Recursosinformáticos

JAVA 8

Los fundamentos del lenguaje Java

(con ejercicios prácticos corregidos)

Thierry GROUSSARD

Presentación

1. Historia **11**

1.1 ¿ Por qué Java ? 11
1.2 Objetivos del diseño de Java 12
1.3 Auge de Java 13

2. Características de Java **14**

2.1 El lenguaje de programación Java 14
2.1.1 Sencillo 15
2.1.2 Orientado a objetos 16
2.1.3 Distribuido 16
2.1.4 Interpretado 17
2.1.5 Robusto 17
2.1.6 Securizado 17
2.1.7 Independiente de las arquitecturas 18
2.1.8 Portable 19
2.1.9 Eficaz 19
2.1.10 Multitarea 19
2.1.11 Dinámico 19
2.2 La plataforma Java 20
2.2.1 La máquina virtual Java (JVM) 21
2.2.2 La API Java 22
2.2.3 Las herramientas de despliegue de las aplicaciones 25
2.2.4 Las herramientas de ayuda al desarrollo 25
2.3 Ciclo de diseño de un programa Java 26

3. Instalación del SDK versión Win32 para el entorno Windows **27**

3.1 Descarga 27
3.2 Instalación 28
3.3 Configuración 29
3.4 Prueba de la configuración del SDK 30
3.5 Instalación de la documentación del SDK y de las API estándar 31

4. Las diferentes etapas de creación de un programa Java **33**

4.1 Creación de los archivos fuente 33
4.2 Compilar un archivo fuente 34
4.3 Ejecutar una aplicación 36

5. Nuestra primera aplicación Java **37**

5.1 Esqueleto de una aplicación 37
5.2 Argumentos en línea de comando 39
5.2.1 Principios y utilización 39
5.2.2 Paso de argumentos a una aplicación Java en tiempo de ejecución 40

Fundamentos del lenguaje

1. Las variables, constantes y enumeraciones **43**

1.1 Las variables 43
1.1.1 Nombre de las variables 44
1.1.2 Tipo de variables 44
1.1.3 Valores por defecto 48
1.1.4 Valores literales 48
1.1.5 Conversión de tipos 49
1.1.6 Declaración de las variables 53
1.1.7 Alcance de las variables 54
1.1.8 Nivel de acceso de las variables 54
1.1.9 Ciclo de vida de las variables 55
1.2 Las constantes 55
1.3 Las enumeraciones 56
1.4 Los arrays 59
1.5 Las cadenas de caracteres 64
1.6 Fecha y hora 71

2. Los operadores **74**

2.1 Los operadores unarios 75
2.2 Los operadores de asignación 76

2.3 Los operadores aritméticos 76
2.4 Los operadores bit a bit 77
2.5 Los operadores de comparación 77
2.6 El operador de concatenación 78
2.7 Los operadores lógicos 79
2.8 Orden de evaluación de los operadores 80

3. Las estructuras de control 81

3.1 Estructuras de decisión 81
3.1.1 Estructura if 81
3.1.2 Estructura switch 83
3.2 Las estructuras de bucle 85
3.2.1 Estructura while 85
3.2.2 Estructura do ... while 86
3.2.3 Estructura for 86
3.2.4 Interrupción de una estructura de bucle 89

4. Ejercicios 92

5. Correcciones 93

Programación orientada a objetos

1. Introducción 101

2. Puesta en práctica con Java 104

2.1 Creación de una clase 105
2.1.1 Declaración de la clase 105
2.1.2 Creación de los campos 106
2.1.3 Creación de métodos 107
2.1.4 Los métodos accesores 112
2.1.5 Constructores y destructores 113
2.1.6 Campos y métodos estáticos 114

2.1.7 Las anotaciones 116
2.2 Utilización de una clase 119
2.2.1 Creación de una instancia 120
2.2.2 Inicialización de una instancia 120
2.2.3 Destrucción de una instancia 123
2.3 Herencia 127
2.3.1 this y super 129
2.3.2 Clases abstractas 134
2.3.3 Clases finales 135
2.3.4 Conversión de tipo 136
2.3.5 La clase Object 143
2.4 Interfaces 149
2.4.1 Creación de una interfaz 150
2.4.2 Utilización de una interfaz 151
2.4.3 Métodos por defecto 155
2.5 Clases anidadas 160
2.5.1 Clases anidadas estáticas 160
2.5.2 Clases internas 161
2.5.3 Clases anónimas 163
2.6 Expresión lambda 168
2.7 Referencia de método 175
2.8 Los genéricos 178
2.8.1 Clases genéricas 179
2.8.2 Métodos genéricos 186
2.8.3 Los genéricos y la herencia 187
2.8.4 Limitación de los genéricos 193
2.9 Los paquetes 195
2.9.1 Creación de un paquete 196
2.9.2 Utilización e importación de un paquete 198

3. Gestión de las excepciones 200

3.1 Los errores de sintaxis 200
3.2 Los errores de ejecución 202
3.3 Les errores de lógica 202
3.3.1 Las excepciones 203
3.3.2 Recuperación de excepciones 204

3.3.3 Excepciones asociadas a recursos 208
3.3.4 Creación y activación de excepciones 212

4. Las colecciones 214

4.1 La clase ArrayList 215
4.2 La clase HashSet 219
4.3 La clase LinkedList 228
4.4 Streams y pipelines 229

5. Ejercicios 232

6. Correcciones 233

Aplicaciones gráficas

1. Introducción 255

1.1 Las bibliotecas gráficas 256
1.1.1 La biblioteca AWT 256
1.1.2 La biblioteca Swing 256
1.2 Construcción de la interfaz gráfica de una aplicación 257

2. Diseño de una interfaz gráfica 258

2.1 Las ventanas 258
2.2 La gestión de los eventos 263
2.3 Aspecto de los componentes 293
2.4 El posicionamiento de los componentes 294
2.4.1 FlowLayout 295
2.4.2 BorderLayout 297
2.4.3 GridLayout 303
2.4.4 BoxLayout 305
2.4.5 GridBagLayout 308
2.4.6 Sin renderizador 313
2.5 Los componentes gráficos 316

2.5.1 La clase JComponent 317
2.5.2 Visualización de la información 320
2.5.3 Los componentes de edición de texto 325
2.5.4 Los componentes de activación de acciones 331
2.5.5 Los componentes de selección 338
2.6 Los cuadros de diálogo 346
2.6.1 El cuadro para introducir información 346
2.6.2 El cuadro de mensaje 349
2.6.3 El cuadro de confirmación 350

Los applets

1. Principio de funcionamiento 353

2. Creación de un applet 354
2.1 Ciclo de vida de un applet 355
2.1.1 Métodos relacionados con el ciclo de vida del applet 355
2.1.2 Métodos de gestión del aspecto gráfico del applet 356
2.2 Construir la interfaz usuario de un applet 359
2.2.1 Creación de una fuente de caracteres 360
2.2.2 Obtener las dimensiones del applet 360
2.2.3 Dibujar los caracteres 361
2.2.4 Determinar las dimensiones de una cadena 361
2.3 Las imágenes en los applets 366
2.3.1 Carga de una imagen 367
2.3.2 Tratamiento de la imagen 369
2.3.3 Trazado de la imagen 369
2.4 Los hilos en los applets 372
2.4.1 Creación de un nuevo hilo 374
2.4.2 Definir el tratamiento a efectuar 374
2.4.3 Lanzar y parar un hilo 377
2.5 Los sonidos en los applets 379

3. Despliegue de un applet 381

3.1 La etiqueta <applet> 381
3.2 Configuración de un applet 383
3.2.1 Definir los parámetros 383
3.2.2 Recuperación de los parámetros en el applet 384
3.3 Seguridad en un applet 385
3.4 Comunicación entre applets 386
3.5 Interacción con el navegador y el sistema 388
3.5.1 Visualización en la consola 388
3.5.2 Utilización de la barra de estado del navegador 390
3.5.3 Visualización de una página html 391
3.5.4 Obtener ciertas propiedades del sistema 392

Acceso a las bases de datos

1. Principios del funcionamiento de una base de datos 395
1.1 Terminología 395
1.2 El lenguaje SQL 396
1.2.1 Búsqueda de información 397
1.2.2 Inserción de datos 398
1.2.3 Actualización de datos 399
1.2.4 Supresión de datos 400

2. Acceso a una base de datos desde Java 400
2.1 Presentación de jdbc 402
2.2 Carga del driver 403
2.3 Establecer y manipular la conexión 404
2.3.1 Establecer la conexión 404
2.3.2 Manipular la conexión 405
2.4 Ejecución de instrucciones SQL 410
2.4.1 Ejecución de instrucciones básicas con el objeto Statement 410
2.4.2 Ejecución de instrucciones configuradas con el objeto PreparedStatement 418
2.4.3 Ejecución de procedimientos almacenados con el objeto CallableStatement 422
2.5 Utilización de los juegos de registros con la interfaz ResultSet 425
2.5.1 Posicionamiento en un ResultSet 427

2.5.2 Lectura de los datos en un ResultSet 429
2.5.3 Modificación de los datos en un ResultSet 432
2.5.4 Supresión de datos en un ResultSet 435
2.5.5 Inserción de datos en un ResultSet 437
2.6 Gestión de las transacciones 438
2.6.1 Puesta en marcha de las transacciones 440
2.6.2 Puntos de salvaguarda 441
2.6.3 Niveles de aislamiento 441

Despliegue de aplicaciones

1. Archivos Java 443

1.1 Presentación 443
1.2 Manipulación de un archivo 444
1.2.1 Creación de un archivo 444
1.2.2 Visualización del contenido 445
1.2.3 Extracción 446
1.2.4 Actualización 446
1.2.5 Ejecución 447
1.3 El manifest 447
1.3.1 Presentación 447
1.3.2 Creación 448
1.4 Empaquetar y firmar un archivo 449
1.4.1 Empaquetado 449
1.4.2 Firma 453

2. Java Web Start 458

2.1 Presentación 458
2.2 Ejecución de una aplicación 458
2.2.1 Desde un navegador 459
2.2.2 Desde la caché local 459
2.3 Despliegue de una aplicación 462
2.3.1 Configuración del servidor Web 462
2.3.2 Creación del archivo JNLP 465

2.3.3 Desplegar la aplicación en el servidor 468
2.3.4 Creación de la página Web de inicio 471

índice 473

JAVA 8

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.

Los tres primeros capítulos presentan los fundamentos del lenguaje, de la **programación orientada a objetos** y las novedades de la versión 8. El lector descubrirá, en particular, las nuevas API de **gestión de datos**, las **expresiones Lambda** y su aplicación en la **gestión de colecciones**. Los siguientes capítulos abordan el desarrollo de **aplicaciones gráficas** con la **biblioteca Swing** y la creación de **applets** que permiten enriquecer fácilmente el contenido de las páginas Web. Se presenta también el desarrollo de aplicaciones cliente/servidor utilizando **la API JDBC** que asegura **el acceso a las bases de datos**. Siendo el despliegue una etapa importante para el éxito de una aplicación, el último capítulo presenta la **distribución** de una aplicación mediante la solución clásica de los **ficheros de archivos** o el uso más flexible de la tecnología **Java Web Start**.

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

El libro no necesita herramientas de desarrollo específicas. Basta con un editor de texto y las herramientas gratuitas disponibles en el sitio de Oracle para llevar a cabo un buen aprendizaje de este lenguaje apasionante y en pleno auge.

Existen elementos complementarios para su descarga en esta página.

Los capítulos del libro:
Prólogo - Presentación - Fundamentos del lenguaje - Programación orientada a objetos - Aplicaciones gráficas - Los applets - Acceso a las bases de datos - Despliegue de aplicaciones

---

Thierry GROUSSARD

Al cabo de más de 10 años como analista y desarrollador, **Thierry GROUSSARD** se orientó 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 sus libros sean especialmente adecuados 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 veinte años más tarde 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 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 del lenguaje más utilizado en la enseñanza.

El objetivo de este libro es permitirle descubrir los fundamentos de este lenguaje para permitirle a continuación evolucionar hacia el desarrollo de aplicaciones importantes utilizando numerosas tecnologías disponibles con este lenguaje (JEE, JME...).

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.

Los capítulos Aplicaciones gráficas y Los applets le permiten estudiar el diseño de aplicaciones gráficas autónomas con la biblioteca SWING, y el desarrollo de aplicaciones que se ejecutan en el contexto de un navegador web con la tecnología de applets.

Sus futuras aplicaciones requerirán sin duda procesar información alojada en una base de datos. El capítulo Acceso a las bases de datos, dedicado a este tema, le proporcionará una preciosa ayuda para realizar esta tarea correctamente. Se familiarizará con el uso de JDBC que es la tecnología utilizada por Java para la gestión del acceso a una base de datos.

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 a 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.

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 bautizaron finalmente 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

En base a 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 a 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, en particular en el caso de los applets de Java en los cuales el motor de ejecución de Java se encarga de que el applet no realice ninguna manipulación u operación peligrosa.

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 Java es poco voluminoso, su descarga desde Internet requiere poco tiempo.
- un programa 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 Java, llamados applets, en páginas HTML.

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).

Características de Java

Java es a la vez un lenguaje y una plataforma de desarrollo.

Esta sección le presenta ambos aspectos. Le presentará las características de Java y le ayudará a evaluar la importancia del interés creado en torno a Java.

1. El lenguaje de programación Java

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

Dichas características son el resultado del manual escrito en mayo de 1996 por James Gosling y Henry Mc Gilton y disponible en la dirección siguiente: http://www.oracle.com/technetwork/java/langenv-140151.html

Vamos a explicar detallamente cada una de estas características.

a. 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 sólo 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 "el 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 sólo 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++.

b. Orientado a objetos

Salvo los tipos de datos primitivos, todo en Java es un objeto. Y además, Java se ha provisto de clases incorporadas que encapsulan los tipos primitivos.

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 (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 manipular todo tipo de objetos (interfaz gráfica, acceso a la red, gestión de entradas/salidas...).

c. Distribuido

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

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

- RMI (*Remote Method Invocation*) permite a los objetos Java comunicarse entre ellos tanto si se ejecutan en diferentes máquinas virtuales 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 Java, C++, Lisp, Python, Smalltalk, COBOL, Ada, que se ejecutan en diferentes máquinas físicas.

d. Interpretado

Un programa Java no lo ejecuta sino que lo interpreta la máquina virtual o JVM (*Java Virtual Machine*). Esto hace que sea más lento. Sin embargo conlleva también sus ventajas, en particular el hecho de no tener que recompilar un programa 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, sólo 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.

e. 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 manipulación de datos corruptos.

f. 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 (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 Java, el JRE utiliza un proceso llamado el 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. Es la configuración por defecto para los applets Java.
- Autorizar el acceso a máquinas en la red externa (más allá del firewall), 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.

g. Independiente de las arquitecturas

El compilador 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 Java. Esta última se encarga de traducir el bytecode a código nativo.

h. Portable

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

Además, a diferencia del lenguaje 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 estándar de Java facilitan la escritura de código fuente que, a continuación, se puede desplegar en diferentes plataformas sin adaptación.

i. Eficaz

Incluso si un programa 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++.

j. 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 del CPU o repartiendo las tareas entre varios procesadores.

k. Dinámico

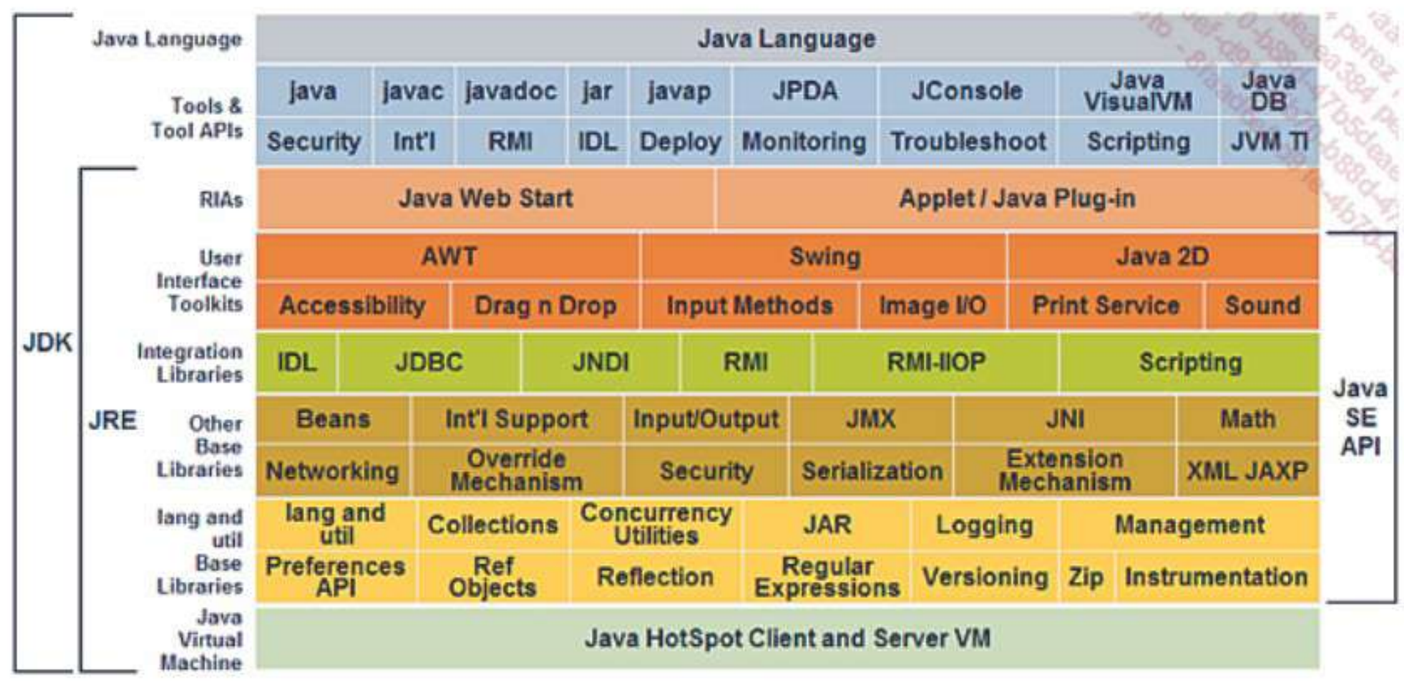
En Java, como dijimos, el programador no tiene que editar los vínculos (obligatorio en C y C++). Por lo tanto es posible modificar una o varias clases sin tener que efectuar una actualización de estas modificaciones para el conjunto del programa. La comprobación de la existencia de las clases se realiza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

2. La plataforma Java

Por definición, una plataforma es un entorno de hardware o de software en la cual se puede ejecutar un programa. La mayoría de las plataformas actuales son la combinación de una máquina y de un sistema operativo (ej: PC + Windows).

La plataforma Java se distingue por el hecho de que sólo se compone de una parte de software que se ejecuta en numerosas plataformas físicas y diferentes sistemas operativos.

El esquema siguiente procede del sitio web de Oracle sobre el lenguaje Java y muestra los diferentes componentes de la plataforma Java:



Como muestra el esquema, se compone de los elementos siguientes:

- la máquina virtual Java (JVM),
- la interfaz de programación de aplicación Java (API Java), repartida en tres categorías (API básicas, API de acceso a los datos y de integración con lo existente, API de gestión de la interfaz de las aplicaciones con el usuario),
- las herramientas de despliegue de las aplicaciones,
- las herramientas de ayuda al desarrollo.

Veamos en detalle estos diferentes elementos.

a. La máquina virtual Java (JVM)

La máquina virtual es la base de la plataforma Java. Es necesaria para la ejecución de los programas 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 que contengan: cuando un programa invoca la creación de objetos o invoca miembros de una clase, la JVM tiene como misión cargar el bytecode 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 de encargarse de la liberación automática de la memoria (recolector de basura) en cuanto sale del dominio del programa, es decir cuando ninguna variable le hace referencia.
- 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. En el caso de los applets Java, la JVM prohíbe al programa el acceso a los periféricos de la máquina en la cual se ejecuta el applet y autoriza el acceso a la red sólo hacia el host que difunde el applet.
- 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 o 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).

b. La API Java

La API Java contiene una colección de componentes de software prefabricados que proporcionan numerosas funcionalidades.

La API Java en su versión 8 se organiza en más de 220 paquetes, el equivalente a las librerías de C. Cada paquete contiene las clases e interfaces prefabricadas y directamente reutilizables. Hay disponibles unas 4300 clases e interfaces.

La plataforma Java proporciona API básicas. 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...

Las API Java se dividen en tres categorías:

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...
- los applets Java en el entorno del navegador Web.
- la red, con los protocolos estándar tales como FTP, HTTP, UDP, TCP/IP más las URL y la manipulación de los sockets.
- la internacionalización y la adaptación de los programas 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*),
  - autentificar 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.

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

Las API de integración permiten gestionar:

- aplicaciones cliente/servidor en una arquitectura distribuida, al permitir la comunicación en local o por red entre objetos Java que funcionan en contextos de JVM diferentes, gracias a la API RMI (*Remote Method Invocation*).
- aplicaciones cliente/servidor en una arquitectura distribuida, al permitir la comunicación en local o por red entre objetos Java y objetos compatibles CORBA tales como C++, Lisp, Python, Smalltalk, COBOL, Ada, gracias al soporte de la API CORBA (*Common Object Request Broker Architecture*), basada en el trabajo del OMG (http://www.omg.org).
- 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*).

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

Las API de gestión de la interfaz 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.
- 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 por la 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 por la 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.

c. Las herramientas de despliegue de las aplicaciones

La plataforma Java proporciona dos herramientas que permiten ayudar en el despliegue de las aplicaciones:

- Java Web Start: destinada a simplificar el despliegue y la instalación de las aplicaciones Java autónomas. Las aplicaciones están disponibles en un servidor, los usuarios pueden lanzar la instalación desde su máquina mediante la consola Java Web Start y todo se hace automáticamente. Lo interesante es que después, con cada lanzamiento de una aplicación, Java Web Start comprueba si está disponible una actualización en el servidor y procede automáticamente a su instalación.
- Java Plug-in: destinada a permitir el funcionamiento de los applets Java con la máquina virtual 8. En efecto, cuando se accede, mediante el navegador web, a una página html que contiene un applet, es la máquina virtual del navegador la encargada de hacerlo funcionar. El

problema es que las máquinas virtuales de los navegadores son compatibles con antiguas versiones de Java. Para no tener limitaciones a nivel de funcionalidades y por lo tanto no encontrar problemas de incompatibilidad entre los navegadores, se puede instalar el Java Plug-in en los terminales de los clientes. El Java Plug-in consiste en instalar un motor de ejecución Java 8 (el JRE compuesto por una JVM y por el conjunto de API). Con ello se consigue que los navegadores Web utilicen este JRE y no el suyo propio.

d. 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 del J2SE.

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 estándar proporcionadas por Sun.
- lanzar la ejecución (java.exe) de las aplicaciones autónomas Java.
- visualizar, con la ayuda de un visualizador (appletviewer.exe), la ejecución de un applet Java en una página HTML.

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

- JPDA (*Java Platform Debugger Architecture*), que 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*), que 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.

3. Ciclo de diseño de un programa Java

Para desarrollar una aplicación Java, primero se debe buscar la plataforma J2SE de desarrollo (SDK - *Software Development Kit*) compatible con su máquina y su sistema operativo: puede encontrar la suya en el listado del sitio Java de Oracle: http://www.oracle.com/technetwork/java/index.html

A continuación, podrá utilizar las API estándar de Java para escribir su código fuente. En Java, la estructura básica de un programa es la clase y cada clase se debe encontrar en un archivo con la extensión java. Un mismo archivo .java puede contener varias clases, pero sólo una de ellas puede ser declarada pública. El nombre de esta clase declarada pública da su nombre al archivo .java.

A lo largo del desarrollo, podrá proceder a la fase de compilación utilizando la herramienta javac.exe. Como resultado obtendrá al menos un archivo que lleva el mismo nombre pero con la extensión .class. El archivo .class compilado sigue siendo de todas formas independiente de cualquier plataforma o sistema operativo.

A continuación, es el intérprete (java.exe) quien ejecuta los programas Java. Para la ejecución de los applets, se incorpora el intérprete al navegador de Internet compatible con Java. Para la ejecución de aplicaciones Java autónomas, es necesario lanzar la ejecución de la máquina virtual proporcionada ya sea con la plataforma de desarrollo Java (SDK) o con el kit de despliegue de aplicaciones Java (JRE - *Java Runtime Environment*).

Instalación del SDK versión Win32 para el entorno Windows

1. Descarga

En primer lugar, es necesario descargar la última versión del SDK para el entorno Windows (Win32) a partir del sitio web de Oracle: http://www.oracle.com/technetwork/java/javase/downloads/index.html

Actualmente, el archivo de descarga se llama jdk-8u5-windows-i586.exe y ocupa 152 MB. En todo caso, se debe descargar siempre la última versión disponible.

Ya que está en el sitio web de Oracle, aproveche para descargar otro elemento indispensable para programar en Java: la documentación de las API estándar.

Actualmente, el archivo de descarga se llama jdk-8u5-apidocs.zip y ocupa 85 MB. Para poder descomprimirlo en nuestra máquina, necesitamos 300 MB de espacio de disco disponible. ¡Esto representa mucha lectura!

2. Instalación

Antes de instalar el SDK en el ordenador, debemos asegurarnos de que no hay ningún otra herramienta de desarrollo Java ya instalada, para evitar problemas de conflictos de configuración.

Para empezar la instalación, hacemos doble clic en el archivo de instalación descargado previamente: jdk-8u5-windows-i586.exe.

Primero aparece un cuadro de diálogo **Welcome**, para indicarle que está a punto de instalar el SDK y le pide confirmar si quiere continuar con la instalación.

Haga clic en **Next**.

Una nueva ventana, **Custom Setup**, le permite seleccionar los elementos del SDK que quiere instalar y la carpeta de destino de la instalación.

Una vez haya seleccionado sus opciones o haya dejado la selección por defecto, pulse **Next**. El programa instala así los archivos en nuestro ordenador. Instantes más tarde, el cuadro de diálogo siguiente nos informa del éxito de la instalación.

3. Configuración

Ahora falta configurar el sistema, indicando en qué carpeta se encuentran almacenadas las herramientas como java.exe (máquina virtual), appletviewer.exe (visionador de applets) o también javac.exe (compilador). Para ello, debemos modificar la variable de entorno PATH para añadir la ruta de acceso hacia la carpeta bin del jdk. Si ha dejado las opciones por defecto durante la instalación, la ruta debe ser C:\Program Files\Java\jdk1.8.0\bin

4. Prueba de la configuración del SDK

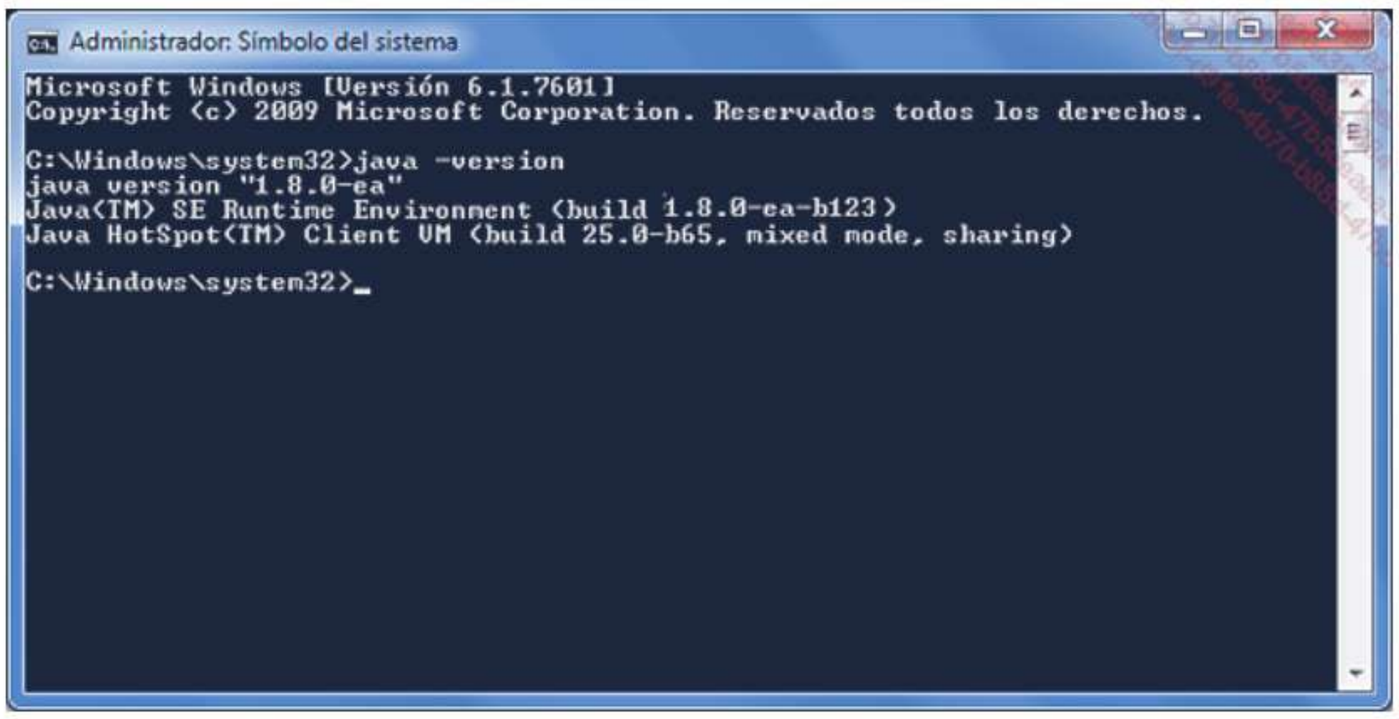
Vamos a comprobar si el ordenador ha tenido en cuenta las modificaciones que acabamos de aportar a la variable PATH y, por lo tanto, vamos a comprobar, si encuentra la ruta donde están las herramientas del SDK.

Para probar la configuración del SDK, vamos a utilizar una ventana de **comandos**.

En Símbolo del sistema, introducimos el comando siguiente que va a permitir determinar si la instalación del SDK es correcta o no:

```
java -version
```

Debemos ver aparecer el mensaje siguiente como respuesta a la línea que hemos introducido:



Este comando muestra información relativa a la versión de la máquina virtual Java.

Si obtenemos un mensaje del estilo: `No se reconoce a 'java' como archivo interno o externo, un programa ejecutable o un archivo de comandos`, significa que la carpeta donde se almacenan las herramientas del SDK no ha sido encontrado por nuestro sistema.

En este caso, comprobamos si la variable PATH contiene efectivamente las modificaciones que hemos aportado y que no hemos cometido un error de sintaxis al definir la ruta de la carpeta bin.

5. Instalación de la documentación del SDK y de las API estándar

Con la ayuda de una herramienta de descompresión como WinZip, abrimos el archivo que hemos descargado previamente. Extraemos todos los archivos que contenga en la carpeta raíz de instalación del SDK, es decir, por defecto `C:\Program Files\Java\jdk1.8.0`

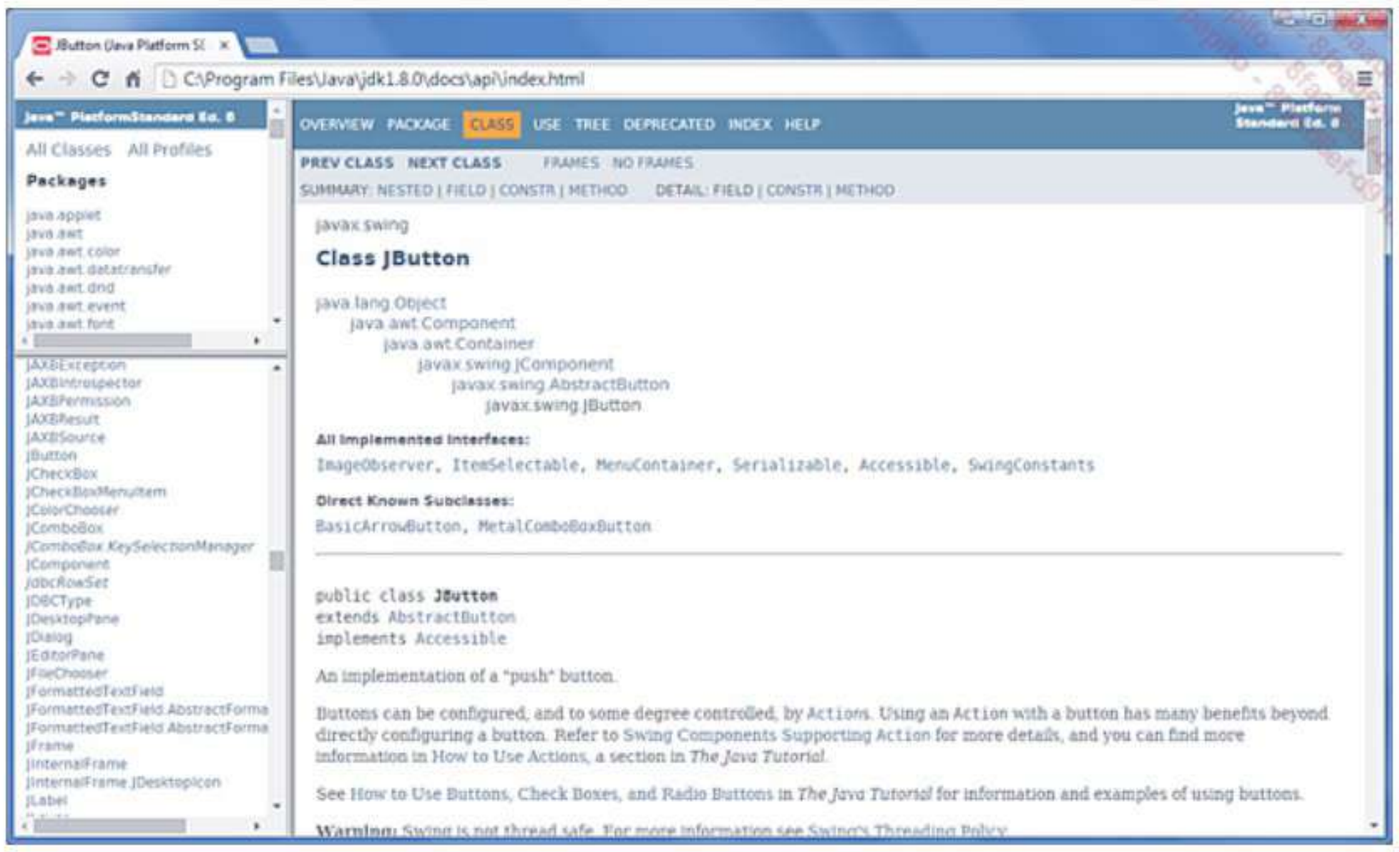
Se deben prever 270 MB de espacio disponible en disco para instalar la documentación.

Una vez extraídos todos los archivos, cerramos la herramienta. En el explorador Windows, en la carpeta `C:\Program Files\Java\jdk1.8.0`, debemos tener una nueva carpeta `docs`. Es la carpeta que contiene el conjunto de la documentación del SDK en formato HTML.

En esta carpeta `docs`, hacemos doble clic en el archivo `index.html`. Este archivo contiene enlaces hacia el conjunto de la documentación Java, que está instalada en su ordenador, o accesible en un sitio Web.

Lo más importante de la documentación se encuentra en la subcarpeta `api`, al hacer doble clic en el archivo `index.html`. Este archivo contiene las especificaciones de la API Java, o más específicamente, la descripción del conjunto de las clases de la librería Java. Sin esta documentación, no podremos desarrollar eficazmente en Java.

Se recomienda crear en su escritorio un acceso directo hacía este documento.



Esta página se organiza en tres ventanas:

- la ventana superior izquierda contiene la lista de los paquetes (más de 220).
- la ventana inferior izquierda contiene la lista de las clases contenidas en el paquete seleccionado en la ventana anterior.
- la ventana más grande contiene la descripción de una interfaz o de una clase seleccionada en la ventana anterior. La descripción de una clase se organiza de la manera siguiente:
  - un esquema de la jerarquía de las superclases de la interfaz o de la clase en curso.
  - una explicación sobre la utilización de la clase o de la interfaz.
  - *Field Summary*: lista de los atributos.
  - *Constructor Summary*: lista de los constructores de la clase.
  - *Method Summary*: lista de los métodos.
  - *Field Details*: descripción detallada de los atributos.
  - *Constructor Details*: descripción detallada de los constructores de la clase.
  - *Method Details*: descripción detallada de los métodos de la clase.

Las diferentes etapas de creación de un programa Java

1. Creación de los archivos fuente

En primer lugar, debe crear uno o varios archivos de código fuente, según la importancia de su programa.

Todo código java se encuentra en el interior de una clase contenida ella misma en un archivo con la extensión `java`.

Varias clases pueden coexistir en un mismo archivo .java pero sólo una puede ser declarada pública, y es esta última la que da su nombre al archivo.

Como en muchos otros lenguajes de programación, los archivos fuente Java son archivos de texto sin formato.

Un simple editor de texto capaz de grabar en formato de texto ASCII, como el Bloc de notas de Windows o VI de Unix, es suficiente para escribir archivos de código fuente Java.

Una vez escrito hay que guardar el código de su archivo fuente con la extensión `java`, que es la extensión de los archivos fuente.

Si usa el Bloc de notas de Windows, tenga cuidado de que al guardar su archivo el Bloc de notas no añada una extensión `.txt` al nombre. Para evitar este tipo de problemas, dé nombre a su archivo con la extensión `java`, todo ello escrito entre comillas.

Sin embargo, existe algo mejor que un simple editor. Puede, previo pago del coste de una licencia, utilizar herramientas comerciales o, aún mejor, utilizar productos open source como el excelente **Eclipse**. Se trata en un principio de un proyecto de IBM pero numerosas empresas se han unido a este proyecto (Borland, Oracle, Merant...). Es una herramienta de desarrollo Java excelente y gratuita a la cual se pueden acoplar otras aplicaciones mediante un sistema de plug-in. Oracle proporciona también **NetBeans**, una herramienta muy eficaz y de uso fácil.

2. Compilar un archivo fuente

Una vez creado y guardado su archivo fuente con la extensión .java, debe compilarlo.

Para compilar un archivo fuente Java, hay que utilizar la herramienta en línea de comando javac proporcionada con el SDK.

Abra una ventana **Símbolo del sistema**.

En la ventana, sitúese en la carpeta que contiene su archivo fuente (.java), con la ayuda del comando **cd** seguido de un espacio y del nombre de la carpeta que contiene su archivo fuente.

Una vez que esté en la carpeta correcta, puede lanzar la compilación de su archivo fuente usando el siguiente comando en la ventana de Símbolo del sistema: `javac <nombrearchivo>.java`

`javac`: compilador Java en línea de comando, proporcionado con el JDK.

`<nombrearchivo>`: nombre del archivo fuente Java.

`.java`: extensión que indica que el archivo es una fuente Java.

Si quiere compilar varios archivos fuente al mismo tiempo, basta con escribir el comando anterior y añadir los demás archivos a compilar separándolos por un espacio.

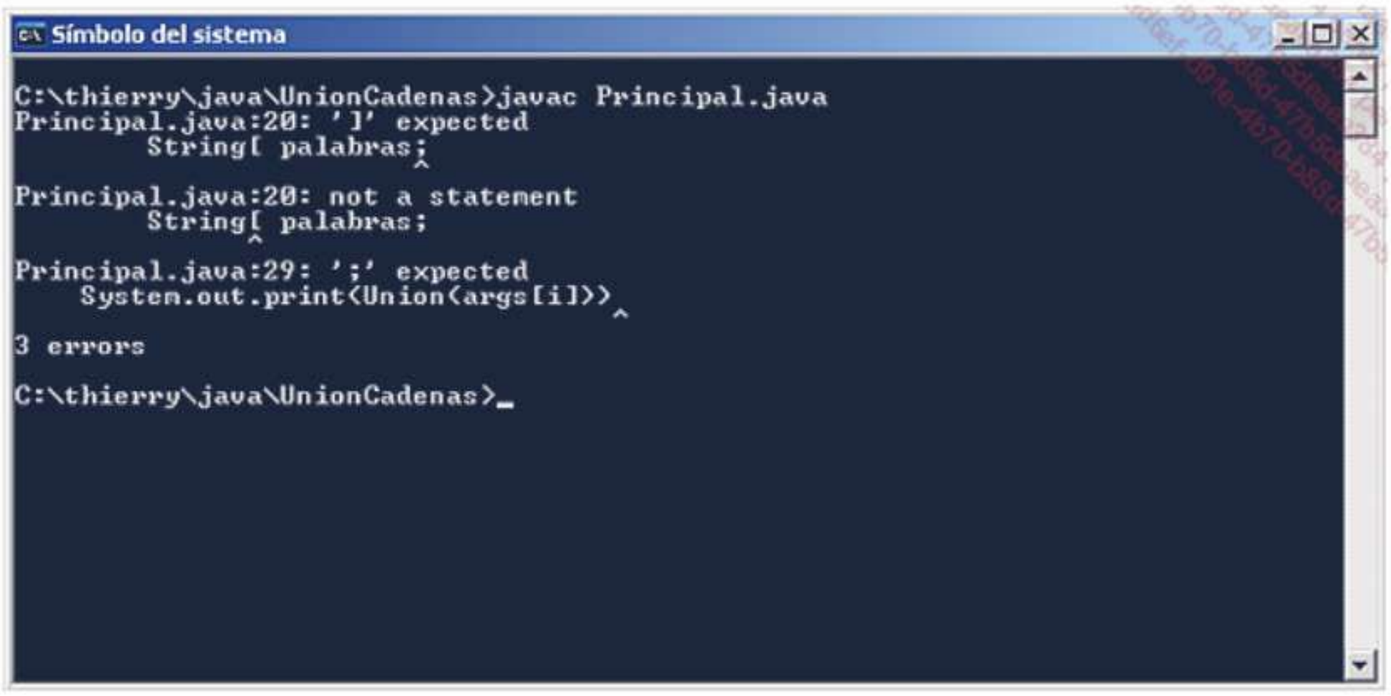
```
javac <nombrearchivo1>.java <nombrearchivo2>.java
```

Si después de unos segundos ve aparecer de nuevo la ventana de Símbolo de sistema, significa que nuestro archivo no contiene errores y se ha compilado. En efecto, el compilador no muestra ningún mensaje cuando la compilación se ejecuta correctamente.

El resultado de la compilación de un archivo fuente Java es la creación de un archivo binario que lleva el mismo nombre que el archivo fuente pero con la extensión .class.

Un archivo binario .class contiene el pseudo-código Java que la máquina virtual Java puede interpretar.

Si, por el contrario, ve aparecer una serie de mensajes, de los cuales el último le indica un número de errores, esto quiere decir que el archivo fuente contiene errores y que javac no consiguió compilarlo.



En este caso, se debe corregir el archivo fuente.

Para ayudarle a encontrar los errores en su o sus archivos fuente, javac le proporciona información muy útil:

```
<nombreArchivo.java> : <numLínea> : <mensaje> <línea de código>
```

`<nombreArchivo>`

Nombre del archivo fuente Java que contiene un error.

`<numLínea>`

Número de la línea de su archivo fuente donde javac encontró un error.

`<mensaje>`

Mensaje que indica el tipo de error.

`<línea>`

Línea de código que contiene un error, javac indica con una flecha dónde se ubica el error en la línea.

Después de haber corregido el código, vuelva a compilarlo. Si javac le sigue reportando errores, repita la operación de corrección y de recompilación del archivo hasta obtener la creación del archivo binario .class.

Por defecto, los archivos compilados se crean en la misma carpeta que sus archivos fuente. Puede indicar a la herramienta javac crearlos en otra carpeta mediante la opción -d "directory".

3. Ejecutar una aplicación

Una aplicación Java es un programa autónomo, similar a los programas que conoce pero que, para ser ejecutado, necesita el uso de un intérprete Java (la máquina virtual Java) que carga el método `main()` de la clase principal de la aplicación.

Para iniciar la ejecución de una aplicación Java, se debe utilizar la herramienta en línea de comando `java` proporcionada con el JDK.

Abra una ventana **Símbolo del sistema**. Ubíquese en la carpeta que contiene el o los archivos binarios (.class) de su aplicación. A continuación, introduzca el comando con la sintaxis siguiente:

```
java <archivoMain> <argumentoN> <argumentoN+l>
```

`java`: herramienta en línea de comandos que lanza la ejecución de la máquina virtual Java.

`<archivoMain>`: es obligatoriamente el nombre del archivo binario (.class) que contiene el punto de entrada de la aplicación, el método `main()`. Importante: no ponga la extensión .class después del nombre del archivo porque la máquina virtual Java lo hace de manera implícita.

`<argumentoN> <argumentoN+1>`: argumentos opcionales en línea de comandos para pasar a la aplicación en el momento de su ejecución.

Si lanzamos la ejecución correctamente (sintaxis correcta, con el archivo que contiene el método `main()`, debe ver aparecer los mensajes que ha insertado en su código. Si por el contrario, ve un mensaje de error similar a `Exception in thread "main" java.lang.NoClassDefFoundError:...` es que su programa no se puede ejecutar.

Varias razones pueden ser la causa de ello:

- El nombre del archivo a ejecutar no tiene el mismo nombre que la clase (diferencia entre mayúsculas y minúsculas).
- Ha introducido la extensión `.class` después del nombre del archivo a ejecutar en la línea de comando.
- El archivo que ejecutó no contiene método `main()`.
- Está intentando ejecutar un archivo binario (`.class`) que se ubica en una carpeta distinta que desde donde se lanzó la ejecución.

Nuestra primera aplicación Java

1. Esqueleto de una aplicación

Una aplicación Java es un programa autónomo que se puede ejecutar en cualquier plataforma que disponga de una máquina virtual Java.

Es posible desarrollar cualquier tipo de aplicación en Java: interfaz gráfica, acceso a las bases de datos, aplicaciones cliente/servidor, multihilo...

Una aplicación se compone de, al menos, un archivo .class y él mismo debe contener como mínimo el punto de entrada de la aplicación, el método `main()`.

Ejemplo de la estructura mínima de una aplicación

```
public class MiAplicación {
public static void main(String argumentos[]) {
/* cuerpo del método principal */
}
}
```

Si la aplicación es importante, se pueden crear tantas clases como sean necesarias. Las clases que no contengan el método `main()` se llaman clases auxiliares.

El método `main()` es el primer elemento llamado por la máquina virtual Java al lanzar la aplicación.

El cuerpo de este método debe contener las instrucciones necesarias para el arranque de la aplicación, es decir, la creación de instancias de clase, la inicialización de variables y la llamada a métodos.

Idealmente, el método `main()` puede contener una única instrucción.

La declaración del método `main()` siempre se hace según la sintaxis siguiente:

```
public static void main(String <identificador>[ ] ) {...}
```

`public`

Modificador de acceso utilizado para hacer que el método sea accesible al conjunto de las demás clases y objetos de la aplicación, y también para que el intérprete Java pueda acceder a él desde el exterior al ejecutar la aplicación.

`static`

Modificador de acceso utilizado para definir el método `main()` como método de clase. La máquina virtual Java puede por tanto invocar a este método sin tener que crear una instancia de la clase en la cual está definido.

`void`

Palabra clave utilizada para indicar que el método es un procedimiento que no devuelve valor.

`main`

Identificador del método.

`String <identificador>[ ]`

Parámetro del método, es un vector de cadenas de caracteres. Este parámetro se utiliza para pasar argumentos en línea de comando al ejecutar la aplicación. En la mayoría de los programas, el nombre utilizado para `<identificador>` es `argumentos` o `args`, para indicar que la variable contiene argumentos para la aplicación.

2. Argumentos en línea de comando

a. Principios y utilización

Al ser una aplicación Java un programa autónomo, puede ser interesante proporcionarle parámetros u opciones que van a determinar el comportamiento o la configuración del programa en el momento de su ejecución.

Los argumentos en línea de comando se almacenan en un vector de cadenas de caracteres. Si quiere utilizar estos argumentos con otro formato, debe efectuar una conversión de tipo, del tipo String hacía el tipo deseado durante el procesamiento del argumento.

¿En qué casos se deben utilizar los argumentos en línea de comandos?

Los argumentos en línea de comandos se deben utilizar al arrancar una aplicación en cuanto uno o varios datos utilizados en la inicialización de nuestro programa pueden adoptar valores variables según el entorno. Por ejemplo:

- nombre del puerto de comunicación utilizado en el caso de una comunicación con un dispositivo físico.
- dirección IP de una máquina en la red en el caso de una aplicación cliente/servidor.
- nombre del usuario y contraseña en el caso de una conexión a una base de datos con gestión de los permisos de acceso.

Por ejemplo, en el caso de una aplicación que accede a una base de datos, es habitual tener que proporcionar un nombre de usuario y una contraseña para abrir una sesión de acceso a la base de datos. Diferentes usuarios pueden acceder a la base de datos, pero con permisos diferentes.

Por lo tanto pueden existir varias sesiones diferentes. No es factible crear una versión de la aplicación para cada usuario.

Además, esta información es susceptible de ser modificada. Así que no parece juicioso integrarla en su código, porque cualquier cambio le obligaría a modificar su código fuente, volver a compilarlo y tener una versión para cada usuario.

La solución a este problema reside en los argumentos en línea de comando.

Basta con utilizar en su código el vector de argumentos del método main que contiene las variables (nombre y contraseña) de su aplicación.

A continuación, y en función del usuario del programa, hay que acompañar el nombre de la clase principal, en el momento de la ejecución del programa mediante la instrucción `java`, con el valor de los argumentos por línea de comandos.

b. Paso de argumentos a una aplicación Java en tiempo de ejecución

El paso de argumentos a una aplicación Java se hace al lanzar la aplicación mediante la línea de comando. El siguiente ejemplo de programa muestra cómo utilizar el paso de argumentos por línea de comandos en una aplicación Java.

```
/* Declaración de la clase principal de la aplicación */
public class MiClase
{
/* Declaración del método de punto de entrada de la aplicación*/
public static void main(String args[])
{

/* Visualización de los argumentos de la línea de comando */

for (int i = 0 ; i < args.length; i++)



    System.out.printIn("Argumento " +i + " = " + args[i]) ;

}

/* Conversión de dos argumentos de la línea de comando de
String a int, suma de los valores enteros y
visualización del resultado */

int suma;
suma=(Integer.parselnt(args[3]))+(Integer.parselnt(args[4]));
System.out.println("Argumento 3 + Argumento 4 = " + suma);
}
}
|
```

Tras la compilación, el programa se ejecuta con la línea de comando siguiente:

```
java MiClase ediciones ENI "ediciones ENI" 2 5
```

La ejecución del programa muestra la información siguiente:

```
Argumento    0    =    ediciones
Argumento    1    =    ENI
Argumento    2    =    ediciones ENI
Argumento    3    =    2
Argumento    4    =    5
Argumento    3    +    Argumento 4 = 7
```

Las variables, constantes y enumeraciones

1. Las variables

Las variables nos van a permitir almacenar en memoria diferentes valores útiles para el funcionamiento de nuestra aplicación durante su ejecución. Se debe declarar obligatoriamente una variable antes de utilizarla en el código. Al declarar una variable debemos definir sus características. 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. Sólo existirá si una instancia de la clase está disponible. Cada instancia de clase tendrá su propio ejemplar de la variable.
- Declarada con la palabra clave `static` en el interior de una clase, 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.
- Declarada en el interior de una función, la variable es una variable local. Sólo existe durante la ejecución de la función y sólo se puede acceder a ella desde el código de ésta.
- Los parámetros de las funciones se pueden considerar como variables locales. La única diferencia reside en la inicialización de la variable efectuada durante la llamada a la función.

a. Nombre de las variables

Veamos las reglas que se deben respetar para nombrar a las variables.

- 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, los nombres de variable se ortografían con letras minúsculas salvo la primera letra de cada palabra si el nombre de la variable incluye varias palabras (edadDelCapitan).

b. Tipo de variables

Al determinar un tipo para una variable, indicamos cuál es la información que vamos a poder almacenar en esta variable y 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 lenguaje Java dispone de siete tipos primitivos que se pueden clasificar en tres categorías.

Los tipos numéricos enteros

Tipos enteros firmados			
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, tendrá que tener en cuenta los valores mínimo y máximo que piensa almacenar en ella con el fin de optimizar la memoria de la que hacen uso. De hecho, es inútil utilizar un tipo largo 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.

Todos los tipos enteros son firmados. Es, no obstante, posible trabajar con valores enteros no firmados utilizando las clases `Integer` y `Long`. Esto permite extender el valor positivo máximo admisible en un tipo `int` hasta 4294967296 y hasta 18446744073709551616 para un tipo `long`. Es preciso, sin embargo, tomar ciertas precauciones. Por ejemplo, el siguiente código no compilará.

```
distancia=new Integer(3000000000);
```

El compilador verifica que el valor literal provisto al constructor no supera los límites admitidos para el tipo `int` y genera un error. Para poder extender este límite, hay que utilizar el método estático `parseUnsignedInt`, que acepta como parámetro una cadena de caracteres.

```
int distancia;
distancia=Integer.parseUnsignedInt("3000000000");
```

El posterior uso de esta variable deberá tener en cuenta la especificidad de su tipo no firmado. La visualización de su contenido deberá realizarse mediante el método estático `toUnsignedString`. El siguiente código permite aclarar esta especificidad.

```
System.out.println("visualización como int:" + distancia);
System.out.println("visualización como int no firmado:"
   +Integer.toUnsignedString(distancia));
```

Este código muestra la información siguiente en la consola:

```
visualización como int:-1294967296
visualización como int no firmado:3000000000
```

Los tipos decimales

float	1.4E-45	3.4028235E38	4 bytes
double	4.9E-324	1.7976931348623157E308	8 bytes

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

El tipo carácter

El tipo `char` se utiliza para almacenar un carácter único. Una variable de tipo `char` utiliza 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. 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
\t	Tabulación
\b	Retroceso
\n	Salto de línea
\r	Retorno de carro
\f	Salto de página
\'	Comilla simple
\"	Comilla doble
\\	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.

Para poder almacenar cadenas de caracteres hay que utilizar el tipo `String` que representa una serie de cero a n caracteres. Este tipo no es un tipo primitivo sino una clase. Sin embargo, para facilitar su utilización, se puede utilizar como un tipo primitivo del lenguaje. Las cadenas de caracteres son invariables, porque durante la asignación de un valor a una variable de tipo cadena de caracteres se reserva algo de espacio en memoria para el almacenamiento de la cadena. Si más adelante esta variable recibe un nuevo valor, se le atribuye una nueva ubicación en memoria. Afortunadamente, este mecanismo es transparente para nosotros y la variable seguirá haciendo referencia automáticamente al valor que se le asignó. Con este mecanismo, las cadenas de caracteres pueden tener un tamaño variable. Se ajusta automáticamente el espacio ocupado en memoria según la longitud de la cadena de caracteres. Para atribuir una cadena de caracteres a una variable es necesario introducir el contenido de la cadena entre " y " como en el ejemplo siguiente.

Ejemplo

```
nombreDelCapitan = "Garfio";
```

Existen muchas funciones de la clase `String` que permiten manipular las cadenas de caracteres y que se detallan más adelante en este capítulo.

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.

c. Valores por defecto

La inicialización de las variables no siempre es obligatoria. Es el caso, por ejemplo, de las variables de instancia que se inicializan con los valores por defecto siguientes.

Tipo	Valor por defecto
byte	0
short	0
int	0
long	0
float	0.0
double	0.0
char	\u0000

Tipo	Valor por defecto
boolean	false
String	null

En cambio, las variables locales se deben inicializar antes de utilizarlas. El compilador efectúa de hecho una comprobación cuando encuentra el uso de una variable local y activa un error si la variable no ha sido inicializada.

d. Valores literales

Los valores numéricos enteros se pueden utilizar con su representación decimal, octal, hexadecimal o binario. Las cuatro líneas siguientes de código son equivalentes.

```
i=243;
i=0363;
i=0xF3;
i=0b11110011;
```

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

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

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

```
precio=1_234_876_567;
```

```
precio=1234876567;
```

Los valores literales están también caracterizados. Los valores numéricos enteros se consideran por defecto como tipos `int`. En cuanto a los valores numéricos reales se consideran como tipos `double`. Esta asimilación puede ser a veces fuente de errores de compilación al utilizar el tipo `float`. Las líneas siguientes generan un error de compilación porque el compilador considera que intentamos asignar a una variable de tipo `float` un valor de tipo `double` y piensa que hay riesgo de perder información.

```
float superficie;
superficie=2356.8;
```

Para resolver este problema, tenemos que forzar el compilador a considerar el valor literal real como un tipo `float` añadiéndole el carácter `f` o `F`.

```
float superficie;
superficie=2356.8f;
```

e. Conversión de tipos

Las conversiones de tipos consisten en transformar una variable de un tipo en otro. Las conversiones se pueden hacer hacia un tipo superior o hacia un tipo inferior. Si se utiliza una conversión hacia un tipo inferior, existe el riesgo de perder información. Por ejemplo la conversión de un tipo `double` en un tipo `long` provocará la pérdida de la parte decimal del valor. Por eso el compilador exige en este caso que le indiquemos de manera explícita que deseamos realizar esta operación. Para ello, debe prefijar el elemento que desea convertir con el tipo que quiere obtener ubicándolo entre paréntesis.

```
float superficie;
superficie=2356.8f;
int aproximacion;
aproximacion=(int)superficie;
```

En este caso, se pierde la parte decimal, pero a veces éste puede ser el objetivo de este tipo de conversión.

Las conversiones hacia un tipo superior no implican riesgo de perder información y por ello se realizan directamente mediante una simple asignación.

La siguiente tabla resume las conversiones posibles y si deben ser explícitas (☹) o implícitas (☺).

	Tipo de datos a obtener							
Tipo de datos de origen		byte	short	int	long	float	double	char
	byte		☺	☺	☺	☺	☺	☺
	short	☹		☺	☺	☺	☺	☺
	int	☹	☹		☺	☺	☺	☺
	long	☹	☹	☹		☺	☺	☹
	float	☹	☹	☹	☹		☺	☹
	double	☹	☹	☹	☹	☹		☹
	char	☹	☺	☺	☺	☺	☺	

Las conversiones desde y hacia cadenas de caracteres son más específicas.

Conversión hacia una cadena de caracteres

Las funciones de conversión hacia el tipo cadena de caracteres son accesibles mediante la clase `String`. El método de clase `valueOf` asegura la conversión de un valor de un tipo primitivo hacia una cadena de caracteres.

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

En determinadas situaciones, el uso de estas funciones es opcional porque la conversión se efectúa de manera implícita. Es el caso, por ejemplo, de una variable de un tipo primitivo que está concatenada con una cadena de caracteres. Las dos versiones de código siguientes son equivalentes.

Versión 1

```
double precioBruto;
precioBruto=152;
String recap;
recap="el importe del pedido es: " + precioBruto*1.16;
```

Versión 2

```
double precioBruto;
precioBruto=152;
String recap;
recap="el importe del pedido es: " +String.valueOf(precioBruto*1.16);
```

Conversión desde una cadena de caracteres

Ocurre a menudo que un valor numérico se presenta en una aplicación bajo la forma de una cadena de caracteres (lo introduce manualmente el usuario, lectura de un fichero…).

