El lenguaje de programación Java

INTRODUCCIÓN

El lenguaje Java ha sido, desde su aparición pública en 1995, uno de los temas que más interés ha despertado en el mundo de la Informática. Ha merecido incluso la atención de publicaciones no especializadas y "Java" ha llegado a ser una de las palabras mágicas asociadas a las tecnologías de la información, como "Internet", "sistemas abiertos" o "cliente/servidor".

En su origen, la mayor parte de las aplicaciones Java se utilizaron para dotar de contenido dinámico e interactivo a las páginas del World Wide Web, mediante los llamados ***applets***. Sin embargo, aunque este sigue siendo un dominio de aplicación importante, el uso de Java se está extendiendo a un gran número de sistemas y aplicaciones, en las que el modelo ofrecido por Java de un entorno distribuido y completamente transportable entre plataformas es enormemente atractivo.

Java es un lenguaje de programación orientado a objetos con una sintaxis similar a C o C++, pero ofreciendo una mayor simplicidad y robustez en el ciclo de desarrollo: las construcciones y características más complicadas de C y C++ han sido eliminadas y el lenguaje contiene mecanismos implícitos para garantizar la seguridad de las aplicaciones construidas con él. Por otro lado, incorpora también dos mecanismos muy importantes a la hora de escribir programas simples, potentes y robustos: un tratamiento interno de multitarea y un sistema de excepciones que normaliza el procesado de errores por parte del programador.

Sin embargo, el principal atractivo de Java es que cumple uno de los más viejos sueños de los programadores: las aplicaciones escritas en Java son totalmente portables. De hecho, uno de los eslóganes más oídos sobre Java es: "compilar una vez, ejecutar en cualquier sitio". En realidad, Java no es solamente el lenguaje de programación, sino además la definición de una ***máquina virtual*** (la *máquina virtual Java*, o ***JVM***, sus siglas en inglés) encargada de ejecutar las aplicaciones. El compilador de Java produce instrucciones para esta máquina virtual (llamadas ***bytecodes***), que es independiente del sistema operativo o el procesador donde se esté ejecutando.

El lenguaje tiene su origen en 1990, en un grupo de investigadores de Sun Microsystems liderado por James Gosling. Este grupo trabajaba en técnicas para producir software capaz de ser incluido dentro de cualquier elemento electrónico (teléfonos, faxes, videos, y electrodomésticos en general) para proveerlos de "inteligencia" y capacidades de acceso a través de las redes de comunicación como Internet. El objetivo del grupo era conseguir un entorno de desarrollo para construir aplicaciones robustas, seguras y de altas prestaciones en entornos heterogéneos y altamente distribuidos. En principio, consideraron la posibilidad de utilizar lenguajes ya existentes como C++ o Smalltalk, pero pronto se hizo patente la necesidad de definir un máquina virtual capaz de garantizar la portabilidad de las aplicaciones, a la vez que cumpliera los requisitos de seguridad derivados de su uso en dispositivos de uso común.

Originalmente, el lenguaje recibió el nombre de *Oak*, pero al descubrir que ya existía un lenguaje con ese nombre, la afición de los miembros del equipo por el café les llevó a bautizarlo con su nombre actual: *java* es una forma coloquial de referirse al café. Otra cuestión curiosa es que, en principio, el lenguaje iba a caer en desuso: el grupo dedicado a dotar de inteligencia a los aparatos de uso común fue disuelto dado que los directivos de Sun no veían un mercado potencial para sus resultados. Sin embargo, Gosling intuyó las posibilidades del len-

guaje para proporcionar contenidos activos en las páginas Web y se embarcó en la construcción de un browser de Web basado en Java: *HotJava*, que constituyó la demostración general de las capacidades del lenguaje.

A partir de ahí comenzó la historia de Java como lenguaje íntimamente ligado a Internet y el comienzo de la revolución que ha llevado asociada a él. Puede que no se trate del lenguaje en el que van a escribirse las aplicaciones del futuro, pero sí es cierto que Java ha supuesto un hito (como el sistema operativo *Unix* o la aparición del PC) en el desarrollo de la Informática.

1.1 Las principales características de Java

Como ya hemos dicho, Java es un lenguaje de programación que está soportado por dos elementos fundamentales: el compilador y la máquina virtual. El compilador, encargado de traducir los programas a un formato capaz de ser entendido por el ordenador que debe ejecutarlo, es un elemento común con cualquier lenguaje de programación. La máquina virtual Java ("el ordenador encargado de ejecutar" los programas traducidos por el compilador) aporta nuevas características al modelo de programación del lenguaje. Utilizando como fuente uno de los primeros artículos sobre Java, escrito por su creador, podemos decir que las principales características de Java son que se trata de un lenguaje:

- *Simple*
- *Orientado a objetos*
- *Distribuido*
- *Robusto*
- *Seguro*
- *Arquitecturalmente neutro*
- *Portable*
- *Concurrente*
- *Dinámico*
- *Interpretado*
- *De alto rendimiento*

Veamos ahora qué es lo que hay detrás de cada una de estas características.

1.1.1 Simple

Java ofrece toda la funcionalidad de lenguajes potentes como C y C++ (en los que está basado), pero sin las características menos usadas y más confusas de éstos. De hecho, dado que C y C++ son lenguajes enormemente difundidos, la sintaxis de Java es muy similar, con el objetivo de facilitar su aprendizaje.

Las características que Java no incluye de C y C++ son:

- Gestión de la memoria dinámica y aritmética de punteros.
- Registros (`struct`)
- Redefinición de tipos (`typedef`)
- Macros (`#define`)

Según los creadores del lenguaje, esto reduce en un 50% los errores de programación más comunes.

Además, Java es compacto: una máquina virtual ocupa menos de 0.5 Mb de memoria.

1.1.2 Orientado a objetos

Java implementa la tecnología básica de C++ con algunas mejoras y elimina algunas características para mantener el objetivo de simplicidad del lenguaje. Java trabaja con sus datos como objetos y con interfaces a esos objetos.

1.1.3 Distribuido

Java se ha construido con extensas capacidades de interconexión a través de redes, que forman parte de sus componentes básicos. El modelo de acceso a la información a través de la red es en todo equivalente al acceso a la información disponible localmente.

1.1.4 Robusto

Java realiza verificaciones en busca de potenciales problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores lo antes posible en el ciclo de desarrollo. El manejo de memoria es realizado internamente, sin que el programador deba reservarla o liberarla explícitamente, eliminando así problemas de corrupción de datos y accesos no autorizados. Siguiendo esta filosofía, los arrays constituyen un tipo específico del lenguaje.

1.1.5 Seguro

La seguridad en Java tiene dos facetas. En el lenguaje, aspectos como los punteros se eliminan para prevenir el acceso ilegal a la memoria. Por otro lado, el código Java pasa muchos tests antes de ser ejecutado por la máquina virtual.

El código pasa a través de un verificador de *bytecodes* que comprueba el formato de las fragmentos de código y aplica un probador de teoremas para detectar fragmentos de código ilegal, como por ejemplo accesos no autorizados a las zonas de memoria de los objetos o intentos de cambio del tipo de algún objeto. Si los *bytecodes* pasan la verificación sin producir un mensaje de error, la máquina virtual se ha asegurado de que:

- El código no produce desbordamientos de operandos en la pila.
- El tipo de los parámetros de todos los códigos de operación son conocidos y

correctos.

- No ha ocurrido ninguna conversión ilegal de datos.
- El acceso a los campos de los objetos es legal.

El ***cargador de clases*** es el otro elemento que ayuda a mantener la seguridad en la máquina virtual Java, separando el espacio de nombres del sistema local del de los recursos procedentes de la red. Esto limita cualquier aplicación del tipo *Caballo de Troya* ya que las clases se buscan primero entre las locales y luego entre las procedentes del exterior. Las clases importadas de la red se almacenan en un espacio de nombres privado, asociado con el origen. Cuando una clase del espacio de nombres privado accede a otra clase primero se busca entre las clases predefinidas del sistema local y luego en el espacio de nombres de la clase que hace la referencia. Así se evita que las clases puedan ser suplantadas.

1.1.6 Arquitecturalmente neutro

El fichero objeto producido por el compilador de Java tiene un formato independiente de la arquitectura (sistema operativo, procesador, etc.) de la máquina donde se ejecuta. Como ya hemos visto, este fichero contiene código ejecutable (*bytecodes*) en una máquina virtual que es implementada por el sistema de *run-time*, que es el que tiene en cuenta las particularidades de la máquina donde se ejecuta.

1.1.7 Portable

Más allá de la portabilidad básica por ser independiente de la arquitectura, Java implementa otros estándares de portabilidad para facilitar el desarrollo. Por ejemplo, los enteros son siempre de 32 bits y almacenados en complemento a 2, mientras que las cadenas de caracteres utilizan ***Unicode***. Además, el interface de ventanas de Java (AWT) es un sistema abstracto de ventanas que permite su implementación en entornos diferentes, como X-Window, MS-Windows, MacOS, OS/2, etc.

1.1.8 Concurrente

Java permite que múltiples flujos de control (***threads***) puedan ser ejecutados por el mismo programa. Los threads son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads construidos en el lenguaje, son más fáciles de usar, más robustos y, por supuesto, más portables que sus homólogos en otros lenguajes que no los soportan de manera nativa. La capacidad de programación concurrente permite mejorar el rendimiento interactivo y el comportamiento en tiempo real.

1.1.9 Dinámico

Java se beneficia en todo lo posible de la tecnología orientada a objetos. Los módulos que componen la aplicación no son conectados hasta el tiempo de ejecución. Las librerías nuevas o actualizadas son incorporadas (siempre que mantengan el interface) automáticamente.

Por otro lado, Java proporciona mecanismos para cargar dinámicamente clases desde la red, de manera que nuevos contenidos de información podrán ser tratados por manejadores específicos.

1.1.10 Interpretado

Dado que la máquina virtual Java es un programa que se ejecuta sobre el sistema operativo del ordenador, la ejecución de un programa Java se basa, en esencia, en un intérprete (la máquina virtual). Aunque no se trata de un intérprete en el sentido tradicional: los *bytecodes* Java han pasado ya por las etapas de validación del compilador.

1.1.11 De alto rendimiento

Para los casos en que la velocidad del intérprete Java no resulte suficiente, se dispone ya de mecanismos como los compiladores ***JIT*** (*Just In Time*), que se encargan de traducir (a medida que va siendo necesario) los *bytecodes* Java a instrucciones de código máquina. Otros mecanismos, como compiladores incrementales y sistemas dedicados para tiempo real, se encuentran también en el mercado.

CONCEPTOS DE PROGRAMACIÓN ORIENTADA A OBJETOS

Como ocurre con el mismo término “Java”, la programación orientada a objetos ha sido una de esas palabras de moda en los últimos años dentro del campo de la Informática. Se han dado numerosas definiciones de lo que es la programación orientada a objetos y de los conceptos que lleva asociados. Existen también diferentes *metodologías* que pretenden definir cómo debe aplicarse el diseño orientado a objetos a problemas concretos.

En esta sección veremos una serie de conceptos básicos de la programación orientada a objetos, necesarios para entender y utilizar adecuadamente el lenguaje Java. No se trata de una presentación exhaustiva y deliberadamente utilizaremos los términos más próximos a los que se emplean en Java para definir los diferentes conceptos.

2.1 Objetos

Como implica el nombre *orientado a objetos*, estos son la base para entender la programación orientada a objetos: un objeto es un conjunto de elementos software (***variables*** y ***métodos***) que, típicamente, pretende modelar un objeto del mundo real. En el mundo real basta mirar alrededor para ver miríadas de objetos: un perro, una mesa, un televisor, una bicicleta...

Estos objetos del mundo real tienen dos características en común: tienen un ***estado*** y un ***comportamiento***. Por ejemplo, los perros se encuentran en un estado (nombre, color, raza, hambre...) y se comportan de una determinada manera (ladran, corren, duermen, comen...). Igual ocurre para el estado (marcha, velocidad de pedaleo, presión de los frenos...) y el comportamiento (frenado, cambio de marcha...) de las bicicletas. De la misma forma, los objetos software tienen un estado y un comportamiento. Un objeto software almacena su estado en ***variables*** y lleva a cabo su comportamiento por medio de ***métodos***.

Los objetos software pueden representar objetos reales, como podrían ser modelos de un perro en un programa de animación o de una bicicleta en un sistema de simulación, pero su capacidad llega más allá: los objetos software pueden modelar conceptos abstractos. Por ejemplo, un ***evento*** es una abstracción utilizada comúnmente para representar la acción que el usuario realiza (mover el ratón, pulsar una tecla) en un interface gráfico.

Todo lo que el objeto software sabe (su estado) y puede hacer (su comportamiento) viene expresado por las variables y métodos del objeto. Todo lo que el objeto no sabe o no puede hacer está excluido de él. Así el objeto que represente una bicicleta tendrá variables para indicar la marcha en que está y un método “frenar”, pero no una variable que contenga su raza o un método “ladrar”.

El estado del objeto (sus variables) constituyen el núcleo del mismo. Su comportamiento (los métodos) rodean este núcleo y lo esconden frente a otros objetos. El empaquetamiento de las variables de un objeto dentro de la envoltura protectora de los métodos se conoce como ***encapsulación***. Se utiliza, típicamente, para ocultar detalles de implementación no relevantes para el comportamiento del objeto. Para cambiar de marcha en una bicicleta no es necesario

saber cómo funciona el mecanismo del cambio, basta accionar la palanca. De la misma manera, en un programa, no es necesario saber cómo se ha implementado un objeto, basta con saber qué métodos hay que invocar. La encapsulación supone dos beneficios fundamentales a la hora de escribir programas:

- Modularidad: El código fuente de un objeto puede ser desarrollado y mantenido independientemente de los otros objetos, e incluso ser reutilizado en otros programas.
- Independencia de la implementación: Un objeto tiene un interface público que otros objetos pueden usar para comunicarse con él. Pero el objeto puede mantener información interna que puede cambiar en cualquier momento sin afectar a los otros objetos que dependen de él.

A menudo, sin embargo, por motivos de eficiencia, puede resultar conveniente eliminar hasta cierto punto la encapsulación. En Java, como en otros lenguajes, existen mecanismos para que un objeto controle cuáles de sus variables pueden ser leídas o modificadas por otros objetos. De manera similar, ciertos métodos pueden hacerse internos, impidiendo el acceso a todos o parte de los otros objetos. En definitiva, el objeto tiene un control completo de qué otros objetos tienen acceso a sus métodos y variables.

2.2 Clases

En el mundo real existen muchos objetos del mismo tipo. Una bicicleta particular no es más que una de las muchas bicicletas que existen en el mundo. Utilizando la terminología de la programación orientada a objetos, diríamos que esa bicicleta es una ***instancia*** de la ***clase*** de los objetos conocidos como bicicletas. Las bicicletas tienen características de estado y de comportamiento comunes. Sin embargo, el estado particular y el comportamiento concreto de cada bicicleta es diferente del resto de las mismas. En definitiva, una clase es un prototipo que define las variables y métodos que son comunes a todos los objetos del mismo tipo. De esta manera, un objeto particular es una instancia de una clase.

El estado de un objeto viene definido por el valor de sus variables (variables de instancia). De manera que, después de crear la clase “bicicleta”, hay que ***instanciarla*** (crear una instancia de ella) para poder utilizarla. Cuando se crea una instancia de la clase, estamos creando un objeto de ese tipo y podremos invocar a sus métodos para utilizarlo.

Junto con los variables y métodos de instancia, asociados con un objeto particular, pueden definirse también variables y métodos de clase. Las variables y métodos de clase pueden accederse sin necesidad de crear una instancia de la clase. Los métodos de clase pueden operar únicamente sobre variables de clase, ya que no tienen acceso a los valores particulares que existan para las variables de instancia. Un ejemplo de variable de clase podría ser el número de ruedas en la clase “bicicleta”, que es común para todos los objetos de este tipo y no está sujeta a cambios en una realización particular de la misma. Todas las instancias comparten el mismo valor y, si este valor cambia, cambia automáticamente para todas ellas.

2.3 Herencia

En términos generales, los objetos se definen por medio de clases. Puede saberse mucho de un objeto a partir de la clase a la que pertenece. Sin necesidad de ver una bicicleta, cualquiera puede hacerse a la idea de que tiene dos ruedas, un manillar, pedales...

La programación orientado a objetos lleva esto un paso más allá y permite definir clases en términos de otras clases. Por ejemplo, las bicicletas de montaña, las bicicletas de carreras y los tándems son diferentes tipos de bicicleta. En términos de programación orientada a objetos, todas ellas son ***subclases*** de la clase "bicicleta". De manera similar la clase "bicicleta" es la ***superclase*** de las "bicicleta de montaña", "bicicleta de carreras" y "tándem".

Cada subclase hereda la definición de estado (en la forma de declaración de variables) de su superclase. Las bicicletas de montaña, las de carreras y los tándems tienen variables de estado comunes: frecuencia de pedaleo, velocidad, etc. También heredan los métodos de su superclase. Las bicicletas de montaña, las de carreras y los tándems tienen comportamientos comunes: frenar, cambiar la velocidad de pedaleo, etc.

Sin embargo, las subclases no están limitadas al estado y comportamiento de su superclase. Las subclases pueden añadir variables y métodos a los que heredan de su superclase: los tándems tienen dos asientos y cuatro pedales, las bicicletas de montaña suelen tener más marchas...Las subclases también pueden cambiar los métodos y proporcionar una implementación especializada de los mismos. Por ejemplo, una bicicleta de montaña con más marchas ofrece un método "cambiar marcha" que implica el uso de las marchas adicionales, no disponibles en su superclase.

Este esquema no está limitado, como es lógico, a un solo nivel. El árbol de herencia (o ***jerarquía de clases***), puede ser tan profundo como sea necesario. Los métodos y variables se heredan a través de sus diferentes niveles. En general, cuanto más profundo se encuentre una clase en la jerarquía, más especializado será su comportamiento.

Las subclases proporcionan comportamientos especializados a partir de una base de elementos comunes proporcionados por la superclase. Utilizando la herencia, los programadores pueden reutilizar el código de la superclase en casos más concretos.

Por último, pueden definirse superclases llamadas ***clases abstractas***, que definen comportamientos genéricos. La superclase abstracta define (y posiblemente implementa parcialmente) el comportamiento, pero la mayor parte de la clase está sin implementar. Otros programadores proporcionan la implementación detallada con subclases especializadas.

LOS COMPONENTES DEL LENGUAJE

Aquí tenemos el primer ejemplo de código Java: una clase llamada `Contador` y uno de sus métodos (llamado `cuentaCaracs`) que lee y cuenta caracteres utilizando un Lector (un objeto que ofrece un flujo de caracteres de entrada) y, al terminar, presenta el número de caracteres leídos. Usaremos esta pequeña pieza de código para ilustrar las características del lenguaje, como los tipos, variables, operadores y expresiones.

```
import java.io.*;
public class Contador {
   public static void cuentaCaracs (Lector in) throws IOException {
      int cuenta = 0;
      while (in.lee() != -1)
         cuenta++;
      System.out.println("Contados " + cuenta + " caracteres.");
   }
            // ... Otros metodos omitidos ...
}
```

3.1 Variables y tipos

Las ***variables*** son los nombres de un lenguaje de programación: son las entidades (valores y datos) que actúan y sobre las que se actúa. Una ***declaración*** de variable siempre contiene dos componentes: el ***tipo*** de la variable y su ***nombre***. La localización de la declaración, es decir, el lugar donde aparece en relación con otros elementos del programa, determina su ***ámbito***.

3.1.1 Tipos

Todas las variables en un programa Java deben tener un tipo. El tipo de una variable determina los valores que puede contener y las operaciones que pueden realizarse sobre ella. Por ejemplo, la declaración `int cuenta` declara que `cuenta` es un entero. Este tipo de variables puede contener sólo valores enteros y pueden usarse los operadores aritméticos habituales (+, -, * y /) sobre ellas para realizar las operaciones aritméticas correspondientes.

Hay dos categorías básicas de tipos en Java: tipos ***primitivos*** y ***de referencia***. La siguiente tabla incluye los tipos primitivos soportados por Java, sus tamaños y una pequeña descripción de cada uno de ellos.

Identificador	Tamaño/Formato	Descripción
Números enteros		
`byte`	8 bits, complemento a 2	Entero de un byte
`short`	16 bits, complemento a 2	Entero pequeño
`int`	32 bits, complemento a 2	Entero
`long`	64 bits, complemento a 2	Entero grande
Números reales		
`float`	32 bits, IEE 754	Coma flotante en simple precisión
`double`	64 bits, IEE 754	Coma flotante en doble precisión
Otros tipos		
`char`	16 bits, Unicode	Carácter
`boolean`	`true` o `false`	Valor booleano

Una variable de tipo primitivo contiene un único valor del tamaño y formato correspondientes a su tipo: un número, carácter o valor booleano. Los arrays, clases e interfaces son tipos de referencia. El valor de una variable de tipo de referencia, a diferencia de una de tipo primitivo, es una referencia al valor o conjunto de valores real representado por la variable. Una referencia es como una dirección: no es el valor en sí mismo pero sí contiene el medio para alcanzarlo.

El método `cuentaCaracs` usa una variable de tipo de referencia, `in`, que es un objeto `Lector`. Cuando se usa en una sentencia o una expresión, el nombre se evalúa como una referencia al objeto. De manera que el nombre de un objeto se utiliza para acceder a sus variables o invocar sus métodos (tal como `cuentaCaracs` hace al llamar a `lee`).

3.1.2 Nombres de variable

En un programa, las variables son referenciadas por su nombre. En Java, un nombre de variable debe cumplir las siguientes reglas:

1. Debe ser un identificador legal Java conteniendo caracteres ***Unicode***.Unicode es una norma de codificación de caracteres diseñada para acomodar texto escrito en diferentes idiomas. Permite la codificación de hasta 65.536 caracteres, de los cuales están ya asignados 34.168. De esta manera, `numero`, `número` y `νυμερο` son identificadores válidos en un programa Java.
2. No debe ser una palabra clave del lenguaje o un literal booleano (`true` o `false`).
3. No debe tener el mismo nombre que otra variable cuya declaración aparezca en el mismo ***ámbito***.

Por convención, los nombres de variables suelen comenzar por una letra minúscula y los nombres de clases por una letra mayúscula. Si el nombre de una variable consiste en más de una palabra (como en el caso de `esVisible`), las palabras se unen y cada palabra después de la primera se empieza con una letra mayúscula.

3.1.1 Ámbito

El ámbito de una variable es el bloque de código dentro del cual la variable es accesible y determina en qué momentos la variable es creada y destruida. La localización de una variable dentro de un programa define su ámbito y la pone en una de estas cuatro categorías:

- Variable miembro
- Variable local
- Parámetro de método
- Parámetro de manejador de excepciones

Una variable miembro es miembro de una clase o un objeto y define su estado. Puede ser declarada en cualquier lugar de una clase pero no dentro de un método. Una variable miembro es accesible a todo el código de la clase. Veremos más sobre ellas en el próximo capítulo, relativo a clases y objetos en Java.

Las variables locales pueden declararse en cualquier lugar de un método o dentro de un bloque de código dentro de un método. En `cuentaCaracs`, `cuenta` es una variable local. Su ámbito, es decir, el código que puede acceder a `cuenta`, se extiende desde su declaración hasta el final del método (indicado por la llave hacia la derecha). En general, una variable local es accesible desde su declaración hasta el final del bloque en el que fue declarada.

Los parámetros de método son argumentos formales de los métodos y constructores y se usan para pasar valores a los mismos. En `cuentaCaracs`, `in` es un parámetro para este método. El ámbito de uno de estos parámetros es el método entero para el que se han definido.

Los parámetros de manejador de excepciones son similares a los parámetros de método, con la diferencia de que son argumentos para un manejador de excepciones. En el capítulo correspondiente veremos cómo se tratan los errores en Java a través de excepciones y las características de estos parámetros.

3.1.2 Inicialización

Las variables locales y las variables miembro pueden ser inicializadas cuando se declaran mediante una sentencia de asignación. El tipo de ambos lados de la asignación debe ser el mismo. En el método `cuentaCaracs` se da un valor inicial de cero a `cuenta` cuando se la declara:

```
int cuenta = 0;
```

El valor asignado a la variable debe ser coherente con su tipo.

Los parámetros de método o de manejador de excepciones no pueden inicializarse de este modo. La asignación de valor se produce en la llamada.

3.1.3 Variables finales

Una variable puede declararse final, incluyendo aquéllas que son parámetros para métodos. El valor de una variable final no se puede cambiar una vez ha sido inicializada.

