguiente y devuelve un booleano igual a `true` si el resultado siguiente es un juego de registros. En tal caso, dicho juego de registros también es accesible mediante el método `getResultSet`. Si la función `getMoreResults` devuelve un booleano `false`, es que el resultado siguiente no es un juego de registros o que sencillamente no hay ningún resultado siguiente. Para resolver la duda, hay que invocar la función `getUpdateCount` y comprobar el valor devuelto por esta función. Si este valor es igual a `-1`, es que no hay ningún resultado siguiente; en caso contrario, el valor devuelto representa el número de registros modificados en la base de datos.

La función siguiente permite ejecutar varias instrucciones SQL separadas por puntos y coma:

|

```

package es.eni.editions.jdbc.peticion;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestStatementGetMoreResults {

    public static void main(String[] args) {
        boolean resultado;
        ResultSet rs;
        try (Connection cnx = getConnection();
            Statement stm = cnx.createStatement
(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY)) {

            BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
            System.out.println("introducir sus instrucciones
SQL separadas por; :");
            String peticion = br.readLine();
            resultado = stm.execute(peticion);
            int i = 1;
            // tratamiento del resultado generado por la
primera
            // instrucción
            if (resultado) {
                System.out.println("su instrucción N°
" + i + " ha generado un juego de registros");
                rs = stm.getResultSet();
                rs.last();
                System.out.println("contiene " +
rs.getRow() + " registros");
            } else {

```

```

        System.out.println("su instrucción
Nº " + i + " ha modificado unos registros en la base");
        System.out.println("número de registros
modificados:" + stm.getUpdateCount());
    }
    i++;
    // desplazamiento del puntero sobre un eventual
    // resultado siguiente
    resultado = stm.getMoreResults();
    // bucle mientras haya todavía un resultado de tipo
    // juego de registro -> resultado==true
    // o mientras haya todavía un resultado de tipo
número
    // de registros modificados -> getUpdateCount !=
-1
    while (resultado || stm.getUpdateCount() != -1) {
        if (resultado) {
            System.out.println("su instrucción
Nº " + i + " ha generado un juego de registros");
            rs = stm.getResultSet();
            rs.last();
            System.out.println("contiene " +
rs.getRow() + " registros");
        } else {
            System.out.println("su instrucción
Nº " + i + " ha modificado registros en la base");
            System.out.println("número
de registros modificados:" + stm.getUpdateCount());
        }
        i++;
        // desplazamiento del puntero sobre un
eventual
        // resultado siguiente
        resultado = stm.getMoreResults();
    }
} catch (SQLException e) {
    System.out.println("su instrucción no ha funcionado
correctamente");

```

```

        e.printStackTrace();

    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static Connection getConnection() throws SQLException {
    String url =
"jdbc:mysql://localhost/demo_java?
serverTimezone=UTC&allowMultiQueries=true";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

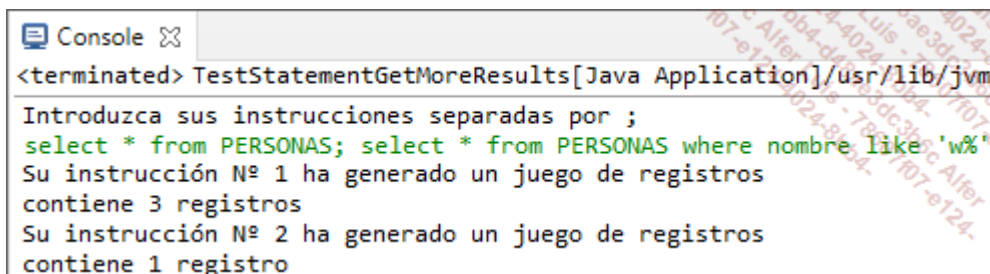
```

Observe la modificación de la cadena de conexión en el método `getConnection` del ejemplo:

```
String url = "jdbc:mysql://localhost/demo_java?serverTimezone=
UTC&allowMultiQueries=true";
```

Para que la ejecución de las consultas múltiples funcione con MySQL, es necesario añadir el argumento `allowMultiQueries` con el valor `true`.

Aquí hay un ejemplo de ejecución de este programa:



The screenshot shows a console window titled "Console" with the following text:

```

<terminated> TestStatementGetMoreResults[Java Application]/usr/lib/jvm/
Introduzca sus instrucciones separadas por ;
select * from PERSONAS; select * from PERSONAS where nombre like 'w%'
Su instrucción Nº 1 ha generado un juego de registros
contiene 3 registros
Su instrucción Nº 2 ha generado un juego de registros
contiene 1 registro

```

b. Ejecución de instrucciones configuradas con el objeto `PreparedStatement`

Ocorre a menudo que hay que ejecutar varias veces una petición SQL con solo una pequeña modificación entre dos ejecuciones. El ejemplo clásico corresponde a una petición de selección con una restricción.

```
String petition = "select * from PERSONAS where nombre = 'McQueen'";
```

El valor sobre el cual recae la restricción suele introducirlo el usuario de la aplicación y en este caso está disponible en una variable. La primera solución que viene a la mente consiste en construir la petición SQL por concatenación de varias cadenas de caracteres.

```
String petition = "select * from PERSONAS where nombre = '" +  
nombreIntroducido + "'";
```

Esta solución presenta varios inconvenientes:

- La concatenación de cadenas de caracteres consume muchos recursos.
- El servidor debe analizar cada vez una nueva petición.
- La sintaxis se va a volver compleja rápidamente si es necesario concatenar varias cadenas. Siempre hay que recordar que al final debemos obtener una instrucción SQL correcta.
- Finalmente, la concatenación puede conducir a riesgos de inyecciones SQL que desvirtuarían la función inicial de la consulta. Imagine la búsqueda para identificar a una persona por su nombre de usuario y contraseña, introducidos en un formulario. Podría ser de la siguiente forma:

```
String petition = "select * from PERSONAS where login = '" +  
loginIntroducido + "' and contrasenia = '" + contraseniaIntroducido  
+ "'";
```

Si el usuario rellena los valores «admin» y «mdp1234», entonces la consulta generada es la siguiente:

```
Select * from PERSONAS where login = 'admin' and contrasenia =  
'mdp1234'
```

En consecuencia, si el usuario introduce los valores «admin» y «' or 'a'='a », entonces la

consulta generada es la siguiente:

```
Select * from PERSONAS where login = 'admin' and contrasenia = '' or 'a'='a' ;
```

La consulta adquiere otro aspecto y el comportamiento cambia. Un atacante puede iniciar sesión sin conocer la contraseña del usuario. Esto se llama inyección SQL. El error de seguridad en este ejemplo es interpretar las comillas introducidas por el usuario como un elemento de sintaxis SQL. Debe escapar este carácter para indicar que se tiene que interpretar solo como un carácter de texto.

Los diseñadores de JDBC tienen prevista una solución eficaz para resolver estos inconvenientes. El objeto `PreparedStatement` aporta una solución eficaz a este problema al permitirnos crear peticiones con parámetros. En este tipo de petición, se sustituyen los parámetros por signos de interrogación. Antes de la ejecución de la petición, hay que facilitar al objeto `PreparedStatement` los valores que debe utilizar para remplazar los diferentes signos de interrogación. Se puede crear un objeto `PreparedStatement` utilizando el mismo principio que para la creación de un objeto `Statement`.

El método `prepareStatement` accesible a partir de una conexión devuelve un objeto `PreparedStatement`. Este método está disponible bajo dos formas. La primera recibe como argumento una cadena de caracteres que representa la petición SQL. En esta petición, la ubicación de los parámetros debe estar reservada por signos de interrogación. Si se crea un juego de registros con la ejecución de esta petición, será de solo lectura y de desplazamiento únicamente hacia delante. La segunda forma del método `prepareStatement` recibe, además de la cadena de caracteres, un argumento que indica el tipo de juego de registros y otro que determina las posibilidades de modificación de los datos contenidos en el juego de registros.

Se pueden utilizar las mismas constantes que para el método `createStatement`.

```
PreparedStatement pstmt=cnx.prepareStatement("select * from  
PERSONAS where nombre=?",ResultSet.TYPE_SCROLL_SENSITIVE,  
ResultSet.CONCUR_READ_ONLY);
```

Antes de la ejecución de la instrucción SQL, debemos facilitar un valor para cada uno de

los signos de interrogación que representan un parámetro. Para ello, el objeto `PreparedStatement` dispone de numerosos métodos que permiten asignar un valor a un parámetro. Cada uno de estos métodos corresponde al tipo de datos SQL que se ha de insertar en lugar de cada signo de interrogación. Estos métodos siguen la siguiente nomenclatura: `setXxx`, donde `Xxx` representa un tipo de datos SQL. Cada uno de estos métodos recibe como primer argumento un entero que corresponde al rango del parámetro en la instrucción SQL. El primer parámetro se sitúa en el rango 1. El segundo argumento corresponde al valor que se ha de transferir en el parámetro. El tipo de este argumento corresponde por supuesto al tipo de datos SQL que hay que trasladar hacia el parámetro. El driver JDBC convierte a continuación el tipo Java en tipo SQL.

Por ejemplo, el método `setInt (int indiceParam, int value)` efectúa una conversión del tipo `int` java en tipo `INTEGER` SQL. Cada proveedor de un controlador JDBC proporciona documentación que enumera las relaciones entre los tipos Java y los tipos SQL. Esta documentación para el controlador JDBC está disponible en la siguiente dirección: <https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-reference-type-conversions.html>

Se conservan los valores almacenados en los parámetros de una ejecución de la instrucción SQL a la siguiente. El método `clearParameters` permite reinicializar el conjunto de los parámetros.

Se pueden utilizar los parámetros en una instrucción SQL en sustitución de valores, pero nunca para sustituir el nombre de un campo, y aun menos de un operador. Por supuesto, la sintaxis siguiente está prohibida:

```
PreparedStatement pstmt = cnx.prepareStatement("select * from  
PERSONAS where ? = ?", ResultSet.TYPE_SCROLL_SENSITIVE,  
ResultSet.CONCUR_READ_ONLY);  
pstmt.setString(1, "nombre");  
pstmt.setString(2, "McQueen");  
rs=pstmt.executeQuery();
```

Los demás métodos disponibles con un objeto `PreparedStatement` son idénticos a los definidos para un objeto `Statement`, ya que la interfaz `PreparedStatement` hereda directamente de la interfaz `Statement`. El siguiente ejemplo muestra la lista de personas que responden al nombre proporcionado por

el usuario de la aplicación.

```
package es.eni.editions.jdbc.peticion;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class TestPreparedStatement {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            PreparedStatement pstmt =
cnx.prepareStatement("select * from PERSONAS where nombre = ?",
                        ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY)) {

            BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Introducir el nombre de la
persona
buscada:");

            String nombre = br.readLine();
            //Aplicar argumentos
            pstmt.setString(1, nombre);
            //Ejecutar la consulta y obtener el resultado
            ResultSet rs = pstmt.executeQuery();
            //Mientras haya registros
            while (rs.next()) {
                //Recorrer en un bucle el número de
columnas para
mostrar la información

                for (int i = 1; i <=
```



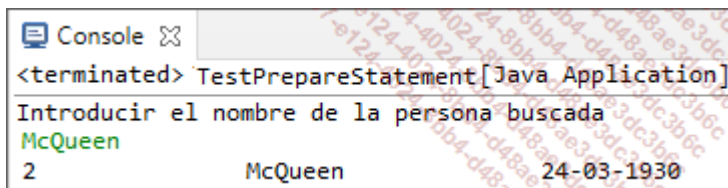
```

rs.getMetaData().getColumnCount(); i++) {
    System.out.print(rs.getString(i) +
"\t");
    }
    System.out.println();
}
} catch (SQLException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
}

private static Connection getConnection() throws SQLException {
    String url = "jdbc:mysql://localhost/
demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

La ejecución de este programa da el siguiente resultado:



El tipo `PreparedStatement` también se usa mucho para las operaciones de escritura (insertar, actualizar, eliminar). Dentro del marco de la inserción, una funcionalidad muy apreciable hace posible repetir la posible clave autogenerada por la base de datos (este suele ser el caso de una clave primaria del tipo de incremento automático).

Para recuperar la clave generada automáticamente, debe usar la segunda firma del método `prepareStatement` del objeto de tipo `Connection`. Este método espera como parámetro una cadena de caracteres que corresponde a la consulta parametrizada, seguida

de un número entero que indica si el desarrollador desea recuperar la clave autogenerada por la base de datos. La constante `Statement.RETURN_GENERATED_KEYS` se usa para especificar esto.

La ejecución de la consulta parametrizada con el método `executeUpdate` de la interfaz `PreparedStatement` permite evaluar un `ResultSet` que contiene un único registro con la clave autogenerada. El acceso a este `ResultSet` se hace a través del método `getGeneratedKeys` de la interfaz `Statement`, de la que deriva la interfaz `PreparedStatement`. La clave está disponible en la primera columna del único registro de `ResultSet`.

El siguiente código resalta esta funcionalidad.

