Procedimientos iniciales con Java™

VERSIÓN 8

Borland®
JBuilder®

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

En el archivo `Deploy.html` ubicado en el directorio raíz del producto JBuilder encontrará una lista completa de archivos que se pueden distribuir de acuerdo con la licencia de JBuilder y la limitación de responsabilidad.

Borland Software Corporation puede tener patentes concedidas o en tramitación sobre los temas tratados en este documento. La entrega de este documento no otorga ninguna licencia sobre estas patentes. Diríjase al CD del producto o al cuadro de diálogo Acerca de para la lista de patentes. La modificación de este documento no le otorga derechos sobre las licencias de estas patentes.

Si desea más información acerca de las condiciones de contrato de terceras partes y acerca de la limitación de responsabilidades, consulte las notas de esta versión en su CD de instalación de JBuilder.

Impreso en Irlanda.

JBE0080WW21000gsjava 4E4R1002
0203040506-9 8 7 6 5 4 3 2 1
PDF

Índice de materias

Capítulo

1

Introducción a Java

Java es un lenguaje de programación *orientado a objetos*. Cambiar a la programación orientada a objetos (OOP) desde otros métodos de programación puede ser difícil. Java se centra en la creación de objetos (estructuras de datos o comportamientos) que pueden ser evaluados y manipulados por el programa.

Al igual que otros lenguajes de programación, Java proporciona soporte para la lectura y la escritura de datos desde y hacia distintos dispositivos de entrada y salida. Java utiliza procesos que incrementan la eficiencia de los procesos de entrada/salida, facilita la internacionalización y proporciona mejor soporte para plataformas no UNIX. Java inspecciona el programa mientras se ejecuta y libera automáticamente memoria que ya no se necesita. Esto significa que no hay que seguir el rastro de los punteros de memoria ni liberarla manualmente. Esta característica implica que un programa es menos proclive a bloquearse y que la memoria no se puede desaprovechar intencionadamente.

Este libro está destinado a servir de introducción general al lenguaje de programación Java a los programadores que utilizan otros lenguajes. The Borland Community site provides an annotated list of books on Java programming and related subjects at `http://community.borland.com/books/java/0,1427,c|3,00.html`. Examples of applications, APIs, and code snippets are at `http://codecentral.borland.com/codecentral/ccweb.exe/home`.

Este libro incluye los siguientes capítulos:

- Sintaxis de Java: Capítulo 2, "Elementos del lenguaje Java", Capítulo 3, "Estructura del lenguaje Java" y Capítulo 4, "Control en el lenguaje Java".

  Estos tres capítulos definen la sintaxis básica de Java e introducen en los conceptos básicos de la programación orientada a objetos. Cada sección se divide en dos partes principales: "Términos" y "Aplicación de conceptos".

"Términos" aumenta el vocabulario, adding a los que ya se comprenden. "Aplicación de conceptos" demuestra el uso de los conceptos presentados hasta ese punto. Algunos conceptos son repasados varias veces, con niveles crecientes de complejidad.

- Capítulo 5, "Las bibliotecas de clase Java"

  Este capítulo presenta una visión general de las bibliotecas de clases Java 2 y de las ediciones de la Plataforma Java 2.

- Capítulo 6, "Programación orientada a objetos en Java"

  Este capítulo realiza una introducción a las características de orientación a objetos de Java. Creará clases Java, instanciará objetos, y accederá a variables miembro en un breve tutorial. Aprenderá a utilizar la herencia para crear nuevas clases, a utilizar interfaces para añadir nuevas capacidades a sus clases, a utilizar el polimorfismo para hacer que clases relacionadas respondan de distintas maneras al mismo mensaje, y a utilizar paquetes para agrupar clases relacionadas.

- Capítulo 7, "Técnicas de hilos"

  Un hilo es un único flujo de control secuencial dentro de un programa. Uno de los aspectos potentes del lenguaje Java es que se pueden programar fácilmente múltiples hilos de ejecución para correr simultáneamente dentro del mismo programa. Este capítulo explica cómo crear programas multihilo, y proporciona enlaces a otros recursos con información más profunda.

- Capítulo 8, "Serialización"

  Serialización guarda y recupera el estado de un objeto Java. Este capítulo describe cómo serializar objetos utilizando Java. Describe el interfaz `Serializable`, cómo grabar un objeto en disco y cómo traerlo de nuevo a memoria desde el disco.

- Capítulo 9, "Introducción a la máquina virtual de Java"

  La JVM es el software nativo que permite a un programa Java ejecutarse en una máquina en particular. Este capítulo explica el propósito y la estructura general de la JVM. Analiza los aspectos principales de la JVM, particularmente la seguridad en Java y explica más detalladamente tres características específicas de seguridad: el verificador Java, el Administrador de Seguridad y el cargador de Clases.

- Capítulo 10, “Utilización de la Interfaz nativa de Java (JNI)”

  Este capítulo explica cómo invocar métodos nativos en aplicaciones Java utilizando la Interfaz de Método Nativo Java (JNI). Comienza exponiendo cómo funciona el JNI, a continuación explica la palabra clave `native` y cómo cualquier método Java puede convertirse en un

Capítulo

2

Elementos del lenguaje Java

Este capítulo le proporciona conceptos básicos sobre los elementos del lenguaje de programación Java que se utilizarán a lo largo de este capítulo. Supone que usted comprende los conceptos generales de la programación pero tiene poca o ninguna experiencia con Java.

Términos

En este capítulo se tratan los siguientes términos y conceptos

Identificador

El identificador es el nombre que se elige para llamar a un elemento (tal como una variable o un método). Java aceptará cualquier identificador válido pero, por razones de facilidad de uso, es mejor utilizar un término en lenguaje claro, adaptado para cumplir los siguientes requisitos:

- Debe comenzar por una letra. Estrictamente hablando, puede comenzar con un símbolo de moneda Unicode o con un subrayado (_), pero alguno de estos símbolos puede ser utilizado en archivos importados o en procesos internos. Es mejor evitarlos.

- A continuación, puede contener cualquier carácter alfanumérico (letras o números), subrayados o símbolos de moneda Unicode (tales como libra esterlinao $), pero no otros caracteres especiales.
- Debe ser una sola palabra (sin espacios ni guiones).

El uso de mayúsculas en un identificador depende de la clase de identificador que sea. Java diferencia entre mayúsculas y minúsculas, por lo que hay que tener cuidado con el uso de las mayúsculas. Los estilos correctos de uso de mayúsculas se mencionan en el contexto.

Tipo de datos

Los tipos de datos clasifican la clase de información que pueden contener ciertos elementos de programación Java. Los tipos de datos se distribuyen en dos categorías principales:

- Primitivo o básico
- Compuesto o referencia

Naturalmente, distintas clases de tipos de datos pueden almacenar distintas clases y cantidades de información. Se puede convertir el tipo de datos de una variable a otro tipo de datos diferente dentro de unos límites: no se puede convertir de tipo `booleano` a otro o viceversa, y no se puede convertir un objeto a otro de una clase no relacionada.

Java impedirá que sus datos corran peligro. Esto significa que le permitirá convertir fácilmente una variable o un objeto a un tipo mayor, pero intentará impedirle hacerlo a un tipo más pequeño. Cuando cambie un tipo de dato a otro de menor capacidad, debe utilizar un tipo de sentencia llamada *type cast*.

Tipos de datos primitivos

Los tipos de datos primitivos o básicos, se clasifican en booleanos (especifican un estado activo/inactivo o verdadero/falso), carácter (para caracteres sencillos y Unicode), entero (para números enteros) o de coma flotante (para números decimales). En el código, los tipos de datos primitivos se escriben siempre en minúsculas.

El tipo de datos booleano se llama `boolean` y toma uno de dos posibles valores: `true` (verdadero) o `false` (falso). Java no almacena estos valores numéricamente, sino que utiliza el tipo de datos `boolean` para ello.

El tipo de datos carácter se llama `char` y admite caracteres Unicode con valores de hasta 16 bits de largo. En Java, los caracteres Unicode (letras, caracteres especiales y signos de puntuación) se ponen entre comillas simples: `'b'`. El valor Unicode por defecto de Java es `\u0000`, abarcando desde `\u0000` hasta `\uFFFF`.

En pocas palabras, el sistema de numeración Unicode acepta números entre 0 y 65535, pero los números deben especificarse en notación hexadecimal, precedidos por la secuencia de escape `\u`.

No todos los caracteres especiales se pueden representar de esta forma. Java proporciona su propio conjunto de secuencias de escape, muchas de las cuales se pueden encontrar en la tabla de "Secuencias de escape" en la página 11-19.

En Java, el tamaño de los tipos de datos primitivos es absoluto, en lugar de depender de la plataforma. Esto mejora la portabilidad.

Diferentes tipos de datos numéricos aceptan números de diferentes clases y tamaños. Sus nombres y capacidades se listan a continuación:

Tipo	Atributos	Rango
`double`	El tipo de dato por defecto en Java. Un tipo de coma flotante que admite un número de ocho bytes con quince espacios decimales.	$+/- 9.00x10^{18}$
`int`	La opción más común. Un tipo entero con capacidad para números enteros de 4 bytes.	$+/- 2x10^{9}$
`long`	Un tipo entero que acepta números enteros de 8 bytes.	$+/- 9x10^{18}$
`float`	Un tipo de coma flotante que acoge números de 4 bytes con siete espacios decimales.	$+/- 2.0x10^{9}$
`short`	Un tipo entero con capacidad para números enteros de 2 bytes.	+/– 32768
`byte`	Un tipo entero con capacidad para números enteros de 1 byte.	+/– 128

Tipos de datos complejos

Cada uno de los tipos de datos anteriores acepta un número, un carácter o un estado. Los tipos de datos compuestos, o referencias, se componen de más de un único elemento. Los tipos de datos compuestos son de dos tipos: clases y matrices. Los nombres de clase y de matriz comienzan con una letra mayúscula, y la primera letra de cada palabra que compone su nombre es también una mayúscula, por ejemplo, `NameOfClass`.

Una clase es un trozo de código, coherente y completo, que define un conjunto de objetos unidos lógicamente y su comportamiento. Si desea obtener más información sobre las clases consulte el Capítulo 6, "Programación orientada a objetos en Java".

Cualquier clase puede ser utilizada como un tipo de datos, una vez que haya sido creada e importada en el programa. Debido a que la clase `String` es la más utilizada como un tipo de datos, nos centraremos en ella en este capítulo.

Cadenas

El tipo de datos `String` es realmente la clase `String`. La clase `String` almacena cualquier secuencia de caracteres alfanuméricos, espacios y signos normales de puntuación (denominados *cadenas*), encerrados entre comillas. Las cadenas pueden contener cualquier secuencia de escape Unicode y requieren `\"` para incluir comillas dobles dentro de la cadena, aunque, generalmente, la clase `String`, en sí misma, le indica al programa cómo interpretar los caracteres correctamente.

Matrices

Una matriz es una estructura de datos que contiene un grupo de valores del mismo tipo de datos. Por ejemplo, una matriz puede aceptar un grupo de valores `String`, un grupo de valores `int`, o un grupo de valores `boolean`. Siempre que todos los valores sean del mismo tipo de datos, pueden incluirse en la misma matriz.

Las matrices se caracterizan por un par de corchetes. Cuando declare una matriz en Java, puede poner los corchetes tanto tras el identificador como tras el tipo de datos:

```
int studentID[];
char[] grades;
```

Observe que el tamaño de la matriz no se especifica. La declaración de una matriz no asigna memoria para ella. En la mayoría de los otros lenguajes, debe incluirse el tamaño de la matriz en la declaración, pero en Java no se especifica su tamaño hasta que se utiliza. En ese momento, se asigna la memoria conveniente.

Variable

Una variable es un valor que un programador nombra y define. Las variables necesitan un identificador y un valor.

Literal

Un literal es la representación real de un número, un carácter, un estado o una cadena. Un literal representa el valor de un identificador.

Los literales alfanuméricos incluyen cadenas entre comillas, caracteres `char` individuales entre comillas simples y valores `boolean` verdadero/falso.

Los literales enteros pueden almacenarse como decimales, octales o hexadecimales, pero hay que pensar en la sintaxis utilizada: cualquier entero con un 0 delante (como en una fecha) se interpretará como octal.

Los literales de coma flotante sólo se pueden expresar como decimales. Se tratarán como `double` a menos que se especifique el tipo.

Para obtener una explicación más detallada acerca de los literales y sus posibilidades, consulte *The Java Handbook* de Patrick Naughton.

Aplicación de conceptos

Los siguientes apartados demuestran cómo aplicar los términos y conceptos introducidos anteriormente en este capítulo.

Declaración de variables

El acto de declarar una variable reserva memoria para ella. Declarar una variable sólo requiere dos cosas: un tipo de datos y un identificador, en ese orden. El tipo de dato indica al programa cuánta memoria asignar. El identificador etiqueta la memoria asignada.

Declare la variable sólo una vez. Una vez haya declarado la variable adecuadamente, sólo tiene que referirse a su identificador para acceder a ese bloque de memoria.

Las declaraciones de variables tienen este aspecto:

`boolean isOn;`	Al tipo de datos `boolean` se le pueden asignar los valores `true` o `false` (verdadero o falso). El identificador `isOn` es el nombre que el programador le ha dado a la memoria asignada para esa variable. El nombre `isOn` tiene significado para el lector humano como algo que aceptará lógicamente valores verdadero/falso.
`int studentsEnrolled;`	El tipo de datos `int` le indica que va a tratar con un número entero de menos de 10 dígitos. El identificador `studentsEnrolled` sugiere lo que el número va a indicar (estudiantes matriculados). Puesto que los estudiantes son personas enteras, el tipo de datos apropiado requiere números enteros.
`float creditCardSales;`	El tipo de datos `float` es adecuado, dado que el dinero se representa habitualmente en decimales. Se sabe que se trata de dinero porque el programador ha nombrado adecuadamente esta variable como `creditCardSales` (ventas por tarjeta de crédito).

Métodos

En Java, los *Métodos* son equivalentes a las funciones o subrutinas en otros lenguajes. El método define una acción a realizar sobre un objeto.

Los métodos constan de un nombre y un par de paréntesis:

```
getData()
```

Aquí, `getData` es el nombre y los paréntesis le indican al programa que es un método.

Si el método necesita alguna información en particular para realizar su trabajo, lo que necesite va dentro de los paréntesis. A lo que va dentro de los paréntesis se le llama el *argumento*, o *arg* para abreviar. En una declaración de método, el argumento debe incluir un tipo de datos y un identificador:

```
drawString(String remark)
```

Aquí, `drawString` es el nombre del método y `String remark` es el tipo de datos y el nombre de variable para la cadena que el método debe obtener.

Debe indicarle al programa qué tipo de dato debe devolver el método o, si no, no devolverá nada. A eso se le llama el *tipo devuelto*. Se puede hacer que un método devuelva datos de cualquier tipo primitivo. Si el método no tiene que devolver nada (como en la mayoría de los métodos que realizan acciones), el tipo devuelto debe ser `void`.

El tipo devuelto, el nombre y los paréntesis con los argumentos necesarios, proporcionan una declaración de método muy básica:

```
String drawString(String remark);
```

Su método será probablemente más complejo. Una vez que lo haya escrito, nombrado y haya indicado los argumentos necesarios (si hacen falta), tiene que definirlo completamente. Esto se hace bajo el nombre del método, alojando el cuerpo de la definición entre un par de llaves. Esto produce una declaración de método más compleja:

```
String drawString(String remark) {              //Declara el método
  String remark = "¡Qué dientes más grandes tienes!" //Define el contenido del
método.
}                                               //Cierra el cuerpo del método.
```

Una vez que haya definido el método, sólo tiene que referirse a él por su nombre y pasarle los argumentos que necesite para hacer correctamente su trabajo: `drawString(remark);`

Capítulo

3

Estructura del lenguaje Java

Este apartado le proporciona conceptos fundamentales sobre la estructura del lenguaje de programación Java que se utilizarán a lo largo de este capítulo. Supone que usted comprende los conceptos generales de la programación pero tiene poca o ninguna experiencia con Java.

Términos

En este capítulo se tratan los siguientes términos y conceptos:

Palabras clave

Las palabras clave son términos reservados de Java que modifican otros elementos de la sintaxis. Las palabras clave pueden definir la accesibilidad de un objeto, el flujo de un método o el tipo de datos de una variable. Las palabras clave no pueden emplearse como identificadores.

Muchas de las palabras clave de Java se han tomado prestadas de C/C++. También, como en C/C++, se escriben siempre en minúscula. Hablando en términos generales, las palabras clave de Java se pueden clasificar de acuerdo a su función (entre paréntesis, ejemplos):

- Tipos de datos, tipos devueltos y términos (`int, void, return`)
- Paquete, clase, miembro e interfaz (`package, class, static`)
- Modificadores de acceso (`public, private, protected`)
- Bucles y control de bucles (`if, switch, break`)
- Tratamiento de excepciones (`throw, try, finally`)
- Palabras reservadas no utilizadas aún, pero no disponibles (`goto, assert, const`)

Algunas palabras clave se explican en el contexto en estos capítulos. Si desea consultar una lista completa de palabras clave y su significado, consulte las tablas "Palabras clave de Java" en la página 11-3.

Operadores

Los operadores le permiten acceder, manipular, relacionar o referirse a elementos del lenguaje Java, desde variables a clases. Los operadores tienen propiedades de *precedencia* y *asociatividad*. Cuando varios operadores actúan sobre el mismo elemento (u *operando*), la precedencia de cada uno de ellos determina qué operador actúa primero. Cuando más de un operador tiene la misma precedencia, se aplican las reglas de asociatividad. Estas reglas son, generalmente, matemáticas; por ejemplo, los operadores se usarán normalmente de izquierda a derecha y las expresiones de operadores entre paréntesis se evaluarán antes que las expresiones de operadores, fuera de paréntesis.

Los operadores generalmente se dividen en seis categorías: asignación, aritméticos, lógicos, comparación, bit a bit, y ternarios.

Asignación significa almacenar el valor a la *derecha* del signo `=` en la variable a la *izquierda* de él. Se puede asignar un valor a una variable tanto *cuando* se la declara, como *después* de haberla declarado. Al ordenador le da igual; el programador decide lo más adecuado en su programa y de acuerdo a sus costumbres:

```
double bankBalance;           //Declaración
bankBalance = 100.35;         //Asignación

double bankBalance = 100.35;   //Declaración con asignación
```

En ambos casos, el valor `100.35` se almacena en la memoria reservada por la declaración de la variable `bankBalance`.

Los operadores de asignación permiten asignar valores a las variables. También permiten realizar una operación en una expresión y asignar el nuevo valor al operando de la derecha, utilizando una única expresión combinada.

Los operadores aritméticos realizan cálculos matemáticos sobre valores enteros y de coma flotante. Se emplean los signos matemáticos habituales: + suma, - resta, * multiplica y / divide dos números.

Los operadores lógicos, o Booleanos, permiten al programador agrupar expresiones `boolean` de forma muy útil, diciéndole al programa exactamente cómo determinar una condición específica.

Los operadores de comparación evalúan expresiones individuales comparándolas con otras partes del código. Las comparaciones más complejas (como comparaciones de cadenas) se hacen por programación.

Los operadores bit a bit actúan sobre los 0 y 1 individuales de los dígitos binarios. Los operadores bit a bit de Java pueden preservar el signo del número original. No todos los lenguajes hacen esto.

El operador ternario, `?:`, proporciona una manera resumida de escribir una sentencia `if-then-else` muy simple. Se evalúa la primera expresión. Si es cierta, se evalúa la segunda. Si la segunda expresión es falsa, se utiliza la tercera.

A continuación, una lista parcial de otros operadores y sus atributos:

Operador	Operando	Comportamiento
.	objeto miembro	Accede a un miembro del objeto.
(<type>)	tipo de datos	Convierte un tipo de datos a otro.[1]
+	Cadena	Une cadenas (concatenador).
	número	Suma.
-	número	El menos unario [2] (invierte el signo del número).
	número	Resta.
!	boolean	El operador `NOT booleano`.
&	entero, booleano	El operador `AND`, tanto `booleano` como bit a bit (entero). Cuando aparece doble (`&&`), es el `AND` condicional `booleano`.
=	la mayoría de los elementos con variables	Asigna un elemento a otro (por ejemplo, un valor a una variable, o una clase a una instancia). Puede combinarse con otros operadores para realizar otra operación y asignar el valor resultante. Por ejemplo, `+=` añade el valor de la izquierda a la derecha y después asigna el nuevo valor al lado derecho de la expresión.

1. Es importante distinguir entre operación y puntuación. Los paréntesis se usan encerrando argumentos como puntuación. Se utilizan alrededor de un tipo de datos en una operación que cambia el tipo de datos de una variable por el que está entre los paréntesis.
2. Un operador unario actúa sobre un solo operando, un operador binario sobre dos y un ternario sobre tres operandos.

Comentarios

Comentar el código es una excelente práctica de programación. Unos buenos comentarios pueden ayudarle a analizar su código más rápidamente, seguir la pista de lo que ha hecho mientras construía un programa complejo y recordarle cosas que quiere añadir o afinar. Puede utilizar comentarios para ocultar partes del código que quiere guardar para situaciones especiales, o para apartarlas mientras trabaja en algo con lo que puedan entrar en conflicto. Los comentarios le pueden ayudar a recordar qué estaba haciendo cuando vuelva sobre un proyecto después de trabajar en otro, o cuando vuelva de vacaciones. En un entorno de desarrollo en equipo, o en cualquier situación en la que el código se va pasando entre los programadores, los comentarios pueden ayudar a los otros a comprender el propósito y las asociaciones de todo lo que comente, sin necesidad de analizar cada línea para asegurarse de que lo comprenden.