Para que la aplicación lo pueda manipular debe convertirse a un tipo numérico. Este tipo de conversión es accesible mediante clases equivalentes a los tipos primitivos. Permiten manipular valores numéricos bajo el formato de objetos. Cada tipo básico tiene su clase asociada.

Tipo básico	Clase correspondiente
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Estas clases se llaman clases Wrapper puesto que se utilizan para "embalar" en un objeto los tipos básicos del lenguaje. Pueden utilizarse como clases normales creando una instancia a partir de uno de los constructores disponibles. Esta solución puede esquivarse gracias al mecanismo llamado "autoboxing" del compilador.

Este mecanismo permite asignar un tipo básico del lenguaje a una variable del tipo wrapper correspondiente. Las siguientes dos líneas de código son equivalentes.

```
Integer entero=new Integer (10);
Integer entero=10;
```

El mecanismo inverso, llamado "unboxing", permite convertir automáticamente un tipo wrapper en un tipo básico. La variable entera del ejemplo anterior puede asignarse a una variable de tipo `int`.

Estas clases proporcionan un método parse... que recibe como parámetro una cadena de caracteres y permite convertirla en el tipo primitivo asociado a la clase.

```
int x);
x=entero;
```

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

Para recordar cómo realizar una conversión, se trata de aplicar un principio muy sencillo: el método que se debe utilizar se encuentra en la clase correspondiente al tipo de datos que se desea obtener.

f. Declaración de las variables

La declaración de una variable está constituida por el tipo de la variable seguido por el nombre de la variable. Por lo tanto la sintaxis básica es la siguiente:

```
int contador;
double precio;
String nombre;
```

También se pueden especificar modificadores de acceso y un valor inicial durante la declaración.

```
private int contador=0;
protected double precio=123.56;
public nombre=null;
```

La declaración de una variable puede aparecer en cualquier sitio del código. Sólo es necesario que la declaración preceda al uso de la variable. Se aconseja agrupar las declaraciones de variables al principio de la definición de la clase o de la función con el fin de facilitar la relectura del código.

La declaración de varias variables del mismo tipo se puede agrupar en una sola línea, separando los nombres de las variables con una coma.

```
protected double precioBruto=123.56, precioNeto, GastosEnvio;
```

g. Alcance de las variables

El alcance de una variable es la región de código en la que se puede manipular dicha variable. Cambia, pues, en función de la ubicación de la declaración.

Se puede hacer esta declaración en el bloque de código de una clase, en el bloque de código de una función o en un bloque de código en el interior de una función. Sólo el código del bloque donde se declara la variable puede utilizarlo. Si el mismo bloque de código se ejecuta varias veces durante la ejecución de la función, como es el caso de un bucle while por ejemplo, la variable se creará con cada paso del bucle. En este caso, la inicialización de la variable es obligatoria. No se pueden tener dos variables con el mismo nombre y con el mismo alcance. Sin embargo, tenemos la posibilidad de declarar una variable interna a una función, o un parámetro de una función con el mismo nombre que una variable declarada a nivel de la clase. En este caso, la variable declarada al nivel de la clase queda oculta por la variable interna de la función.

h. Nivel de acceso de las variables

El nivel de acceso de una variable se combina con el alcance de la variable y determina qué sección de código tiene derecho a leer y escribir en la variable. Un conjunto de palabras clave permite controlar el nivel de acceso. Se utilizan en la declaración de la variable y deben informarse delante del tipo de la variable. Sólo pueden utilizarse para declarar una variable en el interior de una clase. Queda prohibido su uso en el interior de una función.

private: la variable sólo se utiliza con el código de la clase donde está definida.

protected: la variable se utiliza en la clase donde está definida, en las subclases de esta clase y en las clases que forman parte del mismo paquete.

public: la variable es accesible desde cualquier clase sin importar el paquete.

ningún modificador: la variable es accesible desde todas las clases que forman parte del mismo paquete.

static: esta palabra clave se asocia a una de las palabras clave anteriores para transformar una declaración de variable de instancia en declaración de variable de clase (permite utilizarla sin que exista una instancia de la clase).

i. Ciclo de vida de las variables

El ciclo de vida de una variable nos permite especificar durante cuánto tiempo el contenido de una variable estará disponible a lo largo de la ejecución de la aplicación.

Para una variable declarada en una función, la duración del ciclo de vida corresponde a la duración de la ejecución de la función. En cuanto termine la ejecución del procedimiento o función, la variable se elimina de la memoria. Volverá a crearse con la próxima llamada a la función. Una variable declarada en el interior de una clase puede utilizarse mientras esté disponible una instancia de la clase. Las variables declaradas con la palabra clave `static` están accesibles durante todo el tiempo de ejecución de la aplicación.

2. Las constantes

En una aplicación puede ocurrir a menudo que se utilicen valores numéricos o cadenas de caracteres que no se modificarán durante la ejecución de la aplicación. Para facilitar la lectura del código, se aconseja crear esos valores bajo la forma de constantes.

La definición de una constante se realiza añadiendo la palabra clave `final` delante de la declaración de una variable. Es obligatorio inicializar la constante en el momento de su declaración (es el único sitio donde se puede asignar valor a la constante).

```
final double TASAIVA=1.21;
```

A continuación es posible utilizar la constante en el código en lugar del valor literal que representa.

```
precioNeto=precioBruto*TASAIVA;
```

Las reglas relativas al ciclo de vida y al 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 double TOTAL=100;
final double SEMI=TOTAL/2;
```

Existen muchas constantes que ya forman parte del lenguaje Java. Se definen como miembros `static` de las numerosas clases del lenguaje. Por convenio los nombres de las constantes se ortografían totalmente en mayúsculas.

3. Las enumeraciones

Una enumeración nos va a permitir agrupar un conjunto de constantes relacionadas entre sí. La declaración se hace de la siguiente manera:

```
public enum Días
{
    DOMINGO,
    LUNES,
    MARTES,
    MIÉRCOLES,
    JUEVES,
    VIERNES,
    SÁBADO
}
```

El primer valor de la enumeración se inicializa a cero. A continuación, las constantes siguientes se inicializan con un incremento de uno. Por lo tanto la declaración anterior se hubiera podido escribir:

```
public class Días
{
    public static final int DOMINGO=0;
    public static final int LUNES=1;
    public static final int MARTES=2;
    public static final int MIÉRCOLES=3;
    public static final int JUEVES=4;
    public static final int VIERNES=5;
    public static final int SÁBADO=6;
}
```

De manera aproximada, esto es lo que hará el compilador cuando analice el código de la 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 otra clase, puede contener atributos, constructores y métodos. El siguiente ejemplo de código presenta estas posibilidades.

```
public enum Daltons
{
   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 Daltons(double altura, double peso)
   {
      this.altura = altura;
      this.peso = peso;
   }
   private double altura() { return altura; }
   private double peso() { return peso; }


   double imc()
   {

      return peso/(altura*altura);
   }

}
```

El constructor se ha utilizado de manera implícita para inicializar las constantes de cada uno de los elementos de la enumeración. El constructor de una enumeración debe, obligatoriamente, declararse como `private`. Existen varios métodos de la clase base (`java.lang.Enum`) que permite obtener

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

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

El método `valueOf` realiza la operación inversa y devuelve uno de los elementos de la enumeración cuyo nombre indica la cadena de caracteres que se pasa como parámetro.

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

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

```
System.out.println("Hermanos Dalton");
for(Daltons d: Daltons.values())
{
System.out.println(d.toString());
}
```

Una vez definida, una enumeración se puede utilizar como un nuevo tipo de datos. Podemos declarar una variable con nuestra enumeración para el tipo.

```
Días referencia;
```

Por lo tanto se puede utilizar la variable asignándole uno de los valores definidos en la enumeración.

```
referencia=Días.LUNES;
```

Al hacer referencia a un elemento de su enumeración, debe estar precedido por el nombre de la enumeración como en el ejemplo anterior. Está prohibido asignar a la variable un tipo distinto a los valores contenidos en la enumeración, y si se intenta se obtiene un error de compilación.

La declaración de una enumeración no se puede llevar a cabo dentro de un procedimiento o de una función. Por el contrario, se puede declarar en una clase pero habrá que prefijar el nombre de la enumeración con el nombre de la clase en la cual se determina su utilización. Para que la enumeración sea autónoma, basta con declararla en su propio fichero.

El alcance de una enumeración sigue las mismas reglas que el de las variables (utilización de las palabras clave `public`, `private`, `protected`).

Una variable de tipo enumeración se puede utilizar fácilmente en una estructura `switch ... case`. En este caso, no es necesario que el nombre de la enumeración preceda a los miembros de la enumeración.

```
public static void testDía(Días d)
     {
          switch (d)
          {
          case LUNES:
          case MARTES:
          case MIÉRCOLES:
          case JUEVES:
               System.out.println("qué duro es trabajar");
               break;
          case VIERNES:
               System.out.println("¡pronto el fin de semana!");
               break;
          case SÁBADO:
               System.out.println("¡por fin!");
               break;
          case DOMINGO:
               System.out.println("¡y vuelta a empezar!");
               break;
          }
     }
```

4. Los arrays

Los arrays nos van a permitir 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 uno. Después de su creación, no está permitido modificar las características del array (número de celdas, tipo de elementos almacenados en la tabla). La manipulación de un array se realiza 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 los símbolos [y] (corchetes) después del tipo de datos o del nombre de la variable. Es preferible, para una mejor legibilidad del código, asociar los caracteres [y] 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. Consiste en definir de forma simultánea la declaración de la variable, la creación del array y la 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.

Utilización del array

Los elementos de los arrays son accesibles de la misma manera que una variable clásica. Sólo es necesario añadir el índice del elemento que se quiere manipular.

```
cifraNegocio[0]=12456;
```

El contenido de una celda de array puede utilizarse exactamente de la misma manera que una variable del mismo tipo. Hay que tener cuidado al manipular un array y no intentar acceder a una celda que no exista, bajo el riesgo de obtener una excepción del tipo `ArrayIndexOutOfBoundException`.

Arrays de varias dimensiones

Los arrays de varias dimensiones son de hecho arrays que contienen otros arrays. La sintaxis de declaración es similar a la de un array excepto que se deben especificar tantos pares de corchetes como dimensiones se desea tener.

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

Su creación también es similar a la de un array de una dimensión excepto que deseamos indicar un tamaño para cada una de las dimensiones.

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

El acceso a un elemento de la matriz se realiza de manera idéntica: indicando los índices que permiten identificar la celda de la matriz en cuestión.

```
matriz[0][0]=99;
```

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},{31,32,33}};
```

Este ejemplo crea una matriz con dos dimensiones de tres celdas por tres.

La creación de matrices de gran tamaño con esta técnica puede ser peligrosa.

Manipulaciones habituales con arrays

Cuando se trabaja con arrays, hay ciertas operaciones que se realizan regularmente. Este párrafo describe las operaciones más corrientes realizadas con arrays. 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.

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

Buscar un elemento en un array: la función `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 esta función realice la búsqueda de forma óptima se debe ordenar el array previamente.

```
int[] cifraNegocio={1234,563,657,453,986,678,564,234,786,123,534,975};
Arrays.sort(cifraNegocio);
System.out.println(Arrays.binarySearch(cifraNegocio, 123));
```

Ordenar un array: la función `sort` ordena el array que recibe como parámetro. La ordenación sigue un orden alfabético para los arrays de cadena de caracteres y un orden ascendente para los de valores numéricos.

```
int[] cifraNegocio={1234,563,657,453,986,678,564,234,786,123,534,975};
Arrays.sort(cifraNegocio);
for (int i=0;i<cifraNegocio.length;i++)
{
      System.out.print(cifraNegocio[i] + "\t");
}
```

Muestra el resultado siguiente:

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

La función `parallelSort` realiza también la ordenación de la tabla, aunque utiliza un algoritmo que explota las capacidades de una máquina multiprocesador.

Mostrar un array: la función `toString` permite obtener una representación con la forma de cadena de caracteres del array que se pasa como parámetro.

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

Muestra el resultado siguiente:

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

La función `deepToString` efectúa la misma operación pero para una matriz.

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

Muestra el resultado siguiente:

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

Copiar un array: están disponibles dos funciones para la copia de arrays.

La función `copyOf` copia un array entero con la posibilidad de modificar su tamaño. La función `copyOfRange` efectúa una copia de una parte del array.

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

Muestra el resultado siguiente:

```
[1234, 563, 657, 453, 986, 678, 564, 234, 786, 123, 534, 975, 0, 0,  0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

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

Muestra el resultado siguiente:

```
[1234, 563, 657]
```

Rellenar un array: la función `fill` se utiliza para rellenar todas las celdas de un array con el mismo valor.

5. Las cadenas de caracteres

Las variables de tipo `String` permiten manipular cadenas de caracteres.

Vamos a ver cómo realizar las operaciones más corrientes en las cadenas de caracteres.

Creación de 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 utilizarlo, 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.

Asignación de un valor a una cadena

Hemos visto que para asignar un valor a una cadena es necesario especificarla entre los caracteres " y ": 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 como en el ejemplo siguiente:

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

Podemos visualizar: Dijo: "¡Basta ya!"

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

Extracción de un carácter particular

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

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 tercer carácter de la cadena1 es " +
cadena1.charAt(2));
```

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");
```

Subcadenas

La función `substring` de la clase `String` devuelve una sección de una cadena 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));
```

Podemos ver lo siguiente:

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

Comparación de cadenas

Cuando hacemos una comparación entre dos cadenas, tendemos a utilizar el doble igual (==), como se realizaba antes. Este operador funciona correctamente con los tipos primitivos pero no podemos olvidar que las cadenas de caracteres son de tipo objeto. 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 `boolean` igual a `true` si las dos cadenas son idénticas y por supuesto un `boolean` 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.

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");
}
```

Por el contrario, para realizar una clasificación tenemos que utilizar el método `compareTo` de la clase `String` o la función `compareToIgnoreCase`. Con estas dos soluciones, hay que pasar como parámetros la cadena que se debe comparar. 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 fichero.

```
String nombre="Código.java";

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

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

Búsqueda 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 ó -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, para esta versión, 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
```

Sustitución en 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 substitución 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
```

Dar formato a una cadena

El método `format` de la clase `String` permite evitar largas y complicadas operaciones de conversión y de concatenación. 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 representador 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

El siguiente ejemplo de código:

```
boolean b=true;
   int i=56;
   double d=19.6;
   String s="cadena";
   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" +
                    "decimal en formato científico: %e%n",
                    b,s,i,i,i,d,d));
```

muestra el siguiente resultado en la consola:

```
booleano: true
cadena de caracteres: cadena
entero: 56
entero en hexadecimal: 38
entero en octal: 70
decimal: 19,600000
decimal en formato científico: 1,960000e+01
```

6. Fecha y hora

La gestión de la fecha y la hora ha sido, durante mucho tiempo, la bestia negra de los desarrolladores Java. La clase `GregorianCalendar` está disponible para responder a las problemáticas de manipulación de fecha y hora. 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 61380284400000 no nos evoca ninguna fecha, decir sin embargo 25/12/2014 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. En lugar de tener una o dos clases dedicadas a dicha gestión, y con las que jugar, han hecho aparición numerosas clases especializadas.

`LocalDate`	Representa una fecha (día, mes, año) sin hora.
`LocalDateTime`	Representa una fecha y una hora sin tener en cuenta el huso horario.
`LocalTime`	Representa una hora sin tener en cuenta el huso horario.
`OffsetDateTime`	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 uso 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(2014, 12,25);
```

- `from`: conversión entre los 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");
```

- `withxxxxxx`: devuelve una nueva instancia modificando la componente indicada por xxxxxx por el valor que se pasa como parámetro.

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

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

```
LocalDate pascua;
   pascua=LocalDate.of(2014,4,20);
   LocalDate ascension;
   ascension=pascua.plusDays(39);
```

- `atxxxxxx`: 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(2014,7,13);

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

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

El pequeño ejemplo de código que se muestra a continuación muestra algunas operaciones sobre las fechas teniendo en cuenta que los días festivos son los sábados y domingos.

```
      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=2014;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");
       }
```

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 sólo en un operando.
- Los operadores binarios que necesitan dos operandos.

Se pueden utilizar los operadores unarios con la notación prefijada. En este caso, el operador se sitúa antes del operando. También la notación puede aparecer postfijada, en cuyo caso, el operador se sitúa después del operando. La posición del operador determina el momento en el que éste se aplica a la variable. Si el operador está prefijado, se aplica en el operando antes que éste sea utilizado en la expresión. Con la notación postfijada del operador se aplica en la variable sólo después de su uso en la expresión. Esta distinción puede influir en el resultado de una expresión.

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

Muestra 3 porque el incremento se ejecuta después de la utilización de la variable con la instrucción `println`.

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

Muestra 4 porque el incremento se ejecuta antes de utilizar la variable con la instrucción `println`.

Si la variable no se utiliza en una expresión, las dos versiones llevan al mismo resultado.

La siguiente línea de código:

```
i++;
```

es equivalente a la línea de código:

```
++i;
```

Se pueden repartir los operadores en siete categorías.

1. Los operadores unarios

Operador	Acción
-	Valor negativo
~	Complemento a uno
++	Incremento
--	Decremento
!	Negación

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

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.

Operador	Operación efectuada	Ejemplo	Resultado
+	Suma para valores numéricos o concatenación para cadenas	6+4	10
-	Sustracción	12-6	6

Operador	Operación efectuada	Ejemplo	Resultado
*	Multiplicación	3*4	12
/	División	25/3	8.3333333333
%	Módulo (resto entero de la división)	25 % 3	1

4. Los operadores bit a bit

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 & 255	45
|	O binario	99 | 46	111
ˆ	O exclusivo	99 ˆ 46	77
>>	Desplazamiento hacia la derecha (división por 2)	26>>1	13
<<	Desplazamiento hacia la izquierda (multiplicación por 2)	26<<1	52

5. Los operadores de comparación

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

Operador	Operación efectuada	Ejemplo	Resultado
==	Igualdad	2 == 5	false
!=	Desigualdad	2 != 5	true
<	Inferior a	2 < 5	true
>	Superior a	2 > 5	false
<=	Inferior o igual a	2 <= 5	true
>=	Superior o igual a	2 >= 5	false
instanceof	Comparación del tipo de la variable con el tipo indicado	O1 instanceof Cliente	True si la variable O1 hace referencia a un objeto creado a partir de la clase Cliente o de una subclase

6. El operador de concatenación

El operador + (más) ya utilizado para la suma se utiliza 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. En realidad 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 `StringBuffer`.

Ejemplo

```
long duración;
  String liebre;
  String tortuga="";
  long principio, fin;
  principio = System.currentTimeMillis();
  for (int i = 0; i <= 10000; i++)
  {
      tortuga = tortuga + " " + i;
  }
  fin = System.currentTimeMillis();
  duración = fin-principio;
   System.out.println("duración para la tortuga: " + duración + "ms");
  principio = System.currentTimeMillis();
  StringBuffer sb = new StringBuffer();
  for (int i = 0; i <= 10000; i++)
  {
      sb.append(" ");
      sb.append(i);
  }
  liebre = sb.toString();
  fin = System.currentTimeMillis();
  duración = 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: 953ms
duración para la liebre: 0ms
las dos cadenas son idénticas
```

¡Este resultado no necesita comentarios!

7. Los operadores lógicos

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

Operador	Operación	Ejemplo	Resultado
&	Y lógico	if ((test1) & (test2))	verdad si test1 y test2 es verdad
|	O lógico	if ((test1) | (test2))	verdad si test1 o test2 es verdad
ˆ	O exclusivo	if ((test1) ˆ (test2))	verdad si test1 o test2 es verdad pero no si los dos son verdaderos al mismo tiempo
!	Negación	if (! Test)	Invierte el resultado del test
&&	Y lógico	if((test1) && (test2))	Igual al Y lógico pero test2 sólo se evaluará si test1 es verdad
||	O lógico	if ((test1) || (test2))	Igual al O lógico pero test2 sólo se evaluará si test1 es falso

Tendremos que 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, ésta sólo se modificará en los casos siguientes:

- primer test verdad en el caso del &&.
- primer test falso en el caso del ||.

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 (*, /)
- División entera (\)
- Módulo (Mod)
- 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) ˆ (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

Las estructuras de control permiten modificar el orden de 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.

1. 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 es verdadera, la instrucción se ejecuta. La condición debe ser una expresión que, una vez evaluada, debe devolver un valor `boolean` `true` o `false`. Con esta sintaxis, sólo la instrucción situada después del `if` será ejecutada si la condición es verdadera. 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 es verdadera.

También podemos especificar una o varias instrucciones que se ejecutarán si la condición es falsa.

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

También podemos anidar las condiciones en la sintaxis.

```
if (condición1)
{
      Instrucción 1
      ...
      Instrucción n
}
else if (condición2)
{
      Instrucción 1
      ...
      Instrucción n
}
else if (condición3)
{
      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.

También existe un operador condicional que permite efectuar un `if ... else` en una sóla instrucción.

```
condición_?_expresión:expresión2;
```

Esta sintaxis equivale a la siguiente:

```
If (condición)
expresión1;
else
expresión2;
```

b. 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 1 se ejecuta.

En caso contrario, se compara el valor obtenido con el valor del `case` siguiente y, 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 en el `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, sólo 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.

```
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");
}
```

2. 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 `true`.

```
while (condición)
{
     Instrucción 1;
     ...
     Instrucción n;
}
```

La condición se evalúa antes del primer ciclo. Si es `false` 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 ; después del `while` porque en este caso, el bloque de código dejará de asociarse con el bucle.

b. Estructura do ... while

```
int i=0;
while (i<10)
{
     System.out.println(i);
     i++;
}
```

```
do
{
     Instrucción 1;
     ...
     Instrucción n;
}
while (condición);
```

Esta estructura tiene un funcionamiento idéntico a la anterior excepto que la condición se examina después de la ejecución del bloque de código. Nos permite garantizar que el bloque de código se ejecuta 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 `true`, el bloque se ejecuta de nuevo hasta que la condición sea `false`. Debemos tener 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: para ello, hace falta evaluar la condición como `true`. 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 evaluada como `true`.

El ejemplo muestra dos bucles `for` en acción para visualizar una tabla de multiplicar.

```
int k;
for(k=1;k<10;k++)
{
    for (int l = 1; l < 10; l++)
    {
       System.out.print(k * l + "\t");
    }
    System.out.println();
}
```

Obtenemos el resultado siguiente:

```
1    2     3     4     5     6     7     8     9
2    4     6     8     10    12    14    16    18
3    6     9     12    15    18    21    24    27
4    8     12    16    20    24    28    32    36
5    10    15    20    25    30    35    40    45
6    12    18    24    30    36    42    48    54
7    14    21    28    35    42    49    56    63
8    16    24    32    40    48    56    64    72
9    18    27    36    45    54    63    72    81
```

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`. La sintaxis general de esta instrucción es la siguiente:

```
for (tipo variable: 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 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. Sólo 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"};
    int cpt;
    for (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.

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;
   st=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);
      f(st.endsWith("3"))
      {
         lst.add("cliente 4");
      }
   }
```

d. Interrupción de una estructura de bucle

Tres instrucciones pueden modificar el funcionamiento normal de las estructuras de bucle.

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");
       }
```

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 `continue` de manera condicional y acepta, a

su vez, el uso de una etiqueta.

He aquí un ejemplo de código que utiliza un bucle sin fin y dos instrucciones para mostrar los números impares hasta que el usuario teclee un retorno de carro.

```
import java.io.IOExcepción;
public class TestEstructuras {
     static boolean stop;
     public static void main(String[] args)
     {
          new Thread()
          {
               public void run()
               {
                    int c;
                    try
                    {
                         c=System.in.read();
                         stop=true;
                    }
                    catch (IOExcepción e)
                    {
                         e.printStackTrace();
                    }
               }
          }.start();
          long contador=0;
          while(true)
          {
               contador++;
               if (contador%2==0)
                    continue;
               if (stop)
                    break;
               System.out.println(contador);
          }
     }
}
```

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.

Ejercicios

Ejercicio 1

Crear una tabla de diez cadenas de caracteres y, a continuación, rellenar esta tabla con direcciones de correo electrónico. Calcular a continuación, 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 de acceso es la parte de la dirección situada después del carácter @ de la dirección de correo electrónico.

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, repita hasta tener la combinación par, par, impar. Mostrar, a continuación, el número de ensayos realizados para obtener dicha combinación.

Pista: la clase `Math` proporciona el método estático `random` que genera un número aleatorio comprendido entre 0 y 1.

Ejemplo: `double num=Math.random();`

Ejercicio 3

Generar un número aleatorio comprendido entre 0 y 1000. Pedir, a continuación, al usuario adivinar 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 `Scanner`. Tenemos a nuestra disposición una serie de funciones que permiten recuperar valores enteros, `float`, cadenas de caracteres... Estas funciones se denominan `nextxxxx`, donde `xxxx` debe remplazarse por el tipo de datos que se espera obtener, por ejemplo `nextInt` para un valor entero, `nextLine` para una cadena de caracteres, etc.

```
String cadena;
Scanner sc;
sc=new Scanner(System.in);
cadena=sc.nextLine();
```

Ejercicio 4

Agregar al ejercicio 3 la representación del tiempo que ha tardado el usuario en adivinar la respuesta correcta.

Correcciones

Corrección del ejercicio 1

Descompongamos el problema.

En primer lugar hay que crear y rellenar la tabla que contiene las direcciones de correo electrónico que queremos procesar. La tabla `direcciones`juega este rol, y se inicializa en las líneas 13 a 23.

A continuación es preciso extraer, de las direcciones, la información relativa al nombre del proveedor. El resultado se almacena en una nueva tabla llamada `listaProveedores`. Este procesamiento se realiza en las líneas 25 a 29.

Esta tabla se analiza, a continuación, para extraer los distintos nombres de proveedor, que se almacenan en una nueva tabla `nombresProveedores`. Las líneas 31 a 48 realizan este trabajo recorriendo la tabla `listaProveedores` y verificando si cada nombre ya está presente en la tabla `nombresProveedores` gracias a la función `busquedaProveedor`. Devuelve el índice de la posición si se encuentra el proveedor, o -1 en caso de no encontrarlo. En este caso, el nombre del proveedor se agrega a la tabla `nombresProveedores`. A continuación, se incrementa el contenido del tamaño correspondiente a la tabla `numCliente`.

Para terminar, se recorre la tabla `nombresProveedores` y se recupera en la tabla `numCliente` el número de veces que aparece el proveedor, para calcular, a continuación, el porcentaje (líneas 49 a 63).

```
1. package ejercicios.capitulo2.ejercicio1;
2.
3. public class Principal
4. {
5.      static String[] direcciones;
6.      static String[] listaProveedores;
7.      static String[] nombresProveedores;
8.      static int[] numCliente;
9.      static int posicion;
10.     static String proveedor;
11.     public static void main(String[] args)
12.     {
13.         direcciones=new String[10];
14.         direcciones[0]="jpp@telefonica.es";
15.         direcciones[1]="tom@gmail.com";
16.         direcciones[2]="fred@telefonica.es";
17.         direcciones[3]="victor@telefonica.es";
18.         direcciones[4]="cristina@telefonica.es";
19.         direcciones[5]="roberto@orange.es";
20.         direcciones[6]="paula@telefonica.es";
21.         direcciones[7]="lisa@gmail.com";
22.         direcciones[8]="tomas@eni.es";
23.         direcciones[9]="maria@eni.es";
24.
25.         listaProveedores =new String[10];
26.         for (int i=0;i<direcciones.length;i++)
27.         {
28. listaProveedores[i]=direcciones[i].substring(direcciones
[i].indexOf('@')+1);
29.         }
30.
31.         numCliente=new int[10];
32.         nombresProveedores=new String[10];
33.         for(int i=0;i<listaProveedores.length;i++)
34.         {
35.             int resultado;
36. resultado=busquedaProveedor(listaProveedores[i]);
37.             if(resultado==-1)
38.             {
39. nombresProveedores[posicion]=listaProveedores[i];
40.                     numCliente[posicion]++;
41.                     posicion++;
42.             }
43.             else
44.             {
45.                     numCliente[resultado]++;
46.             {}
47.
48.             }
49.             for(int i=0;i<posicion;i++)
50.             {
51.                 System.out.println(nombresProveedores[i] + " :
" + (double)numCliente[i]/listaProveedores.length*100 + "%");
52.             }
53.     }
54.
55.     static int busquedaProveedor(String nombre)
56.     {
57.         for(int i=0;i<nombresProveedores.length;i++)
58.         {
59.             if(nombresProveedores[i]!=null &&
nombresProveedores[i].equals(nombre))
60.             {
61.                return i;
62.            }
63.         }
64.         return -1;
65.     }
66. }
```

Corrección del ejercicio 2

Para este ejercicio, debemos utilizar la función **random** para obtener un número aleatorio. Esta función devuelve un número decimal aleatorio comprendido entre 0 y 1. Como queremos obtener números enteros, basta con multiplicar el valor devuelto por la función por 1000 y, a continuación, transformar el resultado en un número entero. Es el objetivo de las líneas 11 a 13. El contador de número de ensayos se incrementa, a continuación, y se generan tres números (líneas 14 y 15). Este conjunto se sitúa en un bucle `do ... while` para volver a comenzar de nuevo tantas veces como sea necesario mientras no se cumplan las condiciones. Si la condición indicada en la instrucción `while` se evalúa como verdadera, entonces el bucle realiza una nueva iteración. Como deseamos detener la ejecución tan pronto como obtengamos un número par, el segundo número par y el tercer número impar, será preciso invertir dicha condición para continuar la iteración. Una vez invertida, se convierte en: continuamos la evaluación si el primer número es impar, o si el segundo es impar o si el tercero es par.

Para verificar la paridad de un número, se divide entre dos y, a continuación, se recupera el resto de dicha división. Si el resto es igual a 0, el número es par, en caso contrario es impar. Esta verificación se realiza en la línea 17. La última línea de código muestra, simplemente, el número de intentos.

```
1. package ejercicios.capitulo2.ejercicio2;
2.
3. public class Principal
4. {
5.     public static void main(String[] args)
6.     {
7.           int contador=0;
8.           int num1,num2,num3;
9.         do
10.             {
11.             num1=(int)(Math.random()*1000);
12.             num2=(int)(Math.random()*1000);
13.             num3=(int)(Math.random()*1000);
14.             contador++;
15.             System.out.println("número 1:" + num1 +
" número 2:" + num2 + " número 3:" + num3);
16.             }
17.             while(num1 % 2==1 || num2 % 2==1 || num3 % 2==0);
18.             System.out.println("Resultado obtenido en "
+ contador + " intento(s)");
19.          }
20.     }
```

Corrección del ejercicio 3

El código de este ejercicio comienza, como es habitual, instanciando los objetos. Las líneas 9 a 14 inicializan el número secreto y el objeto `Scanner`, que permite recuperar los datos introducidos por el usuario.

El bucle `do...while` gestiona, a continuación, el valor entero introducido por el usuario y lo compara con el número secreto para, a continuación, 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.

```
1. package ejercicios.capitulo2.ejercicio3;
2.
3. import java.util.Scanner;
4.
5. public class Principal
6. {
7.     public static void main(String[] args)
8.     {
9.         int numEnsayos=0;
10.        int numero;
11.        int numIntroducido;
12.        numero=(int)(Math.random()*1000);
13.        Scanner sc;
14.        sc=new Scanner(System.in);
15.        do
16.        {
17.              numIntroducido=sc.nextInt();
18.              numEnsayos++;
19.              if(numIntroducido<numero)
20.              {
21.                   System.out.println("es mayor");
22.              }
23.              if(numIntroducido>numero)
24.              {
25.                   System.out.println("es menor");
26.              }
27.        } while (numero!=numIntroducido);
28.        System.out.println("¡Bravo! Resultado obtenido en "
+ numEnsayos + " ensayo(s)");
29.     }
30.   }
```

Corrección del ejercicio 4

Para evolucionar la aplicación anterior vamos a agregar, simplemente, la inicialización de la variable **inicio** con la hora en curso antes de comenzar con el bucle (línea 21). Al finalizar el bucle (y, por tanto, el juego), realizamos la misma acción con la variable **fin** (línea 35). La duración de la partida es la diferencia entre ambas horas. Se calcula mediante la línea 36. A continuación, se convierte en horas, minutos y segundos para mostrar el resultado (líneas 38 a 43).

```
1.package ejercicios.capitulo2.ejercicio4;
2.
3.import java.time.Duration;
4.import java.time.LocalTime;
5.import java.time.OffsetTime;
6.import java.util.Scanner;
7.
```

```
8.public class Principal
9.{
10.    public static void main(String[] args)
11.    {
12.        int numEnsayos=0;
13.        int numero;
14.        int numIntroducido;
15.        OffsetTime inicio;
16.        OffsetTime fin;
17.        Duration tiempo;
18.        numero=(int)(Math.random()*1000);
19.        Scanner sc;
20.        sc=new Scanner(System.in);
21.        inicio=OffsetTime.now();
22.        do
23.        {
24.             numIntroducido=sc.nextInt();
25.             numEnsayos++;
26.               if(numIntroducido<numero)
27.               {
28.                   System.out.println("es mayor");
29.               }
30.          if(numIntroducido>numero)
31.               {
32.                   System.out.println("es menor");
33.               }
34.          } while (numero!=numIntroducido);
35.          fin=OffsetTime.now();
36.          tiempo=Duration.between(inicio,fin);
37.          LocalTime duracion;
38.          duracion=LocalTime.ofSecondOfDay(tiempo.getSeconds());
39.          System.out.println("¡Bravo! Resultado encontrado en " +
numEnsayos +
40.                 " ensayo(s) y " + duracion.getHour() + " hora(s) " +
41.                 duracion.getMinute() + " minuto(s) " +
42.                 duracion.getSecond() + " segundo(s)");
43.        }
44.    }
```

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, vamos a intentar 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 construidos 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.

Por el contrario, si el plano se modifica, todas las puertas fabricadas tras la modificación se beneficiarán de los cambios aportados al plano (con el riesgo de no seguir siendo compatibles con versiones anteriores).

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 sólo 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 de la 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. 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. 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 reflexionan sobre la caja de cambios de seis marchas de su futuro coche retomarán la versión precedente.

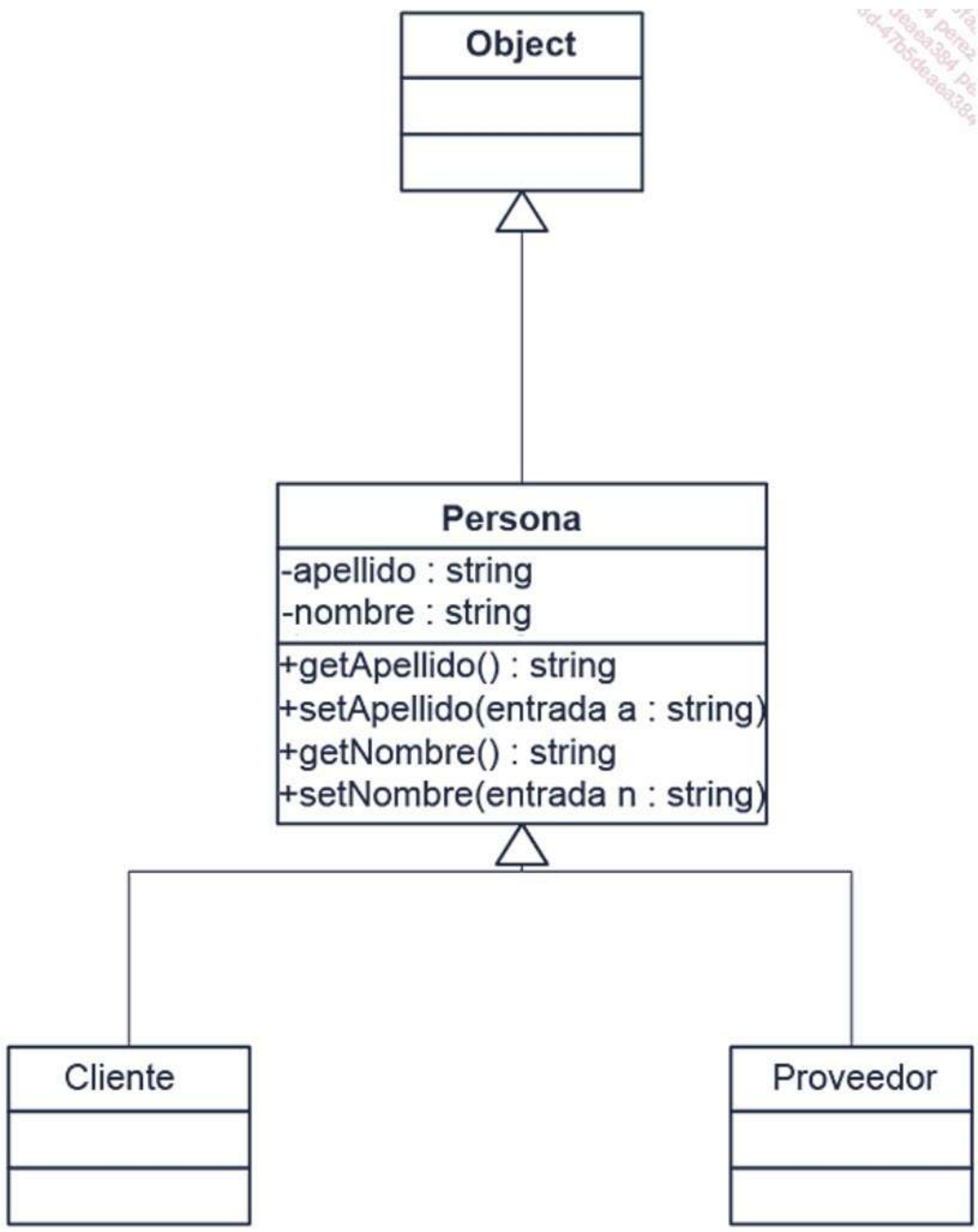
El polimorfismo es otra noción importante en la programación orientada a objetos. Gracias a él, es posible utilizar varias clases de manera intercambiable incluso si el funcionamiento interno de estas clases muestra diferencias. Si usted sabe cambiar la marcha en un coche Peugeot, también sabrá hacerlo 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 sobreescritura 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 sobreescritura 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

En el resto del capítulo, vamos a trabajar con la clase `Persona`, cuya representación UML (*Unified Modeling Language*) 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 en la colección Recursos Informáticos de Ediciones ENI.

1. 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.

```
[lista de 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 cómo se utiliza. 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 sólo podrán utilizarla clases que formen parte del mismo paquete.

`abstract`: indica que la clase es abstracta y no puede ser instanciada. Sólo se la puede utilizar como clase básica en una relación de herencia. En este tipo de clase sólo 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 y sólo 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 nuestra 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 y la extensión .java.

b. Creación de los campos

A continuación, nos vamos a interesar en el contenido de nuestra clase. Debemos crear los diferentes campos 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;
```

La visibilidad de la variable responde a las reglas siguientes:

private: la variable sólo 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 tener siempre variables private o protected, pero nunca public.

La variable debe también tener un tipo. No existe ningún límite en cuanto al tipo de una variable y por lo tanto podemos utilizar tanto los tipos básicos del lenguaje Java tales como int, float, char… como tipos de objetos.

En cuanto al nombre de la variable, debe respetar sencillamente las reglas de nomenclatura (no utilizar palabras clave del lenguaje).

Vemos, a continuación, que la clase Persona tiene la forma siguiente:

```
public class Persona
{
     private String apellido;
     private String nombre;
     private LocalDate fecha_naci;
}
```

c. Creación de métodos

Los métodos son simplemente funciones definidas en el interior de una clase. Se suelen utilizar para manipular los campos de la clase. A continuación se describe la sintaxis general de declaración de un método.

```
[modificadores] tipoDeRetorno nombreDelMétodo ([listaDeLosParámetros])
                              [throws listaExcepción]
{

}
```

Java cuenta con los siguientes modificadores:

private: indica que el método sólo puede utilizarse en la clase donde está definido.

protected: indica que sólo 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.

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 fichero externo escrito en otro lenguaje.

synchronized: indica que el método sólo puede ser ejecutado por un único hilo a la vez.

El tipo de retorno puede ser cualquier tipo de dato, tipo básico del lenguaje o tipo objeto. Si el método no tiene información a 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.

Añadimos dos métodos a nuestra clase `Persona`.

```
public class Persona
{
     private String apellido;
     private String nombre;
     private LocalDate fecha_naci;

     public long calculoEdad()
     {
       return fecha_naci.until(LocalDate.now(),ChronoUnit.YEARS);
     }

     public void visualización()
     {
          System.out.println("apellido: " + apellido);
          System.out.println("nombre: " + nombre);
          System.out.println("edad: " + calculoEdad());
     }
}
```

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.

Para poder crear una función sobrecargada, hace falta que al menos uno de sus elementos cambie respeto a una función ya existente. Como el nombre de la función debe seguir siendo el mismo para poder hablar de sobrecarga, sólo podemos actuar sobre el número de parámetros o su tipo. Por ejemplo, podemos añadir la función siguiente a la clase `Persona`:

```
public void visualización(boolean español)
     {
          if (español)
          {
               System.out.println("apellido: " + apellido);
               System.out.println("nombre: " + nombre);
               System.out.println("edad: " + calculoEdad());
          }
          else
          {
               System.out.println("name: " + apellido);
               System.out.println("first name: " + nombre);
               System.out.println("age: " + calculoEdad());
          }

     }
```

En efecto, posee una firma diferente de la primera función `visualización` 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 visualización(boolean mayúscula)
   {
        if (mayúscula)
        {
             System.out.println("apellido: " + apellido.toUpperCase());
           System.out.println("nombre: " + nombre.toUpperCase());
           System.out.println("edad: " + calculoEdad());
        }
        else
        {
             System.out.println("apellido: " + apellido.toLowerCase());
           System.out.println("nombre: " + nombre.toLowerCase());
           System.out.println("edad: " + calculoEdad());
        }
   }
```

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. Para simplificar la llamada a este tipo de función, podemos utilizar la declaración siguiente para indicar que una función espera un número indefinido de parámetros.

```
     public void visualización(String...colores)
     {
       {
          if (colores==null)
          {
               System.out.println("ningun 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.visualización("rojo");
p.visualización("verde","azul","rojo");
p.visualización();
```

Sólo hay una pequeña anomalía en el momento de la ejecución, porque la última llamada de la función visualización 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.visualización(null);
```

En el momento de la llamada a una función, los parámetros se pasan por valores tanto para los tipos básicos del lenguaje (`int`, `float`, `boolean`…) como para los tipos objetos. Sin embargo, si un objeto se pasa como parámetro a una función, el código de la función tiene acceso a los campos del objeto y puede por lo tanto modificar los valores. Por el contrario, si el código de la función modifica la referencia hacia el objeto, ésta se restablecerá después del retorno de la función.

Hay que resaltar que todo este comportamiento es uniforme incluso si los campos de la clase están declarados como `private`, porque estamos en el interior de la clase.

d. Los métodos 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 sólo 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 a 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 sólo lectura, el accesor `set` no debe estar disponible; si un campo debe ser de sólo escritura, entonces se debe omitir la función `get`.

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 fecha_naci;

     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();
     }
}
```

A continuación, los campos de la clase son accesibles desde el exterior mediante estos métodos.

```
          p.setApellido("garcía");
          p.setNombre("josé");
          System.out.println(p.getApellido());
          System.out.println(p.getNombre());
```

e. 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`. No se le llama nunca de manera explícita en el código sino de manera implícita tras 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. Por el contrario, si su clase contiene un constructor sobrecargado, debe también poseer un constructor por defecto. 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()
     {
          apellido="";
          nombre="";
          fecha_naci=null;
     }
public Persona(String a,String n,LocalDate f)
     {
          apellido=a;
          nombre=n;
          fecha_naci=f;
     }
```

Los destructores son otros métodos particulares de una clase. Al igual que los constructores, se invocan implícitamente pero únicamente 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, sólo 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
     {

     }
```

El código presente en el destructor debe permitir la liberación de recursos utilizados por la clase. En él podemos encontrar, por ejemplo, código que cierra un archivo abierto por la clase o el cierre de una conexión hacia un servidor de base de datos. Veremos con detalle, en la sección Destrucción de una instancia, las circunstancias en las cuales se llama al destructor.

f. 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. También, en algunos lenguajes 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 sólo 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 código haciendo referencia a ellos por el nombre de la clase o gracias a una instancia de la clase. No se aconseja esta segunda solución ya que no demuestra el hecho de que estamos trabajando con un miembro estático. El compilador genera una advertencia si detecta dicha situación.

Los métodos estáticos siguen las mismas reglas y pueden ser útiles en la creación de bibliotecas de funciones. El ejemplo clásico es la clase `Math` ya que cuenta con un gran número de funciones estáticas. Los métodos estáticos poseen sin embargo una limitación, y es que sólo pueden utilizar variables locales u otros miembros estáticos de la clase. No pueden usar miembros de instancia de una clase porque puede ocurrir que el método se utilice sin que exista una instancia de la clase.

El compilador detectará esta situación y lo indicará: `non-static variable cannot be referenced from a static context`.

Los miembros estáticos deben declararse con la palabra clave `static`. Como para cualquier otro miembro de una clase, podemos especificar una visibilidad. En cambio, una variable local a una función no puede ser estática.

Para ilustrar la utilización de los miembros estáticos, vamos a añadir a la clase `Persona` un campo numérico. El valor de este campo se forma automáticamente tras la creación de cada instancia de la clase y será único para cada instancia. Los constructores de nuestra clase están perfectamente adaptados para llevar a cabo este trabajo. En cambio, tenemos que memorizar cuántas instancias se han creado para poder asignar un único número a cada instancia. Una variable estática privada se encargará de esta operación. A continuación, le presentamos el código correspondiente.

```
public class Persona
{
    private String apellido;
    private String nombre;
    private LocalDate fecha_naci;
    // campo privado representando el número de la Persona
    private int numero;
    // campo estático privado representando el contador de Personas
    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()
    {
         apellido="";
         nombre="";
         fecha_naci=null;
         //  creación de una nueva Persona y por lo tanto
incrementación del contador
         numInstancias++;
         // asignación a la nueva Persona de su número
         numero=numInstancias;
    }
}
```

g. 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. Se pueden aplicar a una clase, a un campo, o a un método. Debe especificarse antes del elemento al cual se refiere. Una anotación viene precedida por el símbolo @ y está seguida del nombre de la anotación. El compilador reconoce tres tipos de anotaciones que van a permitir la modificación de su comportamiento en tiempo de compilación.

`@Deprecated` se utiliza para indicar que un método ya no debe ser utilizado. 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 `visualización` 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
{
private String apellido;
private String nombre;
private LocalDate fecha_naci;
private int número;
private static int numInstancias;

...

@Deprecated
public void visualización()
{
          System.out.println("apellido: " + apellido);
          System.out.println("nombre: " + nombre);
          System.out.println("edad: " + calculoEdad());
}

...
}
```

La compilación de una clase que contiene una llamada al método `visualización` de la clase `Persona` genera un aviso en la línea que contiene esta llamada.

```
javac -Xlint:deprecation Main.java
Main.java:16: warning: [deprecation] visualización() in
Persona has been deprecated
               p.visualización();
               ^
1 warning
```

Para obtener un mensaje detallado sobre los avisos, hay que utilizar la opción -Xlint:deprecation en el momento de la llamada del compilador.

`@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 `calculoEdad` 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 long calculoEdad()
     {
          ...
          ...
```

```
    }
}
```

Esta clase se compiló sin problema ya que el método `calculoEdad` 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 long calculoEdad(int unidad)
    {
        ...
        ...

    }
}
```

Obtenemos la respuesta siguiente por parte del compilador.

```
Client.java:6: method does not override or implement a method
from a supertype
@Override
^
1 error
```

Por supuesto tiene razón (¡hay que reconocer que siempre lleva razón!), le hemos anunciado nuestra intención de sustituir el método `calculoEdad`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 se compila pero en este caso se trata de una sobrecarga.

`@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.visualización();
    }
```

Este código se compilará correctamente sin ningún aviso, hasta que el método en cuestión desaparezca completamente de la clase correspondiente. En efecto, tenemos que 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.

2. 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`. 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 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(1956,12,13));
```

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 fecha_naci=LocalDate.of(1963,11,29);
     ...
     ...
}
```

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 igual o superior a 1000.

```
public class Persona
{
     private String apellido="nuevoApellido";
     private String nombre="nuevoNombre";
     private LocalDate fecha_naci=LocalDate.of(1963,11,29);
     private int número=0;
     private static int numInstance;

     static
     {
          while(numInstance<1000)
          {
               numInstance=(int)(10000*Math.random());
          }
     }

     ...
     ...
}
```

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 fecha_naci=LocalDate(1963,11,29);
     private int número=0;
     private static int numInstance=inicContador();

     private static int inicContador()
     {
          int cpt=0;
          while(cpt<1000)
          {
               cpt=(int)(10000*Math.random());
          }
          return cpt;
     }
...
...
}
```

Esta solución presenta la ventaja 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. Afortunadamente, 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, éstos 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. Cuanto 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 intermediaria consiste en repartir los objetos en la memoria según su esperanza de vida o su edad.

Por lo tanto, la mitad de la memoria disponible resulta a veces dividida en tres zonas:

- Una zona para los objetos que tienen 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 de las dos utiliza más a menudo la máquina virtual ya que la implementación del Garbage Collector se deja a la libre elección del diseñador de la máquina virtual.

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 estos recursos han alcanzado un umbral limite (alrededor del 85 %). Sin embargo, es posible forzar la activación del Garbage Collector invocando al 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.

```
import java.util.GregorianCalendar;
public class Persona
{
     private String apellido="nuevoApellido";
     private String nombre="nuevoNombre";
     private LocalDate fecha_naci=LocalDate.of(1963,11,29);
     private int numero=0;
     private static int numInstancia;
     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();
     }

     @Override
     protected void finalize() throws Throwable
     {
          System.out.print("\u2020");
          super.finalize();
     }
     public int getNumero()
     {
          return numero;
     }

     public static int getNumInstancias()
     {
     return numInstancia;
     }
```

```
     public Persona()
     {
          apellido="";
          nombre="";
          fecha_naci=null;
          numInstancia++;
          numero=numInstancia;
     }
     public Persona(String a,String n,LocalDate f)
     {
          apellido=a;
          nombre=n;
          fecha_naci=f;
          numInstancia++;
          numero=numInstancia;
     }
}

/*****************************************************************/
import java.time.LocalDate;


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));
          }
}
}
```

3. Herencia

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`, `Comando`.

Probamos la relación « es un tipo de » para cada una de las clases.

- Un comando es un tipo de cliente.
- Un comando es un tipo de persona.
- Un cliente es un tipo de comando.
- Un cliente es un tipo de persona.
- Una persona es un tipo de cliente.
- Una persona es un tipo de comando.

De entre todos estos intentos, sólo 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, sólo 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).
- 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` que hereda de la clase `Persona` y a la cual se añade el campo tipo y los métodos de acceso correspondientes.

```
public class Client extends Persona
{
     // determinación del tipo de cliente
     // P -> particular
     // E -> empresa
     // A -> administración
     private char tipo;
     public char getTipo()
     {
          return tipo;
     }
     public void setTipo(char 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.setFecha_naci(LocalDate.of(1981,05,15));
     c.setType('E');
     c.visualización();
```

a. 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 `visualización` para tener en cuenta el nuevo campo disponible en la clase.

```
public void visualización()
{
     System.out.println("apellido: " + getApellido());
     System.out.println("nombre: " + getNombre());
     System.out.println("edad: " + calculoEdad());
     switch (tipo)
     {
     case 'P':
          System.out.println("tipo de cliente: Particular");
          break;
     case 'E':
          System.out.println("tipo de cliente: Empresa");
          break;
     case 'A':
          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. ¿Por qué no volver a utilizarla en el método `visualización` de la clase `Cliente` ya que heredamos de ella?

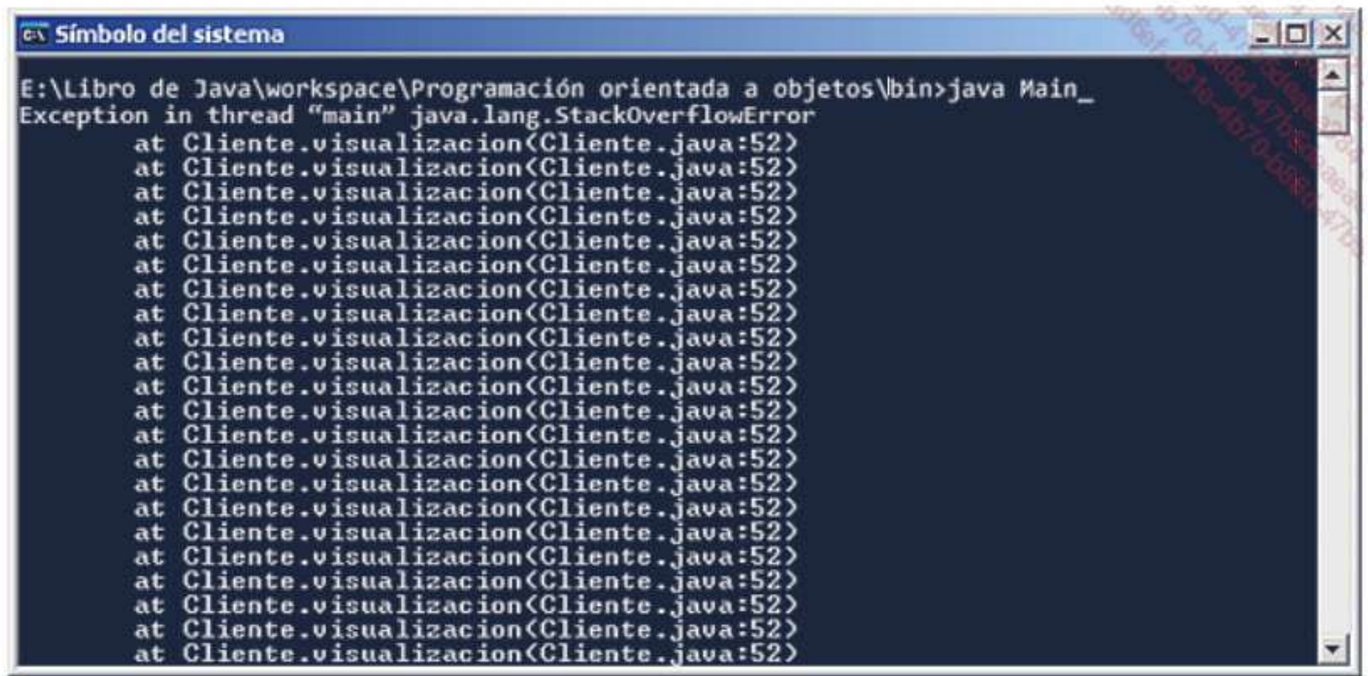
Así nuestro método se convierte en lo siguiente:

```
public void visualización()
{
     visualización();
     switch (tipo)
     {
     case 'P':
          System.out.println("tipo de cliente: Particular");
          break;
     case 'E':
          System.out.println("tipo de cliente: Empresa");
          break;
     case 'A':
          System.out.println("tipo de cliente: Administración");
          break;
     default:
          System.out.println("tipo de cliente: Desconocido");
          break;
     }
}
```

Intentemos utilizarlo:

¡Desafortunadamente, el resultado no es el esperado!

```
Cliente c;
c=new Cliente();
c.setApellido("ENI");
c.setNombre("");
c.setFecha_naci(LocalDate.of(1981,05,15));
c.setTipo('E');
c.visualización();
```

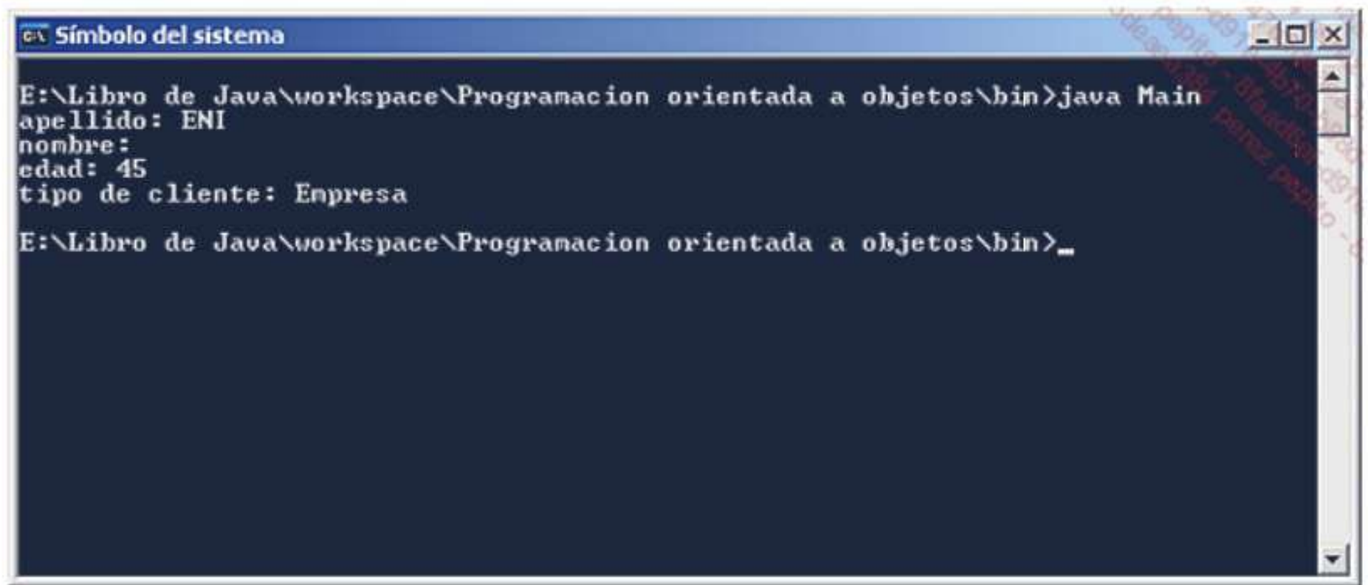


¿Qué ha ocurrido durante la ejecución?

Tras la llamada al método `visualización`, la primera línea de código se limita a invocar al método `visualización` de la clase base. En realidad, la máquina virtual Java ejecuta el primer método `visualización` 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 `visualización` de la clase `Cliente` y, de ahí, el error de desbordamiento de pila que obtenemos. Para evitar este tipo de problemas es preciso indicar que el método `visualización` que debe invocar se encuentra en la clase base, utilizando para ello la palabra clave `super` que apunta al método `visualización` invocado.

```
public void visualización()
{
     super.visualización();
     switch (tipo)
     {
     case 'P':
          System.out.println("tipo de cliente: Particular");
          break;
     case 'E':
          System.out.println("tipo de cliente: Empresa");
          break;
     case 'A':
          System.out.println("tipo de cliente: Administración");
          break;
     default:
          System.out.println("tipo de cliente: Desconocido");
          break;
     }
}
```

Tras realizar esta modificación, todo vuelve al orden, y nuestro código muestra:



La misma palabra clave se puede utilizar para invocar al método constructor de la clase base.

Dicha llamada, de estar presente, debe encontrarse en la primera línea de constructor de la clase derivada.

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. Éste sólo tendrá que invocar al constructor por defecto de la clase básica. De este modo, tenemos que añadir el código siguiente a la clase Cliente .

Comprobamos que el nuevo constructor funciona.