```
package es.eni.editions.jdbc.consulta;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestInsertionViaPreparedStatement {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            //Consulta de inserción con los campos
            obligatorios
            PreparedStatement pstmt =
            cnx.prepareStatement("insert into PERSONAS(nombre, apellido)
            values(?,?)",
                                Statement.RETURN_GENERATED_KEYS))
        {
```

```

        BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Introducir el apellido de la
persona a agregar:");
        String nombre = br.readLine();
        System.out.println("Introducir el nombre de la
persona a agregar:");
        String apellido = br.readLine();
        //Aplicar los argumentos
        pstmt.setString(1, nombre);
        pstmt.setString(2, apellido);
        //Ejecutar la consulta y obtener el resultado
        pstmt.executeUpdate();
        //Recuperar el ResultSet que contiene
eventualmente
una clave

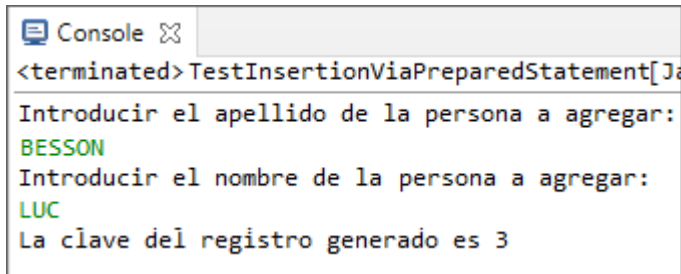
        ResultSet rsClaveGenerada =
pstmt.getGeneratedKeys();
        //Si hay un registro, es que hay una clave
        if(rsClaveGenerada.next())
        {
            System.out.println("La clave del registro
generado es " + rsClaveGenerada.getInt(1));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static Connection getConnection() throws SQLException {
    String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}

```

```
}
```

La ejecución de este programa da el siguiente resultado:



```
Console
<terminated> TestInsertionViaPreparedStatement[Ja
Introducir el apellido de la persona a agregar:
BESSON
Introducir el nombre de la persona a agregar:
LUC
La clave del registro generado es 3
```

c. Ejecución de procedimientos almacenados con el objeto CallableStatement

Un procedimiento almacenado representa código SQL almacenado en el servidor. Este planteamiento ofrece varias ventajas:

- Los tratamientos y los tiempos de respuesta se han mejorado.
- El código SQL se puede compartir entre varias aplicaciones.

A cambio, las aplicaciones se vuelven mucho más dependientes del servidor de base de datos. El cambio de servidor le obligará seguramente a escribir de nuevo los procedimientos almacenados, ya que la sintaxis de un procedimiento almacenado es propia de cada servidor. Los procedimientos almacenados son accesibles desde Java mediante el objeto `CallableStatement`. Como para los objetos `Statement` y `PreparedStatement`, sigue siendo la conexión la que facilitará una instancia de esta clase. En este caso, se debe utilizar el método `prepareCall`. Este método recibe como argumento una cadena de caracteres que identifica el procedimiento almacenado que hay que invocar. Por el contrario, la sintaxis de esta cadena de caracteres es un poco especial, ya que no es suficiente indicar el nombre del procedimiento almacenado.

```
{call nombreProcedimiento( ?, ?, ... )}
```

En esta sintaxis, los signos de interrogación representan los parámetros que recibe el procedimiento almacenado. Como para el objeto `PreparedStatement`, los valores de estos parámetros deben facilitarse mediante los métodos `setXxx` correspondientes al tipo del parámetro.

Para los argumentos utilizados como salida del procedimiento almacenado, (argumentos taguados con la palabra clave `out`), su ejecución almacenará un valor en él, y debemos informar el objeto `CallableStatement` invocando el método `registerOutParameter`. Este método recibe como primer argumento el índice del parámetro 1 para el valor de retorno del procedimiento almacenado, y luego como segundo argumento el tipo SQL del parámetro. Se puede indicar este tipo con una de las constantes definidas en la interfaz `java.sql.Types`. Tras la ejecución del procedimiento almacenado, el valor de los parámetros utilizados como salida está accesible mediante los métodos `getXxx`, donde `Xxx` representa el tipo SQL del parámetro. Estos métodos reciben como argumento el índice del parámetro en la llamada al procedimiento almacenado.

Como para los métodos `createStatement` y `prepareStatement`, el método `prepareCall` propone una segunda sintaxis que permite indicar las características de un eventual juego de registros generado por la ejecución del procedimiento almacenado.

Para ilustrar el uso del objeto `CallableStatement`, vamos a emplear el procedimiento almacenado siguiente:

```
CREATE DEFINER=`mysqluser`@`localhost` PROCEDURE
`numeroPersonasPorNombre`(in ordenar varchar(4), out numeroTotal
integer)
BEGIN
    select count(*) into numeroTotal from PERSONAS;
    if(ordenar='ASC')then
        select count(*) as numeroPersonas,
nombre from PERSONAS group by nombre ASC;
    else
        select count(*) as numeroPersonas,
nombre from PERSONAS group by nombre DESC;
    end if;
END
```

El procedimiento devuelve un conjunto de registros que consta de los nombres de las personas y el número de personas para cada nombre. Este conjunto de registros se ordena según el primer parámetro del procedimiento. El segundo parámetro es de tipo `out`.

Permite devolver a la persona que llama el número total de personas.

El siguiente código se utiliza para ejecutar este procedimiento almacenado y explotar los resultados:

```
package es.eni.editions.jdbc.consulta;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;

public class TestCallableStatement {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            CallableStatement cstm =
cnx.prepareCall("{call numeroPersonasPorNombre ( ?, ? )}",
                ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY)) {
            System.out.println("Introducir la ordenación del
resultado
(ASC o DESC):");

            BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));
            String ordenar = br.readLine();
            // Aplicación de los argumentos
            cstm.setString(1, ordenar);
            cstm.registerOutParameter(2,
java.sql.Types.INTEGER);
            // Ejecución de la consulta
            boolean existeResultado = cstm.execute();
            // Recuperación de los argumentos del tipo out
            int numeroPersonas = cstm.getInt(2);
            System.out.println("número de personas:
```

```

" + numeroPersonas);

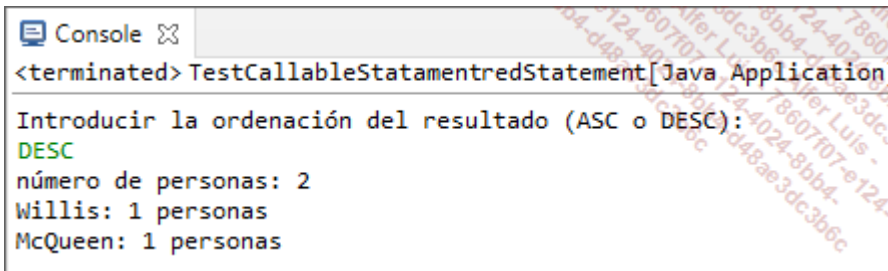
        if (existeResultado) {
            ResultSet rs = cstm.getResultSet();
            while (rs.next()) {

System.out.println(rs.getString("nombre")
+ ": " + rs.getInt("numeroPersonas") + " personas");
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private static Connection getConnection() throws SQLException {
    String url = "jdbc:mysql://localhost/
demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

La ejecución de este programa da el siguiente resultado:



```

Console
<terminated> TestCallableStatementredStatement[Java Application]
Introducir la ordenación del resultado (ASC o DESC):
DESC
número de personas: 2
Willis: 1 personas
McQueen: 1 personas

```

5. Utilización de los juegos de registros con la interfaz

ResultSet

Cuando se ejecuta la instrucción SQL `select` con el método `executeQuery` de un objeto `Statement`, `PreparedStatement` o `CallableStatement`, este devuelve un objeto `ResultSet`. A través de este objeto `ResultSet` vamos a poder intervenir sobre el juego de registros. Nuestras posibilidades de acción sobre este juego de registros están determinadas por las características del objeto `ResultSet`. Se fijan estas características en el momento de la creación de los objetos `Statement`, `PreparedStatement` o `CallableStatement` en función de los argumentos pasados durante la llamada a los métodos `createStatement`, `prepareStatement` o `prepareCall`.

El primer argumento determina el tipo del juego de registros. Se definen las constantes siguientes:

- `ResultSet.TYPE_FORWARD_ONLY`: el juego de registros se desplazará solo hacia delante.
- `ResultSet.TYPE_SCROLL_INSENSITIVE`: el juego de registros podrá recorrerse en ambos sentidos, pero será insensible a los cambios realizados en la base de datos por otros usuarios.
- `ResultSet.TYPE_SCROLL_SENSITIVE`: el juego de registros podrá recorrerse en ambos sentidos y será sensible a los cambios realizados en la base de datos por otros usuarios.

El segundo argumento determina las posibilidades de modificación de los datos contenidos en el juego de registros. Se definen las dos constantes siguientes:

- `ResultSet.CONCUR_READ_ONLY`: los registros son de solo lectura.
- `ResultSet.CONCUR_UPDATABLE`: los registros pueden modificarse en el juego de registros.

Por supuesto, hace falta que las acciones ejecutadas sobre el objeto `ResultSet` sean compatibles con estas características; en caso contrario, se producirá una excepción. Es posible comprobar las características de un objeto `ResultSet` utilizando los métodos `getType` y `getConcurrency`. El siguiente código muestra las características del `ResultSet`. Está disponible en la clase `TestTypeResultSet` del paquete

es.eni.editions.jdbc.consulta.

```
private static void mostrarTipoResultSet(ResultSet rs)
{
    try {
        switch (rs.getType())
        {
            case ResultSet.TYPE_FORWARD_ONLY:
                System.out.println("ResultSet sólo podrá
recorrerse hacia delante");
                break;
            case ResultSet.TYPE_SCROLL_INSENSITIVE:
                System.out.println("ResultSet podrá
recorrerse en ambos sentidos");
                System.out.println("no es sensible a
las modificaciones realizadas por otros usuarios");
                break;
            case ResultSet.TYPE_SCROLL_SENSITIVE:
                System.out.println("ResultSet puede
recorrerse en ambos sentidos");
                System.out.println("es sensible a las
modificaciones realizadas por otros usuarios");
                break;
        }
        switch (rs.getConcurrency())
        {
            case ResultSet.CONCUR_READ_ONLY:
                System.out.println("los datos
contenidos en el ResultSet son de solo lectura");
                break;
            case ResultSet.CONCUR_UPDATABLE:
                System.out.println("los datos
contenidos en el ResultSet son modificables");
                break;
        }
    }
    catch (SQLException e)
    {

```

```
        e.printStackTrace();
    }
}
```

a. Posicionamiento en un ResultSet

El objeto `ResultSet` gestiona un puntero de registro que determina sobre qué registro van a intervenir los métodos ejecutados sobre el propio `ResultSet`. Este registro se llama a veces registro activo o registro en curso. El objeto `ResultSet` contiene siempre dos registros ficticios que sirven de referencia para el principio del `ResultSet` (BOF - *Beginning Of File*) y para el final del `ResultSet` (EOF - *End Of File*). El puntero de registro se puede colocar sobre uno de estos dos registros, pero nunca antes del registro BOF ni después del registro EOF. Estos registros no contienen datos y una operación de lectura o escritura sobre ellos genera una excepción. En el momento de la creación del `ResultSet`, el puntero se coloca antes del primer registro (BOF). Cuando el conjunto de registros no contiene resultados, necesariamente hay estos dos registros ficticios.

Existen muchos métodos disponibles para gestionar el puntero de registros:

- `boolean absolute(int position)`: desplaza el puntero de registro sobre el registro especificado. La numeración de los registros empieza en 1. Si el valor del argumento posición es negativo, se efectúa el desplazamiento partiendo del final del `ResultSet`. Si el número de registro no existe, se coloca el puntero sobre el registro BOF si el valor es negativo e inferior al número de registros, o sobre el registro EOF si el valor es positivo y superior al número de registros. Este método devuelve `true` si el puntero está colocado en un registro válido y `false` en el caso contrario (BOF o EOF).
- `boolean relative(int displacement)`: desplaza el puntero del número de registros especificados por el argumento `displacement`. Si el valor de este argumento es positivo, el cursor baja en el juego de registros, y si el valor es negativo, el cursor sube en el juego de registros. Este método devuelve `true` si el puntero está colocado sobre un registro válido y `false` en el caso contrario (BOF o EOF).
- `void beforeFirst()`: desplaza el puntero de registros antes del primer registro (BOF).

- `void afterLast()`: desplaza el puntero de registros después del último registro (EOF).
- `boolean first()`: desplaza el puntero de registros sobre el primer registro. Este método devuelve `true` si hay un registro en el `ResultSet` y `false` en caso contrario.
- `boolean last()`: desplaza el puntero de registros sobre el último registro. Este método devuelve `true` si hay un registro en el `ResultSet` y `false` en caso contrario.
- `boolean next()`: desplaza el puntero de registros sobre el registro que sigue al registro actual. Este método devuelve `true` si el puntero está en un registro válido y `false` en caso contrario (EOF).
- `boolean previous()`: desplaza el puntero de registros sobre el registro anterior al registro actual. Este método devuelve `true` si el puntero está sobre un registro válido y `false` en caso contrario (BOF).

Para todos estos métodos, excepto para el método `next`, es absolutamente necesario que el `ResultSet` sea de tipo `SCROLL_SENSITIVE` o `SCROLL_INSENSITIVE`. Si el `ResultSet` es de tipo `FORWARD_ONLY`, solo el método `next` funciona y en este caso los demás métodos producen una excepción. Los métodos siguientes permiten comprobar la posición del puntero de registros:

- `boolean isBeforeFirst()`: devuelve `true` si el puntero está situado antes del primer registro (BOF).
- `boolean isAfterLast()`: devuelve `true` si el puntero está situado después del último registro (EOF).
- `boolean isFirst()`: devuelve `true` si el puntero está situado en el primer registro.
- `boolean isLast()`: devuelve `true` si el puntero está situado en el último registro.
- `int getRow()`: devuelve el número del registro sobre el cual se encuentra el puntero de registros. Si no existe un registro en curso (BOF o EOF), devuelve el valor 0.