Java utiliza tres clases de comentarios: comentarios monolínea, comentarios multilínea y comentarios Javadoc.

Comentario	Etiqueta	Propósito
Monolínea	`// ...`	Adecuado para comentarios breves sobre la función o la estructura de una sentencia o expresión. Sólo requieren una etiqueta de apertura: tan pronto como comience una nueva línea, estará de nuevo en el código.
Multilínea	`/* ... */`	Bueno para cualquier comentario que vaya a cubrir más de una línea, como cuando desea entrar en detalle sobre lo que sucede en el código o cuando se necesita incluir avisos legales en el código. Requiere etiquetas de apertura y de cierre.
Javadoc	`/** ... */`	Este es un comentario multilínea que la utilidad JDK de Javadoc puede leer y convertir en documentación HTML. Javadoc tiene etiquetas que puede utilizar para aumentar su utilidad. Se utilizan para proporcionar ayuda sobre las API, generar listas de tareas pendientes e integrar flags en el código. Requiere etiquetas de apertura y de cierre. Si desea aprender más sobre la herramienta Javadoc, vaya a la página Javadoc de Sun, enhttp://java.sun.com/ j2se/1.3/docs/tooldocs/ javadoc/.

He aquí algunos ejemplos:

```
/* Se pueden escribir tantas líneas discutiendo
   un asunto o tantas páginas de
   informe como se desee entre
   estas dos etiquetas.
*/

/* Tengase en cuenta que, si resulta necesario algún asunto de forma prolija,
   se pueden anidar comentarios de una sola línea
```

```
   //dentro de comentarios multilínea
   y el compilador no experimentará ningún tipo de problema
   al manejarlos.
*/

/* Lo que no se puede hacer es
   /* anidar comentarios multilínea
   */
   /** de ningún tipo
   */
   porque se generará un
   error de compilación.
*/

/**Se puede encontrar información muy útil acerca de cómo funciona
   el código en las etiquetas Javadoc. Se pueden utilizar
   las etiquetas especiales, tales como @todo para aprovechar las
   características de ayuda de Javadoc.
*/
```

Sentencias

Una sentencia es un comando único. Un comando puede abarcar varias líneas de código, pero el compilador lo interpreta todo como un sólo comando. Las sentencias individuales (normalmente una sola línea) finalizan con un punto y coma (`;`) y los grupos de sentencias (multilínea) finalizan con un cierre de llave (`}`). Las sentencias multi-línea se suelen llamar *bloques de código*.

Por defecto, Java ejecuta las sentencias en el orden en que están escritas, pero permite adelantar referencias a códigos que todavía no se han definido.

Bloques de código

Un bloque de código es todo lo que hay entre las llaves, e incluye la expresión que introduce la parte entre llaves:

```
class GettingRounder {
  ...
}
```

Comprensión del ámbito

Las reglas de ámbito determinan en qué parte de un programa es reconocida una variable. Las variables se dividen en dos categorías de ámbito principales:

- Variables globales: Variables que son reconocidas dentro de toda una clase.

- Variables locales: Variables que son reconocidas sólo dentro del bloque de código en el que se declaran.

Las reglas de ámbito están estrechamente relacionadas con los bloques de código. La única regla general del ámbito es: una variable declarada en un bloque de código es visible sólo en ese bloque y en los que haya anidados dentro de él. El siguiente código lo ilustra:

```
class Scoping {
  int x = 0;
  void method1() {
    int y;
    y = x;  // Esto funciona. method1 tiene acceso a y.
  }
  void method2() {
    int z = 1;
    z = y;  // Esto no funciona:
            // y es definida fuera del ámbito del método.
  }
}
```

Este código declara una clase llamada `scoping`, que tiene dos métodos: `method1()` y `method2()`. Se considera a la propia clase como el bloque de código principal y los dos métodos son sus bloques anidados.

La variable `x` se declara en el bloque principal, luego es *visible* (reconocida por el compilador) tanto en `method1()` como en `method2()`. Las variables `y` y `z`, por su parte, fueron declaradas en dos bloques independientes anidados, por lo tanto, intentar utilizar `y` en `method2()` es ilegal, puesto que `y` no es visible en ese bloque.

Nota Un programa que se apoya en variables globales puede ser proclive a errores por dos razones:

1. Es difícil hacer un seguimiento de las variables globales.
2. Un cambio en una variable global en una parte del programa puede tener un inesperado efecto secundario en otra parte del mismo.

Las variables locales son más seguras, puesto que tienen un lapso de vida más corto. Por ejemplo, una variable declarada dentro de un método sólo puede ser accedida desde ese método, por lo que no hay riesgo de que sea indebidamente utilizada en alguna otra parte del programa.

Finalice cada sentencia individual con un punto y coma. Asegúrese de que cada llave tiene una pareja. Organice sus llaves de manera coherente (como en los ejemplos anteriores) de forma que pueda seguir con facilidad los distintos pares. Muchos IDE Java (como JBuilder) anidan automáticamente las llaves de acuerdo a los parámetros que se empleen.

Aplicación de conceptos

Los siguientes apartados demuestran cómo aplicar los términos y conceptos introducidos anteriormente en este capítulo.

Utilización de operadores

Revisión

Hay seis tipos básicos de operadores (aritméticos, lógicos, de asignación, de comparación, bit a bit y ternarios) y además, pueden afectar a uno, dos o tres operandos, lo que los divide, a su vez, en unarios, binarios o ternarios. Tienen propiedades de precedencia y asociatividad que determinan el orden en que son procesados.

A los operadores se les asignan números para establecer su precedencia. Cuanto mayor es el número, mayor es el orden de precedencia (es decir, tiene más posibilidades de ser evaluado antes que otros). Un operador de precedencia 1 (la más baja) será evaluado el último y un operador con una precedencia de 15 (la más alta) será evaluado el primero.

Los operadores con la misma precedencia se evalúan normalmente de izquierda a derecha.

La precedencia se evalúa antes que la asociatividad. Por ejemplo, la expresión `a + b - c * d` no se evaluará de izquierda a derecha; la multiplicación tiene precedencia sobre la suma, de manera que `c * d` será lo primero que se evalúe. La suma y la resta tienen el mismo orden de precedencia, por lo que se aplica la asociatividad: `a` y `b` se sumarán primero y a esa suma se le restará el producto de `c * d`.

Es una buena práctica utilizar paréntesis encerrando las expresiones matemáticas que desea que se evalúen primero, independientemente de su precedencia; por ejemplo: `(a + b) - (c * d)`. El programa evaluará esta operación de la misma forma pero, para el lector humano, este formato es más claro.

Operadores aritméticos

Java proporciona un conjunto completo de operadores para cálculos matemáticos. Java, al contrario que otros lenguajes, puede realizar funciones matemáticas tanto sobre valores enteros como de coma flotante. Probablemente, estos operadores le serán familiares.

Estos son los operadores aritméticos:

Operador	Definición	Prec.	Asoc.
`++/--`	Auto-incremento/decremento: Añade uno a o resta uno de su único operando. Si el valor de `i` es 4, `++i` es 5. Un `pre-incremento` (`++i`) incrementa en uno el valor y asigna el nuevo valor a la variable original `i`. Un `post-incremento` (`i++`) incrementa el valor pero deja la variable original `i` con su valor original. Consulte más abajo para obtener más información.	1	Derecha
`+/-`	Más/menos unario: asigna o cambia el valor positivo/negativo de un número.	2	Derecha
`*`	Multiplicación.	4	Izquierda
`/`	División.	4	Izquierda
`%`	Modulus: Divide el primer operando por el segundo y devuelve el resto. Más adelante encontrará un pequeño repaso matemático.	4	Izquierda
`+/-`	Suma/resta	5	Izquierda

Utilice el pre- o el post-incremento/decremento, dependiendo de cuándo quiera que el nuevo valor sea asignado:

```
int y = 3, x; //1. Declaraciones de variables
int b = 9;    //2.
int a;        //3.
x = ++y;      //4. preincremento
a = b--;      //5. postdecremento
```

En la sentencia 4, `preincremento`, el valor de la variable `y` se incrementa en 1 y, a continuación, su nuevo valor (4) se asigna a `x`. Tanto `x` como `y` tenían originalmente un valor de 3; ahora las dos tienen un valor de 4.

En la sentencia 5, `postdecremento`, el valor actual de `b` (9) se asigna a `a` y *a continuación* el valor de `b` se decrementa (a 8). `b` originalmente tenía un valor de 9 y `a` no tenía valor asignado; ahora, `a` es 9 y `b` es 8.

El operador `modulus` requiere una explicación para aquellos que estudiaron matemáticas por última vez hace mucho tiempo. Recuerde que cuando se dividen dos números, raramente la división es exacta. Lo que queda después de dividir los números (sin añadir nuevas posiciones decimales) es el *resto*. Por ejemplo, 3 cabe una vez en 5, y sobran 2. El resto (en este caso, 2) es lo que evalúa el operador `modulus`. Puesto que los restos se repiten en un ciclo de divisiones de forma predecible (por ejemplo, una hora es modulus 60), el operador `modulus` es particularmente útil cuando quiera indicarle a un programa que repita un proceso a intervalos específicos.

Operadores lógicos

Los operadores lógicos (o Booleanos) permiten al programador agrupar expresiones `boolean` para definir ciertas condiciones. Estos operadores realizan las operaciones Booleanas estándar `AND`, `OR`, `NOT`, y `XOR`.

La siguiente tabla relaciona los operadores lógicos:

Operador	Definición	Prec.	Asoc.
`!`	`NOT` booleano (unario) Cambia `true` a `false` o `false` a `true`. Debido a su baja precedencia, puede necesitar utilizar paréntesis encerrando esta sentencia.	2	Derecha
`&`	Evaluación `AND` (binaria) Da como resultado `true` solo si ambos operandos son `true`. Siempre evalúa ambos operandos. Raramente usado como un operador lógico.	9	Izquierda
`^`	Evaluación `XOR` (binaria) Da como resultado `true` sólo si ambos operandos son `true`. Evalúa ambos operandos.	10	Izquierda
`|`	Evaluación `OR` (binaria) Devuelve `true` si uno o ambos operandos son `true`. Evalúa ambos operandos.	11	Izquierda
`&&`	`AND` condicional (binario) Da como resultado `true` sólo si ambos operandos son `true`. Llamado "condicional" porque sólo evalúa el segundo operando si el primero es `true`.	12	Izquierda
`||`	`OR` condicional (binario) Devuelve `true` si uno o ambos operandos son `true`; devuelve `false` si ambos son `false`. No evalúa el segundo operador si el primero es `true`.	13	Izquierda

Los operadores de evaluación siempre evalúan ambos operandos. Los operadores condicionales, en cambio, siempre evalúan el primer operando y, si éste determina el valor de toda la expresión, no evalúan el segundo. Por ejemplo:

```
if ( !isHighPressure && (temperatura1 > temperatura2)) {
  ...
}                                       //Sentencia 1: condicionales

boolean1 = (x < y) || ( a > b);         //Sentencia 2: condicional

boolean2 = (10 > 5) & (5 > 1);          //Sentencia 3: evaluación
```

La primera sentencia evalúa primero `!isHighPressure`. Si `!isHighPressure` es `false` (es decir, si la presión *es* alta; obsérvese la doble negativa lógica de `!` y `false`), el segundo operando, `temperatura1 > temperatura2`, no necesita ser evaluado. `&&` sólo necesita un valor `false` para saber qué valor debe devolver.

En la segunda sentencia, el valor de `boolean1` será `true` si `x` es menor que `y`. Si `x` es mayor que `y`, la segunda expresión será evaluada; si `a` es menor que `b`, el valor de `boolean1` seguirá siendo `true`.

En la tercera sentencia, sin embargo, el compilador calculará los valores de los dos operandos antes de asignar `true` o `false` a `boolean2`, puesto que `&` es un operador de evaluación, no un operador condicional.

Operadores de asignación

Como es sabido, el operador básico de asignación (`=`) permite asignar un valor a una variable. Con el conjunto de operadores de asignación de Java se puede realizar una operación sobre cada operando y asignar el nuevo valor a una variable en un sólo paso.

La siguiente tabla enumera los operadores de asignación:

Operador	Definición	Prec.	Asoc.
`=`	Asigna el valor de la derecha a la variable de la izquierda.	15	Derecha
`+=`	Añade el valor de la derecha al valor de la variable de la izquierda; asigna el nuevo valor a la variable original.	15	Derecha
`-=`	Resta el valor de la derecha del valor de la izquierda; asigna el nuevo valor a la variable original.	15	Derecha
`*=`	Multiplica el valor de la derecha con el valor de la variable de la izquierda; asigna el nuevo valor a la variable original.	15	Derecha
`/=`	Divide el valor de la derecha por el valor de la variable de la izquierda; asigna el nuevo valor a la variable original.	15	Derecha

El primer operador ya nos es familiar. El resto de los operadores de asignación realizan primero una operación y, después, almacenan el resultado de la misma en el operando del lado izquierdo de la expresión. He aquí algunos ejemplos:

```
int y = 2;
y *= 2;  //es lo mismo que (y = y * 2)

boolean b1 = true, b2 = false;
b1 &= b2;  //es lo mismo que (b1 = b1 & b2)
```

Operadores de comparación

Los operadores de comparación le permiten comparar un valor con otro.

La siguiente tabla relaciona los operadores de comparación:

Operador	Definición	Prec.	Asoc.
<	Menor que	7	Izquierda
>	Mayor que	7	Izquierda
<=	Menor o igual que	7	Izquierda
>=	Mayor o igual que	7	Izquierda
==	Igual a	8	Izquierda
!=	No igual a	8	Izquierda

El operador de igualdad se puede utilizar para comparar dos variables objeto del mismo tipo. En este caso, el resultado de la comparación es verdadero sólo si las dos variables se refieren al mismo objeto. He aquí una demostración:

```
m1 = new Mammal();
m2 = new Mammal();
boolean b1 = m1 == m2;  //b1 es falso

m1 = m2;
boolean b2 = m1 == m2;  //b2 es verdadero
```

El resultado del primer test de igualdad es falso, porque `m1` y `m2` se refieren a distintos objetos (aunque sean del mismo tipo). La segunda comparación es verdadera, porque ahora las dos variables representan el mismo objeto.

Nota La mayor parte de las veces, no obstante, se utiliza el método `equals()` de la clase `Object` en lugar del operador de comparación. La clase comparada debe ser una subclase de `Object` antes de que sus objetos puedan ser comparados utilizando `equals()`.

Operadores bit a bit

Los operadores bit a bit son de dos tipos: operadores de desplazamiento y operadores Booleanos. Los operadores de desplazamiento se utilizan para desplazar los dígitos binarios de un entero a la derecha o a la izquierda. Considere el siguiente ejemplo (se utiliza el tipo entero `short` en lugar de `int` para abreviar):

```
short i = 13; //i es 0000000000001101
i = i << 2;   //i es 0000000000110100
```

En la segunda línea, el operador de desplazamiento a la izquierda bit a bit desplazó todos los bits de `i` dos posiciones a la izquierda.

Nota La operación de desplazamiento en Java difiere de la que se realiza en C/C++ en cómo se utiliza con enteros *con signo*. Un entero con signo es uno cuyo primer bit de la izquierda se utiliza para indicar el signo positivo o negativo del entero: el bit es `1` si el entero es negativo, `0` si es positivo. En Java, los enteros siempre tienen signo, mientras que en C/C++ tienen signo sólo por defecto. En la mayoría de las implementaciones de C/C++,

una operación de desplazamiento bit a bit no conserva el signo del número entero; ya que el bit de signo será desplazado. En Java, sin embargo, los operadores de desplazamiento preservan el bit de signo (a menos que se utilice >>> para realizar un *desplazamiento sin signo*). Esto significa que el bit de signo se duplica y, a continuación, se desplaza. Por ejemplo, el desplazar 10010011 a la derecha 1 bit es 11001001.

La siguiente tabla relaciona los operadores bit a bit de Java:

Operador	Definición	Prec.	Asoc.
~	NOT bit a bit Invierte cada bit del operando, de forma que cada 0 se convierte en 1 y viceversa.	2	Derecha
<<	Desplazamiento a la izquierda con signo Desplaza los bits del operando de la izquierda hacia la izquierda el número de dígitos especificado en el operando de la derecha, añadiendo ceros por la derecha. Los bits de orden superior se pierden.	6	Izquierda
>>	Desplazamiento a la derecha con signo Desplaza hacia la derecha los bits del operando de la izquierda, el número de dígitos especificado a la derecha. Si el operando de la izquierda es negativo, se completa con ceros a la izquierda; si es positivo, con unos. De esta forma se preserva el signo original.	6	Izquierda
>>>	Desplazamiento a la derecha rellenando con ceros Desplaza a la derecha, pero siempre rellena con ceros.	6	Izquierda
&	AND bit a bit Pueden utilizarse con = para asignar el valor.	9	Izquierda
|	OR bit a bit Pueden utilizarse con = para asignar el valor.	10	Izquierda
^	XOR bit a bit Pueden utilizarse con = para asignar el valor.	11	Izquierda
<<=	Desplazamiento a la izquierda con asignación.	15	Izquierda
>>=	Desplazamiento a la derecha con asignación.	15	Izquierda
>>>=	Desplazamiento a la derecha con asignación y relleno a ceros.	15	Izquierda

? : — El operador ternario

?: es un operador ternario que Java tomó prestado de C. Proporciona un práctico atajo para crear un tipo de sentencia if-then-else muy sencillo.

La sintaxis es como sigue:

```
<expresión 1> ? <expresión 2> : <expresión 3>;
```

`expresión 1` se evalúa primero. Si es verdadera, se evalúa `expresión 2`. Si `expresión 2` es falsa, se utiliza `expresión 3`. Por ejemplo:

```
int x = 3, y = 4, max;
max = (x > y) ? x : y;
```

Aquí, a `max` se le asigna el valor de `x` o de `y`, dependiendo de cuál sea mayor.

Utilización de métodos

Ya sabe que los métodos son los que consiguen que se hagan las cosas. Los métodos no pueden contener otros métodos, pero pueden contener variables y referencias a clases.

He aquí un breve ejemplo para ilustrarlo. Este método ayuda a una tienda de música con su inventario:

```
//Declara el método: return type, name, args:
public int getTotalCDs(int numRockCDs, int numJazzCDs, in numPopCDs) {
//Declare the variable totalCDs. The other three variables are
  //declared elsewhere:
int totalCDs = numRockCDs + numJazzCDs + numPopCDs;
  //Le hace llevar a cabo algo útil. En este caso muestra esta línea en pantalla:
  System.out.println("Total CDs in stock = " + totalCDs);
}
```

En Java, se puede definir más de un método con el mismo nombre, a condición de que los distintos métodos utilicen argumentos diferentes. Por ejemplo, tanto `public int getTotalCDs(int numRockCDs, int numJazzCDs, int numPopCDs)` como `public int getTotalCDs(int salesRetailCD, int salesWholesaleCD)` están permitidas en la misma clase. Java reconocerá los distintos patrones de los argumentos (las *firmas* de los métodos) y aplicará el método correcto cuando se le llame. Asignar el mismo nombre a distintos métodos es un proceso conocido como *sobrecarga de métodos*.

Para acceder a un método desde otras partes de un programa, debe crearse primero una instancia de la clase en la que el método reside y, entonces, utilizar ese objeto para llamar al método:

```
//Crea una instancia total CD de la clase Inventory:
Inventory totalCD = new Inventory();

//Accede al método getTotalCDs() incluido dentro de Inventory, almacenando el valor
total:
int total = totalCD.getTotalCDs(myNumRockCDs, myNumJazzCDs, myNumPopCDs);
```

Utilización de matrices

Téngase en cuenta que el tamaño de una matriz no forma parte de su declaración. La memoria requerida por una matriz no es realmente asignada hasta que la matriz se inicializa.

Para inicializar la matriz (y asignar la memoria necesaria), hay que utilizar el operador `new` de la siguiente manera:

```
int studentID[] = new int[20];              // Crea una matriz de 20 elementos
de tipo int.
char[] grades = new char[20];               // Crea una matriz de 20 elementos
de tipo char.
float[][] coordinates = new float[10][5];  // Una matriz bidimensional de 10
por 5 elementos
                                            // de tipo float.
```

Nota Al crear matrices bidimensionales, el primer número de la matriz define el número de columnas y, el segundo, el de filas.

Java cuenta posiciones empezando por 0. Esto implica que, en una matriz de 20 elementos, éstos se numerarán de 0 a 19: El primer elemento será el 0, el segundo el 1 y así sucesivamente. Hay que tener cuidado acerca de cómo se cuenta cuando se trabaja con matrices.

Cuando se crea una matriz, el valor de todos sus elementos es `null` o cero; los valores se asignan más tarde.

Nota El uso del operador `new` en Java es similar al del comando `malloc` en C y el operador `new` en C++.