```
public Cliente()
{
     // llamada al constructor por defecto de la super clase
     super();
}
public Cliente(String apellido,String nombre,LocalDate
fecha_naci,char type)
{
     // llamada al constructor sobrecargado de la super clase
     super(apellido,nombre,fecha_naci);
     // inicialización del tipo de cliente
     type=type;
}
```

```
c=new Cliente("ENI","",LocalDate.of(1963,05,15),'E');
c.visualización();
```

Nos muestra lo siguiente:

```
apellido: ENI
nombre:
edad: 45
tipo de cliente: Desconocido
```

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 es nada ilegal, pero no es para nada 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 Client(String apellido,String nombre,LocalDate
fecha_naci,char type)
{
     // llamada al constructor sobrecargado de la super clase
     super(apellido,nombre,fecha_naci);
     this.type=type;
}
```

Nuestro código de prueba muestra ahora la información correcta:

```
apellido: ENI
nombre:
edad: 45
tipo de cliente: Empresa
```

b. Clases abstractas

Las clases abstractas son clases que sólo 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 que deben tener todas unas mínimas características idénticas. Pueden contener campos, propiedades 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 usuario 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 coma.

Una clase abstracta puede contar con métodos no abstractos, pero si contiene un método abstracto debe obligatoriamente ser abstracta también. 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();
}
```

c. 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 Java.

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

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

También se puede declarar un método con la palabra clave `final` para impedir su redefinición en una subclase.

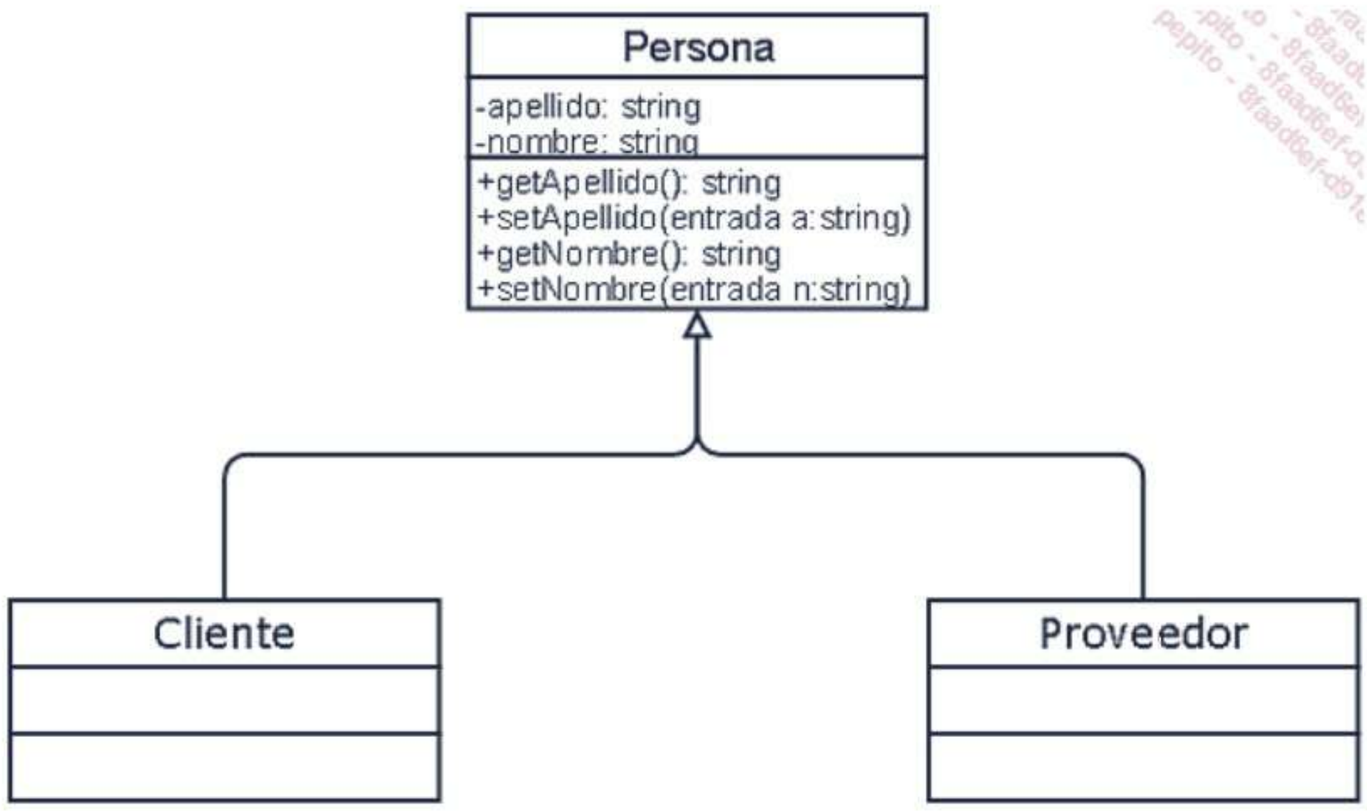
El significado de las palabras clave `abstract` y `final` es algo contradictorio, y está prohibido utilizar ambas palabras clave de manera simultánea. El compilador detecta dicha situación y genera un error.

```
Administrador: Símbolo del sistema
C:\java8\code\tests\src>javac Test.java
Test.java:2 illegal combination of modifiers: abstract and final
public abstract final class Test {
                        ^
1 error

C:\java8\code\tests\src>_
```

d. Conversión de tipo

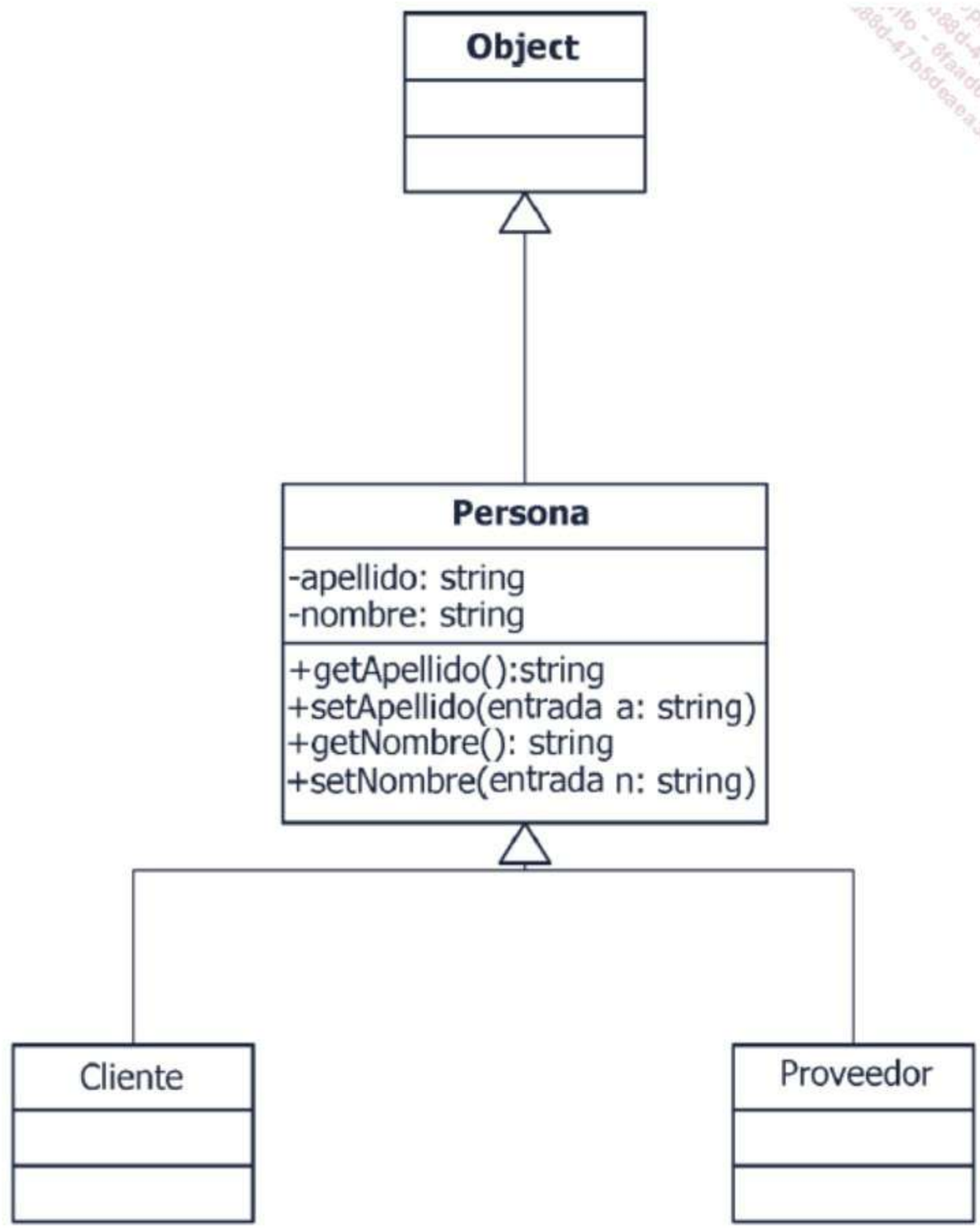
Para entender correctamente las operaciones de conversiones de tipo, hay que tener en mente que, una vez creado en memoria, un objeto no cambiará de tipo hasta el final de su vida. Así, si creamos una instancia de la clase `Persona`, dicha instancia siempre será una instancia de la clase `Persona`. Con 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.

Para ayudarle, hágase siempre la pregunta: lo que puedo hacer con una variable de tipo X, ¿está disponible en la instancia de clase a la que hago referencia?

```
pr=new Proveedor();
o=pr;                    □ funciona     □ no funciona
p=pr;                    □ funciona     □ no funciona
c=pr;                    □ funciona     □ no funciona

c=new Cliente();
o=c;                     □ funciona     □ no funciona
p=c;                     □ funciona     □ no funciona
pr=c;                    □ funciona     □ no funciona

p=new Persona();
o=p;                     □ funciona     □ no funciona
c=p;                     □ funciona     □ no funciona
pr=p;                    □ funciona     □ no funciona

o=new Object();
p=o;                     □ funciona     □ no funciona
pr=o;                    □ funciona     □ no funciona
c=o;                     □ funciona     □ no funciona
```

Aquí está la solución:

```
pr=new Proveedor();
o=pr;                    ■ funciona     □ no funciona
p=pr;                    ■ funciona     □ no funciona
c=pr;                    □ funciona     ■ no funciona

c=new Cliente();
o=c;                     ■ funciona     □ no funciona
```

```
p=c;                    ■ funciona     □ no funciona
pr=c;                   □ funciona     ■ no funciona

p=new Persona();
o=p;                    ■ funciona     □ no funciona
c=p;                    □ funciona     ■ no funciona
pr=p;                   □ funciona     ■ no funciona

o=new Object();
p=o;                    □ funciona     ■ no funciona
pr=o;                   □ funciona     ■ no funciona
c=o;                    □ 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.

```
          ...
          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
     }
```

Aquí está la solución de este segundo ejercicio:

```
...
          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 a 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. En cambio, en tiempo de ejecución, obtenemos una excepción del tipo `java.lang.ClassCastException`. Esta excepción la provoca la máquina virtual cuando descubre que le hemos mentido 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. Por lo tanto se debe escribir la función `prueba` de la manera siguiente:

```
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.

e. La clase Object

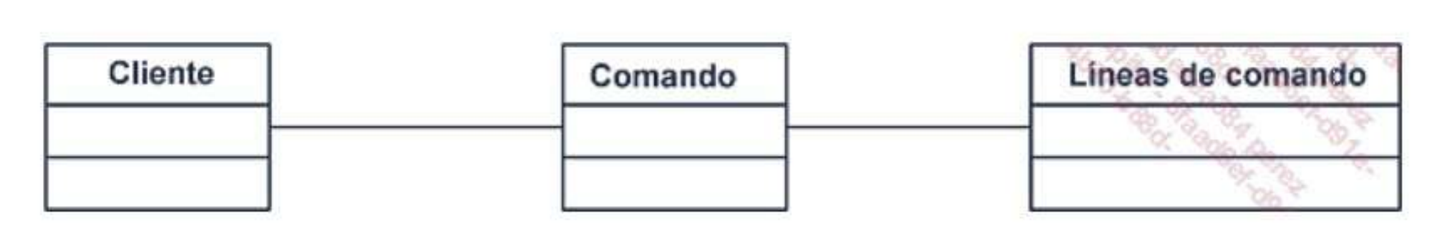
La clase `Object` está directa o indirectamente presente en la jerarquía de todas las clases de una aplicación. Los métodos definidos en la clase `Object` están por lo tanto disponibles para cualquier clase. El inconveniente de estos métodos es que no hacen nada útil o su funcionamiento no está adaptado 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, no obstante a nivel interno es muy utilizado por otros métodos, en particular por el método `equals`.

clone

El método `clone` puede reivindicar el título de "fotocopiador" de objetos. Mediante este método podemos obtener una copia conforme 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 a duplicar. Si la instancia a 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 `Comando`, que a su vez está asociada a una clase `LíneasDeComando`.



Si realizamos una copia de un comando, conservamos la referencia hacia el cliente pero, en cambio, es preciso duplicar las líneas de comando. Debemos diseñar el método `clone` de la clase `Comando` para que duplique también la instancia de la clase `LíneasDeComando` referenciada. A continuación presentamos un extracto de código de estas clases.

```
public class Comando implements Cloneable
{
     Cliente elCliente;
     LíneasDeComando lasLíneas;

     public Comando()
     {
          super();
          lasLíneas=new LíneasDeComando();

     }
     public Object clone() throws CloneNotSupportedException
     {
          Comando cmd;
          // creación de una copia del comando
          cmd=(Comando)super.clone();
          // duplicación de las líneas del comando
          cmd.lasLíneas=(LíneasDeComando)lasLíneas.clone();
          return cmd;

     }
     public Cliente getElCliente()
     {
          return elCliente;
     }
```

```
     public void setElCliente(Cliente elCliente)
     {
          this.elCliente = elCliente;
     }
     public LíneasDeComando getLasLíneas()
     {
          return lasLíneas;
     }
     public void setLasLíneas(LíneasDeComando lasLíneas)
     {
          this.lasLíneas = lasLíneas;
     }
}

/*****************************************************************/
public class LíneasDeComando implements Cloneable
{
     public Object clone() throws CloneNotSupportedException
     {
          return super.clone();
     }
}
```

A continuación, podemos crear instancias de nuestras clases y comprobar su correcto funcionamiento.

```
Cliente c;
c=new Cliente("ENI","",new GregorianCalendar(1981,05,15),'E');
Comando cmd1,cmd2;
// creación e inicialización de un comando
cmd1=new Comando();
cmd1.setElCliente(c);
System.out.println("hashCode del comando: " +cmd1.hashCode());
System.out.println("hashCode del Cliente: "
+ cmd1.getElCliente().hashCode());
System.out.println("hashCode de las líneas: "
+ cmd1.getLasLíneas().hashCode());
cmd2=(Comando)cmd1.clone();
System.out.println("hashCode de la copia: " +cmd2.hashCode());
System.out.println("hashCode del Cliente de la copia: "
+ cmd2.getElCliente().hashCode());
System.out.println("hashCode de las líneas de la copia:"
+ cmd2.getLasLíneas().hashCode());
```

Obtenemos el resultado siguiente en el momento de la ejecución de este código:

```
hashCode del comando: 6413875
hashCode del Cliente: 21174459
hashCode de las líneas: 827574
hashCode de la copia: 17510567
hashCode del Cliente de la copia: 21174459
hashCode de las líneas de la copia: 27744459
```

Tenemos dos comandos distintos que hacen referencia al mismo cliente.

equals

Se utiliza el método `equals` para comparar dos instancias de clase. El código siguiente permite verificar la igualdad de dos clientes.

```
Cliente c1,c2;
c1=new Cliente("ENI","",LocalDate.of(1981,05,15),'E');
c2=new Cliente("ENI","",LocalDate.of(1981,05,15),'E');
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. En realidad se comparan los `hashCode` de 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.

```
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(getApellido())&
                      c.getNombre().equals(getNombre()) &
                      c.getFecha_naci().equals(getFecha_naci()) &
                      c.getTipo()== getTipo() )
            {
                 return true;
            }
            else
            {
                 return false;
            }
       }
}
```

Para conservar una coherencia entre los métodos `equals` y `hashCode`, es importante redefinir este último para que calcule el `hashcode` desde el valor de los campos utilizados en los criterios de comparación. El valor devuelto por esta función tiene poca importancia. Realmente sólo cuenta la garantía de obtener siempre el mismo resultado para cualquier instancia de clase que tenga los mismos valores de campos.

A continuación se muestra una posible implementación del método `hashCode`:

```
public int hashCode()
{

     return this.getApellido().hashCode()+
            this.getNombre().hashCode() +
            this.getFecha_naci().hashCode() +
            (int)this.getTipo();

}
```

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.getPaquete().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(")");
     }
}
```

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.

```
public String toString()
{
     String cadena;
     cadena="apellido: " + getApellido()+ "\r\a";
     cadena=cadena + "nombre: " + getNombre();
     return cadena;
}
```

La llamada al método `toString` está a veces implícita cuando se pasa a una función un objeto como parámetro. Las dos sintaxis siguientes son por lo tanto equivalentes.

```
Cliente c;
c=new Cliente("ENI","",LocalDate.of(1981,05,15),'E');
```

```
System.out.println(c);
```

o

```
Cliente c;
c=new Cliente("ENI","",LocalDate.of(1981,05,15),'E');
System.out.println(c.toString());
```

4. 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, resulta más práctico utilizar las interfaces. Como las clases, las interfaces permiten definir un conjunto de constantes y métodos. Generalmente, 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 Métodos 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.

Creamos nuestra primera interfaz. Ésta 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.

```
// 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;

}
```

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. Por ejemplo, en nuestra clase `Persona`, podríamos implementar la interfaz `Clasificable` de la manera siguiente, al elegir comparar dos instancias de la clase `Persona` mediante el apellido:

```
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 en base al 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 rutina de clasificación genérica de manera que los elementos que se utilizan sean clasificables mediante esta rutina.

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. Sólo 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 void clasificar(Clasificable[] array)
{
}
```

Así definida, nuestra 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;
     for (i=0;i< array.length;i++)
     {
        for( j = i + 1; j<array.length;j++)
        {
            if (array[j].compare(array[i])==Clasificable.INFERIOR)
            {
                c = array[j];
                array[j] = array[i];
                array[i] = c;
            }
            else if (array[j].compare(array[i])==Clasificable.ERROR)
            {
                 return null;
            }
        }
     }
     return array;
}
```

A continuación, para comprobar nuestro procedimiento, vamos a crear e intentar ordenar algunos clientes, para luego visualizar sus apellidos.

```
Persona[] tab;
tab=new Persona[5];
tab[0] = new Persona("juanito2", "nombre2",new
GregorianCalendar(1922,2,15));
tab[1] = new Persona("juanito1", "nombre1 ",new
GregorianCalendar(1911,1,15));
tab[2] = new Persona("juanito5", "nombre5 ",new
GregorianCalendar(1955,05,15));
tab[3] = new Persona("juanito3", "nombre3 ",new
GregorianCalendar(1933,03,15));
tab[4] = new Persona("juanito4", "nombre4 ",new
GregorianCalendar(1944,04,15));
Persona[] tabClasificación;
tabClasificación=(Persona[])clasificar(tab);
   for (int i=0;i<tabClasificación.length;i++)
   {
     System.out.println(tabClasificación[i]);
   }
```

Obtenemos el resultado siguiente:

```
Sr juanito1 nombre1  nacido el 01/01/0001 00:00:00 código Cliente: 1
Sr juanito2 nombre2  nacido el 01/01/0001 00:00:00 código Cliente: 2
Sr juanito3 nombre3  nacido el 01/01/0001 00:00:00 código Cliente: 3
Sr juanito4 nombre4  nacido el 01/01/0001 00:00:00 código Cliente: 4
Sr juanito5 nombre5  nacido el 01/01/0001 00:00:00 código Cliente: 5
```

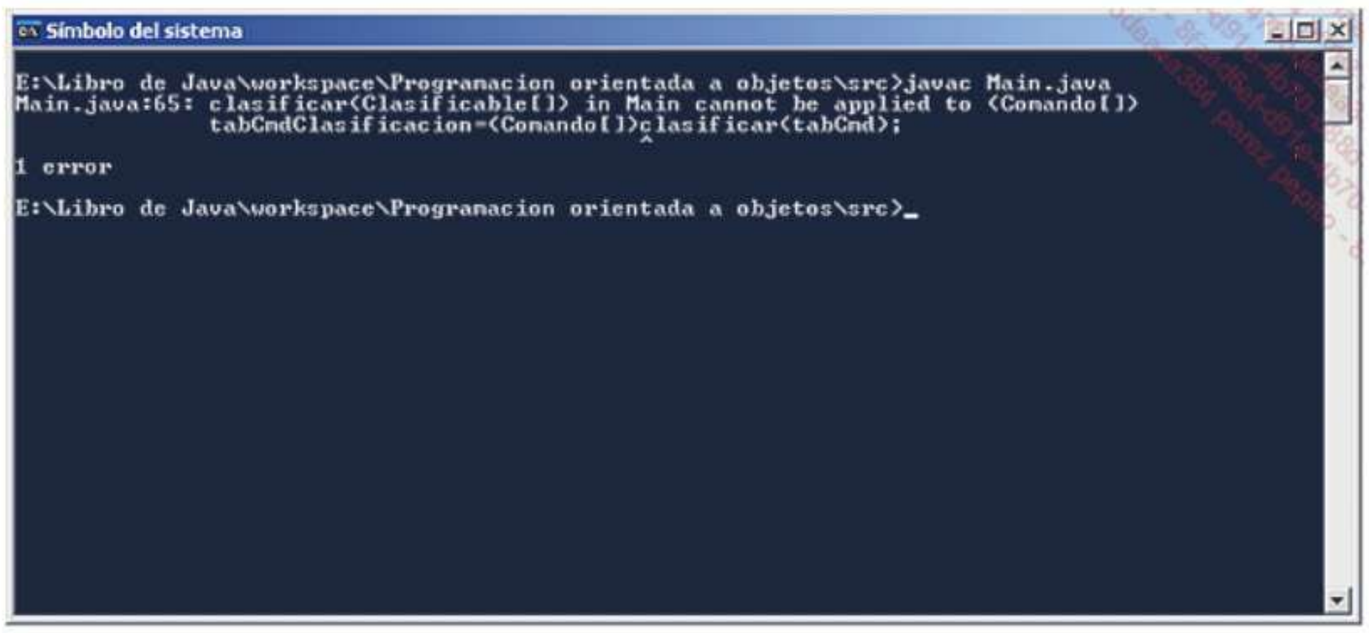
Tenemos la lista de nuestros clientes ordenados por orden alfabético de su apellido.

Intentemos emplear nuestro procedimiento de clasificación con un array de objetos que no implementa la interfaz `Clasificable`.

```
      Comando[] tabCmd;
      tabCmd=new Comando[5];
      tabCmd[0] = new Comando();
      tabCmd[1] = new Comando();
      tabCmd[2] = new Comando();
```

```
tabCmd[3] = new Comando();
tabCmd[4] = new Comando();
Comando[] tabCmdClasificación;
tabCmdClasificación=(Comando[])clasificación(tabCmd);
for (int i=0;i<tabCmdClasificación.length;i++)
{
        System.out.println(tabCmdClasificación[i]);
}
```

En tiempo de compilación, las cosas se complican.



Este error se produce cuando se invoca al procedimiento de clasificación. Los elementos del array, que hemos pasado como parámetro, no implementan la interfaz `Clasificable` y no estamos seguros de que contengan una función `compare`. Destaquemos que, incluso si existe una función `compare` correcta en la clase `Comando`, se debe especificar obligatoriamente que esta clase implementa la interfaz `Clasificable` para que el código pueda funcionar.

c. Métodos por defecto

Ahora que nuestro código funciona correctamente, podemos tratar de 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.

```
// 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);
   boolean isInferior(Object o);
   boolean isSuperior(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;

}
```

Agregar estos dos métodos provoca, de inmediato, un problema en tiempo de compilación de la clase `Persona` y, en general, en todas las clases 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 siendo 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.

```
// 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);
   default boolean isInferior(Object o)
   {
      return false;
   }
   default boolean isSuperior(Object o)
   {
      return false;
   }

   public static final intINFERIOR=-1;
   public static final intIGUAL=0;
   public static final intSUPERIOR=1;
   public static final intERROR=-99;

}
```

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 proveer su propia implementación de dichos métodos. Con esta nueva definición de la interfaz, la clase `Persona` no requiere ser modificada. 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 Coche()
   {
      super();
   }

   public Coche(String matricula,String marca,String
modelo,int potencia)
   {
      this.matricula=matricula;
      this.marca=marca;
      this.modelo=modelo;
      this.potencia=potencia;
   }

   public String getMatricula()
   {
      return matricula;
   }

   public void setMatricula(String matricula)
   {
      this.matricula = matricula;
   }
   public String getMarca()
   {
      return marca;
   }
   public void setMarca(String marca)
   {
      this.marca = marca;
   }
   public String getModelo()
   {
      return modelo;
}

   public void setModelo(String modelo)
   {
      this.modelo = modelo;
   }
   public int getPotencia()
   {
      return potencia;
   }

   public void setPotencia(int potencia)
   {
      this.potencia = potencia;
```

```
    }

    @Override
    public int compare(Object o)
    {
        Coche c;
        if (o instanceof Coche)
        {
            c=(Coche)o;
        }
        else
        {
            return Clasificable.ERROR;
        }
        if (getPotencia()<c.getPotencia())
        {
            return Clasificable.INFERIOR;
        }
        if (getPotencia()>c.getPotencia())
        {
            return Clasificable.SUPERIOR;
        }

        return Clasificable.IGUAL;
    }

    @Override
    public boolean isInferior(Object o)
    {
        Coche c;
        if (o instanceof Coche)
        {
            c=(Coche)o;
        }
        else
        {
            return false;
        }
        if (getPotencia()<c.getPotencia())
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    @Override
    public boolean isSuperior(Object o)
    {
        Coche c;
        if (o instanceof Coche)
        {
            c=(Coche)o;
        }
        else
        {
            return false;
        }
        if (getPotencia()>c.getPotencia())
        {
            return true;
        }
        else
        {
            return false;
        }
    }

}
```

5. 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 el 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 `container`.
- Puede utilizarse (instanciarse) sin que exista una instancia de su clase `container`.
- Puede utilizar los miembros de instancia de su clase `container` únicamente mediante una instancia de dicha clase.

Aquí tenemos un ejemplo sencillo de clase anidada estática.

```
public class Externa
{
    static class Interna
```

```
    {
        public double calculoNeto(double precio)
        {
            return precio*tasa;
        }
    }
    static double tasa=1.21;
}
```

Se puede utilizar de manera muy parecida a cualquier otra clase. La única obligación reside en el nombre utilizado para hacer referencia en el código, que debe ir precedido del nombre de la clase `container`.

```
Externa.Interna ci;
ci=new Externa.Interna();
System.out.println(ci.calculoNeto(100));
```

b. Clases internas

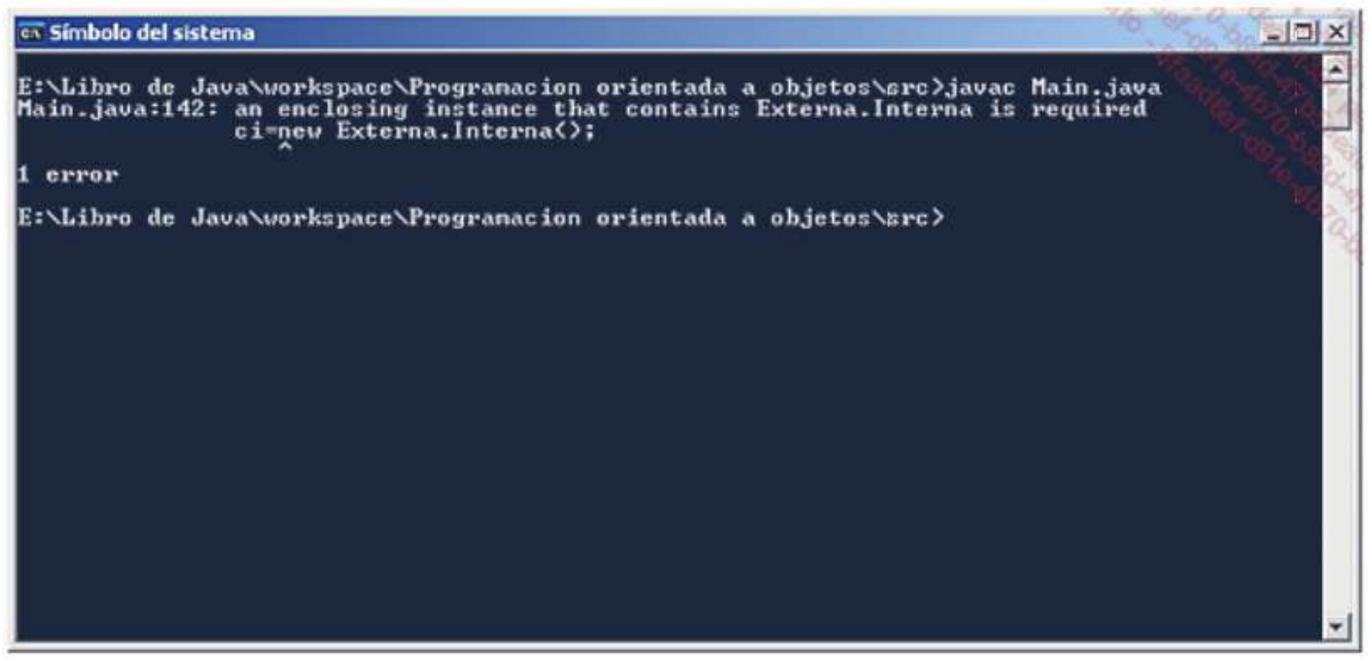
Las clases anidadas se conocen también con el nombre de clases internas. Se las somete a las mismas reglas que cualquier otro elemento declarado en una clase.

- Pueden tener acceso a todos los miembros de la clase en la cual están declaradas, incluso los miembros privados.
- Sólo se las puede emplear si hay disponible alguna instancia de la clase `container`.

La declaración es muy sencilla y similar a la anterior, salvo en la palabra clave `static`, que desaparece.

```
public class Externa
{
     class Interna
    {
        public double calculoNeto(double precio)
        {
            return precio*tasa;
        }
    }
     double tasa=1.21;
}
```

Pero si intentamos utilizarla con el mismo código que la versión `static` de la clase, encontramos problemas en tiempo de compilación.



Efectivamente, debemos disponer obligatoriamente de una instancia de la clase `container` para que la clase interna esté disponible. Por lo tanto, debemos previamente instanciar la clase y, a continuación, pedirle que nos proporcione una instancia de la clase interna. De este modo, la sintaxis correcta es la siguiente:

```
Externa e;
e=new Externa();
Externa.Interna ci;
ci=e.new Interna();
System.out.println(ci.calculoNeto(100));
```

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 si sólo se la utiliza 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. Modificamos ligeramente la función para que, en el momento de la llamada, se le pueda proporcionar además del array a clasificar la función que deberá usar para efectuar las comparaciones. Una solución para transmitir una función a otra función es proporcionarle 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:

```
// esta interfaz deberá ser implementada por las clases
// para las cuales se plantea una comparación de las instancias
public interface Comparador
{
```

```
  // se podrá llamar a este método para comparar los dos objetos
  // recibidos como parámetro
  // el método devuelve un entero cuyo valor depende
  // de las reglas siguientes
  // 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;

}
```

Una interfaz así, que no contiene más que la definición de único método, se llama interfaz funcional.

Ahora podemos revisar nuestra función de clasificación teniendo en cuenta nuestras mejoras.

```
public static Object[] clasificación(Object[] array,
Comparador 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])==
Comparador.INFERIOR)
       {
             c = arrayClasificación[j];
             arrayClasificación[j] = arrayClasificación[i];
             arrayClasificación[i] = c;
       }
       else if
(clasificador.compare(arrayClasificación[j],arrayClasificación[i])==
Comparador.ERROR)
       {
             return null;
       }
       }
   }
    return arrayClasificación;
}
```

Para utilizar esta nueva función de clasificación, debemos proporcionarle dos parámetros:

- El array a 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. La sintaxis de creación de una instancia de una clase interna anónima resulta bastante rara a primera vista.

```
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.

```
Persona[] tab;
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));

tabClasificación=(Persona[])clasificar(tab,
   // creación de una instancia de clase que implementa la interfaz
   // Comparador
     new Comparador()
     // a continuación el código de la clase
     {
          // como lo exije 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 Clasificable.ERROR;
                 }
                   if (p1.getApellido().compareTo(p2.getApellido())<0)
                 {
                       return Clasificable.INFERIOR;
                 }
                    if (p1.getApellido().compareTo(p2.getApellido())>0)
                 {
                       return Clasificable.SUPERIOR;
                 }

                 return Clasificable.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 a la función `clasificación` con una nueva instancia de una clase interna anónima que implemente de manera diferente la función `Compare`. Por ejemplo, podemos clasificar las personas por su edad.

```
tabClasificación=(Persona[]) Clasificación (tab,
// creación de una instancia de clase que implementa la interfaz
// Comparador
     new Comparador()
     // aquí está el código de la clase
     {
     // como lo exije 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 Clasificable.ERROR;
               }
               if (p1.calculoEdad()<p2.calculoEdad())
               {
                    return Clasificable.INFERIOR;
               }
               if (p1.calculoEdad()>p2.calculoEdad())
               {
                    return Clasificable.SUPERIOR;
               }
               return Clasificable.IGUAL;
          } // llave de cierre del método

     } // llave de cierre de la clase
     ); // fin de la llamada de la función de clasificación


        for (int i=0;i<tabClasificación.length;i++)
        {
          System.out.println(tabClasificación[i]);
        }
```

En tiempo de 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:

`Principal.class`: el archivo que corresponde a la clase `Principal`.

`Principal$1.class`: el archivo que corresponde a la primera clase interna anónima.

`Principal$2.class`: el archivo que corresponde a la segunda clase interna anónima.

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!).

6. Expresión lambda

En la sección anterior hemos utilizado una clase interna anónima para transmitir a la función de ordenación el método con el que deseamos realizar la clasificación de los elementos de la tabla. La instancia de la clase interna anónima que hemos transmitido a la función de ordenación debe respetar la interfaz `Comparador`. Esta interfaz es una interfaz funcional (contiene una única definición de método), de modo que la instancia de la clase transmitida contiene, también, un único método. Esta instancia de clase interna anónima puede remplazarse por una expresión lambda. Vamos a transmitir a la función de ordenación, no una instancia de clase que contendrá la función a utilizar para realizar la ordenación, sino directamente la función en sí.

Una expresión lambda se parece a una función, salvo que no tiene nombre. Está formada por un par de paréntesis que contienen los posibles parámetros, seguidos de los caracteres -> y del bloque de código de la expresión delimitada por las llaves.

La llamada a la función de ordenación puede realizarse de la siguiente manera, utilizando una expresión lambda en lugar de la instancia de clase que implementa la interfaz `Comparador`.

```
tabOrdenada=(Persona[])ordena(tab,(Object o1,Object o2)->
    {
       Persona p1,p2;
        if (o1 instanceof Persona & o2 instanceof Persona)
        {
                p1=(Persona)o1;
                p2=(Persona)o2;
        }
        else
        {
                return Clasificable.ERROR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())<0)
        {
                return Clasificable.INFERIOR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())>0)
        {
                return Clasificable.SUPERIOR;
        }

        return Clasificable.IGUAL;
    });
```

Este código se parece mucho al que hemos utilizado para invocar a la función de ordenación pasando como parámetro una instancia de clase anónima.

```
tabOrdenada=(Persona[])ordena(tab,  new Comparador()
         {
          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 Clasificable.ERROR;
               }
                if (p1.getApellido().compareTo(p2.getApellido())<0)
               {
               return Clasificable.INFERIOR;
               }
               if (p1.getApellido().compareTo(p2.getApellido())>0)
               {
                  return Clasificable.SUPERIOR;
             }

               return Clasificable.IGUAL;

          }
         }
         );
```

La sintaxis sigue siendo relativamente compleja. Veamos, a continuación, cómo simpificarla un poco.

En una aplicación, debemos gestionar un conjunto de personas. Para realizar dicha gestión, agrupamos nuestras personas en una tabla.

```
Persona[] tab;
tab=new Persona[5];
tab[0] = new Persona("Wayne", "John",LocalDate.of(1907,5,26));
tab[1] = new Persona("McQueen", "Steve",LocalDate.of(1930,3,24));
tab[2] = new Persona("Lennon", "John",LocalDate.of(1940,10,9));
tab[3] = new Persona("Gibson", "Mel",LocalDate.of(1956,1,3));
tab[4] = new Persona("Willis", "Bruce",LocalDate.of(1955,3,19));
```

Nuestra aplicación debe poseer distintas funcionalidades de búsqueda de una persona en la tabla.

- búsqueda por nombre
- búsqueda por apellido
- búsqueda por nombre y apellido
- búsqueda por edad

Debemos, por tanto, crear cuatro funciones que permitan realizar dichas búsquedas.

```
public static Persona busquedaPorApellido(Persona[]
tabla,String apellido)
{
    for(Persona p:tabla)
    {
        if (p.getApellido().equals(apellido))
        {
            return p;
        }
    }
    return null;

}
public static Persona busquedaPorNombre(Persona[] tabla,String nombre)
{
    for(Persona p:tabla)
    {
        if (p.getNombre().equals(nombre))
        {
            return p;
        }
    }
    return null;
}
public static Persona busquedaPorEdad(Persona[] tabla,int edad)
{
    for(Persona p:tabla)
    {
        if (p.calculaEdad()==edad)
        {
            return p;
        }
    }
    return null;

}
public static Persona busquedaPorApellidoNombre(Persona[]
tabla,String apellido,String nombre)
{
    for(Persona p:tabla)
    {
        if (p.getApellido().equals(apellido)&& p.getNombre().equals(nombre))
        {
            return p;
        }
    }
    return null;

}
```

Estas cuatro funciones se parecen bastante entre sí. La única línea de código que varía es la encargada de realizar la comparación. Para poder "factorizar" nuestro código, puede ser interesante extraer de la función el código de comparación.

Comenzamos definiendo una interfaz que describe la firma que tendrá que respetar la función encargada de verificar la igualdad de dos personas.

```
public interface ComparadorPersona
{
    boolean isIdentica(Persona p);

}
```

Ahora debemos diseñar la nueva versión de la función de búsqueda de una persona que utiliza la interfaz definida anteriormente.

```
public static Persona busquedaPersona(Persona[] tabla,
ComparadorPersona cp)
{
    for(Persona p:tabla)
    {
        if (cp.isIdentica(p))
        {
            return p;
        }
    }
    return null;
}
```

No se aprecia ningún rastro del criterio de comparación en el interior de dicha función. Se definirá en el momento de invocar a la función.

El paquete `java.util.function` proporciona numerosas interfaces que contienen definiciones de funciones cuyo uso se repite de manera recurrente en una aplicación. Para que estas interfaces puedan utilizarse fácilmente, son genéricas (consulte la sección siguiente). Utilizando una de estas interfaces predefinidas, nuestra función de búsqueda puede escribirse de la siguiente manera:

```
public static Persona busquedaPersonaPrd(Persona[]
tabla,Predicate<Persona>pr)
{
    for(Persona p:tabla)
    {
        if (pr.test(p))
        {
            return p;
        }
    }
    return null;
}
```

Apliquemos ahora nuestra primera experiencia de escritura de una expresión lambda a esta nueva función.

```
System.out.println(busquedaPersona(tab, (Persona pe)->
        {
            if(pe.getNombre().equals("Bruce"))
                return true;
            else
                return false;
        }
        ));
```

Esta sintaxis funciona correctamente, aunque es relativamente verbosa. Veamos, a continuación, cómo simplificar esta expresión.

La primera simplificación consiste en utilizar los parámetros de la expresión lambda. No estamos obligados a especificar el tipo de los parámetros y, además, si la expresión recibe un único parámetro los paréntesis son también opcionales.

```
System.out.println(busquedaPersona(tab, pe->
        {
            if(pe.getNombre().equals("Bruce"))
                return true;
            else
                return false;
        }
        ));
```

La segunda simplificación se basa en el cuerpo de la expresión lambda. Si contiene una sola expresión, las llaves son opcionales, así como el uso de la palabra clave `return`. En este caso, la expresión se evalúa en tiempo de ejecución y el valor generado se devuelve al código que la invoca.

```
System.out.println(busquedaPersona(tab, pe->
pe.getNombre().equals("Bruce")));
```

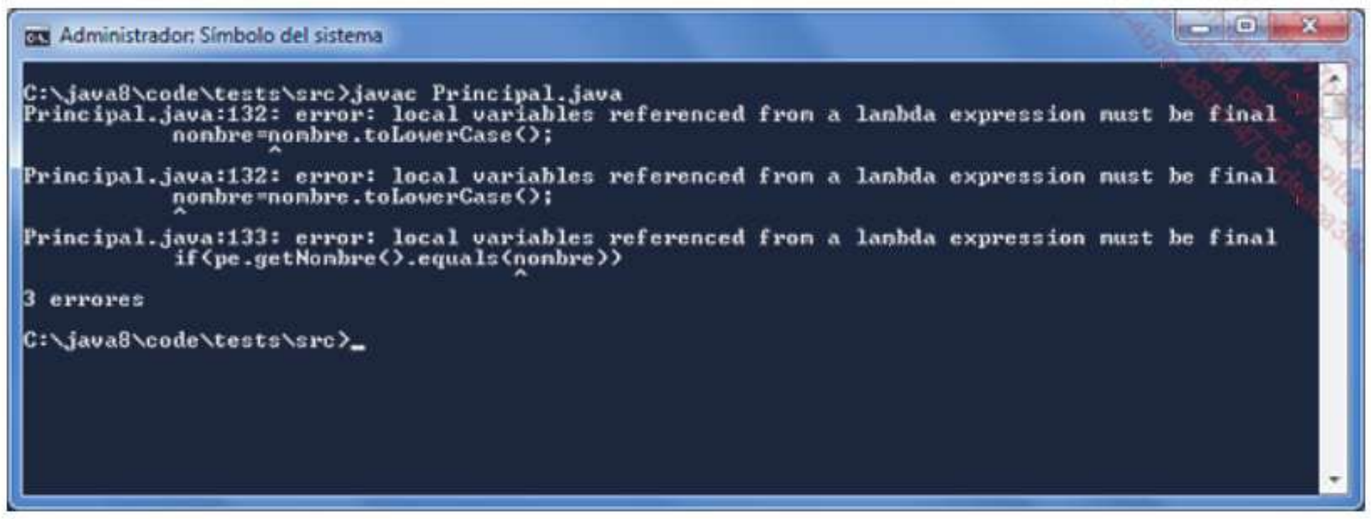
Si utiliza la palabra clave `return` en su expresión debe, obligatoriamente, ubicar un bloque delimitado por llaves, incluso aunque contenga una única expresión.

La expresión lambda puede utilizar las variables disponibles en el contexto donde está definida.

```
        BufferedReader br;
        br=new BufferedReader(new InputStreamReader(System.in));
        String nombre;
        nombre=br.readLine();



        System.out.println(busquedaPersona(tab, pe->
        {

            if(pe.getNombre().equals(nombre))
                return true;
            else
                return false;
        }
        ));
```

En cualquier caso, la expresión lambda no puede modificar el contenido de la variable, sino únicamente utilizarla. La siguiente modificación genera un error de compilación.

```
    BufferedReader br;
        br=new BufferedReader(new InputStreamReader(System.in));
        String nombre;
        nombre=br.readLine();



        System.out.println(busquedaPersona(tab, pe->
        {
            nombre=nombre.toLowerCase();
            if(pe.getNombre().equals(nombre))
                return true;
            else
                return false;
        }
        ));
```



7. Referencia de método

Cuando una expresión lambda se vuelve muy voluminosa o debe reutilizarse en varios lugares de la aplicación, es preferible ubicar su contenido en una función y, simplemente, invocar a dicha función en el cuerpo de la expresión lambda.

Por ejemplo, la siguiente expresión lambda:

```
tabOrdenada=(Persona[])ordena(tab,(Object o1,Object o2)->
    {
       Persona p1,p2;
        if (o1 instanceof Persona & o2 instanceof Persona)
        {
                p1=(Persona)o1;
                p2=(Persona)o2;
        }
        else
        {
                return Clasificable.ERROR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())<0)
        {
                return Clasificable.INFERIOR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())>0)
        {
                return Clasificable.SUPERIOR;
        }

        return Clasificable.IGUAL;
    });
```

puede, fácilmente, externalizarse en una función.

```
public static int comparePersona(Object o1,Object o2)
{
       Persona p1,p2;
        if (o1 instanceof Persona & o2 instanceof Persona)
        {
                p1=(Persona)o1;
                p2=(Persona)o2;
        }
        else
        {
                return Clasificable.ERROR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())<0)
        {
                return Clasificable.INFERIOR;
        }
        if (p1.getApellido().compareTo(p2.getApellido())>0)
        {
                return Clasificable.SUPERIOR;
        }

        return Clasificable.IGUAL;
}
```

Esta función puede utilizarse, ahora, en varias expresiones lambda. El código de la expresión se ve, por otro lado, enormemente simplificado por dicha modificación.

```
tabOrdenada=(Persona[])ordena(tab,(Object o1,Object o2)->
comparaPersona(o1, o2));
```

La simplificación puede llevarse más allá utilizando una referencia de método. Esta solución consiste, simplemente, en identificar el método que se desea utilizar a partir de su nombre. El operador `::` debe utilizarse en esta situación. Permite obtener una referencia hacia la función que se desea utilizar, pero no provoca una llamada a la función, como haría el operador `.`.

```
tabOrdenada=(Persona[])ordena(tab,Principal::comparaPersona);
```

Esta sintaxis está permitida puesto que la función está declarada con la palabra clave `static`. No es necesario disponer de una instancia de la clase en la que se declara para poder utilizarla.

Si la función no está declarada como `static`, la regla es la misma que para todos los demás elementos disponibles en una clase: es obligatorio tener una instancia de la clase para poder utilizarla.

En este caso, debemos utilizar el nombre de la variable que contiene la instancia de la clase para poder hacer referencia al método.

```
OrdenadorPersona op;
op=new OrdenadorPersona();
tabOrdenada=(Persona[])ordena(tab,op::comparaPersona);
```

8. Los genéricos

Los tipos genéricos son elementos de un programa que se adaptan de manera automática para realizar la misma funcionalidad con diferentes tipos de datos. Cuando creamos un elemento genérico, no es necesario diseñar una versión diferente para cada tipo de dato con el que desea llevar a cabo una funcionalidad. Para establecer una analogía con un objeto corriente, vamos a tomar el ejemplo del destornillador. Según el tipo de tornillo que vamos a utilizar, podremos escoger un destornillador específico para este tipo de tornillo (plano, cruciforme, torx…). Una técnica a menudo empleada por alguien que sea un manitas consiste en adquirir un destornillador múltiple con distintas cabezas. Según el tipo de tornillo, elige la cabeza adequada. El resultado final es el mismo que si dispusiera de una multitud de destornilladores distintos: puede atornillar y destornillar.

Cuando se utiliza un tipo genérico, se configura con un tipo de datos. Esto permite al código adaptarse automáticamente y llevar a cabo la misma acción sin importar el tipo de datos. Una alternativa podría ser el uso del tipo universal `Object`. La utilización de tipos genéricos presenta varias ventajas respeto a esta solución:

- Impone la verificación de los tipos de datos en el momento de la compilación y permite eludir las inevitables comprobaciones manuales que se deben hacer al utilizar el tipo `Object`.
- Evita las operaciones de conversión del tipo `Object` hacia un tipo específico y a la inversa.
- La escritura del código resulta más fácil en algunos entornos de desarrollo con la visualización automática de todos los miembros disponibles para un tipo de datos particular.
- Favorece la escritura de algoritmos que son independientes de los tipos de datos.

Los tipos genéricos, sin embargo, pueden imponer algunas restricciones relativas al tipo de datos empleado. Pueden, por ejemplo, imponer que el tipo de dato utilizado implemente una o varias interfaces: o bien un tipo referencia, o bien que posea un constructor por defecto. Sin embargo, es importante entender correctamente algunos términos usados con los genéricos.

- El tipo genérico: es la definición de una clase, interfaz o función para la cual se especifica al menos un tipo de datos en el momento de su declaración.
- El tipo de parámetro: es el espacio reservado para el tipo de parámetro en la declaración del tipo genérico.
- El tipo argumento: representa el tipo de datos que sustituye al tipo parámetro en el momento de la construcción de un tipo desde un tipo genérico.
- Las restricciones: son las restricciones que se imponen y que limitan el tipo de argumento que podemos proporcionar.
- El tipo construido: es la clase, interfaz, o función construida a partir de un tipo genérico para el cual hemos especificado tipos de argumento.

a. Clases genéricas

Una clase que espera un tipo de parámetro se llama clase genérica. Podemos generar una clase construida al proporcionar a la clase genérica un tipo de argumento para cada uno de estos tipos de parámetros.

Definición de una clase genérica

Podemos definir una clase genérica que facilita las mismas funcionalidades en distintos tipos de datos. Para ello, debemos proporcionar uno o varios tipos de parámetro en la definición de la clase. Veamos el ejemplo de una clase capaz de gestionar una lista de elementos con las funcionalidades siguientes.

- Añadir un elemento.
- Suprimir un elemento.
- Desplazarse sobre el primer elemento.
- Desplazarse sobre el último elemento.
- Desplazarse sobre el elemento siguiente.
- Desplazarse sobre el elemento anterior.
- Obtener el número de elementos.

Primero, debemos definir la clase como una clase ordinaria.

```
public class ListaGenerica
{
}
```

La transformación de esta clase en una clase genérica se hace añadiendo un tipo de parámetro justo después del nombre de la clase.

```
public class ListaGenerica <T>
{
}
```

Si se requiere la presencia de varios parámetros, deben ir separados por comas en la declaración. Por convención, se representan los tipos de parámetros con un único carácter en mayúsculas.

Si el código de la clase debe realizar operaciones distintas a las asignaciones, debemos añadir restricciones en el tipo de parámetro. Para ello, se añade la palabra clave `extends` seguida de la restricción. La restricción se puede constituir de una clase específica, cuyo tipo argumento deberá heredar, o de una o varias interfaces que deberá implementar. Si se debe proceder a aplicar varias restricciones, se separan con el carácter &. En

este caso, hay que especificar al principio de la lista la restricción relacionada con una clase y, a continuación, las relacionadas con las interfaces. Mostramos, a continuación, algunos ejemplos para ilustrar esto.

Clase genérica que exige que el tipo argumento herede de la clase `Persona`.

```
public class ListaGenerica <T extends Persona>
{
}
```

Clase genérica que exige que el tipo argumento implemente la interfaz `Clasificable`.

```
public class ListaGenerica <T extends Clasificable>
{
}
```

La misma palabra clave extends se utiliza para una restricción relacionada con una clase o una interfaz.

Clase genérica que exige que el tipo argumento herede de la clase `Persona` e implementa las interfaces `Clasificable` y `Cloneable`:

```
public class ListaGenerica <T extends Persona & Clasificable & Cloneable >
{
}
```

Si no se especifica ninguna restricción, las únicas operaciones autorizadas serán las soportadas por el tipo `Object`.

En el código de clase, cada miembro que debe ser del tipo argumento debe ser definido con el tipo parámetro, en nuestro caso T. Veamos ahora el código completo de la clase.

```
import java.util.ArrayList;
public class ListaGenerica <T>
{
    // para almacenar los elementos de la lista
    private ArrayList<T> lista;
    // puntero de posición en la lista
    private int posición;
    //número de elementos de la lista
    private int numElementos;
    // constructor con un parámetro que permite dimensionar la lista
    public ListaGenerica(int tamaño)
    {
     lista=new ArrayList<T>(tamaño);
    }
    public void añadir(T elemento)
    {
            lista.add(elemento);
            numElementos = numElementos + 1;
    }
   public void insertar(T elemento,int índice)
    {
       // verificamos si el índice no es superior al número de elementos
       //  o si el índice no es inferior a 0
       if (índice >= numElementos || índice < 0)
       {
          return;
       }
       lista.add(índice, elemento);
       // se actualiza el número de elementos
       numElementos = numElementos + 1;
    }

    public void reemplazar(T elemento,int índice)
    {
       // verificamos si el índice no es superior al número de elementos
       // o si el índice no es inferior a 0
       if (índice >= numElementos || índice < 0)
       {
          return;
       }
       lista.set(índice,elemento);
    }
    public void suprimir(int índice)
    {
       int i;
       // verificamos si el índice no es superior al número de elementos
       // o si el índice no es inferior a 0
       if (índice >= numElementos || índice < 0)
       {
          return;
       }
       lista.remove(índice);
       // se actualiza el número de elementos
       numElementos = numElementos - 1;
    }
    public T getElemento(int j)
    {
          return lista.get(j);
    }
    public int getNumElementos()
    {
          return numElementos;
    }
    public T primero() throws Exception
```

```
    {
       if (numElementos == 0)
       {
           throw new Exception("lista vacía");
       }
       // se desplaza el puntero hasta el primer elemento
       posición = 0;
       return lista.get(0);
    }
    public T último() throws Exception
    {
       if (numElementos == 0)
       {
           throw new Exception("lista vacía");
       }
       // se desplaza el puntero hasta el último elemento
       posición = numElementos - 1;
       return lista.get(posición);
    }
    public T siguiente() throws Exception
    {
       if (numElementos == 0)
       {
           throw new Exception("lista vacía");
       }
       // verificamos si no estamos al final de la lista
       if (posición == numElementos - 1)
       {
           throw new Exception("no hay más elementos");
       }
       // se desplaza el puntero sobre el elemento siguiente
       posición = posición + 1;
       return lista.get(posición);
    }
    public T anterior() throws Exception
    {
       if (numElementos == 0)
       {
           throw new Exception("lista vacía");
       }
       // verificamos si no estamos en el primer elemento
       if (posición == 0)
       {
           throw new Exception("no hay ningun elemento anterior");
       }
       // nos desplazamos hasta el elemento anterior
       posición = posición - 1;
       return lista.get(posición);
    }
}
```

Utilización de una clase genérica

Para poder utilizar una clase genérica, debemos generar una clase construida al proporcionar un tipo de argumento para cada uno de dichos tipos de parámetro. A continuación, podemos instanciar la clase construida por uno de los constructores disponibles. Vamos a usar la clase diseñada anteriormente para trabajar con una lista de cadenas de caracteres.

```
ListaGenerica<String> lista = new ListaGenerica<String>(5);
```

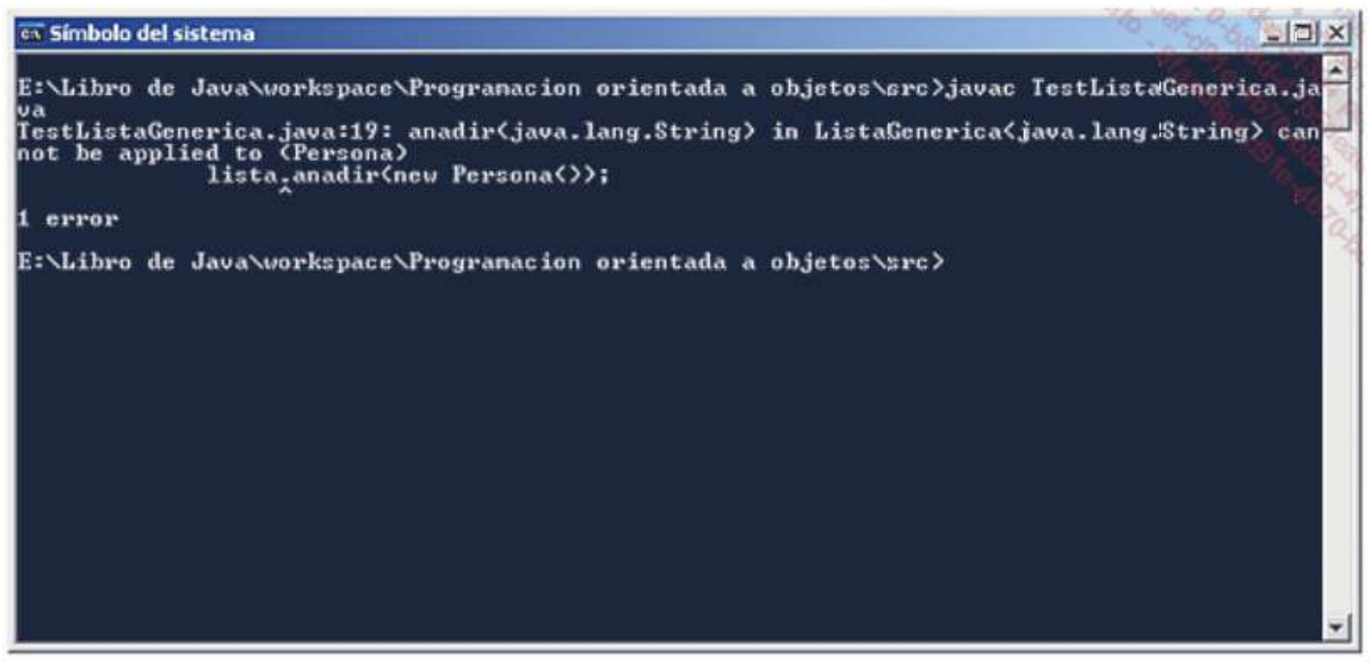
Esta sintaxis puede simplificarse dejando al compilador determinar el tipo de argumento que se debe utilizar durante la llamada al constructor. Basta, simplemente, con omitir el tipo de argumento entre los caracteres <y>.

```
ListaGenerica<String> lista=new ListaGenerica<>(5);
```

Esta declaración permite instanciar una lista de cinco cadenas. A través de ella están disponibles los métodos de la clase.

```
lista.anadir("primero");
lista.anadir("segundo");
```

El compilador verifica por supuesto que utilizamos nuestra clase correctamente, en particular, comprobando los tipos de los datos que le facilitamos.



A continuación se muestra el código de una pequeña aplicación que permite comprobar el buen funcionamiento de nuestra clase genérica:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class TestListaGenerica
{
      static ListaGenerica<String> lista = new
ListaGenerica<String>(5);
      public static void main(String[] args)
      {
             lista.añadir("primero");
             lista.añadir("segundo");
             lista.añadir("tercero");
             lista.añadir("cuarto");
             lista.añadir("quinto");
             menú();
      }
      public static void menú()
      {
        char selección='\0';
        System.out.println("p (primero) < (anterior) >(siguiente)u
(último) f (fin)");
        while (selección!= 'f')
        {
            try
            {
                BufferedReader br;
                   br=new BufferedReader(new
InputStreamReader(System.in));
                   selección=br.readLine().charAt(0);
                   switch (selección)
                   {
                   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 'u':
                        System.out.println("el último " +
lista.último());
                        break;
                   }
            }
            catch (Exception e)
            {
                System.out.println(e.getMessage());
            }
      System.out.println("p (primero) < (anterior) >(siguiente) u
(último) f (fin)");
         }
      }
}
```

También podemos comprobar que nuestra clase funciona sin problema si le pedimos trabajar con personas.