El programa siguiente permite mostrar la posición del puntero en el `ResultSet`:

```
package es.eni.editions.jdbc.consulta;
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
public class TestPosicionPunteroResultSet {
```

```
    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            Statement stm = cnx.createStatement
                (ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)) {
```

```
            // Ejecutar la consulta y obtener el resultado
            ResultSet rs = stm.executeQuery("select *
from PERSONAS");
```

```
            mostrarPosicionPuntero(rs);
            // Mientras haya registros
            while (rs.next()) {
                mostrarPosicionPuntero(rs);
            }
            mostrarPosicionPuntero(rs);
        } catch (SQLException e) {
            e.printStackTrace();
        }
```

```
    }
```

```
    private static void mostrarPosicionPuntero(ResultSet rs) {
        try {
            if (rs.isBeforeFirst()) {
                System.out.println("el puntero
está antes del primer registro");
            }
            if (rs.isAfterLast()) {
```

```

        System.out.println("el puntero está
después del último registro");
    }
    if (rs.isFirst()) {
        System.out.println("el puntero está
en el primer registro");
    }
    if (rs.isLast()) {
        System.out.println("el puntero está
en el último registro");
    }
    int posición;
    posición=rs.getRow();
    if (posición!=0) {
        System.out.println("es el registro
número " + posición);
    }
} catch (SQLException e) {
    e.printStackTrace();
}

private static Connection getConnection() throws SQLException {
    String url = "jdbc:mysql:
//localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

La ejecución de este código permite tener el siguiente resultado:

Cada uno de estos métodos se presenta bajo dos formas. La primera recibe como argumento el número de la columna de la que se quiere obtener el valor. La numeración empieza con el 1. La segunda versión recibe una cadena de caracteres que representa el nombre de la columna en la base de datos. Si la petición utilizada para crear el `ResultSet` contiene alias, entonces las columnas llevan el nombre del alias y no el nombre del campo en la base de datos. Para una mejor legibilidad del código, es preferible usar los nombres de las columnas en vez de sus índices. Cuando en la base de datos un campo no contiene valores (`NULL`), los métodos devuelven un valor igual a 0 para los campos numéricos, un valor `false` para los campos booleanos y un valor `null` para los demás tipos. En algunos casos, podrían existir ciertas dudas relativas al valor realmente almacenado en la base de datos.

Por ejemplo, el método `getInt` puede devolver un valor igual a cero porque está realmente almacenado este valor en la base de datos o porque este campo admite valores nulos en la base de datos. Para resolver la duda, el método `wasNull` devuelve un booleano igual a `true` si el campo donde se hizo la última operación de lectura sobre el `ResultSet` contenía efectivamente un valor `null`.

El siguiente código resalta la manipulación de los tipos Java y SQL al escribir y leer información. Se enfatiza la manipulación de fechas.

Para escribir una fecha en una consulta parametrizada, es posible usar el método `setDate`, que espera en el segundo parámetro un objeto de tipo `java.sql.Date`. Un objeto de este tipo no debe manipularse en el resto de la aplicación. Es preferible utilizar un objeto de tipo `LocalDate` (si la parte horaria no es útil) o un objeto de tipo `LocalTime`. El método `java.sql.Date.valueOf` espera como parámetro un objeto de tipo `LocalDate` para obtener un objeto de tipo `java.sql.Date`.

Para escribir una fecha en una consulta parametrizada, también es posible usar el método `setObject`, que acepta un objeto como segundo parámetro. Este objeto puede ser de tipo `LocalDate`. El controlador JDBC podrá usar este objeto para guardarlo en una columna de tipo `Date`.

Para leer los datos, se adaptan los métodos `getXxx` de la interfaz `ResultSet`. La lectura de una fecha se realiza con el método `getDate`. Devuelve un objeto de tipo

java.sql.Date. Es posible obtener un objeto de tipo LocalDate a partir de un objeto de tipo java.sql.Date utilizando el método toLocalDate.

```
package es.eni.editions.jdbc.consulta;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.TimeZone;

public class TestMappingTypeJavaSQL {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
             PreparedStatement pstmt =
cnx.prepareStatement("insert into PERSONAS(nombre, apellido,
fechaDeNacimiento) values(?,?,?)");
             Statement stm = cnx.createStatement()) {
            insertarPersonas(pstmt);
            leerPersonas(stm);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void insertarPersonas(PreparedStatement pstmt)
throws SQLException {
        //La instrucción siguiente permite hacer corresponder
        //la timezone de JVM con la timezone del servidor MySQL.
        TimeZone.setDefault(TimeZone.getTimeZone("UTC"));

        //Agregar una primera persona
```



```
//Aplicar los argumentos
pstm.setString(1, "Eastwood");
pstm.setString(2, "Clint");
//Escritura de la fecha de nacimiento
//pstm.setObject(3, LocalDate.of(1930, 5, 31));
//o
pstm.setDate(3, Date.valueOf(LocalDate.of(1930, 5, 31)));

//Ejecutar la consulta
int resultado = pstm.executeUpdate();
if(resultado==1)
{
    System.out.println("La inserción ha tenido éxito");
}

//Agregar una segunda persona
//Aplicar los argumentos
pstm.setString(1, "Tarantino");
pstm.setString(2, "Quentin");
pstm setObject(3, null);

//Ejecutar la consulta
resultado = pstm.executeUpdate();
if(resultado==1)
{
    System.out.println("La inserción ha tenido éxito");
}

}

private static void leerPersonas(Statement stm) throws SQLException
{
    ResultSet rs = stm.executeQuery("select id, nombre, apellido, fechaDeNacimiento from PERSONAS");
    System.out.println("id\t\tnombre\t\tapellido\t\t
```

```

tfecha de nacimiento");
    while(rs.next())
    {
        int id = rs.getInt("id");
        System.out.print(rs.isNull()?"desconocido":id);
        System.out.print("\t\t");

        String nombre = rs.getString("nombre");

        System.out.print(rs.isNull()?"desconocido":nombre);
        System.out.print("\t\t");

        String apellido = rs.getString("apellido");

        System.out.print(rs.isNull()?"desconocido":apellido);
        System.out.print("\t\t");

        Date sqlDate = rs.getDate("fechaDeNacimiento");
        System.out.println(rs.isNull()?"desconocida":
        DateTimeFormatter.ofPattern("dd/MM/yyyy").format(sqlDate.toLocalDate()))

    }

}

private static Connection getConnection() throws SQLException {
    String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

El resultado de la ejecución del código es el siguiente:

```
Console [X]
<terminated> TestMappingTypeJavaSQL[Java Application]
La inserción ha tenido éxito
La inserción ha tenido éxito
id    nombre    apellido    fecha de nacimiento
1     Steeve    McQueen    24/03/1930
2     Bruce     Willis     19/03/1955
3     Clint     Eastwood   31/05/1930
4     Quentin   Tarantino  desconocida
```

c. Modificación de los datos en un ResultSet

La modificación de los datos se efectúa simplemente utilizando los métodos `updateXxx`, donde `Xxx` corresponde al tipo de datos de la columna que se ha de actualizar. Como para el caso de los métodos `getXxx`, estos disponen de dos versiones: una recibe como argumento el índice de la columna que se ha de actualizar, la segunda recibe como argumento una cadena de caracteres que representa el nombre de la columna en la base de datos. Si la petición utilizada para crear el `ResultSet` contiene alias, entonces las columnas llevan el nombre del alias y no el nombre del campo en la base de datos. Para una mejor legibilidad del código, es preferible por supuesto usar los nombres de las columnas en vez de sus índices.

El tipo del segundo argumento esperado por estos métodos corresponde, lógicamente, al tipo de datos que hay que actualizar en el `ResultSet`. A continuación, se deben validar las modificaciones con el método `updateRow` o cancelar con el método `cancelRowUpdates`. El `ResultSet` debe ser obligatoriamente de tipo `CONCUR_UPDATABLE` para poder ser modificado. En caso contrario, la ejecución de estos métodos genera una excepción.

El siguiente ejemplo actualiza los nombres de las personas que usan `ResultSet`. Por supuesto, en un contexto profesional, sería mejor usar una instrucción SQL para realizar esta tarea.

```
package es.eni.editions.jdbc.consulta;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.sql.Statement;
import java.time.format.DateTimeFormatter;

public class TestModificationViaResultSet {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            Statement stmModif = cnx.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
            Statement stmLectura =
cnx.createStatement()) {
            leerYModificarPersonas(stmModif);
            leerPersonas(stmLectura);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void leerYModificarPersonas(Statement stmModif)
throws SQLException {
        ResultSet rs = stmModif.executeQuery("select id, nombre,
apellido, fechaDenacimiento from PERSONAS");
        while(rs.next())
        {
            //Modificación del nombre, poniendo en mayúsculas
rs.updateString("nombre", rs.getString("nombre").
toUpperCase());

            rs.updateRow();
        }
    }

    private static void leerPersonas(Statement stm) throws SQLException
{
        ResultSet rs = stm.executeQuery("select id, nombre,
apellido,
fechaDeNacimiento from PERSONAS");
        System.out.println("id\t\tnombre\t\tApellido\t\t
tfecha de nacimiento");
```

```

        while(rs.next())
        {
            int id = rs.getInt("id");
            System.out.print(rs.isNull()?"desconocido":id);
            System.out.print("\t\t");

            String nombre = rs.getString("nombre");

            System.out.print(rs.isNull()?"desconocido":nombre);
            System.out.print("\t\t");

            String apellido = rs.getString("apellido");

            System.out.print(rs.isNull()?"desconocido":apellido);
            System.out.print("\t\t");

            Date sqlDate = rs.getDate("fechaDeNacimiento");
            System.out.println(rs.isNull()?"desconocida":
            DateTimeFormatter.ofPattern("dd/MM/yyyy").format(sqlDate.toLocalDate()))

        }

    }

    private static Connection getConnection() throws SQLException {
        String url = "jdbc:mysql://localhost/demo_java?
serverTimezone=UTC";
        String user = "mysqluser";
        String password = "Pa$$w0rd";
        return DriverManager.getConnection(url, user, password);
    }
}

```

La ejecución de este programa da el siguiente resultado:

Console			
<terminated> TestModificationViaResultSet[Java Application]			
id	nombre	apellido	fecha de nacimiento
1	STEEVE	McQueen	24/03/1930
2	BRUCE	Willis	19/03/1955
3	CLINT	Eastwood	31/05/1930
4	QUENTIN	Tarantino	desconocida

d. Supresión de datos en un ResultSet

La supresión de una línea se realiza de manera sencilla al situar el puntero sobre la línea que se ha de suprimir y llamando, a continuación, al método `deleteRow`. La línea se elimina inmediatamente del `ResultSet` y de la base de datos. La posición del puntero de registros después del borrado depende del driver usado. Algunos drivers desplazan el puntero al registro siguiente, otros lo desplazan al anterior, e incluso algunos no modifican la posición del puntero. En este caso, hay que utilizar alguno de los métodos de desplazamiento para situar el puntero sobre un registro utilizable. El `ResultSet` debe ser obligatoriamente de tipo `CONCUR_UPDATABLE` para poder suprimir datos en él. En caso contrario, la ejecución de un método `deleteRow` produce una excepción.

```
package es.eni.editions.jdbc.consulta;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.format.DateTimeFormatter;

public class TestEliminacionViaResultSet {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            Statement stmEliminacion =
cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
            Statement stmLectura =
cnx.createStatement()) {
            LeerYEliminarPersonas(stmEliminacion);
        }
    }
}
```

```

        leerPersonas(stmLectura);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

private static void leerYEliminarPersonas(Statement
stmEliminacion)
throws SQLException {
    ResultSet rs = stmEliminacion.executeQuery("select id,
nombre,
apellido, fechaDeNacimiento from PERSONAS");
    while(rs.next())
    {
        //Eliminación del registro si la persona se llama
Eastwood

        System.out.println("Persona actual:
" + rs.getString("nombre"));

        if("Eastwood".equalsIgnoreCase(rs.getString("nombre")))
        {
            System.out.println("Elimino la persona");
            rs.deleteRow();
        }
    }
}

private static void leerPersonas(Statement stm) throws SQLException
{
    ResultSet rs = stm.executeQuery("select id, nombre,
apellido,
fechaDeNacimiento from PERSONAS");
    System.out.println("id\t\tnombre\t\tapellido\t\t
tfecha de nacimiento");
    while(rs.next())
    {
        int id = rs.getInt("id");
        System.out.print(rs.isNull()?"desconocido":id);

```

```

        System.out.print("\t\t");

        String nombre = rs.getString("nombre");

        System.out.print(rs.isNull()? "desconocido": nombre);
        System.out.print("\t\t");

        String apellido = rs.getString("apellido");

        System.out.print(rs.isNull()? "desconocido": apellido);
        System.out.print("\t\t");

        Date sqlDate = rs.getDate("fechaDeNacimiento");
        System.out.println(rs.isNull()? "desconocida":
        DateTimeFormatter.ofPattern("dd/MM/yyyy").format(sqlDate.toLocalDate()))

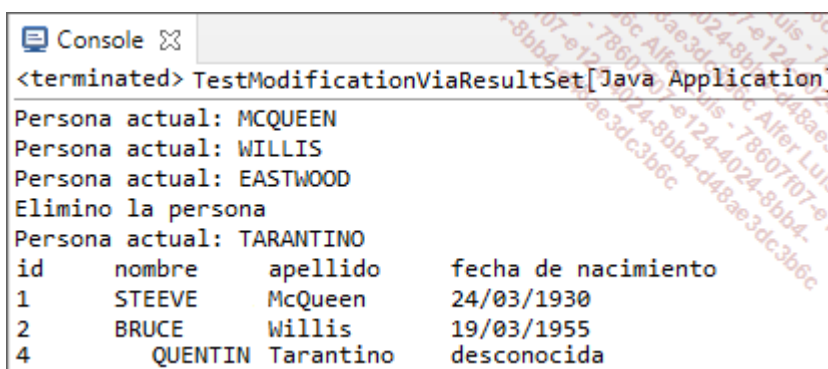
    }

}

private static Connection getConnection() throws SQLException {
    String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

La ejecución de este programa da el siguiente resultado:



```

<terminated> TestModificationViaResultSet[Java Application]
Persona actual: MCQUEEN
Persona actual: WILLIS
Persona actual: EASTWOOD
Elimino la persona
Persona actual: TARANTINO
id      nombre      apellido      fecha de nacimiento
1       STEEVE      McQueen      24/03/1930
2       BRUCE       Willis       19/03/1955
4       QUENTIN     Tarantino    desconocida

```


Este resultado permite concluir que el controlador JDBC para MySQL no modifica la posición del cursor después de la eliminación de un registro.

e. Inserción de datos en un ResultSet

Cada objeto `ResultSet` contiene una línea especial destinada a la inserción de datos. El puntero de registro se debe situar, previamente, en esta línea especial con la instrucción `moveToInsertRow`. Una vez situado el puntero sobre esta línea, es posible actualizarla mediante los métodos `updateXXX`. A continuación, se debe validar la inserción de la línea mediante el método `insertRow`. Este método provoca la actualización de la base de datos. La línea de inserción se convierte en este momento en una línea normal del `ResultSet` y el puntero de registro se posiciona en esta línea. Puede regresar a la línea en la que estaba antes de la inserción gracias al método `moveToCurrentRow`. Si no proporcionamos valores para todas las columnas, se insertarán valores `null` en la base de datos para las columnas que acepten valores `null`. La base de datos debe aceptar los valores nulos para estos campos porque, en caso contrario, se produce una excepción. El `ResultSet` debe ser obligatoriamente de tipo `CONCUR_UPDATABLE` para poder insertar datos en él. En caso contrario, la ejecución del método `moveToInsertRow` genera una excepción.