Para inicializar una matriz, especifique los valores de sus elementos dentro de un par de llaves. Para matrices multi-dimensionales, utilice llaves anidadas. Por ejemplo:

```
char[] grades = {'A', 'B', 'C', 'D', 'F');
float[][] coordinates = {{0.0, 0.1}, {0.2, 0.3}};
```

La primera sentencia crea una matriz de tipo `char` llamada `grades`. Inicializa los elementos de la matriz con los valores `'A'` a `'F'`. Observe que no se ha utilizado el operador `new` para crear esta matriz; al inicializarla, se ha asignado automáticamente memoria suficiente para albergar todos los valores inicializados. Por lo tanto, la primera sentencia crea una matriz tipo `char` de 5 elementos.

La segunda sentencia crea una matriz bidimensional de tipo `float` llamada `coordinates`, cuyo tamaño es 2 x 2. Básicamente, `coordinates` es una matriz compuesta por dos elementos de matriz: la primera fila se inicializa a `0.0` y `0.1` y la segunda fila a `0.2` y `0.3`.

Utilización de constructores

Una *clase* es un trozo completo de código encerrado entre un par de llaves que define un conjunto coherente, lógicamente hablando, de variables, atributos y acciones. Un *paquete* es un conjunto de clases asociadas lógicamente.

Observe que una clase es simplemente un conjunto de instrucciones. No hace nada por sí misma. Es igual que una receta: se puede hacer un pastel con la receta adecuada, pero la receta no es el pastel, es sólo las instrucciones para hacerlo. El pastel es un *objeto* que se crea a partir de las instrucciones de la receta. En Java, se dice que se ha creado una *instancia* de pastel a partir de la receta Pastel.

El acto de crear una instancia de una clase se llama *instanciar* ese objeto. Se *instancia* un *objeto* de una *clase*.

Para instanciar un objeto, utilice el operador de asignación (=), la palabra clave `new` y una clase especial de método llamado *constructor*. Una llamada a un constructor es el nombre de la clase que se está instanciando, seguido por un par de paréntesis. Aunque parezca un método, toma el nombre de una clase; es por ello por lo que está en mayúscula:

```
<NombredeClase> <nombredeInstancia> = new <Constructor()>;
```

Por ejemplo, para instanciar un nuevo objeto de la clase `Geek` y llamar a la instancia `thisProgrammer`:

```
Geek thisProgrammer = new Geek();
```

Un constructor configura una nueva instancia de una clase: inicializa todas las variables de la clase, haciéndolas disponibles inmediatamente. También lleva a cabo cualquier rutina de puesta en marcha que requiera el objeto.

Por ejemplo, cuando se necesita conducir un coche, lo primero que se hace es abrir la puerta, meterse dentro, abrocharse el cinturón y arrancar el motor. (Después de esto, se pueden hacer todas las cosas implicadas en la conducción, como meter marchas y utilizar el acelerador.) El constructor maneja los equivalentes programáticos de las acciones y objetos implicados en entrar y arrancar el coche.

Una vez que se ha creado una instancia, se puede utilizar su nombre para acceder a los miembros de esa clase.

Si desea obtener más información sobre los constructores, consulte "Programación orientada a objetos en Java" en la página 6-1.

Acceso a miembros

El operador de acceso (.) se utiliza para acceder a los miembros de un objeto instanciado. La sintaxis básica es la siguiente:

```
<nombredeInstancia>.<nombredeMiembro>
```

La sintaxis precisa del nombre del miembro depende del tipo de miembro. Así, puede incluir variables (`<nombredeMiembro>`), métodos (`<nombredeMiembro>()`), o subclases (`<NombredeMiembro>`).

Se puede utilizar esta operación dentro de otros elementos de sintaxis, dondequiera que necesite acceder a un miembro. Por ejemplo:

```
setColor(Color.pink);
```

Este método necesita un color para realizar su trabajo. El programador utilizó una operación de acceso como argumento para acceder a la variable `rosa` dentro de la clase `Color`.

Matrices

Los elementos de las matrices se acceden por *indexación* de la variable matriz. Para indexar una variable matriz, se añade al nombre de la variable el número (índice) del elemento encerrado entre corchetes. *Las matrices se indexan siempre empezando por 0.* (Es un error común codificar como si empezaran por 1.)

En el caso de matrices multi-dimensionales, para acceder a un elemento se debe utilizar un índice para cada dimensión. El primer índice es la fila y el segundo la columna.

Por ejemplo:

```
firstElement = grades[0];   //firstElement = 'A'
fifthElement = grades[4];   //fifthElement = 'F'
row2Col1 = coordinates[1][0]; //row2Col1 = 0.2
```

El siguiente fragmento de código muestra un uso de las matrices. Crea una matriz de 5 elementos tipo `int` llamada `intArray` y, a continuación, utiliza un bucle `for` para almacenar los enteros del 0 al 4 en los elementos de la matriz:

```
int[] intArray = new int [5];
int index;
for (index = 0; index < 5; index++) intArray [index] = index;
```

Este código incrementa la variable `index` desde `0` hasta `4` y, en cada paso, almacena su valor en el elemento de `intArray` indexado por la variable `index`.

Capítulo

4

Control en el lenguaje Java

Esta sección proporciona conceptos básicos sobre control en el lenguaje de programación Java. Estos conceptos se utilizarán a lo largo de todo el capítulo. Supone que usted comprende los conceptos generales de la programación pero tiene poca o ninguna experiencia con Java.

Términos

En este capítulo se tratan los siguientes términos y conceptos:

Gestión de cadenas

La clase `String` proporciona métodos que permiten obtener subcadenas o *indexar* caracteres de una cadena. Sin embargo, el valor de un objeto `String` declarado no se puede cambiar. Si necesita cambiar el valor del objeto `String` asociado con esa variable, debe hacer que la variable apunte a un nuevo valor:

```
String text1 = new String("Good evening.");  // Declara text1 y le asigna un valor.
text1 = "¡Hola, cariño, ya estoy en casa!"   // Asigna un nuevo valor a text1.
```

La indexación permite apuntar a un determinado carácter de una cadena. Java cuenta cada posición de una cadena empezando desde 0, de modo que la primera posición es 0, la segunda es 1 y así sucesivamente. Esto otorga un índice 7 a la octava posición de una cadena.

La clase `StringBuffer` proporciona una alternativa a la indexación. Asimismo, ofrece varios métodos para manipular el contenido de una cadena. La clase `StringBuffer` almacena las cadenas en un búfer (un área especial de memoria) cuyo tamaño se puede controlar de forma explícita. De esta forma, se puede cambiar la cadena tantas veces como sea necesario antes de declarar un objeto `String` y hacerla permanente.

En general, la clase `String` está indicada para almacenamiento de cadenas, y la clase `StringBuffer` para su manipulación.

Conversiones implícitas y explícitas de tipos

Los valores de los tipos de datos se pueden convertir de un tipo a otro. Los valores de las clases se pueden convertir entre clases de la misma jerarquía. Observe que la conversión no cambia el tipo original del valor, sólo cambia la percepción que de él tiene el compilador para esa operación particular.

No obstante, existen ciertas restricciones lógicas obvias. Una conversión de ampliación (de un tipo más pequeño a otro más grande) es sencilla; sin embargo, una conversión de reducción (convertir un tipo más grande, como `double` o `Mamífero` a otro más pequeño, como `float` u `Oso`) pone en peligro los datos, a menos que éstos puedan ajustarse sin pérdidas al tamaño del nuevo tipo. Una conversión de reducción requiere una operación especial denominada *conversión explícita de tipos (cast)*.

La siguiente tabla muestra las conversiones de ampliación para valores de tipos primitivos. Estas conversiones no afectan a la integridad de los datos:

Tipo original	Se convierte en un tipo
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Para convertir explícitamente un tipo de datos, ponga el tipo de *destino* entre paréntesis inmediatamente antes de la variable cuyo tipo desea modificar: `(int)x`. Así es como queda en contexto, donde `x` es la variable cuyo tipo se está convirtiendo, `float` es el tipo de datos original, `int` es el tipo de datos de destino e `y` es la variable que almacena el nuevo valor:

```
float x = 1.00;    //asigna a x el valor float
int y = (int)x;     //convierta x en una variable de tipo int llamada y
```

Se asume que el valor de x cabe en un tipo int. Los valores decimales de x se pierden en la conversión. Java redondea a la baja hasta el número entero más cercano.

Tipos devueltos y sentencias return

Una declaración de método requiere un tipo devuelto, del mismo modo que una declaración de variable requiere un tipo de datos. Los tipos devueltos son los mismos que tipo de datos (tales como int, boolean, String, etc.), exceptuándose el tipo void.

void es un tipo devuelto especial. Este tipo indica que el método no necesita devolver nada cuando acaba. Se utiliza habitualmente en métodos que sólo realizan una acción y no devuelven información.

Todos los demás tipos devueltos requieren una sentencia return al final del método. La sentencia return también se puede utilizar en un método void para abandonar el método en un determinado punto, pero en cualquier otro caso no es necesaria.

Una sentencia return consta de la palabra clave return y la cadena, dato, nombre de variable o concatenación precisa:

```
return numCD;
```

En el caso de concatenaciones, se suelen utilizar paréntesis:

```
return ("Número de archivos: " + numFiles);
```

Sentencias de control de flujo

Las sentencias de control de flujo indican al programa cómo ordenar y utilizar la información que se le proporciona. Con el control de flujo, se pueden repetir sentencias, ejecutarlas condicionalmente, crear bucles recursivos y controlar el comportamiento de los bucles.

Las sentencias de control de flujo se pueden agrupar en tres clases: sentencias de *iteración*, tales como for, while y do-while, las cuales permiten crear bucles; sentencias de *selección*, tales como switch, if, if-else, if-then-else, y combinaciones if-else-if escalonadas, las cuales permiten usar sentencias condicionalmente; por último, sentencias de *salto*, como break, continue y return, que permiten transferir el control a otra parte del programa.

Una forma especial de control de flujo es el *tratamiento de excepciones*. El tratamiento de excepciones proporciona un medio estructurado de detectar errores durante la ejecución del programa y hacer que devuelvan una información comprensible para poder manejarlos. Además, se puede hacer que los manejadores de excepciones realicen ciertas acciones antes de dejar que el programa termine.

Aplicación de conceptos

Los siguientes apartados demuestran cómo aplicar los términos y conceptos introducidos anteriormente en este capítulo.

Secuencias de escape

Un tipo especial de literal de caracteres son las llamadas *secuencias de escape*. Al igual que C o C++, Java utiliza secuencias de escape para representar caracteres especiales de control y caracteres que no se pueden imprimir. Una secuencia de escape se representa mediante una barra invertida (\) seguida de un código de carácter. La tabla siguiente resume las secuencias de escape:

Carácter	Secuencia de Escape
Barra invertida	`\\`
Retroceso	`\b`
Retorno de carro	`\r`
Comilla doble	`\"`
Salto de página	`\f`
Tabulación horizontal	`\t`
Nueva línea	`\n`
Carácter octal	`\DDD`
Comilla simple	`\'`
Carácter Unicode	`\uHHHH`

Los caracteres numéricos no decimales son secuencias de escape. Un carácter octal se representa mediante tres dígitos octales, mientras que un carácter Unicode se representa por una `u` minúscula seguida de cuatro dígitos hexadecimales. Por ejemplo, el número decimal 57 se representa mediante el código octal `\071` y la secuencia Unicode `\u0039`.

La cadena de ejemplo de la siguiente sentencia imprime las palabras `Name` y `"Hildegaard von Bingen"`, separadas por dos tabuladores, en una línea, y las palabras `ID` y `"1098"`, también separadas por dos tabuladores, en la segunda línea:

```
String escapeDemo = new
  String("Name\t\t\"Hildegaard von Bingen\"\nID\t\t\"1098\"");
```

Cadenas

La cadena de caracteres que se especifica en un valor de tipo `String` es un literal. El programa utilizará exactamente el literal como se escriba. No obstante, la clase `String` proporciona métodos para adjuntar cadenas (operación denominada *concatenación de cadenas*), ver y utilizar el

contenido de las cadenas (comparar cadenas, buscar cadenas o extraer una subcadena de una cadena) y convertir otros tipos de datos en cadenas. A continuación, se muestran algunos ejemplos:

- Declara variables de tipo `String` y asigna valores:

```
String firstNames = "A Joseph, Elvira y Hans";
String modifier = " de veras ";
String tastes = "les gusta el chocolate.";
```

- Obtiene una subcadena de una cadena, seleccionando desde la octava columna hasta el final de la cadena:

```
String sub = firstNames,substring(8); // "A Elvira y Hans"
```

- Compara parte de la subcadena con otra cadena, convierte una cadena a mayúsculas y después la concatena con otras cadenas para formar un valor de retorno:

```
boolean bFirst = firstNames.startsWith("Emine");  // Devuelve false en este caso.
String caps = modifier.toUpperCase();             // Guita " DE VERAS "
return firstNames + caps + tastes;                // Devuelve la frase entera:
                                          // A Elvira y Hans DE VERAS les gusta el
chocolate.
```

Para obtener más información sobre el uso de la clase `String`, consulte la documentación de la API de Sun en http://java.sun.com/j2se/1.4/docs/api/java/lang/String.html.

StringBuffer

Si desea tener más control sobre las cadenas, utilice la clase `StringBuffer`. Esta clase forma parte del paquete `java.lang`.

`StringBuffer` almacena las cadenas en un búfer, de forma que no es preciso declarar una variable `String` permanente hasta que no se necesite. Una de las ventajas de este enfoque es que no hay que volver a declarar un `String` si su contenido cambia. Se puede reservar para el búfer un tamaño mayor que el que ocupa actualmente la cadena.

`StringBuffer` proporciona métodos adicionales a los de la clase `String` que permiten modificar de otra forma el contenido de las cadenas. Por ejemplo, el método `setCharAt()` de `StringBuffer` cambia el carácter situado en el índice especificado en el primer parámetro según el nuevo valor especificado en el segundo parámetro:

```
StringBuffer word = new StringBuffer ("yellow");
word.setCharAt (0, 'b');  //la palabra es ahora "bellow"
```

Determinación del acceso

Por defecto, los miembros internos de una clase tienen acceso a la información de la clase. Del mismo modo, los miembros de la clase son accesibles entre sí. Sin embargo, este acceso se puede modificar ampliamente.

Los *modificadores de acceso* determinan el grado de visibilidad de la información de una clase o de un miembro respecto a otros miembros o clases. Los modificadores de acceso son:

- `public`: Un miembro público es visible para los miembros externos a su ámbito, siempre que la superclase sea visible. Una clase pública es visible para todas las demás clases de todos los demás paquetes.
- `private`: El acceso de un miembro privado se limita a la propia clase del miembro.
- `protected`: Todos los miembros de una clase tienen acceso a un miembro protegido. Otro tanto ocurre con los miembros de otras clases del mismo paquete (siempre que la superclase del miembro sea accesible), pero no con los de otros paquetes. Una clase protegida está disponible para otras clases del mismo paquete, pero no para las de otros paquetes.
- Si no se declara ningún modificador de acceso, el miembro estará disponible para todas las clases internas a la superclase, pero no para clases fuera del paquete.

A continuación, se muestra un ejemplo en contexto:

```
class Waistline {
  private boolean invitationGiven = false;  // Esto es private.
  private int weight = 170                  // Y se le asigna este valor.

  public void acceptInvitation() {          // Esto es public.
    invitationGiven = true;
  }

  //se declara Class JunKFood y el objeto junkFood se instancia en otra parte:
  public void eat(JunkFood junkFood) {

    /*Este objeto sólo acepta más junkFood si ha recibido una invitación
    * y puede aceptarla. Observe que isAcceptingFood()
    * trata de ver si el objeto es demasiado grande para aceptar más comida:
    */
    if (invitationGiven && isAcceptingFood()) {

      /*El nuevo peso del objeto será igual a la suma del peso
      * tenga en ese momento más el peso añadido por junkFood. El peso se
incrementa
      * mientras se siga añadiendo más junkFood:
      */
      weight += junkFood.getWeight();
    }
  }
  //Sólo el objeto sabe si está aceptado comida:
  private boolean isAcceptingFood() {
    // Este objeto sólo acepta comida si hay espacio para ello:
    return (isTooBig() ? false : true);
  }
```

```
  //Los objetos que estén en el mismo paquete pueden ver si este objeto es
demasiado grande:
  protected boolean isTooBig() {
    //Puede aceptar comida si su peso es menor de 185:
    return (weight > 185) ? true : false;
  }
}
```

Observe que `isAcceptingFood()` y `invitationGiven` son miembros `private`. Sólo los miembros internos de esta clase conocen si este objeto es capaz de "aceptar comida" o si "dispone de una invitación".

`isTooBig()` es un miembro `protegido`. Sólo las clases internas de este paquete pueden ver si el peso de este objeto supera o no su límite.

Los únicos métodos expuestos al exterior son `acceptInvitation()` y `eat()`. Estos métodos son visibles para cualquier clase.

Tratamiento de métodos

El método `main()` merece una atención especial. Constituye el punto de entrada de un programa (excepto para un applet). Se escribe del siguiente modo:

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

Dentro de los paréntesis, se permiten algunas variaciones específicas, pero la forma general es constante.

La palabra clave `static` es muy importante. Un método `static` (estático) está siempre asociado a su clase y no a una instancia particular de la clase. (La palabra clave `static` también se puede aplicar a clases. Todos los miembros de una clase `static` están asociados a la superclase de esa clase.) Los métodos `static` también se denominan *métodos de clase*.

Como el método `main()` es el punto de inicio del programa, debe ser del tipo `static` para que permanezca independiente de los múltiples objetos que el programa puede generar a partir de la superclase en la que está incluido.

La asociación de una clase `static` con una clase entera afecta a cómo se realiza la llamada a un método `static` y a cómo se llama a otros métodos desde un método `static`. Se puede llamar a los miembros `static` desde otros tipos de miembros simplemente utilizando el nombre del método; asimismo, los miembros `static` pueden llamarse entre sí del mismo modo. No es necesario crear una instancia de la clase para poder acceder a un método `static` interno a la clase.

Para acceder a miembros no `static` de una clase no `static` desde el interior de un método `static`, se debe instanciar la clase del miembro que se desea alcanzar, y utilizar esa instancia con el operador de acceso, como se haría para cualquier otra llamada a un método.

Observe que el argumento para el método `main()` es una matriz de tipo `String`, y que se permiten otros argumentos. Recuerde que en este método es donde el compilador empieza a trabajar. Cuando se especifica un argumento en la línea de comandos, el argumento se pasa como una cadena a la matriz de tipo `String` de la declaración del método `main()`, y éste utiliza el argumento para empezar a ejecutar el programa. Cuando se pasa un tipo de datos distinto de `String`, se recibe de todos modos como una cadena. Debe programar dentro del cuerpo del método `main()` la conversión que se necesite del tipo `String` al tipo de datos que necesite.

Conversiones de tipos

Revisión

La conversión de tipos es un proceso que convierte el tipo de datos de una variable sólo durante la ejecución de una determinada operación. La forma estándar para una conversión de reducción de tipo se denomina conversión explícita. Esta puede poner en peligro la integridad de los datos.

Conversiones implícitas

A veces, es el compilador el que realiza una conversión de tipos implícita. A continuación, se muestra un ejemplo:

```
if (3 > 'a') {
  ...
}
```

En este caso, el valor de `'a'` se convierte en un valor entero (el valor ASCII de la letra `a`) antes de compararlo con el número `3`.

Conversiones explícitas

La sintaxis para una conversión explícita de ampliación de tipos es muy simple:

```
<nombreDeValorAnterior> = (<nuevo tipo>) <nombreDeValorNuevo>
```

Java intenta evitar las conversiones de reducción de tipos, de modo que hay que ser más explícito cuando se pretende hacer una:

```
floatValue = (float)doubValue;   // A float "floatValue"
                                 // desde double "doubValue".

longValue = (long)floatValue;    // A long "longValue"
                                 // desde float "floatValue".
                                 // Esta es una de las cuatro construcciones
posibles.
```

(Observe que, por defecto, los decimales se redondean hacia abajo.) Asegúrese de que comprende perfectamente la sintaxis de los tipos que desea convertir explícitamente, ya que este proceso puede producir resultados no deseados.

Para obtener más información, consulte "Guía de referencia rápida del lenguaje Java" en la página 11-1.

Control de flujo

Revisión Existen tres tipos de sentencias de bucle: sentencias de iteración (`for`, `while` y `do-while`), que permiten crear bucles, sentencias de selección (`switch` y todas las sentencias `if`), que indican al programa bajo qué circunstancias puede ejecutar determinadas sentencias, y sentencias de salto (`break`, `continue` y `return`), que transfieren el control a otra parte del programa.

Bucles

Cada sentencia de un programa se ejecuta una sola vez. Sin embargo, a veces es necesario ejecutar determinadas sentencias varias veces hasta que se cumpla una condición. Java proporciona tres tipos de bucles para repetir sentencias: `while`, `do` y `for`.