```
lista.anadir(new Persona("juanito2","nombre2",
LocalDate.of(1922,2,15)));
lista.anadir(new Persona("juanito1","nombre1",
LocalDate.of(1911,1,15)));
lista.anadir(new Persona("juanito5","nombre5",
LocalDate.of(1955,5,15)));
lista.anadir(new Persona("juanito3","nombre3",
LocalDate.of(1933,3,15)));
lista.anadir(new Persona("juanito4","nombre4",
LocalDate.of(1944,4,15)));
```

b. Métodos genéricos

Una función genérica es un método definido con al menos un tipo de parámetro. Esto permite al código que la invoca especificar el tipo de datos que necesita con cada llamada de la función. Sin embargo, es posible emplear dicho método sin indicar información alguna para el tipo argumento. En este caso, el compilador intenta determinar el tipo según los argumentos que se pasan al método. Para ilustrar la creación de funciones genéricas, vamos a transformar la función clasificación en versión genérica.

```
public static <T extends Clasificable> void
clasificación(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.getElemento(j).compare(lista.getElemento(i))==Clasificable.
INFERIOR)
                 {
                     c = lista.getElemento(j);
                     lista.reemplazar(lista.getElemento(i), j);
                     lista.reemplazar(c,i);
                 }
```

```
                    else if
(lista.getElemento(j).compare(lista.getElemento(i))==Clasificable.ERROR)
                    {
                         throw new Exception("error durante la clasificación");
                    }
               }
          }
}
```

Como para la creación de una clase genérica, el tipo de parámetro se especifica entre los caracteres < y >. Si se deben aplicar restricciones a los tipos argumentos, podemos aplicarlas con las mismas reglas que para las clases genéricas.

La llamada de una función genérica es similar a la llamada de una función normal, salvo que podemos especificar tipos argumentos para cada uno de los tipos parámetro. Esto no es una obligación ya que el compilador es capaz de inferir los tipos argumentos en tiempo de compilación.

Por lo tanto, es posible invocar a la función con las dos sintaxis siguientes:

```
TestListaGenerica.<Persona>clasificación(lista);
```

o

```
TestListaGenerica.clasificación(lista);
```

c. Los genéricos y la herencia

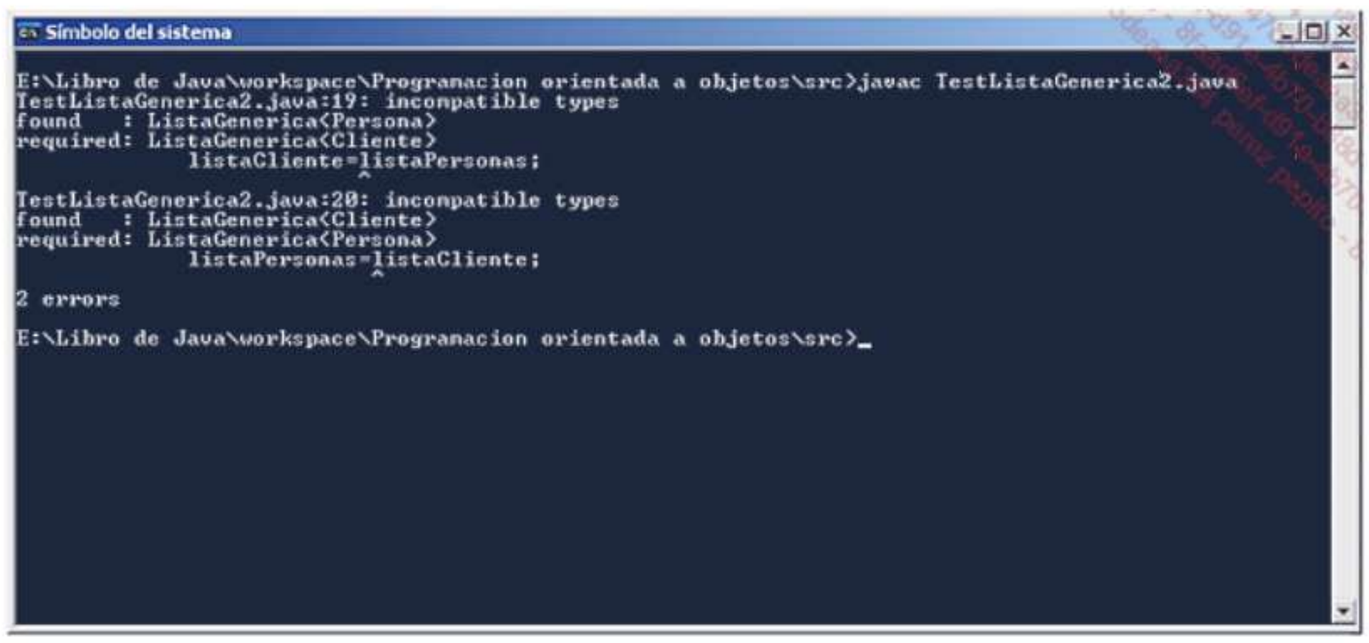
El uso combinado de los genéricos y la herencia puede provocar a veces algunos problemas para el desarrollador de una aplicación. Hemos determinado en el párrafo dedicado a la herencia que con una variable de cierto tipo, podíamos referenciar una instancia de clase de dicho tipo, pero también una instancia de cualquiera de sus subtipos. El código siguiente es por lo tanto perfectamente legal y funciona correctamente.

```
Persona p;
Cliente c;
c=new Cliente();
p=c;
p.setApellido("García");
p.setNombre("pablo");
```

Si intentamos realizar lo mismo con los genéricos, probando el código siguiente:

```
ListaGenerica<Persona> listaPersonas;
ListaGenerica<Cliente> listaCliente;
listaCliente=new ListaGenerica<Cliente>(10);
listaPersonas=listaCliente;
```

Es legítimo pensar que ya que se puede asignar a una variable de tipo `Persona` una referencia hacia una instancia de una de sus subclases (`Cliente`), seguramente se pueda realizar la misma operación con una `ListaGenerica<Persona>` y una `ListaGenerica<Cliente>`. Sin embargo, el compilador no ve las cosas de la misma manera.



De hecho, esta limitación está relacionada con el mecanismo de borrado de tipo empleado por el compilador. Su meta principal es hacer compatible el código compilado con las versiones anteriores de las máquinas virtuales Java.

De hecho, en el momento de la compilación de una clase genérica, el compilador sustituye el tipo parámetro por el tipo `Object` o por un tipo correspondiente a la restricción puesta sobre el tipo parámetro. El compilador tratará el código de la clase `ListaGenerica` de la manera siguiente.

```
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 posición;
    //número de elementos de la lista
    private int numElementos;
    // constructor con un parámetro que permite dimensionar la lista
    public ListaGenerica(int tamaño)
    {
         lista=new ArrayList<T Object>(tamaño);
    }
    public void anadir(T Object elemento)
    {
```

```
            lista.add(elemento);
            numElementos = numElementos + 1;
    }
   public void insertar(T Object elemento,int índice)
    {
        // se comprueba si el índice no es superior al número de
        // elementos o si el índice no es inferior a 0
        if (índice >= numElementos || índice < 0)
        {
           return;
        }
        lista.add(índice,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 desplaza el puntero sobre el primer elemento
        posición = 0;
        return lista.get(0);
    }
    public T Object último() throws Exception
    {
        if (numElementos == 0)
        {
            throw new Exception("lista vacía");
        }
        // se desplaza el puntero sobre el último elemento
        posición = numElementos - 1;
        return lista.get(posición);
    }
...
...
}
```

Para convencernos de ello, probamos el código siguiente en la clase `ListaGenerica`.

```
   public void anadir(T elemento)
   {
            lista.add(elemento);
            numElementos = numElementos + 1;
   }
   public void anadir(Object elemento)
   {
            lista.add(elemento);
            numElementos = numElementos + 1;
   }
```

En tiempo de compilación obtenemos, en efecto, un mensaje de error que nos indica que el método `anadir(Object)` está definido dos veces en la clase.

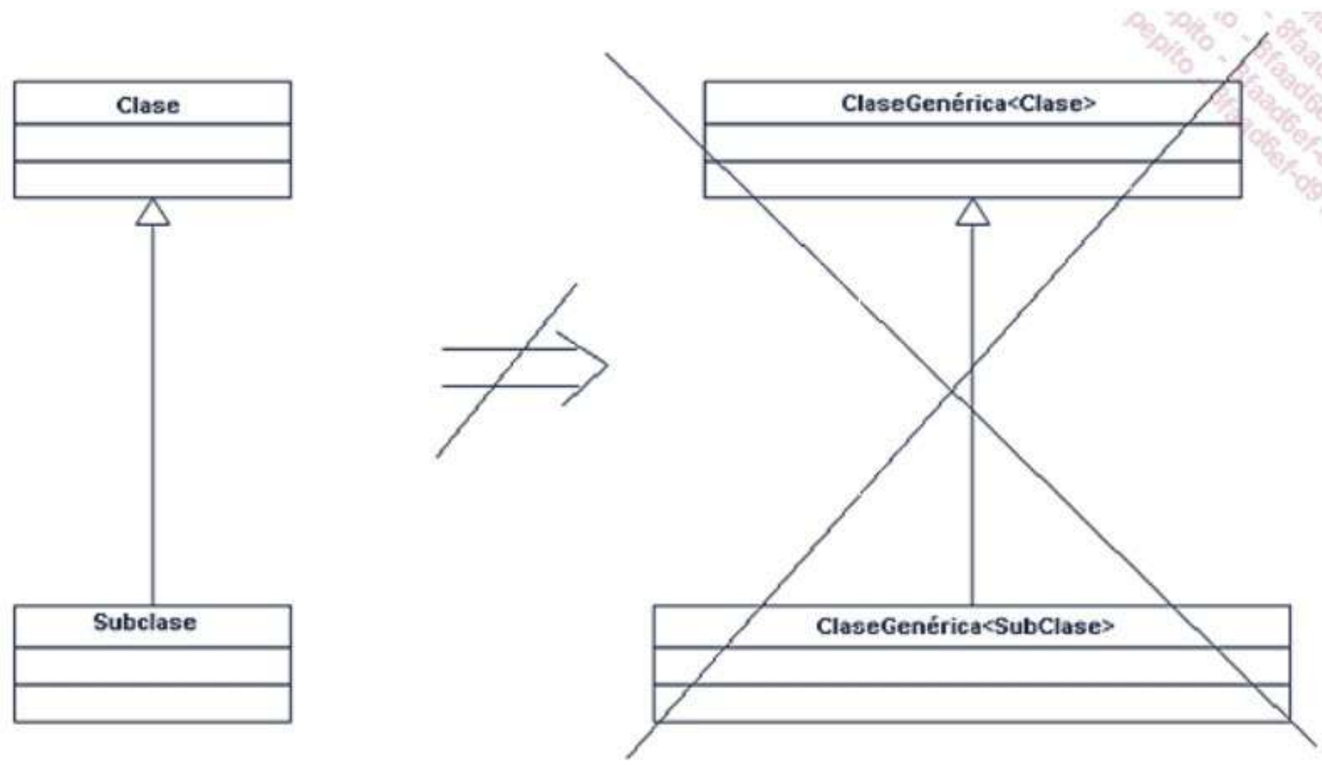
Otra operación realizada por el compilador, consiste en efectuar una conversión para los valores devueltos por las funciones de la clase genérica. El código siguiente:

```
static ListaGenerica<Persona> lista = new ListaGenerica<Persona>(5);
public static void main(String[] args) throws Exception
{
  lista.anadir(new Persona("juanito2", "nombre2",
LocalDate.of(1922,2,15)));
  lista.anadir(new Persona("juanito1", "nombre1",
LocalDate.of(1911,1,15)));
  lista.anadir(new Persona("juanito5", "nombre5",
LocalDate.of(1955,5,15)));
  liste.anadir(new Persona("juanito3", "nombre3",
LocalDate.of(1933,3,15)));
  lista.anadir(new Persona("juanito4", "nombre4",
LocalDate.of(1944,4,15)));
  Persona p;
  p=lista.getElemento(0);
}
```

se compilará en realidad de manera implícita de la forma siguiente:

```
static ListaGenerica<Persona> lista = new
ListaGenerica<Persona>(5);
public static void main(String[] args) throws Exception
{
  lista.anadir(new Persona("juanito2", "nombre2",
LocalDate.of(1922,2,15)));
  lista.anadir(new Persona("juanito1", "nombre1",
LocalDate.of(1911,1,15)));
  lista.anadir(new Persona("juanito5", "nombre5",
LocalDate.of(1955,5,15)));
  lista.anadir(new Persona("juanito3", "nombre3",
LocalDate.of(1933,3,15)));
  lista.anadir(new Persona("juanito4", "nombre4",
LocalDate.of(1944,4,15)));
  Persona p;
  p=(Persona)lista.getElemento(0);
}
```

Lo que cabe destacar de estas experiencias es el hecho de que una clase construida con una clase cualquiera como tipo parámetro no es la subclase de una clase genérica construida con un tipo que es el supertipo de esta clase cualquiera.



Esta limitación puede resultar a veces molesta. Si queremos crear una función que reciba como parámetro una `ListaGenerica` construida desde cualquier tipo de datos, nuestra primera intuición sería escribir el código siguiente:

```
public static void visualización(ListaGenerica<Object> lista)
{

}
```

Desafortunadamente, nuestra primera intuición no es la buena ya que con esta sintaxis la función `visualización` sólo acepta como parámetro instancias de `ListaGenerica` de `Object`.

Para representar el hecho de que la función `visualización` recibe como parámetro una instancia de `ListaGenerica` construida desde cualquier tipo, debemos utilizar el carácter comodín "?" (símbolo de interrogación) en la definición de la función `visualización`. Se obtiene:

```
public static void visualización(ListaGenerica<?> lista)
{

}
```

Con esta sintaxis podemos utilizar la función `visualización` recibiendo como parámetro una instancia de `ListaGenerica` construida desde cualquier clase. Por el contrario, en el código de la función, sólo podremos emplear métodos de la clase `Object` en los elementos presentes en la lista. Podemos añadir restricciones sobre el tipo argumento de la función al hacer seguir el carácter '?' (símbolo de interrogación) por una restricción definida con exactamente la misma sintaxis que para una clase genérica. Para que la función `visualización` acepte como parámetro una `ListaGenerica` de `Persona` y de cualquiera de sus subclases, debemos emplear la sintaxis siguiente:

```
public static void visualización(ListaGenerica<? extends
Persona> lista) {
     lista.primero();
     for (int i=0;i<lista.getNumElementos();i++)
     {
         System.out.println(lista.getElemento(i).getApellido());
         System.out.println(lista.getElemento(i).getNombre());
         System.out.println("----------------------------");
     }
}
```

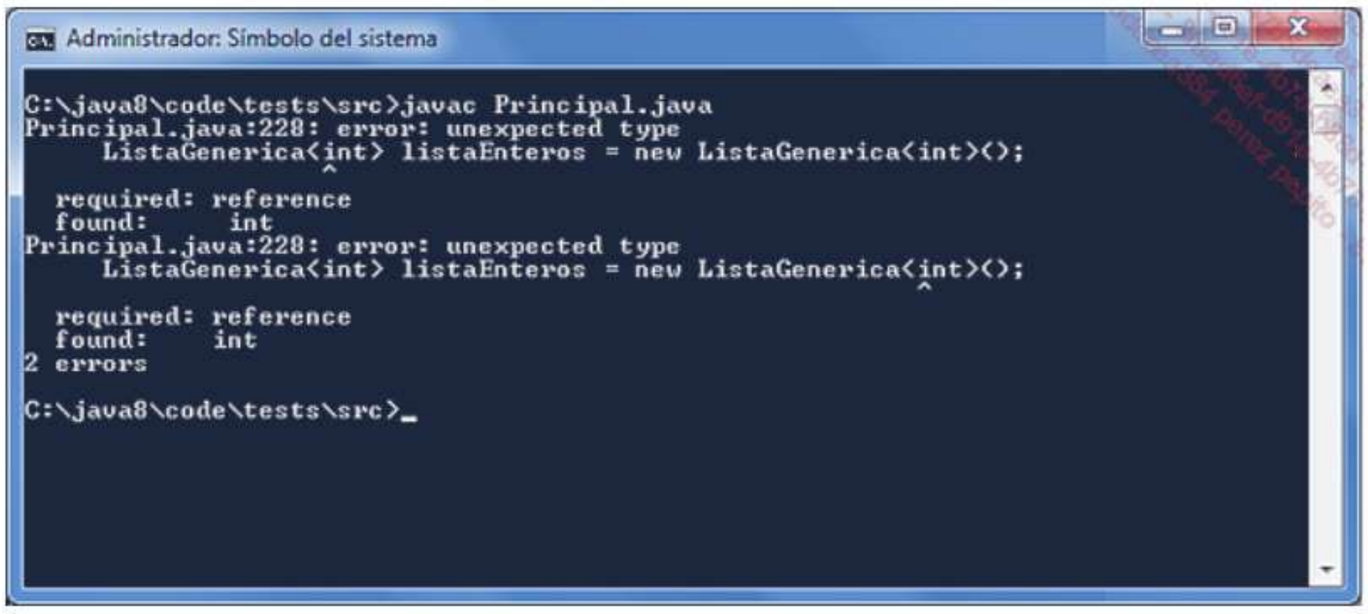
Destaquemos que, en este caso, las instancias presentes en la lista serán al menos instancias de la clase `Persona` y se las podrá utilizar como tal y con toda seguridad en el interior de la función.

d. Limitación de los genéricos

El uso de tipos genéricos impone ciertas restricciones. Estas limitaciones están vinculadas al tipo de argumento utilizado y el uso que se realiza del mismo.

a - El tipo de argumento es, necesariamente, un tipo por referencia. Por ejemplo, es imposible crear una instancia de `ListaGenerica` de tipo `int`.

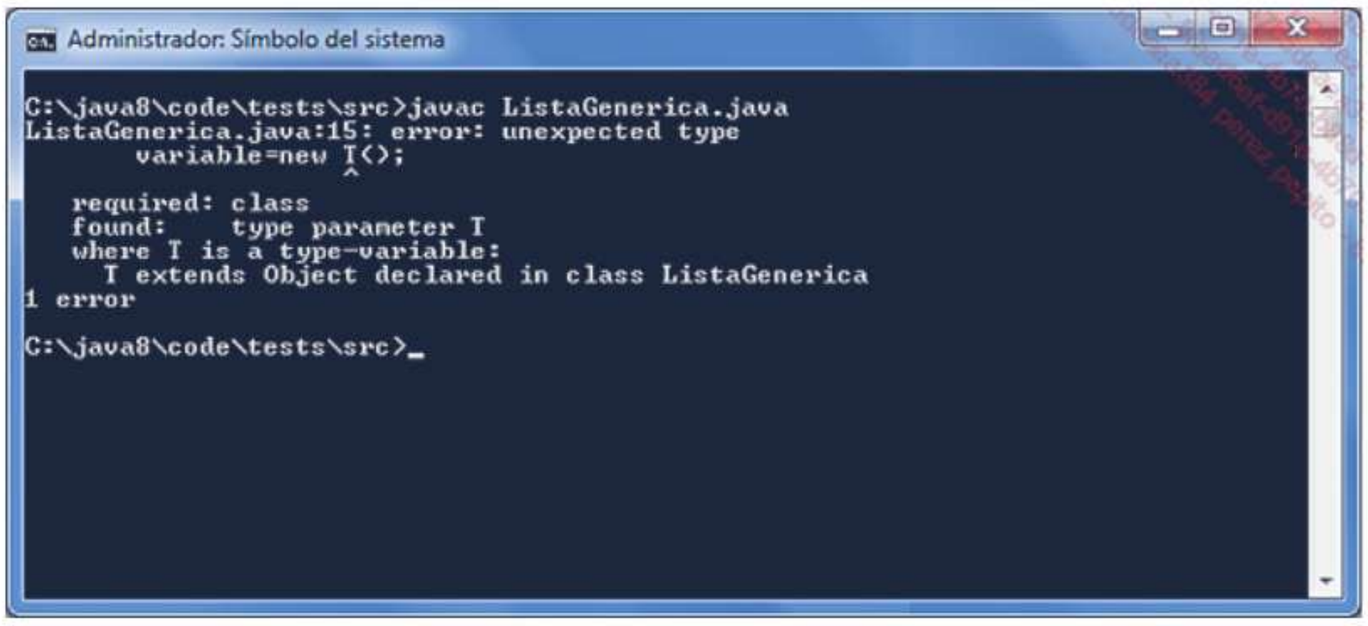
```
ListaGenerica<int> listaEnteros = new ListaGenerica<int>();
```



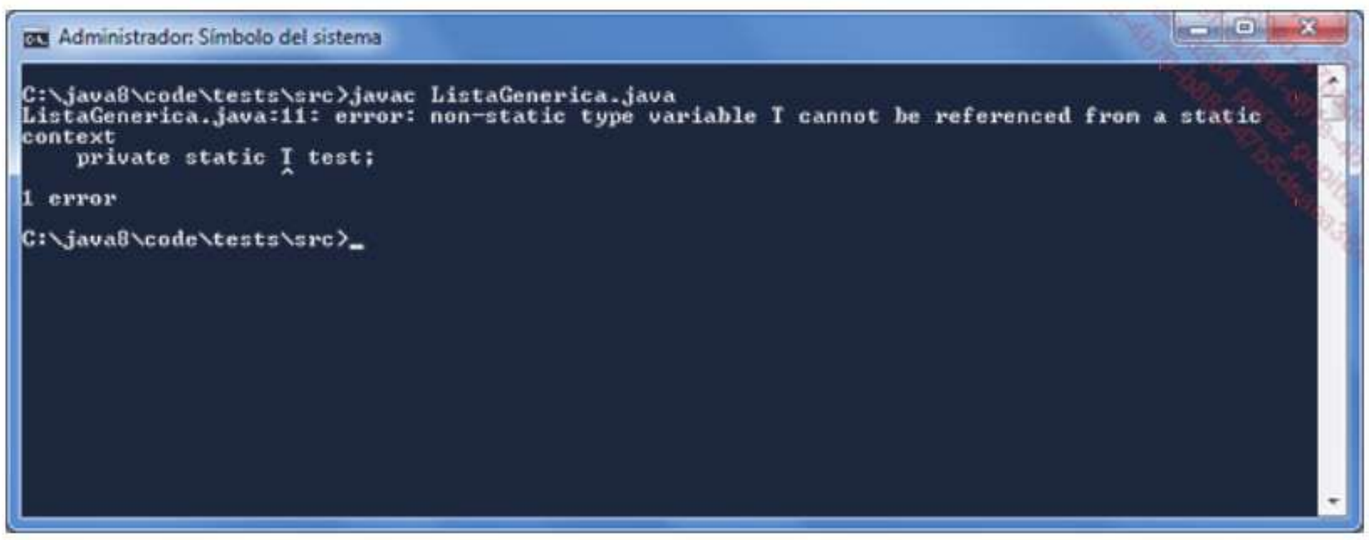
Esta limitación puede evitarse utilizando en su lugar las clases wrapper correspondientes a los tipos simples. En nuestro caso, el uso de la clase `Integer` resuelve el problema. Los mecanismos de autoboxing y de unboxing hacen transparente el uso de los tipos wrapper en lugar de sus tipos simples.

```
     ListaGenerica<Integer> listaEnteros =
new ListaGenerica<Integer>(5);
     listaEnteros.agrega(10);
     listaEnteros.agrega(7);
     listaEnteros.agrega(19);
     listaEnteros.agrega(45);
     listaEnteros.agrega(8);
```

b - El tipo parámetro no puede utilizarse para crear una instancia de clase en el interior de un tipo genérico. El siguiente código está prohibido en el interior de la clase `ListaGenerica`.



c - El tipo de parámetro no puede utilizarse para declarar variables de clase (`static`) en el interior de un tipo genérico.



La variable `test`, única para todas las instancias de la clase `ListaGenerica`, no puede cambiar su tipo en función del tipo de argumento utilizado para la creación de cada una de las instancias. Por ejemplo, con la línea de código siguiente, la variable test sería de tipo `String`.

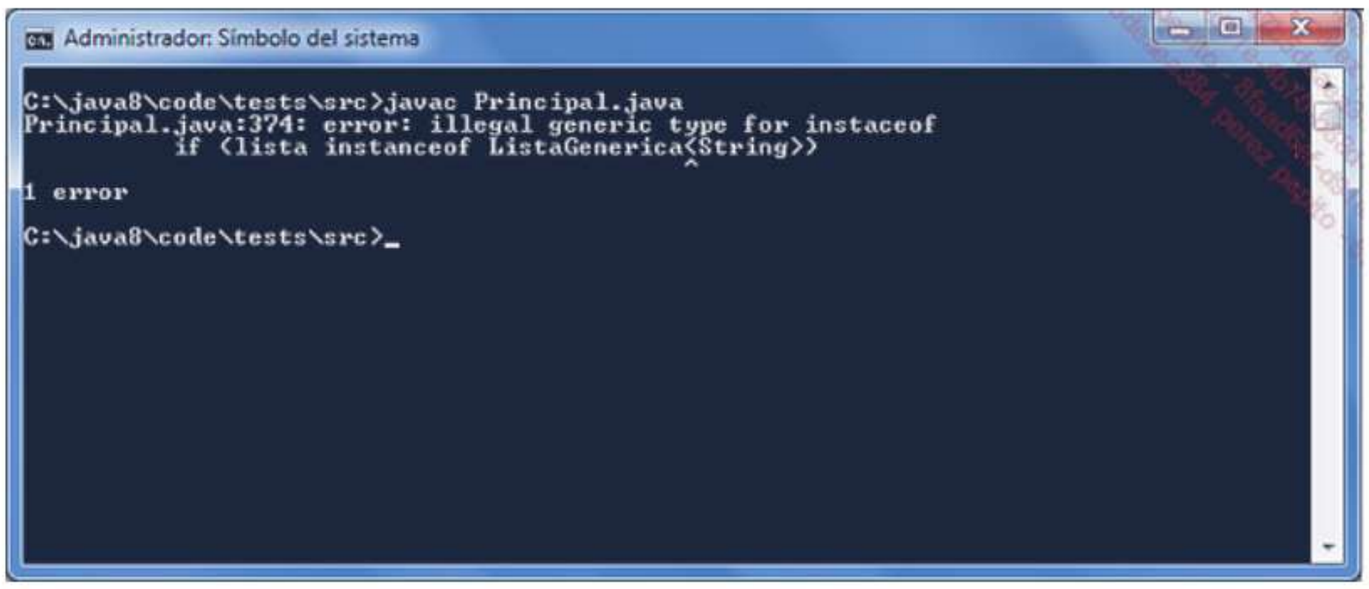
```
ListaGenerica<String> listaCadena=new ListaGenerica<String>(5);
```

Si se crea a continuación otra instancia de la clase `ListaGenerica` con un tipo diferente, será preciso que la variable de clase test cambie de tipo, lo cual resulta imposible.

d - El operador `instanceOf` no puede utilizarse con tipos genéricos. El mecanismo para ocultar el tipo hace desaparecer cualquier traza del tipo argumento, tras la compilación, la siguiente prueba no funcionará.

```
public  static <T extends Clasificable> void ordena(ListaGenerica<T>
lista) throws Exception
{
      int i,j;
      T c;
        if (lista instanceof ListaGenerica<String>)
```

```
        {
            ...
            ...
        }
```



9. Los paquetes

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 además de las estándares (`private`, `protected`, `public`).

a. 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.

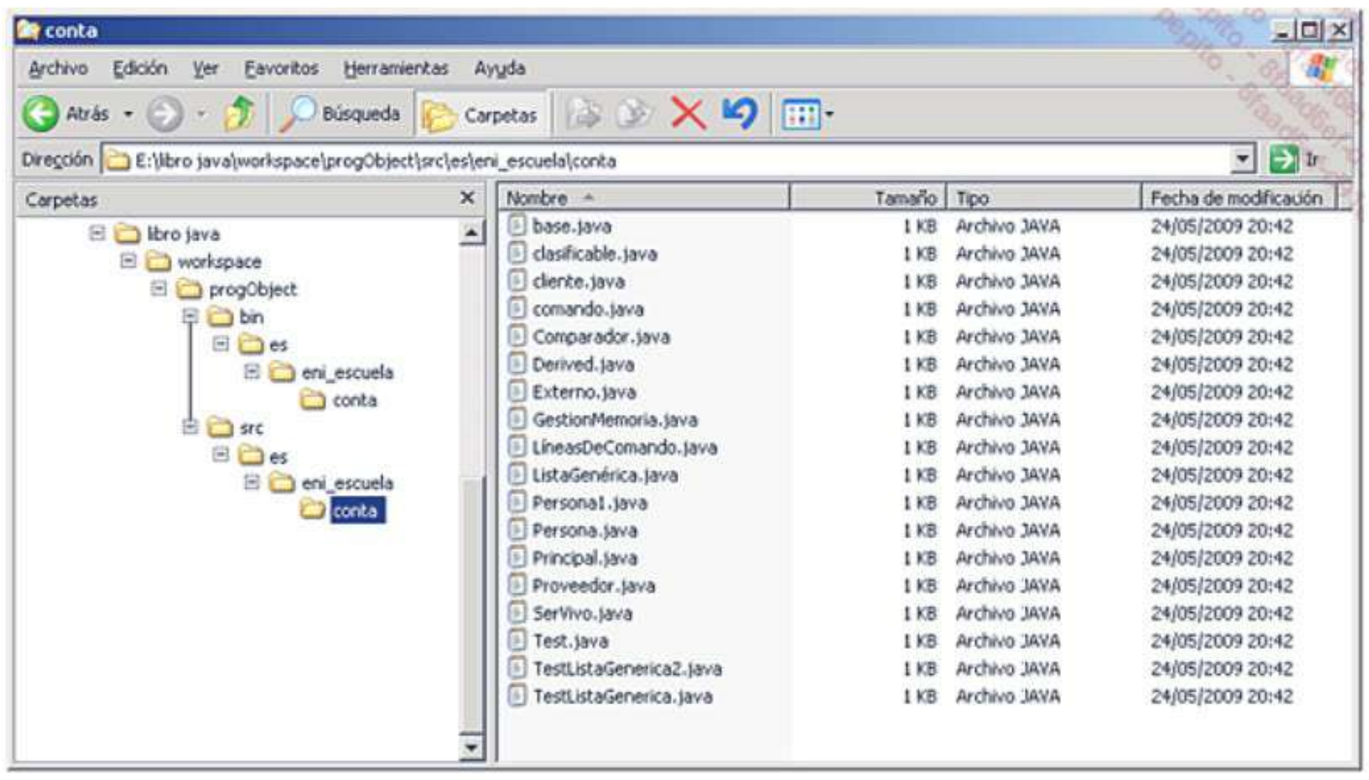
Para asegurar la unicidad del nombre de un paquete, por convención se utiliza su nombre de dominio invirtiendo el orden de los elementos como primera parte para el nombre del paquete.

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 les sustituye 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. 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). Podriamos 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 software. Por convención, los nombres de los paquetes se escriben en minúsculas. Conviene indicar el nombre elegido en el archivo de código fuente precedido de la palabra clave `paquete`. La declaración del paquete se hace obligatoriamente en la primera línea del fichero. 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 los elementos definidos en este fichero serán considerados como formando 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 nombrado.

Entre otros, este es el motivo por el que los diseñadores de Java recomiendan ubicar siempre una clase en un paquete nombrado.

El uso de los paquetes nos 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 ficheros 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 separados 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.

b. Utilización e importación de un paquete

Cuando se utiliza una clase que forma parte de un paquete en una aplicación, se debe utilizar 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
("garcía","pablo",LocalDate.of(1953,11,8),'E');
```

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("garcía","pablo",
LocalDate.of(1953,11,8),'E');
```

Pero, 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("garcía","pablo", LocalDate.of(1953,11,8),'E');
```

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. 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 dejar 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.*;` sólo 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.*;
```

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;

}
}
```

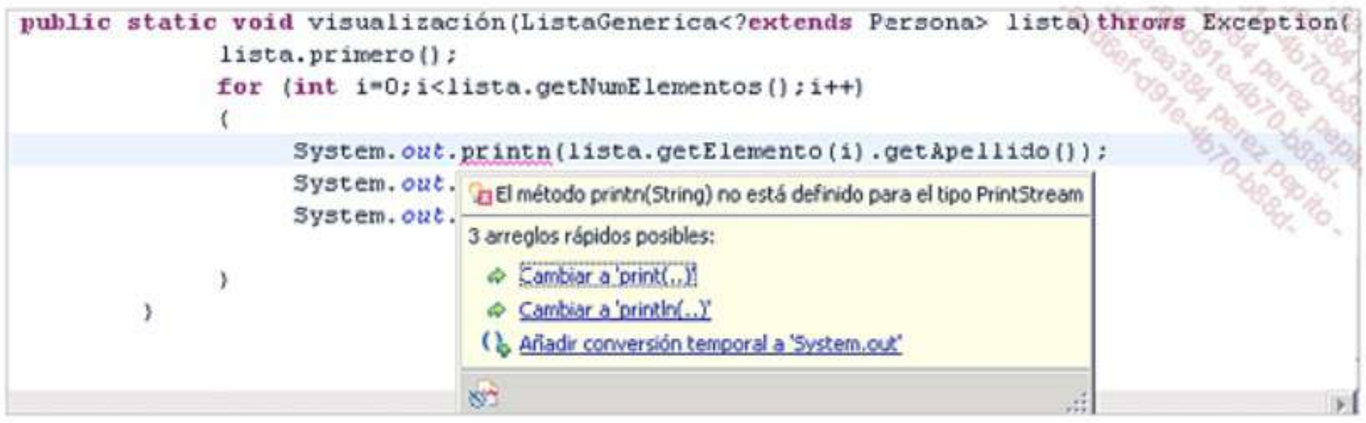
Gestión de las excepciones

¡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.

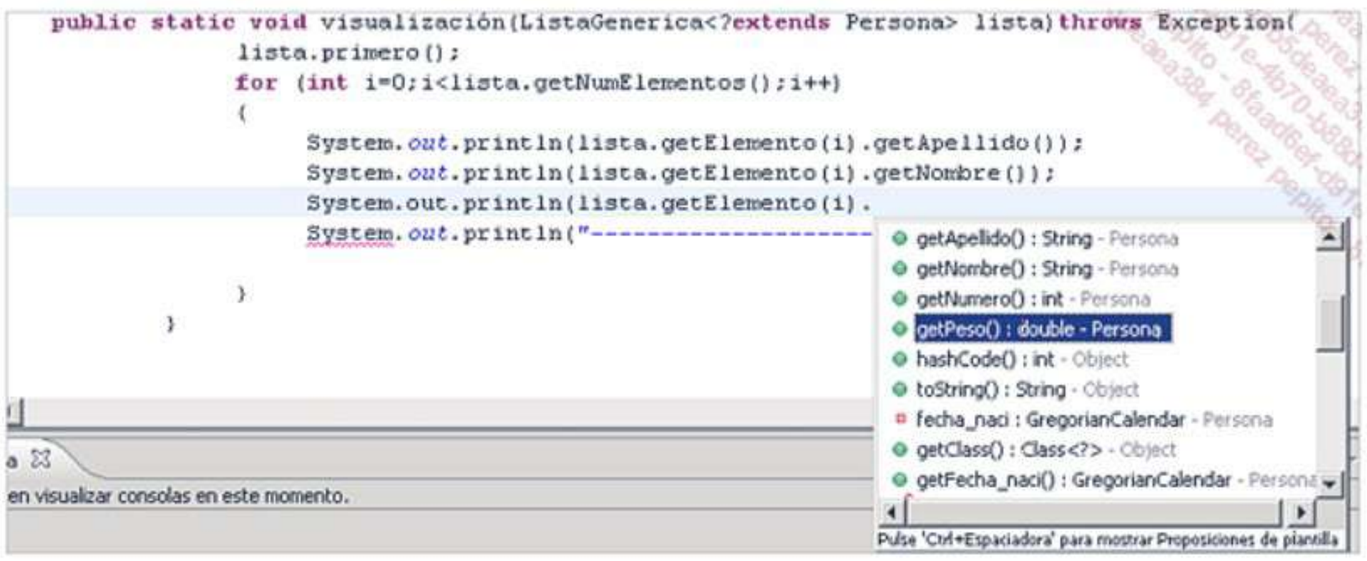
1. Los errores de sintaxis

Este tipo de errores se produce en tiempo de compilación cuando una palabra clave del lenguaje está mal ortografiada. 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. Los siguientes ejemplos se han obtenido a partir del entorno Eclipse.

Si se detecta un error de sintaxis, el entorno propone soluciones que permiten corregir este error.

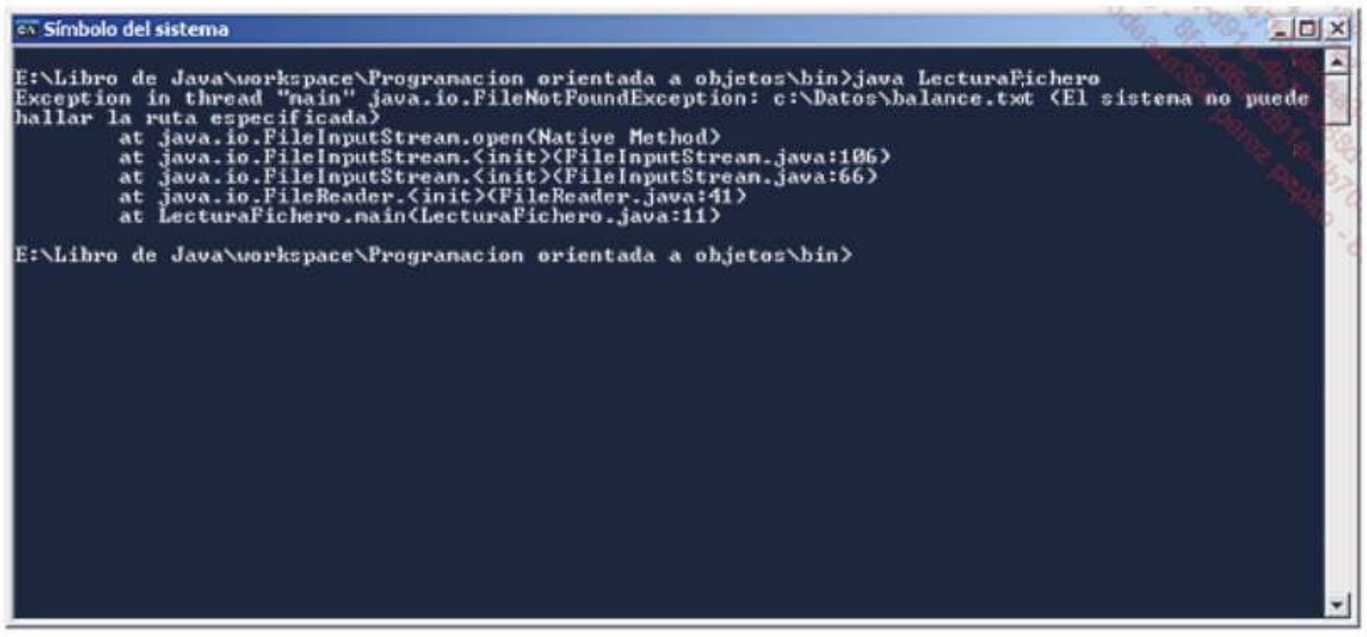


Además, las "faltas de ortografía" en los nombres de campos o métodos se eliminan fácilmente mediante las funcionalidades disponibles en estos entornos.



2. 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. Se da el caso, por ejemplo, si intenta abrir un archivo que no existe en el disco de su máquina. Sin duda, obtendrá un mensaje de este tipo.



¡No es un mensaje muy simpático para el usuario!

Afortunadamente, Java nos permite recuperarnos de este tipo de errores y evita, así, la visualización de este mensaje preocupante. Vamos a detallar esto más adelante en este capítulo.

3. Les errores de lógica

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 nos 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).

a. Las excepciones

Cuando se produce un error durante la ejecución de un método, se crea un objeto `Exception` para representar el error que 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 `Exception` 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.

- Las excepciones verificadas corresponden a una situación anormal durante el funcionamiento de la aplicación. Esta situación suele estar relacionada con algún elemento exterior 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 un 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`. Si bien estas excepciones podrían procesarse dentro de bloques `try catch`, no se recomienda hacerlo. Es preferible analizar el código y modificarlo para evitar que aparezcan.

b. Recuperación de 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 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 misma excepción se activa 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 (excepción1 e1)
{
...
código ejecutado si se produce una excepción de tipo Excepción1
...
}
catch (excepción2 e2)
{
...
código ejecutado si se produce una excepción de tipo Excepción2
...
}
finally
{
...
código ejecutado en cualquier caso antes 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 void leerFichero(String nombre)
{
          FileInputStream fichero=null;
          BufferedReader br=null;
          String línea=null;
          try
          {
               fichero=new FileInputStream("c:\\Datos\\balance.txt");
          }
          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 nuestro ejemplo de la manera siguiente:

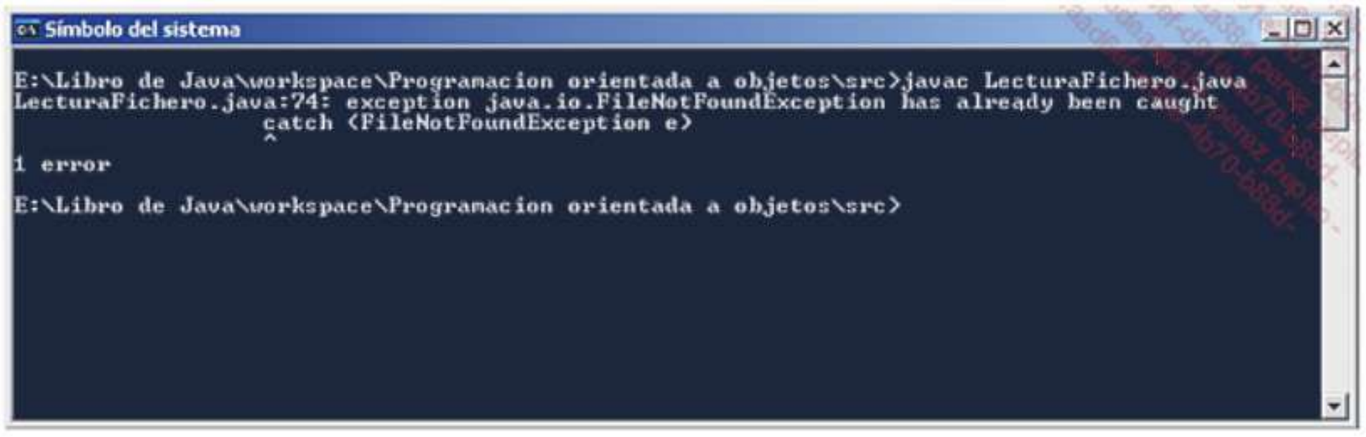
```
public void leerFichero(String nombre)
{
        FileInputStream fichero=null;
        BufferedReader br=null;
        String línea=null;
        try
        {
            fichero=new FileInputStream(nombre);
            br=new BufferedReader(new InputStreamReader(fichero));
            línea=br.readLine();
            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, pero a cambio 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 preve un bloque `catch` para gestionar un tipo particular de excepción, éste 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 detecta la situación y genera un error. Si modificamos nuestro código de la manera siguiente:

```
public 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();
        }


}
```

obtenemos este error en tiempo de compilación.



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 leerFichero(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();
         }
}
```

c. 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, sigue con la explotación del recurso, y 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, éstas 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(IOExcepcion e)
             {
                    e.printStackTrace();
             }
```

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 fichero 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 ficheros históricos de funcionamiento de la aplicación. He aquí un ejemplo que permite grabar esta información en un fichero 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);
          }
     }
}
```

d. 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(String message, Throwable cause)
      {
            super(message,cause);
      }
      public NoFuncionaExcepcion(Throwable cause)
      {
            super(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, a continuación, 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. Cuando, más tarde, se use esta función en otra, tendrá que 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 Java quien las recupera y detiene la aplicación de forma brusca.

Las colecciones

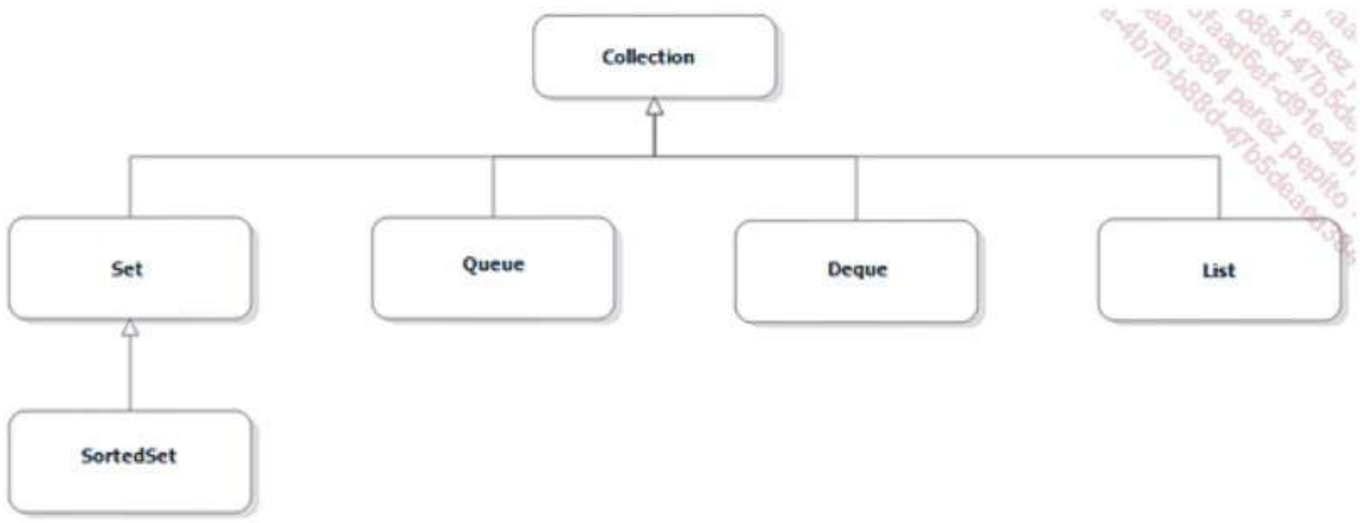
Las aplicaciones necesitan, con frecuencia, manipular una gran cantidad de información. Existen numerosas estructuras que facilitan la gestión de dicha información. Se agrupan bajo el término "colección". Como ocurre en la vida, existen distintos tipos de colecciones y de coleccionadores. Puede haber 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.

Por cada una de estas interfaces existen una o varias clases asociadas. Son las que nos van a interesar.

1. 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 remplaza 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 indicar 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.ListIterator;
public class testArrayList
{
  public static void main(String[] args)
  {
     ArrayList<Persona> lista1;
     ArrayList<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");

     // 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());
     }


     // 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());

     // eliminación selectiva de la lista
     // la expresión lambda determina qué elementos se eliminarán
     lista1.removeIf((Persona pe) ->pe.getFecha_naci().getYear()<1940);
     // recorrer la lista para ver el resultado
     it=lista1.iterator();
     // mientras queden elementos
     while (it.hasNext())
     {
         // recuperar el elemento en curso
         p=it.next();
         System.out.println(p.getApellido()+ " nacido en " +
p.getFecha_naci().getYear() );
     }
     // otra forma de recorrer la lista
     // la expresión lambda se ejecuta para cada
     // elemento de la lista

     lista1.forEach((Persona per)->System.out.println(per.getApellido()+
"  edad " + per.calculaEdad()));
```

```
    }
}
```

2. 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`, éste verifica que no exista un elemento idéntico comparando el `hashCode` del elemento con el de los elementos que ya están presentes en la lista. 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`.

Este método debe respetar una regla básica: si se considera que dos objetos son idénticos, entonces el método `hashCode` debe devolver el mismo valor para cada uno de ellos.

Para mantener la coherencia, es indispensable también redefinir el método `equals` de la clase con los mismos criterios de igualdad.

Para poder almacenar en un `HashSet` instancias de la clase `Persona`, debemos agregar estos dos métodos.

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class Persona      implements Clasificable

{
    private String apellido;
    private String nombre;
    private LocalDate fecha_naci=LocalDate.of(1963,11,29);

    public Persona()
    {
    }

    public Persona(String n,String p,LocalDate d)
    {
        this.apellido=n;
        this.nombre=p;
        this.fecha_naci=d;
    }


    public String getApellido()
    {
        return apellido;
    }

    public void setApellido(String apellido)
    {
        this.apellido = apellido;
    }

    public String getNombre()
    {
        return nombre;
    }

    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }

    public LocalDate getFecha_naci()
    {
        return fecha_naci;
    }

    public void setFecha_naci(LocalDate fecha_naci)
    {
        this.fecha_naci = fecha_naci;
    }

    @Override
    public int hashCode()
    {
        // Seleccionamos dos números impares
        int resultado = 7;
        final int multiplicar = 17;

        // Para cada atributo, se calcula el hashcode
        // que se agrega al resultado tras multiplicar
        // por el número "multiplicar" :
        resultado = multiplicar*resultado + (apellido==null ? 0 :
apellido.hashCode());
        resultado = multiplicar*resultado + (nombre==null ? 0 :
nombre.hashCode());
        resultado = multiplicar*resultado + (fecha_naci==null ? 0 :
fecha_naci.hashCode());

        return resultado;
    }

    @Override
    public boolean equals(Object obj)
    {
        // si el segundo objeto es nulo
        // no puede haber igualdad
        if (obj == null)
```

```
        {
            return false;
        }
        // si ambos objetos no son del mismo tipo
        // no puede haber igualdad
        if (getClass() != obj.getClass())
        {
            return false;
        }
        // es preciso verificar la igualdad de cada
        // atributo
        Persona p = (Persona) obj;
        if (!apellido.equals(p.getApellido()))
            return false;
        if (!nombre.equals(p.getNombre()))
            return false;
        if (!fecha_naci.equals(getFecha_naci()))
            return false;
        return true;
    }

    public long calculaEdad()
    {
       return fecha_naci.until(LocalDate.now(),ChronoUnit.YEARS);
    }

    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;
    }
}
```

Verifiquemos, a continuación, nuestras modificaciones de la clase `Persona` con el siguiente código:

```
public static void main(String[] args)
   {
      Persona p1,p2,p3;
      p1 = new Persona("Wayne", "John",LocalDate.of(1907,5,26));
      p2 = new Persona("McQueen","Steve",LocalDate.of(1930,3,24));
      p3 = new Persona("Wayne", "John",LocalDate.of(1907,5,26));

      System.out.println("hashCode de p1: " + p1.hashCode());
      System.out.println("hashCode de p2: " + p2.hashCode());
      System.out.println("hashCode de p3: " + p3.hashCode());

      if(p1.equals(p2))
          System.out.println("p1 y p2 son idénticos");
      else
          System.out.println("p1 y p2 son diferentes");

      if(p1.equals(p3))
          System.out.println("p1 y p3 son idénticos");
      else
          System.out.println("p1 y p3 son diferentes");

   }
```

Obtenemos el siguiente resultado:

```
hashCode de p1: -1636676846
hashCode de p2: 633943827
hashCode de p3: -1636676846
p1 y p2 son diferentes
p1 y p3 son idénticos
```

lo que confirma nuestras modificaciones.

Podemos, ahora, 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. La unica solución consiste en utilizar el objeto `iterator` asociado al `HashSet`.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.ListIterator;
public class TestHashSet1
```

```
{
    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");

        // 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());
        }
        // borrado selectivo en el HashSet
        // la expresión determina qué elementos se eliminarán
        hash1.removeIf((Persona pe)->pe.getFecha_naci().getYear()<1940);
        // recorrer el HashSet para ver el resultado
        it=hash1.iterator();
        // mientras queden elementos en el HashSet
        while (it.hasNext())
        {
            // recuperar el elemento en curso
            p=it.next();
            System.out.println(p.getApellido()+ " nacido en " +
p.getFecha_naci().getYear() );
        }
        // otra forma de recorrer el HashSet
        // la expresión lambda se ejecuta para cada
        // elemento del HashSet

        hash1.forEach((Persona per)->System.out.println(
per.getApellido()+ " edad " + per.calculaEdad()));

      }
   }
```

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 traaducir 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 TestHashSet
{
    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("agunos 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());
    }

    }
}
```

3. 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();
    }
}
```

4. Streams y pipelines

Generalmente, las colecciones se utilizan para almacenar información y recuperarla más adelante, o para realizar algún procesamiento sobre ella. La técnica clásica para realizar operaciones sobre los elementos contenidos en una colección consiste en realizar un bucle `for` o en utilizar el `iterator` asociado a la colección. Esta solución puede remplazarse por el uso de `stream` y de `pipeline`. Para ilustrar el funcionamiento de estos dos elementos podemos establecer la correspondencia con el funcionamiento de una fábrica. Nuestra fábrica recibe la materia prima, que atraviesa distintas unidades de transformación para obtener, a la salida, el producto terminado. En nuestro caso, los datos presentes en la colección representan la materia prima. Se transporta mediante objetos que implementan la interfaz `Stream`. Las unidades de transformación se representan mediante `Pipelines`. Éstos pueden producir un resultado directamente explotable (el producto terminado) o bien otro objeto `Stream` que va a alimentar a un nuevo `Pipeline`.

Todas las clases que implementan la interfaz `Collection` son capaces de proveer un objeto de tipo `Stream`. El objeto es responsable de iterar sobre los elementos presentes en la colección.

Los `Pipelines` son un poco más complejos, pues están formados por varios elementos.

El primero representa el origen desde el que el `Pipeline` va a obtener la información sobre la que va a realizar los procesamientos. Es, por supuesto, un objeto `Stream`, que será la fuente de un `Pipeline`. Los procesamientos se representan mediante los distintos métodos definidos en la interfaz `Stream`. Estos métodos se corresponden con los distintos tipos de procesamiento que pueden realizarse sobre los datos. Por defecto, estos métodos no hacen nada. Su comportamiento debe adaptarse en función del tipo de objeto que tienen que procesar. Generalmente se utiliza para ello una expresión lambda o una referencia de método. Estos métodos pueden generar el resultado final, otro objeto `Stream` formado por objetos del mismo tipo que los objetos originales o bien otro `Stream` que contenga otro tipo de datos. Podemos, por ejemplo, citar los siguientes métodos como representativos de cada uno de los casos típicos.

`boolean allMatch(Predicate<? super T> predicate)`: esta función devuelve un valor booleano que indica si todos los elementos se corresponden con la condición indicada por el objeto `Predicate`.

`IntStream mapToInt(ToIntFunction<? super T> mapper)`: esta función devuelve un objeto `Stream` de valores enteros generados por el resultado de la función `ToIntFunction`.

`Stream<T> filter(Predicate<? super T> predicate)`: esta función genera un nuevo objeto `Stream` que contiene, únicamente, los objetos del flujo original que cumplen la condición definida por el objeto `Predicate`.

La interfaz `Stream` contiene varias decenas de métodos que responden a necesidades más habituales. Resulta, por tanto, primordial consultar la documentación correspondiente.

El siguiente ejemplo muestra algunas de las posibilidades disponibles.

```
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
public class TestStream
{
    public static void main(String[] args)
    {
       ArrayList<Persona> lista;

       // creación de las dos instancias
       lista=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));

        // se agregan cinco personas a la lista
        lista.add(p1);
        lista.add(p2);
        lista.add(p3);
        lista.add(p4);
        lista.add(p5);

        // la función allMatch devuelve true si todos los elementos
        // de la lista cumplen con la condición expresada
        // en la expresión lambda


        if(lista.stream().allMatch(p->p.getFecha_naci().getYear()>1945))
          System.out.println("todas las personas han nacido después
de 1945");
        else
           System.out.println("algunas personas han nacido antes de 1945");

        // filtrado de personas nacidas el mes de marzo
        // este filtrado genera un nuevo Stream
        // forEach recorre este nuevo Stream y
        // ejecuta la expresión lambda para cada elemento

        lista.stream().filter(p->p.getFecha_naci().getMonthValue()
==3).forEach(p->System.out.println(p.getApellido()));

        // busca la persona mayor de la lista
        // la expresión lambda representa la implementación
        // de la interfaz Comparador

        System.out.println(lista.stream().max((pe1,pe2)->
               {
                      if (pe1.calculaEdad()>pe2.calculaEdad())
                          return 1;
                      if (pe1.calculaEdad()<pe2.calculaEdad())
                          return -1;
                      return 0;
               }).get().getApellido());

        // calcula la edad media de las personas presentes en la
        // lista mapToLong y 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 transforma en tipo double
        double edadMedia = lista
      .stream()
      .mapToLong(Persona::calculaEdad)
      .average()
      .getAsDouble();

        System.out.println("Edad media de las personas de la lista:
" + edadMedia + " años");
    }
}
```

Ejercicios

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.

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.

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.

Ejercicio 4

Modificar el código anterior para poder obtener rápidamente la lista de artículos relacionados con un autor o un realizador.

Correcciones

Corrección del 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 sólo 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 utilizar los métodos setXXXX para acceder, 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 getXXXX y setXXXX 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 `Articulo`.

```
/******************** Articulo **********************/
1. package ejercicios.capitulo3.ejercicio1;
2.
3. public class Articulo
4. {
5.     private int referencia;
6.     private String designación;
7.     private double precio;
8.
9.     public Articulo()
10.        {
11.            super();
12.        }
13.
14.        public Articulo(int referencia)
15.        {
16.            this();
17.            setReferencia(referencia);
18.        }
19.        public Articulo(int referencia,String designación)
20.        {
21.            this(referencia);
22.            setNombre(designación);
23.        }
24.
25.        public Articulo(int referencia,String designación,
double precio)
26.        {
27.            this(referencia,designación);
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 getDesignación()
42.        {
43.            return designación;
44.        }
45.
46.        public void setDesignación(String designación)
47.        {
48.            this.designación = designación;
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() + " " + getDesignación() +
" " + getPrecio();
64.        }
65.
66.        }
67.
/****************** Clase Principal *******************/
1. package ejercicios.capitulo3.ejercicio1;
2.
```

```
3.
4. public class Principal
5. {
6.     public static void main(String[] args)
7.     {
8.         Articulo a1,a2;
9.         a1=new Articulo();
10.        a1.setReferencia(100);
11.        a1.setNombre("Mortadelo y Filemón");
12.        a1.setPrecio(8.5);
13.
14.        a2=new Articulo(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(Articulo a)
25.    {
26.        System.out.println("referencia: " + a.getReferencia());
27.        System.out.println("designación: "  +
a.getDesignación());
28.        System.out.println("precio: " + a.getPrecio() + " €");
29.    }
30. }
```

Corrección del ejercicio 2

Las dos clases `Libro` y `Dvd` heredan de la clase `Articulo` 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 `Articulo`. Se agregan los atributos correspondientes exigidos por las clases `Libro` y `Dvd`. Los demás atributos y métodos se heredan de la clase `Articulo`. La llamada a los métodos de la clase `Articulo` se realiza mediante la palabra clave `super`.