```
package es.eni.editions.jdbc.consulta;

import java.sql.Connection;
import java.sql.Date;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.time.format.DateTimeFormatter;

public class TestInsercionViaResultSet {

    public static void main(String[] args) {
        try (Connection cnx = getConnection();
            Statement stmInsertion =
cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

```

        Statement stmLectura =
cnx.createStatement()) {
            agregarPersona(stmInsercion);
            leerPersonas(stmLectura);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void agregarPersona(Statement stmInsertion)
throws SQLException {
        ResultSet rs = stmInsertion.executeQuery("select id,
nombre,
apellido, fechaDeNacimiento from PERSONAS");
        rs.moveToInsertRow();
        rs.updateString("nombre", "BESSON");
        rs.updateString("apellido", "Luc");
        rs.insertRow();
    }

    private static void lirePersonnes(Statement stm) throws
SQLException {
        ResultSet rs = stm.executeQuery("select id, nombre,
apellido,
fechaDeNacimiento from PERSONAS");
        System.out.println("id\t\tnombre\t\tapellido\t\t
tfecha de nacimiento");
        while(rs.next())
        {
            int id = rs.getInt("id");
            System.out.print(rs.isNull()?"desconocido":id);
            System.out.print("\t\t");

            String nombre = rs.getString("nombre");

System.out.print(rs.isNull()?"desconocida":nombre);
            System.out.print("\t\t");

            String apellido = rs.getString("apellido");

```

```

System.out.print(rs.isNull()?"desconocido":apellido);
        System.out.print("\t\t");
        Date sqlDate = rs.getDate("fechaDeNacimiento");
        System.out.println(rs.isNull()?"desconocida":
DateTimeFormatter.ofPattern("dd/MM/yyyy").format(sqlDate.toLocalDate()))

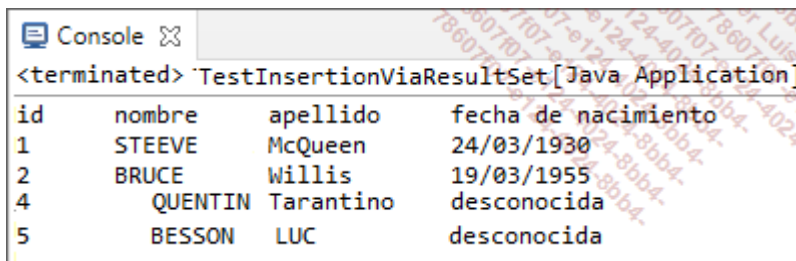
    }

}

private static Connection getConnection() throws SQLException {
    String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

La ejecución de este programa da el siguiente resultado:



id	nombre	apellido	fecha de nacimiento
1	STEEVE	McQueen	24/03/1930
2	BRUCE	Willis	19/03/1955
4	QUENTIN	Tarantino	desconocida
5	BESSON	LUC	desconocida

6. Gestión de las transacciones

Las transacciones van a permitir asegurar que un conjunto de instrucciones SQL se ejecuten todas con éxito. Si una de ellas falla, las instrucciones que se completaron con éxito de previamente simplemente se cancelan. Esto hace posible mantener consistente la información en la base de datos. La transferencia de un importe de dinero entre dos cuentas bancarias representa el ejemplo clásico en el cual se necesita una transacción. Imagínese la situación siguiente: nuestro banco debe realizar el cobro de un cheque de 1000 € a adeudar en la cuenta 12345 y a abonar en la cuenta 67890. Por razones de seguridad, después de

cada operación efectuada en una cuenta (débito o crédito), se edita un informe. A continuación, presentamos un extracto del código que puede llevar a cabo estas operaciones.

```
public static void movimiento(String cuentaDebito,String
cuentaCredito,double importe)
{
    try
    {

        Connection cnx=...;

        PreparedStatement stm;
        stm=cnx.prepareStatement("update cuentas set saldo=saldo
+ ? where numero=?");
        stm.setDouble(1,importe * -1);
        stm.setString(2,cuentaDebito);
        stm.executeUpdate();
        impresiónInforme(cuentaDebito, importe);
        stm.setDouble(1,importe);
        stm.setString(2,cuentaCredito);
        stm.executeUpdate();
        impresiónInforme(cuentaCredito, importe);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

Para el 99,9999 % de los movimientos realizados con este código, no hay ningún problema; pero un buen día la impresora que se encarga de las ediciones se bloquea y este bloqueo genera una excepción en el método `impresiónInforme`. A priori esta excepción no es un problema, ya que la llamada a este método está ubicada dentro de un bloque `try` y existe un bloque `catch` encargado de gestionar la excepción. Sin embargo, no debemos olvidar que, si se produce una excepción y se ejecuta un bloque `catch`, la ejecución continua por la instrucción que sigue al bloque `catch`. En este supuesto, las instrucciones

ubicadas entre la que produjo la excepción y el final de bloque `try` simplemente no se ejecutan. En nuestro caso, eso puede resultar problemático si esta excepción se produce en el método `impresiónInforme` ejecutado justo después de la operación de débito. La operación de crédito, simplemente, no se lleva a cabo. Por lo tanto, se pierde el importe. Una solución consiste en cancelar las instrucciones SQL ya ejecutadas.

En realidad, es este mecanismo el que se pone en marcha durante una transacción pero, por supuesto, de manera automática. Es a nivel de la conexión hacia el servidor de base de datos donde se gestionan las transacciones.

a. Puesta en marcha de las transacciones

Hasta ahora, no nos hemos preocupado por las transacciones y, sin embargo, las estamos realizando desde nuestra primera instrucción SQL ejecutada mediante JDBC. El funcionamiento por defecto de la API JDBC consiste efectivamente en incluir cada instrucción ejecutada en una transacción y, a continuación, validar la transacción si la instrucción se ha ejecutado correctamente (`commit`) o cancelarla en el caso contrario (`rollback`). Este modo de funcionamiento se llama modo `autoCommit`. Si desea gestionar usted mismo el fin de una transacción validando o anulando todas las instrucciones que contiene, debe desactivar el modo `autoCommit` invocando el método `setAutoCommit(false)` sobre el objeto `Connection`. En lo sucesivo, será responsable de poner fin a las transacciones. Los métodos `commit` y `rollback` del objeto `Connection` permiten validar o cancelar las instrucciones ejecutadas desde el principio de la transacción. Una nueva transacción empieza automáticamente tras el fin de la anterior o desde la apertura de la conexión. El código que permite transferir un importe entre dos cuentas debería tener la forma siguiente.

Para que el siguiente programa funcione, es necesario crear una tabla con un conjunto de prueba. Aquí está el script para crear la tabla CUENTAS y agregar dos cuentas:

```
create tabla CUENTAS(  
    numero char(9) primary key,  
    saldo integer not null  
);  
insert into CUENTAS(numero, saldo) values("123456789",2500);  
insert into CUENTAS(numero, saldo) values("987654321",10000);
```

Ahora que la base de datos está configurada, el siguiente código permite destacar el funcionamiento de las transacciones:

```
package es.eni.editions.jdbc.consulta;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class TestTransaction {

    public static void main(String[] args) {
        movimiento("123456789", "987654321", 1000);
    }

    public static void movimiento(String cuentaDebito,
String cuentaCredito, double suma) {
        try (Connection cnx = getConnection()) {
            try (PreparedStatement pstmt =
cnx.prepareStatement("update CUENTAS set saldo=saldo + ? where
numero=?")); {

                cnx.setAutoCommit(false);
                pstmt.setDouble(1, suma * -1);
                pstmt.setString(2, cuentaDebito);
                pstmt.executeUpdate();
                impresionInforme(cuentaDebito, suma * -1);
                pstmt.setDouble(1, suma);
                pstmt.setString(2, cuentaCredito);
                pstmt.executeUpdate();
                impresionInforme (cuentaCredito, suma);
                //Validación de los tratamientos
                System.out.println("Validación
de la transacción");

                cnx.commit();
            } catch (Exception e) {
                try {

                    //Anulación de las operaciones
```

```

        System.out.println("Anulación
de la transacción");

        cnx.rollback();
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
    e.printStackTrace();
}
} catch (SQLException e2) {
    e2.printStackTrace();
}
}

private static void impresionInforme (String cuenta, double suma)
throws Exception {
    System.out.println(cuenta + " : " + suma);
    //Descomente la línea siguiente para provocar la anulación
de
las operaciones
    //throw new Exception("Error en la edición del informe");
}

private static Connection getConnection() throws SQLException {
    String url =
"jdbc:mysql://localhost/demo_java?serverTimezone=UTC";
    String user = "mysqluser";
    String password = "Pa$$w0rd";
    return DriverManager.getConnection(url, user, password);
}
}

```

Ejecute este código comentando o descomentando la segunda declaración del método `imprimirInforme`. Luego observe los datos en la tabla para comprender el mecanismo de las transacciones. En caso de una excepción como esta:

```
Console
<terminated> TestTransaction[Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (5 July 2020 18:42:15)
123456789: -1000.0
Anulación de la transacción
Java.sql.SQLException: Error de edición del informe
    at Proyecto_JavaSE_Cap6/es.eni.editions.jdbc.consulta.TestTransaction.impressionInforme (TestTransaction.java:48)
    at Proyecto_JavaSE_Cap6/es.eni.editions.jdbc.consulta.TestTransaction.movimiento (TestTransaction.java:21)
    at Proyecto_JavaSE_Cap6/es.eni.editions.jdbc.consulta.TestTransaction.main (TestTransaction.java:11)
```

Las instrucciones SQL se anulan y el contenido de la base de datos no se modifica.

b. Puntos de salvaguarda

En el momento de la llamada al método `rollback`, se cancela el conjunto de instrucciones SQL ejecutadas desde el principio de la transacción. Este método propone una segunda versión que recibe como parámetro un objeto `SavePoint`. Este objeto representa una referencia en la ejecución de las instrucciones SQL. Lo crea el método `setSavePoint`. La llamada al método `rollback` pasando como argumento un objeto `SavePoint` provoca la cancelación de todas las instrucciones ejecutadas hasta este punto de salvaguarda.

c. Niveles de aislamiento

Mientras una transacción está activa, los datos modificados por las instrucciones ejecutadas dentro de la transacción pueden bloquearse en la base de datos para evitar conflictos e incoherencias. Las bases de datos cuentan con diferentes tipos de bloqueos para una transacción. Para entender sus efectos de forma correcta, hay que identificar previamente los tipos de problemas que podemos encontrar cuando una transacción está activa.

- **Lectura errónea:** se produce esta anomalía cuando una aplicación accede a datos que están siendo modificados por una transacción que aún no ha sido validada.
- **Lectura no reproducible:** ocurre esta anomalía cuando las ejecuciones sucesivas de una misma instrucción `select` no producen el mismo resultado. Este caso se da cuando los datos que está leyendo están siendo modificados por otra transacción.
- **Lectura fantasma:** ocurre esta anomalía si ejecuciones sucesivas de una misma petición devuelven datos de más o de menos. Puede darse el caso si otra transacción está suprimiendo o añadiendo datos a la tabla.

La API JDBC prevé varios niveles de aislamiento, que determinan la manera en que se bloquean los datos durante la transacción. Este bloqueo puede ser de lectura, de escritura, o bien de lectura y escritura sobre los datos a los que se accede mediante las instrucciones de

la transacción.

El método `setTransactionIsolationLevel` permite definir los tipos de problemas que se pueden evitar. Este método recibe como parámetro alguna de las constantes presentadas en el cuadro siguiente. Este cuadro presenta también el efecto de cada nivel de aislamiento en los datos.

constante	Lectura errónea	Lectura no reproducible	Lectura fantasma
TRANSACTION_READ_UNCOMMITTED	posible	posible	posible
TRANSACTION_READ_COMMITTED	imposible	posible	posible
TRANSACTION_REPEATABLE_READ	imposible	imposible	posible
TRANSACTION_SERIALIZABLE	imposible	imposible	imposible

El hecho de especificar un nivel de aislamiento para una transacción puede tener efectos sobre otras aplicaciones que accedan a los datos manipulados durante la transacción, ya que estos pueden estar bloqueados y, por lo tanto, ser inaccesibles para las demás aplicaciones.

Archivos de Java

1. Presentación

Un archivo de Java es un formato de archivo particular que permite agrupar en un único archivo a varios. En general, se agrupan de esta manera todos los archivos necesarios para el funcionamiento de una aplicación. Esto comprende por supuesto todos los archivos `.class`, pero también todos los demás recursos indispensables para el correcto funcionamiento de la aplicación. Esta posibilidad de agrupación proporciona numerosas ventajas para el despliegue de aplicaciones.

- ❑ La primera y seguramente más notable se encuentra en el hecho de que, para desplegar una aplicación en otro puesto cliente, solo se necesita copiar un único archivo, incluso si la aplicación exige para su funcionamiento varios recursos organizados en forma de árbol de manera precisa. Se crea este árbol en el interior del archivo y no necesita reproducirse en el puesto cliente.
- Los archivos pueden comprimirse para optimizar su almacenamiento y su intercambio a través de una red.
- La aplicación puede incorporar todos los recursos necesarios y un entorno minimalista de ejecución de Java necesario para la ejecución de la aplicación sin tener que instalar el JRE.