- **Bucle while**

  Los bucles `while` se utilizan para crear un bloque de código que se ejecuta mientras se cumpla una determinada condición. La sintaxis general de un bucle `while` es la siguiente:

  ```
  while ( <sentencia condicional booleana> ) {
    <código a ejecutar mientras la condición se cumpla>
  }
  ```

  En primer lugar, el bucle evalúa la condición. Si el valor de la condición es `true` (verdadero), ejecuta todo el bloque de sentencias. A continuación, vuelve a evaluar la condición, y repite el proceso hasta que la condición se evalúa como `false` (falsa). En ese punto, el bucle deja de ejecutarse. Por ejemplo, para mostrar en pantalla "Looping" 10 veces:

  ```
  int x = 0;                          //Inicializa x con un valor 0.
  while (x < 10){                     //Sentencia condicional booleana.
  ```

```
    System.out.println("Looping");      //Muestra en pantalla "Looping" una
vez.
    x++;                                //Incrementa x para la siguiente
iteración.
  }
```

Cuando el bucle se ejecuta por primera vez, se comprueba si el valor de `x` es menor que `10`. Como sí lo es, el cuerpo del bucle se ejecuta. En este caso, la palabra "`Looping`" se muestra en la pantalla y, a continuación, el valor de `x` se incrementa en 1. El bucle continúa hasta que el valor de `x` llega a 10, momento en el cual el bucle deja de ejecutarse.

A menos que pretenda escribir un bucle infinito, asegúrese de que en algún punto del bucle la condición toma el valor `false` para que el bucle termine. Asimismo, un bucle se puede terminar mediante las sentencias `return`, `continue` o `break`.

- **El bucle do-while**

  El bucle `do-while` es similar al bucle `while`, con la diferencia de que evalúa la condición *después* de las sentencias en vez de antes. El siguiente código muestra el anterior bucle `while` convertido en un bucle `do`:

```
int x = 0;
do{
  System.out.println("Looping");
  x++;
}
while (x < 10);
```

  La principal diferencia entre los dos bucles es que el bucle `do-while` siempre se va a ejecutar al menos una vez, mientras que el bucle `while` no se ejecutará nunca si no se cumple inicialmente la condición.

- **Bucle for**

  El bucle `for` es el modelo de bucle más versátil. La sintaxis general de un bucle `for` es la siguiente:

```
for ( <inicialización> ; <condición booleana> ; <iteración> ) {
  <código de ejecución>
}
```

  El bucle `for` consta de tres partes: una expresión de inicialización, una condición o expresión booleana, y una expresión de iteración. La tercera expresión se utiliza habitualmente para actualizar la variable de control del bucle inicializada en la primera expresión. A continuación, se muestra el bucle `for` equivalente para el bucle `while` anterior:

```
for (int x = 0; x < 10; x++){
  System.out.println("Looping");
}
```

  Este bucle `for` y su bucle equivalente `while` son muy similares. Para casi todos los bucles `for`, existe un bucle `while` equivalente.

El bucle `for` es el modelo de bucle más versátil, sin que ello signifique perder eficiencia. Por ejemplo, tanto un bucle `while` como un bucle `for` pueden sumar los números del 1 al 20, pero un bucle `for` puede hacerlo con una línea menos de código.

While:

```
int x = 1, z = 0;
while (x <= 20) {
  z += x;
  x++;
}
```

For:

```
int z = 0;
for (int x=1; x <= 20; x++) {
  z+= x;
}
```

Se puede incluso programar el bucle `for` de modo que se ejecute la mitad de veces:

```
for (int x=1,y=20, z=0; x<=10 && y>10; x++, y--) {
  z+= x+y;
}
```

Descompongamos el bucle en sus cuatro secciones principales:

1. La expresión de inicialización: `int x =1, y=20, z=0`
2. La condición booleana: `x<=10 && y>10`
3. La expresión de iteración: `x++, y--`
4. El cuerpo principal del código ejecutable: `z+= x + y`

Sentencias de control de bucles

Estas sentencias añaden capacidad de control del flujo de ejecución a las sentencias de bucle.

- **Sentencia break**

  La sentencia `break` permite salir de la estructura de un bucle antes de que se cumpla la condición de prueba. Cuando se llega a una sentencia `break`, el bucle termina inmediatamente sin ejecutar el código restante. Por ejemplo:

```
int x = 0;
while (x < 10){
  System.out.println("Looping");
  x++;
  if (x == 5)
    break;
    else
    ... //hace algo diferente
}
```

  En este ejemplo, el bucle deja de ejecutarse cuando `x` es igual a `5`.

- **Sentencia continue**

  La sentencia `continue` se utiliza para omitir el resto del bucle y continuar la ejecución en la siguiente iteración del bucle.

```
for ( int x = 0 ; x < 10 ; x++){
  if(x == 5)
    continue; //vuelve al principio del bucle cuando x=6
  System.out.println("Looping");
}
```

  En este ejemplo, no se imprime "`Looping`" si `x` es 5, pero sí se imprimirá una vez que x sea igual o superior a 6.

Sentencias condicionales

Las sentencias condicionales se utilizan para proporcionar capacidad de decisión al código. En Java, existen dos estructuras condicionales: la sentencia `if-else` y la sentencia `switch`.

- **Sentencia if-else**

  La sintaxis de una sentencia `if-else` es la siguiente:

```
if (<condición1>) {
  ... //bloque de código 1
}
else if (<cóndición2>) {
  ... //bloque de código 2
}
  else {
  ... //bloque de código 3
}
```

  La sentencia `if-else` se compone habitualmente de varios bloques. Sólo uno de los bloques se ejecuta cuando se ejecuta la sentencia `if-else`, según cuál de las condiciones se evalúe como verdadera.

  Los bloques `else-if` y `else` son opcionales. Asimismo, la sentencia `if-else` no se limita a tres bloques: puede contener tantos bloques `else-if` como sea necesario.

  Los siguientes ejemplos muestran el uso de la sentencia `if-else`:

```
if ( x % 2 == 0)
  System.out.println("x es par");
    else
  System.out.println("x es impar");
if (x == y)
  System.out.println("x es igual a y");
else if (x < y)
  System.out.println("x es menor que y");
    else
  System.out.println("x es mayor que y");
```

- **Sentencia switch**

  La sentencia `switch` es similar a la sentencia `if-else`. A continuación, se muestra la sintaxis general de la sentencia `switch`:

```
switch (<expresión>){
  case <valor1>: <bloqueCódigo1>;
      break;
  case <valor2>: <bloqueCódigo2>;
      break;
default   : <bloqueCódigo3>;
}
```

  Observe lo siguiente:

  - Si sólo existe una sentencia en un bloque de código, no es necesario encerrar el bloque entre paréntesis.
  - El bloque `default` se corresponde con el bloque `else` de una sentencia `if-else`.
  - Los bloques de código se ejecutan según el valor de una variable o expresión, no según una condición.
  - El valor de `<expresión>` debe ser del tipo entero o de un tipo que se pueda convertir sin problemas al tipo `int`, tal como el tipo `char`.
  - Los valores de las etiquetas `case` deben ser expresiones constantes del mismo tipo que el argumento `expresión` original.
  - La palabra clave `break` es opcional. Se utiliza para finalizar la ejecución de la sentencia `switch` una vez que se ejecuta un bloque de código. Si no se utiliza después de `bloqueCódigo1`, entonces `bloqueCódigo2` se ejecuta inmediatamente después de que `bloqueCódigo1` termine de ejecutarse.
  - Si un bloque de código debe ejecutarse cuando `expression` está dentro de entre una serie de valores, cada uno de los valores debe especificarse del siguiente modo: `case <valor>:`.

  A continuación, se muestra un ejemplo, en el que `c` es del tipo `char`:

```
switch (c){
  case '1': case '3': case '5': case '7': case '9':
    System.out.println("c es un número impar");
    break;
  case '0': case '2': case '4': case '6': case '8':
    System.out.println("c es un número par");
    break;
  case ' ':
    System.out.println("c es un espacio");
    break;
default   :
    System.out.println("c no es ni un número ni un espacio");
  }
```

La sentencia `switch` evalúa `c` y salta a la etiqueta `case` cuyo valor es igual a `c`. Si no existe ningún valor de `case` igual a `c`, se ejecuta la sección `default`. Observe cómo es posible utilizar varios valores para cada bloque.

Tratamiento de excepciones

El tratamiento de excepciones proporciona un medio estructurado de capturar errores durante la ejecución del programa y hacer que devuelvan una información comprensible para poder manejarlos. Además, se puede hacer que los manejadores de excepciones realicen ciertas acciones antes de dejar que el programa termine. El tratamiento de excepciones utiliza las palabras clave `try`, `catch`, y `finally`. Un método puede declarar una excepción mediante las palabras clave `throws` y `throw`.

En Java, una excepción puede ser una subclase de las clases `java.lang.Exception` o `java.lang.Error`. Cuando un método declara que se ha producido una excepción, se dice que *lanza* una excepción. *Capturar* una excepción significa gestionar una excepción.

Las excepciones que se declaran explícitamente en la declaración de un método *deben* capturarse; de lo contrario, el código no se compilará. Las excepciones que no se declaran explícitamente en la declaración de un método aún pueden provocar la detención del programa en ejecución, y éste sí se compilará. No deje de tener en cuenta que un buen tratamiento de excepciones hace que el código sea más robusto.

Para capturar una excepción, se debe encerrar el código que puede producirla en un bloque `try` y, a continuación, encerrar el código que se desee utilizar para tratar la excepción en un bloque `catch`. Si existe código importante (como, por ejemplo, código de limpieza) que se debe ejecutar incluso en el caso de que se lance una excepción y el programa se cierre, incluya ese código en un bloque `finally` al final. A continuación, se muestra un ejemplo de cómo funciona esto:

```
try {
  ... // Algún bloque de código que haya lanzado una excepción llega aquí.
}
catch( Exception e ) {
  ... // El código de tratamiento de excepciones llega aquí
  // La siguiente línea da como salida el seguimiento de la pila de la
excepción:
    e.printStackTrace();
}
finally {
  ... // Queda garantizado que él código que va aquí se va a ejecutar,
      // se lance o no una excepción en el bloque try.
}
```

El bloque `try` se utiliza para encerrar cualquier parte del código que pueda producir una excepción que necesite tratamiento. Si no se lanza ninguna excepción, se ejecutará todo el código del bloque `try`. Sin embargo, si se lanza una excepción, el código del bloque `try` deja de ejecutarse en el punto donde se produjo la excepción, y el control se transfiere al bloque `catch`, en donde la excepción recibe tratamiento.

Se puede hacer todo lo que sea necesario para tratar la excepción en uno o varios bloques `catch`. El modo más simple de tratar excepciones consiste en tratarlas todas en un único bloque `catch`. Para ello, el argumento entre paréntesis después de `catch` debería indicar la clase `Exception`, seguida de un nombre de variable que se asignará a esta excepción. Esto quiere decir que se capturará cualquier excepción que sea una instancia de `java.lang.Exception` o cualquiera de sus subclases; en otras palabras, cualquier excepción.

Si se necesita escribir un código distinto para cada tipo de excepción, se pueden utilizar varios bloques `catch`. En ese caso, en vez de utilizar `Exception` como el tipo de excepción en el argumento `catch`, se indica el nombre de la clase del tipo específico de excepción que se desea capturar. Ésta puede ser cualquier subclase de `Exception`. Tenga en cuenta que el bloque catch siempre captura el tipo de excepción indicado y cualquiera de sus subclases.

El código del bloque `finally` se ejecuta siempre, incluso aunque el código del bloque `try` no se complete por alguna razón. Por ejemplo, el código del bloque `try` podría no completarse si se lanza una excepción pero, aún así, el código del bloque `finally` sí se ejecutará. Esto hace que el bloque `finally` sea un buen lugar para colocar código de limpieza.

Si se sabe que un método que se está escribiendo va a ser llamado por otra parte del código, podría dejarse a la parte que lo llama la tarea de tratar las excepciones que pudiera lanzar el método. En ese caso, simplemente se declararía que el método puede lanzar una excepción. El código susceptible de causar una excepción puede utilizar la palabra clave `throws` para declarar la excepción. Esto constituye una alternativa a capturar la excepción, ya que si un método declara que lanza una excepción, no tiene por qué tratar esa excepción.

A continuación, se muestra un ejemplo del uso de `throws`:

```
public void myMethod() throws SomeException {
  ... // El código que empieza aquí podría lanzar SomeException, o una de sus
subclases.
      // se asume que SomeException es una subclase de Exception.
}
```

La palabra clave `throw` también se puede utilizar para indicar una situación anómala. Por ejemplo, se podría utilizar para lanzar una excepción cuando un usuario introduce información no válida y se le desea avisar con un mensaje de error. Para ello, utilice una sentencia como la siguiente:

```
throw new SomeException("entrada no válida");
```

Capítulo

5

Las bibliotecas de clase Java

La mayoría de los lenguajes de programación utilizan bibliotecas de clases preprogramadas para implementar determinadas funciones. En el lenguaje Java, estos grupos de clases relacionadas, llamadas paquetes, varían según la edición de Java. Cada edición se utiliza para propósitos específicos, tales como aplicaciones, aplicaciones empresariales y productos de consumo.

Ediciones de la plataforma Java 2

La Plataforma Java 2 está disponible en varias ediciones utilizadas para propósitos diversos. Como Java es un lenguaje que se puede ejecutar en cualquier sitio y sobre cualquier plataforma, se utiliza en una gran variedad de entornos: Internet, intranets, productos de electrónica de consumo y aplicaciones informáticas. Debido a la gran variedad de aplicaciones, Java se presenta en varias ediciones: Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE), y Java 2 Micro Edition (J2ME). En algunos casos, tales como en el desarrollo de aplicaciones para empresa, se utiliza un conjunto mayor de paquetes. En otros, como en los productos de electrónica de consumo, sólo se utiliza una pequeña porción del lenguaje. Cada edición contiene un Kit de Desarrollo de Software Java 2

(SDK), utilizado para desarrollar aplicaciones, y un Entorno de Ejecución Java 2 (JRE) utilizado para ejecutar aplicaciones.

Tabla 5.1 Ediciones de la Plataforma Java 2

Compatibilidad con plataformas de Java 2	Abreviatura	Descripción
Edición Standard	J2SE	Contiene las clases que son el núcleo del lenguaje Java.
Edición Enterprise	J2EE	Contiene las clases de J2SE y otras adicionales para el desarrollo de aplicaciones para empresa.
Edición Micro	J2ME	Contiene un subconjunto de las clases de J2SE y se utiliza en productos de informática de consumo.

Edición Standard

La Plataforma Java 2, Edición Estándar (J2SE) proporciona a los desarrolladores un entorno de desarrollo multi-plataforma seguro, estable y rico en prestaciones. Esta edición de Java soporta características fundamentales, tales como conectividad con bases de datos, diseño de interfaces de usuario, entrada/salida y programación en red, e incluye paquetes fundamentales del lenguaje Java.

Consulte

- "Java 2 Platform Standard Edition Overview" en http://java.sun.com/j2se/1.4/
- "Introducing the Java Platform" en http://developer.java.sun.com/developer/onlineTraining/new2java/programming/intro)
- "Paquete Java 2 Standard Edition" en la página 5-3

Edición Enterprise

Java 2 Enterprise Edition (J2EE) proporciona al desarrollador herramientas para generar y distribuir aplicaciones empresariales multinivel. J2EE incluye los paquetes de J2SE así como paquetes adicionales que permiten el desarrollo de JavaBeans Enterprise, servlets Java, páginas JavaServer, XML y control flexible de transacciones.

Consulte

- "Java 2 Platform Enterprise Edition Overview" en http://java.sun.com/j2ee/overview.html)

- Los artículos técnicos sobre Java 2 Enterprise Edition en http://developer.java.sun.com/developer/technicalArticles/J2EE/index.html)

Edición Micro

Java 2 Micro Edition (J2ME) se utiliza en una gran variedad de productos electrónicos de consumo, tales como buscapersonas, tarjetas inteligentes, teléfonos móviles, asistentes personales digitales de bolsillo (PDA) y dispositivos descodificadores para televisión (set-top box). Aunque J2ME proporciona las mismas ventajas de portabilidad de código entre distintas plataformas, capacidad de funcionar en cualquier sitio y seguridad en red que J2SE y J2EE, utiliza un conjunto de paquetes más reducido. J2ME incluye un subconjunto de los paquetes de J2SE al que se suma un paquete adicional específico de la versión Micro Edition, `javax.microedition.io`. Además, las aplicaciones J2ME son ampliables hacia arriba de modo que puedan funcionar con J2SE y J2EE.

Consulte "Java 2 Platform Micro Edition Overview"en http://java.sun.com/j2me/
"Consumer & Embedded Products technical articles" en http://developer.java.sun.com/developer/technicalArticles/ConsumerProducts/index.html

Paquete Java 2 Standard Edition

La Plataforma Java 2 Standard Edition (J2SE) incorpora una imponente biblioteca que incluye conectividad con bases de datos, diseño de interfaces de usuario, entrada y salida (I/O) y programación en red. Estas bibliotecas se organizan en grupos de clases relacionadas denominados paquetes. La siguiente tabla describe brevemente algunos de estos paquetes.

Tabla 5.2 Paquetes J2SE

Paquete	Nombre de Paquete	Descripción
Lenguaje	`java.lang`	Clases que contienen el núcleo principal del lenguaje Java.
Utilidades	`java.util`	Soporte para utilidades de estructuras de datos.
E/S	`java.io`	Soporte para diversos tipos de entrada/salida.
Texto	`java.text`	Soporte para el manejo localizado de texto, fechas, números y mensajes.
Matemáticas	`java.math`	Clases para realizar operaciones aritméticas con enteros y coma flotante.

Tabla 5.2 Paquetes J2SE (continuación)

Paquete	Nombre de Paquete	Descripción
AWT	`java.awt`	Diseño de interfaz de usuario y gestión de sucesos.
Swing	`javax.swing`	Clases para la creación de componentes ligeros 100% Java que se comportan de forma similar en todas las plataformas.
Javax	`javax`	Extensiones al lenguaje Java.
Applet	`java.applet`	Clases para la creación de applets.
Beans	`java.beans`	Clases para desarrollar JavaBeans.
Reflexión	`java.lang.reflect`	Clases utilizadas para obtener información de clases en tiempo de ejecución.
SQL	`java.sql`	Soporte para acceder a y procesar datos en bases de datos.
RMI	`java.rmi`	Soporte para programación distribuida.
Trabajo en red	`java.net`	Clases que soportan el desarrollo de aplicaciones en red.
Seguridad	`java.security`	Soporte para seguridad criptográfica.

Nota Los paquetes de Java varían según la edición de la plataforma Java 2. El kit de desarrollo de software (SDK) de Java 2 está disponible en varias ediciones, las cuales se utilizan para distintos propósitos: Standard Edition (J2SE), Enterprise Edition (J2EE) y Micro Edition (J2ME).

Consulte

- "Ediciones de la plataforma Java 2" en la página 5-1
- "Java 2 Platform, Standard Edition, API Specification" en la documentación API del JDK
- Tutorial de Sun, "Creating and using packages" (Creación y utilización de paquetes) en http://www.java.sun.com/docs/books/tutorial/java/interpack/packages.html
- "Páquetes" en "Gestión de vías de acceso" in *Creación de aplicaciones con JBuilder*

El paquete de lenguaje: java.lang

Uno de los paquetes más importantes de la biblioteca de clases de Java es el paquete `java.lang`. Este paquete, que se importa automáticamente en cada programa Java, contiene las clases principales que son fundamentales en el diseño del lenguaje de programación Java.

Consulte

- `java.lang` en la documentación API del JDK
- "Clases java.lang principales" en la página 5-12.

El paquete de utilidades: java.util

El paquete `java.util` contiene diversas clases de utilidad e interfaces que son importantes para el desarrollo en Java. Las clases de este paquete permiten utilizar colecciones así como servicios de fecha y hora.

Consulte

- `java.util` en la Documentación API del JDK
- "Clases java.util principales" en la página 5-19

El paquete de E/S: java.io

El paquete `java.io` proporciona funcionalidad para leer y escribir datos en diversos dispositivos. Java también permite utilizar flujos de caracteres de entrada y salida. Además, la clase `File` del paquete `java.io` utiliza una representación abstracta e independiente del sistema de los nombres de archivos y directorios que resulta apropiada para plataformas no UNIX. Las clases de este paquete se dividen en los siguientes grupos: clases de flujo de entrada, clases de flujo de salida, clases de archivo y la clase `StreamTokenizer`.

Consulte

- `java.io` en la documentación API del JDK
- "Clases java.io principales" en la página 5-22

El paquete de texto: java.text

El paquete `java.text` suministra clases e interfaces que proporcionan funcionalidad de adaptación a distintos idiomas para texto, fechas, números y mensajes. Las clases de este paquete, tales como `NumberFormat`, `DateFormat` y `Collator`, pueden aplicar formato a números, fechas, horas y cadenas según un idioma específico. Otras clases permiten analizar, buscar y ordenar cadenas.

Consulte

- `java.text` de la documentación API del JDK
- "Internacionalización de programas con JBuilder" en *Creación de aplicaciones con JBuilder*

El paquete de matemáticas: java.math

El paquete `java.math` (no confundirlo con la clase `java.lang.Math`) proporciona clases para realizar operaciones aritméticas enteras (`BigInteger`) y de punto flotante (`BigDecimal`) de precisión arbitraria.

La clase `BigInteger` proporciona funcionalidad para representar enteros arbitrariamente grandes.

La clase `BigDecimal` se utiliza para cálculos que requieren decimales, tales como cálculos monetarios y también permite operaciones de aritmética básica, manipulación de escala, comparación, conversión de formato y "hashing".

Consulte

- `java.math` en la documentación API del JDK
- `java.lang.Math` en la documentación API del JDK
- "Matemática de precisión arbitraria" en la Guía de características del JDK

El paquete AWT: java.awt

El paquete "Abstract Window Toolkit" (AWT), que forma parte de Java Foundation Classes (JFC), proporciona funcionalidad para la programación de interfaces gráficas de usuario (GUI) e incluye características tales como componentes de interfaz de usuario, modelos de tratamiento de sucesos, gestores de diseño, herramientas gráficas y de creación de imágenes y clases de transferencia de datos para operaciones de cortar y pegar.