```
/********************* Articulo **********************/
1. package ejercicios.capitulo3.ejercicio2;
2.
3. public class Articulo
4. {
5.     private int referencia;
6.     private String designación;
7.     private double precio;
8.
9.     public Articulo()
10.         {
11.             super();
12.         }
13.
14.    public Articulo(int referencia)
15.    {
16.            this();
17.            setReferencia(referencia);
18.    }
19.    public Articulo(int referencia,String designación)
20.    {
21.            this(referencia);
22.            setDesignación(designación);
23.    }
24.
25.    public Articulo(int referencia,String designación,
double precio)
26.    {
27.            this(referencia,designación);
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 getDesignación()
42.    {
43.            return designación;
44.    }
45.
46.    public void setDesignación(String designación)
47.    {
48.            this.designación = designación;
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() + " " + getDesignación() +
" " + getPrecio();
63.    }
64.
65.   }
66.
/******************** Libro **********************/
1. package ejercicios.capitulo3.ejercicio2;
2.
3. public class Libro extends Articulo
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 designación,
double precio,String isbn,int numPaginas,String autor)
15.         {
16.             super(referencia,designación,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 String getAutor() {
39.             return autor;
40.         }
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 ejercicios.capitulo3.ejercicio2;
2.
3. import java.time.Duration;
4.
5.
6. public class Dvd extends Articulo
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() + " " +
getDuracion().toMinutes() + " " + getRealizador();
47.          }
48.    }
```

/********** Clase Principal **********/

```
1. package ejercicios.capitulo3.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.        testLibro(l);
18.        System.out.println(l.toString());
19.
20.        d=new Dvd();
21.        d.setReferencia(110);
22.        d.setNombre("La escopeta nacional");
23.        d.setPrecio(24.62);
24.        d.setDuracion(Duration.ofMinutes(91));
25.        d.setRealizador("Berlanga");
26.        testDvd(d);
27.        System.out.println(d.toString());
28.
29.    }
30. }
```

Corrección del ejercicio 3

Para agregar información suplementaria relativa al autor o al realizador, es preferible crear una clase `Persona` que agrupe el 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 getXXXX y setXXXX para tener en cuenta dicha modificación.

/********** Articulo **********/

```
1. package ejercicios.capitulo3.ejercicio3;
2.
3. public class Articulo
4. {
5.     private int referencia;
6.     private String designación;
7.     private double precio;
8.
9.     public Articulo()
10.        {
11.            super();
12.        }
13.
14.    public Articulo(int referencia)
15.    {
16.            this();
17.            setReferencia(referencia);
18.    }
19.    public Articulo(int referencia,String designación)
20.    {
21.            this(referencia);
22.            setDesignación(designación);
23.    }
24.
25.    public Articulo(int referencia,String designación,double precio)
26.    {
27.            this(referencia,designación);
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 getDesignación()
42.    {
43.           return designación;
44.    }
45.
46.    public void setDesignación(String designación)
47.    {
48.           this.designación = designación;
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() + " " + getDesignación() +
" " + getPrecio();
63.    }
64.
65. }
```

/********** Libro **********/

```
1. package ejercicios.capitulo3.ejercicio3;
2.
3. public class Libro extends Articulo
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 designación,double
precio,String isbn,int numPaginas,Persona autor)
15.        {
16.           super(referencia,designación,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 ejercicios.capitulo3.ejercicio3;
2.
3. import java.time.Duration;
4.
5.
6. public class Dvd extends Articulo
7. {
8.     private Duration duracion;
9.     private Persona realizador;
10.
11.        public Dvd()
12.    {
13.        super();
14.    }
15.
```

```
16.     public Dvd(int referencia,String designación,double
precio,Duration duracion,Persona realizador)
17.     {
18.         super(referencia,designación,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 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 ejercicios.capitulo3.ejercicio3;
2.
3. import java.time.LocalDate;
4.
5. public class Persona
6. {
7.     private String apellido;
8.     private String nombre;
9.     private LocalDate fecha_naci;
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.fecha_naci=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 getFecha_naci()
44.     {
45.         return fecha_naci;
46.     }
47.
48.     public void setFecha_naci(LocalDate fecha_naci)
49.     {
50.         this.fecha_naci = fecha_naci;
51.     }
52.
53.     public String toString() {
54.         return nombre + " " + apellido;
55.     }
56.
57. }
```

/********** Clase Principal **********/

```
1. package ejercicios.capitulo3.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: " +
a.getReferencia());
37.         System.out.println("designación: " +
a.getDesignación());
38.         System.out.println("precio: " + a.getPrecio() + " €");
39.     }
40.
41.     public static void testLibro(Libro l)
42.     {
43.         test(l);
44.         System.out.println("número de páginas: " +
l.getNumPaginas());
45.         System.out.println("autor: " +
l.getAutor().toString());
46.     }
47.
48.     public static void testDvd(Dvd d)
49.     {
50.         test(d);
51.         System.out.println("duración: " +
d.getDuracion().toMinutes() + " minutos");
52.         System.out.println("realizador: " +
d.getRealizador().toString());
53.     }
54. }
```

Corrección del 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, simplemente, con agregar a la clase Persona un atributo de tipo `Arraylist`. Éste 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.

```
/********************* Articulo **********************/

1. package ejercicios.capitulo3.ejercicio4;
2.
3. public class Articulo
4. {
5.     private int referencia;
6.     private String designación;
7.     private double precio;
8.
9.     public Articulo()
10.     {
11.         super();
12.     }
13.
14.     public Articulo(int referencia)
15.     {
16.         this();
17.         setReferencia(referencia);
18.     }
19.     public Articulo(int referencia,String designación)
20.     {
21.         this(referencia);
22.         setNombre(designación);
23.     }
```

```
24.
25.     public Articulo(int referencia,String designación,
double precio)
26.     {
27.         this(referencia,designación);
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 getDesignación()
42.     {
43.         return designación;
44.     }
45.
46.     public void setNombre(String designación)
47.     {
48.         this.nombre = designación;
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() + " " + getDesignación() + "
" + getPrecio();
63.     }
64.
65. }
```

/********** Libro **********/

```
1. package ejercicios.capitulo3.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 designación,double
precio,String isbn,int numPaginas,Persona autor)
19.         {
20.             super(referencia,designación,precio);
21.             setIsbn(isbn);
22.             setNumPaginas(numPaginas);
23.             setAutor(autor);
24.         }
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.             this.autor = autor;
48.             ArrayList<Articulo> lst;
49.             lst=autor.getObras();
50.             if (!lst.contains(this))
51.             {
```

```
52.                 lst.add(this);
53.             }
54.         }
55.         public String toString()
56.         {
57.             return super.toString() + " " + getNumPaginas() + "
" + getAutor();
58.         }
59.
60.     }
```

/********** Dvd **********/

```
1. package ejercicios.capitulo3.ejercicio4;
2.
3. import java.time.Duration;
4. import java.util.ArrayList;
5.
6.
7. public class Dvd extends Articulo
8. {
9.     private Duration duracion;
10.         private Persona realizador;
11.
12.         public Dvd()
13.         {
14.             super();
15.         }
16.
17.         public Dvd(int referencia,String designación,double
precio,Duration duracion,Persona realizador)
18.         {
19.             super(referencia,designación,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.            this.realizador = realizador;
43.         ArrayList<Articulo> lst;
44.         lst=realizador.getObras();
45.         if (!lst.contains(this))
46.         {
47.             lst.add(this);
48.         }
49.     }
50.
51.   public String toString()
52.     {
53.         return super.toString() + " " +
getDuracion().toMinutes() + " " + getRealizador();
54.     }
55. }
```

/********** Persona **********/

```
1. package ejercicios.capitulo3.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 fecha_naci;
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.fecha_naci=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 getFecha_naci()
48.         {
49.          return fecha_naci;
50.         }
51.
52.         public void setFecha_naci(LocalDate fecha_naci)
53.         {
54.          this.fecha_naci = fecha_naci;
55.         }
56.
57.         public ArrayList<Articulo> getObras()
58.         {
59.             return obras;
60.         }
61.
62.         public String toString() {
63.             return nombre + " " + apellido;
64.         }
65.  }
```

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, sólo conservaremos nuestro propio editor de texto, un compilador y la máquina virtual.

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.

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 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 cuyas diferentes ramas están constituídas 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. Se estudia este elemento en detalle en el capítulo correspondiente.

`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;
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:

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 el párrafo dedicado 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.

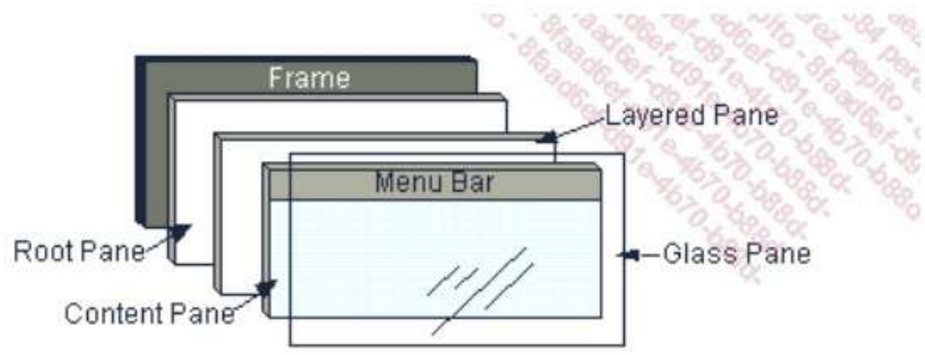
Class JFrame

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame
                  └ javax.swing.JFrame
```

All Implemented Interfaces:
ImageObserver, MenuContainer, Serializable, Accessible, RootPaneContainer, WindowConstants

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.

Ahora que somos capaces de visualizar una ventana, el grueso del trabajo va a consistir en añadir un contenido a la ventana. 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 éstos 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`. Es técnicamente 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`.

Por lo tanto, 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;

import java.awt.Graphics;

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.

2. 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 transmitirlas 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 qué 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 `MouseMotionListener` 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 MouseMotionListener
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.

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 `XXXXXListener` una clase abstracta correspondiente que implementa la interfaz, y que contiene los métodos exigidos por la interfaz. 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:

- Crear una clase "normal" que herede de una de estas clases.
- Crear una clase interna que herede de una de estas clases.
- Crear 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. Este ejemplo 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 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;

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);
     }
}


package es.eni;

public class Main {

     public static void main(String[] args)

     {
          // creación de la ventana
          Pantalla ventana;
          ventana=new Pantalla();
          // visualización de la ventana
          ventana.setVisible(true);
     }
}
```

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.

Utilización de una clase "normal" que implementa la interfaz

```
package es.eni;

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;

public class Main {

     public static void main(String[] args)

     {
          // creación de la ventana
          Pantalla ventana;
          ventana=new Pantalla();
          // 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
            ventana.addWindowListener(ev);
            // visualización de la ventana
            ventana.setVisible(true);
      }
}
```

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;

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 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;

public class Main {

      public static void main(String[] args)

      {
            // creación de la ventana
            Pantalla ventana;
            ventana=new Pantalla();
            // visualización de la ventana
            ventana.setVisible(true);
      }
}
```

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;

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 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;

public class Main {

     public static void main(String[] args)

     {
          // creación de la ventana
          Pantalla ventana;
          ventana=new Pantalla();
          // visualización de la ventana
          ventana.setVisible(true);
     }
}
```

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;

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
}// fin de la clase Pantalla

package es.eni;

public class Main {

      public static void main(String[] args)

      {
           // creación de la ventana
           Pantalla ventana;
           ventana=new Pantalla();
           // visualización de la ventana
           ventana.setVisible(true);
      }
}
```

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.

Por el contrario, 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 XXXXAdapter

```
package es.eni;

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;

public class Main {

      public static void main(String[] args)

      {
            // creación de la ventana
            Pantalla ventana;
            ventana=new Pantalla();
            // 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
            ventana.addWindowListener(ev);
            // visualización de la ventana
            ventana.setVisible(true);
      }
}
```

Crear una clase interna que herede de una clase XXXXAdapter

```
package es.eni;

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();
            // como receptor de eventos para la ventana

            addWindowListener(ev);
      }

      public class EscuchadorVentana extends WindowAdapter
      {
            public void windowClosing(WindowEvent arg0)
            {
                  System.exit(0);
            }
      }
}

package es.eni;

public class Main {

      public static void main(String[] args)

      {
            // creación de la ventana
            Pantalla ventana;
            ventana=new Pantalla();
            // visualización de la ventana
            ventana.setVisible(true);
```

```
      }
}
```

Crear una clase interna anónima que herede de una clase XXXXAdapter

```
package es.eni;

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 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
}// fin de la clase Pantalla

package es.eni;

public class Main {

      public static void main(String[] args)

      {
            // creación de la ventana
            Pantalla ventana;
            ventana=new Pantalla();
            // visualización de la ventana
            ventana.setVisible(true);
      }
}
```

Por supuesto, esta solución es la más económica en número de líneas y también la que utilizan 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 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 a 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 al modificar el color de fondo correspondiente al botón o al menú usado. Como debemos utilizar el mismo receptor para dos fuentes de eventos, es preferible utilizar una clase interna para la creación del receptor.

A continuación, presentamos el código correspondiente.

```
package es.eni;

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 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 al mismo método `actionPerformed` con un clic en cualquiera de los botones. La elección de la acción a 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;
```

```
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 los tres receptores 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 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 en 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 titulo 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;

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");
           btnRojo.setActionCommand("red");
           btnVerde=new JButton("Verde");
           btnVerde.setActionCommand("green");
           btnAzul=new JButton("Azul");
           btnAzul.setActionCommand("blue");
           // creación de los tres receptores de eventos
           ReceptorColor ec;
           ec=new ReceptorColor();
           // 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);
           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);
           // 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 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 eventualmente 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;

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;

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(0,0,300,100);
            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 los tres receptores de eventos
            ReceptorColor ec;
```

```
            ec=new ReceptorColor();
            // asociamos 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");
             // agregamos 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())
                        {
                              // agregamos 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
                        {
                        // eliminamos 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);
            // añadido del 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 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.

```
27/11/2008 09:19:43 clic en el botón Rojo
27/11/2008 09:19:45 clic en el botón Verde
27/11/2008 09:19:47 clic en el botón Azul
27/11/2008 09:19:51 clic en el menú Rojo
27/11/2008 09:19:54 clic en el menú Verde
27/11/2008 09:19:56 clic en el menú Azul
```

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 cuales 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 `addXXXXXListener` 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:
JDK1.1
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`.

Method Summary

void	`mouseClicked(MouseEvent e)` Invoked when the mouse button has been clicked (pressed and released) on a component.
void	`mouseEntered(MouseEvent e)` Invoked when the mouse enters a component.
void	`mouseExited(MouseEvent e)` Invoked when the mouse exits a component.
void	`mousePressed(MouseEvent e)` Invoked when a mouse button has been pressed on a component.
void	`mouseReleased(MouseEvent e)` 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.

Method Summary

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` 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.

3. 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 del componente. 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 forma 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.

```
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
```

- 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.

4. El posicionamiento de los componentes

Si ya ha utilizado otro lenguaje de programación que permite desarrollar interfaces gráficas, una cosa le debe haber parecido extraña en los ejemplos de código que hemos utilizado.

- En ninguna parte hemos indicado el tamaño y la posición de los componentes utilizados.
- 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, éste 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 a su 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 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 quien 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 al indicar 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 el centro del contenedor, deja un espacio horizontal de cincuenta píxeles y un espacio vertical de veinte píxeles entre los componentes.

```
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.

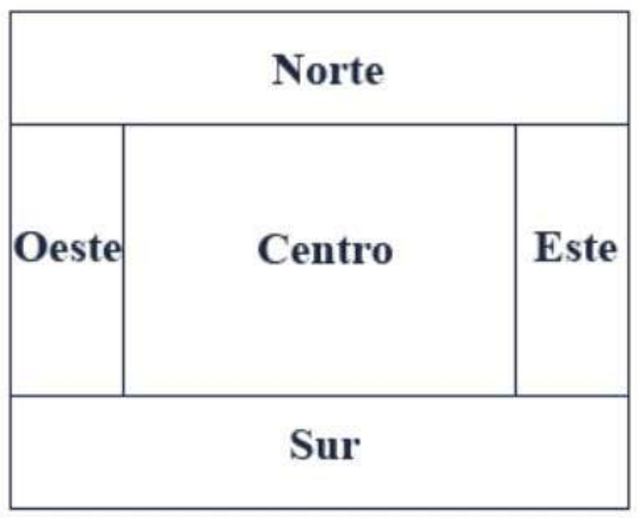
- `setHgap`: para especificar el espaciado horizontal entre los componentes.
- `setVgap`: para especificar el espaciado vertical entre los componentes.

En este caso, es necesario obtener una referencia sobre el `FlowLayout` asociado al `JPanel` empleando el método `getLayout` del mismo. 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.

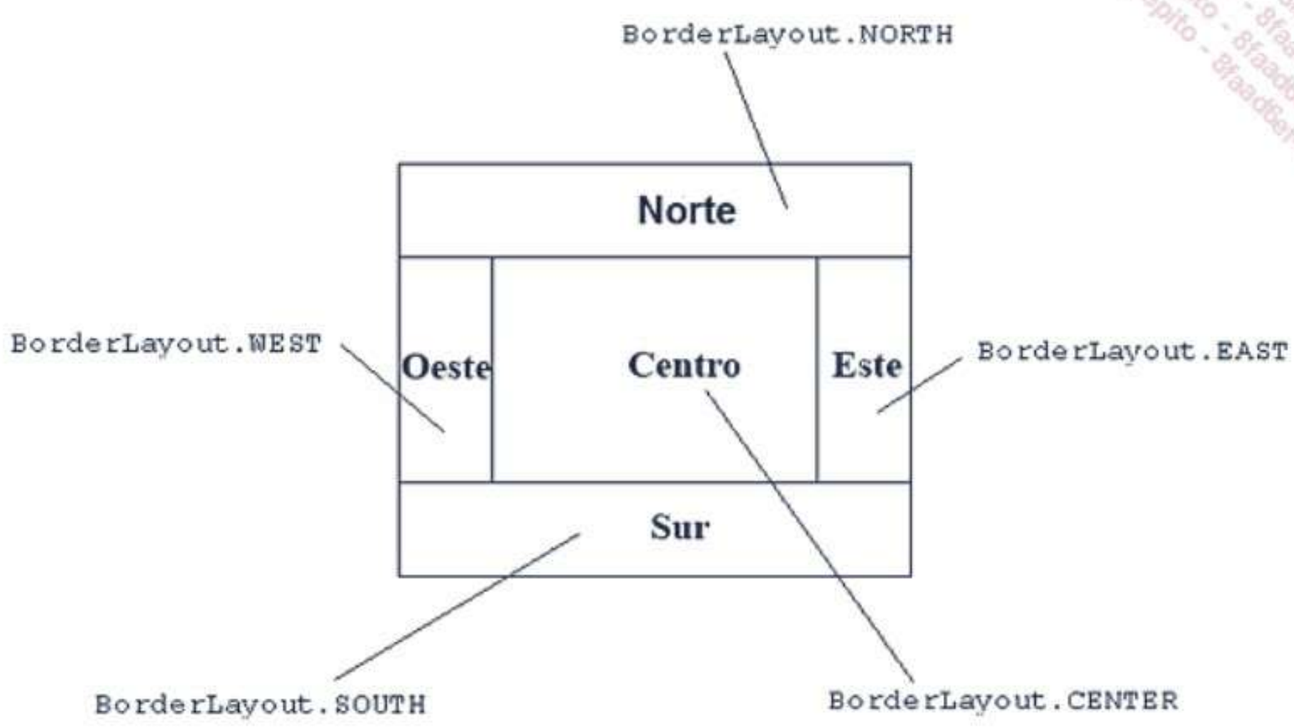
```
((FlowLayout)panel.getLayout()).setAlignment(FlowLayout.LEFT);
((FlowLayout)panel.getLayout()).setHgap(50);
((FlowLayout)panel.getLayout()).setVgap(20);
```

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 a 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, éste se ubica sistemáticamente en la zona Centro.

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, 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` sólo puede contener un único componente. Si, a pesar de todo, se añaden varios componentes a una misma zona de un `BorderLayout`, éste sólo se encargará del último añadido y los demás no serán visibles.

Por esta limitación, se suele utilizar el `BorderLayout` para posicionar contenedores en comparación 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 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 a 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 a marcar 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;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
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 Pantalla9 extends JFrame

{
      JPanel panelBotones;
      JPanel panelChk;
      JPanel panelColor;
      JButton btnRojo,btnVerde,btnAzul;
      JMenuItem mnuRojo,mnuVerde,mnuAzul;
      ConsoleLog lg;
      public Pantalla9()
      {
            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");
            btnRojo.setActionCommand("red");
            btnVerde=new JButton("Verde");
            btnVerde.setActionCommand("green");
            btnAzul=new JButton("Azul");
            btnAzul.setActionCommand("blue");
            // creación de los tres receptores 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);
                 }
            }
            );
            ((FlowLayout)panelBotones.getLayout()).setAlignment
(FlowLayout.LEFT);
            ((FlowLayout)panelBotones.getLayout()).setHgap(50);
            ((FlowLayout)panelBotones.getLayout()).setVgap(20);
            }
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);
            }
       }
}

}
```

Nuestra ventana presenta ahora la disposición siguiente:

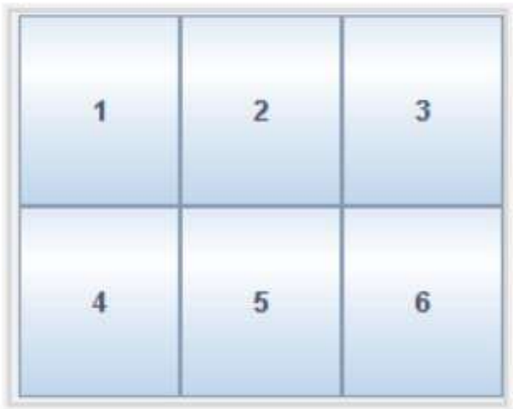
c. GridLayout

Este renderizador organiza la superficie que se le confía con la forma de una malla invisible. Se debe indicar el número de líneas y columnas en la llamada al constructor. Cuando este renderizador ubica los componentes que se le confían, empieza por rellenar todas las casillas de la primera línea y, a continuación, sigue con la línea 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`, éste creará casillas adicionales aumentando el número de columnas pero respetando el número de líneas de origen. Por ejemplo, con el código siguiente prevemos una malla de 2x2 casillas pero añadimos seis componentes.

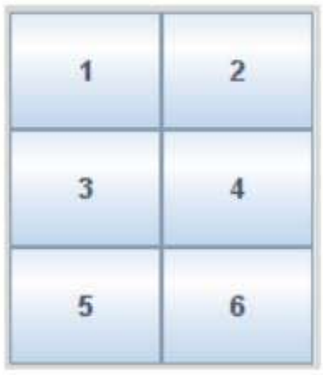
```
JPanel malla;
malla=new JPanel();
GridLayout grl;
grl=new GridLayout(2,2);
malla.setLayout(grl);
malla.add(new JButton("1"));
malla.add(new JButton("2"));
malla.add(new JButton("3"));
malla.add(new JButton("4"));
malla.add(new JButton("5"));
malla.add(new JButton("6"));
panel.add(malla);
```

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 líneas y, para ello, debemos especificar el valor 0 para este argumento en el constructor. En este caso, el renderizador añade líneas adicionales para los componentes en exceso.

```
JPanel malla;
malla=new JPanel();
GridLayout grl;
grl=new GridLayout(0,2);
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"));
panel.add(malla);
```



d. BoxLayout

Se utiliza este renderizador cuando se necesita organizar componentes en una línea o una columna. Aunque se pueda obtener el mismo resultado con un `GridLayout` de una única línea 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 si este contenedor gestiona una línea (`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.

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 favoritas 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.

```
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);
```

Este código nos devuelve 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.

Se cuenta con 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 el 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`.

```
JPanel línea;
línea=new JPanel();
BoxLayout bl;
bl=new BoxLayout(línea,BoxLayout.X_AXIS);
línea.setLayout(bl);
JButton b1,b2;
  // 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);
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);
```

e. GridBagLayout

Puede considerar el `GridBagLayout` como el "súper" renderizador 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 líneas 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 a 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 de este objeto:

- **gridx**: columna de la esquina superior izquierda del componente.
- **gridy**: línea de la esquina superior izquierda del componente.

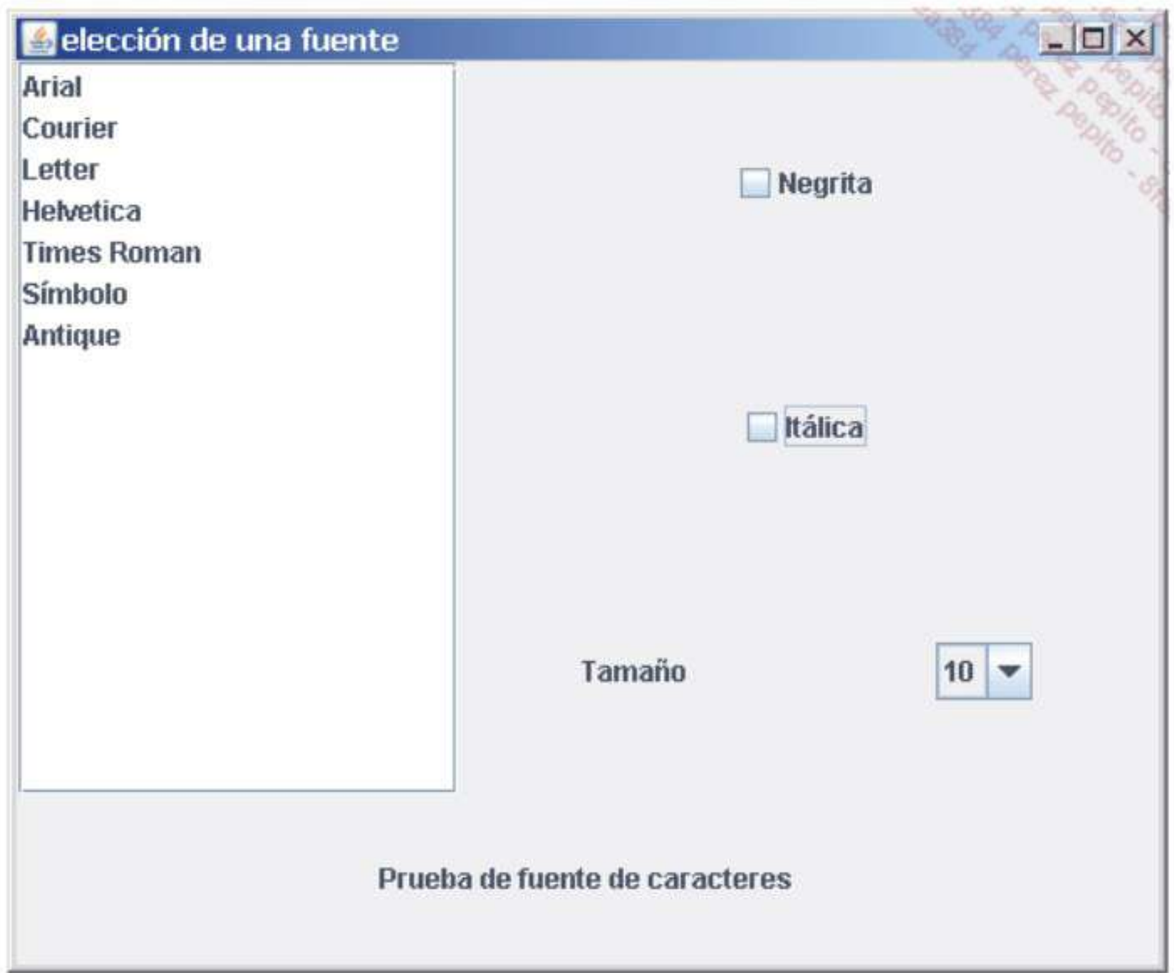
Para estas dos propiedades, la numeración empieza por cero.

- **gridwidth**: número de columnas ocupadas por el componente.
- **gridheight**: número de líneas 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 cómo. 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 cómo está posicionado el componente en el espacio disponible si no se redimensiona. Puede adoptar los siguientes valores constantes.

NORTHWEST	NORTH	NORTHEAST
WEST	CENTER	EAST
SOUTHWEST	SOUTH	SOUTHEAST

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;

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 cboTamaño;
      JList lstFuentes;
      JScrollPane desplaFuentes;

      public Pantalla()
      {
            setTitle("elección de una fuente");
            setBounds(0,0,300,100);
            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","Símbolo", "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 línea en 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 línea en 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 línea en 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 línea en 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 línea en altura
            gbc.gridheight=1;
            // ponderación en caso de ampliación del contenedor
            gbc.weightx=100;
            gbc.weighty=100;
            panel.add(lblEjemplo,gbc);
            getContentPane().add(panel);
      }
}
```

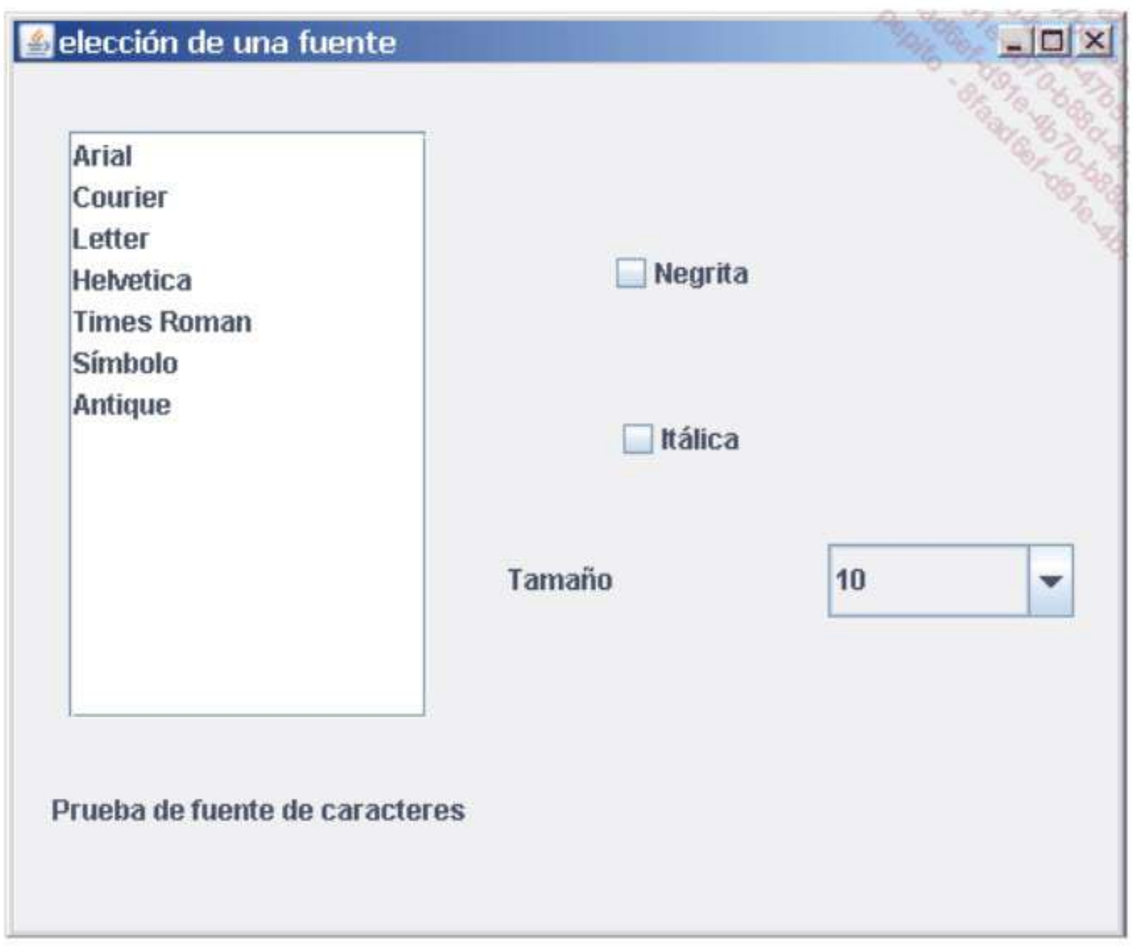
f. Sin renderizador

Si no le convence la utilidad de un renderizador, tiene la posibilidad de gestionar personalmente el tamaño y 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 éste. 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;

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 cboTamaño;
      JList lstFuentes;
      JScrollPane desplaFuentes;

public Pantalla()
{
setTitle("elección de una fuente");
setBounds(0,0,300,100);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// creación de los componentes
panel=new JPanel();
chkNegrita=new JCheckBox("Negrita");
chkCursiva=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",
"Símbolo", "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);
```

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.

5. 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.

javax.swing

Class JComponent

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ javax.swing.JComponent
```

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`. Éste 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 el 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 el 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 rendizadores 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 sólo tienen realmente interés cuando se asume completamente la presentación de los componentes en el interior del contenedor 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, el estilo de la fuente 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. 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 (salvo la primera letra) la etiqueta del botón.

```
   // creación de tres botones
   JButton b1,b2,b3;
   b1=new JButton("<html><font color=red>R<i>ojo</i></font>
</html>");
   b2=new JButton("<html><font color=green>V<i>erde</i></font>
</html>");
   b3=new JButton("<html><font color=blue>A<i>zul</i></font>
</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 al método `setVisible` o desactivarlo mediante el método `setEnabled`. 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);
chkCursiva.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);
     }

});
```

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 al 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` al especificar 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.

```
leyenda=new JLabel("nombre");
leyenda.setBorder(BorderFactory.createEtchedBorder());
leyenda.setPreferredSize(new Dimension(200,50));
leyenda.setHorizontalAlignment(SwingConstants.LEFT);
leyenda.setVerticalAlignment(SwingConstants.TOP);
```

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`.

```
JLabel leyenda;
leyenda=new JLabel("nombre");
leyenda.setBorder(BorderFactory.createEtchedBorder());
leyenda.setPreferredSize(new Dimension(200,200));
leyenda.setIcon(new ImageIcon("duke.gif"));
leyenda.setHorizontalTextPosition(SwingConstants.LEFT);
leyenda.setVerticalTextPosition(SwingConstants.BOTTOM);
leyenda.setIconTextGap(40);
```

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`.

```
leyenda.setDisplayedMnemonic('n');
leyenda.setLabelFor(lstFuentes);
```

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;


import java.util.GregorianCalendar;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JProgressBar;

public class Pantalla13 extends JFrame

{
     JPanel panel;
     JProgressBar pgbHora,pgbMinutos,pgbSegundo,pgbDespla;

     public Pantalla13()
     {
         setTitle("reloj");
         setBounds(0,0,300,100);
         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();
}
}
```

c. Los componentes de edición de texto

JTextField

Se utiliza el componente `JTextField` para permitir al usuario la introducción de datos. Este componente sólo 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, este componente 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`.

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.

```
txt=new JTextField(10);
txt.addFocusListener(new FocusListener()
{
     public void focusGained(FocusEvent e)
     {
          txt.selectAll();
     }
     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 a 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 como en el ejemplo siguiente:

```
txt=new JTextField(10);
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)
     {
          txt.copy();
     }
}
);
mnuCortar.addActionListener(new ActionListener()
{
     public void actionPerformed(ActionEvent e)
     {
```

```
            txt.cut();
        }
}
);
mnuPegar.addActionListener(new ActionListener()
{
        public void actionPerformed(ActionEvent e)
        {
            txt.paste();
        }
}
);
mnu.add(mnuCopiar);
mnu.add(mnuCortar);
mnu.add(mnuPegar);
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 sólo 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**. Sólo 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.getKeyChar()=='z' & e.isControlDown())
               {
                    udm.undo();
               }
          }
    });
```

JPasswordField

Este componente es básicamente 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 substitución durante la visualización, y el método `getPassword`, que permite obtener la contraseña introducida por el usuario.

JTextArea

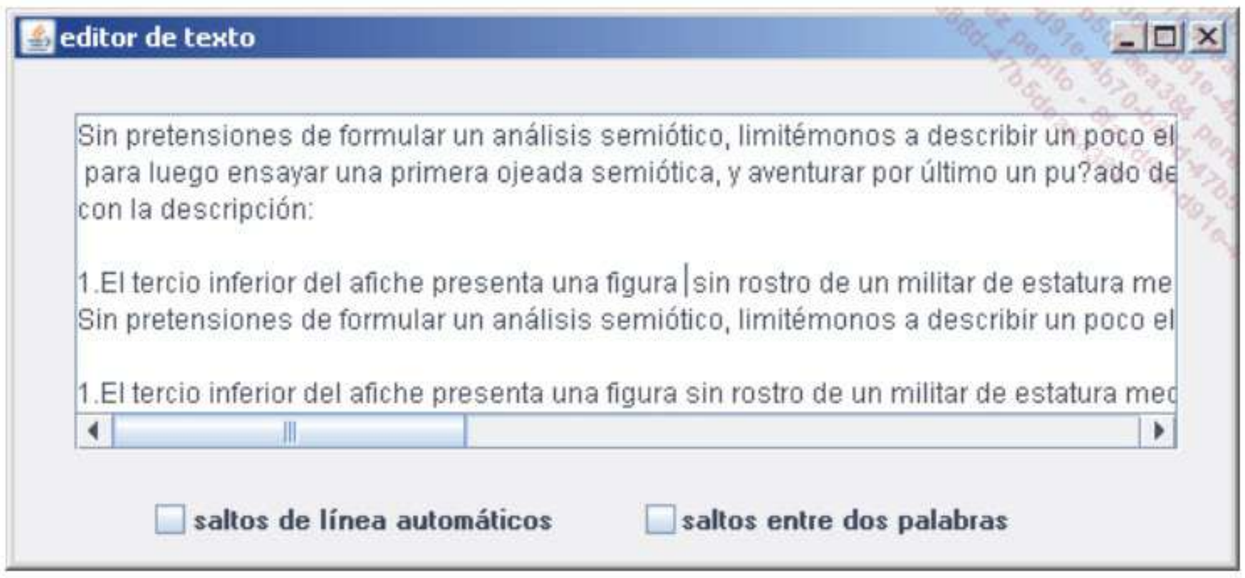
Este componente ofrece funcionalidades más avanzadas que un mero `JTextField`. La primera mejora importante aportada por este componente reside en su capacidad de gestionar la introducción de datos en varias líneas. Por este motivo, en el momento de la construcción de un objeto `JTextArea` debemos especificar el número de líneas 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 línea 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 línea con el riesgo de ver que algunas palabras se muestran en dos líneas. 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.

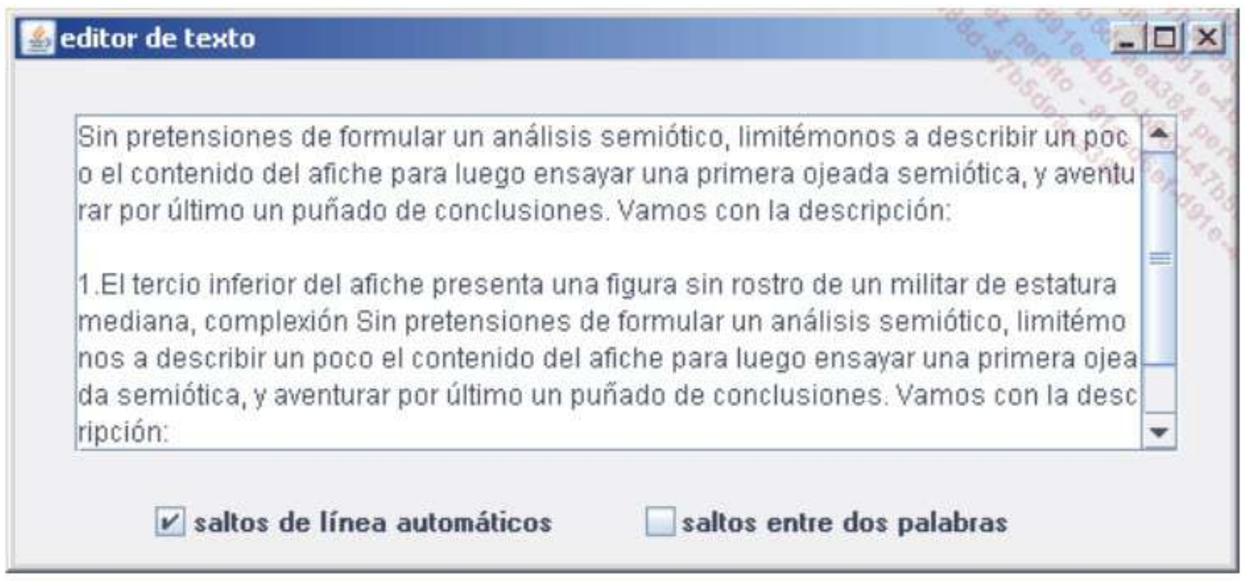
```
package es.eni;

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;

public class Pantalla extends JFrame
{
     JPanel panel;
     JTextArea txt;
     JCheckBox chkWrap,chkWrapWord;
     JScrollPane defil;
```

```
    public Pantalla()
    {
        setTitle("editor de texto");
        setBounds(0,0,300,100);
        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 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);
    }
}
```

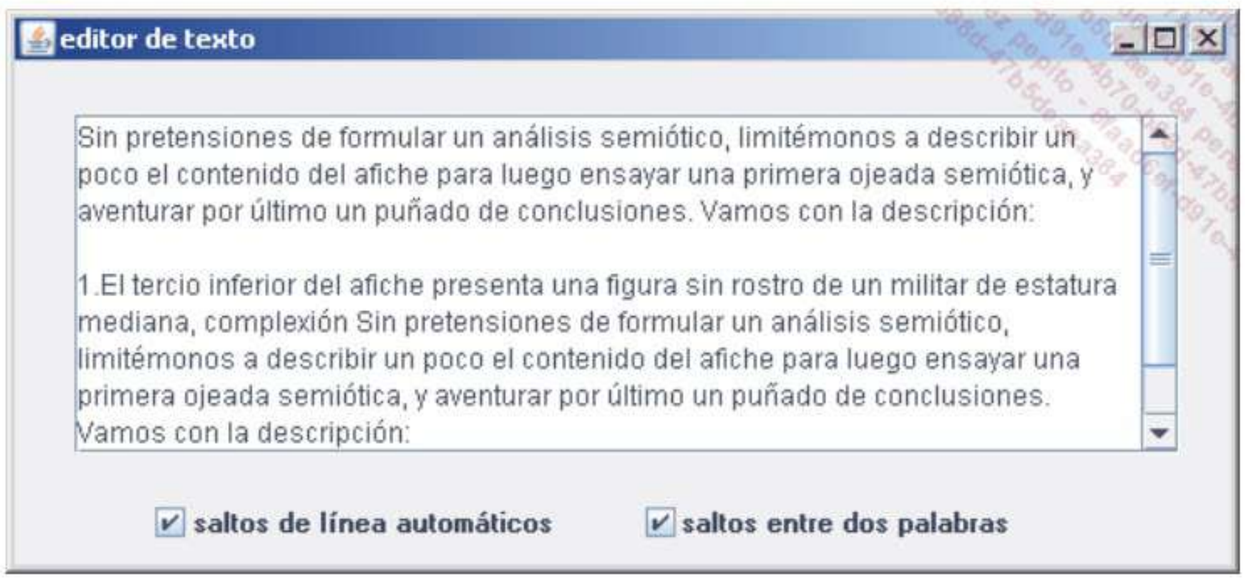
Con las opciones por defecto las líneas demasiado largas no son visibles en su totalidad y la barra de desplazamiento horizontal se activa automáticamente.



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, ahora se necesita una barra de desplazamiento vertical porque el número de líneas es superior a la capacidad del componente.



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. Al final el `JMenuBar` obtenido se ubica en su contenedor que no es otro que `JFrame` de la aplicación. Naturalmente también hace falta pensar en tratar los eventos producidos por estos elementos. En general, sólo 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;

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.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.PrintWriter;

import javax.swing.JCheckBox;
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 Pantalla14 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 Pantalla14()
     {
          setTitle("editor de texto");
          setBounds(0,0,300,100);
          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()
        {
```

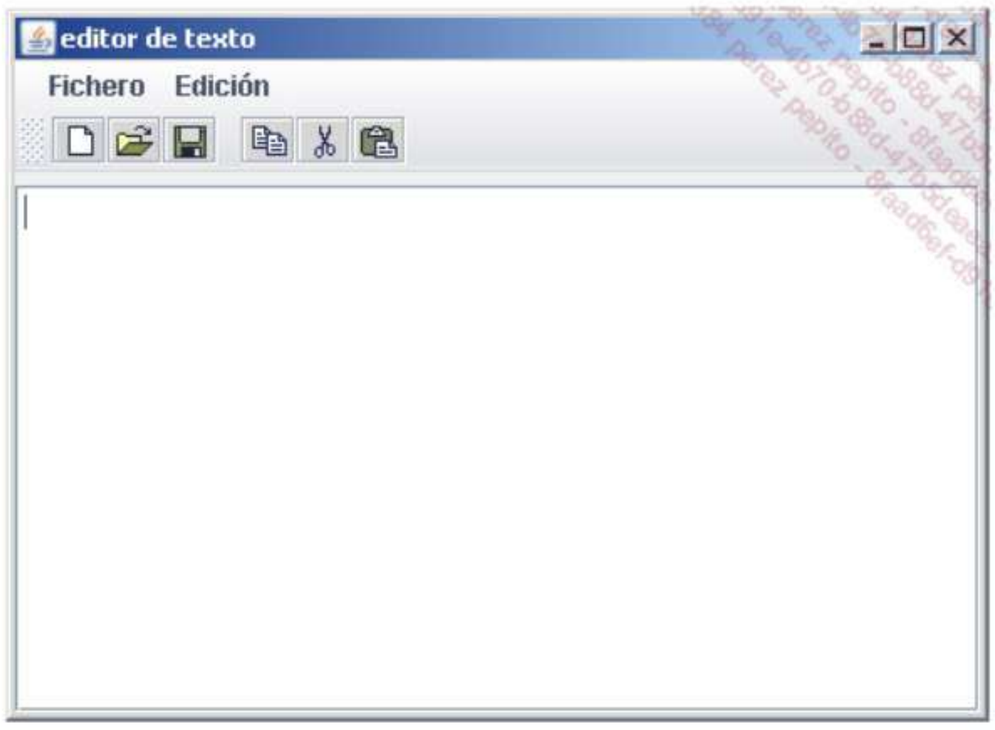
```
            public void actionPerformed(ActionEvent e)
            {
                 txt.paste();
            }
        });
        mnuGuardar.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent arg0)
            {
                try
                 {
                        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();
                }
            }
        });
        }
}
```

JToolBar

Este componente sirve de hecho de contenedor para los elementos que constituyen una barra de herramientas. Se añaden estos 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 ubicado en esta zona.

Se puede desactivar esta funcionalidad invocando al método `setFloatable(false)`. La barra de herramientas se queda entonces en su posición de origen. El código siguiente añade una barra de herramientas a nuestro editor de texto.

```
JToolBar tlbr;
tlbr=new JToolBar();
JButton btnNuevo,btnAbrir,btnGuardar;
JButton btnCopiar,btnCortar,btnPegar;
// creación de los botones
btnNuevo=new JButton(new ImageIcon("new.jpg"));
btnAbrir=new JButton(new ImageIcon("open.jpg"));
btnGuardar=new JButton(new ImageIcon("save.jpg"));
btnCopiar=new JButton(new ImageIcon("copy.jpg"));
btnPegar=new JButton(new ImageIcon("paste.jpg"));
btnCortar=new JButton(new ImageIcon("cut.jpg"));
// 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 a marcar 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 a marcar. 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 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");
     chkCursiva=new JCheckBox("Cursiva");
     opciones.add(chkNegrita);
     opciones.add(chkCursiva);
     chkNegrita.addActionListener(new ActionListener()
     {
          public void actionPerformed(ActionEvent e)
          {
               changeFuente();
          }
     });
     chkCursiva.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 (chkCursiva.isSelected())
          {
               atributos=atributos+Font.ITALIC;
          }

          Font fuente;
          fuente=new
Font(txt.getFont().getName(),atributos,txt.getFont().getSize());
          txt.setFont(fuente);
     }
```

JRadioButton

El componente `JRadioButton` permite también proponer al usuario diferentes opciones entre las cuales sólo 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 lógicamente. La clase `ButtonGroup` proporciona esta funcionalidad. Funciona de modo que entre todos los `JRadioButton` que se le han confiado, mediante su método `add`, sólo uno podrá estar seleccionado. Para agrupar físicamente unos `JRadioButton`, éstos 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 al añadir dos conjuntos de botones que permiten elegir el color del texto y el color de fondo de nuestro editor.

```
     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);
     // 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);
     // se agrega un borde con título
     color.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"color fuente"));
     colorFond.setBorder(BorderFactory.createTitledBorder
(BorderFactory.createEtchedBorder(),"color 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);
          }
     });
```

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 sólo 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:

- `ListSelectionModel.SINGLE_SELECTION`: sólo se puede seleccionar un único elemento de la lista a la vez.
- `ListSelectionModel.SINGLE_INTERVAL_SELECTION`: se pueden seleccionar varios elementos pero deben estar seguidos en la lista.
- `ListSelectionModel.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 `setSelectedIndex`. 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 `isSelectionEmpty`. 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.

```
JList fuentes;
JScrollPane desplaFuentes;

String[] nombresFuentes={"Dialog","DialogInput","Monospaced","Serif",
"SansSerif"};
fuentes=new JList(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 (chkCursiva.isSelected())
     {
          atributos=+Font.ITALIC;
     }
     Font fuente;
     fuente=new
Font(fuentes.getSelectedValue().toString(),atributos,
txt.getFont().getSize());
     txt.setFont(fuente);
}
```

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 ésta 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 sólo 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 nuestro editor de texto proponiendo al usuario la selección de un tamaño de fuente de caracteres.

```
JComboBox cboTamaño;
String 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 (chkCursiva.isSelected())
         {
              atributos=atributos+Font.ITALIC;
         }

         Font fuente;
         System.out.println(cboTamaño.getSelectedItem().toString());
         fuente=new Font(fuentes.getSelectedValue().toString(),
atributos,Integer.parseInt(cboTamano.getSelectedItem().
toString()));
         txt.setFont(fuente);
}
```

6. 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, éste 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, tenemos a nuestra disposición varios cuadros de diálogo predefinidos.

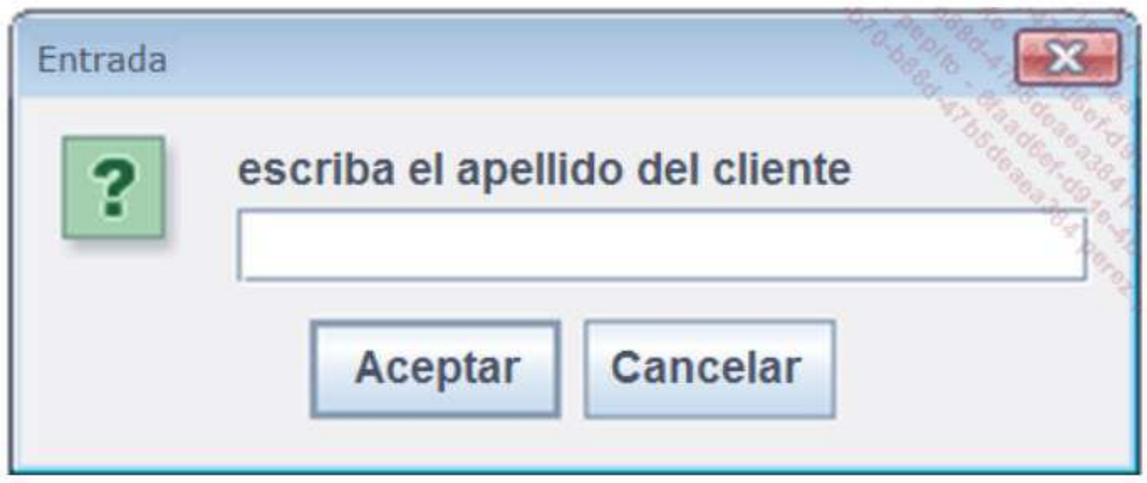
Están disponibles mediante métodos `static` en la clase `JOptionPane`.

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:

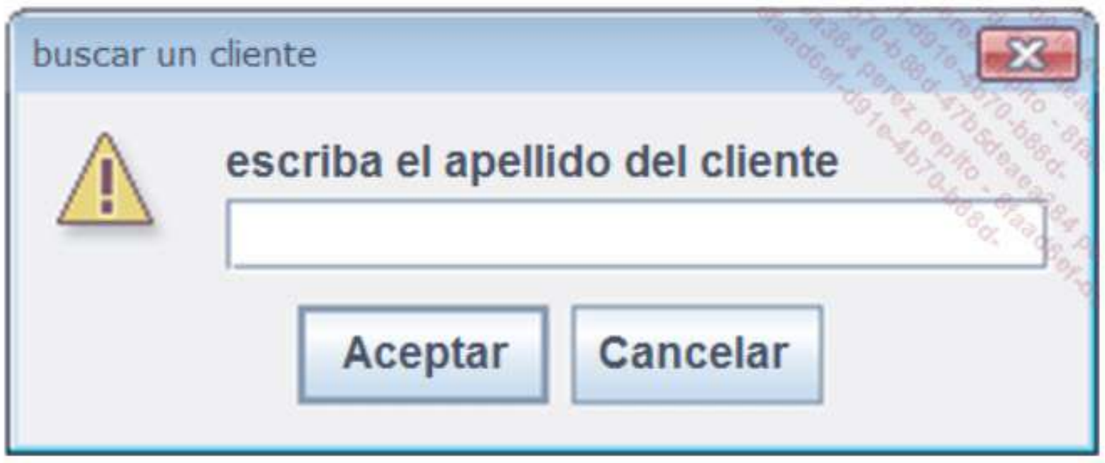
- el modo de visualización: ventana autónoma o ventana interna de un componente. En este caso, se le pasa como primer parámetro el componente afectado, o el valor `null` si la ventana es autónoma.
- el mensaje que se muestra en el cuadro de diálogo. Es el segundo parámetro que se utiliza para este propósito.
- 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	

Constante	Icono
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",
"buscar un cliente",JOptionPane.WARNING_MESSAGE);
```



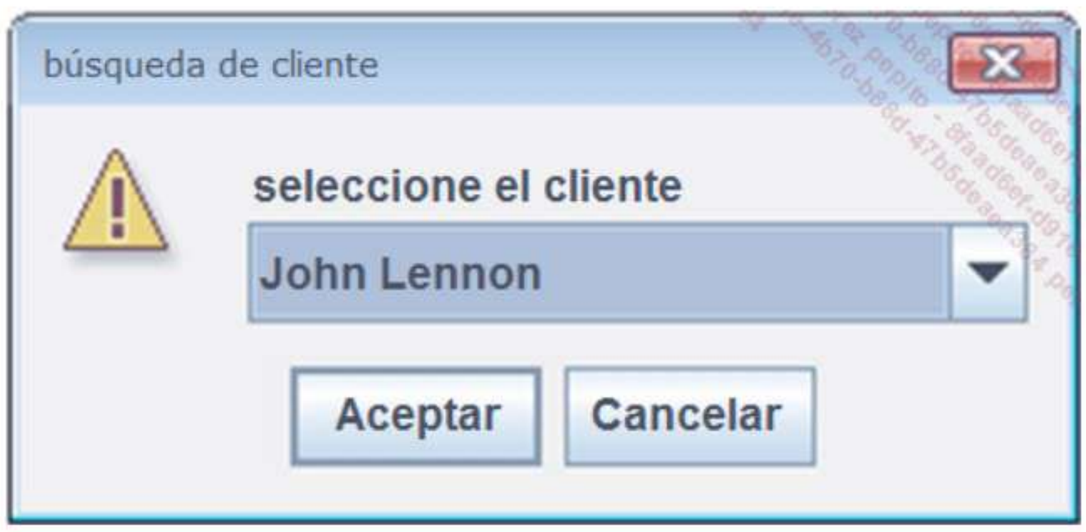
Existe otra versión que tiene un funcionamiento ligeramente distinto. No proporciona una zona de libre introducción de texto, sino una lista de objetos entre los que el usuario debe realizar su elección. La lista representa los objetos invocando al método `toString` de cada uno de ellos. Recibe como 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 **Anular** 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. El cuadro de mensaje está disponible mediante la función `showMessageDialog`.