2. Manipulación de un archivo

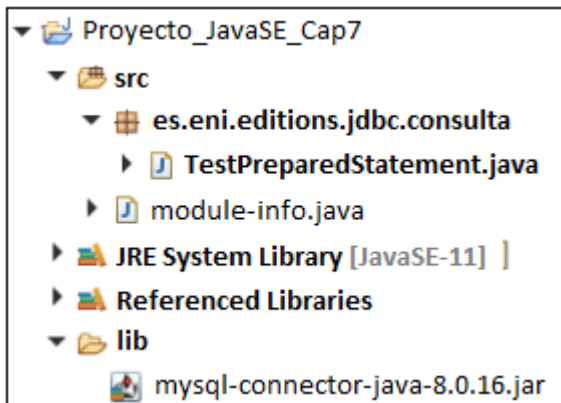
La manipulación de un archivo de Java (archivo `jar` - *Java ARchive*) se basa en los mismos principios que la manipulación de un archivo en el mundo Unix con el comando `tar`. Las opciones del comando `jar` que permiten manipular un archivo Java son, por

otra parte, extrañamente similares a las del comando `tar` de Unix. El formato utilizado internamente por los archivos es también muy conocido, ya que se trata del formato ZIP. Paralelamente, un archivo de Java puede ser procesado por herramientas destinadas a la manipulación de archivos ZIP.

El comando estándar de manipulación de archivo es el comando `jar`. Forma parte de las herramientas proporcionadas con el JDK que se puede encontrar en el directorio `bin` de la instalación de JDK.

a. Creación de una aplicación

Antes de crear un archivo, primero debe crear una aplicación. Los ejemplos en el capítulo se basarán en una aplicación de demostración que hace consultas a una base de datos. La estructura del proyecto es la siguiente:



La aplicación consta de una sola clase `TestPreparedStatement` que contiene un método `main`, que será el punto de entrada de la aplicación. Esta aplicación tiene un archivo `module-info.java` para indicar el uso del módulo `java.sql`. Al final, esta aplicación hace referencia al controlador JDBC que permite el acceso a MySQL.

El resultado de la compilación de este programa está disponible en el directorio `bin` del proyecto. La ejecución del comando `ls -lR` en un entorno Linux le permite ver el contenido de un directorio de forma recursiva. Este es el resultado para el directorio `bin` del proyecto:

```

java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ ls -lR
.:
total 8
drwxrwxr-x 3 java java 4096 julio 07 15 :12 es
-rw-rw-r-- 1 java java 181 julio 07 15 :12 module-info.class

./es:
total 4
drwxrwxr-x 3 java java 4096 julio 07 15 :12 eni

./es/eni:
total 4
drwxrwxr-x 3 java java 4096 julio 07 15 :12 editions

./es/eni/editions:
total 4
drwxrwxr-x 3 java java 4096 julio 07 15 :12 jdbc

./es/eni/editions/jdbc:
total 4
drwxrwxr-x 2 java java 4096 julio 07 15 :12 consulta

./es/eni/editions/jdbc/consulta:
total 4
-rw-rw-r-- 1 java java 3007 julio 07 15 :12 TestPreparedStatement.class

```

Puede ver la presencia de dos archivos `.class` (`module-info.class` en la raíz del directorio `bin` y `TestPreparedStatement.class` en el subdirectorio `es/eni/editions/jdbc/consulta`).

El transporte de esta estructura para el despliegue puede ser complejo. El uso de un archivo parece perfectamente adecuado para simplificar este trabajo.

b. Creación de un archivo

La sintaxis básica de creación de un archivo de Java es la siguiente:

```
jar cf nombreDelArchivoJAR listaDeArchivosAIncorporar
```

El parámetro `c` se destina por supuesto a indicar al comando `jar` que deseamos crear un archivo (`create`). El parámetro `f`, por su parte, indica que el comando debe generar un archivo (`file`). El nombre del archivo se indica por el tercer parámetro de este comando. Por convención, la extensión de este archivo será `.jar`. El último elemento representa el archivo o los archivos que se deben incorporar al archivo. Si varios archivos se deben incluir en el archivo, sus nombres se deben separar por un espacio. Se acepta el uso del comodín `*` en la lista. Si a su vez hay un nombre de carpeta en la lista, se añade su

contenido entero al archivo.

El archivo se genera en el directorio actual. Se incluye también, por defecto, un archivo *manifest* (llamado `MANIFEST.MF`) con un contenido dentro del archivo en un subdirectorio `META-INF`. Una sección se dedica a este archivo un poco más adelante en este capítulo.

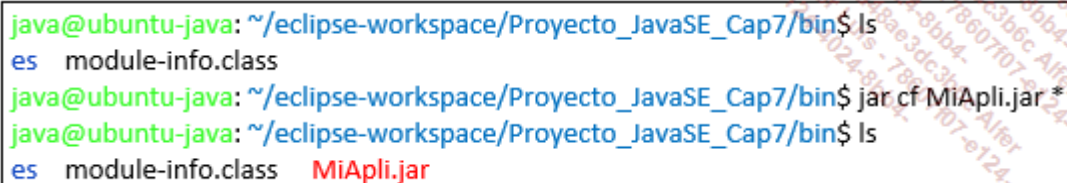
Las opciones siguientes están disponibles en el comando `jar`:

- `v` muestra el nombre de los archivos al añadirlos en el archivo.
- `0` desactiva la compresión del archivo.
- `M` desactiva la generación del manifest.
- `m` añade el manifest indicado al archivo.
- `C` suprime el nombre de directorio en el archivo.

En el marco de este ejemplo, el comando que se ha de ejecutar es el siguiente si nos situamos en el directorio `bin`:

```
jar cf MiApli.jar *
```

Esto es lo que se observa durante la ejecución:



```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ ls
es module-info.class
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jar cf MiApli.jar *
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ ls
es module-info.class  MiApli.jar
```

Después de la ejecución del comando, se crea un nuevo archivo (`MiApli.jar` en el ejemplo).

c. Visualización del contenido

El contenido de un archivo se puede visualizar con el comando siguiente:

```
jar tf nombreDelArchivoJAR.jar
```

El comando muestra en la consola el contenido del archivo. En el marco de este ejemplo, el resultado es el siguiente:

```

java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jar tf MiApli.jar
META-INF/
META-INF/MANIFEST.MF
module-info.class
es
es/eni
es/eni/editions
es/eni/editions/consulta
es/eni/editions/consulta/TestPreparedStatement.class

```

Se puede obtener información extra al añadir la opción `tfv` al comando. Datos como la fecha de modificación y el tamaño de los archivos se incorporan al resultado del comando.

```

java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jar tfv MiApli.jar
0   Tue  July  7  15:54:07  CEST  2019  META-INF/
66  Tue  July  7  15:54:07  CEST  2019  META-INF/MANIFEST.MF
242 Tue  July  7  15:54:07  CEST  2019  module-info.class
0   Tue  July  7  15:54:07  CEST  2019  es
0   Tue  July  7  15:54:07  CEST  2019  es/eni
0   Tue  July  7  15:54:07  CEST  2019  es/eni/editions
0   Tue  July  7  15:54:07  CEST  2019  es/eni/editions/consulta
3007 Tue  July  7  15:54:07  CEST  2019  es/eni/editions/consulta/TestPreparedStatement.class

```

Las rutas de acceso a los archivos se visualizan con el carácter `/` como separador y son relativas a la raíz del archivo. Por supuesto, el contenido del archivo no se modifica con la ejecución de este comando.

d. Extracción

Es posible extraer archivos mediante el comando siguiente:

```
jar xvf nombreDelArchivoJAR
```

Se crean de nuevo los archivos en la carpeta actual del disco. Si el archivo contiene un árbol de carpetas, se reproduce en la carpeta actual. Los eventuales archivos y carpetas existentes se sobrescriben por los presentes en el archivo. La extracción de un archivo puede ser selectiva si se indica como parámetro adicional la lista de los archivos que se quiere extraer del archivo separando los nombres de estos archivos por un espacio. El comando siguiente permite extraer únicamente el archivo

`TestPreparedStatement.class`.

```

jar xvf MiApli.jar
es/eni/editions/jdbc/consulta/TestPreparedStatement.class

```

El contenido del archivo no se modifica con este comando. Este es el resultado de la ejecución de este comando:

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jar xvf MiApli.jar
es/eni/editions/jdbc/consulta/TestPreparedStatement.class
descomprimido: es/eni/editions/jdbc/consulta/TestPreparedStatement.class
```

e. Actualización

El contenido de un archivo se puede actualizar añadiendo nuevos archivos después de su creación. En este caso, se debe utilizar el comando siguiente:

```
jar uf nombreDelArchivoJAR nombreDeLosArchivosAModificarOAgregar
```

El último parámetro de este comando representa la lista de los archivos que se deben actualizar en el archivo. Si estos archivos no existen en el archivo, se añaden, y si ya existen, entonces se sustituyen por la nueva versión. Si el archivo contiene carpetas, debe especificarse la ruta completa en la lista de los archivos que se desea añadir.

f. Ejecución

Una aplicación presente en un archivo de Java puede ejecutarse directamente desde el archivo sin necesitar extraer su contenido. Debemos indicar a la máquina virtual de Java que debe extraer ella misma el contenido del archivo utilizando la opción `-jar` durante el lanzamiento de la aplicación. A continuación, debe especificarse el nombre del archivo.

```
java -jar nombreDelArchivoJAR.jar
```

Sin embargo la máquina virtual de Java necesita información adicional para determinar el archivo que contiene el método `main` por el cual debe empezar la ejecución de la aplicación. Para ello, busca el `MANIFEST.MF` del archivo, que debe contener, efectivamente, esta información para que la aplicación se pueda ejecutar a partir del archivo.

A los fines del ejemplo, no se especificó el archivo que contiene el método principal. Si se ejecuta el método `java -jar MiApli.jar`, el resultado es el siguiente:

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ java -jar MiApli.jar
No hay ningún atributo manifest pricipal en MiApli.jar
```

El mensaje indica que no se encontró información útil en el archivo `MANIFEST.MF`.

3. El manifest

Los archivos de Java son mucho más que meros archivos comprimidos, ya que proporcionan una multitud de funcionalidades complementarias:

- Ejecución directa desde el archivo.
- Gestión de las versiones.
- ...

Todas estas funcionalidades están disponibles mediante el archivo `MANIFEST.MF` (llamado a partir de ahora `manifest`) incluido en el archivo.

a. Presentación

El archivo `manifest` es un mero archivo de texto que contiene parejas de parámetros formadas por el nombre de parámetro y el valor de parámetro. Estos dos datos se separan por el carácter «:».

Este archivo siempre se llama `MANIFEST.MF` y se encuentra en la carpeta `META-INF` del archivo.

b. Creación

Al crear un archivo `jar`, se crea un archivo `manifest` por defecto. Contiene la información siguiente:

```
Manifest-Version: 1.0
Created-By: 11.0.3-ea (Oracle Corporation)
```

La primera línea indica la versión del archivo `manifest`, la segunda indica la versión del `JDK` con la cual se generó el archivo.

Para añadir más información al archivo `manifest`, debemos proceder en dos etapas. Primero,

debemos preparar un archivo de texto que contenga la información que deseamos incluir en el manifest del archivo. La última línea de este archivo debe terminar obligatoriamente con un carácter de retorno de carro o salto de línea (o ambos). A continuación, debemos fusionar este archivo de texto con el manifest por defecto del archivo utilizando la opción `m` del comando `jar`. Por lo tanto, la sintaxis del comando es la siguiente:

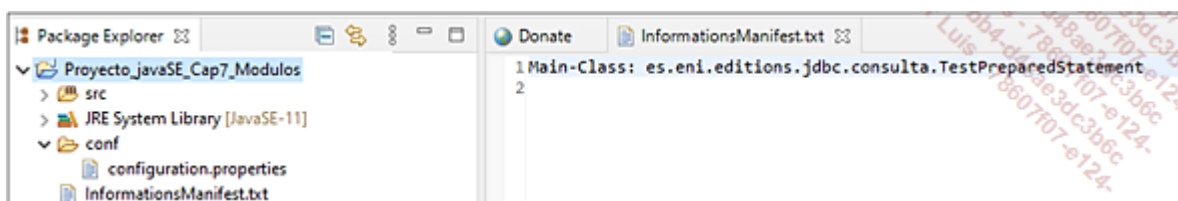
```
jar cfm nombreDelArchivoJAR InformationsManifest.txt  
listaDeArchivosAIncorporar
```

En el marco del ejemplo, el comando puede ser el siguiente:

```
jar cfm MiApli.jar ../InformationsManifest.txt *
```

Este comando genera un archivo llamado `MiApli.jar` que contiene los elementos de la carpeta actual y añade al manifest por defecto los datos contenidos en el archivo `InformationsManifest.txt`. Este archivo puede contener por ejemplo la información necesaria para definir el nombre de la clase que contiene el método `main` por el cual se debe empezar la ejecución de la aplicación. En este caso, el argumento que hay que utilizar se llama `Main-Class`.

Por lo tanto, el archivo `InformationsManifest.txt` debe existir en la estructura del proyecto. En el ejemplo, se coloca en el directorio raíz del proyecto:



No olvide el retorno de carro al final de la línea y el espacio después del carácter «:». En el caso contrario, el archivo no se generará correctamente.