Consulte

- `java.awt` en la documentación API del JDK
- "Abstract Window Toolkit (AWT)" en la Guía de características del JDK
- "Fundamentos de AWT" en http://developer.java.sun.com/developer/onlineTraining/awt/
- "Tutorial: Creación de un applet" en *Introducción a JBuilder*
- *Diseño de aplicaciones con JBuilder*

El paquete Swing: javax.swing

El paquete `javax.swing` proporciona un conjunto de componentes "ligeros" (escritos totalmente en lenguaje Java) que presentan automáticamente el estilo de interfaz de cualquier sistema operativo. Los componentes Swing

son versiones 100% Java del conjunto de componentes AWT existente, tales como botón, barra de desplazamiento y etiqueta, con un conjunto de componentes adicionales, como vista de árbol, tabla y panel con pestañas.

Nota los paquetes `javax` son extensiones del lenguaje Java básico.

Consulte

- `javax.swing` en la documentación API del JDK
- "Java Foundation Classes (JFC)" en `http://java.sun.com/docs/books/tutorial/post1.0/preview/jfc.html`
- Tutorial Swing de Sun, "Trail: Creating a GUI with JFC/Swing" en `http://www.java.sun.com/docs/books/tutorial/uiswing/index.html`
- Los capítulos relativos en *Diseño de aplicaciones con JBuilder*:
  - "Introducción"
  - "Gestión de la paleta de componentes"
  - "Gestores de diseño"
  - "Paneles y diseños anidados"
  - "Tutorial: Creación de editor de texto en Java"

Los paquetes Javax: javax

Los distintos paquetes existentes`javax` son extensiones del lenguaje Java básico. Entre estos, se incluyen paquetes como `javax.swing`, `javax.sound`, `javax.rmi`, `javax.transactions` y `javax.naming`. Los desarrolladores también pueden crear sus propios paquetes `javax` personalizados.

Consulte

- `javax.accessibility` en la documentación API del JDK
- `javax.naming` en la documentación API del JDK
- `javax.rmi` en la documentación API del JDK
- `javax.sound.midi` en la documentación API del JDK
- `javax.sound.sampled` en la documentación API del JDK
- `javax.swing` en la documentación API del JDK
- javax.transaction en la documentación API del JDK

El paquete Applet: java.applet

El paquete `java.applet` suministra clases para crear applets, así como clases que utilizan los applets para comunicarse con su contexto, normalmente un navegador Web. Los applets son programas Java no diseñados para ejecutarse de forma autónoma, sino para incluirse dentro de otra aplicación. Normalmente, los applet se almacenan en un servidor de Internet/intranet, se llaman desde una página HTML y se descargan en múltiples plataformas cliente, en donde se ejecutan en una máquina virtual de Java (JVM) residente en el navegador del equipo cliente. Un Administrador de seguridad supervisa este proceso de transferencia y ejecución para impedir que las applets realicen determinadas tareas como formatear el disco duro o establecer conexiones con ordenadores "no fiables".

Por motivos de seguridad y compatibilidad con el JDK del navegador, es importante entender bien lo que son los applets para poder desarrollarlos. Los applets no presentan la funcionalidad completa de los programas Java por razones de seguridad. Los applets se basan en la versión del JDK del navegador, que puede no estar actualizada. Muchos de los navegadores, en el momento de escribir esto, no admiten totalmente el último JDK. Por ejemplo, la mayoría de los navegadores incluyen una versión antigua del JDK que no admite Swing. Por lo tanto, los applets que utilizan componentes Swing no funcionan en estos navegadores.

Consulte

- El resumen del paquete `java.applet` en la documentación API del JDK
- El tutorial de Sun, “Trail: Writing applets” en http://www.java.sun.com/docs/books/tutorial/applet/index.html
- Capítulo 9, “Introducción a la máquina virtual de Java”
- “Las applets” en la *Guía del desarrollador de aplicaciones Web*
- “Tutorial: Generación de un applet” en *Introducción a JBuilder*

El paquete Beans: java.beans

El paquete `java.beans` contiene clases relacionadas con el desarrollo de JavaBeans. JavaBeans son clases Java que actúan como componentes autocontenidos y reutilizables, y que extienden la capacidad de "escritura única y ejecución en cualquier sitio" de Java al diseño de componentes reutilizables. Estas piezas reutilizables de código se pueden manipular y actualizar con un impacto mínimo sobre las pruebas del programa.

Consulte

- `java.beans` de la documentación API del JDK

- Los artículos técnicos sobre la tecnología JavaBeans en http://developer.java.sun.com/developer/technicalArticles/jbeans/index.html
- "Creación de JavaBeans con BeansExpress" en *Creación de aplicaciones con JBuilder*

El paquete de reflexión: java.lang.reflect

El paquete `java.lang.reflect` proporciona clases e interfaces para examinar y manipular clases durante la ejecución. La reflexión permite el acceso a información sobre campos, métodos y constructores de clases cargadas. El código Java puede utilizar esta información reflejada para actuar sobre equivalentes de objetos.

Las clases de este paquete proporcionan a aplicaciones como depuradores, intérpretes, inspectores de objetos y examinadores de clases, y a servicios, como Serialización de objetos y JavaBeans, acceso a miembros públicos o miembros declarados por una clase.

Consulte

- `java.lang.reflect` en la documentación API del JDK
- "Reflection", en la guía de características del JDK

Procesado de XML

Java, junto con XML (Extensible Markup Language - Lenguaje de marcas ampliable), proporciona un entorno de trabajo portable y flexible para la creación, intercambio y manipulación de la información entre aplicaciones y a través de Internet, así como para la transformación de documentos XML en otros tipos de documento. La API de Java para el procesado XML (JAXP) incluye las facilidades básicas para trabajar con documentos XML: DOM (Document Object Model - Modelo de objetos de documento), SAX (Simple API for XML Parsing - La API sencilla de XML), XSLT (XSL Transformation - La transformación de XSL) y diferentes niveles de acceso para analizadores.

Consulte

- `org.w3c.dom` en la documentación API del JDK
- `org.xml.sax` en la documentación API del JDK
- `javax.xml.transform` en la documentación API del JDK
- `javax.xml.parsers` en la documentación API del JDK
- "Java Technology & XML Home Page" en `http://java.sun.com/xml/`
- "XML in the Java 2 Platform" en la guía de características del JDK

- El tutorial sobre XML de Sun en `http://java.sun.com/xml/tutorial_intro.html`

El paquete SQL: java.sql

El paquete `java.sql` contiene clases que proporcionan a la API acceso y procesamiento de datos de una fuente de datos. El paquete `java.sql` también se conoce como API JDBC 2.0 (Conectividad de base de datos Java). Esta API incluye una plataforma para instalar dinámicamente diversos controladores para acceder a diferentes tipos de fuentes de datos. JDBC es un estándar de la industria que permite a la plataforma Java conectarse con prácticamente cualquier base de datos, incluso aquellas escritas en otros lenguajes, como SQL.

El paquete `java.sql` incluye clases, interfaces y métodos para realizar conexiones con bases de datos, enviar sentencias SQL a bases de datos, recuperar y actualizar resultados de consultas, asignar valores SQL, proporcionar información sobre bases de datos, lanzar excepciones y proporcionar seguridad.

Consulte

- `java.sql` en la documentación API del JDK
- El tutorial de Sun, "Trail: JDBC Database Access" en `http://web2.java.sun.com/docs/books/tutorial/jdbc/index.html`
- "La referencia SQL" de JBuilder en la *Guía del desarrollador de JDataStore*

El paquete RMI: java.rmi

El paquete `java.rmi` proporciona clases para invocación de métodos remotos de Java (RMI). La llamada a métodos remotos permite crear aplicaciones distribuidas "Java a Java" en las que se puede llamar a los métodos de los objetos remotos de Java desde otras máquinas virtuales de Java, incluso en distintos anfitriones. Los programas de Java pueden llamar a un objeto remoto después de haber obtenido la correspondiente referencia, ya sea buscando el objeto en el servicio de nomenclatura de inicio que proporciona la RMI, ya sea recibiendo la referencia como argumento de un valor devuelto. Los clientes pueden llamar a un objeto remoto en un servidor, que puede ser, a su vez, cliente de otros objetos remotos. RMI utiliza la serialización de objetos para recodificar los parámetros y devolverlos a su formato original y no trunca los tipos; admite polimorfismo orientado a objetos.

En el directorio `samples/Rmi` de la instalación de JBuilder, se encuentra una aplicación RMI de ejemplo, `SimpleRMI.jpr`. Consulte el archivo de proyecto HTML, SimpleRMI.html, que contiene una descripción de la aplicación de ejemplo. (Este ejemplo es una función de JBuilder SE y Enterprise.)

Puede encontrar un ejemplo de escritura de una aplicación de base de datos distribuida por medio de RMI y DataSetData en el subdirectorio /samples/DataExpress/StreamableDataSets del directorio de instalación de JBuilder. Este ejemplo incluye una aplicación servidor que extrae datos de la tabla de muestra de empleados y los envía por medio de RMI en forma de DataSetData. La aplicación cliente se comunica con el servidor por medio de un Provider y un Resolver personalizados y presenta los datos en una rejilla. (Este ejemplo es una función de JBuilder Enterprise.)

Consulte

- `java.rmi` de la documentación API del JDK
- "Java Remote Method Invocation (RMI)" en la Guía de características del JDK
- "The Java Remote Method Invocation - Distributed Computing for Java (a White Paper)" en `http://java.sun.com/marketing/collateral/javarmi.html`
- Aplicaciones RMI de ejemplo en JBuilder: `samples/RMI/` y `samples/DataExpress/StreamableDataSets/` en el directorio de la instalación de JBuilder

El paquete de red: java.net

El paquete java.net contiene clases para desarrollar aplicaciones en red. Mediante las clases socket, es posible comunicarse con cualquier servidor de Internet o implementar un servidor de Internet propio. Las clases también permiten recuperar datos de Internet.

Consulte

- `java.net` en la documentación API del JDK
- "Networking Features" en la Guía de características del JDK

El paquete de seguridad: java.security

El paquete de seguridad, `java.security`, define clases e interfaces para el modelo de seguridad. Existen dos categorías de clases:

- Clases que implementan el control de accesos e impiden que el código no autorizado pueda ejecutar operaciones sensibles.
- Clases de autentificación, que implementan compendios de mensajes y firmas digitales, y permiten autenticar clases y otros objetos.

Con estas clases, los desarrolladores pueden proteger el acceso a applets y código Java, incluidos beans, servlets y aplicaciones, mediante permisos y criterios de seguridad. Cuando se carga el código, se le asignan permisos

según los criterios de seguridad. Los permisos especifican a qué recursos se puede acceder, tales como acceso de lectura/escritura o conexión. Los criterios que controlan qué permisos están disponibles se suelen inicializar desde un archivo externo configurable que define las políticas de seguridad del código. El uso de permisos y política de seguridad permite un control de accesos flexible, configurable y extensible para el código.

Consulte

- `java.security` en la documentación API del JDK
- "Security" en la guía de características del JDK

Clases java.lang principales

La clase Object: java.lang.Object

La clase `Object` del paquete `java.lang` es la superclase de todas las demás clases de Java. Esto significa que la clase `Object` es la raíz de la jerarquía de clases, y todas las demás clases de Java se derivan de ella. La propia clase `Object` contiene un constructor y varios métodos importantes, entre los que se incluyen `clone()`, `equals()` y `toString()`.

Método	Argumento	Descripción
`clone`	`()`	Crea y devuelve una copia de un objeto.
`equals`	`(Object obj)`	Indica si otro objeto es "igual" al objeto especificado.
`toString`	`()`	Devuelve una representación de cadena de un objeto

Un objeto que utiliza el método `clone()` simplemente realiza una copia de sí mismo. Cuando se realiza una copia, primero se asigna memoria para el objeto clonado y, después, se copia el contenido del objeto original en la memoria asignada. En el siguiente ejemplo, en el que la clase `Document` implementa la interfaz `Cloneable`, se crea una copia de la clase `Document` con una propiedad `text` y una propiedad `author` mediante el método `clone()`. Un objeto no se puede clonar a menos que implemente la interfaz `Cloneable`.

```
Document document1 = new Document("docText.txt", "Joe Smith");
Document document2 = document1.clone();
```

El método `equals()` comprueba la igualdad de dos objetos del mismo tipo comparando las propiedades de ambos objetos. Simplemente, devuelve un valor booleano que es el resultado de la comparación entre el objeto que realiza la llamada y el objeto que se pasa como parámetro. Por ejemplo, si la llamada a `equals()` la realiza un objeto idéntico al objeto que se pasa como parámetro, el método `equals()` devuelve un valor true.

El método `toString()` devuelve una representación de cadena que representa el objeto. Para que este método devuelva información apropiada sobre diferentes tipos de objetos, la clase del objeto debe redefinirlo.

Consulte `java.lang.Object` en la documentación API del JDK

Clases englobadoras de tipos

En Java, por razones de rendimiento, los tipos de datos primitivos no se utilizan como objetos. Entre estos tipos de datos primitivos, se incluyen números, valores booleanos y caracteres.

No obstante, algunas clases y métodos de Java requieren que los tipos de datos primitivos sean objetos. Java utiliza clases englobadoras para encapsular el tipo de datos primitivo como un objeto, tal como se muestra en la tabla siguiente.

Tipo primitivo	Descripción	Englobador
`boolean`	True o False (1 Bit)	`java.lang.Boolean`
`byte`	_128 127 (entero de 8 bits con signo)	`java.lang.Byte`
`char`	Carácter Unicode (16 bits)	`java.lang.Character`
`double`	+1.79769313486231579E+308 a +4.9406545841246544E-324 (64 bits)	`java.lang.Double`
`float`	+3.40282347E+28 a +1.40239846E-45 (32 bits)	`java.lang.Float`
`int`	_2147483648 2147483647 (entero de 32 bits con signo)	`java.lang.Integer`
`long`	_9223372036854775808 9223372036854775807 (entero de 64 bits con signo)	`java.lang.Long`
`short`	_32768 a 32767 (entero de 16 bits con signo)	`java.lang.Short`
`void`	Una clase que simplemente realiza la función de contenedor, de la que no se pueden crear instancias y que mantiene una referencia al objeto Class que representa el tipo primitivo void de Java.	`java.lang.Void`

El constructor para las clases englobadoras, tales como `Character(char value)`, simplemente toma un argumento del tipo de la clase que está englobando. Por ejemplo, el siguiente fragmento de código muestra cómo se construye una clase englobadora para `Character`.

```
Character charWrapper = new Character('T');
```

Aunque cada una de estas clases contiene sus propios métodos, algunos métodos son estándar para todos los objetos. Entre éstos, se incluyen métodos que devuelven un tipo primitivo, `toString()` y `equals()`.

Cada clase englobadora dispone de un método, tal como `charValue()`, que devuelve el tipo primitivo de la clase englobadora. El siguiente fragmento de código utiliza el objeto `charWrapper`. Observe que `charPrimitive` es un tipo de datos primitivo (declarado como un `char`). Mediante este método, los tipos de datos primitivos se pueden asignar a englobadores de tipos.

```
char charPrimitive = charWrapper.charValue();
```

Los métodos `toString()` y `equals()` se utilizan de forma similar al caso de la clase Object.

La clase Math: java.lang.Math

La clase `Math` del paquete `java.lang` (no confundir con el paquete `java.math`) proporciona métodos útiles que implementan funciones matemáticas habituales. Esta clase no está instanciada, y se declara `final`, de modo que no se puede derivar en subclases. Algunos de los métodos incluidos en esta clase son: `sin()`, `cos()`, `exp()`, `log()`, `max()`, `min()`, `random()`, `sqrt()` y `tan()`. A continuación, se muestran algunos ejemplos de estos métodos.

```
double d1 = Math.sin(45);
double d2 = 23.4;
double d3 = Math.exp(d2);
double d4 = Math.log(d3);
double d5 = Math.max(d2, Math.pow(d1, 10));
```

Algunos de estos métodos están sobrecargados para que puedan aceptar y devolver diferentes tipos de datos.

La clase `Math` también declara las constantes `PI` y `E`.

Nota El paquete `java.math`, a diferencia de `java.lang.Math`, proporciona clases auxiliares para trabajar con grandes cifras.

Consulte

- `java.lang.Math` de la documentación API del JDK
- `java.math` de la documentación API del JDK

La clase String: java.lang.String

La clase `String` del paquete `java.lang` se utiliza para representar cadenas de caracteres. A diferencia de C o C++, Java no utiliza matrices de caracteres para representar cadenas. Las cadenas son constantes, lo que significa que su valor no puede cambiar una vez creadas. Normalmente, la clase `String` se construye cuando el compilador de Java encuentra una cadena entre comillas. No obstante, las cadenas se pueden construir de varias formas.

La siguiente tabla muestra varios de los constructores de `String` y los argumentos que aceptan.

Constructor	Argumento	Descripción
`Cadena`	`()`	Inicializa un nuevo objeto `String`.
`Cadena`	`(String value)`	Inicializa un nuevo objeto `String` con el contenido del argumento `String`.
`Cadena`	`(char[] value)`	Crea un objeto `String` que contiene la matriz en la misma secuencia.
`Cadena`	`(char[] value, int offset, int count)`	Crea un nuevo objeto `String` que contiene una submatriz del argumento.
`Cadena`	`(StringBuffer buffer)`	Inicializa un nuevo objeto `String` con el contenido del argumento de tipo `StringBuffer`.

La clase `String` contiene varios métodos importantes que son esenciales para procesar cadenas. Estos métodos se utilizan para modificar, comparar y analizar cadenas. Como las cadenas son constantes y no se pueden cambiar, ninguno de estos métodos puede alterar la secuencia de caracteres. La clase `StringBuffer`, descrita en la próxima sección, proporciona métodos para cambiar cadenas.

La siguiente tabla muestra algunos de los métodos más importantes y declara qué valores aceptan y devuelven.

Método	Argumento	Devuelve	Descripción
`length`	`()`	`int`	Devuelve el número de caracteres de la cadena.
`charAt`	`(int index)`	`char`	Devuelve el carácter correspondiente al índice especificado de la cadena.
`compareTo`	`(String value)`	`int`	Compara una cadena con la cadena que se pasa como argumento.
`indexOf`	`(int ch)`	`int`	Devuelve el valor del índice correspondiente a la primera aparición del carácter especificado.
`substring`	`(int beginIndex, int endIndex)`	`Cadena`	Devuelve una subcadena definida por los índices inicial y final.
`concat`	`(String str)`	`Cadena`	Añade la cadena `String` especificada al final de esta cadena.
`toLowerCase`	`()`	`Cadena`	Devuelve la cadena en minúsculas.
`toUpperCase`	`()`	`Cadena`	Devuelve la cadena en mayúsculas.
`valueOf`	`(Object obj)`	`Cadena`	Devuelve la representación de cadena del argumento `Object`.

Una característica muy eficiente de muchos de estos métodos es que están sobrecargados, lo que les confiere una mayor flexibilidad. El siguiente

ejemplo muestra cómo se pueden utilizar la clase `String` y algunos de sus métodos.

Importante Recuerde que una matriz y los índices String comienzan por cero

```
String s1 = new String("Hola a todos.");

char cArray[] = {'J', 'B', 'u', 'i', 'l', 'd', 'e', 'r'};
String s2 = new String(cArray);          //s2 = "JBuilder"

int i = s1.length();                     //i = 12
char c = s1.charAt(6);                   //c = 'W'
i = s1.indexOf('e');                     //i = 1 (índice de 'e' en "Hola a todos.")

String s3 = "abcdef".substring(2, 5);    //s3 = "cde"
String s4 = s3.concat("f");              //s4 = "cdef"
String s5 = String.valueOf(i);           //s5 = "1" (valueOf() es static)
```

Consulte

- `java.lang.String` de la documentación API de JDK

La clase StringBuffer: java.lang.StringBuffer

La clase `StringBuffer` del paquete `java.lang`, al igual que la clase `String`, representa una secuencia de caracteres. A diferencia de una cadena, el contenido de un `StringBuffer` sí se puede modificar. Mediante diversos métodos de la clase `StringBuffer`, es posible cambiar la longitud y el contenido de sus objetos. Además, los objetos `StringBuffer` pueden aumentar su longitud cuando sea necesario. Por último, después de modificar un objeto `StringBuffer`, se puede crear una cadena que represente el contenido del `StringBuffer`.

La clase `StringBuffer` dispone de varios constructores que se muestran en la siguiente tabla.

Constructor	Argumento	Descripción
`StringBuffer`	`()`	Crea un objeto StringBuffer vacío que puede contener hasta 16 caracteres.
`StringBuffer`	`(int length)`	Crea un objeto StringBuffer que puede almacenar el número de caracteres especificado por el argumento `length`.
`StringBuffer`	`(String str)`	Crea un objeto StringBuffer que contiene una copia del objeto `String str`.

Existen varios métodos importantes que diferencian la clase `StringBuffer` de la clase `String`: `capacity()`, `setLength()`, `setCharAt()`, `append()`, `insert()` y `toString()`. Los métodos `append()` e `insert()` están sobrecargados para que puedan aceptar diversos tipos de datos.