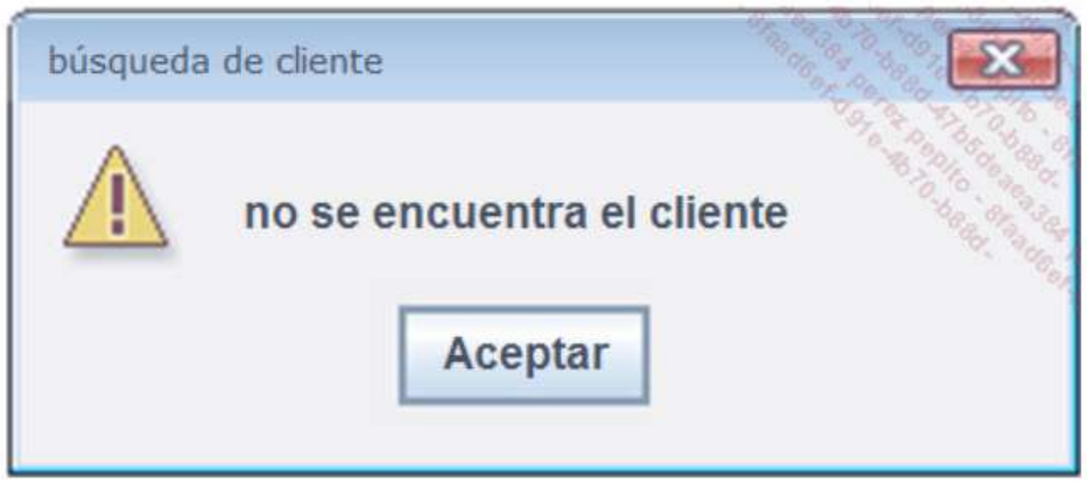
Esta función recibe los siguientes parámetros:

- 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

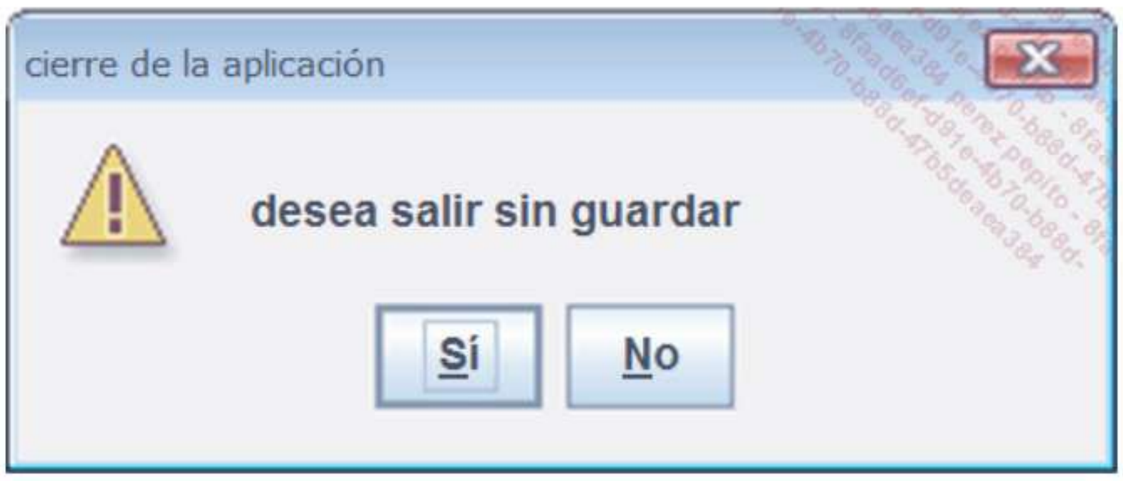
Este cuadro de diálogo se utiliza para realizar una pregunta al usuario y permitirle responder mediante uno de los botones que proporciona. Los botones disponibles están predefinidos por el cuadro de confirmación. La función `showConfirmDialog` provoca la apertura del cuadro de confirmación. Devuelve 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 a 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);
```



Principio de funcionamiento

Un applet es un tipo de aplicación Java específica que permite ejecutar código Java en el interior de una página Web. El principal objetivo de los applets es añadir interactividad y dinamismo a una página web. La llamada del applet se incorpora al código html de la página. Cuando el navegador analiza la página html que el servidor web acaba de transmitirle y encuentra una etiqueta que corresponde a un applet, descarga el código del applet y lanza su ejecución. La ventaja principal de un applet en comparación con una aplicación es sin duda la ausencia de instalación necesaria en los puestos clientes. Otras ventajas de la utilización de un applet son:

- Los clientes siempre dispondrán de la última versión del código. Cada vez que el navegador descarga la página web, el código del applet se descarga también desde el servidor web. Si una nueva versión del código está disponible, basta con desplegarla en el servidor web para que todos los clientes se puedan beneficiar de ello.
- Todos los recursos útiles para el buen funcionamiento de la aplicación estarán disponibles. Como el código del applet, los recursos que necesita se transmiten también del servidor web al navegador. Por lo tanto, sólo deben estar presentes en el servidor web.
- No hay riesgo de que la ejecución del código del applet provoque daños en el puesto cliente. La máquina virtual Java del navegador se encarga de la ejecución del código y, dado que lo considera potencialmente peligroso, lo ejecuta con un conjunto limitado de permisos.

Sin embargo los applets presentan una pequeña desventaja ya que la máquina virtual Java del jdk no los puede ejecutar de manera autónoma. Los applets deben incluirse, necesariamente, en una página html para que la máquina virtual Java del navegador se encargue de ellos. Naturalmente, este navegador debe disponer de una máquina virtual.

Creación de un applet

Para que un navegador pueda encargarse de un applet, éste último debe tener unas características determinadas. La solución utilizada para asegurarse de que el applet dispone de estas características consiste en crear una clase que herede de una superclase que disponga ya de dichas características.

La personalización se lleva a cabo al sustituir en la clase así creada algunos de los métodos heredados de la superclase. Las clases `Applet` y `JApplet`pueden utilizarse como clases básicas para la creación de un applet. La clase `JApplet` definida en el paquete `javax.swing` permite utilizar componentes de este mismo paquete para la construcción de la interfaz de usuario del applet. Para los applets que aseguran ellos mismos la gestión de su aspecto gráfico o que utilizan los componentes de la biblioteca `awt`, se debe utilizar la clase `Applet`. La clase `JApplet` hereda de la clase `Applet`, y por lo tanto extiende estas funcionalidades. La clase `Applet` forma parte de una importante jerarquía.

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Panel
              └ java.applet.Applet
```

Dada esta jerarquía, un applet es por lo tanto un objeto gráfico del cual es el navegador el encargado de mostrarlo.

1. Ciclo de vida de un applet

Cuando el navegador se encarga de un applet, ejecuta ciertos métodos en función de las circunstancias. Se pueden clasificar estos métodos en dos categorías.

- métodos relacionados con el ciclo de vida del applet.
- métodos de gestión del aspecto gráfico del applet.

Para diseñar applets eficaces, se debe entender muy bien cuándo se invoca a estos métodos y qué se puede esperar de ellos. La implementación de algunos de estos métodos en la clase `Applet` está vacía. Por lo tanto es indispensable sustituirlos en nuestra clase derivada.

a. Métodos relacionados con el ciclo de vida del applet

```
public void init()
```

Este método se ejecuta desde el final de la carga o de la nueva carga del applet a partir del servidor Web. De hecho es el primer método del applet ejecutado por el navegador. Permite inicializar el contexto en el cual va a funcionar el applet. Efectivamente, dentro de este método podemos realizar las siguientes tareas:

- creación de las instancias de las demás clases útiles para el funcionamiento del applet;
- inicialización de las variables;
- creación de los hilos;
- carga de las imágenes utilizadas por el applet;
- recuperación de los parámetros pasados desde la página html.

En una aplicación clásica, se suelen ejecutar estas operaciones en el constructor de la clase.

```
public void start()
```

El navegador llama este método después de la fase de inicialización del applet realizada por el método `init`. Se le llama también cada vez que el navegador muestra de nuevo la página en la que está insertado. Se ejecuta, por ejemplo, si el usuario cambia de página y vuelve a continuación a la página anterior (la que contiene el applet). También, algunos navegadores vuelven a llamar al método `init`. Corresponde a la etapa de lanzamiento o de relanzamiento del applet. Encontramos en ella por ejemplo el código que permite lanzar o volver a lanzar los hilos creados en el método `init`.

```
public void stop()
```

Este método se utiliza durante la fase de detención del applet. Esta fase interviene principalmente cuando el usuario cambia de página. También se puede llevar a cabo cuando la ejecución del applet se termina normalmente. En este caso, es el propio applet el que debe llamar a este método. El cierre del navegador provoca también la ejecución de este método justo antes de la llamada al método `destroy`.

```
public void destroy()
```

Este método es el último del ciclo de vida de un applet. El navegador lo llama justo antes de su cierre. Tiene el mismo papel que el destructor de una clase (método `finalize`). Se utiliza para eliminar de la memoria los objetos que hayan sido creados durante el funcionamiento del applet y esencialmente los que hayan sido creados durante el curso de la ejecución del método `init`. La sobrecarga de este método no es obligatoria porque de todas maneras el recolector de basura interviene para liberar los recursos de memoria utilizados por el applet. Por otra parte, no está garantizado que este método se pueda ejecutar completamente antes de que el navegador pare la máquina virtual Java.

b. Métodos de gestión del aspecto gráfico del applet

Hay que contemplar dos supuestos:

Si utilizamos dos componentes gráficos, tales como los disponibles en las bibliotecas `awt` y `swing`, lo reflejado por la interfaz del applet se asegura automáticamente. Esta posibilidad procede de las clases presentes en la jerarquía de la clase `Applet` o `JApplet`. En este caso es la clase `container`, de la cual heredan, entre otros, los applets, quien efectúa este trabajo con el método `paintComponents`.

Si asumimos completamente la representación gráfica del applet, debemos sobrecargar el método `paint` para que sea capaz de gestionar la visualización del applet cada vez sea necesario dibujarlo en la página web.

Para visualizar el orden de ejecución de estos diferentes métodos, vamos a escribir nuestro primer applet. Debemos crear una clase que herede de la clase `Applet`. En esta clase declaramos una variable de tipo `String` para memorizar los pasos en los diferentes métodos. En el método `paint`, mostramos la cadena de caracteres en el contexto gráfico del applet.

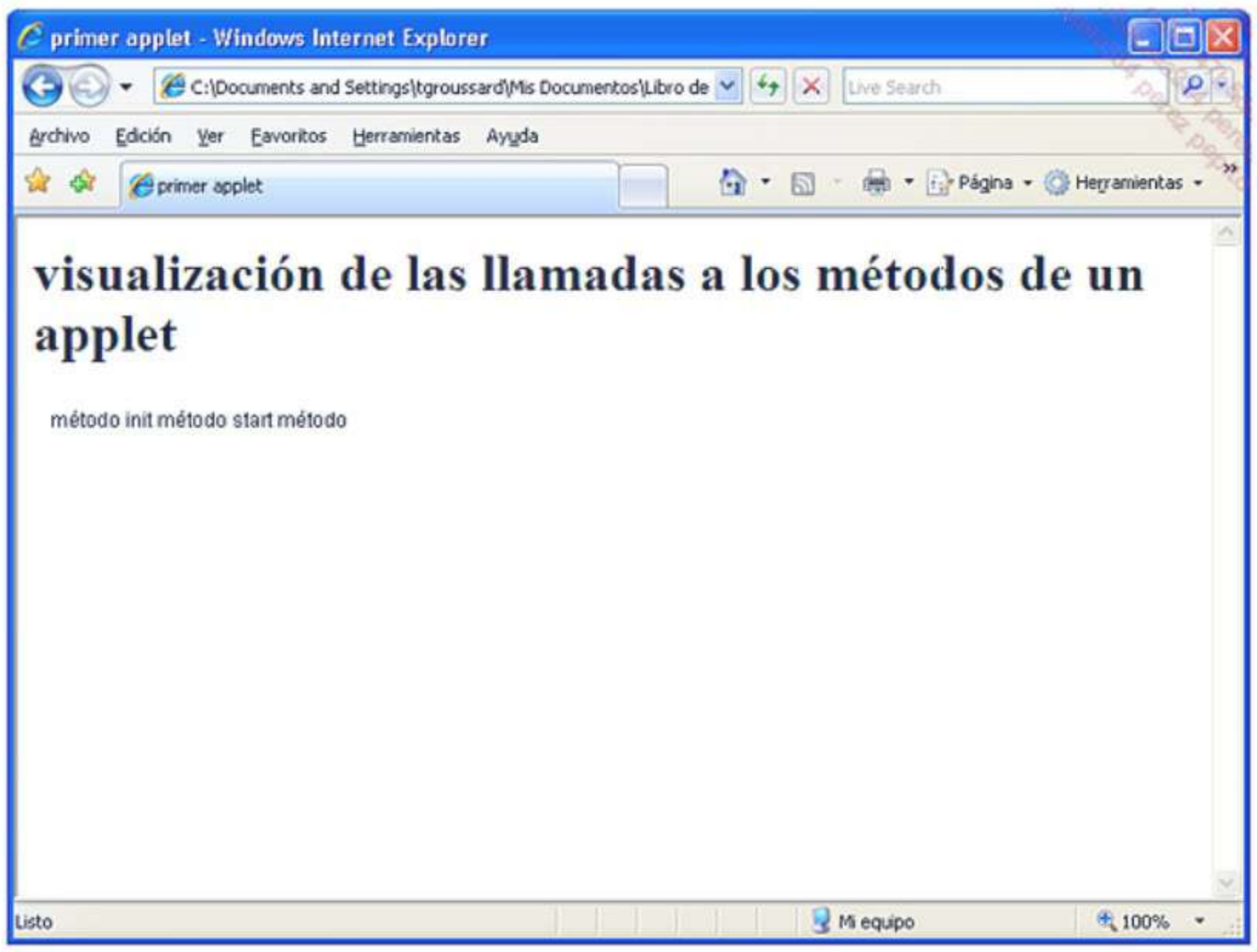
```
import java.applet.Applet;
import java.awt.Graphics;


public class TestApplet extends Applet
{
     private String mensaje="";
     public void destroy()
     {
          mensaje=mensaje + "método destroy \r\n";
     }

     public void init()
     {
          mensaje=mensaje + "método init \r\n";
     }

     public void start()
     {
          mensaje=mensaje + "método start \r\n";
     }

     public void stop()
     {
          mensaje=mensaje + "método stop \r\n";
     }

     public void paint(Graphics g)
     {
          mensaje=mensaje + "método paint \r\n";
          g.drawString(mensaje, 10, 20);
     }

}
```

Para comprobar el buen funcionamiento del applet, debemos insertarlo en una página html y visualizar esta última en un navegador o con la herramienta **appletViewer** del jdk.

```
<html>
<head>
<title> primer applet </title>
</head>
<body>
<h1>visualización de las llamadas a los métodos de un applet</h1>
<applet code="TestApplet"></applet>
</body>
</html>
```

La etiqueta `<applet>` indica al navegador que debe, durante el análisis de la página html, cargar la clase que corresponde al applet cuyo nombre está especificado por el atributo `code` de la etiqueta. Esta etiqueta dispone de otras posibilidades que se detallarán más adelante en este capítulo.

Se presenta a continuación la visualización de la página html en un navegador.



Se confirma que se han ejecutado en primer lugar el método `init` y, a continuación, el método `start`. También parece que al menos se ha ejecutado otro método, pero no podemos visualizar el texto mostrado en su totalidad. Este problema está relacionado con el hecho de que el navegador reserva un espacio para el applet en el documento. El espacio necesario para la visualización de la cadena es superior a la anchura del applet y, por ello, obtenemos esta visualización truncada. Veremos que podemos actuar sobre las dimensiones del espacio atribuido al applet en la

página html con los atributos `height` y `width` de la etiqueta, pero esta solución sólo aplazará el problema porque forzosamente tendremos en algún momento una cadena de caracteres que necesite una anchura superior que la disponible para el applet. Tenemos que tener más cuidado, durante el diseño del applet, en cuanto a su representación gráfica.

2. Construir la interfaz usuario de un applet

Para diseñar la interfaz usuario del applet tenemos dos soluciones. Podemos utilizar una biblioteca de componentes gráficos (`awt` o `swing`) y utilizar sus componentes para definir el aspecto visual del applet. Esta solución presenta la ventaja de ser rápida a la hora de ponerla en marcha y de ser conveniente para la mayoría de los casos. El único reproche que se le puede hacer es justamente el hecho de utilizar componentes clásicos para la visualización y de no permitir presentaciones específicas. Sin embargo, la mejora y la evolución de las bibliotecas gráficas proporciona cada vez más posibilidades. Si, sin embargo, no encuentra el componente adaptado a sus necesidades, podrá asumir completamente la representación gráfica del applet sobrecargando su método `paint`. En una primera fase, vamos a utilizar esta solución intentando eliminar los problemas encontrados en nuestro ejemplo anterior. Tenemos que tener cuidado para no dibujar más allá de los limites del applet. Para ello, vamos a realizar el proceso siguiente:

- Crear una fuente de caracteres para la visualización,
- Obtener las dimensiones de la zona reservada al applet,
- Dibujar los caracteres uno tras otro verificando cada vez que el espacio disponible es suficiente,
- Si el espacio no es suficiente en la línea actual, debemos gestionar el cambio de línea.

Vamos a intentar resolver los problemas por etapas.

a. Creación de una fuente de caracteres

Una fuente de caracteres se define por su nombre y su tamaño. Existe una multitud de fuentes de caracteres y nada nos garantiza que la fuente que vamos a elegir esté disponible en el ordenador en el cual se va ejecutar nuestro applet. Para resolver este problema, Java permite trabajar con fuentes lógicas que se convierten en tiempo de ejecución, en fuentes disponibles en el puesto cliente. Las siguientes fuentes lógicas están disponibles:

- SansSerif
- Serif
- Monospaced
- Dialog
- DialogInput

La fuente real utilizada se determinará en el momento de la ejecución en función de las disponibilidades. Por ejemplo, la fuente lógica SansSerif se convertirá en fuente Arial en un sistema Windows.

La creación de la fuente se realiza invocando al constructor de la clase `Font` y pasándole como parámetros el nombre de la fuente, el estilo de la fuente (negrita, cursiva...) y el tamaño de la fuente.

```
Font fuente=null;
fuente =new Font("SansSerif",Font.PLAIN,14);
```

b. Obtener las dimensiones del applet

Las dimensiones del applet pueden estar fijadas por el navegador o por el diseñador de la página html cuando éste inserta el applet en la página. Se pueden obtener estas dimensiones en el momento de la ejecución al utilizar los métodos `getHeight` y `getWidth`.

```
int anchura;
int altura;
anchura=getWidth();
altura=getHeight();
```

c. Dibujar los caracteres

El método `drawString` de la clase `Graphics` permite la visualización de una cadena de caracteres con la fuente determinada previamente por el método `setFont`. El método `drawString` recibe los siguientes parámetros:

- la cadena de caracteres;
- la posición horizontal donde se efectúa la visualización;
- la posición vertical donde se efectúa la visualización.

Este método no tiene noción de cursor por lo que debemos indicar, cada vez que lo invoquemos, una posición correcta para la visualización.

```
g.drawString(texto,posicionX,posicionY);
```

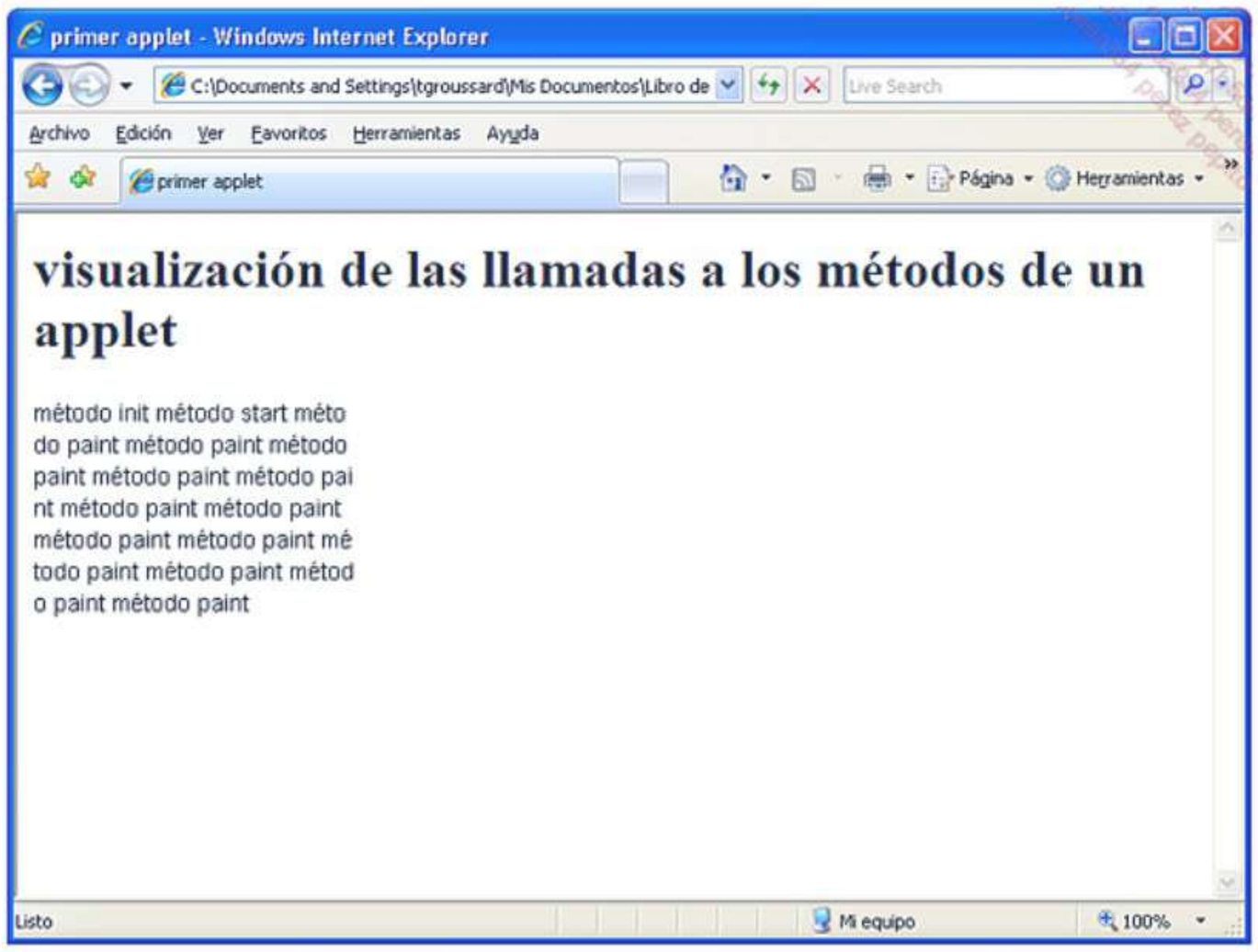
d. Determinar las dimensiones de una cadena

Para poder gestionar correctamente la visualización, debemos comprobar si queda bastante espacio disponible para la cadena que se quiere mostrar. Por supuesto este espacio depende del número de caracteres de la cadena y también de la fuente utilizada. Además, algunas fuentes al tener un espaciado proporcional, necesitan más espacio para mostrar el carácter W que para mostrar el carácter i. No es posible, por lo tanto, basarse en una anchura constante para cada carácter. La clase `FontMetrics` nos aporta una ayuda notable para resolver este problema. Un objeto de `FontMetrics` se puede obtener al utilizar el método `getFontMetrics` del contexto gráfico del applet. Este método recibe como parámetro la fuente de caracteres con la que va a efectuar sus cálculos. La anchura necesaria para mostrar una cadena de caracteres determinada se obtiene, a continuación, invocando al método `stringWidth` al cual se le debe proporcionar la cadena de caracteres a "medir". La altura puede obtenerse mediante el método `getHeight`. La altura, al estar relacionada únicamente con la fuente de caracteres, no exige proporcionar la cadena de caracteres para obtener esta información.

Ahora que tenemos todos los elementos útiles, presentamos a continuación la manera de utilizarlos para resolver nuestro problema.

```
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import javax.swing.JApplet;


public class TestApplet2 extends JApplet
{
     private String mensaje="";
     public void destroy()
     {
          mensaje=mensaje + "método destroy \r\n";
     }

     public void init()
     {
          mensaje=mensaje + "método init \r\n";
     }

     public void start()
     {
          mensaje=mensaje + "método start \r\n";
     }

     public void stop()
     {
          mensaje=mensaje + "método stop \r\n";
     }

public void paint(Graphics g)
{
// creación de la fuente de caracteres
          Font =null;
          fuente =new Font("SansSerif",Font.PLAIN,14);
// asignación de la fuente al contexto gráfico
          g.setFont(fuente);
// determinación de la anchura y altura del applet
          int anchuraApplet;
          int alturaApplet;
          anchuraApplet=getWidth();
          alturaApplet=getHeight();
// creación de un objeto FontMetrics para obtener información
          // relativa al tamaño de los caracteres
          FontMetrics fm;
          fm=g.getFontMetrics(fuente);
          int alturaFuente;
          int anchuraCaracter;
          alturaFuente=fm.getHeight();
// declaración de las variables para gestionar la posición
// de la visualización
          int cursorX;
          int cursorY;
          cursorX=0;
          cursorY=alturaFuente;
          mensaje=mensaje + "método paint \r\n";
          // bucle de tratamiento de los caracteres uno por uno
          for (int i=0;i<mensaje.length();i++)
          {
     // recuperación del carácter a tratar y de su anchura
               String caracterActual;
               caracterActual=mensaje.substring(i,i+1);
               anchuraCaracter=fm.stringWidth(caracterActual);
          // verificación de si queda espacio en la línea
               if (cursorX+anchuraCaracter>anchuraApplet)
               {
                    // paso al principio de la línea siguiente
                    cursorY=cursorY+alturaFuente;
                    cursorX=0;
               }
          // visualización del carácter en la posición actual
               g.drawString(caracterActual,cursorX,cursorY);
          // actualización de la posición del cursor
               cursorX=cursorX+anchuraCaracter;
          }
     }
}
```

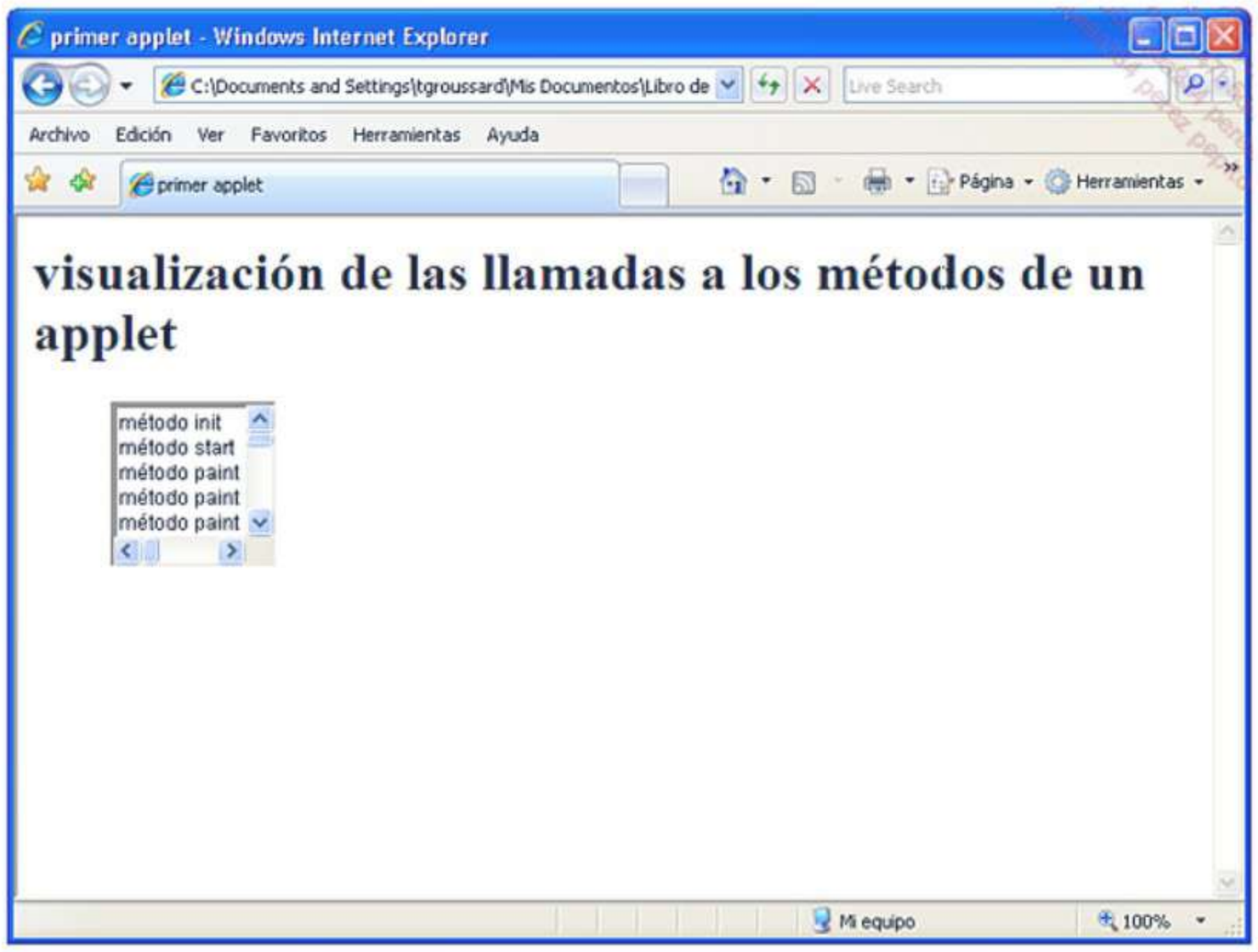
Al comprobar el funcionamiento del applet, constatamos que hay una clara mejora en comparación con la versión anterior.



Sigue habiendo mejoras que aportar, en particular en lo referente a la gestión del desplazamiento vertical pero vamos a probar otra solución y confiar el aspecto gráfico del applet a componentes especializados. La visualización con desplazamiento de texto es una funcionalidad corriente que debe ofrecer una aplicación y por supuesto los diseñadores de bibliotecas gráficas de Java han diseñado componentes adaptados a estas necesidades. Las clases `TextArea` y `ScrollPane` nos van a permitir resolver nuestro problema. Su puesta en marcha es muy sencilla porque basta con crear una instancia de la clase `TextArea` indicando en la llamada del constructor las dimensiones deseadas. A continuación, se delega esta instancia al componente `ScrollPane`, que se encargará del desplazamiento del texto. Luego el conjunto se añade al applet. Añadir texto es una tarea que se lleva a cabo de una forma muy sencilla invocando al método `append` de la clase `TextArea`. Por lo tanto, nuestra nueva versión de applet tiene la forma siguiente.

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


public class TestApplet3 extends Applet
{
     ScrollPane desplazamiento;
     TextArea txt;
     private String mensaje="";

     public void destroy()
     {
          mensaje=mensaje + "método destroy \r\n";
     }

     public void init()
     {
          txt=new TextArea(50,50);
          desplazamiento=new ScrollPane();
          desplazamiento.add(txt);
          add(desplazamiento);
          txt.append("método init \r\n");
     }

     public void start()
     {
          txt.append("método start \r\n");
     }

     public void stop()
     {
          txt.append("método stop \r\n");
     }

     public void paint(Graphics g)
     {
          txt.append("método paint \r\n");
     }
}
```

El código es mucho más sencillo que el de la versión anterior y sin embargo el resultado es mucho más profesional.



3. Las imágenes en los applets

La visualización de una imagen en un applet se efectúa en tres operaciones distintas:

- la carga de la imagen a partir de un archivo,
- el tratamiento eventual de la imagen,
- el trazado de la imagen en el contexto gráfico del applet.

Vamos a detallar cada una de estas operaciones.

a. Carga de una imagen

La clase `Applet` proporciona dos versiones del método `getImage` que permiten cargar una imagen a partir de un archivo gif o jpeg. La primera versión recibe como parámetro la URL completa del archivo que contiene la imagen. La segunda recibe como primer parámetro también una URL y como segundo parámetro un nombre relativo a esta URL que permita acceder al archivo que contiene la imagen. Se puede usar cualquiera de las dos soluciones siguientes.

```
private Imagen img;
img=getImage(new URL("http://www.eni-ecole.fr/imágenes/
escuela.jpg"));


private Imagen img;
img=getImage(new URL("http://www.eni-ecole.fr/"),"imágenes/
escuela.jpg");
```

Estas dos soluciones presentan el gran inconveniente de que utilizan rutas absolutas para las URL. Es muy probable que en el momento del diseño del applet no se sepa todavía en qué servidor y con qué nombre se va a desplegar la aplicación. Por lo tanto, es imposible utilizar directamente esta información en el código. En cambio, podemos estar seguros de que, si nuestro código se ejecuta, es que el navegador lo ha descargado. Por lo tanto podemos preguntarle al applet que nos indique la ubicación a partir de la cual el navegador la descargó. El método `getCodeBase` nos permite obtener la URL de la carpeta que contiene el applet en el servidor. Sin embargo no hay que olvidar que si el applet está presente en el navegador es porque éste descargó previamente la página html que lo contiene. La URL de esta página se obtiene con el método `getDocumentBase`.

Por lo tanto es posible informar el archivo que contiene la imagen a partir de una u otra de estas ubicaciones. La organización de la aplicación es la encargada de indicarnos el método adecuado.

- Si la imagen se encuentra en la carpeta, o en una subcarpeta, donde se ubica la página html debemos utilizar el método `getDocumentBase`.
- Si la imagen se encuentra en la carpeta, o en una subcarpeta, donde está ubicado el código del applet, debemos utilizar el método `getCodeBase`.

Con la organización siguiente:

Se puede obtener la imagen en el applet con el código siguiente:

```
img=getImage(getDocumentBase(),"imágenes/duke.gif");
```

Con esta otra organización:

La imagen está accesible con este código:

```
img=getImage(getCodeBase(),"imágenes/duke.gif");
```

b. Tratamiento de la imagen

Si se debe procesar la imagen antes de su visualización, podemos obtener información de ella gracias a los métodos de la clase `Image`.

Method Summary

void	flush() Flushes all reconstructable resources being used by this Image object.
float	getAccelerationPriority() Returns the current value of the acceleration priority hint.
ImageCapabilities	getCapabilities(GraphicsConfiguration gc) Returns an ImageCapabilities object which can be inquired as to the capabilities of this Image on the specified GraphicsConfiguration.
abstract Graphics	getGraphics() Creates a graphics context for drawing to an off-screen image.
abstract int	getHeight(ImageObserver observer) Determines the height of the image.
abstract Object	getProperty(String name, ImageObserver observer) Gets a property of this image by name.
Image	getScaledInstance(int width, int height, int hints) Creates a scaled version of this image.
abstract ImageProducer	getSource() Gets the object that produces the pixels for the image.
abstract int	getWidth(ImageObserver observer) Determines the width of the image.
void	setAccelerationPriority(float priority) Sets a hint for this image about how important acceleration is.

c. Trazado de la imagen

De hecho es el contexto gráfico del applet el que realmente va a encargarse de la visualización de la imagen sobre el applet. Para ello, la clase `Graphics` proporciona seis versiones diferentes del método `drawImage`.

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  ImageObserver observer)
```

Muestra la imagen en las coordenadas `x` e `y`.

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  Color bgcolor,
                                  ImageObserver observer)
```

Muestra la imagen en las coordenadas `x` e `y` y rellena las porciones transparentes de la imagen con el color especificado por `bgcolor`. El resultado es equivalente a una visualización de la imagen sobre un rectángulo de color `bgcolor`.

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  int width,
                                  int height,
                                  ImageObserver observer)
```

Muestra la imagen en las coordenadas `x` e `y` y vuelve a dimensionar la imagen con la anchura `width` y la altura `height`.

```
public abstract boolean drawImage(Image img,
                                  int x,
                                  int y,
                                  int width,
                                  int height,
                                  Color bgcolor,
                                  ImageObserver observer)
```

Muestra la imagen en las coordenadas `x` e `y`, vuelve a dimensionar la imagen con la anchura `width` y la altura `height` y rellena las porciones transparentes de la imagen con el color especificado por `bgcolor`.

```
public abstract boolean drawImage(Image img,
                                  int dx1,
                                  int dy1,
                                  int dx2,
                                  int dy2,
                                  int sx1,
                                  int sy1,
                                  int sx2,
                                  int sy2,
                                  ImageObserver observer)
```

Muestra la parte de la imagen delimitada por el rectángulo formado por (`sx1`, `sy1`) y (`sx2`, `sy2`) en el rectángulo formado por (`dx1`, `dy1`) y (`dx2`, `dy2`). La parte de la imagen a dibujar se vuelve a dimensionar automáticamente para rellenar de manera exacta el rectángulo de destino.

```
public abstract boolean drawImage(Image img,
                                  int dx1,
                                  int dy1,
                                  int dx2,
                                  int dy2,
                                  int sx1,
                                  int sy1,
                                  int sx2,
                                  int sy2,
                                  Color bgcolor,
                                  ImageObserver observer)
```

Muestra la parte de la imagen delimitada por el rectángulo formado por (`sx1`, `sy1`) y (`sx2`, `sy2`) en el rectángulo formado por (`dx1`, `dy1`) y (`dx2`, `dy2`). La parte de la imagen a dibujar se vuelve a dimensionar automáticamente para rellenar de manera exacta el rectángulo de destino y rellena las porciones transparentes de la imagen con el color especificado por `bgcolor`.

Para todos estos métodos, se debe proporcionar una instancia de clase que implementa la interfaz `ImageObserver`. Esta instancia de clase se utiliza para seguir la evolución de la carga de la imagen.

A continuación se presentan algunos ejemplos de código y la visualización correspondiente.

```
img=getImage(getCodeBase(),"imágenes/duke.gif");
g.drawImage(img,0,0,this);
```

Duke, la mascota

```
img=getImage(getCodeBase(),"imágenes/duke.gif");
g.drawImage(img,0,0,20,20,this);
g.drawImage(img,30,0,50,50,this);
g.drawImage(img,90,0,100,100,this);
```

La familia Duke

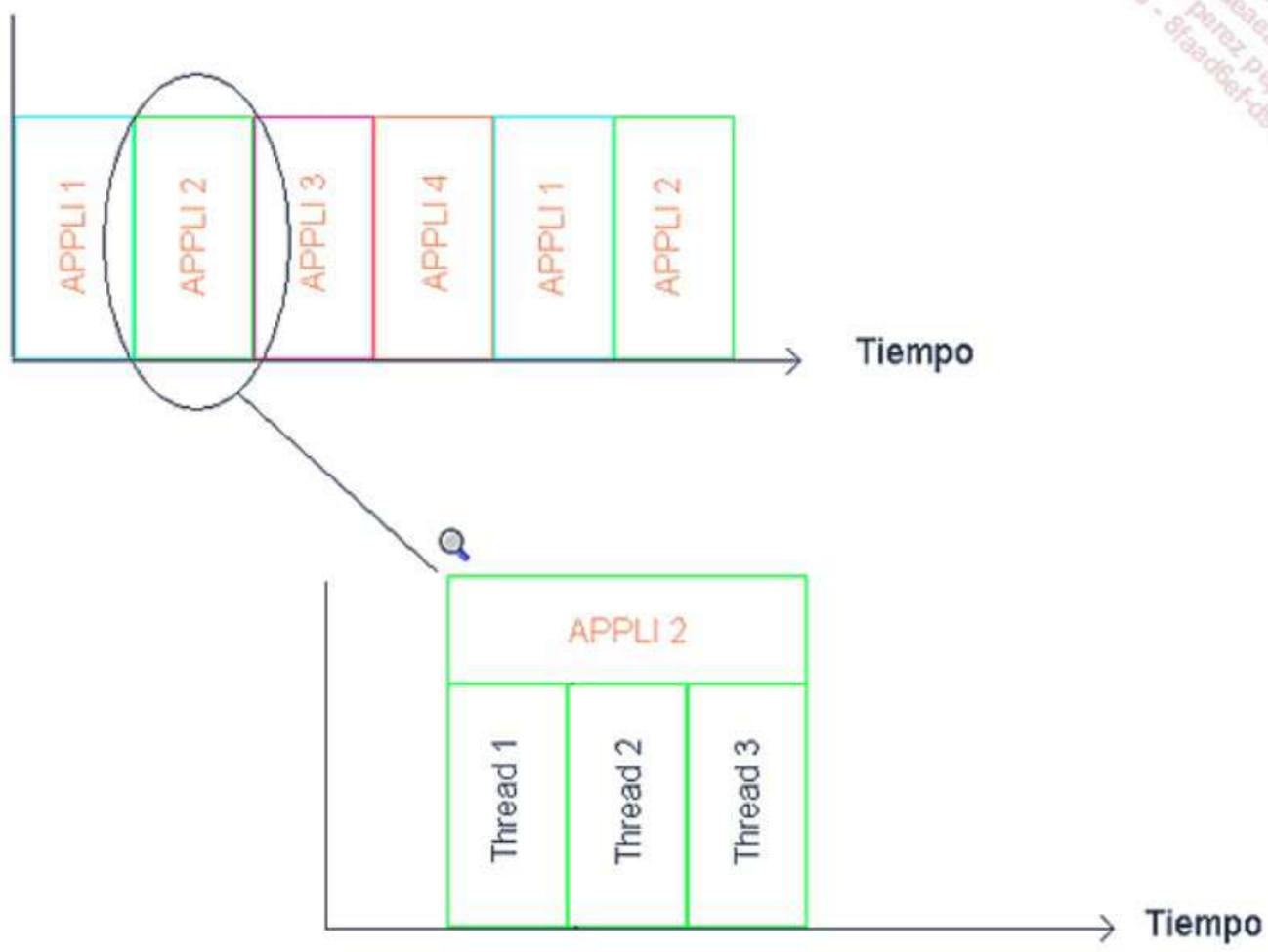
```
img=getImage(getCodeBase(),"imágenes/duke.gif");
g.drawImage(img,0,0,100,100,30,30,50,50,this);
```

Detalle de la nariz de Duke

4. Los hilos en los applets

Antes de estudiar cómo crear hilos, primero vamos a definir este término. La mayoría de las máquinas disponen de un único procesador para gestionar su funcionamiento. Por supuesto, este procesador sólo puede realizar una operación elemental a la vez, y sin embargo con la mayoría de los sistemas operativos podemos ejecutar varias aplicaciones de manera simultánea. De hecho, tenemos la sensación de que varias aplicaciones se ejecutan a la vez, pero en realidad no es así. Para simular esta simultaneidad, el sistema operativo "divide" el tiempo de procesador disponible en capas muy finas y las distribuye a las diferentes aplicaciones. La conmutación es tan rápida que tenemos la sensación de que todas las aplicaciones se ejecutan a la vez. La programación multithilo aplica este mismo principio a una aplicación.

Si tomamos el ejemplo de tratamiento de texto, éste puede efectuar por ejemplo varias operaciones a la vez (introducción de texto, verificación de la ortografía, configuración de la página, impresión…). Para poder efectuar todos estos tratamientos, la aplicación realiza de nuevo una división de cada capa de tiempo de procesador que le proporciona el sistema operativo y asigna una porción de esta capa para cada tarea que se debe realizar.



Por defecto, una aplicación o un applet está asociado a un hilo responsable de la ejecución de nuestro código. A veces se le llama hilo principal. En ocasiones, puede resultar útil añadir uno o varios hilos adicionales para conservar un buen nivel de dinamismo de la aplicación. En el caso de un applet, podemos añadir por ejemplo un hilo para efectuar una operación de inicialización relativamente larga. Si esta operación se efectúa con el método `init` del applet, el navegador tendrá que esperar a que finalice este método para proseguir el lanzamiento del applet. En cambio, si desde el método `init` lanzamos un nuevo hilo para efectuar la operación de inicialización, el resto del lanzamiento del applet se ejecutará más rápidamente mientras el hilo siga efectuando la inicialización. Esta técnica es muy útil para un applet que utiliza archivos de sonidos. Así se puede descargar el archivo mientras se prosigue la inicialización del applet.

Los hilos son también muy útiles cuando un applet debe realizar un tratamiento repetitivo. Podemos confiar a un nuevo hilo la tarea de ejecutar este tratamiento sin penalizar al funcionamiento del resto del applet.

La utilización de los hilos necesita tres etapas en el diseño de la aplicación:

- crear un nuevo hilo,
- definir el tratamiento que debe efectuar el nuevo hilo,
- gestionar los lanzamientos y paradas del hilo.

Mostramos, a continuación, el detalle de estas tres etapas.

a. Creación de un nuevo hilo

La clase `Thread` permite la creación y la gestión de un hilo. Como para cualquier otra clase, hay que crear una instancia mediante alguno de los constructores disponibles. Es importante señalar que la creación de una instancia de la clase `Thread` no activa el lanzamiento de la ejecución del hilo.

b. Definir el tratamiento a efectuar

Cuando se lanza un hilo, éste ejecuta automáticamente su método `run`. Por defecto este método se define en la clase `Thread` pero no contiene ningún código. Por lo tanto, es obligatorio volver a definir este método. Se puede hacer creando una clase que hereda de la clase `Thread` y definiendo de nuevo en ésta el método `run`.

```
public class ThreadPerso extends Thread
{

     public void run()
     {
          // código a ejecutar por el hilo
     }

}
```

Por supuesto, en este caso se tendrá que crear una instancia de esta clase.

También podemos obtener el mismo resultado al crear una instancia de clase interna anónima.

```
     Thread t;
     t=new Thread(){
          public void run()
          {
               // código a ejecutar por el hilo
          }
     };
```

La última solución consiste en indicar al hilo que el método `run` que debe ejecutar se encuentra en otra clase. En este caso, se debe proporcionar una instancia de esta clase en el momento de la llamada al constructor de la clase `Thread`. Para asegurarse de que la clase utilizada contiene en efecto un método `run`, ésta debe implementar la interfaz `Runnable`. La clase del applet se puede encargar de ello.

```
public class TestApplet5 extends Applet implements Runnable
{

     Thread th;
     public void run()
     {
          // código a ejecutar por el hilo
     }



     public void start()
     {
          th=new Thread(this);
     }

}
```

Falta ahora definir el contenido del método `run`. Existen dos posibilidades:

- El método `run` ejecuta un tratamiento único en este caso, se concibe como un método clásico y su ejecución terminará con la última instrucción de este método.
- El método `run` ejecuta un tratamiento cíclico y en este caso debe contener un bucle que termina con la desaparición del hilo que lo ejecuta. Se puede utilizar por ejemplo la sintaxis siguiente para este bucle.

```
public void run()
{
     Thread t;
     t= Thread.currentThread();

      while(th==t)
          {
               // código a ejecutar por el hilo
          }
}
```

Al permitir el método estático `currentThread` obtener una referencia en el hilo actual, el bucle terminará en cuanto la variable `th` deje de hacer referencia al hilo actual. Será el caso, por ejemplo, si se asigna el valor `null` a esta variable.

A veces puede ser necesario controlar la frecuencia de ejecución de las instrucciones del bucle insertando en él una llamada al método estático `sleep` de la clase `Thread`. Esta método "duerme" el hilo actual durante el número de milisegundos que se pasa como parámetro. La llamada a este método se debe proteger con un bloque `try catch`.

```
Thread t;
t= Thread.currentThread();
while(th==t)
{
      // código a ejecutar por el hilo
      // duerme el hilo durante 500 ms
     try
     {
          Thread.sleep(500);
     }
     catch (InterruptedException e){}
}
```

c. Lanzar y parar un hilo

El lanzamiento de un hilo se activa con una llamada a su método `start`. Esta llamada provoca la ejecución del método `run` del hilo. En ningún caso se debe llamar directamente al método `run` del hilo porque se ejecutaría dentro del hilo actual (el principal) del applet. Esto volvería inútiles nuestros esfuerzos e incluso podría provocar el bloqueo del applet si el método `run` contuviera un bucle (al no poder ejecutarse más el código que permite avanzar en la condición de salida del bucle ya que en este caso la ejecución del método `run` monopoliza el hilo principal).

El detención del hilo está provocada por el fin de la ejecución de su método `run` ya sea porque terminó la última instrucción que contiene o porque terminó el bucle que contiene. El método `stop`, a pesar de estar presente en la clase `Thread`, no debería utilizarse ya que conlleva riesgos de bloqueo de la aplicación.

Para ilustrar la utilización de los hilos, a continuación veremos un applet que permite engordar y adelgazar a Duke de manera continua.

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

public class TestApplet5 extends Applet implements Runnable
{
     Thread th;
     final int MAXI=100;
     final int MINI=10;
     int anchura=MINI;
     int altura=MINI;
     Imagen img;

     public void run()
     {
           boolean engordar=true;
           Thread t;
           t=Thread.currentThread();
           while (th==t)
           {
                if(anchura>MAXI & engordar)
                {
                     engordar=false;
                }
                if (anchura<MINI & !engordar)
                {
                     engordar=true;
                }
                if (engordar)
                {
                     anchura++;
                     altura++;
                }
                else
                {
                     anchura--;
                     altura--;
                }
                repaint();
                try
                {
                     th.sleep(10);
                }
                catch (InterruptedException e)
                 {}
           }

     }

     public void init()
     {
          img=getImage(getCodeBase(),"imágenes/duke.gif");
     }

     public void start()
     {
          th=new Thread(this);
          th.start();
     }

     public void stop()
     {
          th=null;
     }

     public void paint(Graphics g)
     {
          g.drawImage(img,30,0,anchura,altura,this);
     }
}
```

5. Los sonidos en los applets

La clase `Applet` proporciona dos métodos que permiten la carga de un archivo de audio. El método `getAudioClip` sólo puede invocarse desde un applet, mientras que el método `newAudioClip` puede invocarse desde cualquier tipo de aplicación ya que se declara `static` en la clase `Applet` y por lo tanto es accesible sin que exista una instancia de la clase `Applet`. Estos dos métodos reciben como argumento un objeto `URL` que representa la ubicación del archivo de audio. También está disponible la sobrecarga del método `getAudioClip` que recibe como segundo parámetro una cadena de caracteres. Este segundo parámetro representa la ruta relativa a la URL pasada como primer parámetro para obtener el archivo de audio. Este método es muy útil porque permite emplear los métodos `getCodeBase` o `getDocumentBase` para obtener la URL del applet o la URL de la página html y, a continuación, especificar la ruta de acceso al archivo de audio en relación con esta URL. Estos métodos devuelven una instancia de clase que implementa la interfaz `AudioClip`. Esta interfaz define los métodos `play`, `loop` y `stop` que permiten escuchar el archivo de audio una vez, en bucle o pararlo.

Los sonidos en los applets deberían utilizarse con moderación y precaución para evitar de enfadar rápidamente al usuario del applet. Por ejemplo hay que tener en cuenta que los archivos de audio son a veces voluminosos y por ello tardan en ser descargados. Si el método `getAudioClip` se utiliza en el método `init` del applet, habrá que esperar el final de la descarga del archivo de audio para que el applet se pueda utilizar. Es preferible lanzar, en este método `init`, un hilo responsable de descargar el archivo de audio y lanzar su restitución al final de la descarga.

También hay que recordar que la máquina virtual Java del navegador conserva las instancias de los applets creados hasta el cierre del navegador. Si un applet ejecutó el método `loop` en un objeto `AudioClip`, se restituirá el fichero en bucle hasta el cierre del navegador incluso si el usuario navega por una página diferente de la que contiene el applet. Por lo tanto es prudente llamar al método `stop` del objeto `AudioClip` en el método `stop` del applet.

```
import java.applet.Applet;
import java.applet.AudioClip;

public class TestAppletAudio extends Applet implements Runnable
{
     AudioClip ac;
     public void init()
     {
          Thread th;
          th=new Thread(this);
          th.start();
     }

     public void start()
     {
          if (ac!=null)
          {
               ac.loop();
          }
     }
     public void stop()
     {
          if (ac!=null)
          {
               ac.stop();
          }
     }
     public void run()
     {
          ac=getAudioClip(getCodeBase(),"sonido.wav");
          ac.loop();
     }
}
```

Despliegue de un applet

Para que los usuarios puedan visualizar un applet, éste debe integrarse en una página html. Antes de realizar dicha inserción en una página html, conviene almacenar en un archivo jar el conjunto de archivos necesarios para su funcionamiento. Es, también, conveniente firmar el archivo creado para garantizar la autenticidad del applet. Ambas operaciones se describen en el capítulo dedicado al despliegue.

1. La etiqueta <applet>

Esta etiqueta es la etiqueta estándar del html para la inserción de un applet. La configuración del applet se hace por medio de varios atributos de la etiqueta. Algunos de estos atributos son obligatorios.

- `code="nombre de la clase del applet"`: este atributo permite indicar al navegador el nombre del archivo que contiene la clase principal del applet. No olvide, al indicar el nombre de este archivo, que estamos en el mundo Java y que hay distinción entre minúsculas y mayúsculas. Por lo tanto respete bien las mayúsculas / minúsculas de este nombre de archivo. A falta de otra información, el navegador intentará cargar este archivo a partir de la misma ruta de donde procede la página html.
- `width="anchura en píxeles"`: este atributo indica al navegador la anchura de la superficie que debe reservar en la página html para la visualización del applet.
- `height="altura en píxeles"`: este atributo indica al navegador la altura de la superficie que debe reservar en la página html para la visualización del applet.

La sintaxis mínima básica de la etiqueta applet es por lo tanto la siguiente:

```
<applet code= "duke.class" width="200" height="200"> </applet>
```

Los atributos siguientes forman parte también de la etiqueta applet:

- `codebase="ruta de acceso"`: este atributo indica al navegador la ruta de acceso al archivo que contiene la clase principal del applet si éste no está en el mismo directorio que la página html que contiene el applet. Si el applet forma parte de un paquete, debería indicarse su nombre junto al nombre de la clase y no con este atributo.
- `archive="nombre de un archivo jar"`: si el applet necesita varios archivos para poder funcionar, suele ser más eficaz agruparlos en un archivo Java. En este caso, el navegador sólo debe descargar este archivo para obtener todo lo necesario para hacer funcionar el applet. En caso contrario, debe descargar los archivos uno a uno creando, cada vez que se descarga un archivo, una nueva conexión http con el servidor. Se debe generar el archivo mediante la herramienta `jar`, cuyo funcionamiento se detalla en el capítulo dedicado al despliegue de aplicaciones.
- `align="constante de alineación"`: este atributo indica cómo se alinea el applet en relación con el elemento siguiente en la página html. Se dispone de ocho valores de alineación:
  - `left`: el applet se alinea a la izquierda del elemento siguiente.
  - `right`: el applet se alinea a la derecha del elemento siguiente.
  - `texttop`: el alto del applet se alinea con el alto del elemento texto siguiente.
  - `top`: el alto del applet se alinea con el alto del elemento siguiente.
  - `absmiddle`: el centro del applet se alinea con el centro del elemento siguiente.
  - `middle`: el centro del applet se alinea con el centro de la línea de base del texto siguiente.
  - `bottom`: la parte inferior del applet se alinea en la línea de base del texto siguiente.
  - `absbottom`: la parte inferior del applet se alinea en el elemento más bajo siguiente.
- `vspace="espacio vertical en píxeles"`: este atributo indica al navegador el espacio que debe dejar libre encima y debajo del applet.
- `hspace="espacio horizontal"`: este atributo indica al navegador el espacio que debe dejar libre a izquierda y derecha del applet.

La etiqueta `applet` puede ella misma contener texto. Este texto se muestra en el navegador si no es capaz de ejecutar el applet Java.

Ejemplo de etiqueta <applet>:

```
<applet code="TestApplet4"
   width="150"
   height="150"
   align="left"
   vspace="50"
   hspace="50"
   codebase="appletsApplication"
   archive="data.jar"
 >
Su navegador no gestiona los applets
</applet>
```

2. Configuración de un applet

Las aplicaciones clásicas pueden recibir datos en tiempo de ejecución mediante los parámetros que se pasan por línea de comando. Estos parámetros se utilizan en el código de la aplicación gracias al array de strings que se recibe como argumento en el método `main` de la aplicación. Los applets, al no ejecutarse a través de la línea de comandos sino desde el navegador que analiza la página html, define los parámetros que recibe en el interior de la propia página. A continuación el código del applet los recupera.

a. Definir los parámetros

Los parámetros se definen mediante etiquetas `<param>` anidadas en el interior de la etiqueta `<applet>`. Estas etiquetas deben disponer obligatoriamente de dos atributos. El primero, `name`, permite identificar el parámetro. El segundo, `value`, corresponde al valor del parámetro.

b. Recuperación de los parámetros en el applet

Se obtiene el valor de un parámetro con el método `getParameter` de la clase `applet`. Recibe como argumento una cadena de caracteres que representa el nombre del parámetro cuyo valor esperamos obtener. Este método devuelve siempre una cadena de caracteres. Si un parámetro representa otro tipo de datos, habrá que convertirlo de manera explícita dentro del código del applet utilizando por ejemplo las clases `wrapper(Integer,Float,Long,...)`. Si no existe en la etiqueta `<applet>` un parámetro con el nombre indicado, el método `getParameter` devuelve un valor `null`.

Por supuesto, esta situación no debe afectar la ejecución del applet que debe proporcionar un valor por defecto para el parámetro que falta. La recuperación de los parámetros se puede realizar en el método `init` del applet como en el ejemplo siguiente:

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

public class TestApplet5 extends Applet implements Runnable
{
     Thread th;
     int MAXI=100;
     int MINI=10;
     int anchura=MINI;
     int altura=MINI;
     Image img;


     public void init()
     {
          String min;
          String max;
          min=getParameter("minimum");
          if(min!=null)
          {
               MINI=Integer.parseInt(min);
          }
          max=getParameter("maximum");
          if(max!=null)
          {
               MAXI=Integer.parseInt(max);
          }
          MAXI=Integer.parseInt(max);
          img=getImage(getCodeBase(),"imágenes/duke.gif");

     }
...
}
```

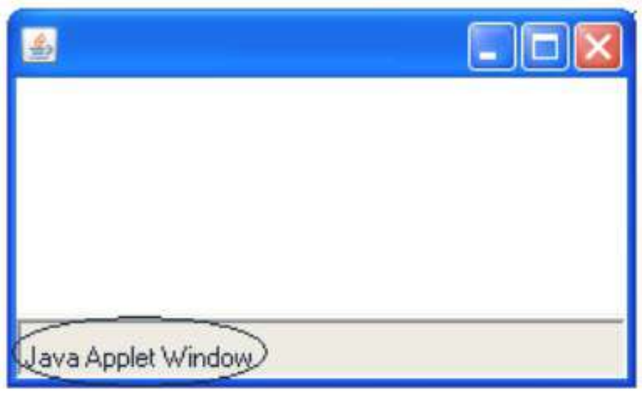
El código siguiente permite insertar el applet en una página html especificando un valor para los dos parámetros que espera recibir.

```
<applet code="TestApplet5" width="250" height="250">
<param name="minimum" value="20">
<param name="maximum" value="250">
</applet>
```

3. Seguridad en un applet

El mundo de Internet no disfruta de una fama de seguridad absoluta. Y es aún más notable cuando se trata de código que procede de Internet como es el caso para un applet. Para garantizar la seguridad del sistema sobre el cual se ejecuta un applet, los navegadores establecen una política de seguridad bastante estricta respeto a un applet. Cada navegador implementa su propia política de seguridad pero de forma general se aplican las siguientes restricciones a un applet.

- Un applet no puede cargar una biblioteca ni tampoco llamar un método nativo como por ejemplo una función del sistema operativo. Por lo tanto los applets deben conformarse con su propio código y sus propias funcionalidades puestas a su disposición por la máquina virtual Java que asegura su ejecución.
- Un applet no puede leer un archivo existente en la máquina sobre la cual se ejecuta. Esta limitación también se aplica lógicamente a la escritura de un archivo y a su eliminación.
- Un applet no puede establecer una conexión de red con otra máquina salvo con la máquina de donde proviene.
- Un applet no puede lanzar la ejecución de una aplicación en la máquina sobre la cual se ejecuta él mismo.
- Un applet dispone de un acceso limitado a las propiedades del sistema.
- Cuando un applet muestra una ventana durante su ejecución, esta ventana está marcada para señalar de manera efectiva al usuario que procede de la ejecución del applet y no de una aplicación local.



4. Comunicación entre applets

Los applets presentes en una página html tienen la posibilidad de dialogar entre sí. Este diálogo sólo es posible bajo ciertas condiciones impuestas por el navegador.

- Los applets deben proceder todas del mismo servidor;
- También deben proceder de la misma carpeta en este servidor (codebase idéntico);
- También se deben ejecutar en la misma página, en una misma ventana de navegador.

Para que un applet pueda establecer un diálogo con otro applet, éste debe obtener una referencia del applet a contactar. Esta referencia se puede obtener utilizando el método `getApplet` y pasándole como argumento el nombre del applet en cuestión. Naturalmente, el applet debe estar nombrado en el momento de su inserción en la página html. Es posible asignar nombre a un applet informando, en su etiqueta, un atributo o un parámetro `name`.

Las dos sintaxis siguientes son idénticas:

```
<applet code="TestApplet5" width="250" height="250">
<param name="minimum" value="20">
<param name="maximum" value="250">
<param name="name" value="duke">
</applet>


<applet code="TestApplet4" width="250" height="250" name="duke">
</applet>
```

El método `getApplet` se asocia al contexto del applet del que podemos obtener una referencia mediante el método `getAppletContext`.

Es prudente comprobar el valor devuelto por este método para confirmar que se encontró realmente el applet en la página, antes de intentar acceder a él.

```
Applet ap;
ap=getAppletContext().getApplet("duke1");
if (ap!=null)
{
     ap.setBackground(Color.CYAN);
}
```

Esta solución exige por supuesto conocer el nombre con el cual se insertó el applet en la página html.