Para declarar una variable como final se incluye la palabra `final` en la declaración de la variable antes de su tipo:

```
final int unaVariableFinal = 0;
```

La línea anterior declara una variable final y la inicializa, todo a la vez. Posteriores intentos de asignar un valor a `unaVariableFinal` resultarán en errores de compilación. Si es necesario, la inicialización de una variable final puede hacerse después de su declaración:

```
final int bFinal;
                . . .
bFinal = 0;
```

3.2 Operadores

Los operadores realizan una determinada función sobre uno, dos o tres operandos. Los operadores que requieren un operando son llamados ***unarios***. Por ejemplo, `++` es un operador unario que incrementa en 1 el valor de su operando. Los operadores que requieren dos operandos son llamados ***binarios***. Por ejemplo, `=` es un operador binario que asigna el valor de su operando derecho a su operando izquierdo. Por último, los operadores ***ternarios*** requieren tres operandos. Java tiene un operador ternario, `?:`, que es una manera simplificada de escribir sentencias `if-else`.

Los operadores unarios de Java pueden usarse en notación prefija o postfija:

```
operador op
op operador
```

Todos los operadores binarios de Java usan notación infija:

```
op1 operador op2
```

El operador terciario usa también notación infija:

```
expr ? op1 : op2
```

Además de realizar la operación, un operador retorna también un valor. El valor y su tipo dependen del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos retornan números y su tipo depende del tipo de sus operandos: sumar dos enteros retorna a su vez un entero. Se dice que una operación se evalúa a su resultado.

3.2.1 Operadores aritméticos

Esta tabla resume los operadores binarios aritméticos disponibles en Java:

Operador	Uso	Descripción
`+`	op1 + op2	Suma op1 y op2
`-`	op1 - op2	Resta op2 de op1
`*`	op1 * op2	Multiplica op1 por op2
`/`	op1 / op2	Divide op1 por op2
`%`	op1 % op2	Calcula el resto de dividir op1 por op2

Java extiende la definición del operador + para incluir la concatenación de cadenas de caracteres. El método `cuentaCaracs` usa + para concatenar "Contados", el valor de `cuenta` y "caracteres":

```
System.out.println("Contados " + cuenta + " caracteres.");
```

Esta operación automáticamente traduce el valor de `cuenta` al tipo `String`, del que veremos más un poco más adelante.

Los operadores + y - tienen versiones unarias que realizan las siguientes operaciones:

Operador	Uso	Descripción
`+`	+ op	Transforma op a `int` si es `byte`, `short` o `char`
`-`	- op	Cambia el signo de op

Hay también dos operadores unarios para simplificar ciertas expresiones: ++ y --, que incrementan y decrementan su operando en 1. El método `cuentaCaracs` usa ++ para incrementar la variable `cuenta` cada vez que lee un carácter de la entrada con al sentencia:

```
cuenta++;
```

Aquí hemos usado la versión postfija del operador. La versión prefija tiene el mismo efecto sobre el operando. La diferencia entra ambas es el valor que retorna el operador: en la versión postfija se retorna el valor del operando *antes* de su incremento, mientras que la prefija retorna el valor que tiene *después* del incremento.

Esta diferencia no tiene importancia en `cuentaCaracs` pero puede se crítica en otras situaciones. Por ejemplo, el siguiente bucle se ejecutará una vez menos si cambiamos `ct++` por `++ct`:

```
do {
      . . .
} while (ct++ < 6);
```

Las características de estos operadores se resumen en la siguiente tabla:

Operador	Uso	Descripción
`++`	op ++	Incrementa op en 1. Retorna el valor antes del incremento
`++`	++ op	Incrementa op en 1. Retorna el valor después del incremento
`--`	op --	Decrementa op en 1. Retorna el valor antes del decremento
`--`	-- op	Decrementa op en 1. Retorna el valor después del decremento

3.2.2 Operadores relacionales y condicionales

Un operador relacional compara dos valores y determina la relación entre ellos. En esta tabla se resumen los operadores relacionales de Java:

Operador	Uso	Retorna `true` si
`>`	op1 > op2	op1 es mayor que op2
`>=`	op1 >= op2	op1 es mayor o igual que op2
`<`	op1 < op2	op1 es menor que op2
`<=`	op1 <= op2	op1 es menor o igual que op2
`==`	op1 == op2	op1 y op2 son iguales
`!=`	op1 != op2	op1 y op2 no son iguales

Los operadores relacionales se utilizan junto con los condicionales para construir expresiones más complejas. Es importante tener en cuenta que, en ciertos casos, el segundo operando de un operador condicional puede que no sea evaluado. Tomemos como ejemplo el operador && que retorna `true` si ambos operandos son `true`. Consideremos la línea:

```
((cuenta > NUM_ENTRADAS) && (in.lee() != -1))
```

Si `cuenta` es menor que `NUM_ENTRADAS`, la operador izquierdo de && se evalúa a `false`. Dado que esto implica que && retornará `false`, `in.lee` no será invocado y no se leerá un nuevo carácter de la entrada.

El operador & es similar a &&, pero siempre evalúa ambos operandos.

Java soporta cinco operadores condicionales binarios:

Operador	Uso	Retorna `true` si
&&	op1 && op2	op1 y op2 son `true`
||	op1 || op2	op1 o op2 son `true`
!	! op	op es `false`
&	op1 & op2	op1 y op2 son `true`. Siempre se evalúa op2
|	op1 | op2	op1 o op2 son `true`. Siempre se evalúa op2

Además, Java incluye otro operador condicional: ?: Este operador es terciario y, básicamente, consiste en una forma simple de escribir sentencias `if-else`:

```
expresion ? op1 : op2
```

El operador ?: evalúa `expresion` y retorna `op1` si es `true` y `op2` si es `false`.

3.2.3 Operadores de bit

Un operador de bit permite la manipulación directa de los bits que representan los datos. Los operadores de bit que ofrece Java son:

Operador	Uso	Descripción
>>	op1 >> op2	Desplaza los bits de op1 op2 posiciones hacia la derecha
<<	op1 << op2	Desplaza los bits de op1 op2 posiciones hacia la izquierda
>>>	op1 >>> op2	Desplaza los bits de op1 op2 posiciones hacia la derecha (sin signo)
&	op1 & op2	AND booleano
|	op1 | op2	OR booleano
^	op1 ^ op2	XOR booleano
~	~ op1	NOT booleano

Los operadores de desplazamiento constituyen una forma rápida de realizar multiplicaciones por potencias de 2: un desplazamiento de un bit a la derecha es equivalente a dividir por 2, mientras que hacia la izquierda es equivalente a multiplicar por 2.

Entre otras cosas, las manipulaciones de bit pueden ser útiles para manejar conjuntos de flags booleanos. Supongamos, por ejemplo, que tenemos varios flags en un programa para indicar el estado de los componentes del mismo: si está visible, si se puede mover, etc. Mejor que definir una variable booleana para cada condición, se puede definir una sola variable para todas. Cada bit en esta variable representa el estado de una de las condiciones, utilizando las operaciones de manejo de bits para establecerlas y acceder a ellas.

En primer lugar, se definen constantes que indiquen el valor de los diferentes flags del programa. Estos flags deben ser diferentes potencias de 2 para asegurarnos de que su bit no "machaca" el de otra condición. En el siguiente ejemplo inicializa los flags a 0, indicando que ninguna de las condiciones es verdadera:

```
final int VISIBLE = 1;
final int MOVIL = 2;
final int SELECCIONABLE = 4;
final int EDITABLE = 8;
int flags = 0;
```

Para colocar el flag de "visible" cuando pasa a ese estado bastaría con hacer:

```
flags = flags | VISIBLE;
```

Y para comprobar si el flag está o no:

```
flags & VISIBLE
```

3.2.4 Operadores de asignación

El operador básico de asignación, =, se utiliza para asignar un valor a otro. Java proporciona además otros operadores de asignación para simplificar ciertas sentencias. Aquí se incluye una lista de ellos:

Operador	Uso	Equivalente a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
>>=	op1 >>= op2	op1 = op1 >> op2
<<=	op1 <<= op2	op1 = op1 << op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

3.3 Expresiones

Una expresión es una serie de variables, operadores y llamadas a métodos construida de acuerdo con la sintaxis del lenguaje y que se evalúa a un único valor.

Las expresiones realizan el trabajo efectivo en un programa Java. Entre otras cosas, se utilizan para calcular y asignar valores a las variables y para controlar el flujo de ejecución del programa. El trabajo de una expresión es doble: realiza el cálculo definido por sus elementos y retorna un valor como resultado del cálculo.

Como vimos antes, los operadores retornan un valor, de manera que el uso de un operador es en sí una expresión. Por ejemplo, la siguiente sentencia de `cuentaCaracs` es una expresión:

```
cuenta++;
```

Esta expresión en particular se evalúa al valor de `cuenta` antes de que la operación se produzca. El tipo del valor que retorna una expresión depende de los elementos que se utilizan en ella. Así, `cuenta++` retorna un entero porque ++ retorna un valor del mismo tipo que su operando y `cuenta` es un entero. Otras expresiones retornan valores booleanos, cadenas, etc.

Otra expresión interesante en `cuentaCaracs` es la siguiente:

```
in.lee() != -1;
```

Esta expresión consiste en dos expresiones simples. La primera es una llamada a un método:

```
in.lee()
```

Una llamada a un método se evalúa al valor de retorno del método. De esta manera, el tipo de una expresión de llamada a un método es el mismo que el de retorne del método. El método `in.lee` declara que retorna un entero, de manera que la expresión `in.lee()` se evalúa a un entero.

La segunda expresión en la sentencia de arriba usa el operador !=. Este operador compara dos operandos para comprobar si son distintos. En este case, los operandos son in.lee() y -1, de manera que != compara dos enteros y se evalúa a un valor booleano.

En definitiva, Java permite construir expresiones compuestas de varias expresiones simples siempre y cuando los tipos requeridos por una de las partes de la expresión satisfaga los requisitos de las otras. Por otro lado, el orden en que se evalúan estas expresiones es importante.

Para que el compilador Java evalúe las expresiones en el orden pretendido por el programador, las expresiones que deben evaluarse antes deben encerrarse entre paréntesis. De esta manera, en:

```
(x + y) / 100
```

La suma se efectúa antes que la división.

Si el compilador no recibe de forma explícita el orden de evaluación, decide este orden basado en la ***precedencia*** de los operadores. Los operados con más alta precedencia son evaluados antes. Por ejemplo, la división tiene una precedencia mayor que la suma, de manera que:

```
x + y / 100
```

Es equivalente a:

```
x + (y / 100)
```

En la tabla siguiente se incluye la precedencia asignada a los operadores de Java. Los operadores aparecen en orden de precedencia: cuanto más arriba aparece el operador en la tabla, mayor precedencia tiene. Los operadores que aparecen en la misma línea tienen la misma precedencia:

postfijos	`[] . (params) expr++ expr--`
unarios	`++expr --expr +expr -expr ~ !`
multiplicativos	`* / %`
aditivos	`+ -`
desplazamiento de bits	`<< >> >>>`
relacionales	`< > <= >= instanceof`
de igualdad	`== !=`
AND de bits	`&`
XOR de bits	`^`
OR de bits	`|`
AND lógico	`&&`
OR lógico	`||`
condicional	`?:`
de asignación	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Cuando aparecen en una expresión operadores de igual precedencia, la regla que gobierna el orden de evaluación es que éstos se evalúan de izquierda a derecha, excepto los de asignación, que se evalúan de derecha a izquierda.

3.4 Control de flujo

En el método `cuentaCaracs` se utiliza una sentencia `while` para hacer un bucle con todos los caracteres de la entrada y contarlos. En general, una sentencia `while` realiza una acción mientras una determinada condición mantenga el valor `true`. La sintaxis general de `while` es:

```
while (expresion)
   sentencia
```

`sentencia` puede ser una sentencia simple, como en el caso de `cuentaCaracs`, o puede ser un bloque de sentencias. Un bloque de sentencias es una serie de sentencias Java contenidas entre llaves ('`{`' y '`}`'). Por ejemplo, supongamos que además de incrementar `cuenta` quisiéramos ver su valor en cada iteración del bucle. En ese caso el bucle sería:

```
      . . .
while (in.lee() != -1) {
   cuenta++;
   System.out.println("Caracter leido. Cuenta = " + cuenta);
}
      . . .
```

Una sentencia como while es una sentencia de control de flujo: determina el orden en que las demás sentencias se ejecutan. Además de while, Java ofrece varias sentencias de control de flujo que veremos en los siguientes apartados.

3.4.1 La construcción if-else

Mediante `if-else` un programa decide selectivamente ejecutar otras sentencias de acuerdo con el valor de una expresión booleana. El formato general de una construcción `if-else` es:

```
if (expresion1)
   sentencia
else if (expresion2)
   sentencia
      . . .
else
   sentencia
```

Un `if` puede tener un número arbitrario de construcciones `else if` y una sola `else`. De todos ellos se ejecutará el primero de ellos para los que la expresión en cuestión se evalúe a `true`.

3.4.2 La construcción switch

La construcción switch permite ejecutar condicionalmente sentencias de acuerdo con el valor de una determinada expresión. Por ejemplo, supongamos un trozo de código para calcular el número de días de un mes a partir del número del mismo:

```
int mes;
int año;
int dias;
      . . .
switch (mes) {
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
```

```
   dias = 31;
   break;
case 4:
case 6:
case 9:
case 11:
   dias = 30;
   break;
case 2:
   if ( ((año % 4 == 0) && !(año % 100 == 0)) || (año % 400 == 0) )
      dias = 29;
   else
      dias = 28;
   break;
default:
   System.out.println (“Número de mes erróneo”);
   break;
}
```

El `switch` evalúa el valor de la expresión (`mes` en nuestro caso) y las sentencias asociadas a cada `case` se ejecutan. Hay que tener aquí en cuenta dos aspectos importantes: si no se usa un `break` al final de cada `case`, las sentencias correspondientes al `case` siguiente se ejecutarán también (lo que se conoce como *fall-through*). Por último, la palabra especial `default` se usa de forma similar a un `else` en la construcción `if-else` para todos los valores no contemplados: si no se usa `default`, un valor no contemplado implicará que no se ejecutará ninguna sentencia incluida en el `switch`.

3.4.3 Construcciones de bucle

Aparte de `while`, que ya hemos visto, Java tiene otras dos construcciones de bucle: `for` y `do-while`.

El uso del bucle `for` está indicado cuando se conocen los límites del bucle (su inicialización, condición de fin y mecanismos de incremento). Su estructura general es:

```
for (inicialización; fin; incremento)
   sentencias
```

`inicialización` es una sentencia que inicializa el bucle: se ejecuta una sola vez al comienzo del mismo. `fin` es una expresión que determina cuándo se acaba el bucle: se evalúa al comienzo de cada iteración y si es `false` el bucle se termina. Por último, `incremento` es una expresión que es invocada al final de cada iteración del bucle. Cualquiera (o todas) ellas pueden ser expresiones vacías (un simple punto y coma).

El bucle `do-while` es similar al `while`, pero la expresión es evaluada al final de bucle, de manera que se garantiza que las sentencias que contiene se ejecutan al menos una vez:

```
do {
   sentencias
} while (expresiónBooleana);
```

3.4.4 Construcciones para el manejo de excepciones

Cuando ocurre un error dentro de un método Java, el método puede ***lanzar una excepción*** para indicar al código llamante que se ha producido un error y el tipo del mismo. El método llamante puede usar las sentencias `try`, `catch` y `finally` para tratar la excepción. Estos mecanismos los veremos con detalle más adelante (“Manejadores de excepciones”, página 54).

3.4.5 Construcciones de salto

Vimos en la construcción `switch` la sentencia `break`, que hace que el flujo de control salte hasta el final de la construcción actual. La sentencia `break` puede usarse también para dirigir el flujo de control a un punto particular del código utilizando una etiqueta. Una sentencia se puede etiquetar colocando un identificador Java seguido de dos puntos (el carácter ‘`:`’) antes de la sentencia:

```
estoEsUnaEtiquta: sentencia
```

Para saltar a esta sentencia `break` se debe usar de la forma:

```
break estoEsUnaEtiqueta;
```

La sentencia `continue` se puede utilizar dentro de bucles para saltar a otra sentencia. Tiene también dos formas: etiquetada y sin etiquetar. Si se usa la forma sin etiquetar, el control se transfiere a la condición de final del bucle. Esta condición se vuelve a evaluar y el bucle continúa o no dependiendo del resultado. Si se usa la forma etiquetada de `continue`, el bucle sigue a partir de la sentencia que tiene la etiqueta.

La última construcción de salto de Java es la sentencia `return`. Esta sentencia se utiliza para salir del método actual y volver a la sentencia dentro del método llamante que está inmediatamente después de la llamada. Hay dos formas de la sentencia `return`: una que devuelve un valor y otra que no lo hace. En el primer caso, simplemente se coloca el valor (o una expresión que lo calcule) después de la palabra `return`:

```
return ++cuenta;
```

El valor retornado debe coincidir con el tipo declarado para el método. Cuando un método se declara `void` se debe utilizar la forma de `return` que no retorna valor:

```
return;
```

3.5 Arrays y cadenas

Como otros lenguajes de programación, Java permite agrupar y manejar múltiples valores de un mismo tipo en arrays, así como manejar datos compuesto de múltiples caracteres por medio de cadenas de caracteres (objetos de clase `String`).

3.5.1 Arrays

Como en el caso de cualquier otra variable, los arrays deben ser declarados antes de utilizarlos. De la misma manera, la declaración de una array tiene dos componentes primarios: el tipo del array y su nombre. El tipo de un array define el tipo de los elementos que contiene. Por ejemplo un array de enteros se declara:

```
int [] arrayDeInts;
```

O bien, al estilo más tradicional de C:

```
int arrayDeInts [];
```

Aunque la primera forma es preferible.

La declaración de un array no reserva memoria para los elementos del mismo. Para ello debe instanciarse el array utilizando el operador `new`. En realidad, dado que un array es un objeto más de Java, la instanciación es necesaria como para cualquier otra variable de tipo no primitivo, como veremos en el siguiente capítulo. La siguiente sentencia reserva memoria para que `arrayDeInts` contenga diez elementos:

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

En general, cuando se crea un array, se utiliza el operador `new`, más el tipo de los elementos del array, más el número de elementos deseados entre corchetes:

```
tipo[] nombreDelArray = new tipo[tamañoDelArray];
```

Ahora que se ha asignado memoria para el array se puede asignar valores a sus elementos y acceder a los mismos:

```
for (int j = 0; j < arrayDeInts.length; j ++) {
   arrayDeInts[j] = j;
   System.out.println("[j] = " + arrayDeInts[j]);
}
```

Este ejemplo muestra que, para referenciar el elemento de una array, se añaden corchetes al nombre del mismo. Dentro de los corchetes se indica el índice del elemento en cuestión. En Java, los índices de los elementos de un array comienzan en 0 y terminan en la longitud del array menos 1.

Los arrays pueden contener cualquier tipo Java incluyendo tipos de referencia como objetos u otros arrays. Por ejemplo, la siguiente línea declara un array que puede contener diez objetos `String`:

```
String[] arrayDeStrings = new String[10];
```

En este punto, se ha reservado memoria para las referencias a los 10 objetos `String`, pero no para ellos. Si se intenta aquí acceder a alguno de los elementos del array se generará una `NullPointerException` porque el array está vacío y no contiene nada. Esto es a menudo una fuente de errores para los nuevos programadores en Java. Es necesario además reservar espacio para los objetos `String`:

```
for (int i = 0; i < arrayDeStrings.length; i ++) {
   arrayDeStrings[i] = new String("Hola " + i);
}
```

3.5.2 Cadenas

Una secuencia de datos en forma de caracteres es llamada una cadena de caracteres (*string*) y se implementa en Java por medio de la clase `String`. El método `cuentaCaracs` utiliza dos objetos `String` en forma literal (una cadena de caracteres entre comillas dobles). El programa implícitamente reserva espacio para dos objetos `String`, uno por cada uno de las cadenas literales.

Los objetos `String` son inmutables: no se pueden cambiar una vez se crean. Java proporciona otra clase, `StringBuffer`, para crear y manipular cadenas dinámicamente. Las características más relevantes de estas clases (y otras clases básicas para utilizar Java) las veremos más adelante.

Java permite concatenar cadenas de una manera simple utilizando el operador +. El método `cuentaCaracs` utiliza esta característica de Java para imprimir su resultado. Las dos cadenas de los lados son, como ya hemos visto, literales. La de en medio es un entero que se convierte automáticamente a `String` y es concatenado con las otras cadenas.

OBJETOS Y CLASES

4.1 Una breve introducción a las clases

Este es el código correspondiente a una clase llamada `Punto` que representa un punto en el plano:

```
public class Punto {
   public int x = 0;
   public int y = 0;
}
```

Este segmento de código declara una clase llamada `Punto`, que contiene dos variables miembro `x` e `y`. La palabra public que precede las declaraciones de `x` e `y` indica que cualquier otra clase puede acceder a ellas.

Para crear un objeto a partir de una clase como `Punto` es necesario instanciar la clase. Cuando se crea un nuevo objeto `Punto` se reserva espacio para el objeto y para sus miembros `x` e `y`. Además, estas variables son inicializadas a 0, de acuerdo con su declaración.

Consideremos ahora otra clase, `Rectángulo`, que representa un rectángulo en el plano:

```
public class Rectángulo {
   public int base = 0;
   public int altura = 0;
   public Punto origen = new Punto();
}
```

Este segmento de código declara una nueva clase que contiene dos miembros enteros, `base` y `altura`, junto con un tercer miembro, origen, cuyo tipo corresponde a `Punto`. El nombre de la clase `Punto` es utilizada en una declaración de variable como el tipo de la misma: el nombre de una clase puede usarse en los mismos sitios en que se usa un tipo primitivo.

De la misma manera que `base` *es un* entero y `altura` *es un* entero, `origen` *es un* `Punto`. Por otro lado, un `Rectángulo` *tiene un* `Punto`. La diferencia entre *es un* y *tiene un* es crítica porque solamente un objeto que *es un* `Punto` puede ser usado cómo tal.

Como en el caso de `Punto`, cuando se crea un nuevo objeto `Rectángulo` se reserva espacio para el objeto y sus miembros y éstos son inicializados de acuerdo con sus declaraciones. Por tanto, la inicialización de los miembros de `Rectángulo` crea un objeto `Punto`.

Para continuar veamos una implementación un poco más completa de las clases `Punto` y `Rectángulo`:

```
public class Punto {
   public int x = 0;
   public int y = 0;
                              // Un constructor
   public Punto(int x, int y) {
      this.x = x;
      this.y = y;
```

```
   }
}
```

Ahora, para crear un `Punto` es posible proporcionar valores iniciales:

```
new Punto(44, 78)
```

Para la clase `Rectángulo` añadimos métodos para mover el rectángulo y calcular su área y un método específico de finalización:

```
public class Rectángulo {
   public int base = 0;
   public int altura = 0;
   public Punto origen;
                           // Cuatro constructores
   public Rectángulo() {
      origen = new Punto(0, 0);
   }
   public Rectángulo(Punto p) {
      origen = p;
   }
   public Rectángulo(int b, int a) {
      this(new Punto(0, 0), b, a);
   }
   public Rectángulo(Punto p, int b, int a) {
      origen = p;
      base = b;
      altura = a;
   }
                           // Mover el rectángulo
   public void mover(int x, int y) {
      origen.x = x;
      origen.y = y;
   }
                           // Calcular el área
      public int área() {
      return base * altura;
   }
                           // Limpieza
   protected void finalize() throws Throwable {
      origen = null;
      super.finalize();
   }
}
```

Los cuatro constructores permiten diferentes formas de inicialización. Puede crearse un nuevo Rectángulo con valores por defecto para todo o especificar los valores iniciales del origen, la base o la altura.

4.2 El ciclo de vida de un objeto

Un programa Java crea un gran número de objetos de diversas clases. Estos objetos interaccionan entre sí mediante la invocación de sus métodos y el acceso a sus variables. Una vez un objeto ha completado el trabajo para el que fue creado es ***recolectado*** y sus recursos son reciclados para que otros objetos puedan usarlos.

4.2.1 Creación

En Java, un objeto se crea cuando se instancia una clase. Una sentencia común de instanciación es:

```
Rectángulo rect = new Rectángulo();
```

Esta sentencia realiza tres acciones:

- Declaración: `Rectángulo rect` es una declaración de variable que informa al compilador que el nombre `rect` va a ser usado como referencia a un objeto `Rectángulo`.
- Instanciación: `new` es un operador Java que crea un nuevo objeto (reservando memoria para él).
- Inicialización: `Rectángulo()` es una llamada al constructor de `Rectángulo`, que inicializa el objeto.

Para declarar un objeto se siguen las mismas reglas que para declarar una variable utilizando su tipo y su nombre. El operador `new` requiere un solo argumento: una llamada a un constructor. Cada clase Java proporciona un conjunto de constructores para inicializar los objetos de ese tipo. El operador `new` devuelve una referencia al nuevo objeto.