Método	Argumento	Descripción
`setLength`	`(int newLength)`	Define la longitud del objeto `StringBuffer`.
`capacidad`	`()`	Devuelve la cantidad de memoria asignada al objeto `StringBuffer`.
`setCharAt`	`(int index, char ch)`	Asigna el valor `ch` al carácter que corresponde al índice especificado del objeto `StringBuffer`.
`append`	`(char c)`	Añade la representación de cadena del tipo de datos indicado por el argumento al objeto `StringBuffer`. Este método está sobrecargado para que pueda aceptar diversos tipos de datos.
`insert`	`(int offset, char c)`	Inserta la representación de cadena del tipo de datos del argumento en este objeto `StringBuffer`. Este método está sobrecargado para que pueda aceptar diversos tipos de datos.
`toString`	`()`	Convierte el objeto `StringBuffer` en un objeto `String`.

El método `capacity()`, que devuelve la cantidad de memoria asignada para el objeto `StringBuffer`, puede devolver un valor mayor que el método `length()`. La memoria asignada para un `StringBuffer` se puede definir mediante el constructor `StringBuffer(int length)`.

El siguiente código muestra algunos de los métodos asociados con la clase `StringBuffer`.

```
StringBuffer s1 = new StringBuffer(10);

int c = s1.capacity();        //c = 10
int len = s1.length();        //len = 0

s1.append("Bor");             //s1 = "Bor"
s1.append("land");            //s1 = "Borland"

c = s1.capacity();            //c = 10
len = s1.length();            //len = 7

s1.setLength(2);              //s1 = "Bo"

StringBuffer s2 = new StringBuffer("Hola a todos");
s2.insert(3, "l");            //s2 = "Hola a todos"
```

Consulte `java.lang.StringBuffer` de la documentación API del JDK

La clase System: java.lang.System

La clase `System` del paquete `java.lang` contiene varios métodos y campos de clase útiles para acceder a recursos e información del sistema independientes de la plataforma, así como copiar matrices, cargar archivos y bibliotecas y obtener y definir propiedades. Por ejemplo, el método `currentTimeMillis()` proporciona acceso a la hora actual del sistema. También es posible recuperar y cambiar recursos del sistema mediante los métodos `getProperty` y `setProperty`. Otra característica útil que proporciona la clase `System` es el método `gc()`, que produce una liberación completa de la memoria no utilizada; por último, la clase `System` permite a los desarrolladores cargar bibliotecas de vínculo dinámico (DLL) mediante el método `loadLibrary()`.

La clase `System` se declara como una clase `final`; por lo tanto, no se puede derivar en subclases. Asimismo, declara sus métodos y variables como `static`. De esta forma, están disponibles sin necesidad de instanciar la clase.

La clase `System` también declara algunas variables que se utilizan para interactuar con el sistema. Entre estas variables, se incluyen `in`, `out` y `err`. La variable `in` representa el flujo de entrada estándar del sistema, mientras que la variable `out` representa el flujo de salida estándar. La variable `err` es el flujo de salida de errores estándar. Los flujos de datos se describen con mayor detalle en la sección correspondiente al paquete E/S.

Método	Argumento	Descripción
`arrayCopy`	`arraycopy(Object src, int src_position, Object dst, int dst_position, int length)`	Copia la matriz fuente especificada desde la posición indicada en la posición también definida de la matriz de destino.
`currentTimeMillis`	`()`	Devuelve la hora actual en milisegundos.
`loadLibrary`	`(String libname)`	Carga la biblioteca del sistema especificada en el argumento.
`getProperty`	`(String key)`	Obtiene la propiedad del sistema indicada por la clave.
`gc`	`()`	Ejecuta el proceso de liberación de memoria no utilizada, el cual elimina los objetos que ya no se usan.
`load`	`(String filename)`	Carga un archivo de código desde el sistema de archivos local como biblioteca dinámica.

Método	Argumento	Descripción
exit	(int status)	Permite salir del programa actual.
setProperty	(String key, String value)	Define la propiedad de sistema especificada por el argumento key.

Consulte java.lang.System en la documentación API del JDK

Clases java.util principales

La interfaz Enumeration: java.util.Enumeration

La interfaz Enumeration del paquete java.util se utiliza para implementar una clase capaz de enumerar valores. Las clases que implementan la interfaz Enumeration pueden facilitar el recorrido de estructuras de datos.

Los métodos definidos en la interfaz Enumeration permiten al objeto Enumeration recuperar de forma continua todos los elementos de un conjunto de valores, uno por uno. Sólo existen dos métodos declarados en la interfaz Enumeration: hasMoreElements() y nextElement().

El método hasMoreElements() devuelve el valor true si aún quedan elementos en la estructura de datos. El método nextElement() se utiliza para devolver el siguiente valor de la estructura que se está enumerando.

En el siguiente ejemplo, se crea una clase denominada CanEnumerate, la cual implementa la interfaz Enumeration. Se utiliza una instancia de esa clase para mostrar en pantalla todos los elementos del objeto Vector, v.

```
CanEnumerate enum = v.elements();

while (enum.hasMoreElements()) {
  System.out.println(enum.nextElement());
}
```

Los objetos Enumeration presentan una limitación: sólo se pueden utilizar una vez. No existe ningún método definido en la interfaz que permita a un objeto Enumeration retroceder hacia elementos anteriores. Por ello, una vez que se enumera el conjunto completo de valores, el objeto se consume.

Consulte java.util.Enumeration de la documentación API del JDK

La clase Vector: java.util.Vector

Java no admite todas las estructuras de datos dinámicas, sólo define una clase Stack. No obstante, mediante la clase Vector del paquete java.util se pueden implementar fácilmente estructuras de datos dinámicas.

La clase Vector es muy eficiente, ya que asigna más memoria de la necesaria cuando se añaden elementos. Por lo tanto, la capacidad de un Vector suele ser mayor que su tamaño real. El argumento capacityIncrement del cuarto constructor, que se muestra en la tabla siguiente, define el incremento de capacidad de un Vector cuando se le añade un elemento.

Constructor	Argumento	Descripción
Vector	()	Permite construir un vector vacío con un tamaño de 10 elementos y un incremento de capacidad igual a cero.
Vector	(Collection c)	Permite construir un vector que contiene los elementos de la colección, en el orden en que son devueltos por el iterador de la colección.
Vector	(int initialCapacity)	Permite construir un vector vacío con la capacidad inicial especificada y un incremento de capacidad igual a cero.
Vector	(int initialCapacity, int capacityIncrement)	Permite construir un vector vacío con una capacidad inicial y un incremento de capacidad determinados.

La tabla siguiente muestra algunos de los métodos más importantes de la clase Vector y los argumentos que aceptan.

Método	Argumento	Descripción
setSize	(int newSize)	Define el tamaño de un vector.
capacity	()	Devuelve un valor que indica la capacidad de un vector.
size	()	Devuelve un valor que indica el número de elementos almacenados en un vector.
elements	()	Devuelve una enumeración de elementos de un vector.
elementAt	(int)	Devuelve el elemento situado en la posición especificada.
firstElement	()	Devuelve el primer elemento de un vector (correspondiente al índice 0).
lastElement	()	Devuelve el último elemento de un vector.
removeElementAt	(int index)	Elimina el elemento correspondiente al índice especificado.
addElement	(Object obj)	Añade el objeto especificado al final de un vector e incrementa en uno el tamaño del vector.
toString	()	Devuelve una representación de tipo cadena para cada elemento de un vector.

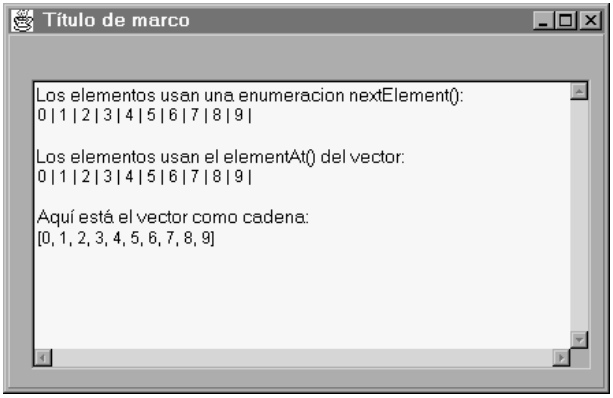
El siguiente código muestra el uso de la clase Vector. Se crea un objeto Vector, denominado vector1, y se enumeran sus elementos de tres maneras: mediante el método nextElement() de Enumeration, mediante el método

elementAt() de Vector y mediante el método toString() de Vector. Para mostrar el resultado, se crea un componente AWT, textArea. La propiedad text se define mediante el método setText().

```
Vector vector1 = new Vector();

for (int i = 0; i < 10;i++) {
  vector1.addElement(new Integer(i)); //addElement acepta objetos o tipos compuestos
}                                     //pero no tipos primitivos

//enumera vector1 usando nextElement()
Enumeration e = vector1.elements();
textArea1.setText("Los elementos usan una enumeración nextElement():\n");
while (e.hasMoreElements()) {
  textArea1.append(e.nextElement()+ " | ");
}
textArea1.append("\n\n");

//enumera mediante el método elementAt()
textArea1.append("Los elementos usan el elementAt() del vector:\n");
for (int i = 0; i < vector1.size();i++) {
  textArea1.append(vector1.elementAt(i) + " | ");
}
textArea1.append("\n\n");

//enumera mediante el método  toString()
textArea1.append("Aquí está el vector como cadena:\n");
textArea1.append(vector1.toString());
```

La siguiente figura muestra qué es lo que este código hubiera conseguido dentro de una aplicación.

Figura 5.1 Ejemplo de Vector y Enumeración



Consulte java.util.Vector en la documentación API del JDK

Clases java.io principales

Clases de flujo de entrada

Los flujos de entrada se utilizan para leer datos de una fuente de entrada, tales como un archivo, una cadena o la memoria. Ejemplos de clases de flujos de datos de entrada del paquete `java.io` son: `InputStream`, `BufferedInputStream`, `DataInputStream` y `FileInputStream`.

El método básico para leer datos mediante una clase de flujo de entrada es siempre el mismo:

1. Crear una instancia de una clase de flujo de entrada.
2. Indicar dónde han de leerse los datos.

Nota Las clases de flujo de entrada leen los datos como un flujo continuo de bytes. Si no hay datos disponibles en un momento dado, la clase de flujo de entrada se bloquea o espera hasta que haya datos disponibles.

Además de las clases de flujo de entrada, el paquete `java.io` proporciona clases lectoras (excepto para `DataInputStream`). Ejemplos de este tipo de clases son: `Reader`, `BufferedReader`, `FileReader` y `StringReader`. Las clases lectoras son idénticas a las clases de flujo de entrada, excepto que aquellas leen caracteres Unicode en vez de bytes.

Clase InputStream: java.io.InputStream

La clase `InputStream` del paquete `java.io` es una clase abstracta, y constituye la clase raíz o superclase de todas las demás clases de flujo de entrada. Proporciona la interfaz básica para leer un flujo de bytes. La tabla siguiente muestra algunos de los métodos definidos en la clase `InputStream` y los argumentos que aceptan. Todos estos métodos devuelven un valor de tipo `int`, excepto el método `close()`.

Método	Argumento	Descripción
`read`	`()`	Lee el siguiente byte del flujo de entrada y lo devuelve como un entero. Cuando alcanza el final del flujo de datos, devuelve -1.
`read`	`(byte b[])`	Lee múltiples bytes y los almacena en la matriz `b`. Devuelve el número de bytes leídos o -1 cuando se alcanza el final del flujo de datos.
`read`	`(byte b[], int off, int len)`	Lee hasta `len` bytes del flujo de datos de entrada, empezando en la posición indicada por el desplazamiento `off`, y los almacena en una matriz de bytes.

Método	Argumento	Descripción
available	()	Devuelve el número de bytes que se pueden leer de un flujo de entrada sin que se produzca un bloqueo por causa de una llamada a otro método que utiliza el mismo flujo de entrada.
skip	(long n)	Omite la lectura de n bytes de datos de un flujo de entrada y los descarta.
close	()	Cierra un flujo de entrada y libera los recursos del sistema utilizados por el flujo de datos.

Consulte java.io.InputStream en la documentación API del JDK

Clase FileInputStream: java.io.FileInputStream

La clase FileInputStream del paquete java.io es muy similar a la clase InputStream, sólo que está diseñada para leer archivos. Contiene tres constructores: FileInputStream(String filename), FileInputStream(File fileobject) y FileInputStream(FileDescriptor fdObj). El primer constructor toma el nombre del archivo como argumento, mientras que el segundo simplemente toma un objeto archivo. El tercer constructor toma un objeto descriptor de archivo. Las "Clases File" se tratan en la página 5-28.

Constructor	Argumento	Descripción
FileInputStream	(String filename)	Crea un objeto FileInputStream abriendo una conexión con el archivo especificado por la vía de acceso filename del sistema de archivos.
FileInputStream	(File fileobject)	Crea un objeto FileInputStream abriendo una conexión con el archivo especificado en el archivo fileobject del sistema de archivos.
FileInputStream	(FileDescriptor fdObj)	Crea un objeto FileInputStream mediante el descriptor de archivo fdObj, el cual representa una conexión existente con un archivo real del sistema de archivos.

El siguiente ejemplo muestra el uso de la clase FileInputStream.

```
import java.io.*;

class FileReader {
public static void main(String args[]) {
  byte buff[] = new byte[80];
            try {
    InputStream fileIn = new FileInputStream("Readme.txt");
    int i = fileIn.read(buff);
    String s = new String(buff);
    System.out.println(s);
```

```
    }
    catch(FileNotFoundException e) {
    }
    catch(IOException e) {
    }
   }
  }
```

En este ejemplo, se crea una matriz de caracteres que almacena los datos de entrada. A continuación, se crea una instancia de un objeto `FileInputStream`, y el nombre del archivo de entrada se pasa a su constructor. Después, se utiliza el método `FileInputStream read()` para leer un flujo de caracteres y almacenarlo en la matriz `buff`. Los primeros 80 bytes se leen del archivo `Readme.txt` y se almacenan en la matriz `buff`.

Nota También se podría utilizar la clase `FileReader` en lugar del método `FileInputStream()`. Los únicos cambios necesarios serían una matriz de tipo `char` en lugar de la matriz de tipo `byte`, y que el objeto `reader` se instanciaría del siguiente modo:

```
Reader fileIn = new FileReader("Readme.txt");
```

Por último, para poder ver el resultado de la llamada a `read`, se crea un objeto `String` a partir de la matriz `buff` y, a continuación, se pasa al método `System.out.println()`.

Como ya se mencionó anteriormente, la clase `System`, definida en `java.lang`, proporciona acceso a recursos del sistema. `System.out`, un miembro `static` de `System`, representa el dispositivo de salida estándar. El método `println()` se utiliza para enviar el resultado al dispositivo de salida estándar. El objeto `System.out` object es del tipo `PrintStream`, que se trata en el apartado siguiente.

El objeto `System.in`, otro miembro `static` de la clase `System`, es de tipo `InputStream` y representa el dispositivo de entrada estándar.

Consulte `java.io.FileInputStream` en la documentación API de JDK

Clases de flujo de salida

Las clases de flujo de salida son las homólogas de las clases de flujo de entrada. Se utilizan para enviar flujos de datos a diversos dispositivos de salida. Las clases de flujo de salida principales, ubicadas en el paquete `java.io`, son: `OutputStream`, `PrintStream`, `BufferedOutputStream`, `DataOutputStream` y `FileOutputStream`.

Clase OutputStream: java.io.OutputStream

Para dar salida a un flujo de datos, se crea un objeto `OutputStream` y se le indica que dirija los datos a una fuente de salida particular. Como era de esperar, también existen las clases "writer" (de escritura) correspondientes

para cada clase, excepto para la clase `DataOutputStream`. Algunos de los métodos que incluye la clase `OutputStream` son:

Método	Argumento	Descripción
`write`	`(int b)`	Escribe `b` en un flujo de datos de salida.
`write`	`(byte b[])`	Escribe la matriz `b` en un flujo de datos de salida.
`write`	`(byte b[], int off, int len)`	Escribe `len` bytes de la matriz de bytes, empezando desde la posición indicada por el desplazamiento `off`, en el flujo de salida.
`flush`	`()`	Vacía el flujo de datos y fuerza la salida de cualquier dato almacenado en el búfer.
`close`	`()`	Cierra el flujo de datos de salida y libera cualquier recurso del sistema asociado con él.

Consulte `java.io.OutputStream` en la documentación API de JDK

Clase PrintStream: java.io.PrintStream

La clase `PrintStream` del paquete `java.io`, diseñada principalmente para escribir datos de texto, dispone de dos constructores. El primer constructor vacía los datos almacenados en el búfer según condiciones especificadas, mientras que el segundo vacía los datos cuando encuentra un carácter de retorno de carro (si `autoflush` tiene asignado el valor `true`).

Constructor	Argumento	Descripción
`PrintStream`	`(OutputStream out)`	Crea un flujo de datos de impresión.
`PrintStream`	`(OutputStream out, boolean autoflush)`	Crea un flujo de datos de impresión.

Algunos de los métodos definidos en la clase `PrintStream` se muestran en la siguiente tabla.

Método	Argumento	Descripción
`checkError`	`()`	Vacía el flujo de datos y devuelve un valor false si se detecta un error.
`print`	`(Object obj)`	Imprime un objeto.
`print`	`(String s)`	Imprime un objeto.

Método	Argumento	Descripción
println	()	Imprime y termina la línea utilizando la cadena de separación de línea que se define mediante la propiedad line.separator, la cual no es necesariamente un simple carácter de retorno de carro('\n').
println	(Object obj)	Imprime un objeto y termina la línea. Este método se comporta como si se invocara primero a print(Object) y después a println().

Los métodos print() y println() están sobrecargados para poder recibir diferentes tipos de datos.

Consulte java.io.PrintStream en la documentación API de JDK

Clase BufferedOutputStream: java.io.BufferedOutputStream

La clase BufferedOutputStream del paquete java.io implementa un flujo de salida con almacenamiento intermedio que incrementa la eficiencia almacenando valores en un búfer y escribiéndolos sólo cuando el búfer está lleno o se realiza una llamada al método flush().

Constructor	Argumento	Descripción
BufferedOutputStream	(OutputStream out)	Crea un flujo de salida para escritura de datos con almacenamiento intermedio en búfer de 512 bytes.
BufferedOutputStream	(OutputStream out, int size)	Crea un flujo de salida para escritura de datos con almacenamiento intermedio en un búfer del tamaño especificado.

BufferedOutputStream dispone de tres métodos para vaciar el flujo de salida y escribir en él.

Método	Argumento	Descripción
flush	()	Permite vaciar el flujo de salida.
write	(byte[] b, int off, int len)	Escribe len bytes de la matriz de bytes, empezando en la posición dada por el desplazamiento off, en el flujo de salida con búfer.
write	(int b)	Escribe el byte en el flujo de salida con búfer.

Consulte java.io.BufferedOutputStream en la documentación API del JDK

Clase DataOutputStream: java.io.DataOutputStream

Un flujo de salida de datos permite a una aplicación escribir tipos de datos primitivos de Java mediante un formato binario portable. Una aplicación puede utilizar entonces un flujo de entrada de datos para leer de nuevo los datos.

La clase `DataOutputStream` del paquete java.io dispone de un único constructor, `DataOutputStream(OutputStream out)`, que crea un flujo de datos de salida para escribir datos.

La clase `DataOutputStream` utiliza varios métodos `write()` para escribir tipos de datos primitivos, así como un método `flush()` y un método `size()`.

Método	Argumento	Descripción
`flush`	`()`	Permite vaciar el flujo de salida de datos.
`size`	`()`	Devuelve el número de bytes escritos en el flujo de salida de datos.
`write`	`(int b)`	Escribe el byte en el flujo de salida.
`writeType`	`(type v)`	Escribe el tipo primitivo como secuencia de bytes en el flujo de salida.

Consulte `java.io.DataOutputStream` en la documentación API del JDK

Clase FileOutputStream: java.io.FileOutputStream

Un flujo de salida de tipo archivo es un flujo de salida indicado para escribir datos en un archivo o en un FileDescriptor. La disponibilidad o la posibilidad de creación de un archivo depende de la plataforma subyacente. En algunas plataformas, `FileOutputStream` sólo puede abrir un archivo para escritura a la vez. En esos casos, el constructor de esta clase fallaría si el archivo implicado está aún abierto. `FileOutputStream`, del paquete `java.io`, una subclase de `OutputStream`, dispone de varios constructores.

Constructor	Argumento	Descripción
`FileOutputStream`	`(File file)`	Crea un flujo de salida para escribir en el archivo especificado.
`FileOutputStream`	`(FileDescriptor fdObj)`	Crea un flujo de salida para escribir en el descriptor de archivo, el cual representa una conexión existente con un archivo real del sistema de archivos.
`FileOutputStream`	`(String name)`	Crea un flujo de salida para escribir en el archivo con el nombre especificado.
`FileOutputStream`	`(String name, boolean append)`	Crea un flujo de salida para escribir en el archivo con el nombre especificado.

FileOutputStream dispone de varios métodos, incluidos `close()` y `finalize()`, y varios métodos `write()`.