Una segunda solución permite evitar este inconveniente obteniendo el listado de todos los applets presentes en la página html. En este caso, el método `getApplets` permite obtener una enumeración del listado de los applets.

```
Applet ap;
Enumeration e;
e=getAppletContext().getApplets();
while (e.hasMoreElements())
{
     ap=(Applet)e.nextElement();
     if (ap!=this)
     {
          ap.setBackground(Color.CYAN);
     }
     else
     {
          ap.setBackground(Color.PINK);
     }
}
```

Conviene ser prudente con esta solución porque obviamente el applet desde el cual se ejecuta este código forma parte del listado (¡está también en la página html!). En este caso, conviene realizar una comprobación si no queremos que el tratamiento se aplique al applet actual o si deseamos efectuar un tratamiento diferente.

5. Interacción con el navegador y el sistema

El navegador gestiona los applets y analiza la página html en la cual están insertados. Por lo tanto podemos interactuar con ellos. También los applets tienen derecho a un acceso limitado a ciertas propiedades del sistema.

a. Visualización en la consola

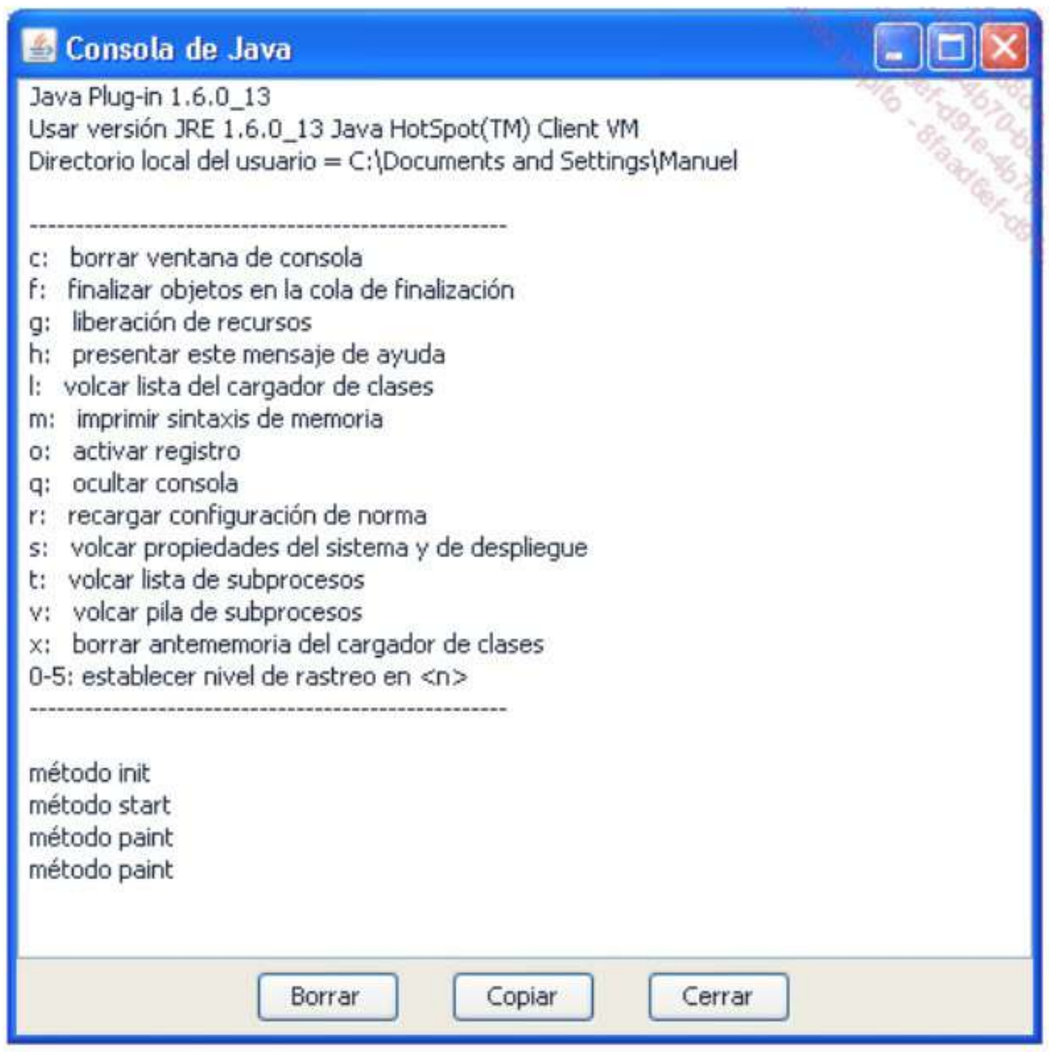
Durante el desarrollo de un applet, nos daremos cuenta rápidamente de que desgraciadamente nos falta un elemento fundamental: la consola. Efectivamente, juega un papel importante durante la fase de pruebas de una aplicación porque nos muestra mensajes para indicar que se ejecutó correctamente una sección de código, o también porque nos muestra el contenido de ciertas variables. Realmente esta consola sí que está disponible incluso para un applet pero en este caso el navegador se encarga de ella. Como consecuencia, es el navegador el que realiza una función de agente de la consola. La forma de visualizar la consola es específica de cada navegador.

El aspecto de la consola depende también del navegador. Puede ser una mera ventana de Símbolo de sistema en modo texto o una aplicación gráfica como en el caso de Internet Explorer. Sea cual sea su aspecto, los dos flujos `System.out` y `System.err` siempre se dirigen a esta consola. Para nuestro primer applet, hubiéramos podido utilizar el código siguiente:

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


public class TestApplet extends Applet
{
     public void destroy()
     {
          System.out.println("método destroy");
     }

     public void init()
```

```
    {
        System.out.println("método init");
    }

    public void start()
    {
        System.out.println("método start");
    }

    public void stop()
    {
        System.out.println("método stop");
    }

    public void paint(Graphics g)
    {
        System.out.println("método paint");
    }

}
```

Y de esta manera obtener el resultado siguiente en la consola:



b. Utilización de la barra de estado del navegador

También podemos utilizar la barra de estado del navegador para la visualización de mensajes destinados al usuario del applet. El acceso a la barra de estado se hace gracias al método `showStatus`. Este método recibe como argumento la cadena de caracteres a visualizar. A veces esta barra de estado no es muy visible y sobre todo su contenido puede verse modificado en cualquier momento por otro applet o por el propio navegador.

c. Visualización de una página html

La visualización de un documento html es por supuesto la especialidad de un navegador. Por lo tanto, el applet debe dirigirse al navegador si necesita visualizar un documento. El método `showDocument` del contexto del applet permite llevar a cabo esta operación. Puede hacer uso de este método bajo dos formas. La primera recibe como parámetro la URL del documento a visualizar. Se muestra el documento en lugar de la página html donde se encuentra el applet.

```
Try
{
    getAppletContext().showDocument(new URL("http://www.eni.es"));
}
catch (MalformedURLException e)
{
    e.printStackTrace();
}
```

Este método no recibe como parámetro una cadena de caracteres sino una instancia de la clase URL. Ésta debe crearse a partir de la cadena de caracteres que representa la URL de la página a visualizar. La utilización del constructor de la clase URL exige la presencia del bloque try catch o de una instrucción throws.

La segunda forma recibe una cadena de caracteres como segundo parámetro para identificar la ubicación donde se visualizará la página. Esta cadena acepta los valores siguientes.

`_blank`: se visualiza el documento en una nueva ventana del navegador.

`_self`: se visualiza el documento en lugar de la página donde se encuentra el applet.

`_parent`: la ventana html que contiene el applet visualiza el documento en su ventana madre.

`_top`: se visualiza el documento en el marco de mayor nivel.

`nombreDeMarco`: se visualiza el documento en el marco que lleva el nombre especificado. Si este marco no existe, se crea una nueva ventana para visualizar el documento.

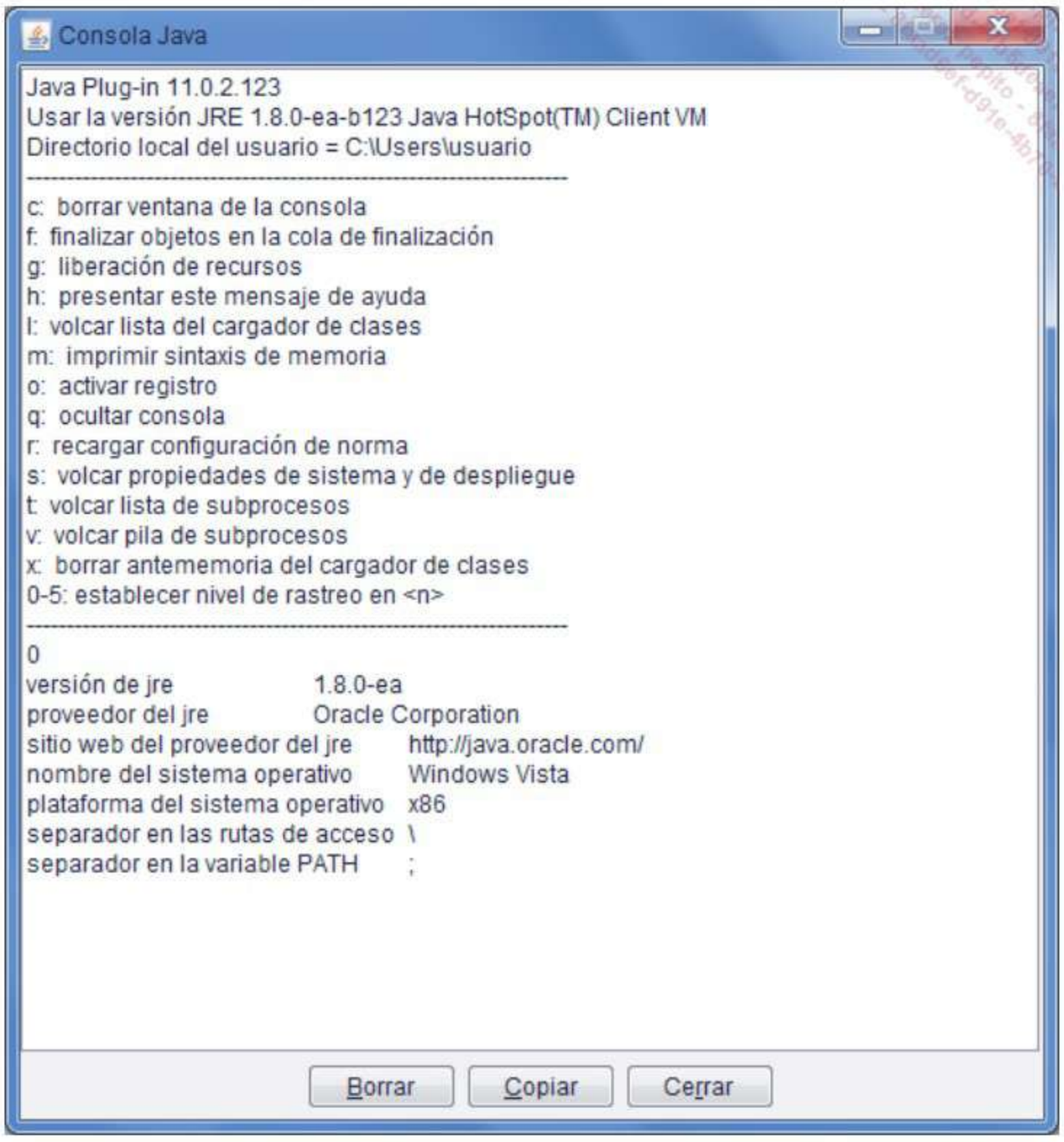
d. Obtener ciertas propiedades del sistema

Para que el applet se pueda adaptar mejor al entorno donde se ejecuta, tiene que tener acceso a ciertas propiedades del sistema. Estas propiedades son accesibles mediante el método `getProperty` de la clase `System`. Este método recibe como parámetro una cadena de caracteres que indica el nombre de la propiedad cuyo valor queremos obtener. El applet puede acceder a las propiedades siguientes:

Nombre de la propiedad	Valor obtenido
`file.separator`	Carácter utilizado como separador en las rutas de acceso a los archivos
`path.separator`	Carácter utilizado como separador entre dos rutas de acceso (variable de entorno path)
`java.vendor`	Nombre del proveedor del JRE
`java.vendor.url`	URL del sitio web del proveedor del JRE
`java.version`	Versión del JRE
`line.separator`	Carácter de separación de líneas
`os.arch`	Plataforma del sistema operativo
`os.name`	Nombre del sistema operativo

El applet siguiente muestra estas diferentes propiedades en la consola Java del navegador:

```
import java.applet.Applet;


public class Propiedades extends Applet
{

  public void start()
  {
  System.out.print("versión del jre \t");
  System.out.println(System.getProperty("java.version"));
  System.out.print("proveedor del jre \t");
  System.out.println(System.getProperty("java.vendor"));
  System.out.print("sitio web del proveedor del jre \t");
  System.out.println(System.getProperty("java.vendor.url"));
  System.out.print("nombre del sistema operativo \t");
  System.out.println(System.getProperty("os.name"));
  System.out.print("plataforma del sistema operativo \t");
  System.out.println(System.getProperty("os.arch"));
  System.out.print("separador en las rutas de acceso \t");
  System.out.println(System.getProperty("file.separator"));
  System.out.print("separador en la variable PATH \t");
  System.out.println(System.getProperty("path.separator"));
  }
}
```


Consola Java
Java Plug-in 11.0.2.123
Usar la versión JRE 1.8.0-ea-b123 Java HotSpot(TM) Client VM
Directorio local del usuario = C:\Users\usuario
c: borrar ventana de la consola
f: finalizar objetos en la cola de finalización
g: liberación de recursos
h: presentar este mensaje de ayuda
l: volcar lista del cargador de clases
m: imprimir sintaxis de memoria
o: activar registro
q: ocultar consola
r: recargar configuración de norma
s: volcar propiedades de sistema y de despliegue
t: volcar lista de subprocesos
v: volcar pila de subprocesos
x: borrar antememoria del cargador de clases
0-5: establecer nivel de rastreo en <n>
0
versión de jre 1.8.0-ea
proveedor del jre Oracle Corporation
sitio web del proveedor del jre http://java.oracle.com/
nombre del sistema operativo Windows Vista
plataforma del sistema operativo x86
separador en las rutas de acceso \
separador en la variable PATH ;
Borrar
Copiar
Cerrar

Principios del funcionamiento de una base de datos

Las bases de datos 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 (líneas) 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. Une 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 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 a 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.

```
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 2005.

```
SELECT * from Pedidos WHERE FechaPedido BETWEEN '01/11/05' AND '30/11/05'
```

Cláusula WHERE ... LIKE

Se puede utilizar la cláusula `WHERE ... LIKE` para devolver todos los registros para los cuales existe una condición particular para 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, sólo 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 para 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 a actualizar así como el valor a 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 sólo 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 líneas 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 sólo 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 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:

- Comunicar 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 biblioteca jdbc para el acceso a una base de datos. Precisamente, la biblioteca 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 Sun 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 resultados 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 sólo ofrece resultados mediocres porque dependen del número de capas 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

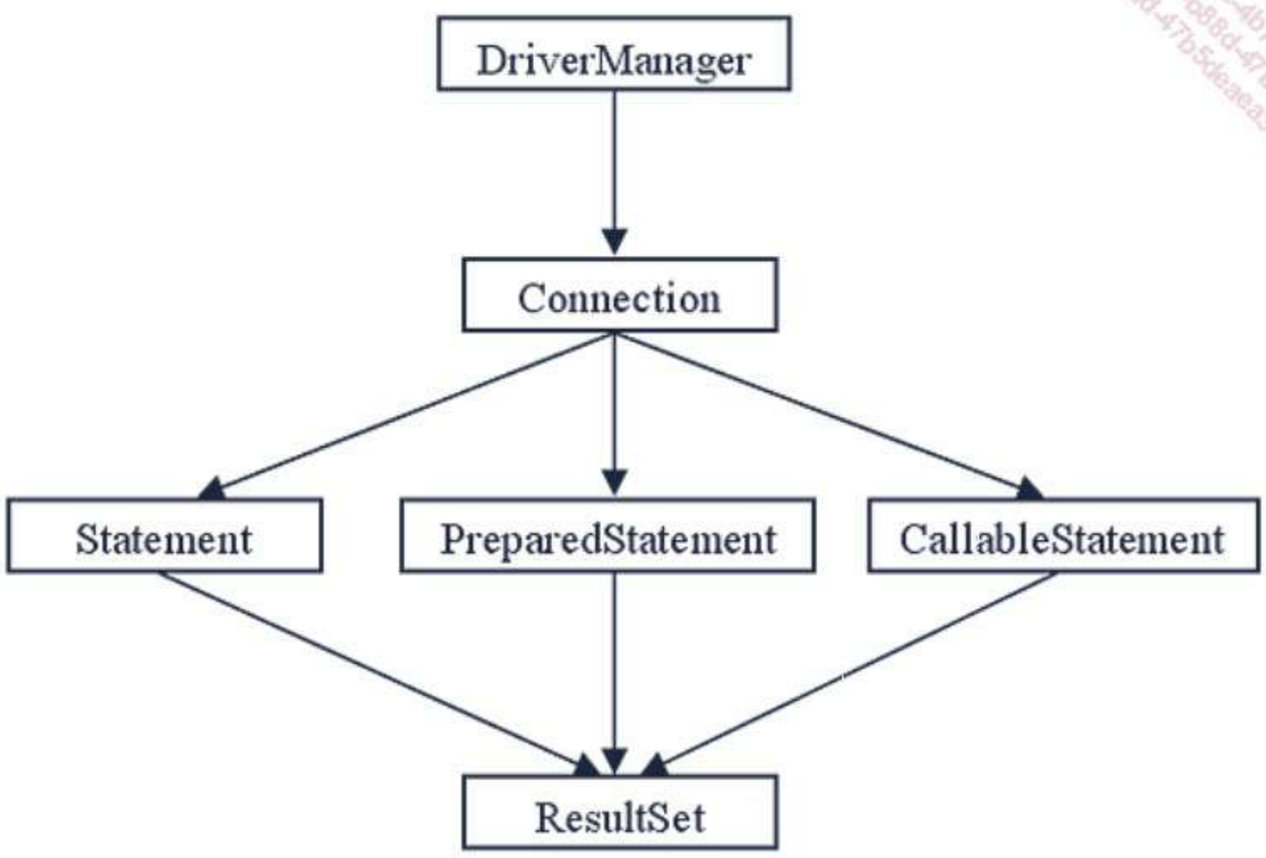
  Este puente de protocolo de red, desarrollado completamente en Java, no comunica directamente con la base de datos sino que se dirige a un servidor intermediario. El diálogo entre el driver y el servidor intermediario es estándar sea cual sea el tipo de base de datos. Es, a continuación, este servidor intermediario quien transmite las peticiones utilizando un protocolo específico a 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 biblioteca 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 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 párrafos 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 Microsoft para acceder a un servidor de base de datos SQL Server. Es posible descargarlo en la dirección URL siguiente: http://www.microsoft.com/es-es/download/details.aspx?id=11774

Tras descomprimir el archivo, obtendrá una carpeta que contiene el propio driver bajo la forma de un archivo java (sqljdbc.jar) y varios archivos de ayuda relativos al uso de este controlador. El archivo jar contiene las clases desarrolladas por Microsoft 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.

A continuación, se debe cargar este driver mediante 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.microsoft.sqlserver.jdbc.SQLServerDriver
```

El registro del driver se crea por lo tanto con la instrucción siguiente:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

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 facilitarnos 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 con la forma siguiente `jdbc:nombreDelProtocolo`. El nombre del protocolo es propio a cada driver y es gracias a este nombre como 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 Microsoft, la sintaxis básica es la siguiente:

```
jdbc:sqlserver://localhost;databaseName=northwind;
user=usuario;password=secreto;
```

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.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConexionDirecta
{
public static void main(String[] args)
{
     try
     {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
     }
     catch (ClassNotFoundException e)
     {
       System.out.println("error durante la carga del driver");
     }
     Connection cnxDirect=null;
     try
     {
       cnxDirect=DriverManager.getConnection("jdbc:sqlserver:
//localhost;databaseName=northwind;
user=thierry;password=secret;");
     }
     catch (SQLException e)
     {
       System.out.println("error durante la conexión");
     }
     }
}
```

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`. Después del cierre de una conexión, ya no es posible usarla y debe volver a crearse para poder utilizarla de nuevo. 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 al intentar enviar una instrucción SQL y verificar que obtiene una respuesta de parte del servidor. Este método no está implementado en

todos los drivers, en cuyo caso produce una excepción del tipo `java.lang.UnsupportedOperationException` o un error del tipo `java.lang.AbstractMethodError`.

Si únicamente se efectúan operaciones de lectura en la base de datos, podemos optimizar la comunicación precisando que la conexión está en sólo 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 testSóloLectura(Connection cnx)
  {
      boolean estado;
      try
      {
            estado = cnx.isReadOnly();
            cnx.setReadOnly(!estado);
            if (cnx.isReadOnly() != estado)
            {
                  System.out.println("este driver se encarga
del modo sólo lectura");
            }
            else
            {
                  System.out.println("este driver no se encarga
del modo sólo 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(aviso.getMessage());
                        System.out.println(aviso.getSQLState());
                         System.out.println(aviso.getErrorCode());
                        aviso=aviso.getNextWarning();
                  }
            }
            cnx.clearWarnings();
      }
      catch (SQLException e)
      {
                  e.printStackTrace();
      }
   }
```

En general, en el momento de la creación de la URL 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:

```
        System.out.println("base actual: " +cnxDirect.getCatalog());
      System.out.println("cambio de base de datos");
      cnxDirect.setCatalog("garaje");
      System.out.println("base actual: " +cnxDirect.getCatalog());
```

nos muestra:

```
base actual: northwind
cambio de base de datos
base actual: garaje
```

Es posible, también, obtener la estructura de la base de datos mediante el método `getMetaData`. Este método devuelve un objeto `DataBaseMetaData` que facilita numerosa información relativa a la estructura de la base de datos. La función siguiente muestra un análisis rápido de esta información.

Todos estos métodos pueden ser de ayuda algún día, pero la meta principal de una conexión es permitir la ejecución de instrucciones SQL. Por lo

```
public static void infosBase(Connection cn)
{
    ResultSet rs;
    DatabaseMetaData dbmd;
    try
    {
        dbmd=cn.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\tesquema\tnombre tabla\
ttipo 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();
    }
}
```

tanto, es ella la que facilitará 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 un objeto JDBC:

- `Statement` para las peticiones simples
- `PreparedStatement` para las peticiones precompiladas
- `CallableStatement` para los procedimientos almacenados.

Para terminar, es posible obtener cada uno de estos objetos a partir de un método diferente de la conexión:

- `createStatement` para los objetos `Statement`
- `prepareStatement` para los objetos `PreparedStatement`
- `prepareCall` para los objetos `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 Statement

Este objeto 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`), éste estará en sólo lectura, y con un desplazamiento de los datos hacia delante.

Los datos presentes en este juego de registros no podrán modificarse y sólo se podrá hacer el recorrido del juego de registros desde el primero hacia el último registro. 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:

- `ResultSet.TYPE_FORWARD_ONLY`: el juego de registros será de desplazamiento hacia adelante 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 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 sólo 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*). Sólo 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 `booleano` 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.

```
public static void testEjecuta(Connection cnx)
   {
      Statement stm;
      BufferedReader br;
      String petición;
      ResultSet rs;
      boolean resultado;
      try
      {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
            br=new BufferedReader(new InputStreamReader(System.in));
            System.out.println("introducir su instrucción SQL:");
            petición=br.readLine();
            resultado=stm.ejecuta(petición);
            if (resultado)
            {
                  System.out.println("su instrucción generó un juego
de registros");
                  rs=stm.getResultSet();
                  rs.last();
                  System.out.println("contiene " + rs.getRow() + "
registros");
            }
            else
            {
                  System.out.println("su instrucción modificó
registros en la base");
                  System.out.println("número de registros modificados:"
+ stm.getUpdateCount());
            }
      }
      catch (SQLException e)
      {
            System.out.println("su instrucción no funcionó
correctamente");
      }
      catch (IOException e)
      {
            e.printStackTrace();
      }
   }
```

- `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 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 a 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.

```
public static void testExecuteBatch(Connection cnx)
{
```

```
    Statement stm;
    BufferedReader br;
    String petición="";
    int[] resultados;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
      br=new BufferedReader(new InputStreamReader(System.in));
            System.out.println("introducir sus instrucciones SQL
y después run para ejecutar el lote:");
            petición=br.readLine();
            while (!peticion.equalsIgnoreCase("run"))
            {
                  stm.addBatch(petición);
                  requete=br.readLine();
            }
            System.out.println("ejecución del lote
de instrucciones");
            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 + " falló");
                        break;
                  case Statement.SUCCESS_NO_INFO:
                        System.out.println("la ejecución
de la instrucción " + i + " funcionó");
                        System.out.println("el número de
registros modificados es desconocido");
                        break;
                  default:
                        System.out.println("la ejecución
de la instrucción " + i + " funcionó");
                        System.out.println("modificó " + resultados[i]
+ " registros");
                        break;
                  }


            }
    }
    catch (SQLException e)
    {
          e.printStackTrace();
    }
    catch (IOException e)
    {
          e.printStackTrace();
    }
}
```

Otros métodos van a intervenir en el comportamiento del objeto `Statement`:

- `public void setQueryTimeOut(int duración)`: 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 a 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.
- `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 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 sólo 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 es accesible, también, 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 a 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:

```
public static void testExecuteMultiple(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String petición;
    ResultSet rs;
    boolean resultado;
```

```
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
          br=new BufferedReader(new InputStreamReader(System.in));
          System.out.println("introducir sus instrucciones SQL
separadas por; :");
          petición=br.readLine();
          resultado=stm.execute(petición);
          int i=1;
          // tratamiento del resultado generado por la primera
          // instrucción
          if (resultado)
          {
                System.out.println("su instrucción N° " + i + "
generó un juego de registros");
                rs=stm.getResultSet();
                rs.last();
                System.out.println("contiene " + rs.getRow() + "
registros");
          }
          else
          {
                System.out.println("su instrucción N° " + i + "
modificó 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 N° " +
i + " generó un juego de registros");
                   rs=stm.getResultSet();
                   rs.last();
                   System.out.println("contiene " + rs.getRow() +
" registros");
                }
                else
                {
                   System.out.println("su instrucción N° " +
i + " modificó 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 funcionó
correctamente");
          e.printStackTrace();
    }
    catch (IOException e)
    {
          e.printStackTrace();
    }
}
```

b. Ejecución de instrucciones configuradas con el objeto PreparedStatement

Ocurre a menudo que hay que ejecutar varias veces una petición SQL con sólo una pequeña modificación entre dos ejecuciones. El ejemplo clásico corresponde a una petición de selección con una restricción.

```
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
resultado=stm.execute("select * from customers where customerId=
'ALFKI'");
```

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.

```
public static void testPeticiónConcat(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String petición;
    String código;
    ResultSet rs;
    boolean resutlado;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
```

```
ResultSet.CONCUR_READ_ONLY);
         br=new BufferedReader(new InputStreamReader(System.in));
         System.out.println("introducir el código del cliente
buscado:");
         código=br.readLine();
         petición="select * from customers where customerID=\'" +
código + "\'";
         resultado=stm.execute(petición);
    }
    catch (SQLException e)
    {
         e.printStackTrace();
    } catch (IOException e) {
         // TODO Bloc catch auto-generado
         e.printStackTrace();
    }
}
```

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.

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 sólo lectura y de desplazamiento únicamente hacia adelante. 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`.

```
stm=cnx.prepareStatement("select * from customers where
customerID=?",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 a insertar en lugar de cada signo de interrogación. Estos métodos siguen la siguiente nomenclatura: setXXXX donde XXXX 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 a transferir en el parámetro. El tipo de este argumento corresponde por supuesto al tipo de datos SQL a 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. 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:

```
  stm=cnx.prepareStatement("select * from customers where
?=?",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
stm.setString(1,"customerID");
stm.setString(2,"ALFKI");
rs=stm.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`.

```
public static void testPreparedStatement(Connection cnx)
  {
      {
            PreparedStatement stm;
            BufferedReader br;
            String code;
            ResultSet rs;
            try
            {
                  stm=cnx.prepareStatement("select * from customers
where customerID like ?",ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
                  br=new BufferedReader(new
InputStreamReader(System.in));
                  System.out.println("introducir el código del
cliente buscado:");
                  código=br.readLine();
                  stm.setString(1,código);
                  rs=stm.executeQuery();
                  while (rs.next())
                  {
                        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();
            }
        }
    }
```

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 nos facilitará una instancia de clase correcta. En este caso, se debe utilizar el método `prepareCall`. Este método recibe como argumento una cadena de caracteres que identifica el procedimiento almacenado a 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. Hay que tener en cuenta dos supuestos según si el procedimiento devuelve o no un valor. Si el procedimiento almacenado no devuelve un valor, la sintaxis de esta cadena de caracteres es la siguiente:

```
{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 `setXXXX` correspondientes al tipo del parámetro.

Si el procedimiento almacenado devuelve un valor, hay que utilizar la sintaxis siguiente que añade un parámetro adicional para recuperar el retorno del procedimiento almacenado.

```
{ ?=call nombreProcedimiento( ?, ?, ...)}
```

Al utilizar el primer parámetro como salida del procedimiento almacenado, su ejecución almacenará un valor en él, y debemos informar el objeto `CallableStatement` invocando al método `registerOutParameter`. Este método recibe como primer argumento el índice del parámetro 1 para el valor de retorno del procedimiento almacenado, pero se puede utilizar cualquier otro parámetro como salida, 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 `getXXXX` donde `XXXX` 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 los dos procedimientos almacenados siguientes:

```
create PROCEDURE pedidosPorCliente @código nchar(5)
AS
SELECT OrderID,
  OrderDate,
  RequiredDate,
  ShippedDate
FROM Orders
WHERE CustomerID = @código
ORDER BY OrderID

CREATE procedure numPedidos @código nchar(5) as
declare @nb int
select @nb=count(*) from Orders where customerid=@código
return @nb
```

El primero devuelve la lista de todos los pedidos del cliente cuyo código se pasa como parámetro. El segundo devuelve un valor entero que corresponde al número de pedidos correspondientes al cliente cuyo código se pasa como parámetro.

```
public static void testProcedimentoAlmacenado(Connection cnx)
 {
     CallableStatement cstm1,cstm2;
     BufferedReader br;
     String código;
     ResultSet rs;
     int numPedidos;
     try
     {
          br=new BufferedReader(new InputStreamReader(System.in));
          System.out.println("introducir el código del cliente
buscado:");
          código=br.readLine();
          cstm1=cnx.prepareCall("{ ?=call numPedidos ( ? )}");
          cstm1.setString(2,código);
          cstm1.registerOutParameter(1,java.sql.Types.INTEGER);
          cstm1.execute();
          numComandos=cstm1.getInt(1);
          System.out.println("número de pedidos correspondientes
```

```
al cliente " + código + " : " + numPedidos );
          cstm2=cnx.prepareCall("{ call pedidosPorCliente ( ?
)}",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
          cstm2.setString(1,código);
          rs=cstm2.executeQuery();
          System.out.println("detalle de los pedidos");
          System.out.println("numéro de pedido\tfecha de pedido");
          while (rs.next())
          {
                   System.out.print(rs.getInt("OrderID") + "\t");
                   System.out.println(new
SimpleDateFormat("dd/MM/yy").format(rs.getDate("OrderDate")));
          }
     }
     catch (SQLException e)
     {
           e.printStackTrace();
     }
     catch (IOException e)
     {
           e.printStackTrace();
     }
}
```

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`, éste 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 puntero de lectura del juego de registros se desplazará sólo 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 sólo 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`.

```
public static void infosResultset(ResultSet rs)
     {
          try {
              switch (rs.getType())
              {
                     case ResultSet.TYPE_FORWARD_ONLY:
                          System.out.println("el conjunto de
registros sólo podrá recorrerse hacia delante");
                          break;
                     case ResultSet.TYPE_SCROLL_INSENSITIVE:
                          System.out.println("el conjunto de
registros 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("el conjunto de
registros 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 sólo 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) y para el final del `ResultSet` (EOF). 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 estos registros genera una excepción. En el momento de la creación del `ResultSet` el puntero se coloca antes del primer registro (BOF). 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 deplacement)`: desplaza el cursor del número de registros especificados por el argumento `deplacement`. 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`, sólo 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.

```
     public static void posiciónRs(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) {
               // TODO Bloc catch autogenerado
               e.printStackTrace();
          }
     }
}
```

b. Lectura de los datos en un ResultSet

El objeto `ResultSet` facilita numerosos métodos que permiten la lectura de los campos de un registro. Cada uno de estos métodos es específico de un tipo de datos SQL. Por supuesto, hay que elegir adecuadamente el método de lectura según el tipo de datos del campo cuyo valor se quiere obtener. Sin embargo, algunos de estos métodos son relativamente flexibles y permiten la lectura de varios tipos de datos. El cuadro siguiente presenta los principales tipos de datos SQL y los métodos que permiten leer el contenido de un `ResultSet`. Los métodos marcados con el símbolo ☺ son los métodos más aconsejados. Los métodos marcados con el símbolo 😐 podrían utilizarse, aunque conllevan riesgos de perdida de información.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT
getByte	☺	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐												
getShort	😐	☺	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐												
getInt	😐	😐	☺	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐												
getLong	😐	😐	😐	☺	😐	😐	😐	😐	😐	😐	😐	😐	😐												
getFloat	😐	😐	😐	😐	☺	😐	😐	😐	😐	😐	😐	😐	😐												
getDouble	😐	😐	😐	😐	😐	☺	☺	😐	😐	😐	😐	😐	😐												
getBigDecimal	😐	😐	😐	😐	😐	😐	😐	☺	☺	😐	😐	😐	😐												
getBoolean	😐	😐	😐	😐	😐	😐	😐	😐	😐	☺	😐	😐	😐												
getString	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	☺	☺	😐	😐	😐	😐	😐	😐							
getBytes														☺	☺	😐									
getDate											😐	😐	😐				☺		😐						
getTime											😐	😐	😐					☺	😐						
getTimeStamp											😐	😐	😐				😐	😐	☺						
getAsciiStream											😐	😐	☺	😐	😐	😐									
getUnicodeStream											😐	😐	☺	😐	😐	😐									
getBinaryStream														😐	😐	☺									
getClob																				☺					
getBlob																					☺				
getArray																						☺			
getRef																							☺		
getCharacterStream											😐	😐	☺	😐	😐	😐									
getObject	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	😐	☺	☺

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 (`Dbnull`), 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`.

```
    public static void lecturaRs(Connection cnx)
    {
          Statement stm;
          String peticion;
          ResultSet rs;
          try
          {
     stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
                peticion="select *   from products ";
                rs=stm.executeQuery(peticion);
                System.out.println("código producto\
tdesignación\tprecio unitario\tstock\tagotado\tFechaCaducidad");
                while(rs.next())
                {
     System.out.print(rs.getInt("ProductID")+"\t");

     System.out.print(rs.getString("ProductName")+"\t");

     System.out.print(rs.getDouble("UnitPrice")+"\t");
                        rs.getShort("UnitsInStock");
                        if (rs.wasNull())
                        {
                            System.out.print("desconocido\t");
                        }
                        else
                        {

     System.out.print(rs.getShort("UnitsInStock")+"\t");
                        }

     System.out.print(rs.getBoolean("Discontinued")+"\t");
                         if (rs.getDate("FechaCaducidad")!=null)
                         {
                             System.out.println(rs.getDate
("FechaCaducidad"));
                         }
                         else
                                 System.out.println("no perecedero");
                }
                rs.close();
                stm.close();
          }
          catch (SQLException e)
          {
                e.printStackTrace();
```

```
        }
    }
```

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 a actualizar. Como para el caso de los métodos `getXXX`, éstos disponen de dos sobrecargas: una recibe como argumento el índice de la columna a 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 a 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 un método `updateXXX` genera una excepción.

```
   public static void modificaciónRs(Connection cnx)
   {
        Statement stm;
        String peticion;
        ResultSet rs;
        int num=0;
        BufferedReader br;
        String respuesta;
        try
        {
   stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
               peticion="select *   from products ";
               rs=stm.executeQuery(peticion);
               System.out.println("número de línea\tcódigo
producto\tdesignación\tprecio unitario\tstock\tagotado\
tFechaCaducidad");
               while(rs.next())
               {
                         num++;
                         System.out.print(num + "\t");
     System.out.print(rs.getInt("ProductID")+"\t");

     System.out.print(rs.getString("ProductName")+"\t");

     System.out.print(rs.getDouble("UnitPrice")+"\t");
                         rs.getShort("UnitsInStock");
                         if (rs.wasNull())
                         {
                              System.out.print("desconocido\t");
                         }
                         else
                         {

     System.out.print(rs.getShort("UnitsInStock")+"\t");
                         }

     System.out.print(rs.getBoolean("Discontinued")+"\t");
                         if (rs.getDate("FechaCaducidad")!=null)
                         {
                                System.out.println(rs.getDate("FechaCaducidad"));
                         }
                         else
                              System.out.println("no perecedero");
               }
               br=new BufferedReader(new InputStreamReader
               (System.in));
               System.out.println("¿qué línea desea usted
modificar? ");
               respuesta=br.readLine();
               rs.absolute(Integer.parseInt(respuesta));
               System.out.println("designación actual" +
rs.getString("ProductName"));
               System.out.println("introducir el nuevo valor o
enter para conservar el valor actual");
               respuesta=br.readLine();
               if (!respuesta.equals(""))
               {
                   rs.updateString("ProductName",respuesta);
               }
               System.out.println("precio unitario actual " +
rs.getDouble("UnitPrice"));
               System.out.println("introducir el nuevo valor o
enter para conservar el valor actual");
               respuesta=br.readLine();
               if (!respuesta.equals(""))
               {

     rs.updateDouble("UnitPrice",Double.parseDouble(respuesta));
               }
               rs.getShort("UnitsInStock");
               if (rs.wasNull())
               {
               System.out.println ("cantidad almacenada actual
desconocida");
               }
               else
               {
               System.out.println("cantidad almacenada actual "
+ rs.getShort("UnitsInStock"));
               }
               System.out.println("introducir el nuevo valor o
```

```
enter para conservar el valor actual");
                respuesta=br.readLine();
                if (!respuesta.equals(""))
                {

     rs.updateShort("UnitsInStock",Short.parseShort(respuesta));
                }
                System.out.println("¿desea usted validar estas
modificaciones? s/n");
                respuesta=br.readLine();
                if (respuesta.toLowerCase().equals("s"))
                {
                     rs.updateRow();
                }
                else
                {
                     rs.cancelRowUpdates();
                }
                System.out.println("los valores actuales ");
                System.out.print(rs.getString("ProductName")+"\t");
                System.out.print(rs.getDouble("UnitPrice")+"\t");
                rs.getShort("UnitsInStock");
                if (rs.wasNull())
                {
                     System.out.print("desconocido\t");
                }
                else
                {
                System.out.print(rs.getShort("UnitsInStock")+"\t");
                }
                rs.close();
                stm.close();
           }
           catch (SQLException e)
           {
                e.printStackTrace();
           } catch (IOException e) {
                e.printStackTrace();
           }
     }
```

d. Supresión de datos en un ResultSet

La supresión de una línea se realiza de manera sencilla al situar el puntero sobre la línea a suprimir y llamando, a continución, 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 sobre el registro siguiente, otros lo desplazan sobre el 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.

```
     public static void supresiónRs(Connection cnx)
     {
          Statement stm;
          String petición;
          ResultSet rs;
          int num=0;
          BufferedReader br;
          String respuesta;
          try
          {
     stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
                petición="select *  from products ";
                rs=stm.executeQuery(petición);
                System.out.println("número de línea\tcódigo
producto\tdesignación\tprecio unitario\tstock\tagotado\
tFechaCaducidad");
                while(rs.next())
                {
                          num++;
                          System.out.print(num + "\t");
     System.out.print(rs.getInt("ProductID")+"\t");

     System.out.print(rs.getString("ProductName")+"\t");

     System.out.print(rs.getDouble("UnitPrice")+"\t");
                          rs.getShort("UnitsInStock");
                          if (rs.wasNull())
                          {
                               System.out.print("desconocido\t");
                          }
                          else
                          {

     System.out.print(rs.getShort("UnitsInStock")+"\t");
                          }

     System.out.print(rs.getBoolean("Discontinued")+"\t");
                          if (rs.getDate("FechaCaducidad")!=null)
                          {
                               System.out.println(rs.getDate
("FechaCaducidad"));
                          }
                          else
                               System.out.println("no perecedero");
                }
                br=new BufferedReader(new
```

```
InputStreamReader(System.in));
                System.out.println("¿qué línea desea usted
suprimir?");
                respuesta=br.readLine();
                rs.absolute(Integer.parseInt(respuesta));
                rs.deleteRow();
                System.out.println("el puntero está ahora en la
línea " + rs.getRow());
           }
           catch (Exception e)
           {
                e.printStackTrace();
           }
      }
```

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.

```
     public static void inserciónRs(Connection cnx)
     {
          Statement stm;
          String petición;
          ResultSet rs;
          int num=0;
          BufferedReader br;
          String respuesta;
          try
          {
     stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
                petición="select *  from products ";
                rs=stm.executeQuery(petición);
                br=new BufferedReader(new
InputStreamReader(System.in));
                System.out.println("introducir los valores de la
nueva línea");
                rs.moveToInsertRow();
                System.out.print("código producto: ");
                respuesta=br.readLine();
                rs.updateInt ("ProductID",Integer.parseInt
(respuesta));
                System.out.print("Designación: ");
                respuesta=br.readLine();
                rs.updateString ("ProductName",respuesta);
                System.out.print("Precio unitario: ");
                respuesta=br.readLine();
                rs.updateDouble("UnitPrice",Double.parseDouble
(respuesta));
                System.out.print("Cantidad almacenada: ");
                respuesta=br.readLine();
                rs.updateDouble("UnitsInStock",Short.parseShort
(respuesta));
                rs.insertRow();
          }
          catch (Exception e)
          {
                e.printStackTrace();
          }
}
```

6. Gestión de las transacciones

Las transacciones van a permitir asegurar que un conjunto de instrucciones SQL se ejecuten todas con éxito o bien que no se ejecute ninguna. 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
        {

     Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
             Connection cnx=null;

     cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;
database Name=banco; user=sa;password=;");
             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, salvo que uno de estos días la impresora, cargada con 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 el efecto de las instrucciones SQL ya ejecutadas realizando la operación inversa, es decir, abonando en la cuenta, en nuestro caso.

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 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 al 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 de nuestro método que permite transferir un importe entre dos cuentas debería tener la forma siguiente.

```
public static void movimiento1(String cuentaDebito,String
cuentaCredito,double importe)
     {
          Connection cnx=null;
          try
          {

     Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
     cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;
database Name=banco; user=sa;password=;");
             cnx.setAutoCommit(false);
             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);
             cnx.commit();
          }
          catch (Exception e)
          {
             try {
                  cnx.rollback();
             }
             catch (SQLException e1)
             {
                  e1.printStackTrace();
             }
             e.printStackTrace();
          }
     }
```

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 correctamente, 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.

JDBC prevé varios niveles de aislamiento. Esos niveles de aislamiento 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 que se acceden 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 éstos pueden estar bloqueados y por lo tanto ser inaccesibles para las demás aplicaciones.

Archivos Java

1. Presentación

Un archivo 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 sólo 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.
- Esta ventaja es aún más notable para los applets ya que el navegador puede así recuperarlos con una sola petición http.
- La seguridad se mejora también mediante la firma y el sellado del archivo.
- El formato de los archivos, aún siendo estándar, no presenta ninguna restricción respecto a un sistema específico.

2. Manipulación de un archivo

La manipulación de un archivo Java (archivo `jar`) 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, los archivos Java pueden ser procesados por herramientas destinadas a la manipulación de ficheros `ZIP`.

El comando estándar de manipulación de archivo es el comando `jar`. Forma parte de las herramientas proporcionadas con el jdk.

a. Creación de un archivo

La sintaxis básica de creación de un archivo Java es la siguiente:

```
jar cf nombreDelArchivo listaArchivos
```

El parámetro `c` se destina por supuesto a indicar al comando `jar` que deseamos crear un archivo. El parámetro `f`, por su parte, indica que el comando debe generar un archivo. 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 o los archivos que se deben incluir en el 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 dentro del archivo. Las opciones siguientes también están disponibles.

- **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.

b. Visualización del contenido

El contenido de un archivo se puede visualizar con el comando siguiente:

```
jar tf pizarra.jar
```

El comando muestra en la consola el contenido del archivo.

```
META-INF/MANIFEST.MF
.classpath
.project
Cliente$1.class
Cliente$2.class
Cliente$3.class
Cliente$4.class
Cliente$5.class
Cliente$6.class
Cliente.class
ClientePizarraMágica.class
PanelDibujo.class
ThreadEntrada.class
```

Se puede obtener información extra al añadir la opción `v` al comando. Datos como la fecha de modificación y el tamaño de los archivos se incorporan al resultado del comando.

```
jar tvf pizarra.jar

    59 Mon Feb 14 11:34:38 CET 2005 META-INF/MANIFEST.MF
   247 Tue Feb 08 19:07:22 CET 2005 .classpath
   568 Tue Feb 08 19:07:22 CET 2005 .project
```

```
 1050 Mon Feb 14 08:35:58 CET 2005 Cliente$1.class
 1527 Mon Feb 14 08:35:58 CET 2005 Cliente$2.class
 3091 Mon Feb 14 08:35:58 CET 2005 Cliente$3.class
 1023 Mon Feb 14 08:35:58 CET 2005 Cliente$4.class
 1077 Mon Feb 14 08:35:58 CET 2005 Cliente$5.class
 1182 Mon Feb 14 08:35:58 CET 2005 Cliente$6.class
 6731 Mon Feb 14 08:35:58 CET 2005 Cliente.class
  530 Mon Feb 14 08:34:18 CET 2005 ClientePizarraMagica.class
 5585 Mon Feb 14 08:36:10 CET 2005 PanelDibujo.class
 3146 Mon Feb 14 08:39:36 CET 2005 ThreadEntrada.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.

c. Extracción

Es posible extraer archivos mediante el comando siguiente:

```
jar xvf pizarra.jar
```

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 sobreescriben 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 `ClientePizarraMagica.class`.

```
jar xvf pizarra.jar ClientePizarraMagica.class
```

El contenido del archivo no se modifica con este comando.

d. 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 pizarra.jar connect.gif
```

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.

e. Ejecución

Una aplicación presente en un archivo Java puede ejecutarse directamente desde el archivo sin necesitar extraer su contenido. Debemos indicar a la máquina virtual 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 pizarra.jar
```

Sin embargo la máquina virtual Java necesita información adicional para determinar qué clase, dentro del archivo, contiene el método `main` por el cual debe empezar la ejecución de la aplicación. Para ello, busca el manifest del archivo, que debe contener, efectivamente, esta información para que la aplicación se pueda ejecutar a partir del archivo.

3. El manifest

Los archivos Java son mucho más que meros archivos comprimidos ya que proporcionan una multitud de funcionalidades complementarias:

- Ejecución directa desde el archivo.
- Firma del contenido del archivo.
- Sellado de partes del archivo.
- Gestión de las versiones.

Todas estas funcionalidades están disponibles mediante el archivo 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 : (dos puntos).

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: 1.8.0-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 pizarra.jar infos.txt *
```

Este comando genera un archivo llamado **pizarra.jar** que contiene todos los archivos de la carpeta actual y añade al manifest por defecto los datos contenidos en el archivo **infos.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. El archivo **infos.txt** contiene en nuestro caso la línea siguiente:

```
Main-Class: ClientePizarraMagica
```

No olvide el retorno de carro al final de la línea y el espacio después del carácter : (dos puntos).

El archivo se genera con el archivo manifest siguiente:

```
Manifest-Version: 1.0
Created-By: 1.6.0 (Sun Microsystems Inc.)
Main-Class: ClientePizarraMagica
```

La versión del comando `jar` proporcionada con el jdk 8 propone también la opción `e` que permite indicar el punto de entrada en la aplicación sin tener que crear un archivo intermedio. La sintaxis puede ser por lo tanto la siguiente:

```
jar cvfe pizarra.jar ClientePizarraMagica.class *
```

4. Empaquetar y firmar un archivo

Estas dos operaciones le van a permitir reforzar la seguridad de una aplicación. El empaquetado va a garantizar que todas las clases de un paquete provienen de un único archivo jar.

La firma del archivo permite garantizar el origen del archivo jar.

a. Empaquetado

Tras dar los primeros pasos en programación orientada a objetos, hemos visto que es posible restringir la visibilidad de los miembros de una clase. La palabra clave `protected` permite limitar la visibilidad de un elemento a la clase en la que se ha definido, a las subclases de la misma y a otras clases que formen parte del mismo `package`.

En el siguiente ejemplo, el método `calculaEdad` se declara `protected` y, por tanto, está accesible desde las demás clases del paquete es.eni.pk1. Es invisible, por el contrario, para los demás paquetes como, por ejemplo, el paquete es.pirata.eni.

```
package es.eni.pk1;

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Persona
{
    private String apellido;
    private String nombre;
    private LocalDate fecha_naci=LocalDate.of(1963,11,29);

    public Persona()
    {
    }

    public Persona(String a,String n,LocalDate f)
    {
        this.apellido=a;
        this.nombre=n;
        this.fecha_naci=f;
    }

    public String getApellido()
    {
       return apellido;
    }

    public void setApellido(String apellido)
    {
       this.apellido = apellido;
    }

    public String getNombre()
    {
       return nombre;
    }

    public void setNombre(String nombre)
    {
       this.nombre = nombre;
    }

    public LocalDate getFecha_naci()
    {
       return fecha_naci;
    }

    public void setFecha_naci(LocalDate fecha_naci)
    {
       this.fecha_naci = fecha_naci;
```

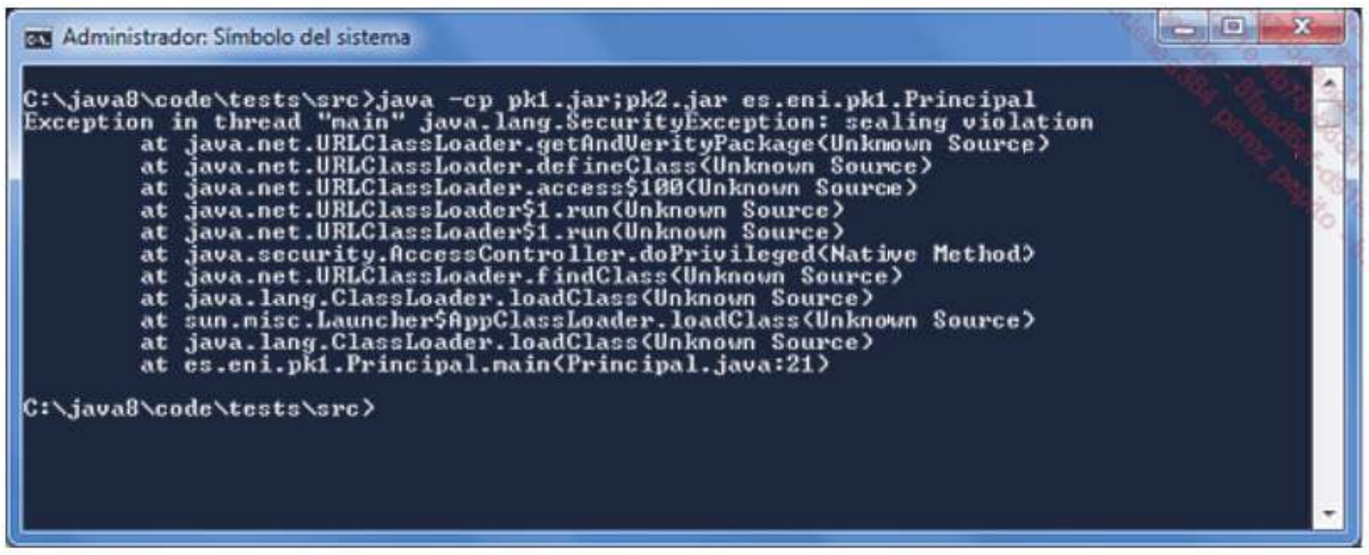
```
    }

    protected long calculaEdad()
    {
       return fecha_naci.until(LocalDate.now(),ChronoUnit.YEARS);
    }
}
```

La compilación de la clase siguiente, definida en otro paquete, provoca un error.

```
package es.pirata.eni;
import es.eni.pk1.*;

public class Principal
{
    public static void main(String[] args)
    {
        Persona p;
        p=new Persona ();
        p.calculaEdad();
    }
}
```

Por el contrario, si utilizamos el mismo nombre de paquete que el que se ha utilizado en la clase `Persona`, no existe ningún problema.

Un usuario que tenga a su disposición el archivo jar puede, fácilmente, tener acceso a los elementos declarados como `protected` en las clases. Para evitar este problema, es posible empaquetar el archivo jar que contiene la clase `Persona`. De hecho, no se empaqueta el archivo jar sino los packages que lo contienen. Realizando esta operación, indicamos a la máquina virtual de Java que va a utilizar el archivo que todas las clases del package empaquetado están en el archivo. Si la máquina virtual encuentra en otro archivo una clase que forme parte del mismo package, se produce una excepción. Para ilustrarlo, el archivo pk1.jar contiene la clase `es.eni.pk1.Persona` y el archivo pk2.jar contiene la clase `es.eni.pk1.Principal`. Hay, por tanto, dos clases que pertenecen al mismo paquete en dos archivos jar. El package `es.eni.pk1` se ha empaquetado en el archivo pk1.jar.



Para empaquetar un package, es preciso agregar al archivo `manifest` del archivo la siguiente información.

```
Name: nombre del package a empaquetar
Sealed: true
```

Esta información debe agregarse en el momento de crear el archivo. Es preciso, para ello, crear un archivo de texto que contenga la información que se quiere agregar al manifiesto del archivo.

Ejemplo de archivo (proteccion.txt):

```
Name: es/eni/pk1/
Sealed: true
```

Debe utilizar el carácter / para separar los términos del nombre del package y no olvidar el / final.

A continuación, se utiliza este archivo en la línea de comandos de creación del archivo.

```
jar -cvfm pk1.jar ./proteccion.txt es/eni/pk1
```

b. Firma

Cuando envía un documento, en papel, a su correspondencia, generalmente incluye su firma en dicho documento para garantizar que es, efectivamente, usted el autor del documento. Cuando el destinatario recibe el documento puede, gracias a su firma, verificar que efectivamente usted es el autor. La firma de un archivo jar utiliza el mismo principio, cuyo objetivo es permitir al destinatario verificar la identidad del autor del mismo. Para que este mecanismo sea eficaz, es imprescindible confiar en la firma del documento. Imagine que una administración le solicita una copia certificada conforme de su carné de identidad. Usted realiza una fotocopia del documento y agrega la mención "conforme al original" y, a continuación, su firma. Hay muchas posibilidades de que la administración rechace este documento, pues usted es a la vez el autor y el garante de la conformidad. Por el contrario, si tras haber realizado una copia del documento, se dirige a un organismo oficial (un ayuntamiento, una comisaría de policía, etc.) y dicho organismo valida la autenticidad del documento incluyendo un sello, el destinatario no puede tener ninguna duda. En la firma de un archivo jar este mecanismo se implementa mediante un certificado. Este certificado jugará el rol del sello del ayuntamiento o de la comisaría, que valida cualquier documento en papel.

Es preciso, también, garantizar que el documento no pueda modificarse tras su autenticación o, si lo hace, que dicha modificación pueda detectarse fácilmente. Por lo general, redacta el documento con un bolígrafo en lugar de con lápiz para evitar que alguien pueda utilizar un borrador y realice alguna modificación en el documento. En un archivo jar firmado se calcula e incluye una firma SHA de cada archivo y, a continuación, se almacena en el manifiesto del archivo. El usuario del archivo realizará el mismo cálculo de la firma y verificará que el resultado está conforme a lo indicado en el manifiesto.

Para poder firmar un archivo jar, necesita adaptar el "material":

- la herramienta que nos va a permitir poner el sello en el archivo. Es la herramienta `jarsigner` del jdk que se utiliza.
- la barra de cera para crear los sellos. Existen dos soluciones para implementar este elemento. Comprarlo en una "tienda especializada" o fabricarlo nosotros mismo. Las tiendas se llaman entidades de certificación. Pueden proveer todos los elementos necesarios para realizar la firma de un archivo jar. Tendrá a su disposición una clave privada, que sirve para firmar realmente el archivo, y una clave pública, que permite verificar la autenticidad de la firma. Esta clave pública la certifica la entidad de certificación para garantizar que se le ha atribuido correctamente.

Si no quiere adquirir estos elementos, puede generarlos usted mismo gracias a la herramienta `keytool` del jdk.

La siguiente línea de comandos ejecuta dicha herramienta para crear los elementos.

```
keytool -genkey -alias java8 -keystore certifs
```

El parámetro `-alias` indica el nombre asociado a la clave. El parámetro `-keystore` indica el nombre del archivo donde se almacenarán los elementos generados. Se le realizarán preguntas suplementarias para que la herramienta pueda obtener la información que necesita.