Ahora, el contenido del archivo manifest es siguiente:

```
Manifest-Version: 1.0  
Main-Class: es.eni.editions.jdbc.consulta.TestPreparedStatement  
Created-By: 11.0.3 (Sun Microsystems Inc.)
```

La ejecución de este archivo modificado de esta manera da el siguiente resultado:

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ java -jar MiApli.jar
java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost:8100/demo_java?serverTimeZone=UTC
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:702)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:228)
    at es.eni.editions.jdbc.consulta.TestPreparedStatement.getConnection(TestPreparedStatement.java:46)
    at es.eni.editions.jdbc.consulta.TestPreparedStatement.main(TestPreparedStatement.java:15)
```

La clase que contiene el método `main` se ejecuta con éxito; sin embargo, se produce un error durante la ejecución. De hecho, el código de este programa requiere el uso del controlador JDBC para MySQL. Actualmente, no está disponible en el entorno de ejecución, por lo que se produce este error.

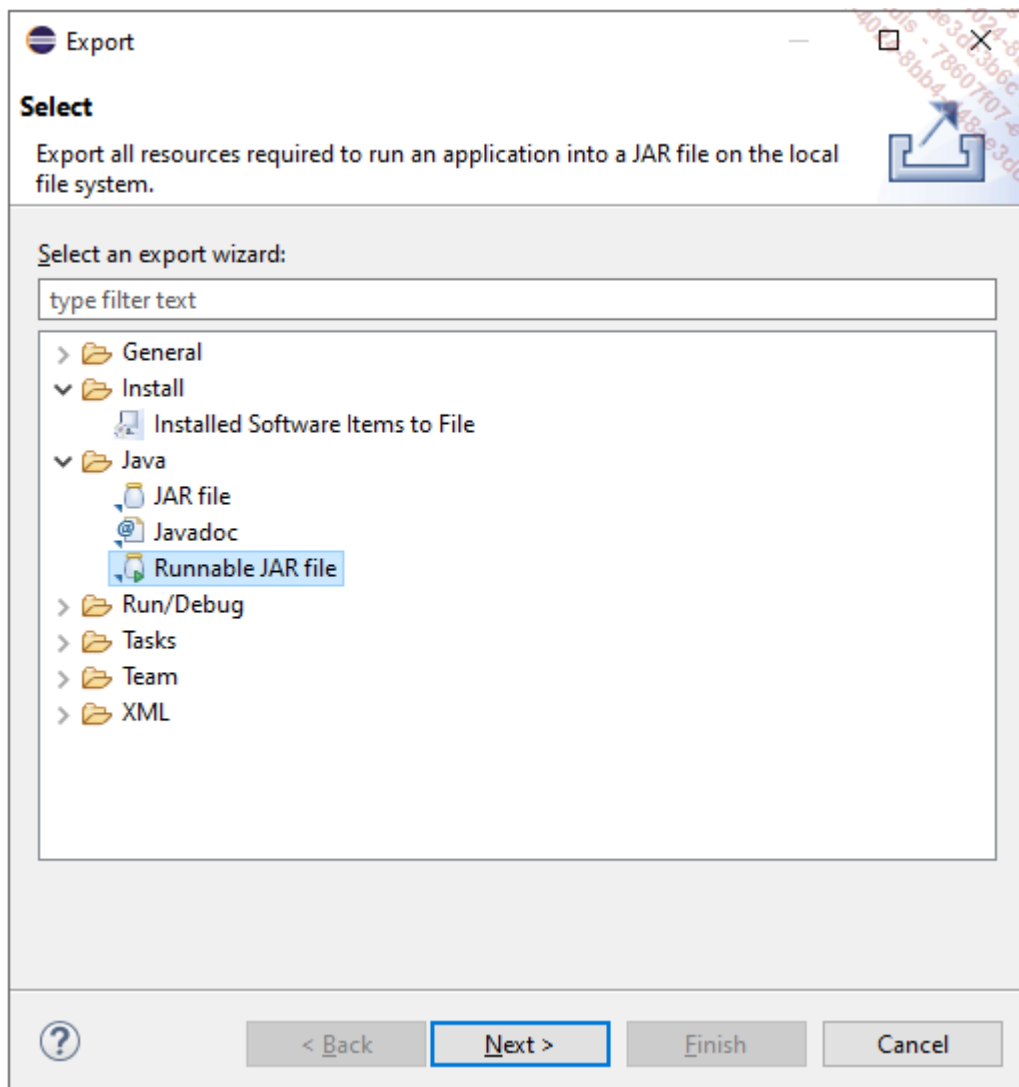
4. La gestión de las dependencias

a. En un subdirectorio

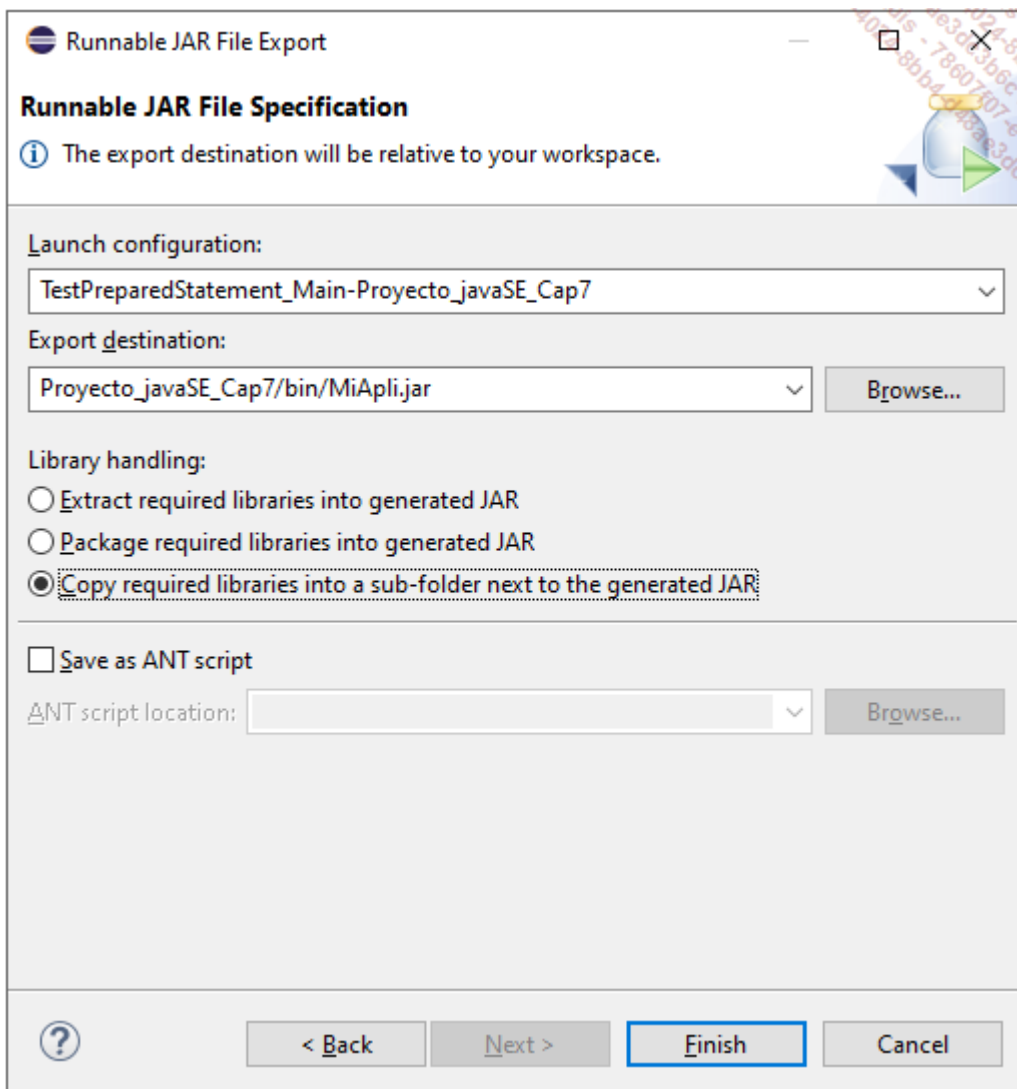
Es posible simplemente copiar el directorio `lib` del proyecto junto al archivo de la aplicación. También es posible pedir a Eclipse que ofrezca facilidades para generar el JAR.

Para hacerlo, siga estas instrucciones:

- Haga clic derecho en el proyecto y haga clic en el menú **Export...**



- Seleccione **Java - Runnable JAR file** y pulse en el botón **Next**.



- Elija la configuración (**Launch configuration**) que se utilizará para definir el punto de entrada del programa. Esto le permite definir el parámetro `Main-Class` del manifest. Defina el nombre del archivo (**Export destination**). Seleccione el modo de gestión de las dependencias (**Library handling**). En esta sección, elija la tercera opción. Cuando todo esté listo, haga clic en el botón **Finish**.
- Observe el directorio de salida. Debe contener el archivo de almacenamiento y un directorio que incluya las dependencias. A través de Eclipse, el nombre de este directorio tiene la siguiente forma: {Nombre del jar sin la extensión} _lib (**MiApli_lib** en el contexto del ejemplo).
- Visualice el contenido del manifest:

Manifest-Version: 1.0

Main-Class: es.eni.editions.jdbc.consulta.TestPreparedStatement

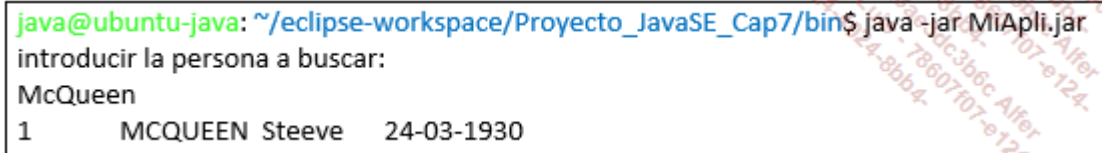
Class-Path: . MiApli_lib / mysql-connector-java-8.0.16.jar

Un nuevo atributo está disponible: `Class-Path`. Permite hacer referencia «automáticamente» al controlador JDBC, útil para la correcta ejecución de la aplicación.

- Intente ejecutar la aplicación con el siguiente comando:

```
java -jar MiAppli.jar
```

El resultado es el siguiente:



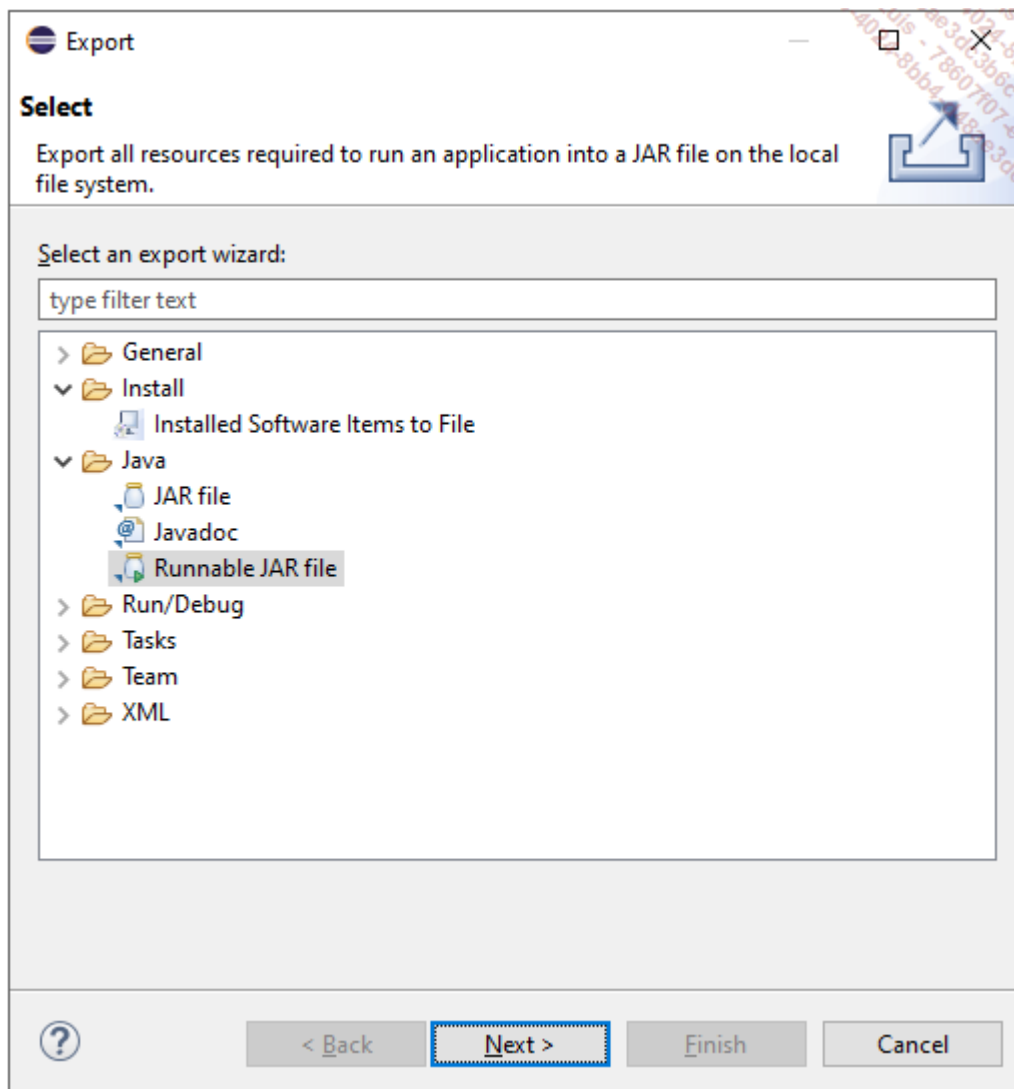
```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ java -jar MiAppli.jar
introducir la persona a buscar:
McQueen
1      MCQUEEN Steeve    24-03-1930
```

La ventaja de esta solución es la facilidad de su implementación. El inconveniente potencial es su mayor dificultad (todo relativo en el ejemplo) para su despliegue. La siguiente sección presenta una solución a este pequeño problema.

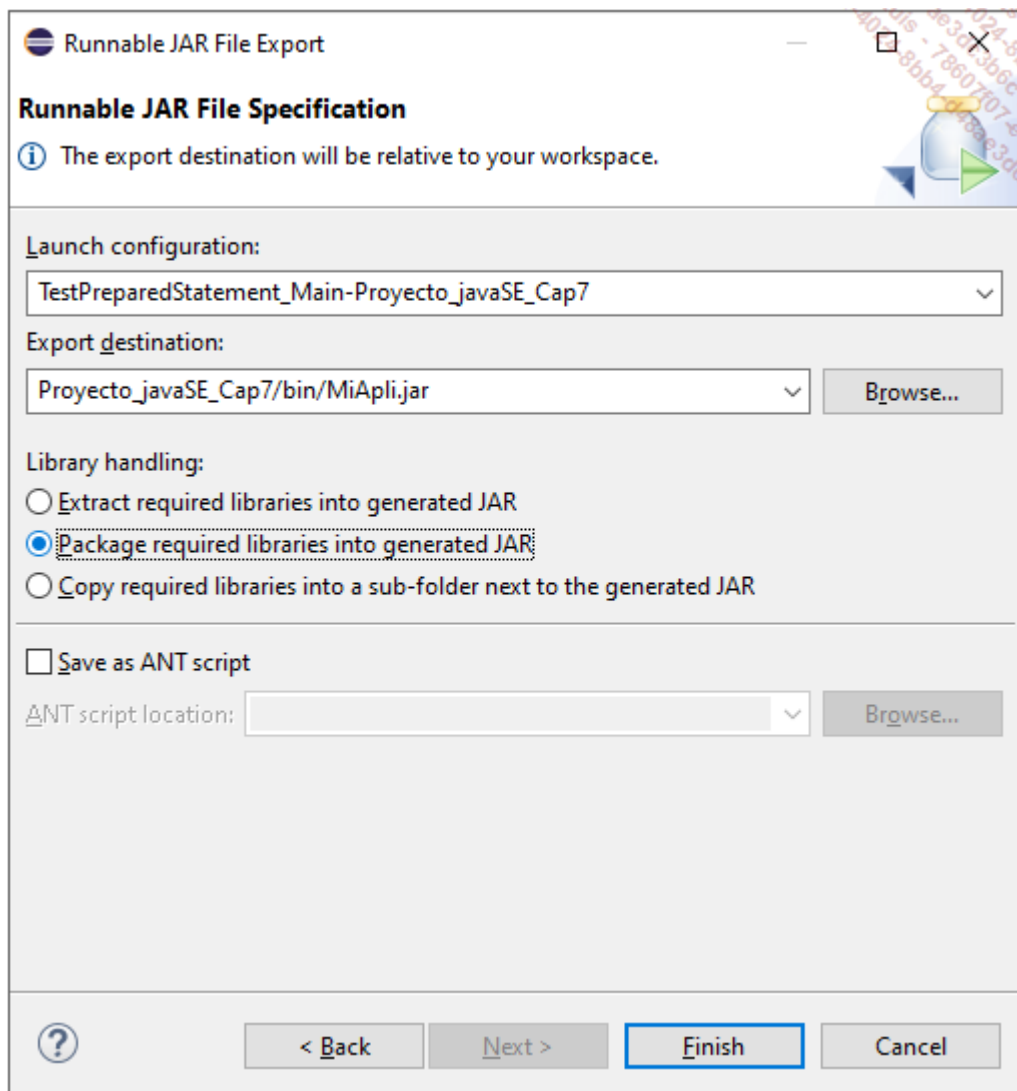
b. Dentro del propio archivo

Es posible integrar las dependencias dentro del propio archivo. Para esto, Eclipse ofrece facilidades para generar el JAR. Siga estos pasos:

- Haga clic derecho en el proyecto y haga clic en el menú **Export...**



- Seleccione **Java - Runnable JAR file** y pulse el botón **Next**.



- Elija la configuración (**Launch configuration**) que se utilizará para definir el punto de entrada del programa. Esto le permite definir el parámetro `Main-Class` del manifest. Defina el nombre del archivo (**Export destination**). Seleccione el modo de gestión de las dependencias (**Library handling**). En esta sección, elija la segunda opción. Cuando todo esté listo, haga clic en el botón **Finish**.
- Observe el directorio de salida. Debe contener el archivo con un tamaño mayor porque también contiene dependencias. He aquí el contenido de este archivo:

Nombre	Tamaño	Comprimido	Tipo	Modificado	CRC32
..			Carpeta de archivos		
META-INF	157	116	Carpeta de archivos		
org	11.510	5.776	Carpeta de archivos	07/07/2020 15:46	
es	154	114	Carpeta de archivos	07/07/2020 15:46	
module-info.class	1.273	537	Archivo CLASS	03/07/2020 18:31	786FD739
mysql-connector-java-8.0.16.jar	656	398	Archivo CLASS	05/07/2020 14:08	D78D5AC

En particular, puede encontrar el controlador JDBC.

- Intente ejecutar la aplicación con el siguiente comando:

```
java -jar MiApli.jar
```

- El resultado es el mismo que en la sección anterior.

Esta solución tiene la ventaja de generar un solo archivo, lo que facilita la implementación. La única restricción ahora sigue siendo la instalación previa de la JVM en las máquinas en las que se ejecutará la aplicación.

Creación de una aplicación autónoma con jlink

Desde Java 9, la API de Java es modular. Una de las consecuencias es que, en tiempo de ejecución, solo es necesario cargar los módulos de los que depende una aplicación para que esta funcione. Oracle fue aún más lejos al hacer posible crear un árbol de directorios y archivos que contengan estos módulos, la herramienta `java` y el archivo de una aplicación. Este directorio se puede comprimir o integrar fácilmente en un instalador para implementarlo en las máquinas de destino.

Para crear el directorio y el árbol de archivos, se requiere la herramienta `jlink`. También se utilizan herramientas adicionales: `jdeps` y `jmod`.

- `jmod` le permite crear el módulo correspondiente a la aplicación desde un archivo.
- `jdeps` le permite determinar los módulos de los que depende la aplicación.
- `jlink` le permite crear el directorio y la estructura de árbol de archivos.

Crear una aplicación autónoma requiere varios pasos:

- La creación de un archivo de Java correspondiente al código de la aplicación, sin definir un punto de entrada (sin atributo `Main-Class` en el archivo manifest).
- La creación de un archivo de módulo (extensión `.jmod`) desde el archivo de Java.
- La búsqueda de dependencias (módulos), necesarias para la correcta ejecución de la aplicación con la herramienta `jdeps`.
- Creación de la estructura de árbol de directorios y archivos a bordo de la aplicación y los módulos necesarios.
- Adición manual de dependencias que no son módulos de Java. Este es el caso del controlador JDBC, en nuestro ejemplo.

1. La creación del archivo de Java

La creación del archivo de Java sin definir un punto de entrada se realiza con el siguiente comando:

```
jar -cf filename JAR listaDeArchivosAIncorporar
```

Por ejemplo, aquí está la secuencia de instrucciones que hay que seguir para crear el archivo de Java en un subdirectorio `build`, suponiendo que esté ubicado en el directorio `bin` de su proyecto.

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ ls
es      module-info.class
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ mkdir build
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jar -cf build/MiAppli.jar *
```

Este archivo no es ejecutable directamente porque el archivo manifest no contiene atributos `Main-Class`.

2. La creación del módulo de la aplicación

La creación de un módulo desde el archivo requiere el uso de la herramienta `jmod`. La instrucción puede parecer compleja porque la herramienta tiene varias opciones. En el contexto del ejemplo, la instrucción ejecutada es la siguiente:

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jmod create
--class-path build/MiAppli.jar --main-class es.eni.editions.jdbc.consulta.TestPreparedStatement
build/MiAppli.jmod
```

La primera opción para este comando es `create`. Le permite pedirle a la herramienta que cree un módulo. El nombre y la ubicación de este módulo están determinados por el último parámetro (`build/MiAppli.jmod`).

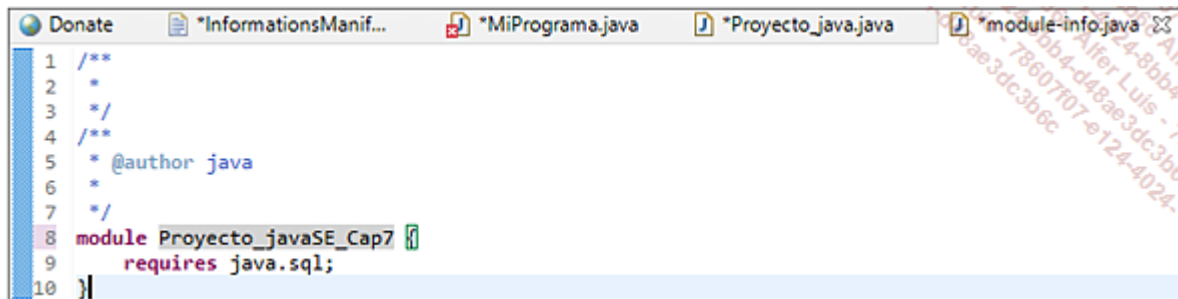
La segunda opción `--class-path` permite determinar el jar que se incorporará en el módulo (`build/MyAppli.jar`).

La tercera opción `--main-class` permite determinar el punto de entrada del módulo (la clase que contiene el método `main`).

3. La búsqueda de las dependencias con `jdeps`

La herramienta `jdeps` le permite descubrir todos los módulos necesarios para el correcto funcionamiento de la aplicación.

Como parte del ejemplo, la aplicación requiere los módulos `java.base` (esencial para cualquier aplicación) y `java.sql`, tal como se define en el archivo `module-info.java`.



De hecho, la aplicación permite el acceso a una base de datos.

La aplicación también requiere un controlador JDBC (`mysql-connector-java-8.0.16.jar`) que, en este caso, no es un módulo. Este controlador tiene él mismo dependencias. Es necesario enumerarlas para saber cuáles agregar, de modo que la aplicación sea autónoma.

Por lo tanto, la herramienta `jdeps` permite enumerar los módulos de los que depende directamente la aplicación, pero también los módulos de los que dependen las dependencias de la aplicación.

El comando para encontrar todas estas dependencias es el siguiente:

```
jdeps --list-deps listaDeJarsSeparadosPorUnEspacio
```

La opción `--list-deps` le permite enumerar los archivos para los que estamos buscando dependencias.

En el contexto de este ejemplo, el comando es el siguiente, suponiendo que el controlador

JDBC se encuentre en un directorio `lib`, en la raíz del proyecto:

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ jdeps
--list-deps build/MiApli.jar ..lib/mysql-connector-java-8.0.16.jar
  java.base
  java.logging
  java.management
  java.naming
  java.security.sasl
  java.sql
  java.transaction.xa
  java.xml
```

Este comando determina la lista de módulos necesarios. Hay ocho de ellos en el ejemplo.

4. La creación de la estructura de árbol con jlink

Ahora que se ha creado el módulo de aplicación y se conoce la lista de módulos necesarios para su ejecución, es posible crear una aplicación independiente utilizando la herramienta `jlink`.

El comando básico se ve así:

```
jlink --output directorioDeSalida
--add-modules ListaDeModulosSeparadosPorUnaComa
--module-path directoriosOModulosAExplorar
--launcher ejecutableName = moduleExecute
```

La primera opción, `--output`, determina el directorio de salida del comando.

La segunda opción, `--add-modules`, le permite enumerar los módulos para agregar.

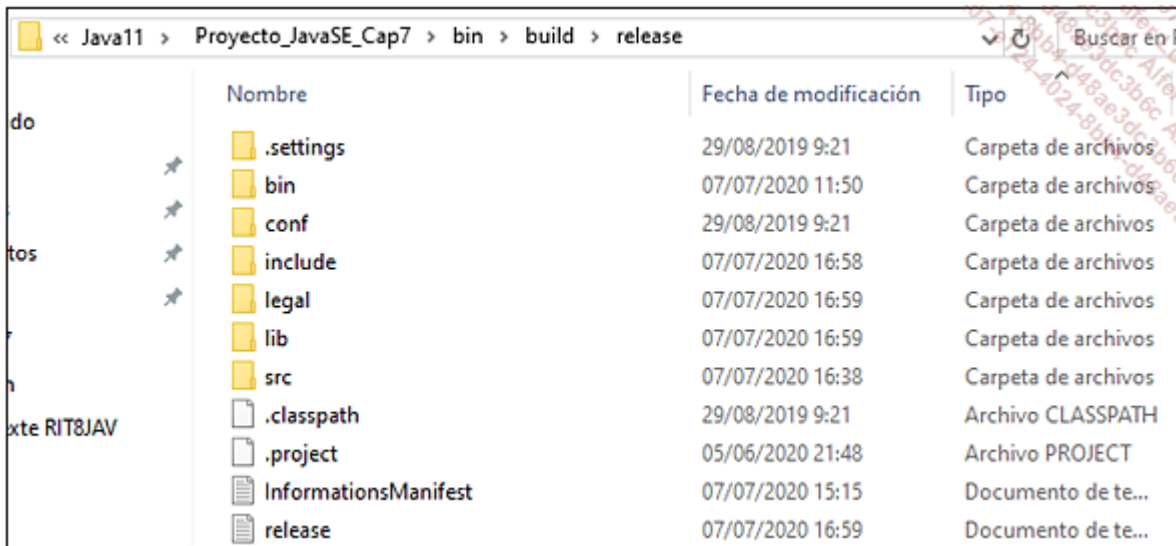
La tercera opción, `--module-path`, le permite indicar la ubicación de los módulos que no forman parte del JDK. Es posible indicar el nombre de directorios o módulos.

La última opción, `--launcher`, permite indicar el nombre del lanzador y el módulo que debe ejecutar. Este módulo debe tener un punto de entrada correspondiente a la clase que contiene el método principal que se va a ejecutar.