Método	Argumento	Descripción
`close`	`()`	Cierra el flujo de salida para archivos y libera cualquier recurso del sistema asociado.
`finalize`	`()`	Despeja la conexión con el archivo y llama al método `close()` cuando ya no existen más referencias al flujo de datos.
`getFD`	`()`	Devuelve el descriptor de archivo asociado con el flujo de datos.
`write`	`(byte[] b, int off, int len)`	Escribe `len` bytes de datos de la matriz de bytes en el flujo de salida de archivos, empezando desde la posición indicada por el desplazamiento off.

Consulte `java.io.FileOutputStream` en la documentación API del JDK

Clases File

Las clases `FileInputStream` y `FileOutputStream` del paquete `java.io` sólo proporcionan funciones básicas para el manejo de operaciones de entrada y salida de archivos. El paquete `java.io` ofrece la clase `File` y la clase `RandomAccessFile` para obtener un mayor control sobre los archivos. La clase `File` ofrece un cómodo acceso a funciones y atributos de archivos, mientras que la clase `RandomAccessFile` ofrece diversos métodos para la lectura y escritura de archivos.

Clase File: java.io.File

La clase `File` de Java utiliza una representación de los archivos y vías de acceso abstracta e independiente de la plataforma. La clase `File` dispone de tres constructores, los cuales se muestran en la tabla siguiente.

Constructor	Argumento	Descripción
`File`	`(String path)`	Crea una instancia de la clase `File` convirtiendo la cadena del nombre de la vía de acceso dada en una vía de acceso abstracta.

Constructor	Argumento	Descripción
File	(String parent, String child)	Crea una instancia de la clase File a partir de una cadena de vía de acceso principal y una cadena de vía de acceso que es su descendiente.
File	(File parent, String child)	Crea una instancia de la clase File a partir de dos cadenas de vías de acceso abstractas: una principal y otra descendiente.

La clase File también implementa muchos métodos importantes que comprueban la existencia, legibilidad, capacidad de escritura, tipo, tamaño y hora de modificación de los archivos y directorios, así como métodos que permiten crear directorios y renombrar y borrar archivos y directorios.

Método	Argumento	Devuelve	Descripción
delete	()	boolean	Borra archivos o directorios.
canRead	()	boolean	Prueba si la aplicación puede leer el archivo especificado por la vía de acceso abstracta.
canWrite	()	boolean	Prueba si la aplicación puede escribir en el archivo.
renameTo	(File dest)	boolean	Permite cambiar el nombre del archivo.
getName	()	String	Devuelve una cadena con el nombre del archivo o directorio.
getParent	()	String	Devuelve una cadena con la vía de acceso del directorio principal del archivo o directorio.
getPath	()	String	Convierte la vía de acceso abstracta en una cadena de vía de acceso.

Consulte java.io.File en la documentación API del JDK

Clase RandomAccessFile: java.io.RandomAccessFile

La clase RandomAccessFile del paquete java.io es más versátil que las clases FileInputStream y FileOutputStream, ya que éstas sólo proporcionan acceso secuencial a un archivo. La clase RandomAccessFile permite leer y escribir de forma arbitraria bytes, texto y tipos de datos de Java en cualquier posición especificada de un archivo. Dispone de dos constructores: RandomAccessFile(String name, String mode) y RandomAccessFile(File file,

String mode). El argumento mode indica si el objeto RandomAccessFile se utiliza para lectura (“r”) o para lectura/escritura(“rw”).

Constructor	Argumento	Descripción
RandomAccessFile	(String name, String mode)	Crea un flujo de datos para archivos de acceso aleatorio con el fin de leer, y opcionalmente escribir, en un archivo con el nombre especificado.
RandomAccessFile	(File file, String mode)	Crea un flujo de datos para archivos de acceso aleatorio con el fin de leer, y opcionalmente escribir, en el archivo especificado por el argumento File.

RandomAccessFile implementa una gran variedad de potentes métodos, como se muestra en la tabla siguiente.

Método	Argumento	Descripción
seek	(long pos)	Define el desplazamiento del puntero de archivo, medido desde el inicio de éste, donde se va a producir la próxima operación de lectura o escritura.
read	()	Lee el siguiente byte de datos del flujo de entrada.
read	(byte b[], int off, int len)	Lee hasta len bytes del flujo de datos de entrada, empezando en la posición indicada por el desplazamiento off, y los almacena en una matriz de bytes.
readType	()	Lee, de un archivo, el tipo de datos especificado, tal como lo haría readChar, readByte o readLong.
write	(int b)	Escribe el byte especificado en el archivo.
write	(byte b[], int off, int len)	Escribe len bytes de la matriz de bytes, empezando desde la posición indicada por el desplazamiento off, en el flujo de salida.
length	()	Devuelve la longitud del archivo.
close	()	Cierra el archivo y libera cualquier recurso asociado del sistema.

Consulte java.io.RandomAccessFile en la documentación API de JDK

Clase StreamTokenizer: java.io.StreamTokenizer

La clase StreamTokenizer del paquete java.io se utiliza para leer un flujo de entrada y descomponerlo en tokens individuales que se pueden procesar de uno en uno. Los tokens son grupos de caracteres que representan un

número o una palabra. El objeto encargado de descomponer un flujo en tokens puede reconocer cadenas, números, identificadores y comentarios. Esta técnica de partición de flujos de datos en tokens se utiliza habitualmente en la escritura de analizadores sintácticos, compiladores o programas que procesan una entrada de caracteres.

Esta clase dispone de un constructor, `StreamTokenizer(Reader r)`, y define las siguientes cuatro constantes.

Constante	Descripción
`TT_EOF`	Indica que se ha llegado al final del archivo.
`TT_EOL`	Indica que se ha llegado al final de la línea.
`TT_NUMBER`	Indica que el token leído es un número.
`TT_WORD`	Indica que el token leído es una palabra.

La clase `StreamTokenizer` utiliza las variables de instancia `nval`, `sval` y `ttype` para albergar el valor de un número, el valor de una palabra y el tipo del token, respectivamente.

La clase `StreamTokenizer` implementa varios métodos que se utilizan para definir la sintaxis léxica de los tokens.

Método	Argumento	Descripción
`nextToken`	`()`	Analiza el siguiente token del flujo de entrada de datos. Devuelve `TT_NUMBER` si el siguiente token es un número y `TT_WORD` si es una palabra o un carácter.
`parseNumbers`	`()`	Analiza los números.
`lineno`	`()`	Devuelve el número de la línea actual.
`pushBack`	`()`	Devuelve el valor actual del campo `ttype` en la próxima llamada al método `nextToken()`.
`toString`	`()`	Devuelve la cadena equivalente del token actual.

Siga estos pasos cuando desee utilizar un objeto StreamTokenizer:

1. Cree un objeto `StreamTokenizer` para un objeto `Reader`.
2. Defina cómo procesar los caracteres.
3. Utilice el método `nextToken()` para obtener el siguiente token.
4. Lea la variable de instancia `ttype` para determinar el tipo del token.
5. Lea el valor del token en la variable de instancia.
6. Procese el token.
7. Repita los pasos 3 a 6 anteriores hasta que `nextToken()` devuelva el valor `StreamTokenizer.TT_EOF`.

Consulte `java.io.StreamTokenizer` en la documentación API de JDK.

Capítulo

6

Programación orientada a objetos en Java

La programación orientada a objetos apareció con la introducción del lenguaje Simula 67 in 1967. No obstante, fue a mediados de los 80 cuando pasó a la vanguardia de los paradigmas de la programación.

Al contrario que la programación estructurada tradicional, la programación orientada a objetos sitúa los datos y las operaciones relacionadas con ellos en una única estructura de datos. En la programación estructurada, los datos y las operaciones sobre ellos están separadas, y las estructuras de datos se envían a los procesos y funciones en las que van a ser manejados. La programación orientada a objetos resuelve muchos de los problemas inherentes a este diseño, porque los atributos y las operaciones forman parte de la misma entidad. Este diseño imita más de cerca al mundo real, en el que todos los objetos poseen tanto atributos como actividades asociadas con ellos.

Java es un lenguaje orientado a objetos *puro*, lo que significa que el nivel más externo de la estructura de datos en Java es el *objeto*. No hay funciones, variables o constantes autónomas en Java. Se accede a todo mediante clases y objetos. Esta es una de las mejores características de Java. Otros lenguajes híbridos orientados a objetos tienen aspectos de lenguajes estructurados además de las extensiones de objetos. Por ejemplo, C++ y Object Pascal son lenguajes orientados a objetos, pero aún se pueden escribir construcciones de programación estructurada, lo que disminuye la efectividad de las extensiones orientadas a objetos. ¡Esto no puede hacerlo en Java!

Este capítulo asume que el lector ya tiene un cierto conocimiento sobre programación en otros lenguajes orientados a objetos. Si no es así, debe dirigirse a otras fuentes para conseguir una explicación más profunda de la programación orientada a objetos. Este capítulo intenta destacar y resumir las características de orientación a objetos de Java.

Clases

Clases y objetos no son lo mismo. Una clase es una definición de tipo, mientras que un objeto es una declaración de una instancia de un tipo de clase. Una vez que se crea una clase, se pueden crear tantos objetos basados en esa clase como se desee. Existe la misma relación entre clases y objetos que entre recetas de pastel de cerezas y pasteles de cerezas; se pueden hacer tantos pasteles como se desee a partir de una sola receta.

El proceso de crear un objeto a partir de una clase se denomina *instanciar* un objeto o crear una *instancia* de una clase.

Declaración e instanciación de clases

Una clase en Java puede ser muy simple. He aquí una definición de clase para una clase vacía:

```
class MyClass {
}
```

Aunque esta clase aún no es útil, es legítima en Java. Una clase más útil contendría algunos miembros de datos y métodos, que añadirá pronto. Para empezar, examine la sintaxis para instanciar una clase. Para crear una instancia de esta clase, utilice el operador `new` en conjunción con el nombre de la clase. Debe declarar una variable de instancia para el objeto:

```
MyClass myObject;
```

No obstante, el mero hecho de declarar una variable de instancia no asigna memoria y otros recursos para el objeto. Crea una referencia llamada `myObject`, pero no instancia el objeto. Es el operador `new` quien realiza esta tarea.

```
myObject = new MyClass();
```

Observe que el nombre de la clase se utiliza aquí como si fuera un método. Esto no es un coincidencia, como verá en la sección siguiente. Una vez que esta línea de código se ha ejecutado, las variables miembro y los métodos de la clase, que no existen todavía, pueden ser accedidos utilizando el operador ".".

Una vez que ha creado el objeto, nunca debe preocuparse de destruirlo. Los objetos en Java se tiran a la basura automáticamente, lo que significa

que cuando una referencia de objeto ya no se utiliza, la máquina virtual libera automáticamente cualquier recurso asignado por el operador `new`.

Miembros de datos

Como se indicó anteriormente, una clase en Java puede contener tanto miembros de datos como métodos. Un miembro de datos o variable miembro es una variable declarada dentro de una clase. Un método es una función o rutina que realiza alguna tarea. He aquí una clase que contiene sólo miembros de datos:

```
public class DogClass {
  String name, eyeColor;
  int age;
  boolean hasTail;
}
```

Este ejemplo crea una clase llamada `DogClass` que contiene miembros de datos: `name` (nombre), `eyeColor` (color de ojos), `age` (edad), y un flag llamado `hasTail` (tiene cola). Puede incluir cualquier tipo de datos como variable miembro de una clase. Para acceder a un miembro de datos, debe crear primero una instancia de la clase, y después acceder a los datos utilizando el operador ".".

Métodos de clase

También puede incluir métodos en clases. De hecho, no hay funciones o procedimientos autónomos en Java. Todas las subrutinas se definen como métodos de clases. He aquí un ejemplo de `DogClass` con un método `speak()` añadido:

```
public class DogClass {
  String name,eyeColor;
  int age;
  boolean hasTail;

  public void speak() {
    JOptionPane.showMessageDialog(null, "Guau! Guau!");
  }
}
```

Observe que, cuando define métodos, la implementación para el método aparece directamente debajo de la declaración. Esto es un caso distinto que en algunos otros lenguajes orientados a objetos, en los que la clase se define en un sitio y el código de implementación aparece en alguna otra parte. Un método debe especificar un tipo devuelto y todos los parámetros que haya recibido. El método `speak()` no tiene parámetros. Tampoco devuelve un valor, luego su tipo devuelto es `void`.

Para llamar al método, acceda a él igual que accede a las variables miembro, es decir, utilizando el operador ".". Por ejemplo:

```
DogClass dog = new DogClass();
dog.age = 4;
dog.speak();
```

Constructores y terminadores

Toda clase Java tiene un método de propósito especial llamado *constructor*. El constructor siempre tiene el mismo nombre que la clase, y no puede especificar un valor de retorno. El constructor asigna todos los recursos que necesita el objeto y devuelve una instancia del mismo. Cuando utiliza el operador `new`, en realidad está llamando al constructor. No necesita especificar un tipo devuelto para el constructor porque el tipo devuelto es la propia instancia del objeto.

La mayoría de los lenguajes orientados a objetos tienen un método correspondiente llamado *destructor*, al que se llama para liberar todos los recursos que el constructor asignó. Sin embargo, y puesto que Java libera automáticamente todos los recursos, no cuenta con un mecanismo destructor.

Hay situaciones, sin embargo, que requieren realizar algún trabajo de limpieza especial que el liberador de memoria no puede realizar cuando la clase desaparece. Por ejemplo, su programa puede haber abierto algunos archivos durante la vida del objeto y quiere asegurarse de que se cierran adecuadamente cuando el objeto se destruye. Hay otro método de propósito especial que se puede definir para una clase, llamado *terminador*. Este método (si está presente) es llamado por el liberador de memoria justo antes de que el objeto se destruya. Por lo tanto, si hay alguna limpieza especial que deba realizarse, el terminador puede encargarse de ella. El recogedor de basura se ejecuta como un hilo de baja prioridad en la máquina virtual. Sin embargo, no se puede predecir cuando va a destruir realmente su objeto. Por eso, no se debería incluir código dependiente del tiempo en el terminador, porque no puede predecir cuándo será llamado.

Estudio de casos: Un ejemplo sencillo de OOP

En este apartado, verá un ejemplo sencillo de definición de clases e instanciación de objetos. Desarrollará una aplicación que crea dos objetos (un perro y un hombre) y muestra sus atributos en un formulario.

Si no tiene experiencia con JBuilder, debería dejar este capítulo de lado y aprender sobre el entorno de desarrollo integrado de JBuilder antes de comenzar con este ejemplo. Comience con *Introducción a JBuilder*. Dentro de este manual, especialmente el tutorial "Creación de una aplicación" y

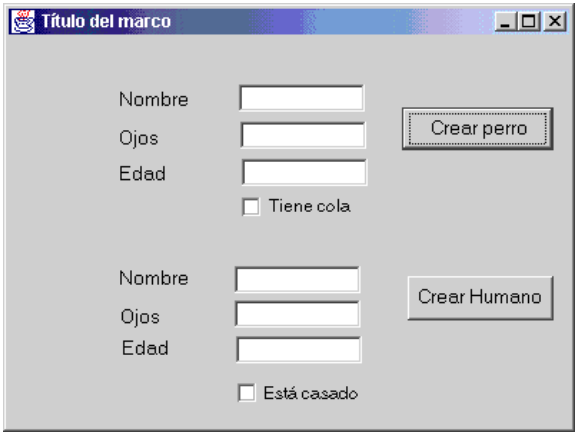
los siguientes capítulos que introducen el entorno de desarrollo integrado de JBuilder. Consulte también los primeros capítulos de *Diseño de aplicaciones con JBuilder* para saber más acerca el diseño de interfaces de usuario.

Estará listo para continuar con este capítulo cuando realice con soltura estas tareas:

- Comenzar una aplicación utilizando el Asistente para aplicaciones de JBuilder.
- Seleccionar componentes de la paleta de componentes, y colocarlos en el diseñador de interfaces.
- Definición de propiedades de componentes utilizando el Inspector.
- Alternar entre el editor y el diseñador de interfaces en el panel de contenido de JBuilder.
- El Editor.

Éste es el aspecto que tendrá, cuando se ejecute, la aplicación ejemplo que va a construir:

Figura 6.1 Aplicación de ejemplo mostrando dos objetos instanciados



Siga los pasos indicados en esta sección para crear una sencilla interfaz para esta aplicación de ejemplo.

1 Comience creando la aplicación y diseñando su interfaz:

a Crea un proyecto. Elija Archivo | Nuevo proyecto para iniciar el asistente para proyectos.

b Introduzca `oop1` en el campo Nombre del Proyecto y pulse Finalizar.

Se abre un nuevo proyecto.

c Elija Archivo | New, pulse la pestaña General y después en el icono Aplicación para iniciar el Asistente para aplicaciones.

d Acepte el nombre de la clase por defecto. El nombre del paquete será `oop1`.

e Haga clic en Siguiente y a continuación en Finalizar para crear un `Marco1.java` y un archivo `Aplicación1.java`.

f Pulse en la pestaña Diseño del panel de contenido para abrir el diseñador de interfaces para `Frame1.java`.

g Seleccione `contentPane` en el panel de estructura. En el Inspector asigne a la propiedad `layout` de `contentPane` el valor `XYLayout` (si es usuario de la edición Personal, asigne `null` a `layout`). `XYLayout` y `null` rara vez son ideales para una aplicación pero, hasta que aprenda el uso de diseños, puede utilizarlas para crear rápidamente una interfaz de andar por casa.

2 Coloque los componentes necesarios en el diseñador de interfaces, utilizando como referencia la imagen anterior:

a Pulse la pestaña Swing en la paleta de componentes y seleccione el componente `JTextField`. (Cuando posicione el cursor sobre un componente, aparece una ayuda inmediata etiquetándolo. Pulse el componente etiquetado `javax.swing.JTextField`.) Pulse en el diseñador de interfaces y mantenga pulsado el botón del ratón mientras dibuja el componente en la pantalla. Repita este paso cinco veces más hasta que tenga dos grupos de tres componentes `JTextField` en su formulario.

b Mantenga pulsada Mayúsculas mientras pulsa sobre cada `JTextField` del diseñador, de forma que todos ellos queden seleccionados. Seleccione la propiedad `text` en el Inspector y elimine el texto que aparece. Esto eliminará todo el texto en todos los componentes `JTextField`.

c Cambie el valor de la propiedad `name` de cada `JTextField`. Llame al primero `txtfldDogName` (nombre de perro), al segundo `txtfldDogEyeColor` (color de ojos del perro), al tercero `txtfldDogAge` (edad del perro), al cuarto `txtfldManName` (nombre del hombre), al quinto `txtfldManEyeColor` (color de ojos del hombre), y al sexto `txtfldManAge` (edad del hombre).

d Trace seis componentes `JLabel` en el formulario, cada uno de ellos adyacente a un componente `JTextField`.

e Cambie los valores de la propiedad `text` de estos componentes para etiquetar adecuadamente cada `JTextField`. Por ejemplo, la propiedad `text` del `JLabel` en la parte superior del formulario debería ser `Nombre`, la segunda `Ojos`, y así sucesivamente.

f Coloque dos componentes `JCheckBox` en el formulario. Coloque el primero bajo el primer grupo de tres componentes `JTextField`, y el segundo bajo el segundo grupo.

g Seleccione sucesivamente cada componente `JCheckBox` y escriba en la propiedad `text` del primero `Tiene Cola`, y en la del segundo, `Está Casado`.

h Cambie los valores de la propiedad `name` de las casillas de selección a `chkboxDog` para el primero y a `chkboxMan` para el segundo.

i Coloque dos componentes `JButton` en el formulario, uno a la derecha del grupo superior de componentes y otro a la derecha del inferior.

j Cambie la propiedad `text` del primer botón a `Crear Perro`, y la del segundo a `Crear Humano`.

El paso final es guardar el proyecto eligiendo Archivo | Guardar todo.

Ahora está listo para empezar a programar. Primero cree una nueva clase:

1. Elija Archivo | Nuevo proyecto para iniciar el asistente para proyectos.
2. Conserve el nombre del paquete como `oop1`, especifique el Nombre de Clase como `DogClass`, y no cambie la Clase base.
3. Seleccione sólo las opciones Pública y Generar constructor por defecto, desmarcando todas las demás.
4. Pulse Aceptar.

El asistente de Clase crea el archivo `DogClass.java` automáticamente. Modifique el código que creado automáticamente para que quede como sigue:

```
package oop1;

public class DogClass {
  String name, eyeColor;
  int age;
  boolean hasTail;

  public DogClass() {
    name = "Snoopy";
    eyeColor = "Marrón";
    age = 2;
    hasTail = true;
  }
}
```

Ha definido `DogClass` con algunas variables miembro. También hay un constructor para instanciar objetos `DogClass`.

Utilizando el asistente de Clase, cree un archivo `ManClass.java` siguiendo los mismos pasos excepto cambiando el Nombre de Clase por `ManClass`. Modifique el código resultante para que quede así:

```
package oop1;

public class ManClass {
  String name, eyeColor;
  int age;
  boolean isMarried;

  public ManClass() {
    name = "Steven";
    eyeColor = "Azul";
    age = 35;
    isMarried = true;
  }
}
```

Las dos clases son muy similares. Aprovechará las ventajas de este parecido en un próximo apartado.