```
Escriba la contraseña del archivo de claves:
Vuelva a escribir la contraseña:
¿Cuál es su apellido y nombre?
  [Unknown]:  ramos francisco
¿Cuál es el nombre de su unidad operacional?
  [Unknown]:  eni
¿Cuál es el nombre de su empresa?
  [Unknown]:  ediciones eni
¿Cuál es el nombre de su ciudad de residencia?
  [Unknown]:  alcobendas
¿Cuál es el nombre de su estado o provincia?
  [Unknown]:  madrid
¿Cuál es el código de dos letras del país para esta unidad?
  [Unknown]:  es
¿Es correcto CN=ramos francisco, OU=eni, O=ediciones eni,
L=alcobendas, ST=madrid, C=es?
  [no]:  si

Escriba la contraseña de la clave para <java8>
        (presione Enter si se trata de la contraseña del archivo
de claves) :
Vuelva a escribir la contraseña:
```

Ahora que tenemos los elementos necesarios, podemos firmar nuestro archivo. Es preciso haberlo generado previamente.

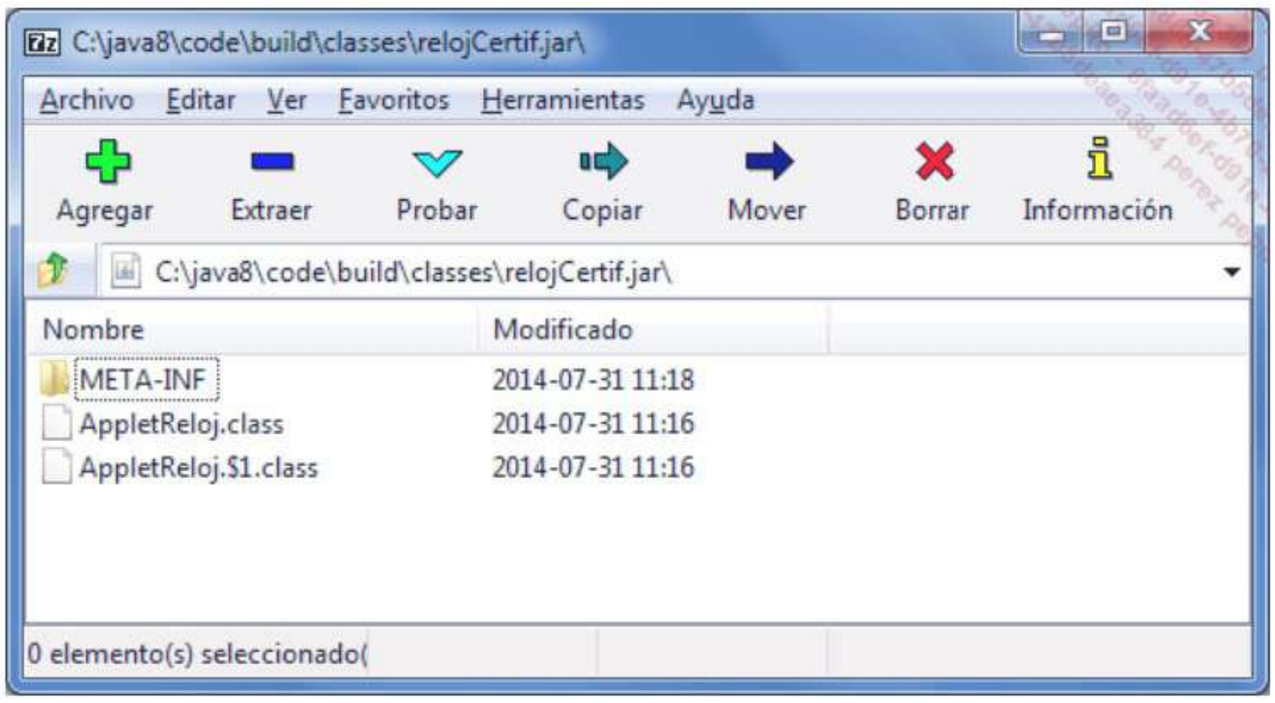
La línea de comandos que permite ejecutar la herramienta `jarsigner` es relativamente larga. Se muestra a continuación (por motivos de legibilidad, los parámetros se ubican en varias líneas, aunque deben introducirse en una sola línea separándolos con espacios).

```
jarsigner -keystore certifs
          -storepass password
          -keypass password
          -tsa http://tsa.safecreative.org
          -signedjar relojCertif.jar
           reloj.jar
           java8
```

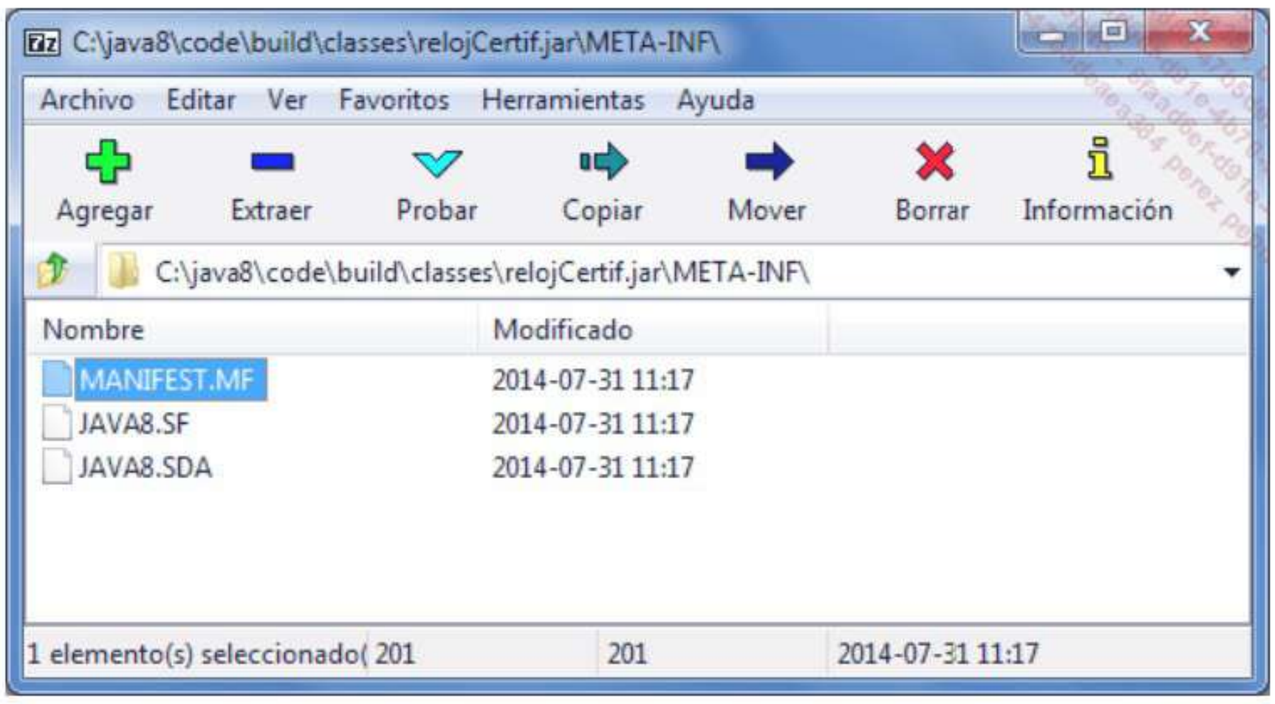
He aquí el rol de cada parámetro:

- -keystore: indica el nombre del archivo que contiene las claves que se utilizarán para la firma.
- -storepass: contraseña para acceder al contenido del archivo.
- -keypass: contraseña asociada a las claves.
- -tsa: URL de un servicio que provea un Time Stamp, utilizado para firmar el archivo.
- -signedjar: nombre del archivo jar firmado.
- el nombre del archivo a firmar.
- el nombre del firmante.

A continuación, podemos verificar el contenido del archivo jar generado con una herramienta como 7Zip, por ejemplo:



Tenemos nuestros archivos compilados y la carpeta `META-INF` en la que se encuentra el manifiesto, así como un archivo `.SF` y un archivo `.DSA`.

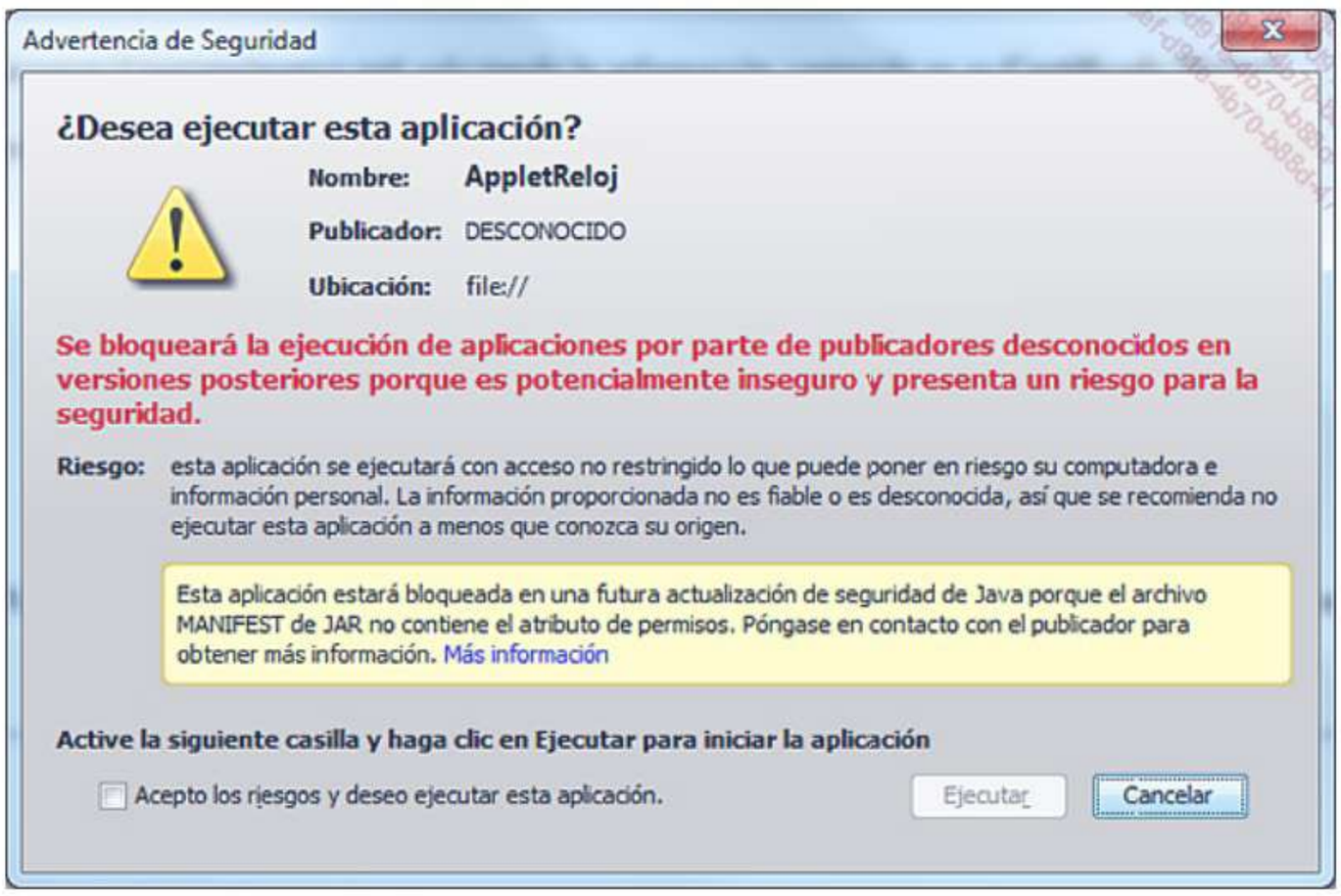


El archivo .DSA contiene la firma del archivo generada con la clave privada de la firma así como el certificado que contiene la clave pública para verificar el archivo. El contenido de dicho archivo es, sin duda, incomprensible para un humano.

El archivo.SF contiene la huella SHA de los distintos archivos. Esta información está, también, presente en el manifiesto.

```
Manifest-Version: 1.0
Created-By: 1.8.0-ea (Oracle Corporation)

Name: AppletReloj$1.class
SHA-256-Digest: X4tyEOAePgqTqGxA9nnvWBmwmS8Iyyg0whulzEYWzS0=

Name: AppletReloj.class
SHA-256-Digest: 3517G6dfPZ3SYq4aAVP6UmxlDuyhtP0pQLfpwq+0qYM=
```

Ahora que se ha firmado el archivo, podemos verificar la reacción de un navegador cuando ejecuta un applet contenido en el archivo. Vemos un cuadro de diálogo similar al que se muestra aquí.



El navegador detecta, siempre, un riesgo potencial con el applet. Detecta, efectivamente, que el applet está firmado aunque es incapaz de validar el nombre del fabricante de la aplicación. El problema proviene, de hecho, del certificado utilizado para firmar el archivo. Al no haberlo generado una entidad de certificación reconocida, el navegador no le otorga ninguna confianza. Para eliminar esta advertencia es obligatorio utilizar un certificado

Java Web Start

1. Presentación

La tecnología Java Web Start permite arrancar la ejecución de una aplicación con un simple clic en un vínculo sin ninguna instalación previa de la aplicación. El vínculo apunta a un fichero JNLP (*Java Network Launching Protocol*) que contiene la información necesaria para que Java Web Start descargue y ejecute la aplicación.

Dado que las aplicaciones Java pueden ejecutarse en cualquier plataforma, no es necesario que coincida con la de su creador. Tras la primera ejecución, la aplicación se almacena en una caché local del puesto cliente. Esta técnica mejora la velocidad de las ejecuciones posteriores y permite también la ejecución de la aplicación incluso si no está disponible ninguna conexión de red. Si una aplicación necesita una versión particular de la plataforma Java y ésta no está disponible en el puesto, Java Web Start descarga e instala automáticamente la versión correcta. También se tiene en cuenta la seguridad ya que Java Web Start sólo autoriza accesos limitados a los recursos de disco y de red que puede utilizar la aplicación.

2. Ejecución de una aplicación

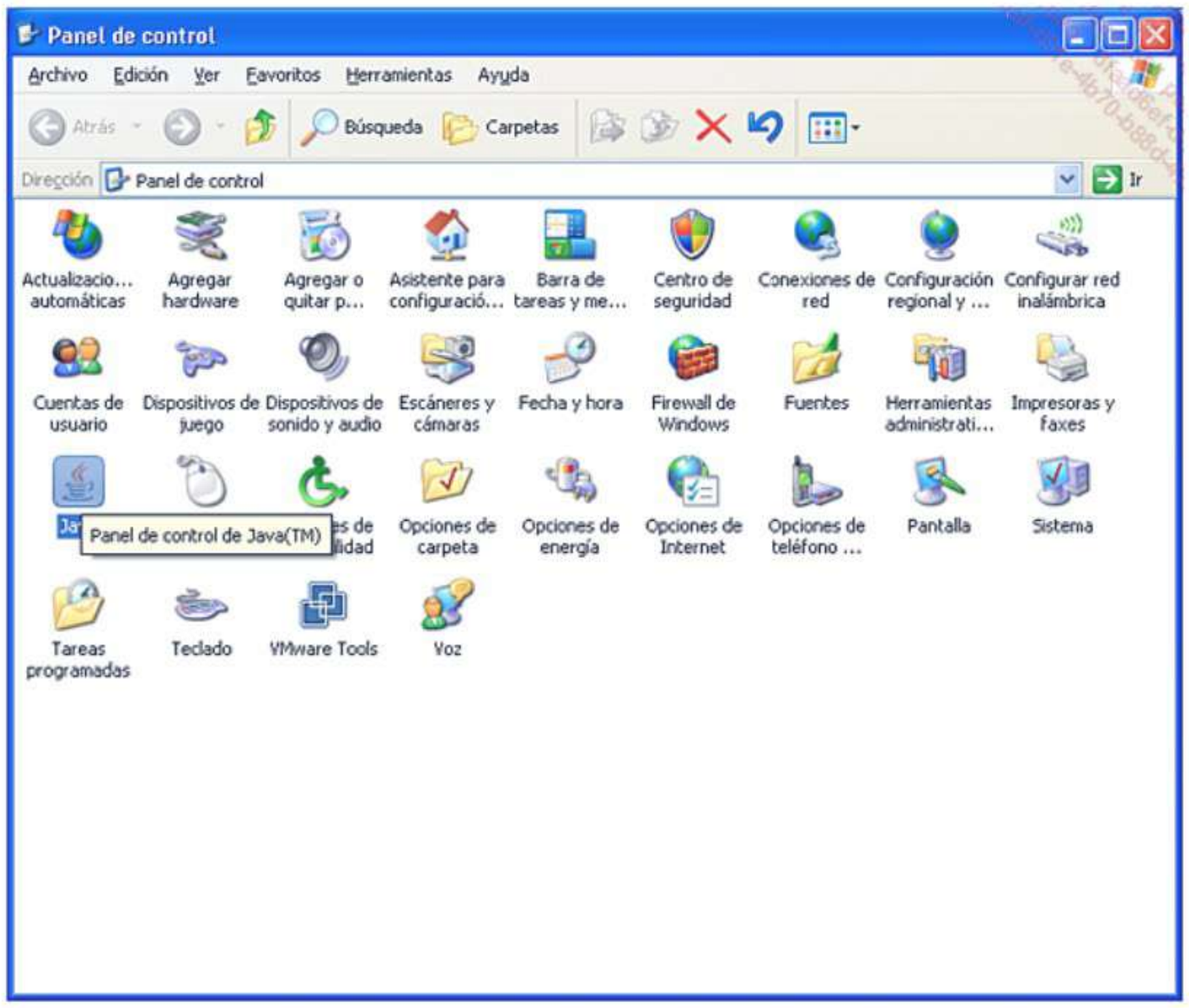
Hay dos soluciones posibles para la ejecución de una aplicación con Java Web Start.

a. Desde un navegador

El lanzamiento de la aplicación se efectúa sencillamente al hacer clic en un vínculo que apunta al archivo JNLP de la aplicación. El archivo se descarga y luego Java Web Start extrae de él toda la información relativa al funcionamiento de la aplicación.

b. Desde la caché local

Se puede ejecutar también una aplicación desde archivos puestos en caché durante una ejecución anterior. Debemos utilizar el visualizador de caché Java para visualizar la lista de las aplicaciones que ya se han ejecutado. El visualizador de caché está disponible mediante el icono Java del Panel de control.



Seleccionamos luego el botón **Ver** de la sección **Archivos Temporales de Internet** de la ficha **General**.

Java Web Start

1. Presentación

La tecnología Java Web Start permite arrancar la ejecución de una aplicación con un simple clic en un vínculo sin ninguna instalación previa de la aplicación. El vínculo apunta a un fichero JNLP (*Java Network Launching Protocol*) que contiene la información necesaria para que Java Web Start descargue y ejecute la aplicación.

Dado que las aplicaciones Java pueden ejecutarse en cualquier plataforma, no es necesario que coincida con la de su creador. Tras la primera ejecución, la aplicación se almacena en una caché local del puesto cliente. Esta técnica mejora la velocidad de las ejecuciones posteriores y permite también la ejecución de la aplicación incluso si no está disponible ninguna conexión de red. Si una aplicación necesita una versión particular de la plataforma Java y ésta no está disponible en el puesto, Java Web Start descarga e instala automáticamente la versión correcta. También se tiene en cuenta la seguridad ya que Java Web Start sólo autoriza accesos limitados a los recursos de disco y de red que puede utilizar la aplicación.

2. Ejecución de una aplicación

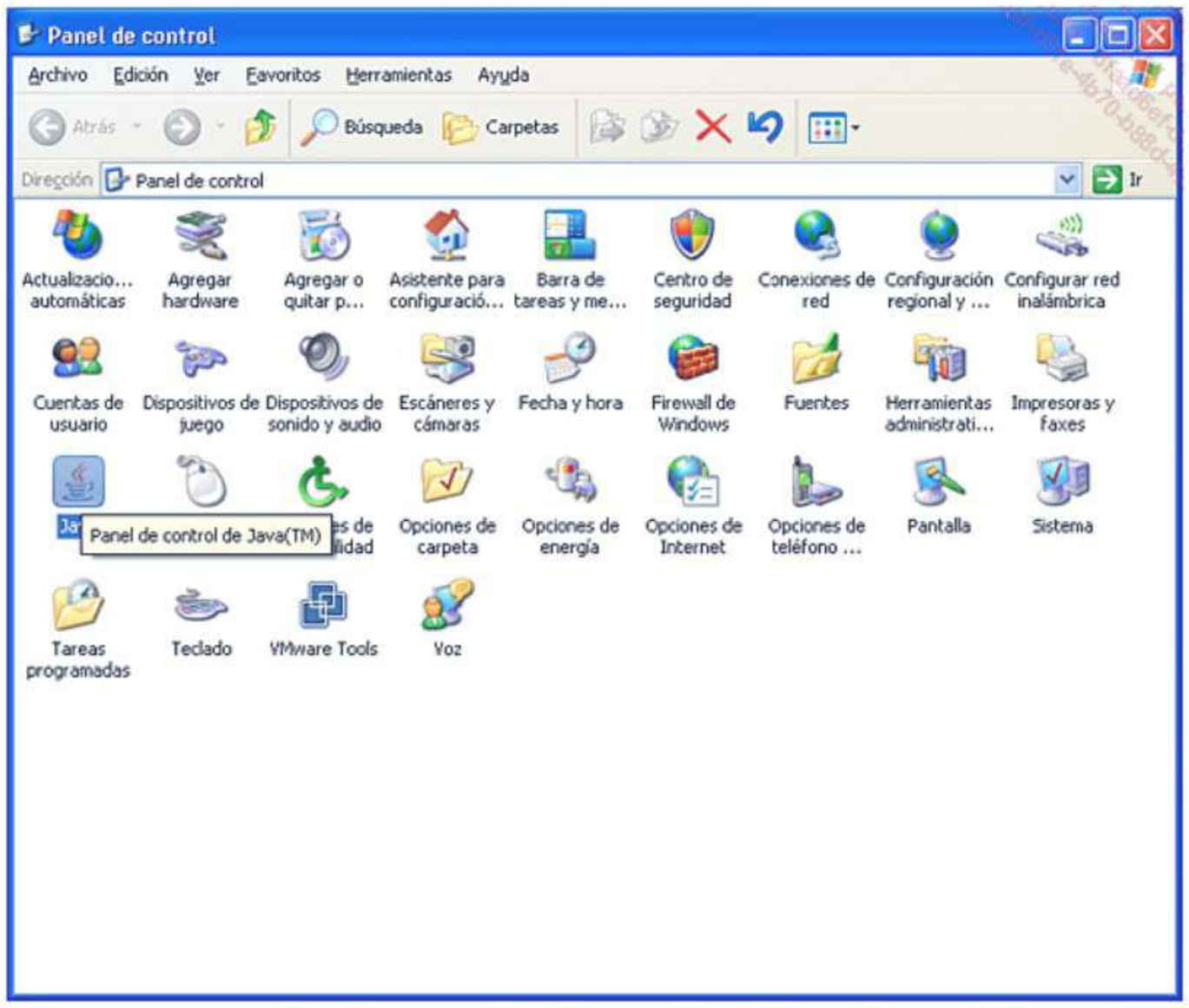
Hay dos soluciones posibles para la ejecución de una aplicación con Java Web Start.

a. Desde un navegador

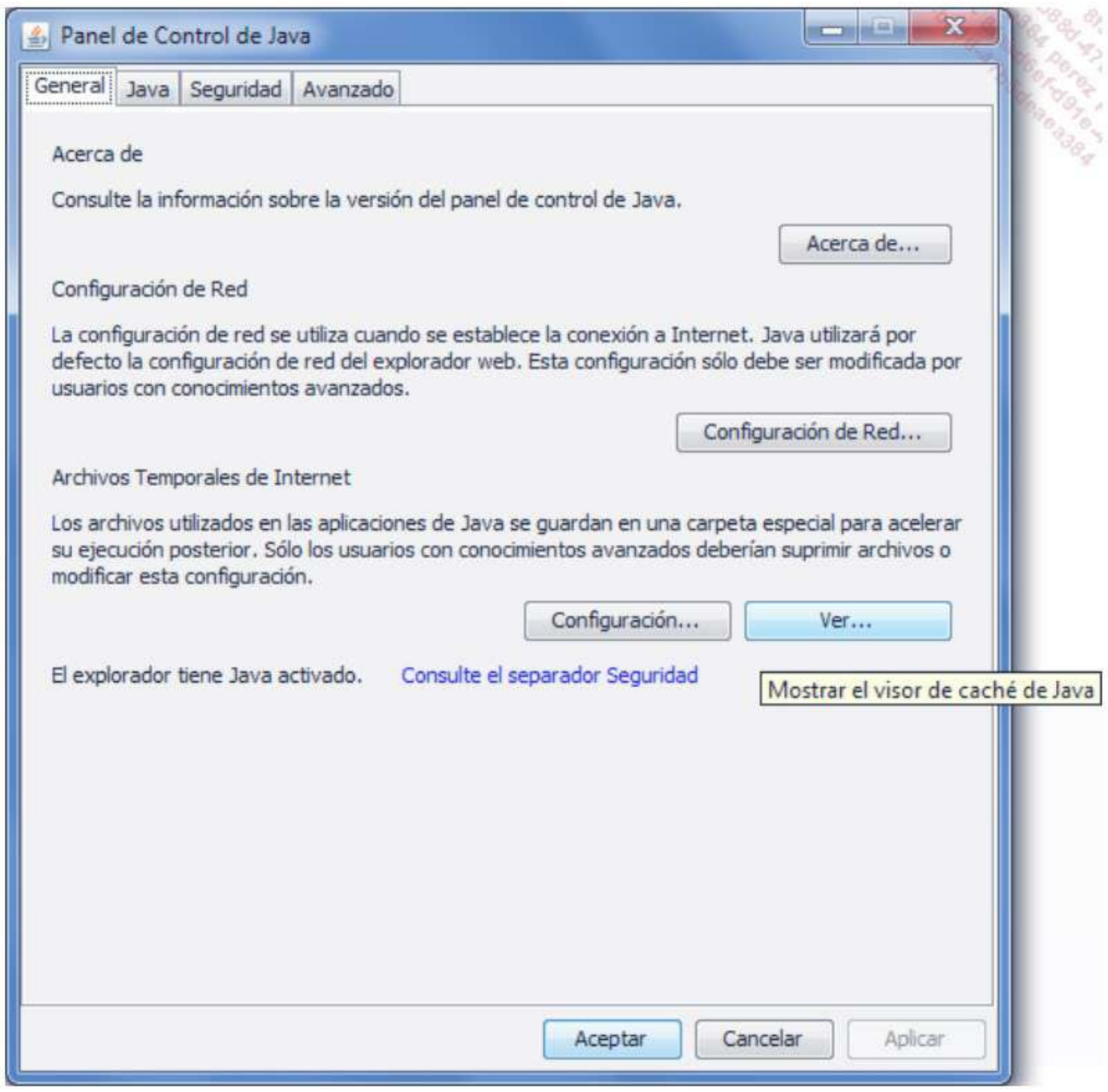
El lanzamiento de la aplicación se efectúa sencillamente al hacer clic en un vínculo que apunta al archivo JNLP de la aplicación. El archivo se descarga y luego Java Web Start extrae de él toda la información relativa al funcionamiento de la aplicación.

b. Desde la caché local

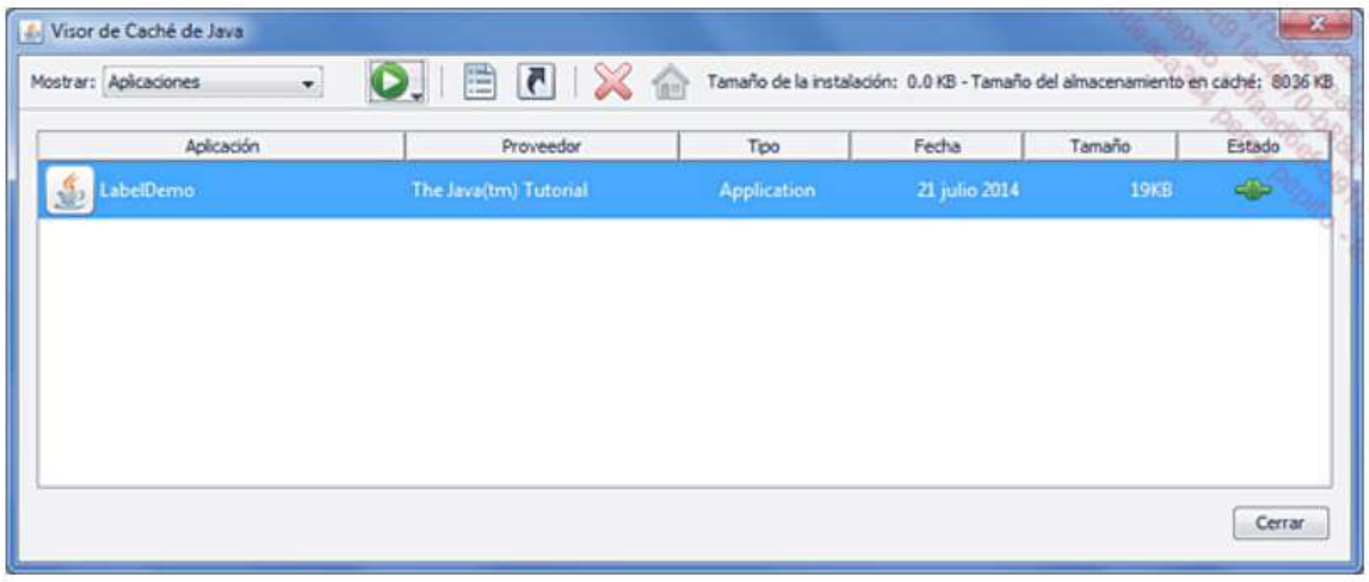
Se puede ejecutar también una aplicación desde archivos puestos en caché durante una ejecución anterior. Debemos utilizar el visualizador de caché Java para visualizar la lista de las aplicaciones que ya se han ejecutado. El visualizador de caché está disponible mediante el icono Java del Panel de control.



Seleccionamos luego el botón **Ver** de la sección **Archivos Temporales de Internet** de la ficha **General**.



El visualizador de la caché Java muestra la lista de las aplicaciones que ya se han utilizado.



Luego, hay varias opciones disponibles mediante los botones siguientes de la barra de herramientas:

Volver a ejecutar la aplicación seleccionada ya sea a partir del servidor (en línea), o a partir de la caché Java (sin conexión).

Muestra el archivo JNLP de la aplicación.

Añade un acceso directo en el escritorio que permite lanzar la aplicación como una aplicación clásica.

Suprime la aplicación de la lista.

Muestra la página de inicio de la aplicación en el navegador.

3. Despliegue de una aplicación

El despliegue de una aplicación con Java Web Start se divide en cuatro operaciones:

- Configurar el servidor Web.
- Crear el archivo JNLP.

- Ubicar la aplicación en el servidor Web.
- Crear la página Web de Inicio.

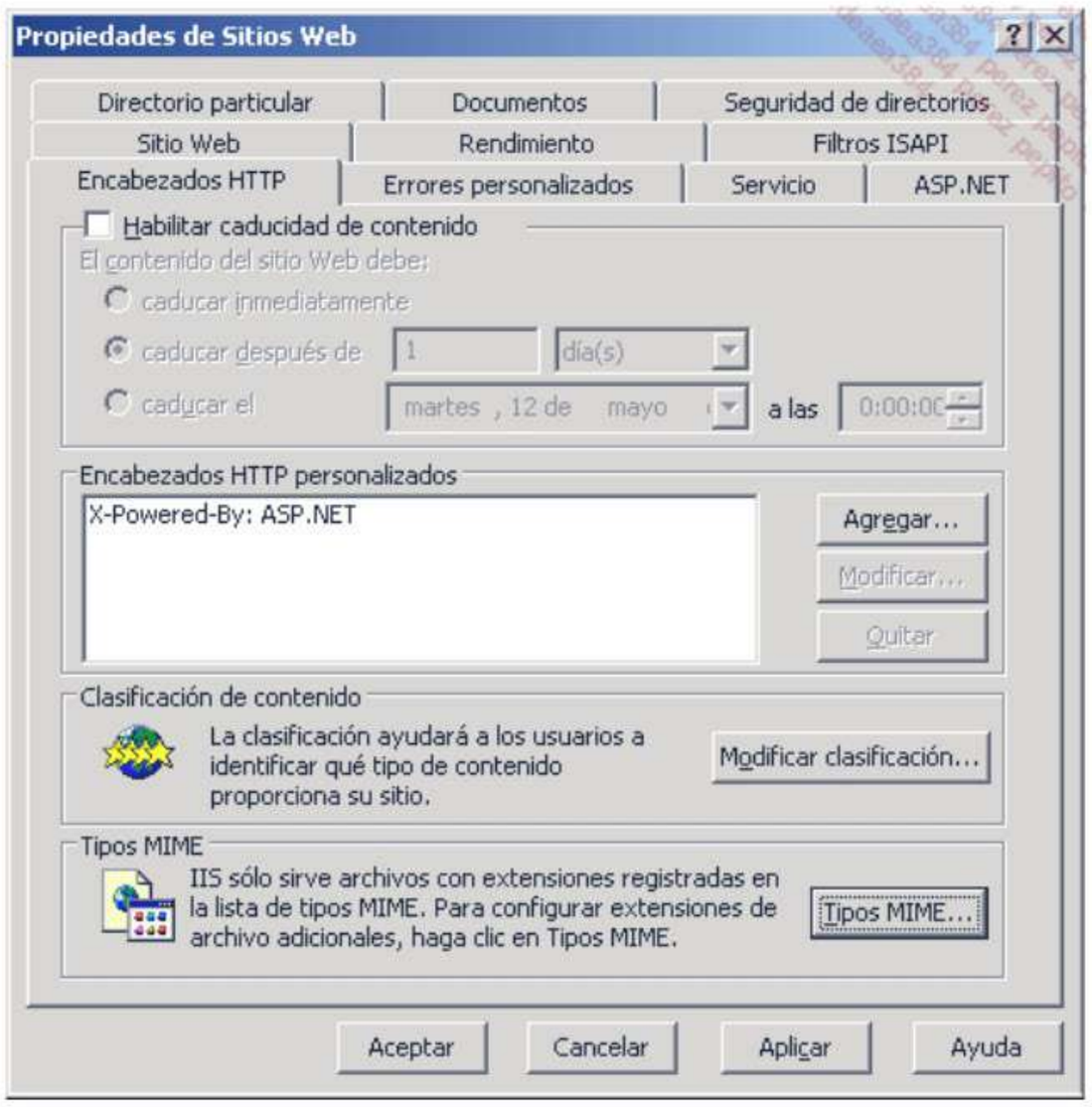
Vamos a describir en detalles cada una de estas etapas.

a. Configuración del servidor Web

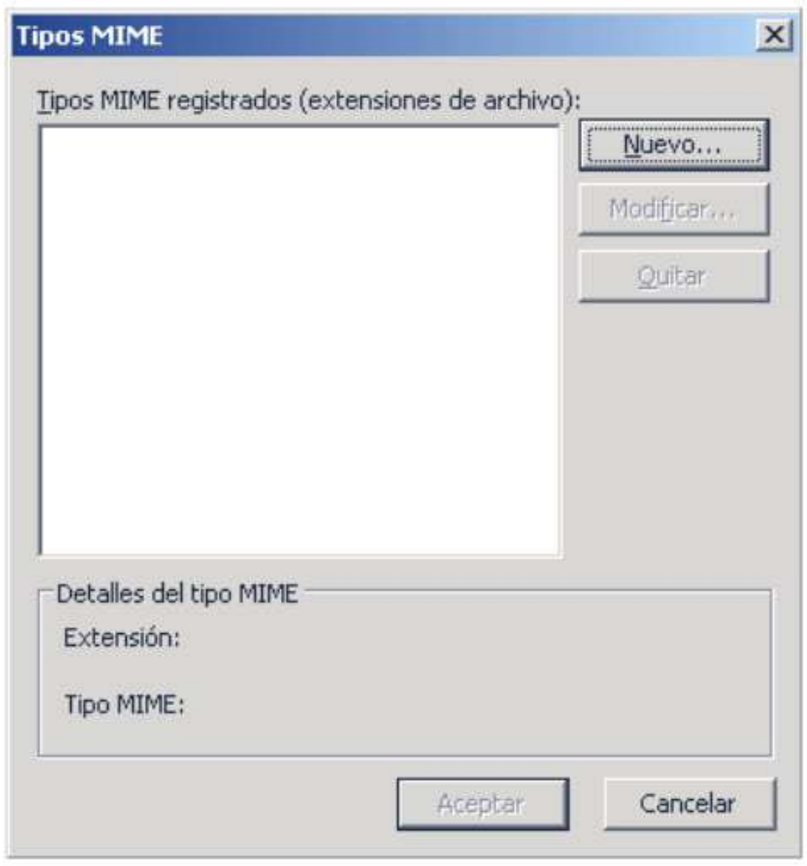
La única modificación necesaria en el servidor Web consiste en configurar el tipo MIME asociado a la extensión de archivo `.jnlp`. Por supuesto, esta configuración es propia de cada tipo de servidor. Para un servidor Apache, sencillamente, basta con añadir al archivo `mime.tipos` la línea siguiente:

```
aplicación/x-java-jnlp-file JNLP
```

Para un servidor IIS, se debe utilizar la ficha **Encabezados HTTP** de la página de propiedades del servidor Web.



El botón **Tipos MIME** permite acceder al cuadro de diálogo de gestión de los tipos MIME reconocidos por el servidor Web.



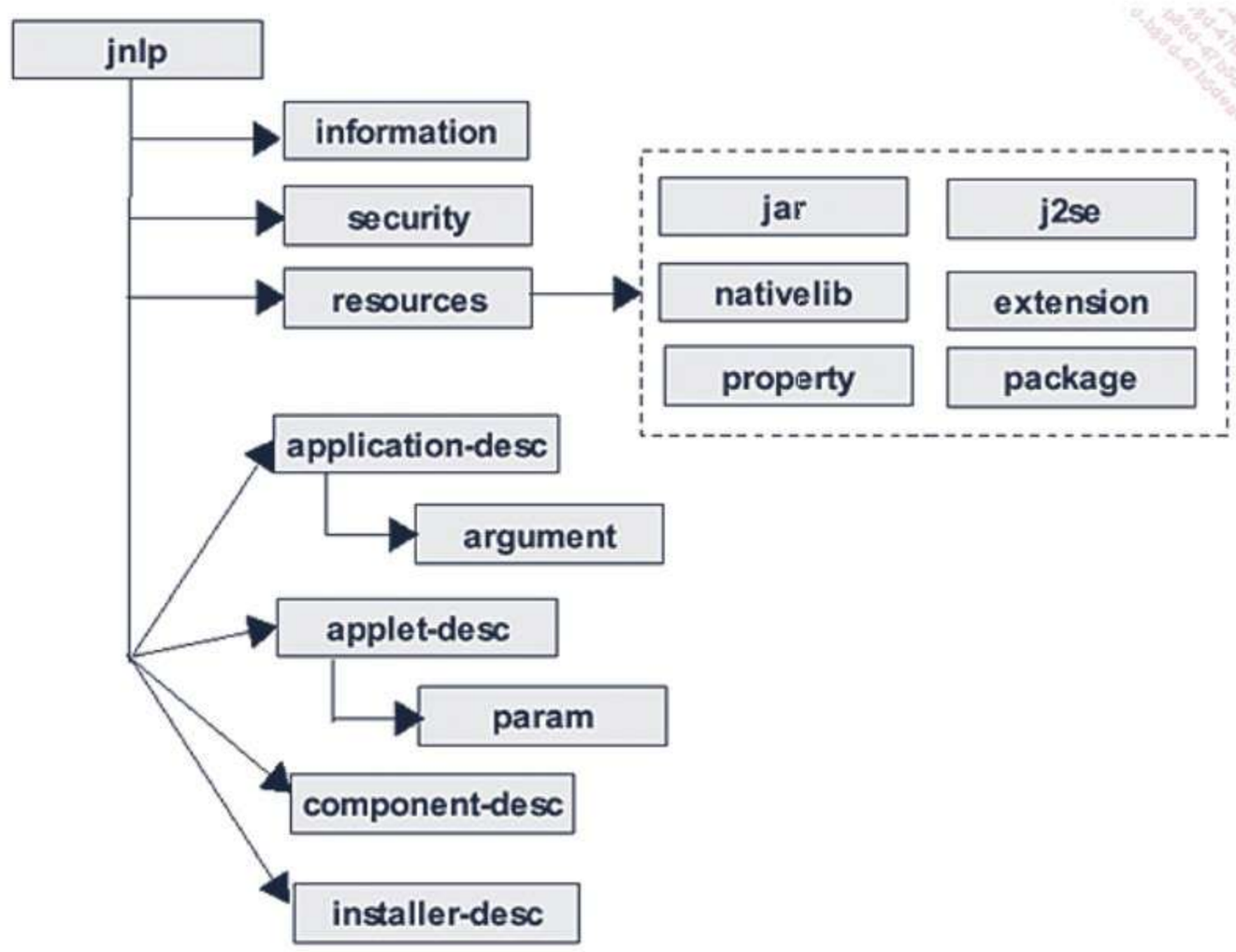
El botón **Nuevo** muestra un cuadro de diálogo que permite introducir información relativa al tipo MIME que se debe añadir al servidor.

A menudo suele ser preferible arrancar de nuevo el servidor para que actualice las modificaciones de configuración.

b. Creación del archivo JNLP

El archivo JNLP es el elemento principal del despliegue con Java Web Start. Este archivo con formato xml contiene toda la información necesaria para la ejecución de la aplicación. El formato de este archivo debe respetar los JSR-56 (*Java Specification Requests*).

El diagrama siguiente representa el formato esperado para este archivo.



El elemento **jnlp** es el elemento raíz del archivo. Sus atributos describen las propiedades del archivo jnlp.

El elemento **information** se utiliza para proporcionar la información relativa a la aplicación. Se utilizará durante la instalación de la aplicación.

El elemento **security** se utiliza para obtener un entorno de seguridad durante la ejecución de la aplicación.

El elemento **resources** indica cuales son los recursos que forman parte de la aplicación.

El archivo **jnlp** termina con un elemento **application-desc**, **applet-desc**, **component-desc** o **installer-desc** según el tipo de aplicación que se deba desplegar. Sólo uno de estos elementos debe estar presente en el archivo.

He aquí un ejemplo de archivo jnlp:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- archivo de despliegue para la pizarra mágica -->

<jnlp spec="1.0+"
      codebase="http://francisco.eni.es/pizarraMagica"
      href="pizarra.jnlp">
   <information>
      <title>pizarra mágica</title>
      <vendor>ramos francisco</vendor>
      <homepage
href="http://francisco.eni.es/pizarraMagica/install.html"/>
      <description kind="short">esta aplicación permite compartir
un espacio de dibujo entre varios usuarios</description>
      <offline-allowed/>
   </information>
   <resources>
        <jar href="pizarra.jar"/>
     <j2se versión="1.6+"
           href="http://java.sun.com/products/autodl/j2se"/>
   </resources>
   <application-desc main-class="ClientePizarraMagica"/>
</jnlp>
```

Vamos a ver con detalle la información incluida en este archivo.

`<?xml version="1.0" encoding="utf-8"?>`:

Esta línea indica que se trata de un documento conforme al estándar xml 1.0 y que la codificación de los caracteres utilizada es la utf-8.

`<!-- archivo de despliegue para la pizarra mágica -->`:

Línea de comentarios en un documento xml.

```
<jnlp spec="1.0+"
           codebase="http://francisco.eni.es/pizarraMagica"
            href="pizarra.jnlp">:
```

Etiqueta raíz del documento jnlp.

El atributo `spec` indica la versión del protocolo jnlp que debe aceptar el cliente para que la instalación sea posible. En nuestro caso, el cliente debe aceptar la versión 1.0 o posterior. Así cualquier cliente podrá instalar la aplicación.

El atributo `codebase` indica la ubicación raíz de los demás documentos referenciados en el archivo jnlp mediante atributos `href`.

El atributo `href` especifica la url relativa del archivo jnlp. Esta información se combina con el valor del atributo `codebase` para obtener una URL absoluta.

```
<title>pizarra mágica</title>
```

Título de la aplicación, se utiliza para identificarla en el visualizador de la caché Java.

```
<vendor>ramos francisco</vendor>
```

Nombre del proveedor de la aplicación que aparece en el visualizador de la caché Java.

```
<homepage href="http://francisco.eni.es/pizarraMagica/install.html"/>
```

URL de la página de inicio de la aplicación. Esta página puede contener un vínculo al archivo jnlp.

```
<description kind="short">esta aplicación permite compartir un espacio de dibujo entre varios usuarios</description>
```

Texto de descripción corta de la aplicación.

```
<offline-allowed/>
```

Indica que se puede ejecutar la aplicación incluso si ninguna conexión de red está disponible. En este caso, se ejecuta la versión puesta en caché. Si una conexión de red está disponible, Java Web Start comprueba si una versión más reciente de la aplicación está disponible en el servidor. Si es el caso, entonces se ejecuta esta nueva versión. Si no, se ejecuta la versión puesta en caché.

```
<jar href="pizarra.jar"/>
```

Nombre del archivo que contiene la aplicación.

```
<j2se version="1.6+" href="http://java.sun.com/products/ autodl/j2se"/>
```

Versión necesaria del jre para el correcto funcionamiento de la aplicación. El signo + después del número de versión indica que se trata de una versión mínima necesaria. Si existe alguna versión posterior disponible en el puesto cliente, se podrá ejecutar la aplicación. Si no se indica el signo +, Java Web Start exigirá la versión exacta. Si no está disponible en el puesto cliente, el atributo `href` indica desde dónde se puede descargar. En ese caso, Java Web Start propone al usuario efectuar esta descarga.

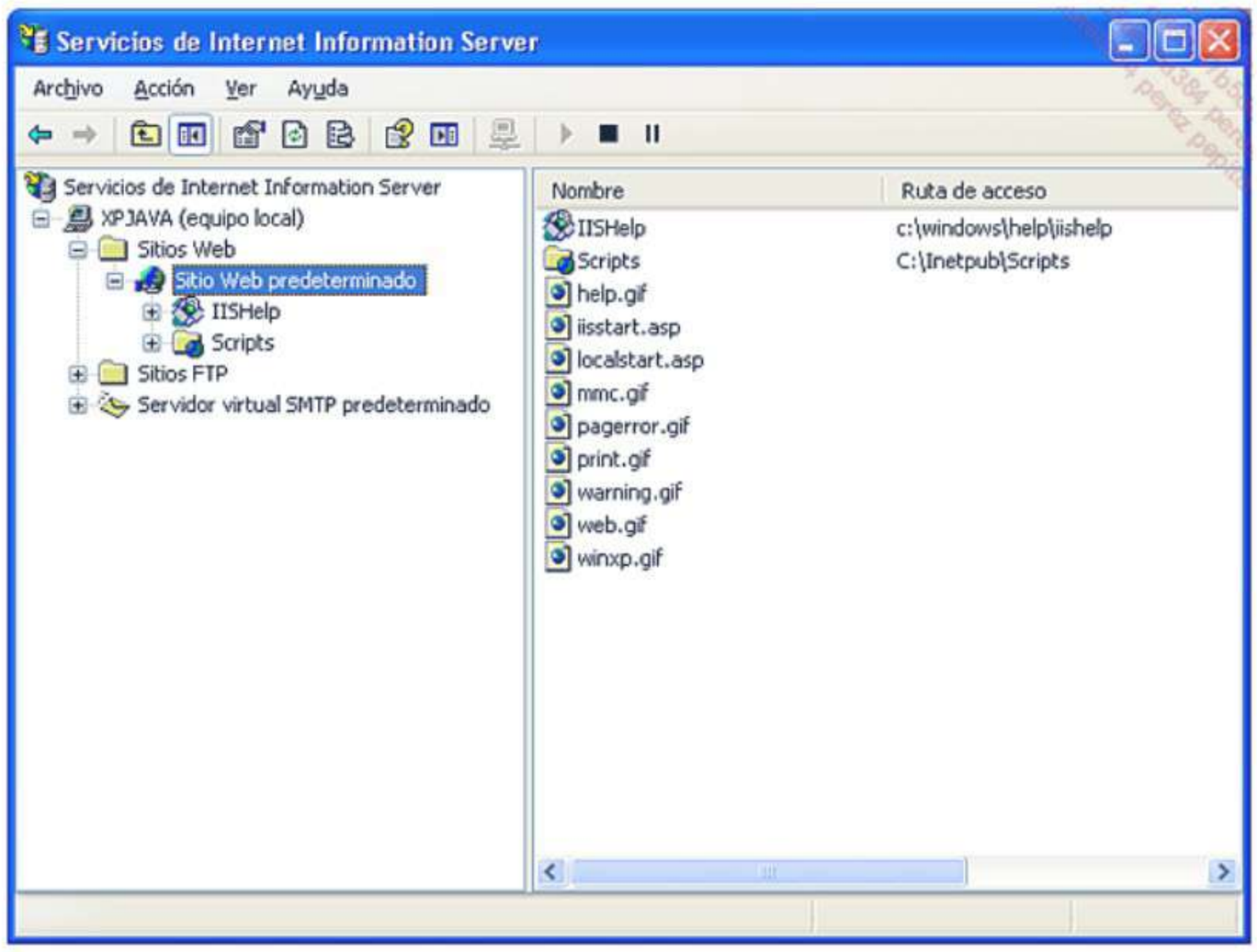
```
<application-desc main-class="ClientePizarraMagica"/>
```

Indica que la aplicación que se desea ejecutar es una aplicación Java autónoma y no un applet. El atributo `main-class` indica el nombre de la clase que contiene el método `main` que permite iniciar la aplicación. Este atributo es opcional si el archivo dispone de un manifest que ya contiene esta información.

c. Desplegar la aplicación en el servidor

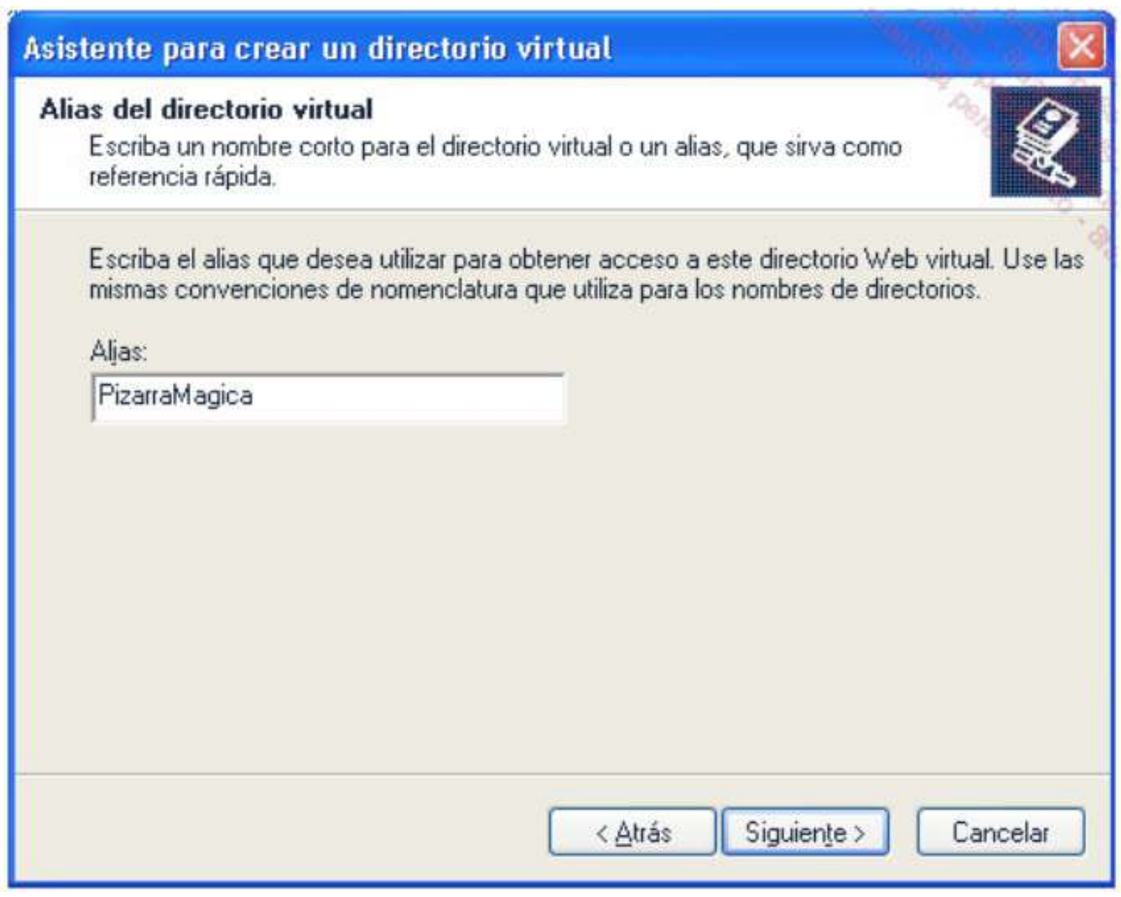
Por supuesto, esta etapa es específica para cada servidor Web. En caso de duda, se aconseja contactar con el administrador del servidor. Aquí se muestra, a título de ejemplo, los pasos que se deben seguir para desplegar la aplicación en un servidor Web IIS de Windows.

Abrimos el Administrador de Internet Information Services con la opción **Herramientas administrativas** del Panel de control. Accedemos, a continuación, al sitio Web por defecto.

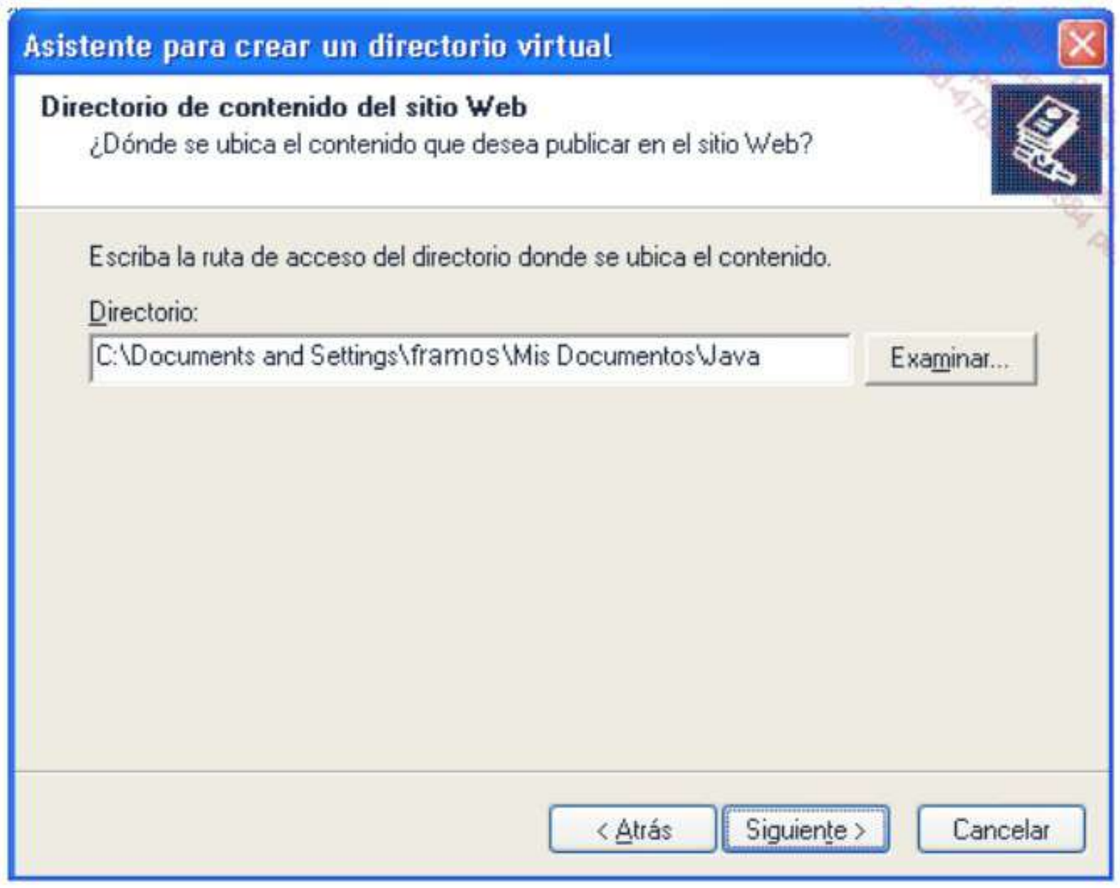


Añadimos, a continuación, un directorio virtual con el menú contextual del sitio Web por defecto. Un asistente nos guiará para la creación del directorio virtual.

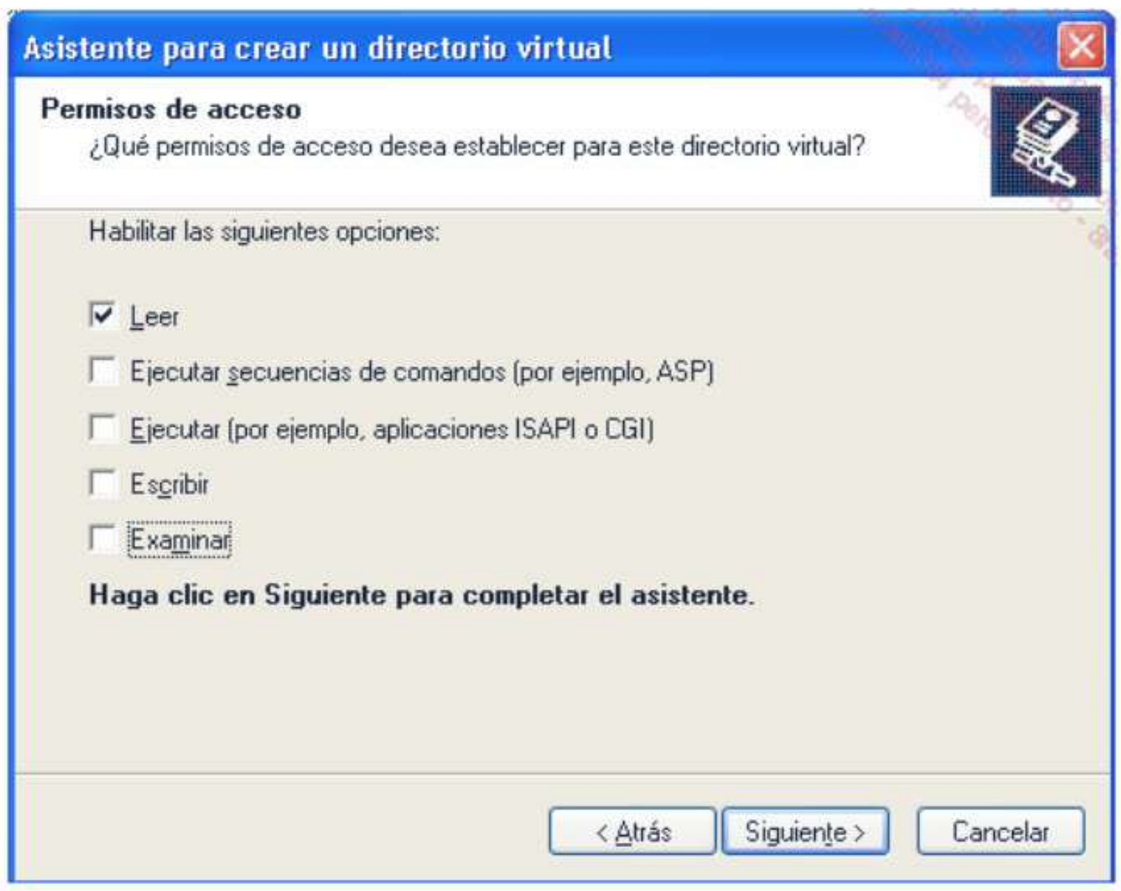
En la primera etapa, debemos proporcionar el alias del directorio virtual. Se trata del nombre utilizado en la URL para alcanzar esta ubicación.



En la segunda etapa, se nos pide proporcionar la ubicación del contenido que deseamos publicar. Suele ser una carpeta de alguno de los discos de la máquina, pero también puede ser un recurso compartido de red.



La última etapa configura las autorizaciones de acceso concedidas a este directorio virtual. Sólo la autorización de lectura es obligatoria.



Después de la creación del directorio virtual, se recomienda apagar y reiniciar el servidor Web.

d. Creación de la página Web de inicio

Durante esta última etapa, podemos expresar nuestro talento artístico para concebir la página de inicio. Los únicos límites son: informar en esta página un vínculo al archivo jnlp y respetar el nombre de la página de inicio tal y como se ha mencionado en el archivo jnlp. Añadir un vínculo se hace con la etiqueta html siguiente:

```
<a href="pizarra.jnlp">instalación de la aplicación pizarra mágica </a>
```