En el contexto del ejemplo, el comando es el siguiente:

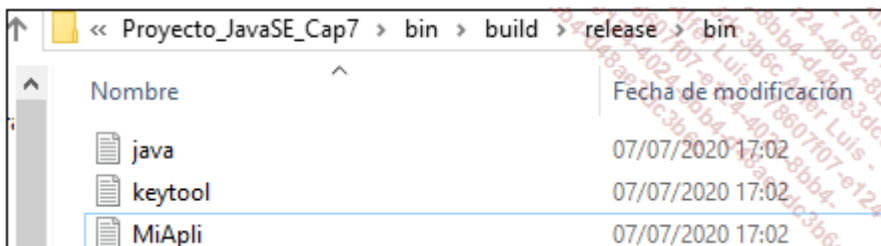
```
java@ubuntu-java:~/eclipse-workspace/ProyectoJavaSE_Cap7/bin$ jlink
--output build/release --add-modules java.logging, java.management, java
.naming, java.security.sasl, java.sql, java.transaction.xa, java.xml, Proyecto
JavaSE_Cap7 --module-path build/MiAplicacion.jmod --launcher MiAplicaci
on=ProyectoJavaSE_Cap7
```

Por lo tanto, se crea una estructura de árbol en el subdirectorio `build/release`.



Nombre	Fecha de modificación	Tipo
.settings	29/08/2019 9:21	Carpeta de archivos
bin	07/07/2020 11:50	Carpeta de archivos
conf	29/08/2019 9:21	Carpeta de archivos
include	07/07/2020 16:58	Carpeta de archivos
legal	07/07/2020 16:59	Carpeta de archivos
lib	07/07/2020 16:59	Carpeta de archivos
src	07/07/2020 16:38	Carpeta de archivos
.classpath	29/08/2019 9:21	Archivo CLASSPATH
.project	05/06/2020 21:48	Archivo PROJECT
InformationsManifest	07/07/2020 15:15	Documento de te...
release	07/07/2020 16:59	Documento de te...

Los subdirectorios más importantes son los directorios `bin` y `lib`. El directorio `lib` contiene los módulos necesarios para la correcta ejecución de la aplicación. El directorio `bin` contiene la herramienta `java` y la herramienta especialmente creada por el comando `jlink` para ejecutar la aplicación. En el contexto del ejemplo, se llama `MiAppli`:



Nombre	Fecha de modificación
java	07/07/2020 17:02
keytool	07/07/2020 17:02
MiAppli	07/07/2020 17:02

El archivo `MiAppli` es simplemente un archivo de comando:

```
#!/bin/sh
JLINK_VM_OPTIONS=
DIR=`dirname $0`
$DIR/java $JLINK_VM_OPTIONS -m ProyectoJavaSE_Cap7/
es.eni.editions.jdbc.consulta.TestPreparedStatement $0
```

Le permite ejecutar la aplicación utilizando la herramienta `java` y proporcionando como parámetro la clase del módulo de aplicación que contiene el método `main`.

La aplicación está potencialmente lista si no requiere dependencias de bibliotecas no

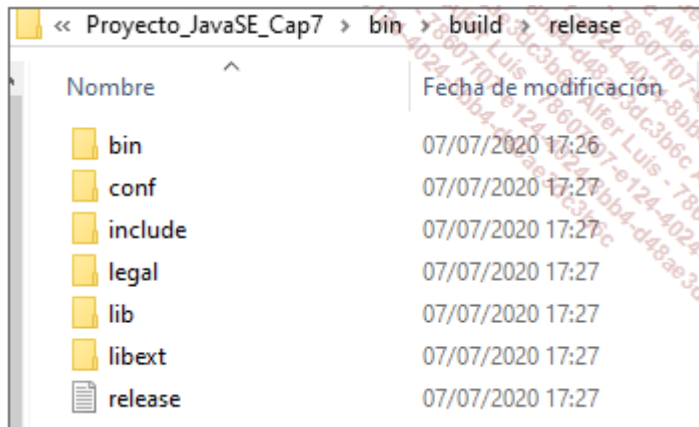
modulares. Sin embargo, en el contexto de este ejemplo, este no es el caso. Al intentar ejecutar el programa, se encontrará con el siguiente problema:

```
java@ubuntu-java: ~/eclipse-workspace/Proyecto_JavaSE_Cap7/bin$ ./MiApli
java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost:8100/demo_java?serverTimezone=UTC
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:702)
    at java.sql/java.sql.DriverManager.getConnection(DriverManager.java:228)
    at Proyecto_JavaSE_Cap7/es.eni.editions.jdbc.consulta.TestPreparedStatement.getConnexion(TestPreparedStatement.java:46)
    at Proyecto_JavaSE_Cap7/es.eni.editions.jdbc.consulta.TestPreparedStatement.main(TestPreparedStatement.java:15)
```

Por lo tanto, es necesario agregar el último tipo de dependencia. Este es el tema de la siguiente sección.

5. La adición de dependencias no modulares

Para agregar estas dependencias, simplemente cópielas en el árbol de la aplicación. Para una buena organización, es posible copiarlas en un directorio llamado `libext` (para librerías externas). La estructura en el subdirectorio `build/release` es ahora la siguiente:



El directorio `libext` contiene el archivo `mysql-connector-java-8.0.16.jar`.

Ahora es necesario modificar el archivo `bin/MiApli` para tener en cuenta esta nueva dependencia de cara a la correcta ejecución de la aplicación. Para esto, es posible utilizar la opción `-cp` (`--class-path`) de la herramienta `java`. Permite definir los directorios y archivos útiles para la correcta ejecución del programa.

En el contexto del ejemplo, el comando modificado se ve así:

```
JLINK_VM_OPTIONS=  
DIR=`dirname $0`  
$DIR/java $JLINK_VM_OPTIONS -cp ../libext/mysql-connector-java-8.0.16.jar -m  
Proyecto_javaSE_Cap7/es.eni.editions.jdbc.consulta.TestPreparedStatement $0
```

6. La prueba de la aplicación

Para probar la aplicación, es suficiente con ejecutar el comando `MiAppli` y asegurarse del correcto funcionamiento de la aplicación.

Externalización de los argumentos

La aplicación así empaquetada es operativa. El principal problema pendiente es la cadena de conexión a la base de datos, que se escribe en la clase `es.eni.editions.jdbc.consulta.TestPreparedStatement`. Es necesario externalizarlo para poder modificarlo según el contexto. Esto requiere modificar el programa antes de volver a empaquetarlo.

1. Creación del archivo de configuración

En Java, los archivos de configuración normalmente son los archivos `properties` o los archivos `xml`. Para los fines de este libro, se utilizó un archivo `properties`.

Un archivo de este tipo contiene pares clave/valor, escritos en una línea de la siguiente forma:

```
unaClave=unValor  
otraClave=otroValor
```

El separador entre la clave y el valor es el carácter `=`.

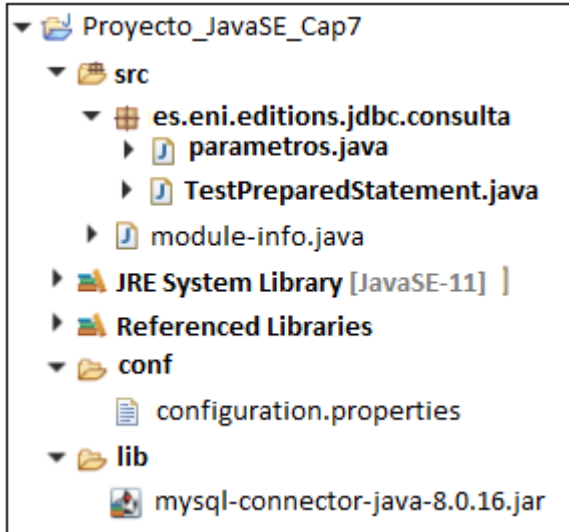
Para crear dicho archivo, siga este procedimiento:

- Cree un directorio `conf` (el nombre no tiene ninguna importancia) en la raíz del proyecto. No lo coloque en el directorio `src` porque no tiene el propósito de ser incorporado al archivo.
- Haga clic con el botón derecho en este directorio y seleccione el menú **New - File**.
- Introduzca un nombre para el archivo (**configuration.properties**, por ejemplo).

- Confirme haciendo clic en **Finish**.

Material para descargar

Al final de esta etapa, el proyecto debe tener el siguiente aspecto:



Observe la presencia del directorio `conf` al mismo nivel que los directorios `src` y `lib`. El archivo de configuración se inicializó con tres claves que permiten variar la URL de acceso a la base de datos, el usuario y la contraseña para conectarse a ella:

```
url=jdbc:mysql://localhost/demo_java?serverTimezone=UTC
user=mysqluser
password=Pa$$w0rd
```

Ahora es suficiente con explotar este archivo en el programa. Este es el tema de la siguiente sección.

2. Utilización del archivo de configuración

En el árbol del proyecto presentado en la sección anterior, puede observar la presencia de una nueva clase: `Parametros`. El propósito de esta clase es leer el archivo `configuration.properties` y proporcionar un método de utilidad para acceder a los diferentes valores asociados con las diferentes claves. El código para esta clase es el siguiente:

```
package es.eni.editions.jdbc.consulta;

import java.io.FileInputStream;
```

```

import java.util.Properties;

public abstract class Argumentos{
    //Objeto que permite almacenar en memoria el contenido del archivo
    properties
    private static Properties props = new Properties();

    //Carga del archivo durante la carga de la clase
    static {
        try(FileInputStream input = new
FileInputStream("./conf/configuration.properties")){
            props.load(input);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     *
     * @param clave
     * @return el valor asociado a la clave o un {@link
RuntimeException} en caso de ausencia de valor
     */
    public static String getValor(String clave) {
        String valor = props.getProperty(clave);
        if(valor!=null) {
            return valor;
        }
        throw new RuntimeException("El archivo
configuration.properties no está disponible");
    }
}

```

Cuando se carga esta clase, se ejecuta un bloque `static`. Permite cargar el archivo en memoria en forma de un objeto de tipo `Properties`. Este objeto permite almacenar un conjunto de pares clave/valor. El método `getValor` le permite leer el valor asociado con la clave pasada como parámetro.

Los cambios que se realizarán en la clase `TestPreparedStatement` están en el método `getConexion`:

```
private static Connection getConexion() throws SQLException {  
    String url = Arguments.getValor("url");  
    String user = Arguments.getValor("user");  
    String password = Arguments.getValor("password");  
    return DriverManager.getConnection(url, user, password);  
}
```

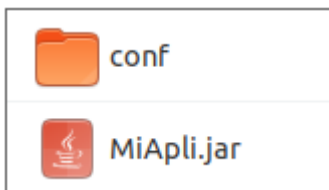
Las modificaciones están terminadas, la aplicación está operativa. Ahora queda empaquetarla nuevamente para integrar esta modificación.

3. Despliegue de la aplicación

La implementación se lleva a cabo como se describe en las secciones anteriores. Solo queda asociar el archivo de configuración.

Caso del archivo ejecutable

En el caso del archivo ejecutable, es suficiente con copiar el directorio `conf` (con el archivo `configuration.properties`) en el directorio que contiene el archivo `MiApli.jar`:



Caso de la aplicación autónoma

En el caso de una aplicación independiente, simplemente copie el directorio `conf` (con el archivo `configuration.properties`) en el directorio `bin` que contiene el iniciador `MiApli`:



conf



java



keytool



MiApli

Conclusión

Este capítulo le ha permitido descubrir algunos aspectos del despliegue de una aplicación escrita en Java. En un contexto profesional, con un proyecto más complejo, será necesario por supuesto profundizar en el tema. Sin embargo, estas pocas páginas lo ayudan a comenzar.

JAVA 11

Los fundamentos del lenguaje Java

Este libro se dirige a todos aquellos informáticos que quieran **desarrollar en Java**. Tanto si es principiante como si ya tiene experiencia con otro lenguaje, el lector encontrará en este libro **todos los fundamentos necesarios** para familiarizarse rápidamente con uno de los lenguajes más utilizados en el mundo.

Después de una presentación de la **plataforma Java SE**, su instalación en **Windows** y **Linux** y del entorno de desarrollo utilizado en el marco del libro (**Eclipse** y **MySQL**), el lector estudia los **fundamentos del lenguaje** y la **programación orientada a objetos**. Entre la riqueza del API de Java, se pone el foco en el API de **gestión de fechas**, en las nuevas posibilidades ofrecidas por las **interfaces**, así como sobre la noción de los **módulos**.

Las **expresiones lambda** y los API subyacentes como el API **java.util.function** y el **API Stream**, se detallan antes de pasar a estudiar el desarrollo de **aplicaciones gráficas** con la **biblioteca Swing**. También se presenta el desarrollo de aplicaciones cliente/servidor con el **API JDBC**, que garantiza el **acceso a las bases de datos relacionales**.

Siendo el despliegue una etapa importante en el éxito de una aplicación, el último capítulo presenta la **distribución** de una aplicación con la solución clásica del **archivo de ficheros** o el uso de la herramienta **jlink**.

Numerosos ejercicios con sus correcciones le permitirán validar sus conocimientos y poner en práctica, de inmediato, las nociones aprendidas.

Existen elementos complementarios para su descarga en el sitio www.ediciones-eni.com.

Después de 8 años como ingeniero en diferentes empresas de informática, **Thierry RICHARD** ahora es formador en la escuela de ENI. Se especializa en técnicas de desarrollo de objetos en entornos .NET y Java. Su experiencia se combina con sus habilidades de enseñanza, para proporcionar al lector un libro verdaderamente operativo, sobre el desarrollo de aplicaciones en Java.

Analista y desarrollador durante más de 10 años, **Thierry GROUSSARD** se orientó más adelante a la formación, en particular al campo del desarrollo de software. Sus conocimientos avanzados de las necesidades de la empresa y sus cualidades pedagógicas, hacen que este libro sea especialmente adaptado para el aprendizaje y puesta en práctica del desarrollo de aplicaciones en Java.



En www.ediciones-eni.com:

- Ejemplos de código fuente descritos en el libro.
- Correcciones de los ejercicios propuestos.

Para más información:



ISBN : 978-2-409-02740-6



9 782409 027406



www.ediciones-eni.com

<https://dogramcode.com/bloglibros>



Dogram