Como se ve en el código de la clase `Rectángulo`, las clases pueden proporcionar uno o más constructores para inicializar un nuevo objeto de ese tipo. Un constructor se caracteriza porque tiene el mismo nombre de la clase y no tiene tipo de retorno. Las declaraciones de los constructores de `Rectángulo` son:

```
public Rectángulo(Punto p)
public Rectángulo(int b, int a)
public Rectángulo(Punto p, int b, int a)
public Rectángulo()
```

Cada uno de estos constructores permite proporcionar valores iniciales para diferentes aspectos del rectángulo: el origen, la base y la altura, los tres o ninguno. Si una clase tiene varios constructores todos ellos deben tener el mismo nombre pero con un número diferente de argumentos o argumentos de diferente tipo. El compilador diferencia los constructores y decide a cuál llamar, dependiendo de los argumentos. De manera que cuando el compilador encuentra la siguiente sentencia, decide que debe llamar al constructor que requiere dos argumentos enteros:

```
Rectángulo rect = new Rectángulo(100, 200);
```

Y cuando encuentra la siguiente línea, decide llamar al constructor que requiere un `Punto`:

```
Rectángulo rect = new Rectángulo(new Punto(44,78));
```

Y este es el constructor que no requiere argumentos:

```
Rectángulo rect = new Rectángulo();
```

Un constructor que no requiere argumentos, como el de arriba, es el constructor por defecto. Si una clase (como `Punto` y `Rectángulo` al comienzo de este capítulo) no define explícitamente ningún constructor, Java proporciona automáticamente un constructor sin argumentos que no hace nada. De esta manera todas las clases tienen al menos un constructor.

4.2.2 Utilización

Una vez un objeto ha sido creado puede ser utilizado para obtener información de él, cambiar su estado o para que realice alguna acción. Hay dos maneras de utilizar un objeto:

- Manipular o inspeccionar sus variables.
- Invocar sus métodos.

La programación orientada a objetos desaconseja, en general, la manipulación directa de las variables de un objeto: cualquier acceso debe hacerse a través de los métodos. Sin embargo, en ciertas situaciones prácticas el acceso directo es una solución recomendable.

Java proporciona un mecanismo de control de acceso de manera que las clases pueden permitir o restringir el acceso a sus variables y métodos. Una clase debe proteger sus variables frente al acceso directo por parte de otros objetos si este acceso puede comprometer su estado interno. En general, si un objeto permite acceder sus variables puede asumirse que pueden ser leídas o escritas sin efectos adversos.

Supongamos un objeto `Rectángulo` que representa un objeto rectangular en un programa de dibujo y que el usuario acaba de moverlo con el ratón a otra posición en la pantalla. Para actualizar el origen del objeto `Rectángulo` existen dos alternativas:

- Manipular directamente la variable `origen`
- Invocar el método `mover`

El miembro `origen` de `Rectángulo` es accesible a otras clases (se ha declarado `public`) de manera que podemos suponer que manipularlo es seguro. Supongamos que hemos creado un `Rectángulo` llamado `rect` como vimos en la sección anterior. Para moverlo a una nueva posición podríamos escribir:

```
rect.origen = new Punto(15, 37);
```

De la misma manera, podemos calcular directamente el área de un `Rectángulo` accediendo directamente a sus miembros `base` y `altura`:

```
a = rect.base * rect.altura;
```

En general, para hacer referencia a una variable de un objeto, se concatena el nombre de la variable detrás de una referencia de objeto, utilizando un punto (el carácter ‘`.`’):

```
referencia.variable
```

`referencia` puede ser cualquier referencia a un objeto, es decir, puede usarse tanto el nombre de una variable como cualquier expresión que retorne un objeto, por ejemplo:

```
a = new Rectángulo().altura;
```

Esta sentencia crea un nuevo `Rectángulo` e inmediatamente accede a su altura. El efecto es obtener la altura por defecto de un `Rectángulo`. Hay que notar que después de ejecutar esta sentencia, el programa no tendrá acceso al `Rectángulo` que ha sido creado porque su referencia no ha sido almacenada en una variable. De manera que el objeto en cuestión se marcará para ser recolectado.

Para mover `rect` a una nueva posición utilizando el método `mover` deberíamos escribir:

```
rect.mover(15, 37);
```

Esta sentencia llama al método `mover` de `rect` con dos parámetros enteros, 15 y 37, que mueve el objeto `rect` dado que asigna nuevos valores a `origen.x` y `origen.y`.

La notación para invocar un método de un objeto es similar a la utilizada para hacer referencia a sus variables. Se concatena el nombre del método detrás de una referencia a objeto utilizando un punto (el carácter '.'). Además, es necesario proporcionar los argumentos que requiere el método entra paréntesis.:

```
referencia.método(listaDeArgumentos);
```

o, si el método no requiere argumentos:

```
referencia.método();
```

De la misma manera que veíamos para las variables, `referencia` puede ser el nombre de un objeto o cualquier expresión que retorne una referencia a un objeto:

```
new Rectángulo(100, 50).área()
```

Algunos métodos, como `área`, retornan un valor. Estos métodos pueden utilizarse en cualquier expresión donde una variable del tipo que retornan pueda ser utilizada:

```
int dobleÁrea = 2 * new Rectángulo(100, 50).área();
```

4.2.3 Eliminación de objetos

Muchos lenguajes orientados a objetos requieren que el usuario lleve la cuenta de los objetos que crea y los elimine cuando ya no sean necesario. Esta técnica de manejar la memoria es tediosa y, a menudo, fuente de errores. Java permite crear todos los objetos necesarios (limitados, por supuesto, por los recursos del sistema) sin necesidad de preocuparse por eliminarlos. El entorno de ejecución Java borra aquéllos objetos cuando detecta que no van a ser usados más. Este proceso recibe el nombre de recolección (*garbage collection*).

Un objeto es marcado para su recolección cuando no existen más referencias a ese objeto. Las referencias contenidas en una variable desaparecen de manera natural cuando se sale del ámbito de la variable. O puede hacerse desaparecer la referencia asignando a la variable el valor `null`.

La máquina virtual Java contiene un recolector (*garbage collector*) que libera periódicamente la memoria utilizada por objetos que no son necesarios. Examina las áreas de memoria y marca aquéllos objetos para los que hay referencias. Una vez todos los posibles caminos de referencias han sido explorados, los objetos no marcados (por tanto, sin referencia) son recolectados.

El recolector es un thread de baja prioridad y puede correr síncrona o asíncronamente dependiendo de la situación del sistema. Se ejecuta de forma síncrona cuando el sistema está escaso de memoria o bajo la petición de un programa Java. Cuando el sistema está inactivo, el recolector trabaja en modo asíncrono, interrumpiéndose en el momento en que otro thread arranca.

Antes de que un objeto sea recolectado tiene una oportunidad de realizar acciones para terminar en un estado limpio a través del método `finalize`. Este proceso se conoce como finalización.

Durante la finalización, un objeto puede liberar recursos del sistema como ficheros abiertos o eliminar referencias a otros objetos de manera que puedan ser más fácilmente recolectados. El método `finalize` de `Rectángulo` elimina la referencia a un objeto `Punto` asignando su variable `origen` a `null`:

```
protected void finalize() throws Throwable {
   origen = null;
   super.finalize();
}
```

El método `finalize` es miembro de la clase `Object`. La clase `Object` es la raíz de la jerarquía de clases Java y el padre de todos los objetos, como veremos en “La clase Object”, página 36. Una clase reescribe el método `finalize` para realizar los mecanismos de finalización necesarios para los objetos de ese tipo. El `finalize` de `Rectángulo` llama a `super.finalize` para darle a la clase padre una oportunidad de realizar también su finalización. En general, un método `finalize` debe contener una invocación de `super.finalize` como última acción.

4.3 Definición de clases

4.3.1 Declaración

El formato general de una declaración de clase en Java es:

```
public
abstract
final
class Nombre
extends Super
implements Interfaces
```

Solamente los campos mostrados en negrita son obligatorios. En cursiva aparecen los campos que no son palabras reservadas del lenguaje.

- `public`: Por defecto, una clase solo puede ser usada por otras clases en el mismo paquete. Este modificador declara que la clase puede ser utilizada por cualquier clase independientemente del paquete al que pertenezca.
- `abstract`: Declara que la clase no puede ser instanciada.
- `final`: Declara que no pueden crearse subclase de esta clase.

- `class`: Indica al compilador que se trata de la declaración de una clase.
- `extends`: Identifica la superclase de la clase, insertando a la clase en la jerarquía.
- `implements`: Declara que la clase implementa una o más interfaces, definidas por una lista de identificadores separados por comas.

4.3.2 El cuerpo de la clase

El cuerpo de la clase contiene todo el código necesario para el ciclo de vida de los objetos de su tipo:

- Declaraciones de las variables que proporcionan el estado de la clase y sus objetos.
- Constructores para inicializar nuevos objetos.
- Métodos para implementar el comportamiento de la clase y sus objetos.
- Cuando sea necesario, un método `finalize`.

4.3.3 Constructores

El cuerpo de un constructor es como el cuerpo de un método. Contiene declaraciones de variables locales, bucles y otras sentencias. Muy a menudo un constructor necesita aprovechar el código de inicialización de su superclase. De hecho, algunas clases deben llamar a un constructor de su superclase para trabajar correctamente. Si está presente, el constructor de la superclase debe ser la primera sentencia en un constructor: un objeto debe realizar las inicializaciones de nivel más alto antes. Un constructor de la superclase se llama mediante:

```
super (parametros);
```

Cuando se declaran los constructores de una clase se pueden utilizar especificadores de acceso en la declaración del constructor para definir qué otros objetos pueden crear instancias de la clase:

- `private`: Ninguna otra clase puede instanciar esta clase. La clase puede contener métodos de clase públicos (llamados a veces métodos *factory*) y esos métodos pueden construir un objeto de la clase y retornarlo.
- `protected`: Solamente las subclases de la clase pueden crear instancias de ella.
- `public`: Cualquier clase puede crear una instancia de esta clase.
- package: Sólo las clases dentro del mismo paquete pueden construir instancias de esta clase.

4.3.4 Variables

Las declaraciones de variable permiten especificar no sólo el tipo y nombre de las mismas, sino otros atributos como su nivel de acceso, si trata de variables de instancia o de clase, y si la variable es constante. Los componentes de una declaración de variable son:

- Nivel de acceso: `public`, `protected`, `package` o `private`.
- `static`: Declara esta variable como una variable de clase.
- `final`: Indica que el valor de la variable no puede cambiar. La convención es que los nombres de las constantes se escriban en mayúsculas.

4.3.5 Métodos

Como mínimo, la declaración de un método debe contener su nombre y un tipo de retorno:

```
Tipo nombre() {
                    . . .
}
```

Si el método no retorna ningún valor, debe usarse la palabra `void` como tipo de retorno.

La declaración de un método puede incluir también la siguiente información:

- Nivel de acceso: `public`, `package`, `protected` o `private`.
- `static`: Declara el método como un método de clase.
- `abstract`: Un método abstracto no tiene implementación y sólo puede ser miembro de una clase abstracta.
- `final`: Es un método que no puede ser reescrito por las subclases.
- `native`: El método está implementado en un lenguaje que no es Java.
- `synchronized`: Cuando se ejecutan varios threads simultáneamente pueden hacerse llamadas a métodos que trabajan sobre los mismos datos. Estos métodos pueden declararse synchronized para evitar problemas de concurrencia.
- `throws` *`Excepciones`*: Si el método lanza excepciones, debe aparecer una lista de ellas separadas por comas, como veremos en “Excepciones lanzadas por un método”, página 57.
- *`Parámetros`*: Es la lista de parámetros que el método requiere al ser invocado. Es importante notar que, en Java, todo el paso de parámetros se realiza por valor. Cuando se trata de tipos primitivos, esto significa que el método no puede cambiar el valor de la variable que se le pasa. Cuando se trata de tipos de referencia, el método no puede cambiar la referencia que contienen, pero sí invocar a los métodos que ofrecen para cambiar el estado de los objetos referenciados.

Java permite la ***sobrecarga*** de los nombres de los métodos, de manera que el mismo nombre puede ser usado para diferentes métodos. Los métodos sobrecargados se diferencian por el número y tipo de los parámetros que se les pasan. No se puede declarar más de un método con el mismo nombre y el mismo número y tipo de parámetros para una misma clase.

Cuando una clase reescribe un método de su superclase, el método reescrito debe tener el mismo nombre, tipo de retorno y tipo y número de parámetros que el de la superclase.

Dentro del cuerpo de un método puede hacerse referencia a las variables miembro del objeto sin más que usar su nombre. Sin embargo, hay veces en la que es necesario eliminar la ambigüedad que se produce cuando una variable miembro y un parámetro del método tienen el mismo nombre. Por ejemplo, en el siguiente constructor sus argumentos tienen los mismos nombres que las variables de instancia que se inicializan con ellas:

```
class Punto {
   int x, y;
   Punto (int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Los nombres de argumento tienen mayor precedencia que los de las variables miembro con el mismo nombre. Para acceder a ellas se utiliza la referencia al objeto actual (usando `this`) explícitamente.

Si las variables o métodos de la superclase son reescritos por la clase, un método puede hacer referencia a cualquier de ellos utilizando `super`. Consideremos la clase:

```
class UnaClase {
   boolean unaVariable;
   void unMetodo() {
      unaVariable = true;
   }
}
```

y su subclase, que oculta `unaVariable` y reescribe `unMetodo`:

```
class OtraClase extends UnaClase {
   boolean unaVariable;
   void unMetodo() {
      unaVariable = false;
      super.unMetodo();
      System.out.println(unaVariable);
      System.out.println(super.unaVariable);
   }
}
```

En primer lugar, `unMetodo` pone `unaVariable` (la declarada en `OtraClase` que esconde la declarada en `UnaClase`) a `false`. Después invoca a `unMetodo` en la superclase con:

```
super.unMetodo();
```

El resultado que aparecerá será, por tanto, el valor de las dos variables (el de la clase y el de su superclase):

```
false
true
```

4.3.6 Miembros de instancia y miembros de clase

Cada vez que se crea una instancia de una clase, la máquina virtual Java crea una copia de cada una de sus variables de instancia, que son propias de esa instancia en concreto.

Por el contrario, para las variables de clase (declaradas con el modificador `static`) solamente se crea una copia, común para toda la clase. El sistema reserva memoria para estas variables una sola vez, la primera vez que encuentra la clase. Todas las instancias comparten la misma copia de las variables de clase. Las variables de clase pueden accederse a través de una instancia o a través de la clase misma.

El mecanismo es similar para los métodos. Los métodos de instancia pueden operar con las variables de instancia del objeto actual pero también tienen acceso a las variables de clase. Los métodos de clase, por otro lado, no pueden acceder a las variables de instancia (a menos que creen un nuevo objeto y las accedan a través de él), pero pueden ser invocados por la clase misma: no es necesaria una instancia de la clase para llamar a un método de clase.

4.3.7 La clase `Object`

La clase `Object` se sitúa en la raíz de la jerarquía de clases de la máquina Java. Cualquier clase de un sistema Java es descendiente, directo o indirecto de la clase `Object`. Esta clase define el estado y comportamiento básicos que todos los objetos deben tener, como la habilidad de compararse entre sí, convertir su contenido a una cadena de caracteres o retornar la clase de la que son instancia.

Una clase puede reescribir los siguientes métodos de la clase `Object`:

- `clone`: Permite crear objetos a partir de objetos del mismo tipo.
- `equals`: Compara dos objetos para comprobar si son iguales.
- `finalize`
- `toString`: Convierte el contenido de un objeto a una cadena de caracteres.

Object, además, contiene los siguientes métodos finales:

- `getClass`: Retorna una referencia a la clase del objeto.
- `notify`: Este método y los dos siguientes están relacionados con la sincronización de múltiples threads.
- `notifyAll`
- `wait`
- `hashCode`: Este método permite asignar un identificador único dentro del sistema al objeto.

4.3.8 Clases y métodos abstractos

A veces es necesario modelar un concepto sin necesidad de crear instancias del mismo. Por ejemplo, la clase `Number` en el paquete java.lang representa el concepto abstracto de un número. Tiene sentido modelar números en un programa, pero no tiene mucho el crear objetos que sean números en sentido general. En lugar de eso, la clase `Number` es únicamente una superclase para clases como `Integer` y `Float`, que implementan clases específicas de números. Una clase como `Number`, que representa un concepto abstracto que no debe ser instanciado, se denomina una ***clase abstracta.*** Una clase abstracta solo puede ser extendida, nunca instanciada.

Una clase abstracta puede contener métodos abstractos, es decir, métodos que no tienen implementación. De esta manera, una clase abstracta puede definir un interface de programación, proporcionando a sus subclases declaraciones para todos los métodos necesarios para implementar ese interface. Y algunos de los detalles de implementación de los métodos se dejan a las subclases.

Para poner un ejemplo, supongamos una aplicación de dibujo. En ella se podrán dibujar diferentes clases de gráficos: rectas, círculos, etc. Cada uno de estos objetos tiene ciertas características (posición, color) y comportamiento (moverse, representarlos) que deben cumplir. Podemos declarar que todos heredan estos atributos de la mima clase: `ObjetoGrafico`.

La clase abstracta `ObjetoGrafico` proporciona variables y métodos que son compartidos en todas las subclases, como la posición del origen y el método `moverA`. `ObjetoGrafico` también define métodos abstractos, como `dibujar`, que deben ser implementados por todas las subclases pero de una manera diferente por cada una (no tiene sentido proporcionar una por defecto en la superclase). `ObjetoGrafico` sería algo como:

```
abstract class ObjetoGrafico {
   int x, y;
   . . .
   void moverA(int nX, int nY) {
      . . .
   }
   abstract void dibujar();
}
```

Cualquier clase no abstracta de `ObjetoGrafico`, como `Circulo` y `Rectangulo`, deben proporcionar una implementación de `dibujar`.

```
class Circulo extends ObjetoGrafico {
   void dibujar() {
      . . .
   }
}
class Rectangulo extends ObjetoGrafico {
   void dibujar() {
      . . .
   }
}
```

No se requiere que una clase abstracta tenga métodos abstractos. Pero cualquier clase que tenga un método abstracto o no proporcione una implementación para todos los métodos abstractos definidos en sus superclases debe ser declarada abstracta.

4.4 Interfaces

Un interface Java define un conjunto de métodos pero no los implementa. Una clase que implementa el interface declara que implementa todos los métodos definidos en él, comprometiéndose así a un cierto comportamiento. Los interfaces pueden definir también constantes.

Consideremos una clase como `Despertador`, que proporciona un servicio determinado a otras clases, notificando a aquellos objetos que así lo indiquen cuando ha transcurrido un cierto período de tiempo.

Para ser incluido en la lista de clientes de `Despertador`, un objeto debe hacer dos cosas:

- Pedir al despertador cuándo quiere ser avisado.
- Implementar el método `ring`.

Para satisfacer el primer requisito, un objeto llama al método `avisarDentroDe` de `Despertador`, que podría ser implementado de la siguiente manera:

```
public synchronized boolean avisarDentroDe(Cliente cl, long t) {
   int id = proximoCliente();
   if (id == NOHAYSITIO) {
      return false;
   }
   else {
      clientes [id] = cl;
      intervalo[id] = t;
      new ThreadReloj(id).start();
      return true;
   }
}
```

Si `Despertador` tiene sitio para el cliente, lo registra y arranca un `ThreadReloj` para él. Una vez el intervalo especificado ha transcurrido invocará el método `ring` del objeto `cl`.

Esto nos lleva al segundo requisito. Un objeto que quiera usar `Despertador` debe implementar el método `ring`. Este requisito se exige mediante el tipo del objeto que se registra. El primer argumento de `avisarDentroDe` es el objeto que quiere usar el servicio. Su tipo es `Cliente`, que es el nombre de un interface:

```
public interface Cliente {
   public void ring();
   public final long UN_SEGUNDO = 1000; // en milisegundos
   public final long UN_MINUTO = 60000;
}
```

`Cliente` define `ring` pero no lo implementa. También define dos constantes que pueden ser útiles. Las clases que implementen este interface “heredan” las constantes y deben implementar `ring`.

Cualquier objeto que es un `Cliente` (y que puede usarse por tanto en una llamada a `avisarDentroDe`) implementa este interface. Esto significa que implementa todos los métodos definidos en `Cliente` y por tanto el método `ring`, de manera que se satisface el segundo requisito.

Por ejemplo, el siguiente segmento de código corresponde a un applet (hablaremos con detalle de los applets en “APPLETS”, página 73) que utiliza un Despertador para actualizar la hora cada minuto:

```
class AppReloj extends Applet implements Cliente {
   . . .
   public void ring() {
      repaint();
      desp.avisarDentroDe(this, UN_MINUTO); } }
```

4.4.1 Interfaces, clases abstractas y herencia múltiple

En este punto, podría pensarse que no hay mucha diferencia entre un interface y una clase abstracta. Un interface es una lista de métodos no implementados (y, por tanto, abstractos). Parece que la siguiente clase abstracta sería equivalente a lo que hemos visto:

```
abstract class Cliente {
   public abstract void ring();
}
```

Pero no es así. Si `Cliente` es una clase abstracta, todos los objetos que quieran usar a `Despertador` deben ser instancias de una clase que se derive de `Cliente`. Sin embargo, muchos objetos que quieran usar `Despertador` ya tienen una superclase. Por ejemplo, `AppReloj` es un `Applet`: debe ser un applet para poder ser ejecutado en un browser de Web. Y Java no soporta la herencia múltiple, de manera que `AppReloj` no puede ser a la vez un `Applet` y un `Cliente`. Por tanto, debemos usar un interface.

Puesto en otros términos: `Despertador` no debe requerir que sus usuarios tengan una relación específica de clase. La clase que tengan no importa, solamente es importante que implementen un método determinado.

A veces se dice que los interfaces son una alternativa para la herencia múltiple. Aunque puede resolver problemas similares, la herencia múltiple y el uso de interfaces se diferencian en varios aspectos:

- Una clase únicamente hereda constantes de un interface.
- Una clase no puede heredar implementaciones de métodos de un interface.
- La jerarquía de interfaces es totalmente independiente de la jerarquía de clases. Clases que implementan el mismo interface pueden o no estar relacionadas a través de la jerarquía de clases.

Esto nos lleva a otro uso habitual que se da a los interfaces: su empleo para controlar si se desea que se active una cierta característica que ya está disponible en la superclase. Estas características están incluidas en métodos que, al principio de su cuerpo, comprueban si la clase del objeto implementa el interface en cuestión. Si es así, el método se activa. Si no es así, el método no hace nada o lanza una excepción. Este es el caso, por ejemplo, del interface `Cloneable` y el método `clone()` de la clase `Object`.

4.5 Clases anidadas

Desde el JDK 1.1, Java permite definir una clase directamente como miembro de otra clase:

```
class Envoltorio {
   . . .
   class Envuelta {
      . . .
   }
}
```

Una clase definida dentro de otra clase se denomina una clase *anidada*. Solamente deben definirse clases anidadas cuando únicamente tengan sentido en el interior de las clases que las definen o cuando estrictamente necesiten los privilegios que tienen las clases anidadas sobre las que las incluyen. Por ejemplo, un cursor que marca el lugar dónde se escribe el texto tendría sentido que fuera una clase anidada de una clase que implementara una ventana de manejo de texto.

Como miembro de la clase que la contiene, una clase anidada goza de un privilegio muy especial: tiene acceso total a todos los demás miembros de la clase que la contiene. Esto es consistente con el modelo de acceso que ofrece Java, dado que la clase anidada está dentro (con todas sus consecuencias) de la clase en la que se define.

Las clases anidadas son miembros de la clase que las contiene y, por tanto, pueden ser estáticas o de instancia. Una clase anidada estática solamente puede acceder a los miembros estáticos de la clase que la contiene. Por su parte, una clase anidada de instancia está asociada con una instancia de la clase y tiene acceso a los miembros de esa instancia. Una clase anidada de instancia recibe el nombre de clase *interior*.

Es importante notar que el término “clase anidada” hacer referencia a un matiz puramente sintáctico: la definición de la clase se realiza dentro de la definición de otra clase. Sin embargo, el término “clase interior” tiene un matiz semántico, dado que hace referencia a la relación entre las instancias de dos clases: una instancia de una clase interior puede existir solamente dentro de una instancia de la clase que la contiene.

Una clase anidada puede ser declarada dentro de cualquier bloque de código. Una clase anidada declarada dentro de un método o de un bloque más pequeño tiene acceso a todas las variables locales dentro del ámbito en que se ha declarado.

4.6 Paquetes

Java ofrece un mecanismo para agrupar en ***paquetes*** grupos de clases relacionadas, de manera que sea más cómodo su uso, se eviten conflictos de nombres y se facilite el control de acceso. Un paquete es una colección de clases e interfaces relacionados que proporciona protección de acceso y gestión del espacio de nombres.

Las clases e interfaces que son parte del JDK están contenidos en diversos paquetes que las agrupan por su funcionalidad: las clases relacionadas con applets están en `java.applet`, las clases de entrada/salida están en `java.io` y las relacionadas con el diseño de interfaces gráficos en `java.awt`.

Retomemos el ejemplo anterior de la clase abstracta `ObjetoGrafico` y sus subclases, junto con un interface llamado `Movil` que implementan si es posible trasladarlos con el ratón:

```
abstract class ObjetoGrafico {
   . . .
}
class Circulo extends ObjetoGrafico implements Movil {
   . . .
}
class Rectangulo extends ObjetoGrafico implements Movil {
   . . .
}
interface Movil {
   . . .
}
```

Es recomendable agrupar estas clases y el interface en un paquete por varias razones:

- De esta manera es fácil determinar que estas clases y el interface están relacionados.
- Así es sencillo encontrar las clases e interfaces relacionadas con objetos gráficos.
- Los nombres de las clases no entrarán en conflicto con los de otros paquetes, ya que cada paquete crea un nuevo espacio de nombres.
- Las clases dentro del paquete pueden tener acceso total entre sí, y limitar el acceso de otras clases de manera selectiva.

4.6.1 Creación de paquetes

Para crear un paquete, simplemente se coloca una sentencia `package` al principio de los ficheros fuente de las clases y los interfaces que lo compongan. Por ejemplo, si el siguiente código aparece en el fichero “Circulo.java”, la clase `Circulo` queda integrada en el paquete `graficos`:

```
package graficos;
class Circulo extends ObjetoGrafico implements Movil {
   . . .
}
```

El ámbito de una sentencia `package` es todo el fichero fuente, de manera que todas las clases e interfaces definidas en él pasan a formar parte del paquete. Si no aparece una sentencia `package` al comienzo del fichero fuente, las clases que contienen son incluidas en un paquete por defecto, que no tienen nombre. En términos generales, el paquete por defecto debe usarse solamente para aplicaciones pequeñas o temporales o al comienzo de un desarrollo. Es recomendable que clases e interfaces pertenezcan a un paquete.

Es más que probable que dos (o más) programadores asignen el mismo nombre a clases diferentes y que éstas tengan que convivir en una misma aplicación en un momento dado. Esto es posible siempre que las clases pertenezcan a paquetes distintos, dado que el nombre completo de una clase incluye el nombre del paquete al que pertenece.

Esto funciona bien siempre que tampoco haya colisiones en el nombre de los paquetes. Para evitar esto se utiliza la siguiente convención: Los paquetes desarrollados por una organización deben comenzar su nombre por el nombre invertido del dominio Internet de la organización. Así, el paquete gráfico de nuestro ejemplo, desarrollado en el CICA (cuyo nombre de dominio Internet es `cica.es`), tendría el identificador:

```
es.cica.graficos
```

Y este es el nombre que debería aparecer en las sentencias `package` del código fuente de cada una de las clases que lo componen.

4.6.2 Uso de los paquetes

Solamente los miembros públicos de un paquete son accesibles desde fuera del mismo. Para utilizar un miembro público de un paquete desde fuera de él, hay tres alternativas:

- Hacer referencia al elemento con su nombre completo, incluyendo el nombre del paquete.
- Importar el elemento.
- Importar el paquete completo.

Para utilizar la clase `Rectangulo` que definíamos antes en una clase que no pertenezca a su paquete, podemos referenciarla usando su nombre largo en cualquier lugar donde deba aparecer el nombre de la clase:

```
es.cica.graficos.Rectangulo rect = new es.cica.graficos.Rectangulo();
```

Esto es práctico para usos muy concretos. No resulta cómodo tener que escribir `es.cica.graficos.Rectangulo` una y otra vez. Además, el código se vuelve más difícil de leer y propenso a errores.

Para importar un elemento específico de un paquete en un fichero fuente, basta colocar una sentencia `import` al comienzo del fichero (pero siempre después de la sentencia `package`, si se usa). Para importar el elemento `Circulo` haríamos:

```
import es.cica.graficos.Circulo;
```

Ya podemos hacer referencia a Circulo usando su nombre sin cualificar:

```
Circulo circ = new Circulo();
```

Este mecanismo funciona bien si se requieren unos pocos elementos del paquete. Pero si se usan muchas clases, lo interesante es importar el paquete completo. Para ello, se utiliza la sentencia import con un asterisco (el carácter '`*`'). Para importar todas las clases e interfaces de nuestro ejemplo:

```
import es.cica.graficos.*;
```

Ahora podemos hacer referencia a cualquier clase o interface del paquete usando su nombre sin cualificar:

```
Circulo circ = new Circulo();
Rectangulo rect = new Rectangulo();
```

El asterisco en una sentencia import se puede usar solamente para especificar todas las clases en un determinado paquete. No es posible utilizarlo para seleccionar un subconjunto de las clases de un paquete. Por ejemplo, la siguiente sentencia no se puede usar para importar las clases del paquete cuyo nombre empiece por 'A':

```
import es.cica.graficos.A*; // Da un error de compilación
```

Por defecto, la máquina Java importa automáticamente siempre tres paquetes:

- El paquete por defecto (sin nombre).
- El paquete `java.lang`.
- El paquete al que pertenezca la clase con la que es está trabajando.

4.6.3 Paquetes y nombres de ficheros. El `CLASSPATH`

Aunque la especificación del lenguaje no lo requiere explícitamente, el JDK se basa en la estructura jerárquica del sistema de ficheros para relacionar ficheros fuente y de clases con los nombres de paquete a los que pertenecen.

El código fuente para una clase o interface debe almacenarse en un fichero con extensión ".java" y cuyo nombre corresponda con el nombre de la clase. Este fichero debe encontrarse en un directorio cuyo nombre refleje el paquete al que la clase o interface pertenece. Por ejemplo, el fichero fuente de la clase Rectangulo debe estar en un fichero llamado "Rectangulo.java" y colocado de acuerdo a la siguiente jerarquía de directorios:

```
es/cica/graficos/Rectangulo.java
```

El directorio origen de esta jerarquía ("es", en este caso) puede estar en cualquier parte del sistema de ficheros. Lo esencial es que exista un relación directa entre los componentes del nombre cualificado de la clase y los directorios que definen donde se encuentra el fichero fuente.

Cuando se compila el fichero fuente, el compilador produce un fichero para cada una de las clases o interfaces definidas en él. El nombre es el de la clase o interface en cuestión, seguido del sufijo ".class". De la misma manera que un fichero ".java", un fichero ".class" debe estar incluido en una jerarquía de directorios que refleje el paquete al que pertenece. Lo que es necesario es que residan en el mismo árbol, de manera que puedan distribuirse paquetes en formato ejecutable, sin necesidad de proporcionar también los fuentes.

Tanto a la hora de compilar una aplicación como a la hora de ejecutarla, el compilador o la máquina virtual Java necesitan un mecanismo para encontrar las clases en uso, de acuerdo con los paquetes a que pertenezcan. Esta es la razón por la que se usa este esquema de almacenamiento en directorios, subdirectorios y ficheros.

El compilador y el intérprete usan una variable interna (el `CLASSPATH`) para determinar a partir de qué punto o puntos en la jerarquía de ficheros deben comenzar a buscar las clases. El JDK soporta también el uso de ficheros comprimidos (en formato ***zip***), así como de ficheros

con un formato específico, llamado *jar*, conteniendo toda una jerarquía de directorios. De esta manera, el `CLASSPATH` es una lista de directorios y ficheros zip o jar en los que se buscarán las clases.

Para establecer el `CLASSPATH` hay dos mecanismos:

- Definir la variable de entorno `CLASSPATH`.
- Utilizar la opción `-classpath` a la hora de invocar al compilador o al entorno de ejecución.

Definiciones típicas para el `CLASSPATH` son:

Unix	`.:~/classes:/JDK/lib/classes.zip`
Windows 95/NT	`.;C:\classes;C:\JDK\lib\classes.zip`

ALGUNAS CLASES ESENCIALES

5.1 El entorno de ejecución de un programa Java

Cualquier programa, cuando se ejecuta, trabaja en un entorno que determina su comportamiento. Por un lado, el creador del programa suele ofrecer opciones para configurarlo y adecuar su funcionamiento y la apariencia del interface de usuario. Estas opciones de configuración pueden aplicarse de forma persistente o de manera específica en cada una de las invocaciones del programa. Por otro lado, el programa se ejecuta para unos determinados atributos del sistema en el que corre: tipo de máquina, sistema operativo, directorio de trabajo, etc. En este apartado vamos a ver los mecanismos que ofrece Java para que un programa utilice estos elementos que configuran su comportamiento.

5.1.1 Configuración de un programa Java. Propiedades

Java ofrece una clase específica (`java.util.Properties`) para definir los atributos permamentes que debe tener un programa a la hora de ser ejecutado. Esta clase maneja un conjunto de pares clave/valor, cada uno de los cuales se puede considerar que corresponde a un atributo que configura el programa. En concreto, la clase `Properties` puede emplearse para:

- Cargar los pares clave/valor en un objeto `Properties` desde un *stream*.
- Obtener el valor asociado a una clave.
- Obtener una lista de las claves y sus valores.
- Guardar los pares clave/valor a un *stream*.

El siguiente trozo de código ilustra cómo un programa puede leer las propiedades almacenadas en un fichero para acceder a su configuración. Esto, típicamente, se haría al comienzo de la ejecución:

```
Properties prProps = new Properties();
FileInputStream pin = new FileInputStream("Propiedades");
prProps.load(pin);
pin.close();
```

Y en el siguiente ejemplo vemos cómo un programa puede guardar su configuración (para una posterior ejecución, por ejemplo) por medio de un objeto `Properties`:

```
FileOutputStream pout = new FileOutputStream("ultimaEjecucion");
prProps.save(out, "Propiedades de la ultima ejecución");
pout.close();
```

El método `save()` requiere un *stream* al que escribir, junto con una cadena que se usa como comentario al comienzo del fichero. Por supuesto, los ficheros de propiedades pueden ser manipulados directamente, dado que su aspecto es como el que sigue:

```
# @(#)javac.properties  1.15

javac.err.internal=Internal error.
javac.err.eof.in.comment=Comment not terminated at end of input.
javac.err.eof.in.string=String not terminated at end of input.
javac.err.unbalanced.paren=Unbalanced parentheses.
```

Una vez se dipone de un objeto Properties es posible consultarlo por medio de alguno de los siguientes métodos:

- `getProperty(String key)`, `getProperty(String key, String default)`. Esta segunda versión retorna el valor por defecto si la clave no es encontrada.
- `list(PrintStream s)`, `list(PrintWriter w)`.
- `propertyNames()` retorna una `Enumeration` que contiene las claves disponibles.
- `size()` retorna el número de pares clave/valor disponibles.

Si como consecuencia de la interacción del programa con el usuario o con el entorno hay que cambiar alguna de las propiedades la clase Properties ofrece los siguientes métodos para su modificación:

- `put(Object key, Object value)` incluye el par clave/valor.
- `remove(Object key)` elimina el par clave/valor correspondiente a la clave.

En ambos casos se utilizan objetos de clase `Object` porque son métodos heredados de la clase `Hashtable` (de la que se deriva `Properties`), aunque deben usarse únicamente objetos de clase `String`.

5.1.2 Invocando un programa Java. Argumentos

Una aplicación Java es capaz de aceptar cualquier número de argumentos desde la línea de comandos. Los argumentos se proporcionan cuando se invoca a la aplicación, detrás del nombre de la clase que se va a ejecutar. Supongamos que tenemos una aplicación Java llamada `Sort`, que ordena las líneas de un fichero. Para ordenar las líneas en el fichero “agenda.txt”, la invocación sería del tipo:

```
java Sort agenda.txt
```

Cuando se invoca una aplicación, la máquina virtual Java pasa los argumentos de la línea de comandos al método `main` de la clase principal por medio de un array de `String`. Cada `String` en el array contiene uno de los argumentos. En el ejemplo anterior, los argumentos que recibe el método `main` de la clase `Sort` es un array que contiene un solo `String`: “agenda.txt”.

A continuación tenemos una aplicación simple que presenta cada uno de sus argumentos en una línea separada:

```
public class Eco {
   public static void main (String[] args) {
      for (int i = 0; i < args.length; i++)
```

```
            System.out.println(args[i]);
    }
}
```

Y esto es un ejemplo de cómo se invocaría la aplicación y el resultado que se obtendría:

```
java Eco eco ecoo ecooo
eco
ecoo
ecooo
```

Si un programa necesita utilizar argumentos numéricos, debe convertir los `String` que recibe, como “34”, a su valor numérico, como en el ejemplo siguiente:

```
int arg1;
if (args.length > 0) arg1 = Integer.parseInt(args[0]);
```

Todas las clases derivadas de la clase abstracta `Number`, como `Integer`, `Float` o `Double` tienen métodos (de clase) del tipo `parseXXX` para convertir una representación en forma de `String` a un objeto de su tipo.

5.1.3 El entorno del sistema. La clase `System`

La clase `System` mantiene un conjunto de propiedades que definen los atributos del entorno de trabajo en el que se desarrolla una aplicación Java. Cuando la máquina virtual comienza su ejecución, las propiedades del sistema se inicializan para contener la información correspondiente al entorno actual. La tabla siguiente resume la clave y el contenido de estas propiedades:

Clave	Contenido
`file.separator`	Separador entre ficheros (p.ej: ‘`/`’)
`java.class.path`	Valor del `CLASSPATH`
`java.class.version`	Versión de las clases en `java.*`
`java.home`	Directorio de instalación de Java
`java.vendor`	Identificador del fabricante de la JVM
`java.vendor.url`	URL del fabricante
`java.version`	Versión de la JVM
`line.separator`	Separador de líneas en un fichero
`os.arch`	Arquitectura del sistema operativo
`os.name`	Nombre del sistema operativo
`os.version`	Versión del sistema operativo
`path.separator`	Separador entre componentes de un path (p.ej: ‘`:`’)

Clave	Contenido
`user.dir`	Directorio de trabajo actual
`user.home`	Directorio por defecto del usuario
`user.name`	Nombre del usuario

La clase `System` proporciona dos métodos para leer las propiedades del entorno de ejecución: `getProperty` y `getProperties`. Las propiedades se pueden modificar utilizando el método `setProperties`.

El método `setProperties` cambia el conjunto de propiedades del sistema únicamente para la aplicación en la que se invoque. Estos cambios no son persistentes. Es decir, el cambio de las propiedades del sistema en la ejecución de una aplicación no afecta a posteriores ejecuciones de la misma o diferentes aplicaciones. La JVM reinicializa los valores de las propiedades del sistema cada vez que arranca.

5.2 Entrada y salida estándar

El concepto de entrada y salida estándar tiene su origen en las librerías de programación de C y ha sido asimilado al entorno Java. Como en C, hay tres *streams* estándar, a los que se accede a través de la clase `java.lang.System`:

- La entrada estándar, referenciada por `System.in`. Se utiliza para proporcionar entradas al programa, típicamente la proporcionada por el usuario.
- La salida estándar, referenciada por `System.out`. Se utiliza para proporcionar la salida del programa al usuario.
- La salida estándar de errores, referenciada por `System.err`. Se utiliza para presentar mensajes de error al usuario.

Probablemente los elementos más utilizados de la clase `System` son los streams de salida y de errores, dado que se utilizan para presentar texto al usuario. Ambos derivan de la clase `PrintStream`, de manera que los métodos que se pueden utilizar para proporcionar la salida son los de esta clase: `print`, `println` y `write`.

`print` y `println` son esencialmente iguales: ambos escriben su argumento (un `String`) en el stream. La única diferencia entre ellos es que `println` añade un carácter de salto de línea al final de la salida y `print` no lo hace. Es decir, esto:

```
System.out.print(“Hola, mundo\n”);
```

es equivalente a esto:

```
System.out.println(“Hola, mundo”);
```

5.3 Las clases `String` y `StringBuffer`

El paquete java.lang contiene dos clases para manipular cadenas de caracteres: `String` y `StringBuffer`. Ya hemos visto la clase `String` varias veces y se utiliza para almacenar cadenas de caracteres que no van a cambiar. La clase `StringBuffer` se utiliza, precisamente, cuando es necesario manipular el contenido de la cadena.

El método `invertir` en el siguiente segmento de código utiliza `String` y `StringBuffer` para invertir los caracteres de una cadena. Dada una lista de palabras, este método se puede utilizar en conjunción con un método de ordenación para crear una lista de palabras que rimen: una lista de palabras ordenadas de acuerdo con sus sílabas finales. Para eso basta invertir las cadenas, ordenar la lista e invertir las cadenas de nuevo.

```
public class CadenaInversa {
   public static String invertir(String src) {
      int i, lon = src.length();
      StringBuffer dest = new StringBuffer(lon);
      for (i = (lon - 1); i >= 0; i--) dest.append(src.charAt(i));
      return dest.toString();
   }
}
```

El método `invertir` acepta como argumento el `String src` que contiene la cadena que va a ser invertida. Crea un `StringBuffer`, `dest`, del mismo tamaño que la cadena origen. A continuación explora la cadena origen en sentido descendente, añadiendo esos caracteres a `dest` en sentido ascendente. Finalmente, antes de retornar se convierte `dest` a `String`.

5.3.1 Conversiones entre objetos y `String`

Muy a menudo se necesita convertir objetos a `String` para pasarlos como parámetros a métodos que aceptan valores `String`. El ejemplo más habitual ya lo hemos visto aquí bastantes veces: el uso de `System.out.println` para proporcionar información al usuario.

Esta es el motivo de que todas las clases hereden el método `toString` de la clase `Object`. Muchas clases en el paquete `java.lang` reescriben este método para proporcionar una implementación significativa para esa clase. Por ejemplo, las clases conocidas como ***wrappers*** (aquéllas que se corresponden con los tipos primitivos del lenguaje), como `Character`, `Integer` o `Boolean`.

A su vez, la clase `String` proporciona el método de clase `valueOf`. Este método puede usarse para convertir variables de diferentes tipos a `String`. Por ejemplo, el valor del número π se puede obtener así:

```
System.out.println(String.valueOf(Math.PI));
```

La clase `String` no proporciona ningún método para convertir objetos `String` a números. Sin embargo, las clases numéricas si tienen métodos (como vimos hace poco) para realizar estas conversiones.

5.4 Lectura y escritura

Una de las tareas más comunes en una aplicación es la de intercambiar información con el exterior de la mima. La información se debe poder intercambiar con diferentes elementos: ficheros en el disco, servidores o clientes en la red, directamente en memoria, o con otro programa. Y, por supuesto, puede ser de cualquier tipo: objetos, caracteres, imágenes, sonidos...

Para recoger información, un programa abre un *stream* en una fuente de información (un fichero, un área de memoria, un ***socket***,...) y lee la información a través de él. De manera similar, un programa envía información a un destino abriendo un stream en él y escribiendo la información a través del stream.

El paquete `java.io` contiene una colección de clases de stream que soportan diferentes algoritmos para leer y escribir. Estas clases están divididas en dos jerarquías basadas en el tipo básico de datos (caracteres o bytes) sobre el que operan. En lo que sigue veremos estas clases, de acuerdo con el propósito de cada una de ellas y la jerarquía en la que están incluidas.

5.4.1 Streams de caracteres y streams de bytes

`Reader` y `Writer` son las superclases abstractas para los streams basados en caracteres definidos en `java.io`. `Reader` define el API y una implementación parcial para ***lectores*** (streams que leen caracteres de 16 bits) y `Writer` lo hace para ***escritores*** (streams que escriben caracteres de 16 bits).

Las subclases de `Reader` y `Writer` implementan streams especializados y podemos dividirlos en dos categorías: aquéllos que leen o escriben directamente en la fuente o destino de los datos y los que realizan algún tipo de procesamiento.

Un programa debe utilizar los streams basados en bytes, descendientes de `InputStream` y `OutputStream`, para leer y escribir bytes de 8 bits. `InputStream` y `OutputStream` definen el API y una implementación parcial para los ***streams de entrada*** (capaces de leer bytes de 8 bits) y los ***streams de salida*** (capaces de escribir bytes de 8 bits). Estos streams son usados primordialmente para leer y escribir datos binarios como imágenes y sonidos, y para conexiones a través de la red.

Como en el caso de `Reader` y `Writer`, las subclases de `InputStream` y `OutptStream` proporcionan funciones especializadas de E/S que se pueden clasificar en dos categorías: streams directos y streams con proceso.

5.4.2 Streams directos

Los streams directos leen o escriben de repositorios especializados de datos como cadenas, ficheros o ***pipes***. Habitualmente, para cada lector o stream de entrada orientado a un tipo específico de fuente de datos, `java.io` contiene un escritor o stream de salida paralelo que puede escribir en él y crearlo.

- **`CharArrayReader` y `CharArrayWriter`. `ByteArrayInputStream` y `ByteArrayOutputStream`**: Permiten leer y escribir en memoria. Estos streams se crean sobre un array existente y sus métodos de lectura y escritura permiten acceder

directamente al array.

- **`FileReader` y `FileWriter`. `FileInputStream` y `FileOutputStream`**: En general, son llamados streams de ficheros y se utilizan para leer o escribir en ficheros.
- **`PipedReader` y `PipedWriter`. `PipedInputStream` y `PipedOutputStream`**: Implementan los componentes de entrada y salida de una pipe. Las pipes se utilizan como un canal de comunicación entre la salida de un programa (o thread) y la entrada de otro.
- **`StringReader` y `StringWriter`. `StringBufferInputStream`**: La primera permite leer de un `String` en memoria. `StringWriter` recoge los caracteres que se le escriben en un `StringBuffer`, que puede ser convertido después a un `String`. `StringBufferInputStream` es similar a `StringReader`, excepto que lee bytes desde un `StringBuffer`.

Como ejemplo, el siguiente programa permite copiar un fichero en otro:

```
import java.io.*;
public class Copy {
   public static void main(String[] args) throws IOException {
      File eFic = new File(args[0]);
      File sFic = new File(args[1]);
      FileReader ent = new FileReader(eFic);
      FileWriter sal = new FileWriter(sFic);
      int c;
      while ((c = ent.read()) != -1)
         sal.write(c);
      ent.close();
      sal.close();
   }
}
```

5.4.3 Streams con proceso

Los streams con proceso realizan algún tipo de operación, como puede ser *buffering* o codificación de caracteres, a la vez que van leyendo o escribiendo. Como en el caso de los streams directos, `java.io` contiene habitualmente pares de streams: uno que realiza una operación particular cuando lee y otro que realiza la misma (o la inversa) mientras escribe.

- **`BufferedReader` y `BufferedWriter`. `BufferedInputStream` y `BufferedOutputStream`**: Realizan buffering de los datos mientras leen o escriben, reduciendo de esa manera el número de accesos directos necesarios sobre el repositorio de datos.
- **`FilterReader` y `FilterWriter`. `FilterInputStream` y `FilterOutputStream`**: Son clases abstractas, que permiten establecer filtros sobre los datos a medida que son leídos o escritos.
- **`InputStreamReader` y `OutputStreamWriter`**: Constituyen un par lector/escritor que proporcionan un enlace entre los streams de carácter y los de byte. Un `InputStreamReader` lee bytes de un `InputStream` y los convierte a caracteres

usando la codificación de caracteres por defecto o una codificación específica. De manera similar, un `OutputStreamWriter` convierte caracteres a bytes usando la codificación de caracteres por defecto o una codificación específica y los escribe después en un `OuputStream`.

- **`SequenceInputStream`**: Concatena varios streams de entrada en uno solo.
- **`ObjectInputStream` y `ObjectOutputStream`**: Se utilizan para ***serializar*** objetos, de manera que puedan escribirse en algún soporte no volátil (como el disco). De esta manera puede implementarse un mecanismo de persistencia de objetos. La operación clave para poder escribir un objeto es representar su estado en una forma serializada, que incluya a todos los demás objetos a los que haga referencia, de manera que pueda ser reconstruido al ser leido. Por eso la operación de leer y escribir objetos es conocida como serialización.
- **`DataInputStream` y `DataOutputStream`**: Leen o escriben tipos primitivos Java en un formato arquitecturalmente neutro.
- **`LineNumberReader`. `LineNumberInputStream`**: Llevan la cuenta del número de líneas mientras leen.
- **`PushbackReader`. `PushbackInputStream`**: Dos streams de entrada con buffer de vuelta atrás de 1 carácter (ó 1 byte).
- **`PrintWriter`. `PrintStream`**: Contienen métodos apropiados para presentar información textual (como `println`). Son los streams a los que es más fácil escribir, así que es muy habitual colocarlos encima de otros streams de salida.

5.4.4 Otras clases

Además de las clases que hemos visto más arriba, `java.io` contiene otras clases e interfaces de interés:

- **`File`**: Representa un fichero en el sistema de ficheros nativo. A partir de un objeto de esta clase se puede obtener información sobre el fichero, como permisos, el pathname completo, etc.
- **`FileDescriptor`**: Representa un descriptor a un fichero o socket abierto.
- **`StreamTokenizer`**: Permite partir el contenido de un stream en ***tokens***. Los tokens son la unidad más pequeña reconocida por una algoritmo de análisis sintáctico de un texto (palabras, símbolos, etc.). Un `StreamTokenizer` puede ser utilizado como base para analizar cualquier texto, como por ejemplo, a la hora de analizar un fichero HTM en busca de determinadas marcas.
- **`FilenameFilter`**: Lo utiliza el método `list` de la clase `File` para determinar qué ficheros deben presentarse de un determinado directorio. `FilenameFilter` acepta los ficheros de acuerdo con sus nombres. Es típicamente utilizado para implementar mecanismos de búsqueda basados en expresiones regulares simples como “*.class”.

MANEJO DE ERRORES: EXCEPCIONES

Si hay algo seguro sobre la vida de un programa es que va a sufrir errores. Esto es algo que no se puede evitar. Lo importante es decidir qué hacer cuando un error ocurre, cómo se maneja, quién lo maneja, cómo decidir si el programa puede recuperarse del error o debe terminar.

El término ***excepción***, en el entorno del lenguaje Java y de la JVM se refiere a un evento que ocurre durante la ejecución de un programa y que afecta al flujo normal de instrucciones. Muchos tipos de error pueden causar excepciones, desde errores serios de hardware como un disco que falla hasta simples errores de programación como intentos de acceso fuera de los límites de un array. Cuando uno de estos errores ocurre dentro de un método Java, el método crea un objeto excepción y lo envía hacia la JVM. El objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa en el momento en que ocurrió el error. La máquina virtual se encarga de encontrar el código encargado de manejar el error. En la terminología Java, crear un objeto excepción y enviarlo hacia la JVM se conoce como ***lanzar una excepción***.

Una vez un método lanza un excepción, el entorno de ejecución entra en acción para encontrar quién debe manejarla. El conjunto de los posibles "quienes" es el de los métodos que se encuentren en la pila de llamadas, a partir del método en el que se produjo el error. La JVM busca hacia atrás en la pila hasta que encuentra un método que contenga un manejador de excepciones adecuado. Un manejador de excepciones se considera adecuado si el tipo de la excepción lanzada es del mismo tipo del de las excepciones manejadas por el manejador. De manera que la excepción sube por la pila de llamadas hasta que se encuentra un manejador apropiado y alguno de los métodos llamantes se hace cargo de ella. Se dice entonces que el manejador de excepciones elegido ***captura la excepción***.

Si el entorno de ejecución no encuentra ningún método en la pila con un manejador adecuado, la ejecución en curso de la JVM (y por tanto del programa) termina.

El uso de excepciones para gestionar los errores ofrece varias ventajas frente a otros métodos tradicionales, como el retorno de valores fuera de rango:

- Permite separar el código que maneja los errores del código "normal", facilitando así su análisis y legibilidad.
- La propagación de los errores es automática a través de la pila de llamadas, sin necesidad de multiplicar esquemas de retorno en los métodos "intermediarios".
- Dado que se utiliza un objeto para dar parte del error, se aprovecha la jerarquía de clases para agrupar los errores del mismo tipo y para diferenciarlos entre sí.

6.1 Capturar o especificar

Java requiere que un método *capture o especifique las excepciones comprobadas que pueden ser lanzadas dentro de su ámbito*. Este requisito tiene varios componentes y necesitamos un análisis detallado de lo que significan términos como “capturar”, “especificar”, “excepción comprobada” y “excepciones que pueden ser lanzadas dentro del ámbito del método”

Un método puede capturar una excepción proporcionando un manejador para ese tipo de excepciones. Si el método no captura una determinada excepción, debe especificar que puede lanzar esa excepción. La razón para requerir esto es que cualquier excepción que pueda lanzar un método forma también parte del interface de programación del método. Los métodos que llamen al método en cuestión deben saber qué excepciones puede lanzar para poder decidir adecuadamente qué hacer con ellas. Por tanto, la ***signatura*** de un método incluye su nombre, sus parámetros y las excepciones que puede lanzar.

Java considera diferentes tipos de excepciones, uno de los cuales es el de las excepciones de *runtime*. Las excepciones de runtime son aquéllas que ocurren dentro de la propia máquina virtual Java. Esto incluye excepciones aritméticas (como una división por cero), excepciones de puntero (como tratar de acceder a un objeto a través de una referencia cuyo valor es `null`) y excepciones de índice (como tratar de acceder al elemento de un array más allá de sus límites).

Las excepciones de runtime pueden ocurrir en cualquier punto de un programa y pueden ser muy numerosas. El coste de comprobar todas las posibles excepciones de runtime suele superar con mucho a los beneficios de capturarlas o especificarlas, aunque siempre es posible hacerlo. Las excepciones comprobadas son aquellas que no son de runtime y que son comprobadas por el compilador. Éste comprueba que tales excepciones son capturadas o especificadas.

La frase “excepciones que pueden ser lanzadas dentro del ámbito del método” puede parecer obvia: basta buscar la sentencia `throw`. Sin embargo, en ese conjunto de excepciones se incluyen tanto:

- Las excepciones que pueden ser directamente lanzadas por el método usando la sentencia `throw`.
- Las excepciones que son lanzadas indirectamente por el método a través de llamadas a otros métodos.

6.2 Manejadores de excepciones

Consideremos el código del siguiente método, que se encarga de volcar a disco el contenido de un array (`lista`) usando para ello otra variable de instancia (`longitud`):

```
public void guardarLista() {
   PrintWriter sal = new PrintWriter(new FileWriter("Lista.txt"));
   for (int i = 0; i < longitud; i++)
      sal.println("Valor en: " + i + " = " + lista.elementAt(i));
   sal.close();
}
```

Este segmento de código daría un error al compilar ya que existen excepciones comprobadas (lanzada por los métodos de E/S que utiliza) que no son capturadas ni especificadas. Por ello vamos a construir un manejador de excepciones.

El primer paso para construir un manejador de excepciones es incluir las sentencias que pueden lanzar una excepción dentro de un bloque `try`. La estructura general de un bloque `try` es:

```
try { sentenciasJava }
```

Para construir el manejador de excepciones en el método `guardarLista` necesitamos incluir las sentencias que pueden lanzar una excepción dentro de bloques `try`. Hay varias maneras de hacer esto: pueden utilizarse diferentes bloques `try` para cada sentencia distinta que pueda lanzar una excepción o usar un único bloque `try` y asociarle múltiples manejadores. Aquí usaremos la segunda estrategia, ya que suele ser más fácil de leer e interpretar:

```
PrintWriter sal = null;
try {
   sal = new PrintWriter( new FileWriter(“Salida.txt”));
   for (int i = 0; i < longitud; i++)
      sal.println(“Valor en:” + i + “ = ”+ lista.elementAt(i));
}
```

La sentencia `try` controla las sentencias definidas dentro de ellas y define el ámbito de los manejadores de excepciones asociadas con ella. En otras palabras, si una excepción es lanzada dentro del bloque `try`, esa excepción es manejada por el manejador adecuado asociado con él. Un bloque `try` debe venir acompañado por al menos un bloque `catch` y (posiblemente) por un bloque `finally`.

Para asociar manejadores de excepciones con un bloque `try` se colocan uno o más bloques `catch` inmediatamente detrás del bloque `try`:

```
try {
   . . .
}
catch ( . . . ) {
   . . . }
catch ( . . . ) {
   . . . }
```

No puede haber ninguna sentencia entre el final del bloque `try` y el comienzo del primer bloque `catch`. La estructura general de un bloque `catch` es:

```
catch (ClaseThrowable nombreVariable) { sentenciasJava }
```

Por tanto, `catch` requiere un argumento formal, que se especifica de manera similar al parámetro de un método. El tipo del argumento, `ClaseThrowable`, declara el tipo de excepciones que el manejador puede manejar y debe ser el nombre de una clase derivada de la clase `Throwable` definida en el paquete `java.lang`. Cuando un programa Java lanza una excepción está pasando un objeto hacia la JVM y solamente objetos que se derivan de `Throwable` pueden ser lanzados.

`nombreVariable` es el nombre que el manejador podrá usar para referirse a la excepción capturada por el manejador. Por ejemplo, los manejadores de excepciones para el método `guardarLista` (que veremos más abajo) llaman al método `getMessage` utilizando el nombre declarado, `e`, de la excepción:

```
e.getMessage()
```

Las variables de instancia y los métodos de una excepción se acceden de la misma manera que los de cualquier otro objeto. `getMessage` es un método proporcionado por la clase `Throwable` que muestra información adicional sobre el error. La clase `Throwable` tiene también métodos para presentar el estado de la pila de ejecución en el momento del error. Las subclases de `Throwable` pueden añadir otros métodos y variables.

El bloque `catch` contiene una serie de sentencias que son ejecutadas cuando el manejador de excepciones es invocado. La JVM invoca al manejador cuando es el primero en la pila de llamadas cuyo tipo coincide con el de la excepción lanzada.

Para el método `guardarLista` vamos a definir dos manejadores de excepciones, uno para los dos diferentes tipos de excepciones que pueden ocurrir dentro del bloque `try`:

```
try {
   . . .
}
catch (ArrayIndexOutOfBoundsException e) {
   System.err.println(“ArrayIndexOutOfBoundsException: “ +
                      e.getMessage());
}
catch (IOException e) {
   System.err.println(“IOException: “ + e.getMessage());
}
```

Estos dos manejadores están especializados. Cada uno maneja un tipo concreto de excepción. Java permite también escribir manejadores generales para varios tipos de excepciones. Un manejador puede definirse para cualquier objeto de la jerarquía que se deriva de la clase `Throwable`. Si usamos un objeto en el extremo de la jerarquía (sin subclases) tenemos un manejador específico para esa clase. Por el contrario, si se emplea un objeto intermedio en la jerarquía, el manejador será activado para excepciones del tipo en cuestión y de cualquiera de sus subclases.

Así, podemos modificar los dos manejadores de arriba en uno que captura, genéricamente, cualquier excepción que ocurra dentro del bloque `try`:

```
try {
   . . .
}
catch (Exception e) {
   System.err.println(“Excepcion: “ + e.getMessage());
}
```

En general, es recomendable escribir manejadores de excepciones más especializados que el anterior. Los manejadores que capturan prácticamente todas las excepciones son poco útiles si queremos implementar mecanismos de recuperación frente a errores. Además, los manejadores muy generales pueden causar errores, dado que es posible que capturen excepciones que no fueron previstas, enmascarándolas a otros manejadores.

El paso final para definir un manejador de excepciones es proporcionar un mecanismo para “limpiar” el estado del método antes de que el control pase (probablemente) a otra parte del programa. Esto se puede hacer a través de un bloque `finally`.

El bloque `try` del método `guardarLista` abre un `PrintWriter`. El stream debería ser cerrado antes de que el método retorne. Y hay que tener en cuenta que el bloque `try` tiene tres posibles salidas:

- `new FileWriter` falla y lanza una `IOException`
- `lista.elementAt(i)` falla y lanza una `ArrayIndexOutOfBoundsException`
- Todo va bien y el bloque `try` termina normalmente

Por supuesto podríamos incluir varias llamadas al método `close` en los diferentes puntos de salida, pero eso significaría duplicar código. Para evitar duplicar el código podemos usar un bloque `finally`. La JVM siempre ejecuta las sentencias dentro de un bloque `finally` independientemente de lo que haya ocurrido en el bloque `try` al que va asociado:

```
finally {
   if (sal != null) sal.close();
}
```

En definitiva, una implementación completa del método `guardarLista` podría ser la siguiente:

```
public void guardarLista() {
   PrintWriter sal = null;
   try {
      sal = new PrintWriter( new FileWriter(“Salida.txt”));
      for (int i = 0; i < longitud; i++)
         out.println(“Valor en: “ + i + “ = “ + lista.elementAt(i));
   }
   catch (ArrayIndexOutOfBoundsException e) {
      System.err.println(“ArrayIndexOutOfBoundsException: “ +
                         e.getMessage());
   }
   catch (IOException e) {
      System.err.println(“IOException: “ + e.getMessage());
   }
   finally {
      if (sal != null) sal.close();
   }
}
```

6.3 Excepciones lanzadas por un método

6.3.1 Declaración de las excepciones

Hemos visto cómo se escribe un manejador de excepciones. Sin embargo, hay situaciones en que lo apropiado es que el método no maneje las excepciones, sino que las progrese hacia arriba en la pila de llamadas. Por ejemplo, si `guardarLista` pertenece a una clase que va incluida en un paquete general para ser utilizado por otros programadores, es recomendable dejar que las excepciones se propaguen y que los usuarios de la clase decidan qué hacer con ellas.

Si el método `guardarLista` no va a capturar las excepciones que pueden ocurrir dentro de él, debe especificar que puede lanzarlas. La versión original de `guardarLista` era:

```
public void guardarLista() {
   PrintWriter sal = new PrintWriter(new FileWriter("Lista.txt"));
   for (int i = 0; i < longitud; i++)
      sal.println("Valor en: " + i + " = " + lista.elementAt(i));
   sal.close();
}
```

Como hemos visto, las excepciones que pueden ser lanzadas son `IOException` (que no es una excepción de runtime) y `ArrayIndexOutOfBoundException` (que sí lo es). Para especificar que `guardarLista` lanza estas dos excepciones se utiliza la sentencia `throws`:

```
public void guardarLista() throws IOException,
ArrayIndexOutOfBoundsException {
```

Hay que recordar que `ArrayIndexOutofBoundsException` es una excepción de runtime, así que no es necesario declararla, aunque siempre es posible hacerlo.

6.3.2 La sentencia `throw`

Cualquier método Java puede usar la sentencia `throw` para lanzar una excepción. La sentencia `throw` requiere un solo argumento: un objeto "lanzable", es decir, una instancia de alguna subclase de la clase `Throwable`:

```
throw objetoThrowable;
```

Si el código trata de lanzar un objeto que no cumpla esta condición se producirá un error de compilación.

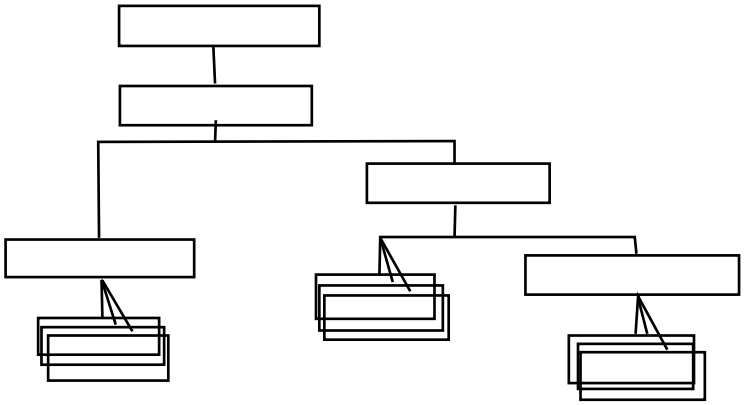
Veamos como funciona la sentencia `throw`. Consideremos el método `pop` de una implementación típica de una pila: `pop` borra el elemento que está en lo más alto de la pila y lo retorna:

```
public Object pop() throws EmptyStackException {
   Object obj;
   if (altura == 0) throw new EmptyStackException();
   obj = pila[altura - 1];
   pila[altura - 1] = null;
   altura--;
   return obj;
}
```

El método `pop` comprueba si hay algún elemento en la pila. Si está vacía (su altura es igual a 0), `pop` instancia un nuevo objeto de clase `EmptyStackException` y lo lanza. La clase `EmptyStackException` se define en el paquete `java.util`. Por supuesto, es posible (y muchas veces deseable) definir clases específicas de excepciones. Para ello, como ya hemos visto, la única condición que impone Java es que la clase se incluya en la jerarquía que tiene su raíz en `Throwable`.

6.4 La clase `Throwable` y sus subclases

Como hemos visto, solamente es posible lanzar objetos que deriven de la clase `Throwable`. Esto incluye tanto a sus descendientes directos (es decir, los que se derivan directamente de la clase `Throwable`) como a los indirectos (los que se derivan de hijos o nietos de la clase `Throwable`). `Throwable` tiene dos descendientes directos: `Error` y `Exception`.



Cuando ocurre algún tipo de fallo “duro” (como un error en el linkado dinámico de las clases) dentro de la máquina virtual, ésta lanza un `Error`. La inmensa mayoría de los programas Java no deben capturar objetos de la clase `Error` y es también muy improbable que deban lanzarlos.

La mayor parte de los programas, pues, lanzan y capturan objetos que se derivan de la clase `Exception`, que indican errores que no son graves problemas del sistema. La clase `Exception` tiene muchos descendientes definidos en los paquetes Java. Por ejemplo, `IllegalAccessException` señala que un método específico no pudo ser encontrado y `NegativeArraySizeException` indica que un programa ha intentado crear un array de tamaño negativo.

Como ya hemos visto, hay una clase especial de excepciones, que no son comprobadas por el compilador cuando evalúa las signaturas y manejadores de excepciones de los métodos: la clase `RuntimeException` y sus descendientes. Esto quiere decir que, si creamos una excepción que se derive de una de estas clases, no entrará en los chequeos del compilador, dado que se asume que este tipo de excepciones pueden ocurrir en cualquier parte del código. Por tanto, colocar excepciones específicas en la jerarquía que tiene su raíz en `RuntimeException` es una práctica poco aconsejable.

THREADS Y CONCURRENCIA

La mayor parte de los programas habituales son secuenciales: tienen un comienzo, una secuencia de ejecución y un final. En cualquier momento de la ejecución del programa existe un único punto de ejecución.

Un thread es similar a un programa secuencial: tiene un comienzo, un fin, una secuencia de ejecución y en cada momento de la ejecución del thread existe un único punto de ejecución. Sin embargo, un thread no es un programa por sí mismo. No puede correr por sí solo, sino que corre dentro de un programa. Para dar una definición estricta de thread diríamos que es un flujo de control secuencial dentro de un programa. Evidentemente, un programa con un solo thread no tiene nada nuevo, lo realmente importante del uso de threads dentro de un programa es poder disponer de varios de ellos ejecutándose al mismo tiempo y realizando diferentes tareas.

En algunos sitios se utiliza el término de procesos ligeros en vez de el de threads. Un thread se parece a un proceso real en que tanto éste como aquél contiene un flujo secuencial de control. Sin embargo, se le considera ligero porque se ejecuta en el contexto de un programa y utiliza los recursos del mismo.

Dado que es un flujo secuencial de control, un thread debe aislar algunos de sus recursos del resto del programa, como por ejemplo su pila de ejecución o su contador de programas. El código que se ejecuta en el thread trabaja dentro de ese contexto aislado.

Java ofrece mecanismos para dotar a los programas de la ejecución concurrente de varios threads, a través de la clase `Thread` y del interface `Runnable`. Ambos proporcionan el método `run`, que es la base de la ejecución de un thread en Java.

7.1 El método `run`

El método `run` proporciona a un thread algo que hacer. Su código implementa el comportamiento del thread durante su ejecución. Dentro del método puede incluirse cualquier cosa que pueda hacerse en Java: desde calcular una lista de números primos a presentar un videoclip.

La clase `Thread` implementa un thread genérico que, por definición, no hace nada. Es decir, la implementación de su método `run` está vacía. De manera que la clase `Thread` define un API que permite a un objeto que cumpla el interface `Runnable` definir el método `run` para un thread.

Existen dos técnicas para proporcionar el método `run` de un thread: extender la clase `Thread` (que a su vez implementa `Runnable`) y reescribir el método `run` o implementar el interface `Runnable`.

Para el primer método veamos un ejemplo, la clase `ThreadSimple`:

```
public class ThreadSimple extends Thread {
   public ThreadSimple(String str) {
      super(str);
```

```
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println(i + " " + getName());
         try { sleep((int)(Math.random() * 1000)); }
         catch (InterruptedException e) {}
      }
      System.out.println("Fin: " + getName());
   }
}
```

El primer método de la clase `ThreadSimple` es un constructor que toma un `String` como argumento. El constructor se implementa llamando al constructor de la superclase, de manera que le da un valor al nombre del thread, el cual se usa más adelante.

El siguiente método en `ThreadSimple` es el método `run`. El método `run` es el corazón de cualquier thread y donde la ejecución del mismo tiene lugar. El método `run` de la clase `ThreadSimple` contiene un bucle que se itera diez veces. En cada iteración el método muestra el número de iteración y el nombre del thread y después duerme durante un intervalo aleatorio de, como máximo, un segundo. Cuando el bucle ha terminado, muestra el mensaje "Fin" junto con el nombre del thread.

La clase `PruebaDeThreads` proporciona un método `main` que crea dos threads `ThreadSimple`: una se llama "Si" y la otra "No". Puede usarse para tomar decisiones complicadas: basta poner el programa a correr y quedarse con la respuesta del thread que acabe primero (o último).

```
public class PruebaDeThreads {
   public static void main (String[] args) {
      new ThreadSimple("Si").start();
      new ThreadSimple("No").start();
   }
}
```

El método `main` arranca cada thread de manera inmediata después de su construcción por medio de la llamada al método `start`. Una posible salida de la ejecución de la clase sería:

```
0 Si
0 No
1 No
1 Si
2 Si
2 No
3 No
3 Si
4 Si
4 No
5 Si
5 No
6 No
6 Si
7 Si
7 No
8 No
9 No
```

```
8 Si
Fin: No
9 Si
Fin: Si
```

Podemos ver que la salida de cada thread está mezclada arbitrariamente con la salida del otro, dado que ambos `ThreadSimple` están siendo ejecutados concurrentemente y mostrando su salida a la vez.

El otro método de proporcionar un método `run` a un thread lo veremos con detalle en el capítulo siguiente, cuando analicemos el uso de threads dentro de applets (“Threads y applets”, página 82). Baste decir que es interesante siempre que la clase para la que queramos proporcionar un método `run` deba derivarse de una clase distinta de `Thread`, como es el caso de los applets (que, curiosamente, deben derivarse de la clase `Applet`).

7.2 El ciclo de vida de un thread

7.2.1 Creación

Una vez un thread ha sido instanciado (y construido), pasa a estar en el estado de “nuevo”. Cuando un thread se encuentra en este estado, está simplemente vacío: aún no se han reservado recursos del sistema para él. Lo único que se puede hacer con un thread en el estado de “nuevo” es arrancarlo mediante una llamada a su método `start`.

Invocar otro método que no sea `start` provoca una `IllegalThreadStateException`. De hecho, la máquina virtual Java lanza una excepción de esta clase cada vez que se invoca un método de un thread y el thread se encuentra en un estado que no permite esa invocación

7.2.2 Arranque

El método `start` crea los recursos del sistema necesarios para la ejecución del thread, prepara su ejecución y llama al método `run` del thread.

Un vez el método `start` retorna, el thread se encuentra en el estado de “ejecución”. En realidad es un poco más complicado: el thread se encuentra en estado “ejecutable”. Lo habitual es que un ordenador tenga menos procesadores que el número de threads en ejecución (la mayoría tiene solamente uno) y es imposible ejecutar todos ellos al mismo tiempo. La máquina virtual Java implementa un mecanismo de administración para permitir que el procesador sea compartido por todos los threads en estado “ejecutable”. De manera que, en un determinado momento, un thread en ejecución puede estar en realidad esperando su turno para entrar en el procesador.

7.2.3 Suspensión

Un thread queda en estado “suspendido” cuando ocurre alguno de estos eventos:

- Su método `sleep` es invocado.

- El thread llama al método `wait` para esperar a que se satisfaga una cierta condición.
- El thread se encuentra bloqueado esperando a que termine una operación de entrada/salida.

Para cada modo de entrar en el estado “suspendido” hay un manera específica y distinta de volver al estado “ejecutable”. Cada una de estas formas de volver al estado “ejecutable” funciona únicamente para el modo de entrar en el estado “suspendido” correspondiente. Estas formas de retorno son:

- Si el thread invocó a `sleep`, cuando transcurra el número especificado de milisegundos.
- Si el thread está esperando una condición por medio de `wait`, algún otro objeto debe notificar al objeto en espera del cambio de la condición llamando a `notify` o `notifyAll`.
- Si el thread está bloqueado por una operación de E/S, cuando la operación se complete.

7.2.4 Parada

Un thread debe pararse a sí mismo por medio de un método `run` que termine de una manera natural, es decir, un thread debe morir de “muerte natural”. Por ejemplo, el bucle `while` del siguiente método `run` es un bucle finito: se itera 100 veces y después termina:

```
public void run() {
   int i = 0;
   while (i < 100) {
      i++;
      System.out.println(“i = “ + i);
   }
}
```

Un thread con este método `run` muere de manera natural cuando el bucle se termina y el método `run` acaba.

Veamos un método más sofisticado, que se utiliza muy a menudo en applets:

```
public class Ejemplo extends Applet implements Runnable {
   private Thread esteThread = null;
   public void start() {
      if (esteThread == null) {
         esteThread = new Thread(this);
         esteThread.start();
      }
            . . .
   }
   public void run() {
      Thread curThread = Thread.currentThread();
      while (esteThread == curThread) {
                  . . .
      }
   }
```

```
   public void stop() {
      esteThread = null;
   }
}
```

La condición de salida de este método `run` es la condición de salida del `while`, ya que no hay código después del final del bucle:

```
while (esteThread == curThread)
```

Esta condición indica que el bucle se terminará cuando el thread actual (en el que el método `run` está siendo ejecutado) sea distinto del valor de la variable de instancia `esteThread`.

Cuando se invoca al método `stop` (lo que en un applet, como veremos, ocurre al dejar la página Web en la que está incluido) la variable `esteThread` se pone a `null`, de manera que el bucle principal del método `run` se termina y el thread muere, por tanto, de manera natural.

7.2.5 El método `isAlive`

El API de la clase `Thread` incluye un método llamado `isAlive`, que retorna `true` si el thread ha sido arrancado y todavía no ha sido parado. Si `isAlive` retorna `false`, únicamente sabemos que el thread está en los estados "nuevo" o "parado", mientras que si retorna `true` sabemos que se encuentra en los estados "ejecutable" o "suspendido".

No hay manera, por tanto, de diferenciar los pares de estados "nuevo"/"parado" y "ejecutable"/"suspendido" usando el API de `Thread`.

7.2.6 Prioridades

Como ya hemos dicho la mayoría de los ordenadores deben compartir el procesador entre los threads que están en estado "ejecutable" en un momento determinado. La máquina virtual Java soporta un algoritmo muy simple para asignar el procesador a un thread determinado basado en un esquema de prioridades fijas.

Cada thread Java recibe una prioridad, expresada por un número entre `MIN_PRIORITY` y `MAX_PRIORITY` (constantes definidas en la clase `Thread`). En un momento dado, cuando hay múltiples threads en estado de ser ejecutados, el thread con la prioridad más alta es el escogido. Solamente cuando ese thread se para o es suspendido por algún motivo, otro thread de prioridad más baja podrá comenzar a ejecutarse.

La asignación del procesador es completamente *preemptive*: si un thread con prioridad más alta que el que se está ejecutando necesita el procesador, éste le es asignado inmediatamente.

La máquina virtual Java no desalojará a un thread en ejecución por otro thread de la misma prioridad. En otras palabras, la JVM no implementa mecanismos de *time-slicing*. Aunque puede ser que la implementación del sistema sí proporcione estos mecanismos, no es recomendable confiar en ellos a la hora de diseñar un programa con múltiples threads.

Por otro lado, un thread dado puede, en cualquier momento, dejar su derecho de uso del procesador usando el método `yield`. Un thread puede ceder el procesador únicamente a threads de la misma prioridad. Los intentos de ceder el procesador a threads de menor prioridad son ignorados.

Cuando todos los threads ejecutables en el sistema tienen la misma prioridad, la máquina virtual selecciona el siguiente thread que se ejecutará por un método simple de *round-robin*.

7.3 Sincronización entre threads

Hasta ahora hemos visto ejemplos en los que aparecían threads que se ejecutaban de manera independiente y asíncrona. Cada thread contenía todos los datos y los métodos necesarios para su ejecución y no requería de ningún recurso o método exterior. Además, estos threads corrían a su propio ritmo, sin preocuparse del estado o actividades de otros threads que estuvieran ejecutándose en concurrencia.

Hay muchas situaciones interesantes en las que diferentes threads que se ejecutan concurrentemente comparten datos y deben tener en cuenta el estado y actividades de otros threads. Un conjunto de estas situaciones se conocen como escenarios de ***productor/consumidor***, en los que el productor genera un flujo de datos que es utilizado por el consumidor.

Por ejemplo, imaginemos una aplicación Java en la que un thread (el productor) escribe datos a un fichero mientras un segundo thread (el consumidor) lee datos del mismo fichero. Otro ejemplo podría ser un programa en el que, a medida que el usuario escribe en el teclado, el thread productor coloca eventos de pulsación de teclas en una cola de eventos y el thread consumidor lee los eventos de la misma cola. Ambos ejemplos utilizan threads concurrentes que comparten un recurso común (un fichero o la cola de eventos). Dado que los threads comparten un recurso común se deben sincronizar de algún modo.

7.3.1 El ejemplo Productor/Consumidor

En el ejemplo que vamos a ver, el `Productor` genera un entero entre 0 y 9, lo almacena en un objeto `Almacen` y escribe en pantalla el número generado. Para hacer que el problema de sincronización sea un poco más complicado, el `Productor` se suspende durante un período aleatorio entre 0 y 100 milisegundos antes de realizar la siguiente iteración del ciclo:

```
public class Productor extends Thread {
   private Almacen almacen;
   private int num;
   public Productor (Almacen a, int num) {
      almacen = a; this.num = num;
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         almacen.guardar(i);
         System.out.println("Productor..." + this.num +
                            "..Guardado:.." + i);
         try {
            sleep((int)(Math.random() * 100));
```

```
         }
         catch (InterruptedException e) { }
      }
   }
}
```

El `Consumidor` no contiene ningún mecanismo de espera y consume los objetos del `Almacen` (exactamente el mismo objeto en el que el `Productor` coloca los números que genera) a medida que están disponibles.

```
public class Consumidor extends Thread {
   private Almacen almacen;
   private int num;
   public Consumidor(Almacen a, int num) {
      almacen = a;
      this.num = num;
   }
   public void run() {
      int val = 0;
      for (int i = 0; i < 10; i++) {
         val = almacen.sacar();
         System.out.println("Consumidor.." + this.num +
                            "..Sacado:...." + val);
      }
   }
}
```

El `Productor` y el `Consumidor` en este ejemplo comparten datos a través de un objeto `Almacen` común. Y puede apreciarse que ni uno ni otro realizan ningún tipo de control para asegurarse que el `Consumidor` está leyendo los valores producidos una y sólo una vez. La sincronización entre estos dos threads, de acuerdo con el modelo que define Java, debe ocurrir a través de los métodos `guardar` y `sacar` del objeto `Almacen`. Por el momento, supongamos que no hay mecanismos de sincronización y veamos los problemas potenciales que pueden presentarse.

Un problema se produce si el `Productor` es más rápido que el `Consumidor` y genera dos números antes de que el `Consumidor` tenga oportunidad de leer el primero. De esta forma el `Consumidor` se saltaría un número y parte de la salida sería del tipo:

```
. . .
Consumidor..1..Sacado:....3
Productor...1..Guardado:..4
Productor...1..Guardado:..5
Consumidor..1..Sacado:....5
 . . .
```

Otro problema se puede presentar cuando el `Consumidor` es más rápido que el `Productor` y consume el mismo valor dos veces, produciendo una salida de este estilo:

```
. . .
Productor...1..Guardado:..4
Consumidor..1..Sacado:....4
Consumidor..1..Sacado:....4
Productor...1..Guardado:..5
. . .
```

En cualquier caso, el resultado no es correcto. El `Consumidor` debe leer cada entero que genere el `Productor` exactamente una vez. Los problemas de este tipo se conocen como ***condiciones de carrera***. Son el resultado de tener múltiples flujos de control accediendo de forma asíncrona a un solo recurso.

Para prevenir las condiciones de carrera en el ejemplo debemos sincronizar el almacenamiento de un nuevo entero en el `Almacen` por parte del `Productor` con la recogida de un entero del `Almacen` por parte del `Consumidor`. De esta manera, el `Consumidor` lee cada número exactamente una vez.

Las actividades del `Productor` y el `Consumidor` se deben sincronizar de dos maneras. En primer lugar, los dos threads no deben acceder simultáneamente al `Almacen`. Un thread Java puede hacer esto bloqueando un objeto. Cuando un objeto está bloqueado por un thread y otro thread trata de llamar a un método sincronizado en el mismo objeto, el segundo thread se bloquea hasta que el objeto es liberado.

En segundo lugar, los dos threads deben coordinarse. El `Productor` debe tener algún medio de indicarle al `Consumidor` que hay un valor disponible y el `Consumidor` debe tener alguna manera de avisar al `Productor` de que el valor ha sido leído. La clase `Object` proporciona una colección de métodos (`wait`, `notify` y `notifyAll`) para que los threads esperen a que una condición se satisfaga o informen a otros threads de un cambio en una determinada condición.

Por último, veamos un pequeño programa que nos permitirá probar el modelo productor/consumidor del ejemplo, creando un `Almacen`, un `Productor` y un `Consumidor` y arrancando los dos threads:

```
public class PruebaProCon {
   public static void main(String[] args) {
      Almacen a = new Almacen();
      Productor p1 = new Productor(a, 1);
      Consumidor c1 = new Consumidor(a, 1);
      p1.start();
      c1.start();
   }
}
```

7.3.2 Bloqueos sobre un objeto: secciones críticas

Los segmentos de código dentro de un programa que acceden al mismo objeto desde threads distintos que se ejecutan concurrentemente se conocen como ***secciones críticas***. En Java, una sección crítica puede ser un bloque o un método, marcados con la palabra `synchronized`. La máquina virtual asocia un mecanismo de bloqueo (un ***monitor***) para cada objeto que contiene código marcado con `synchronized`.

En el ejemplo que hemos visto, los métodos `guardar` y `sacar` del `Almacen` son secciones críticas. El `Consumidor` no debe acceder el `Almacen` mientras el `Productor` lo está cambiando y éste no debe modificar el `Almacen` mientras el `Consumidor` lo está leyendo. Por tanto, `guardar` y `sacar` deben estar marcados con `synchronized`.

Este es un esqueleto de la clase `Almacen`:

```
public class Almacen {
   private int contenido;
   private boolean hayAlgo = false;
   public synchronized int guardar() {
         ...
   }
   public synchronized void sacar (int val) {
         ...
   }
}
```

La declaración de los métodos `guardar` y `sacar` contienen la palabra `synchronized`. De esta manera el sistema asocia un único monitor con cada instancia de `Almacen` (incluyendo la que comparten el `Productor` y el `Consumidor`). Cuando se da control a un método sincronizado, el thread que llamó al método toma el monitor del objeto cuyo método ha sido llamado, bloqueándolo. Otros threads no pueden llamar a ningún método sincronizado del mismo objeto hasta que el monitor del objeto es liberado.

De esta manera, cuando el `Productor` llama al método `guardar` del `Almacen`, éste queda bloqueado, impidiendo que el `Consumidor` pueda llamar al método `sacar`. Cuando el método `guardar` retorna, el `Productor` libera el `Almacen`. La situación es la recíproca cuando el `Consumidor` llama al método `sacar` del `Almacen`.

La adquisición y liberación de un monitor es llevada a cabo por la máquina virtual de manera automática y atómica. Esto asegura que no pueden ocurrir condiciones de carrera en los mecanismos de soporte de los threads y, por tanto, garantiza la integridad de los datos.

Por otro lado, la máquina virtual permite que un thread vuelva a adquirir un monitor que ya tiene: los monitores Java son ***reentrantes***. Esto es importante dado que se elimina así la posibilidad de que un thread se bloquee a sí mismo en un monitor que ya tiene. Para ver esto consideremos la clase:

```
public class Reentrante {
   public synchronized void a() {
      b();
         . . .
}
   public synchronized void b(){
         . . .
   }
}
```

`Reentrante` contiene dos métodos sincronizados: `a` y `b`. El primero de ellos llama al otro.

Cuando el control llega al método `a`, el thread actual adquiere el monitor del objeto `Reentrante`. En ese momento, `a` llama a `b` y, dado que `b` está sincronizado también, el thread trata de adquirir el mismo monitor de nuevo. Como Java ofrece monitores reentrantes, esto funciona sin problemas y el thread adquiere el monitor de `Reentrante` de nuevo, de manera que `a` y `b` se ejecutan como está especificado. En sistemas que no soportan monitores reentrantes, la secuencia de llamadas aquí descrita causaría un bloqueo fatal (*deadlock*).

7.3.3 `notifyAll` y `wait`

El `Almacen` del ejemplo que estamos viendo guarda su valor en una variable privada llamada `contenido`. Hay otra variable privada, `hayAlgo`, que es un `boolean`. `hayAlgo` es `true` cuando un valor ha sido guardado pero todavía no se ha sacado y es `false` en el caso contrario. De acuerdo con esto, esta es una posible implementación de los métodos `guardar` y `sacar`:

```
public synchronized void guardar(int val) { // No funciona
   if (hayAlgo == false) {
      hayAlgo = true;
      contenido = val;
   }
}
public synchronized int sacar() { // No funciona
   if (hayAlgo == true) {
      hayAlgo = false;
      return contenido;
   }
}
```

Tal como aparecen aquí, estos dos métodos no funcionarán como deseamos. Consideremos el método `sacar`. ¿Qué ocurre si el `Productor` no ha puesto nada en el `Almacen` y `hayAlgo` no es `true`? `sacar` no hace nada en ese caso. De manera similar, si el `Productor` llama a `guardar` antes de que el `Consumidor` haya recogido un valor, `guardar` no hace nada.

Lo que queremos realmente es que el `Consumidor` espere hasta que el `Productor` ponga algo en el `Almacen` y que el `Productor` notifique al `Consumidor` cuando lo haga. Igualmente, el `Productor` debe esperar hasta que el `Consumidor` tome un valor y notifique al `Productor` de que lo ha hecho. Los dos threads deben coordinarse más estrechamente y para eso están diseñados los métodos `wait` y `notifyAll` de la clase `Object`:

```
public synchronized void guardar(int val) {
   while (hayAlgo == true) {
      try { // Espera hasta que el Consumidor saque algo
         wait();
      }
      catch (InterruptedException e) { }
   }
   contenido = val;
   hayAlgo = true; // Avisamos al Consumidor
   notifyAll();
}
public synchronized int sacar() {
   while (hayAlgo == false) {
      try { // Espera hasta que el Productor guarde algo
         wait();
      }
      catch (InterruptedException e) { }
   }
   hayAlgo = false; // Avisamos al Productor
   notifyAll();
   return contenido;
```

```
}
```

El código del método `sacar` está en un bucle hasta que el `Productor` ha producido un nuevo valor. En cada posible iteración del bucle se llama al método `wait`. Este método libera el monitor que tenía el `Consumidor` sobre el `Almacen` (de manera que permite al `Productor` adquirir el monitor y actualizar el `Almacen`) y espera la notificación del `Productor`. Cuando éste guarda algo en el `Almacen`, lo notifica al `Consumidor` llamando a `notifyAll`. El Consumidor sale entonces del estado de espera, `hayAlgo` es ahora true, el bucle se acaba y el método `sacar` retorna el valor guardado en el `Almacen`.

El método `guardar` funciona de una manera similar, esperando a que el thread del `Consumidor` lea el valor actual antes de permitir al `Productor` que escriba uno nuevo.

El método `notifyAll` despierta a todos los threads que están en espera en el objeto en cuestión (en este caso, el `Almacen`). Los threads despertados compiten por el monitor. Uno de ellos lo obtiene y los otros vuelven al estado de espera. La clase `Object` define también el método `notify`, que despierta (arbitrariamente) a uno de los threads que está en espera en el objeto.

La clase `Object` contiene otras dos versiones de `wait`, que permiten especificar el tiempo máximo de espera:

- `wait(long timeout)`: Espera hasta que se produce la notificación o hasta que el tiempo especificado en `timeout` ha pasado. `timeout` se expresa en milisegundos.
- `wait(long timeout, int nanos)`: Espera hasta que se produce la notificación o hasta que transcurren `timeout` milisegundos más `nanos` nanosegundos.

Aparte de usar estas versiones de `wait` para sincronizar threads, también pueden usarse en lugar de `sleep`. La razón es sencilla: tanto `wait` como `sleep` retardan la ejecución del thread durante el período especificado, pero un thread que ha llamado a `wait` puede ser despertado con una llamada a `notify`, mientras que un thread que ha llamado a `sleep` no puede ser despertado prematuramente.

7.4 Grupos de threads

Todo thread Java es miembro de un grupo de threads. Los grupos de threads proporcionan un mecanismo para agrupar múltiples thread en un sólo objeto y manipularlos de manera colectiva. Por ejemplo, se pueden arrancar o suspender todos los threads dentro de un determinado grupo con una sola invocación de un método. Los grupos de threads están implementados por medio de la clase `ThreadGroup` del paquete `java.lang`.

La máquina virtual coloca un thread dentro de un grupo durante la construcción del thread. Cuando un thread es creado, el sistema lo coloca en un grupo por defecto a menos que explícitamente se indique el grupo al que debe pertenecer el nuevo thread. A partir de ese momento, el thread es un miembro permanente del grupo al que se unió en su creación: no se puede mover un thread de un grupo a otro una vez el thread ha sido creado.

Si se crea un nuevo thread sin especificar su grupo en el constructor, la máquina virtual lo coloca de manera automática en el mismo grupo al que pertenezca el thread que lo ha creado. Cuando una aplicación Java se empieza a ejecutar, la JVM crea un `ThreadGroup` llamado `main`. A menos que se especifique explícitamente, todos los nuevos threads que se creen son miembros del grupo de threads `main`.

La mayor parte de los programas pueden ignorar los grupos de threads y dejar que la máquina virtual se encargue de ellos. Sin embargo, si un programa crea un número significativo de threads resulta recomendable agruparlos de manera adecuada para facilitar el control del programa.

Como hemos visto, para colocar un nuevo thread en un grupo distinto del grupo por defecto es necesario hacerlo en el momento de su creación. La clase `Thread` tiene tres constructores que permiten especificar el grupo al que va a pertenecer el nuevo thread:

- `public Thread(ThreadGroup group, Runnable target)`
- `public Thread(ThreadGroup group, String name)`
- `public Thread(ThreadGroup group, Runnable target,String name)`

Cada uno de estos constructores crea un nuevo thread, lo inicializa de acuerdo con los parámetros de tipo `String` y `Runnable` y hace al nuevo thread miembro del grupo especificado. El `ThreadGroup` que se le pasa a estos constructores no tiene por qué ser necesariamente un grupo creado por el programa. Puede ser un grupo creado por la máquina virtual o por la aplicación en la que un applet se está ejecutando.

Para saber a qué grupo pertenece un thread se puede llamar al método `getThreadGroup`:

```
grupo = unThread.getThreadGroup();
```

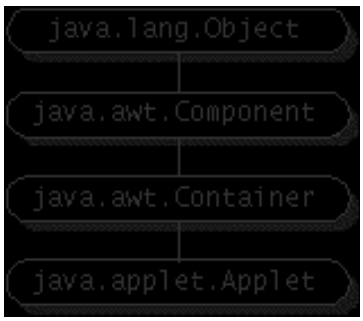
Una vez que se ha obtenido el grupo al que pertenece el thread, puede buscarse información acerca del grupo, como qué otros threads pertenecen a él. También se puede modificar el estado de todos estos threads con una sola invocación a un método.

APPLETS

Los applets son la aplicación que más popularidad ha dado al lenguaje Java y uno de los principales motivos de su rápida difusión. En esencia, un applet no es más que una aplicación que se ejecuta integrada dentro de un *browser* Web. En lo que sigue vamos a ver las principales características de los applets y cuáles son y cómo se utilizan los métodos que permiten dotar de contenido ejecutable a la información en Internet.

8.1 Cómo se implementa y se ejecuta un applet

Cualquier applet se implementa creando una subclase de la clase `java.applet.Applet`. La jerarquía de herencia de la que se deriva la clase `Applet`, y que determina en gran medida su funcionalidad, es la siguiente:



La clase `Applet` extiende a la clase `Panel` del AWT, que a su vez extiende a la clase `Container`, y ésta a la clase `Component`. De `Component`, un applet hereda la capacidad de tener una salida gráfica y de manejar eventos del interface de usuario. Un applet hereda de `Container` la capacidad de incluir otros componentes y la de tener un *layout manager* que controle el tamaño y posición de esos componentes. Directamente de la clase `Applet`, un applet hereda diversas capacidades, incluyendo la de responder a hitos en su ciclo de vida, como el que sea cargado o descargado por el browser.

Para incluir un applet en una página HTML se usa la marca `<APPLET>`. Cuando un browser recibe una página que contiene un applet lleva a cabo las siguientes acciones:

- Localiza el fichero ejecutable del applet (el fichero ".class" que contiene los bytecodes de la subclase de `Applet`). La localización del fichero se especifica con los atributos `CODE` y `CODEBASE` de la marca `<APPLET>`.
- Carga los bytecodes, a través de la red, en el ordenador en el que se está ejecutando.
- Crea una instancia de la subclase de `Applet`. Cuando nos referimos a un applet, nos estamos refiriendo en general a esta instancia.
- Llama al método `init` del applet. Este método debe llevar a cabo todas las inicializaciones necesarias.
- Llama al método `start` del applet.

La subclase de `Applet` a la que pertenece un applet es su clase principal, pero los applets pueden usar también otras clases. Estas otras clases pueden ser locales al browser, parte del entorno Java o clases específicas. Cuando el applet trata de usar una clase determinada por primera vez, el browser trata de encontrarla en el ordenador en el que se ejecuta. Si no la puede encontrar allí, la busca en el mismo lugar desde donde cargó la subclase de `Applet`. Cuando encuentra la clase, carga sus bytecodes (a través de la red si es necesario) y continúa ejecutando el applet.

Cargar código ejecutable a través de la red supone un riesgo. En el caso de los applets Java, parte de este riesgo se reduce debido a que el lenguaje está diseñado para ser seguro. Además, los browsers que soportan Java imponen ciertas restricciones para aumentar la seguridad. Estas restricciones incluyen no permitir a los applets cargar código en cualquier lenguaje que no sea Java e impedirles que lean o escriban ficheros en el ordenador donde se ejecuta el browser.

8.2 Métodos básicos en un applet

Aquí tenemos el código fuente de un applet llamado `Simple`. El applet `Simple` presenta una cadena de texto que describe cada uno de los hitos por los que pasa en su vida, como cuando se llega por primera vez a la página donde está incluido:

```
import java.applet.Applet;
import java.awt.Graphics;
public class Simple extends Applet {
   StringBuffer buffer;
   public void init() {
      buffer = new StringBuffer();
      hito(“inicializacion... “);
   }
   public void start() {
      hito(“arranque... “);
   }
   public void stop() {
      hito(“parada... “);
   }
   public void destroy() {
      hito(“listo para salir...”);
   }
   void hito(String id) {
      System.out.println(id);
      buffer.append(id);
      repaint();
   }
   public void paint(Graphics g) {
// Dibujamos un rectángulo alrededor del display del applet
// En JDK 1.1 se usa getSize() en vez de size()
      g.drawRect(0, 0, size().width - 1, size().height - 1);
// Ponemos la cadena en el rectángulo
      g.drawString(buffer.toString(), 5, 15);
   }
```

```
}
```

8.2.1 Métodos asociados al ciclo de vida

El applet `Simple`, como cualquier otro applet, es una subclase de la clase `Applet`. Esta subclase sobreescribe cuatro métodos para poder responder adecuadamente a los hitos en su ciclo de vida:

- `init`: Para inicializar el applet cada vez que es cargado (o recargado).
- `start`: Para comenzar la ejecución del applet. Esto ocurre cuando el applet es cargado y también cuando el usuario vuelve a visitar la página donde se encuentra el applet.
- `stop`: Para parar la ejecución del applet. Esto ocurre cada vez que el usuario deja la página donde se encuentra el applet.
- `destroy`: Para realizar cualquier tarea de limpieza final antes de que el browser elimine al applet de la memoria.

El método `init` es útil para llevar a cabo inicializaciones que se harán una sola vez en la vida del applet. En general, el método `init` debe contener lo que normalmente se pondría en un constructor. El motivo por el que los applets no deben usar directamente constructores es que no está garantizado que tengan un entorno completamente operativo hasta que se llama al método `init`. Por ejemplo, los métodos de carga de imágenes que proporciona la clase `Applet` no funcionan en un constructor.

Cualquier applet que haga algo después de su inicialización (excepto en respuesta directa a las acciones del usuario) debe reescribir el método `start`. Este método lleva a cabo el trabajo del applet, o arranca uno o más threads para realizar este trabajo.

La mayoría de los applets que reescriben `start` deben también reescribir `stop`. El método `stop` debe suspender la ejecución del applet, de manera que no consuma recursos del sistema cuando el usuario no está accediendo a la página donde se encuentra el applet. Por ejemplo, un applet que presenta una animación debe parar de dibujar la animación cuando el usuario no la está viendo.

En muchos casos no es necesario reescribir el método `destroy`, dado que el método `stop` (que es siempre llamado antes de llamar a `destroy`) puede hacer todo lo necesario para detener la ejecución del applet. Sin embargo, `destroy` está disponible para aquellos applets que necesiten liberar recursos adicionales.

8.2.2 Métodos gráficos y manejo de eventos

El applet `Simple` define su apariencia en pantalla reescribiendo el método `paint`:

```
class Simple extends Applet {
      . . .
   public void paint(Graphics g) {
         . . .
}
      . . .
```

```
}
```

Este es uno de los dos métodos de presentación gráfica que un applet puede reescribir:

- `paint`: Es el método básico para escribir en pantalla y es el que habitualmente se utiliza para presentar el applet en la página a la que pertenece.
- `update`: Es un método que puede utilizarse junto con `paint` para mejorar la rapidez de presentación en ciertas situaciones.

Los applets heredan los métodos `paint` y `update` de la clase `Component`, a través de la clase `Applet`.

Los applets también heredan de `Component` un grupo de métodos para manejar los eventos del interface de usuario, como `action` y `mouseDown` (para manejar tipos de eventos particulares) y un método general llamado `handleEvent`.

Para reaccionar a un evento un applet tiene dos alternativas: reescribir el método específico de manejo del evento en cuestión o reescribir el método `handleEvent`. Por ejemplo, si añadimos el siguiente código al applet `Simple`, éste será capaz de responder a clicks del ratón:

```
import java.awt.Event;
      . . .
   public boolean mouseDown(Event ev, int x, int y)
   {
      hito(“click!... “);
      return true;
   }
```

El código de presentación que hemos usado en el applet `Simple` tiene un grave defecto: no hace *scroll*. Una vez que el área de escritura se llena no es posible ver el nuevo texto que llega. La solución para este tipo de problemas es utilizar dentro del applet componentes “prefabricados” de interface de usuario que ya tengan definido un comportamiento adecuado.

El Abstract Window Toolkit (AWT) de Java proporciona los siguientes componentes de interface de usuario (las clases que los implementan aparecen en paréntesis:

- Botones (`java.awt.Button`).
- Casillas (`java.awt.Checkbox`).
- Campos de texto de una sola línea (`java.awt.TextField`).
- Áreas de presentación y edición de texto (`java.awt.TextArea`).
- Etiquetas (`java.awt.Label`).
- Listas (`java.awt.List`).
- Listas de alternativas seleccionables (`java.awt.Choice`).
- Barras de scroll (`java.awt.Scrollbar`).
- Áreas de dibujo (`java.awt.Canvas`).
- Menús (`java.awt.Menu`, `java.awt.MenuItem`, `java.awt.CheckboxMenuItem`)
- Contenedores (`java.awt.Panel`, `java.awt.Window` y sus subclases).

Dado que la clase `Applet` se deriva de la clase `Container` del AWT, es sencillo añadir componentes a un applet y utilizar managers de layout para controlar la posición de los componentes en la pantalla. Alguno de los métodos de `Container` que un applet puede usar son:

- `add`: Añade el componente especificado.
- `remove`: Elimina el componente especificado.
- `setLayout`: Define el manager de layout.

Veamos cómo se podría modificar el applet `Simple` para utilizar un campo de texto no editable como salida. Los cambios necesarios son los siguientes:

```
//Ya no necesitamos importar java.awt.Graphics, dado que no hay
//que reescribir el método paint
. . .
import java.awt.TextField;
public class Simple extends Applet {
//En lugar del StringBuffer, usamos directamente un TextField:
   private TextField field;
   public void init() {
//Creamos el campo de texto (no editable):
      field = new TextField();
      field.setEditable(false);
//Definimos el layout manager para que el campo de texto sea
//tan ancho como sea posible:
      setLayout(new java.awt.GridLayout(1,0));
//Añadimos el campo de texto al applet:
      add(field);
      validate();
      hito(“inicializacion... “);
   }
      . . .
   void hito(String id) {
//Ahora añadimos la cadena al TextField, en vez de al StringBuffer
      String t = field.getText();
      System.out.println(id);
      field.setText(t + id);
      repaint();
   }
//El método paint ya no es necesario: TextField se repinta
//automáticamente al llamar a repaint
```

El nuevo método `init` crea un campo de texto no editable (una instancia de `TextField`) y define como manager de layout del applet uno que permita que el campo de texto sea tan ancho como sea posible. Después de esto, añade el campo de texto al applet y llama al método `validate` (que el applet hereda de `Component`). Invocar a `validate` una vez se han añadido uno o más objetos `Component` a un applet es un pequeño truco que garantiza que los componentes se dibujarán en pantalla.

8.3 Restricciones de seguridad

Todos los browsers implementan ciertas políticas de seguridad para evitar que los applets puedan comprometer la seguridad del sistema donde se ejecutan. Aquí veremos las políticas de seguridad generales que emplean los browsers actualmente. Sin embargo, es importante notar que la implementación particular de estas políticas de seguridad difiere de un browser a otro. También hay que tener en cuenta que estas políticas están sujetas a cambio. Por ejemplo, un browser diseñado para trabajar en un entorno cerrado puede tener unas políticas de seguridad mucho más laxas que las que describimos aquí.

En la actualidad, los browsers imponen las siguientes restricciones a cualquier applet que se haya cargado a través de la red:

- No puede cargar librerías ni definir métodos nativos.
- No puede leer o escribir ficheros en el ordenador que lo está ejecutando.
- No puede efectuar conexiones a través de la red excepto al ordenador desde el que fue cargado.
- No puede arrancar ningún programa en el ordenador que lo está ejecutando.
- No puede leer ciertas propiedades del sistema.
- Las ventanas que un applet crea tienen un aspecto diferente a las que crean las aplicaciones locales del ordenador donde se ejecuta.

Cada browser tiene un objeto `SecurityManager` que implementa sus políticas de seguridad. Cuando un `SecurityManager` detecta una violación de estas políticas, lanza una `SecurityException`. Un applet puede capturar este tipo de excepciones y actuar en consecuencia.

Por otro lado, el API proporcionado por el paquete `java.applet` ofrece a los applets algunas capacidades que las aplicaciones normales no tienen de manera directa, como la capacidad de presentar sonido. Otras cosas que sí están permitidas a los applets son:

- Normalmente, pueden hacer conexiones a través de la red con el ordenador desde el que fueron cargados.
- Pueden cargar páginas HTML en el browser que los está ejecutando.
- Pueden invocar métodos públicos de otros applets que se encuentren en la misma página.
- Los applets que son cargados desde el sistema de ficheros local (desde un directorio en el `CLASSPATH`) no están sometidos a las restricciones a las que están sometidos los applets cargados a través de la red.
- Aunque la mayor parte de los applets se paran una vez el browser va a otra página distinta a la que los contiene, esto no tienen por qué ser así.

8.4 La marca <APPLET> en HTML

Una vez tenemos el código del applet, lo siguiente es ejecutarlo por medio de un browser o un *appletviewer*. Para ejecutar un applet, debemos añadirlo a una página HTML, usando la marca <APPLET>. El formato general de la marca <APPLET> en HTML es el siguiente, donde los campos opcionales están entre corchetes, lo que son palabras reservadas de HTML en mayúsculas y los valores que debemos proporcionar aparecen en cursiva:

```
< APPLET
  [CODEBASE = URLcodebase]
  CODE = ficheroDelApplet
  [ALT = textoAlternativo]
  [NAME = nombreDeLaInstancia]
  WIDTH = pixels
  HEIGHT = pixels
  [ALIGN = tipoDeAlineacion]
  [VSPACE = pixels]
  [HSPACE = pixels] >
[< PARAM NAME = nombreDelParametro1 VALUE = valorDelParametro1 >]
[< PARAM NAME = nombreDelParametro2 VALUE = valorDelParametro2 >]
                . . .
[textoHTMLalternativo]
</APPLET>
```

- CODEBASE = *URLcodebase*: Este atributo opcional especifica el URL base del applet, es decir, el directorio que contiene el fichero del applet. Si no se especifica, se utiliza el URL del documento.
- CODE = *ficheroDelApplet*: Este atributo obligatorio proporciona el nombre del fichero que contiene el código compilado de la subclase de Applet. El nombre es relativo a lo especificado en CODEBASE. No puede ser absoluto.
- ALT = *textoAlternativo*: Este atributo opcional especifica un texto que debe ser presentado si el browser es capaz de interpretar la marca <APPLET> pero no puede ejecutar applets Java.
- NAME = *nombreDeLaInstancia*: Este atributo opcional especifica el nombre de la instancia del applet. lo que hace posible que los applets en la misma página puedan encontrarse unos a otros y comunicarse entre sí.
- WIDTH = *pixels* HEIGHT = *pixels*: Estos atributos obligatorios proporcionan la anchura y altura iniciales (en pixels) del display del applet.
- ALIGN = *tipoDeAlineacion*: Este atributo opcional define el alineamiento del applet con el resto de los elementos de la página. Sus posibles valores son los mismos que se usan para la marca IMG de HTML: left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom.
- VSPACE = *pixels* HSPACE = *pixels*: Estos atributos opcionales especifican el margen en pixels alrededor del applet.
- <PARAM NAME = *nombreDelParametroN* VALUE = *valorDelParametorN*>: Las marcas <PARAM> son el único mecanismo para definir parámetros específicos del applet. Los applets pueden leer estos valores mediante el método

`getParameter`.

- *`textoHTMLalternativo`*: Si la página que contiene la marca `<APPLET>` es accedida por un browser que no es capaz de interpretar esta marca, el browser ignorará las marcas `<APPLET>` y `<PARAM>` e interpretará cualquier otro código HTML que encuentre entre `<APPLET>` y `</APPLET>`. Los browsers con capacidad Java ignorarán este código HTML extra.

8.5 Otros métodos en el API de un applet

El API al que tiene acceso un applet permite aprovechar la estrecha relación que existe entre los applets y los browsers de Web. Este API es proporcionado por el paquete `java.applet`, principalmente por la clase `Applet` y por el interface `AppletContext`.

8.5.1 Carga de ficheros

Cuando un applet necesita cargar algún fichero de datos que está especificado por medio de un URL relativo (un URL que no define totalmente la localización de un fichero), el applet puede utilizar la base de código o la base de documento para formar el URL completo. La base de código, que se obtiene con el método `getCodeBase` de la clase `Applet`, es un URL que especifica el directorio desde donde se cargó el código del applet. La base de documento, que se obtiene con el método `getDocumentBase` de la clase `Applet`, define el directorio desde donde se cargó la página HTML que contiene el applet.

A menos que la marca <APPLET> especifique una base de código, ésta y la base de documento hacen referencia al mismo directorio en el mismo servidor.

Por razones de seguridad, los browsers limitan los URL desde los que un applet puede leer. Por ejemplo, la mayor parte de los browsers no permiten a un applet utilizar “..” para acceder a directorios por encima de la base de código o de la de documento. También hay que tener en cuenta que, dado que un applet sólo puede efectuar conexiones con el servidor desde el que fue cargado, la base de documento no es útil si la página y el applet se cargaron desde distintos servidores.

La clase `Applet` define mecanismos adecuados para cargar ficheros con imágenes y sonidos relativos a un URL de base. Por ejemplo, la siguiente llamada cargaría el fichero “a.gif”, almacenado en el directorio “Imgs” por debajo del directorio desde el que se cargó la página que contiene el applet:

```
Image imagen = getImage(getCodeBase(), “Imgs/a.gif”);
```

8.5.2 Mostrar información de estado

Todos los browsers permiten que los applets presenten pequeñas cadenas de texto para informar de su estado. En las implementaciones actuales, esta cadena aparece en la línea de estado que suelen incluir los browsers en la parte de abajo de su ventana. Es importante notar que esta línea es compartida por todos los applets de una página y por el browser mismo.

No es recomendable colocar información crucial en la línea de estado. Si la mayor parte de los usuarios van a tener que leer la información, es recomendable ponerla dentro del display del applet. Si solamente unos pocos usuarios (posiblemente con intención de depurar el applet) van a leer la información, es más recomendable utilizar la salida estándar.

La línea de estado no es muy visible y puede ser reescrita por otros applets o por el mismo browser. Por estas razones, su uso es aconsejable solamente para información transitoria y sin mucha relevancia. Por ejemplo, un applet que carga varias imágenes podría mostrar los nombres de los ficheros a medida que va cargándolos.

Para escribir en la línea de estado se utiliza el método `showStatus`. Por ejemplo:

```
showStatus(“Cargando imagen “ + nomImg);
```

8.5.3 Cargar otros documentos en el browser

El método `showDocument` del interface `AppletContext` permite a un applet dirigir al browser para que cargue un determinado URL en una de las ventanas del browser. Esta son dos formas de `showDocument`:

- `public void showDocument(java.net.URL url)`: Hace que el browser presente la página contenida en `url`, sin especificar en qué ventana.
- `public void showDocument(java.net.URL url,String targetWin)`: En esta versión del método es posible especificar en qué ventana o *frame* HTML debe mostrarse el documento.

El parámetro `targetWin` en la segunda forma de invocar a `showDocument` puede tomar los siguientes valores:

- “`_blank`”: Muestra el documento en una ventana nueva.
- “*`nombreDeVentana`*”: Muestra el documento en una ventana llamada *`nombreDeVentana`*. Esta ventana es creada si es necesario.
- “`_self`”: Muestra el documento en la ventana y el frame que contienen el applet.
- “`_parent`”: Muestra el documento en la misma ventana que contiene el applet, pero en el frame padre del que contiene el applet. Si no hay un frame padre, actúa igual que “`_self`”.
- “`_top`”: Muestra el documento en la misma ventana que contiene el applet, pero en el frame de más alo nivel. Si éste es el que contiene el applet, tiene el mismo efecto que “`_self`”.

8.5.4 Sonido

Estos son los métodos que la clase `Applet` ofrece para cargar y reproducir sonidos. El formato con dos parámetros de los métodos toma un URL base, que es habitualmente obtenido mediante `getDocumentBase` o `getCodeBase` y la localización de un fichero relativa al URL

base. Es aconsejable utilizar la base de código para ficheros que sean parte integral del applet, mientras que la base de documento puede usarse para ficheros configurables a través de parámetros:

- `getAudioClip(URL)`, `getAudioClip(URL, String)`: Retornan un objeto que implementa el interface `AudioClip`.
- `play(URL)`, `play(URL, String)`: Reproducen el `AudioClip` correspondiente al URL especificado.

El interface `AudioClip` define los siguientes métodos:

- `loop`: Reproduce el clip repetidamente.
- `play`: Reproduce el clip una vez.
- `stop`: Detiene la reproducción del clip.

8.5.5 Parámetros de un applet

Los parámetros son para un applet equivalentes a lo que constituyen los argumentos de las aplicaciones. Permiten adecuar la ejecución del applet sin necesidad de cambiar su código. Por medio del uso de parámetros es posible aumentar la flexibilidad de un applet, permitiendo su uso en diferentes situaciones.

Los applets utilizan el método `getParameter` de la clase `Applet` para obtener los parámetros proporcionados a través de la página HTML en la que están incluidos. El método `getParameter` se define así:

```
public String getParameter(String name)
```

Del mismo modo que para los argumentos de una aplicación, un applet puede necesitar convertir la cadena que constituye uno de sus parámetros en un valor de otro tipo, como un entero o un valor booleano.

El método `getParameter` no sólo puede usarse para acceder a los valores definidos mediante marcas `<PARAM>`, sino también a los valores de los atributos de la marca `<APPLET>`.

8.6 Threads y applets

Cualquier applet es ejecutado, usualmente, en varios threads. Los métodos de dibujo de un applet (`paint` y `update`) son llamados desde el thread de AWT. Los threads que llaman a los métodos que controlan el ciclo de vida (`init`, `start`, `stop` y `destroy`) dependen del browser que estemos utilizando.

Muchos browsers crean un thread por cada applet que encuentran en una página, utilizando el mismo thread para todos los métodos del ciclo de vida. Algunos crean un grupo de threads específico para cada applet, de manera que es fácil terminar con todos los threads que pertenecen al mismo applet. En cualquier caso, se garantiza que cualquier thread que cree uno de los métodos del ciclo de vida pertenece al mismo grupo de threads.

Uno de los motivos fundamentales para que un applet cree diferentes threads es para llevar a cabo tareas que consuman tiempo sin resultados visibles. Imaginemos un applet que necesita cargar una serie de imágenes en su método `init`. El thread que llama a `init` no puede hacer otra cosa hasta que retorne el método. Esto significa que el browser no puede presentar el applet hasta que todas las imágenes se hayan cargado. Por tanto, tiene sentido permitir que el applet realice otras tareas mientras las imágenes son transportadas por la red.

Otro caso es cuando no es deseable que toda la ejecución del applet se detenga al salir de la página desde la que se cargó. Consideremos un applet que abre una nueva ventana en la pantalla del usuario para permitirle hacer cálculos mientras accede a otras páginas. Si el usuario se mueve a otra página y la ventana en cuestión no está corriendo en un thread separado, al dejar la página desde la que se cargó el applet, el método `stop` es llamado y la ejecución se detiene.

Veamos un ejemplo de un applet (`Reloj`) que implementa el interface `Runnable` (y por tanto el método `run`) que vimos en el capítulo anterior. En su método `init`, `Reloj` crea un nuevo objeto `Thread` consigo mismo como objetivo. Cuando se crea un `Thread` de esta manera, el método `run` es el del objetivo:

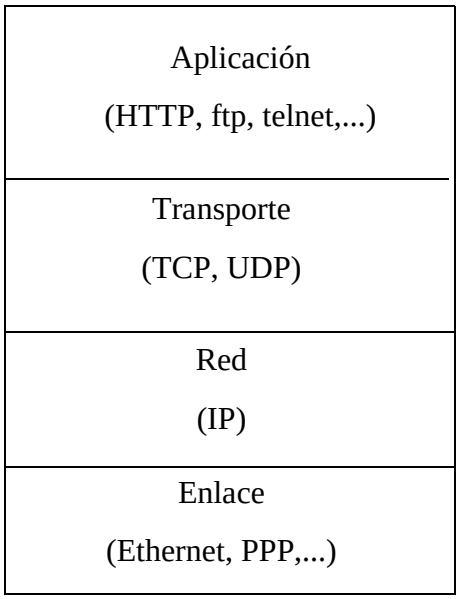
```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;
public class Reloj extends Applet implements Runnable {
   private Thread rThread = null;
   public void start() {
      if (rThread == null) {
         rThread = new Thread(this, “Reloj”);
         rThread.start();
      }
   }
   public void run() {
      Thread lThread = Thread.currentThread()
      while (rThread == lThread) {
         repaint();
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e){ }
      }
   }
   public void paint(Graphics g) {
      Calendar cal = Calendar.getInstance();
      Date hora = cal.getTime();
      DateFormat hForm = DateFormat.getTimeInstance();
      g.drawString(hForm.format(hora), 5, 10);
   }
   public void stop() {
      rThread = null;
   }
}
```

El método `run` del applet `Reloj` permanece en un bucle hasta que el browser lo para mediante el método `stop` de la clase `Applet`, es decir, al dejar la página desde donde fue cargado. Durante cada iteración del bucle, el reloj redibuja el display. El método `paint` obtiene la hora, le da formato de acuerdo a los datos locales y la presenta. Si en algún momento después de dejar la página volvemos a ella, el reloj volverá a aparecer mostrando la hora actual.

COMUNICACIONES

9.1 Conceptos básicos

Los ordenadores conectados a una internet (incluida la propia ***Internet***) se comunican utilizando TCP (*Transport Control Protocol*) o UDP (*User Datagram Protocol*), como se muestra en el diagrama:



Lo habitual es que un programa Java que realiza conexiones a través de la red se sitúe en el nivel de aplicación. No es usual que el programador deba preocuparse por los protocolos de nivel más bajo. Las clases en el paquete `java.net` proporcionan mecanismos de comunicación que abstraen estos detalles. Sin embargo, conviene tener algunos conceptos claros para poder elegir los modos de comunicación y emplear adecuadamente las clases que proporciona el JDK.

9.1.1 TCP

Cuando dos aplicaciones desean comunicarse de manera fiable establecen un circuito y escriben y leen sobre él. El protocolo TCP garantiza que los datos enviados desde uno de los extremos de la comunicación van a llegar al otro y en el mismo orden en el que fueron enviados. En otro caso, el protocolo se encarga de avisar de que ha ocurrido un error.

Por tanto, TCP proporciona un canal punto a punto (un *circuito virtual*) entre dos aplicaciones que necesitan una comunicación fiable. Para ello requiere que, antes de transferir los datos, se establezca una conexión entre las dos aplicaciones. Por ejemplo, el protocolo HTTP y el FTP utilizan este sistema de transporte de la información.

9.1.2 UDP

El protocolo UDP proporciona una comunicación no garantizada entre dos aplicaciones a través de la red. UDP no está orientado a la conexión como TCP. En lugar de eso, envía paquetes específicos de datos (llamados ***datagramas***) desde una aplicación a la otra. Cada datagrama es independiente de los demás datagramas que se hayan intercambiado antes o que puedan intercambiarse en el futuro. De hecho, el orden de entrega no está garantizado.

Para muchas aplicaciones (como el envío de un página HTML) el orden de los datos y su entrega completa son aspectos críticos, pero otros tipos de aplicaciones no exigen estas características y resulta deseable evitar la sobrecarga que supone mantener una conexión fiable. Por ejemplo, consideremos un servidor que envía la hora cuando se la pide uno de sus clientes. Si el cliente pierde algún paquete, no es un problema grave dado que puede pedirlo de nuevo y, de hecho, la retransmisión del paquete perdido no tendría sentido.

9.1.3 Puertos

En general, un ordenador tiene una única conexión a la red (a lo sumo unas pocas conexiones). Todo los datos destinados al ordenador llegan a través de esa conexión, aunque estén dirigidos a diferentes aplicaciones. El ***puerto*** al que van dirigidos esos datos permite identificar a qué aplicación deben entregarse.

Los datos que se transfieren en una internet llevan para su encaminamiento dos clases de direcciones:

- Una dirección IP de 32 bits que identifica el ordenador al que van destinados.
- Un número de puerto de 16 bits, que es utilizado por TCP o UDP para identificar la aplicación a la que deben entregarse.

En las comunicaciones orientadas a conexión, como es el caso de TCP, un servidor se asocia a un ***socket*** que corresponde a un determinado puerto. De esta manera la aplicación se registra en el sistema para recibir todos los paquetes destinados a ese puerto. Un cliente que desee conectarse al servidor solicita una conexión a ese determinado puerto. En las comunicaciones basadas en datagramas (UDP) el servidor se registra con el sistema y el cliente dirige los datos a ese puerto determinado.

Los números de puerto van de 0 a 65.535. Por norma, los puertos del 0 al 1023 se consideran restringidos y están asociados a lo que se denominan ***servicios bien conocidos***, es decir, a servidores que responden a aplicaciones muy extendidas en la red. Por ejemplo el puerto 80 corresponde a HTTP.

9.1.4 URLs

URL es el acrónimo de *Uniform Resource Locator* y constituye una referencia a un recurso disponible en Internet. Un URL consta esencialmente de dos partes separadas por dos puntos (el carácter ‘`:`’):

```
esquema:informaciónDependienteDelEsquema
```

El esquema define el tipo del recurso y el modo de acceso que debe emplearse y suele corresponder al nombre de un protocolo o servicio, como `ftp`, `http`, `gopher`, etc.

La información dependiente del esquema varía con el tipo de esquema que se use. La mayor parte de los esquemas incluyen información de dos tipos: la localización (por nombre o por dirección IP) del servidor donde se encuentra el recurso, y la forma de acceder al recurso una vez conectados al servidor. Por ejemplo:

```
ftp://ftp.cica.es/pub/java-linux/JDK1.1.7/
mailto:drlopez@cica.es
http://www.cica.es/congresos/registro.phtml
```

9.2 Clases en el paquete `java.net`

Como ya ha quedado dicho, las clases contenidas en el paquete `java.net` ofrecen un conjunto de mecanismos para que un programa Java pueda recibir y enviar datos a través de una red. Aquí presentaremos las características principales de las clases más importantes, desde el nivel de abstracción más alto a los que permiten trabajar más directamente con TCP y UDP.

9.2.1 URLs

La clase `java.net.URL` permite a un programa Java acceder a un recurso en Internet (o en una internet cualquiera) e interactuar con él. La manera más simple de crear un objeto de clase `URL` es mediante un constructor que emplea un `String`:

```
URL cica = new URL (“http://www.cica.es/”);
```

Un objeto `URL` puede crearse también relativo a otro `URL` ya existente. Así, por ejemplo, el recurso `http://www.cica.es/congresos/registro.phtml` podría ser accedido del siguiente modo:

```
URL cong = new URL (cica, “congresos/registro.phtml”);
```

Si el argumento del constructor es `null` o se refiere a un protocolo no disponible se lanza una `MalformedURLException`.

Existen otros constructores que permiten crear un objeto `URL` a partir de sus componentes y métodos para descomponer un `URL` dado:

- `getProtocol()`
- `getHost()`
- `getPort()`
- `getFile()`
- `getRef()`

Conviene hacer dos puntualizaciones importantes en relación con la clase `URL`. En primer lugar, el tratamiento que Java da a los URLs está bastante centrado en HTTP (de ahí los métodos de descomposición). En segundo lugar, no hay métodos para modificar los componentes de un objeto `URL`: para acceder a otro recurso debe instanciarse un nuevo objeto.

Para interaccionar con el recurso identificado en el `URL` puede emplearse el método `openStream()`, que devuelve un `inputStream` para acceder a los contenidos del recurso. Por ejemplo, el siguiente programa permite visualizar la página principal de la Web del CICA:

```
import java.net.*;
import java.io.*;
public class CICAMain {
   public static void main (String[] args) throws Exception {
      URL cica = new URL("http://www.cica.es/");
      BufferedReader in = new BufferedReader(
                          new InputStreamReader(
                          cica.openStream()));
      String linea;
      while ((linea = in.readLine()) != null)
      System.out.println(linea);
      in.close();
   }
}
```

Otro mecanismo más completo de comunicación a través de la clase `URL` implica a la clase `URLConnection`, mediante la cual se pueden abrir streams de entrada y salida con el recurso. El ejemplo anterior, usando `URLConnection` quedaría:

```
import java.net.*;
import java.io.*;
public class CICAMain {
   public static void main (String[] args) throws Exception {
      URL cica = new URL("http://www.cica.es/");
      URLConnection cc = cica.openConnection ();
      BufferedReader in = new BufferedReader(
                          new InputStreamReader(
                          cc.getInputStream()));
      String linea;
      while ((linea = in.readLine()) != null)
      System.out.println(linea);
      in.close();
   }
}
```

La diferencia aquí es que la clase `URLConnection` ofrece también un método `getOutputStream()` mediante el cual es posible escribir hacia el recurso, como puede ser el caso, por ejemplo, de un CGI.

9.2.2 Sockets

Un ***socket*** modela uno de los extremos de una conexión punto a punto entre dos programas. Las clases Java relacionadas con sockets se emplean para representar conexiones entre un cliente y un servidor en una internet. El paquete `java.net` proporciona dos tipos de sockets: uno para el lado cliente de la conexión y otro para el lado servidor.

El comportamiento de un cliente debe seguir, en esencia, el siguiente esquema:

1. Abrir un socket.

2. Abrir un stream de salida y un stream de entrada sobre el socket.
3. Leer y escribir de los streams de acuerdo con el protocolo.
4. Cerrar los streams.
5. Cerrar el socket.

El siguiente ejemplo ilustra cómo funcionaría un cliente que se conecte a un servidor que simplemente devuelve todos los datos que se le envían:

```
import java.io.*;
import java.net.*;

public class EcoClient {
  public static void main(String[] args) throws IOException {

    Socket socket = null;
    PrintWriter out = null;
    BufferedReader in = null;

    try {
      socket = new Socket("ecohost", 7);
      out = new PrintWriter(socket.getOutputStream(), true);
      in = new BufferedReader(new
           InputStreamReader(socket.getInputStream()));
    }
    catch (UnknownHostException e) {
      System.err.println("Host desconocido");
      System.exit(1);
    }
    catch (IOException e) {
       System.err.println("Error de E/S");
       System.exit(1);
    }

    BufferedReader stdIn = new BufferedReader(new
                           InputStreamReader(System.in));
    String usIn;

    while ((usIn = stdIn.readLine()) != null) {
      out.println(usIn);
      System.out.println("eco: " + in.readLine());
    }
    out.close();
    in.close();
    stdIn.close();
    socket.close();
  }
}
```

Desde el lado del servidor, lo importante es que lo primero que debe hacerse es utilizar un `ServerSocket`:

```
try {
   serverSocket = new ServerSocket(4444);
```

```
}
catch (IOException e) {
   System.out.println("No puedo excuchar en el puerto 4444");
   System.exit(-1);
}
```

El constructor de `ServerSocket` lanza una excepción si no es posible escuchar en el puerto especificado. Una vez el servidor se ha asociado a un puerto, puede proceder a aceptar conexiones desde un cliente:

```
Socket clientSocket = null;
try {
   clientSocket = serverSocket.accept();
}
catch (IOException e) {
   System.out.println("Accept ha fallado para el puerto: 4444");
   System.exit(-1);
}
```

El método `accept` espera hasta que un cliente pide una conexión y ésta se establece. Una vez la conexión se ha establecido, `accept` devuelve un `Socket` que puede emplearse como veíamos en el ejemplo del cliente para realizar la comunicación.

Esta versión de servidor que vemos aquí es capaz de atender únicamente a un cliente a la vez. Si otro cliente trata de conectarse mientras el servidor está conectado, la máquina virtual Java lo pondrá en espera siempre que no se supere el número establecido por la cola de *backlog* (que es un parámetro que puede especificarse en la construcción del objeto). Si la cola está llena, la conexión es rechazada.

Para disponer de un servidor capaz de atender múltiples conexiones la lógica básica del programa debe ser del tipo:

```
while (true) {
   aceptar una conexion;
   crear un thread que hable con el cliente;
}
```

El thread se encarga del diálogo con el cliente y termina al finalizar la conexión.

9.2.3 Datagramas

Un datagrama es un mensaje autocontenido e independiente enviado a través de la red y cuya entrega, momento de llegada y contenido no están garantizados.

Las clases `java.net.DatagramSocket` y `java.net.DatagramPacket` se utilizan para enviar y recibir datagramas a través de la red. Consideremos el siguiente ejemplo de un servidor que acepta datagramas y devuelve un datagrama con la fecha y hora actuales:

```
byte[] bufr = new byte[256];
byte[] bufs = new byte[256];

try {
// asociación al socket
   DatagramSocket socket = new DatagramSocket(4444);
// recibe la petición
```

```
   DatagramPacket pq = new DatagramPacket(bufr, bufr.length);
   socket.receive(pq);
// envía la respuesta
   bufs = new Date().toString().getBytes();
   InetAddress dir = pq.getAddress();
   int puerto = pq.getPort();
   pq = new DatagramPacket(bufs, bufs.length, dir, puerto);
   socket.send(pq);
}
catch (IOException e) {
   e.printStackTrace();
   socket.close();
}
```

Al principio se crean dos arrays de bytes para realizar la comunicación por medio de datagramas y se abre un `DatagramSocket` asociado con el puerto en el que queremos que el servidor espere las peticiones. Utilizando uno de los buffers se instancia un `DatagramPacket` para recibir ahí las peticiones del cliente y el código llama al método `receive` que espera hasta que hay un datagrama disponible. Una vez se ha recibido un datagrama, se prepara el buffer de salida y se obtiene la dirección IP y el puerto del cliente, mediante sendas llamadas a `getAddress` y `getPort`. Por último se genera el datagrama de respuesta usando un nuevo constructor de `DatagramPacket` que permite especificar la dirección IP y el puerto de destino.

El código de un cliente que permitiría utilizar este servidor es el siguiente:

```
public class Cliente {
  public static void main(String[] args) throws IOException {
  if (args.length != 1) {
    System.out.println("Uso: java Cliente <hostname>");
    return;
  }
  DatagramSocket socket = new DatagramSocket();
  byte[] buf = new byte[256];
  InetAddress dir = InetAddress.getByName(args[0]);
  DatagramPacket pq = new DatagramPacket(buf, buf.length, dir, 4444);
  socket.send(pq);
  pq = new DatagramPacket(buf, buf.length);
  socket.receive(packet);
  String fecha = new String(packet.getData(), 0);
  System.out.println("Fecha: " + fecha);
  socket.close();
 }
}
```

La dirección IP de destino del datagrama de petición se obtiene a partir del nombre del servidor proporcionado por medio de la línea de comandos utilizando el método `getByname` de la clase `InetAddress`. El datagrama de petición no contiene datos, dado que el servidor no espera ningún dato específico. Para obtener la respuesta del servidor, el cliente crea un nuevo paquete y utiliza el método `receive`. Normalmente, un cliente pone algún temporizador para esperar la respuesta, dado que el método `receive` espera indefinidamente. Cuando la respuesta se ha recibido, se emplea el método `getData` para obtener los datos que contiene y mostrarlos.

ACCESO A BASES DE DATOS: JDBC

JDBC (*Java Database Connectivity*) es la solución Java para la explotación de bases de datos y está compuesto por un conjunto de clases e interfaces contenidos en el paquete `java.sql`, incluido en la distribución estándar Java desde el JDK 1.1.

El objetivo de JDBC es seguir ofreciendo las características de neutralidad frente a la arquitectura y de portabilidad del lenguaje en aplicaciones orientadas a las bases de datos, de manera que los programas sean independientes también del gestor de base de datos empleado. A través de JDBC es posible:

- Establecer conexiones con las bases de datos.
- Enviar sentencias SQL hacia las bases de datos.
- Procesar los resultados obtenidos de estas sentencias.

Los objetivos de neutralidad y portabilidad chocan con el hecho de cada SGBD suele utilizar un dialecto particular de SQL, por lo que JDBC no efectúa ningún control sobre las sentencias SQL que envía a la base de datos, y lo hace incluso si están mal construidas. Para paliar esto, siquiera parcialmente, se proporcionan mecanismos para obtener metainformación sobre la base de datos.

10.1 El paquete `java.sql`

Las principales clases e interfaces del paquete `java.sql` son las siguientes:

- `Driver`: Implementa la comunicación entre la aplicación Java y la base de datos.
- `DriverManager`: Encargado de gestionar el conjunto de drivers que están disponibles para la aplicación.
- `DriverPropertyInfo`: Proporciona información acerca de las características y posibilidades que ofrece un driver.
- `Connection`: Modela una conexión activa con una base de datos.
- `Statement`: Ofrece mecanismos para ejecutar sentencias SQL sencillas.
- `PreparedStatement`: Ofrece mecanismos para ejecutar sentencias SQL precompiladas y parametrizables.
- `CallableStatement`: Permite ejecutar llamadas a procedimientos almacenados en la base de datos.
- `ResultSet`: Permite acceder a los resultados obtenidos por la ejecución de una sentencia SQL `SELECT`.
- `DatabaseMetaData`: Proporciona información de las posibilidades y características de los elementos que intervienen en la conexión con la base de datos. También ofrece mecanismos para establecer la semántica de las transacciones en los

SGBD que las soportan.

- `ResultSetMetaData`: Proporciona información sobre los resultados producidos por una sentencia SQL `SELECT`.

10.2 Uso de JDBC

10.2.1 Conexiones

En general, el algoritmo que se sigue para trabajar con bases de datos a través de JDBC es del tipo:

```
cargar controlador(es);
establecer conexion(es);
while (!condicionDeFin) {
   solicitar informacion;
   procesar informacion;
   if (necesario) actualizar informacion;
}
cerrar conexion(es);
```

La carga del o los drivers adecuados para conectar con la base de datos se suele hacer por medio del método estático `forName` de la clase `Class`, que permite cargar una clase disponible en el `CLASSPATH` dentro de la máquina virtual:

```
Class.forName (“sun.jdbc.odbc.JdbcOdbcDriver”);
```

DriverManager ofrece el método estático `getConnection (String url)` para abrir una conexión con la base de datos. El formato de un URL de conexión JDBC es el siguiente:

```
jdbc:protocoloODBC:nombreDeLaBaseDeDatos
```

Donde `protocoloODBC` designa el mecanismo de conectividad a la base de datos y depende de los drivers disponibles. Un ejemplo típico de llamada para abrir una conexión es:

```
DriverManager.getConnection (“jdbc:odbc:personal”);
```

Un mecanismo alternativo es instanciar un driver determinado y utilizar su método connect. Este método debe emplearse con algunas máquinas virtuales (como la del MSIE) que retardan la inicialización de los miembros estáticos de las clases cargadas con `Class.forName` hasta la primera instanciación de la clase:

```
Driver drv = new ClaseDelDriver ();
Connection con = drv.connect (“jdbc:odbc:personal”, null);
```

10.2.2 Sentencias SQL

Una vez se ha abierto la conexión, el interface `Connection` ofrece métodos para crear sentencias SQL que serán posteriormente ejecutadas. El interface `Statement` se usa para ejecutar sentencias simples y se crea por medio de:

```
Statement st = con.createStatement ();
```

Una vez se ha instanciado, puede utilizarse el objeto de clase `Statement` para ejecutar sentencias SQL expresadas directamente mediante cadenas:

```
st.executeUpdate (“DROP TABLE pruebas”);
st.executeQuery (“SELECT * from retenciones”);
```

`PreparedStatement` permite la ejecución de sentencias precompiladas, donde pueden especificarse uno o más parámetros de entrada que se identifican en el fuente SQL con el carácter `?`:

```
PreparedStatement pst = con.prepareStatement (“INSERT INTO personal
(dni, nombre) VALUES (?,?)”);
```

Antes de ejecutar una sentencia por medio de una `PreparedStatement` es necesario proporcionar los valores de los parámetros. Para ello se usan los métodos `setXXX` (donde `XXX` es el tipo de datos apropiado: `setInt`, `setString`, `setDate`, etc.) que aceptan dos parámetros:

- El orden del parámetro dentro del objeto (`int`).
- El valor que debe asignarse al parámetro.

Para ejecutar sentencias pueden emplearse tres métodos diferentes:

- `executeUpdate`: es el indicado para sentencias SQL que actualizan la base de datos, como `INSERT`, `UPDATE`, `DELETE` (el método retorna el número de líneas afectadas), `CREATE TABLE` o `DROP TABLE` (en estos casos el método retorna cero).
- `executeQuery`: es el indicado para sentencias que producen un conjunto de resultados, como `SELECT`. El método retorna un `ResultSet`.
- `execute`: es utilizado para ejecutar sentencias que devuelven más de un conjunto de resultados o más de un contador de líneas afectadas.

10.2.3 Resultados

Como hemos visto, el resultado de ejecutar una sentencia SQL por medio de JDBC puede ser de dos tipos:

- Un entero, que indica el número de líneas afectadas.
- Un objeto de clase `ResultSet`, que contiene las filas que cumplieron la condición de la sentencia SQL ejecutada.

El objeto `ResultSet` mantiene un cursor que apunta a la fila de datos actual. Este cursor sólo se puede mover hacia abajo una fila cada vez, llamando al método `next`. Inicialmente, el cursor se encuentra posicionado antes de la primera fila. El método `next` retorna un valor booleano que indica si existe la siguiente fila. Una vez el cursor posicionado en la fila correspondiente, los valores de los campos pueden leerse mediante métodos `getXXX`, similares a los `setXXX` que veíamos antes, aunque en este caso aceptan un solo parámetro (la posición del campo en la fila) y devuelven un objeto del tipo correspondiente.

Como ejemplo, veamos un programa que realiza una conexión a una base de datos y muestra algunos datos de la misma:

```
import java.sql.*;

public class Consulta {
   static final String DRIVER= “sun.jdbc.odbc.JdbcOdbcDriver”;
   static final String DBURL = “jdbc:odbc:personal”;
   protected Connection con = null;

   public Consulta () throws SQLException, ClassNotFoundException{
      Class.forName (DRIVER);
      con = DriverManager.getConnection (DBURL);
   }

   public void printDatos () throws SQLException {
      Statement sen = con.createStatement ();
      ResultSet res = sen.executeQuery (“SELECT codigo, nombre,
                                         telefono FROM ids”);
      while (res.next ()) {
         int cod = res.getInt (1);
         String nom = res.getString (2);
         String tlf = res.getString (3);
         System.out.println (cod+“...”+“...”+nom+“...”+tlf);
      }
   }

   public static void main (String [] args) {
      try {
         Consulta cs = new Consulta ();
         cs.printDatos ();
      }
      catch (SQLException e) {
         System.out.println (“Error SQL ” + e.getMessage ());
      }
      catch (ClassNotFoundException e) {
         System.out.println (“Driver no disponible” +
                             e.getMessage ());
      }
   }
}
```

10.3 Tipos de drivers JDBC

Existen diferentes tipos de drivers JDBC, cada uno de los cuales es más o menos indicado para ciertas aplicaciones. De acuerdo con Javasoft, los drivers JDBC pueden ser:

- Puentes JDBC-ODBC, que proporcionan acceso JDBC a través de drivers ODBC. En general, necesitan que los ejecutables del driver ODBC se encuentren en el cliente, por lo que son poco indicados para applets o para cualquier entorno en que el código deba ser cargado directamente. Su principal aplicación reside en intranets o integrados en servidores de aplicaciones.
- Drivers basados en un API nativo, que convierten las llamadas JDBC en llamadas al

API de un cliente local de la base de datos. Su rango de aplicaciones es el mismo que en el caso anterior.

- Drivers Java basados en protocolos de red, que traducen las llamadas JDBC en un protocolo independiente del SGBD que, posteriormente, es transformado en llamadas al SGBD por un servidor. Esta es la alternativa más flexible, aunque implica cuestiones relativas a seguridad, accesos a través de firewalls, etc.
- Drivers Java basados en protocolos nativos, que transforman directamente las llamadas JDBC en el protocolo de red propietario utilizado por el SGBD, permitiendo una llamada directa desde el cliente al servidor de la base de datos. Aunque puede ser más rápido, ofrecer acceso de esta manera a través de Internet puede suponer un mayor compromiso de seguridad.

APÉNDICE I. Nociones sobre GUIs

Modelos de manejo de eventos

Manejo de eventos en JDK 1.0

Cuando un usuario actúa sobre un componente de una interface de usuario en JDK 1.0 (un objeto de la clase `Component`) se crea un objeto de clase `Event`. El sistema de manejo de eventos del AWT pasa el `Event` hacia arriba en la jerarquía de componentes, de manera que cada uno de ellos tiene ocasión de reaccionar al evento.

Los manejadores de eventos de cada componente pueden reaccionar a un evento en cualquiera de las siguientes formas:

- Ignorando el evento y permitiendo que pase hacia arriba en la jerarquía de componentes, hacia los componentes que lo contienen.
- Modificando la instancia de `Event` antes de que siga hacia arriba en la jerarquía.
- Reaccionando de una manera determinada al evento.
- Interceptando el evento, de manera que no sea procesado más arriba en la jerarquía de componentes.

Desde el punto de vista de un `Component`, el sistema de manejo de eventos del AWT es un sistema de filtro de eventos. La parte del AWT dependiente de la plataforma genera un evento, pero los componentes tienen la oportunidad de modificarlo, reaccionar frente a él o interceptarlo. Hay que tener en cuenta que, aunque el AWT define una gran variedad de eventos, es posible que no pueda verlos todos: solamente serán detectados aquéllos que la plataforma proporcione.

El objeto `Event`

Como hemos visto, cada evento se traduce en la creación de un objeto de clase `Event`. Uno de estos objetos contiene información sobre:

- El tipo de evento, que puede ser simple (un click de ratón) o más abstracto, como una *acción* o la iconificación de una ventana.
- El objeto para el que se produjo el evento (su *target*).
- Una marca de tiempo que indica cuándo se produjo el evento.
- La localización dónde se produjo el evento. Las coordenadas son relativas al origen del `Component` al que se le está pasando el evento.
- La tecla que se pulsó (para eventos relacionados con el teclado).
- Un argumento arbitrario (como puede ser la etiqueta del componente para el que se

produjo)

- El estado de las teclas de modificación (como “Alt” y “Control”) cuando el evento se produjo.

La clase `Component` define muchos métodos de manejo de eventos y cualquiera de ellos puede ser reescrito para definir los mecanismos de reacción de un programa a las acciones del usuario. Excepto en el caso de un método general (`handleEvent()`), cada método de manejo de eventos es utilizado para un tipo particular de evento. Es recomendable utilizar estos últimos métodos, ya que utilizar el manejador general puede suponer efectos laterales indeseables.

La clase `Component` define los siguientes métodos para responder a los correspondientes tipos de eventos:

Método	Tipo de evento
`action()`	`Event.ACTION_EVENT`
`mouseEnter()`	`Event.MOUSE_ENTER`
`mouseExit()`	`Event.MOUSE_EXIT`
`mouseMove()`	`Event.MOUSE_MOVE`
`mouseDown()`	`Event.MOUSE_DOWN`
`mouseDrag()`	`Event.MOUSE_DRAG`
`mouseUp()`	`Event.MOUSE_UP`
`keyDown()`	`Event.KEY_PRESS, Event.KEY_ACTION`
`keyUp()`	`Event.KEY_RELEASE, Event.KEY_ACTION_RELEASE`
`gotFocus()`	`Event.GOT_FOCUS`
`lostFocus()`	`Event.LOST_FOCUS`
`handleEvent()`	Todos

Cuando se produce un evento, el método de manejo del tipo apropiado es llamado. De hecho, el evento pasa primero a través del método `handleEvent()`, el cual (en la implementación por defecto) llama al método apropiado al tipo de evento que se ha producido.

El método `action()` es especialmente importante en el diseño de interfaces de usuario. Solamente los elementos básicos de control (`Button`, `Checkbox`, `Choice`, `List`, `MenuItem` y `TextField`) producen eventos de acción. Por ejemplo, cuando el usuario hace click sobre un botón se genera un evento de acción. Reescribiendo el método `action()` es posible reaccionar a las acciones del usuario sin necesidad de tener en cuenta los eventos de bajo nivel (pulsación de teclas o clicks de ratón) que las causaron.

Todos los métodos de manejo de eventos reciben al menos un argumento (de clase `Event`) y devuelven un valor booleano. Este valor de retorno indica si el método ha tratado completamente el evento o no. Si retorna `false`, el manejador indica que el evento debe seguir progresando hacia arriba en la jerarquía de componentes. Si retorna `true`, indica que el evento no debe ser enviado más arriba.

Manejo de eventos en JDK 1.1

El nuevo modelo de manejo de eventos que proporciona JDK 1.1 está basado en la distinción entre fuentes de eventos y manejadores de eventos (*event listeners*). Uno o más manejadores se pueden registrar para ser notificados cuando se produzcan eventos de un tipo concreto en una determinada fuente de eventos.

Los manejadores de eventos pueden ser instancias de cualquier clase. El único requisito es que implementen el interface de manejo para el tipo de eventos en cuestión. En cualquier programa que incluya un manejador de eventos es necesario incluir los siguientes segmentos de código:

- En la declaración de la clase que actúa como manejador de eventos es necesario especificar que la clase implementa un interface de manejo (o que extiende a una clase que la implementa). Por ejemplo:

```
public class UnaClase implements ActionListener {
```

- Es necesario registrar una instancia de la clase que actúa como manejador de uno o más componentes AWT. Por ejemplo:

```
unComponente.addActionListener(instanciaDeUnaClase);
```

- Es necesario proporcionar los métodos del interface. Por ejemplo:

```
public void actionPerformed(ActionEvent e) {
       ...//Código que trata la acción
       ...
}
```

Este modelo es mucho más simple y flexible que el del JDK 1.0. Puede definirse cualquier número de manejadores de eventos para cualquier tipo de eventos provenientes de cualquier fuente de eventos. Por ejemplo, un programa puede crear un manejador para cada fuente de eventos, o definir un único manejador para todos los eventos de todas las posibles fuentes, o incluso definir más de un manejador para un determinado tipo de eventos de una fuente particular.

JFC (*Java Foundation Classes*)

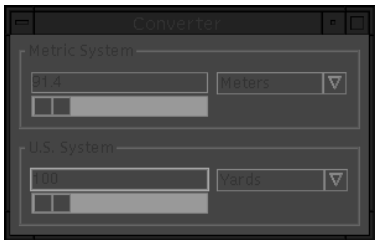
Las *Java Foundation Classes* son un conjunto de facilidades para permitir una construcción más fácil y potente de interfaces gráficos de usuario. Contienen los siguientes elementos:

- Los componentes Swing, un conjunto de elementos para construir GUIs (botones,

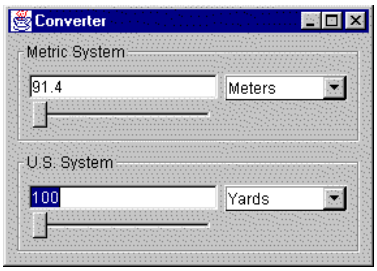
paneles, áreas de texto, etc.) que no utilizan código nativo.

- Soporte para *look & feel* configurable, de manera que (dependiendo de las preferencias del usuario) el interface tenga un aspecto diferente sin necesidad de recompilar el programa. Aquí tenemos algunos ejemplos de los aspectos comúnmente disponibles:

Java Look & Feel



Motif Look & Feel



Windows Look & Feel

- Un API de accesibilidad, que permite el uso de tecnologías para personas discapacitadas.
- Un API 2D para incorporar de manera sencilla gráficos e imágenes.
- Soporte para *Drag & Drop* desde aplicaciones nativas.

Swing

En esencia, los componentes Swing son similares a los componentes del AWT, pero presentan una característica fundamental frente a ellos: dado que no usan código nativo no están restringidos al uso del diseño del mínimo común denominador (uno de los aspectos más criticados del AWT), es decir, a tener solamente las características comunes a todas las plataformas. El modelo de tratamiento de eventos en Swing es el que hemos visto para JDK 1.1, basado en manejadores de eventos.

Como regla general, todos los componentes AWT tienen un equivalente Swing que empieza por la letra 'J' (así, `Button` y `JButton`), además de que Swing ofrece un número mayor de componentes y un conjunto de métodos estáticos de conveniencia para tareas simples como, por ejemplo, hacer que aparezca un diálogo modal. En general, resulta recomendable escribir los interfaces de usuario usando Swing, salvo en el caso de applets para los que debamos tener la garantía que van a poder ser ejecutados en cualquier browser, ya que solamente as versiones más recientes soportan Swing.