Pulse la pestaña Marco1 en la parte superior del panel de contenido para volver a la clase `Marco1`. Pulse la pestaña Fuente en la parte inferior para abrir el editor. Declare dos variables de instancia como referencias a los objetos. He aquí el listado fuente de las declaraciones de variables de `Marco1` mostradas en negrita; añada las líneas en negrita a su clase:

```
public class Marco1 extends JFrame {
// Cree una referencia para los objetos perro y hombre
  DogClass dog;
  ManClass man;

  JPanel contentPane;
  JPanel jPanel1 = new JPanel();
  . . .
```

Pulse en la pestaña Diseño en la parte inferior del panel de contenido para volver a la interfaz que diseñó. Haga doble clic en el botón Crear Perro. JBuilder crea el principio de un manejador de sucesos para ese botón y posiciona el cursor en el código del manejador de sucesos. Rellene el código del manejados de sucesos de forma que instancie un objeto Dog y rellene los campos de texto correspondientes. Su código debería tener este aspecto:

```
void jButton1_actionPerformed(ActionEvent e) {
  dog = new DogClass();
  txtfldDogName.setText(dog.name);
  txtfldDogEyeColor.setText(dog.eyeColor);
  txtfldDogAge.setText(Integer.toString(dog.age));
  chkboxDog.setSelected(true);
}
```

Como muestra el código, está llamando al constructor para el objeto Dog y, a continuación, accediendo a sus variables miembro. Pulse sobre la pestaña Diseño para volver al diseñador de interfaces de usuario. Haga doble clic en el botón Crear Humano. JBuilder crea un manejador de sucesos para el botón Crear Humano. Rellene el manejador de sucesos de la siguiente forma:

```
void jButton2_actionPerformed(ActionEvent e) {
  man = new ManClass();
  txtfldManName.setText(man.name);
  txtfldManEyeColor.setText(man.eyeColor);
  txtfldManAge.setText(Integer.toString(man.age));
  chkboxMan.setSelected(true);
}
```

En este punto, la aplicación se puede compilar y ejecutar. Elija Proyecto | Crear proyecto "oop1.jpx" para compilarlo. Si no se han producido errores, seleccione Ejecutar | Ejecutar Proyecto.
Si todo va bien, el formulario aparecerá en su pantalla. Cuando pulse el botón Create Dog, se crea un objeto `dog` y los valores correspondientes aparecen en los campos del perro. Cuando pulse el botón Crear Humano, se crea un objeto `man` y los valores correspondientes aparecen en los campos del hombre.

Herencia de clases

Los objetos hombre y perro que ha creado tienen muchas similitudes. Uno de los beneficios de la programación orientada a objetos es la capacidad de manejar similitudes como ésta dentro de una jerarquía. Esta capacidad es conocida como *herencia*. Cuando una clase hereda de otra, la clase descendiente hereda automáticamente todas las características (variables miembro) y comportamiento (métodos) de la superclase. La herencia es siempre aditiva; no hay manera de heredar de una clase y obtener menos de lo que tiene la superclase.

La herencia en Java se maneja mediante la palabra clave `extends`. Cuando una clase hereda de otra, la clase descendiente deriva (extend) de la superclase. Por ejemplo:

```
public class DogClass extends MammalClass {
. . .
}
```

Los elementos que el hombre y el perro tienen en común se puede decir que son comunes a todos los mamíferos; por lo tanto, puede crear una clase `MammalClass` para manejar estas similitudes. Puede entonces eliminar las declaraciones de los elementos comunes de `DogClass` y de `ManClass`, declararlos en `MammalClass` en su lugar, y hacer a `DogClass` y a `ManClass` subclases de `MammalClass`.

Mediante el asistente de Clases, cree una clase `MammalClass`. Rellene el código resultante de la siguiente forma:

```
package oop1;

public class MammalClass {
  String name, eyeColor;
  int age;

  public MammalClass() {
    name = "El nombre";
    eyeColor = "Marrón";
    age = 0;
  }
}
```

Observe que `MammalClass` tiene características comunes tanto de `DogClass` como de `ManClass`. Ahora, reescriba `DogClass` y `ManClass` sacando partido a las ventajas de la herencia.
Modifique el código de `DogClass` de la siguiente forma:

```
package oop1;

public class DogClass extends MammalClass {

  boolean hasTail;

  public DogClass() {
    // implied super()
    name = "Snoopy";
    age = 2;
    hasTail = true;
  }
}
```

Modifique el código de `ManClass` para que quede así:

```
package oop1;

public class ManClass extends MammalClass{

  boolean isMarried;

  public ManClass() {
    name = "Steven";
    eyeColor = "Azul";
    age = 35;
    isMarried = true;
  }
}
```

`DogClass` no asigna específicamente un valor a `eyeColor`, pero sí lo hace `ManClass`. `DogClass` no asigna un valor a `eyeColor` porque el perro Snoopy tiene ojos marrones, y `DogClass` hereda ojos marrones de `MammalClass`, que declara una variable `eyeColor` y asigna el valor “Marrón”. El hombre

Steven, sin embargo, tiene ojos azules, por lo que es necesario asignar el valor "Azul" a la variable `eyeColor` heredada de `MammalClass`.

Intente compilar y ejecutar el proyecto de nuevo. (Elegir Ejecutar | Ejecutar el proyecto compilará y ejecutará su aplicación.) Como verá, la interfaz de usuario de su programa tiene el mismo aspecto de antes, pero ahora los objetos `dog` y `man` heredan todas las variables miembro comunes de `MammalClass`.

En cuanto que `DogClass` deriva de `MammalClass`, `DogClass` tiene todas las variables miembro y métodos que tiene `MammalClass`. De hecho, incluso `MammalClass` es heredada de otra clase. Todas las clases en Java derivan finalmente de la clase `Object`; por lo tanto, si una clase no se declara derivada de ninguna otra, implícitamente deriva de la clase `Object`.

Las clases en Java pueden heredar sólo de una clase a la vez (*herencia simple*). Al contrario que Java, algunos lenguajes (como C++) permiten que una clase herede de varias clases a la vez (*herencia múltiple*). Una clase deriva sólo de una clase a la vez. Aunque no hay restricciones al número de veces que puede utilizar la herencia para extender la jerarquía, debe hacer sólo una extensión a la vez. La herencia múltiple es una buena prestación, pero conduce a jerarquías de objetos muy complejas. Java tiene un mecanismo que proporciona muchos de esos beneficios pero sin tanta complejidad, como verá más tarde.

`MammalClass` tiene un constructor que define valores por defecto muy prácticos y convenientes. Sería muy útil que las subclases pudieran acceder a este constructor.

De hecho, pueden. Esto puede hacerse en Java de dos formas distintas. Si no se llama explícitamente al constructor del antecesor, **Java llama automáticamente al constructor por defecto del antecesor como la primera línea del constructor del descendiente**. La única forma de evitar esto es llamar uno mismo a uno de los constructores del antecesor como primera línea del constructor de la clase descendiente. Las llamadas a constructores siempre están encadenadas así, y es un mecanismo que no puede ser evitado. Esta es una buena característica del lenguaje Java porque, en otros lenguajes orientados a objetos, la falta de la llamada al constructor del antecesor es un error habitual. Java siempre lo hará si el programador no lo hace. Este es el significado del comentario en la primera línea del constructor de DogClass (`// implied super()`). Se llama automáticamente al constructor `MammalClass` en ese punto. Este mecanismo se basa en la existencia de un constructor de una superclase que no necesita parámetros. Si el constructor no existe y no se llama a uno de los otros constructores como primera línea del constructor del descendiente, la clase no se compilará.

Llamada al constructor del antecesor

Puesto que, con frecuencia, se querrá llamar explícitamente al constructor de la superclase, hay una palabra clave en Java que lo facilita. `super()` llamará al constructor del antecesor que tenga los parámetros suministrados adecuados.

También es posible tener más de un constructor en una clase. Cuando existe más de un método con el mismo nombre en una clase, se dice que los métodos están `sobrecargados`. Es corriente que una clase tenga múltiples constructores.

Para una misma aplicación, el cambio en la jerarquía es la única diferencia entre las dos primeras versiones del ejemplo. La instanciación de los objetos y el formulario principal no han cambiado en absoluto. Sin embargo, el diseño de la aplicación es más eficiente, porque si ahora necesita cambiar alguna de las características de los mamíferos, puede hacerlo en `MammalClass` y simplemente recompilar las clases descendientes. Los cambios realizados fluyen hasta las clases descendientes.

Modificadores de acceso

Es importante comprender cuándo son accesibles los miembros de una clase (tanto variables como métodos). Hay varias opciones en Java que le permiten definir con exactitud lo accesibles que quiere que sean estos miembros.

Por lo general, querrá limitar el ámbito de los elementos del programa, incluidos los miembros de clases, tanto como sea posible. Cuanto menor número de lugares en que algo es accesible, menor número de lugares en que puede ser incorrectamente accedido.

Hay cuatro diferentes modificadores de acceso para los miembros de clases en Java: `private`, `protected`, `public`, y `default` (o la ausencia de modificador). Esto se complica ligeramente por el hecho de que las clases dentro del mismo paquete tienen diferente acceso que las clases de fuera del paquete. Sin embargo, hay dos tablas que muestran tanto la accesibilidad como la heredabilidad de las clases y de las variables miembros dentro del mismo paquete y fuera de él (los paquetes se explican en una sección posterior).

Acceso desde el mismo paquete de la clase

Modificador de Acceso	Heredado	Accesible
default (sin modificador)	Sí	Sí
Públic	Sí	Sí
Protected	Sí	Sí
Private	No	No

Esta tabla muestra cómo se acceden y heredan los miembros de una clase con respecto a otros miembros del mismo paquete. Por ejemplo, un miembro declarado privado, no puede ser accedido o heredado por otros miembros del mismo paquete. Por otro lado, los miembros declarados utilizando los otros modificadores pueden ser accedidos y heredados por todos los otros miembros del paquete. Todas las partes de la aplicación de ejemplo forman parte del paquete `oop1`, por lo que no tiene que preocuparse de acceder a clases en otro paquete.

Acceso fuera de un paquete

Las reglas cambian si accede a código que está fuera del paquete de su clase:

Modificador de Acceso	Heredado	Accesible
default (sin modificador)	No	No
Públic	Sí	Sí
Protected	Sí	No
Private	No	No

Por ejemplo, esta tabla muestra que un miembro protegido puede ser heredado, pero no accedido, por clases externas a su paquete.

Obsérvese que, en ambas tablas de acceso, los miembros públicos están disponibles para cualquiera que quiera acceder a ellos, (y no hay que olvidar que los constructores son siempre públicos), mientras que los privados nunca son accesibles ni heredables fuera de la clase. Así que, debe declarar privado cualquier variable miembro o método que quiera mantener interno en la clase.

Una práctica recomendada en la programación orientada a objetos es ocultar información dentro de la clase haciendo privadas todas las variables miembro de la clase y accediendo a ellas mediante métodos que están en un formato específico llamados métodos de acceso.

Métodos de acceso

Los métodos de acceso (a veces llamados de obtención/asignación) son métodos que proporcionan una interfaz externa pública con la clase, manteniendo privado en la clase el almacenamiento real de datos. Es una buena idea, porque así puede, en cualquier momento futuro, cambiar la representación interna de los datos en la clase sin tocar los métodos que realmente asignan esos valores internos. Mientras no cambie la interfaz pública de la clase, no estropeará ningún código que se apoye en esa clase y en sus métodos públicos.

Habitualmente, los métodos accesores se presentan por pares en Java: uno para *obtener* el valor interno y otro para *asignar* el valor interno. Por convención, el método Get (obtener) utiliza el nombre de la variable interna privada con "get" como prefijo. El método Set (asignar) hace lo mismo con "set". Una propiedad de sólo lectura tendrá sólo un método Get. Habitualmente, los métodos Get Booleanos utilizan "is" o "has" como prefijo en lugar de "get". Los métodos de acceso facilitan también la validación del dato que se asigna a una variable miembro en particular.

A continuación, se propone un ejemplo. En su `DogClass`, convierta en privadas todas las variables miembro internas y añada métodos accesores para acceder a los valores internos. `DogClass` crea sólo una variable miembro nueva, `tail` (cola).

```
package oop1;

public class DogClass extends MammalClass{

  // métodos de acceso de properties
  // Cola
  public boolean hasTail() {
    return tail;
  }

  public void setTail( boolean value ) {
    tail = value;
  }

  public DogClass() {
    setName("Snoopy");
    setAge(2);
    setTail(true);
  }

  private boolean tail;
}
```

La variable `tail` se ha movido a la parte inferior de la clase y ahora se declara como privada. La ubicación de la definición no es importante, pero es habitual en Java colocar los miembros privados de la clase en la parte inferior de su definición (después de todo, no se les puede acceder

desde fuera; por lo tanto, si va a leer el código, estará interesado antes en los aspectos públicos). `DogClass` tiene ahora métodos públicos para recuperar y asignar el valor de `tail`. El que obtiene es `hasTail()` y el que asigna es `setTail()`.

Siga la misma pauta y revise `ManClass` para que quede así:

```
package oop1;

 public class ManClass extends MammalClass {

  public boolean isMarried() {
    return married;
  }

  public void setMarried(boolean value) {
    married = value;
  }

  public ManClass() {
    setName("Steven");
    setAge(35);
    setEyeColor("Azul");
    setMarried(true);
  }

  private boolean married;
}
```

Observe que los constructores para estas dos clases utilizan ahora métodos de acceso para asignar los valores de las variables de `MammalClass`. Pero `MammalClass` no tiene aún método de acceso para asignar estos valores, así que debe añadírselos.

Cambie `MammalClass` de manera que quede así:

```
public class MammalClass {

  // métodos de acceso de propiedades
  // Nombre
  public String getName() {
    return name;
  }

  public void setName(String Value) {
    name = value;
  }
  // color de ojos
  public String getEyeColor() {
    return eyeColor;
  }

  public void setEyeColor(String value) {
    eyeColor = value;
  }
```

```
  // sonido
  public String getSound() {
    return sound;
  }

  public void setSound(String value) {
    sound = value;
  }

  // edad
  public int getAge() {
    return age;
  }

  public void setAge(int value)  {
    if (value > 0) {
      age = value;
    }
    else
      age = 0;
  }

  public MammalClass() {
    setName("El nombre");
    setEyeColor("Marrón");
    setAge(0);
  }

  private String name, eyeColor, sound;
  private int age;
}
```

Advierta que se ha añadido a `MammalClass` una nueva variable miembro, `sound` (sonido). Y también tiene métodos de acceso. Puesto que `DogClass` y `ManClass` derivan de `MammalClass`, tienen también una propiedad `sound`.

Los manejadores de sucesos de `Marco1.java` deberían utilizar también los métodos de acceso. Modifique los manejadores de sucesos de la siguiente forma:

```
void jButton1_actionPerformed(ActionEvent e) {
  dog = new DogClass();
  txtfldDogName.setText(dog.getName());
  txtfldDogEyeColor.setText(dog.getEyeColor());
  txtfldDogAge.setText(Integer.toString(dog.getAge()));
  chkboxDog.setSelected(true);
}

void jButton2_actionPerformed(ActionEvent e) {
  man = new ManClass();
  txtfldManName.setText(man.getName());
  txtfldManEyeColor.setText(man.getEyeColor());
  txtfldManAge.setText(Integer.toString(man.getAge()));
```

```
    chkboxMan.setSelected(true);
  }
```

Clases abstractas

Es posible declarar un método en una clase como *abstracto*, lo que significa que no habrá implementación para el método en esa clase, pero todas las clases que hereden esta clase deben realizar una implementación.

Por ejemplo, suponga que quiere que todos los mamíferos tengan la capacidad de informar de su velocidad máxima corriendo, pero quiere que cada mamífero informe de una velocidad distinta. En la clase mamífero debe crear un método abstracto llamado *speed()* (velocidad). Añada un método `speed()` a `MammalClass` justo encima de las declaraciones de variables miembro privadas, en la parte inferior del código:

```
abstract public void speed();
```

Una vez que tenga un método abstracto en una clase, la clase completa debe ser declarada como abstracta. Esto indica que una clase que contenga al menos un método abstracto (y por consiguiente sea una clase abstracta) no puede ser instanciada. De modo que añada la palabra clave `abstract` al principio de la declaración de `MammalClass`, de manera que quede así:

```
abstract public class MammalClass {

  public String getName() {
  ...
```

Ahora, cada clase que herede de `MammalClass` debe implementar un método `speed()`. Así que, añada dicho método al código de `DogClass` bajo el constructor `DogClass()`:

```
public void speed() {
  JOptionPane.showMessageDialog(null, "30 mph", "Dog Speed", 1);
}
```

Añada este método `speed()` al código de `ManClass`:

```
public void speed() {
  JOptionPane.showMessageDialog(null, "17 mph", "Man Speed", 1);
}
```

Puesto que cada método `speed()` crea un componente `JOptionPane`, que es un componente Swing, añada esta sentencia justo a continuación de la sentencia de paquete, en la parte superior de `DogClass` y `ManClass`:

```
import javax.swing.*;
```

Esta sentencia hace accesible la biblioteca completa Swing a estas clases. Pronto leerá más sobre sentencias de importación.

Polimorfismo

Polimorfismo es la capacidad de dos clases distintas, pero relacionadas, de recibir el mismo mensaje y actuar cada una a su manera. En otras palabras, dos clases diferentes (pero relacionadas) pueden tener el mismo nombre de método, pero lo implementan de formas distintas.

Por lo tanto, puede tener un método de clase que esté también implementado en una clase descendiente, y puede acceder al código desde la clase antecesora (similar al encadenamiento automático de constructores que se explicó anteriormente). Al igual que en el ejemplo de constructores, puede utilizar la palabra clave `super` para acceder a cualquier método o variable miembro de la superclase.

He aquí un sencillo ejemplo. Tenemos dos clases, `Parent` y `Child`.

```
class Parent {
  int aValue = 1;
  int someMethod(){
    return aValue;
  }
}

class Child extends Parent {
  int aValue;                          // este aValue es parte de esta clase
  int someMethod() {                   // esto redefine el método de Parent
    aValue = super.aValue + 1;         // accesp a aValue de Parentcon super
    return super.someMethod() + aValue;
  }
}
```

El método `someMethod()` de `Child` redefine el `someMethod()` de `Parent`. Un método de una clase descendiente, con el mismo nombre que un método en la superclase, pero implementado de forma distinta y que, por lo tanto, tiene diferente comportamiento, es un método `redefinido`.

¿Puede ver cómo el `someMethod()` de la clase `Child` devolverá el valor 3? El método accede la variable `aValue` de `Parent` utilizando la palabra clave `super`, le añade el valor 1, y asigna el valor resultante, 2, a su propia variable `aValue`. La última línea del método llama al `someMethod()` de `Parent`, que devuelve `Parent.aValue` con valor 1. A esto, le añade el valor de `Child.aValue`, al que se asignó el valor 2 en la línea anterior. Así que 1 + 2 = 3.

Utilización de interfaces

Una interfaz es muy parecida a una clase abstracta, pero con una diferencia importante: una interfaz no puede incluir código. El mecanismo de interfaz en Java está pensado para reemplazar la herencia múltiple.

Una interfaz es una declaración de una clase especializada que puede declarar constantes y declaraciones de métodos, pero no

implementaciones de métodos. Nunca se puede poner código en una interfaz.

He aquí una declaración de interfaz para nuestra aplicación de ejemplo:

Puede utilizar el Asistente de Interfaz de JBuilder para iniciar una interfaz:

1. Abra la Galería de objetos seleccionando Archivo | Nuevo y pulse la pestaña General. Haga doble clic en el icono Interfaz para iniciar el Asistente para interfaces.
2. Especifique el nombre de la interfaz como `SoundInterface`, manteniendo sin cambios todos los restantes valores. (Puede desmarcar Generar Comentarios de Cabecera para omitir cabeceras.)
3. Elija Aceptar para generar la nueva interfaz.

Dentro del nuevo `SoundInterface`, añada una declaración de método `speak()` (habla), de manera que la interfaz quede así:

```
package oop1;

public interface SoundInterface {

  public void speak();
}
```

Obsérvese que se utiliza la palabra clave `interface` en lugar de `class`. Todos los métodos declarados en una interfaz son públicos por defecto, por lo que no hay necesidad de especificar la accesibilidad. Una clase puede implementar una interfaz utilizando la palabra clave `implements`. Además, una clase puede derivar sólo de otra clase, pero una clase puede implementar tantas interfaces como sea necesario. He aquí cómo situaciones que en otros lenguajes se suelen manejar con herencia múltiple se manejan con interfaces en Java. En muchos casos se puede tratar la interfaz como si fuera una clase. En otras palabras, puede tratar objetos que implementan una interfaz como subclases de la interfaz, para más comodidad. Tenga en cuenta, sin embargo, que sólo puede acceder a los métodos definidos en esa interfaz si está convirtiendo un objeto que implementa la interfaz.

Lo que sigue es un ejemplo de polimorfismo y de interfaces. Queremos que la definición de `MammalClass` implemente el nuevo `SoundInterface`. Esto se hace añadiendo las palabras `implements SoundInterface` a la definición de la clase. A continuación, debe definir e implementar un método `speak()` para `MammalClass`. Modifique `MammalClass` de forma que implemente `SoundInterface` y un método `speak()`. He aquí el código completo de `MammalClass`:

```
package oop1;

import javax.swing.*;

abstract public class MammalClass implements SoundInterface {
```

```
// métodos de acceso de propiedades
// nombre
public String getName() {
  return name;
}

public void setName( String value ) {
  name = value;
}

// color de ojos
public String getEyeColor() {
  return eyeColor;
}

public void setEyeColor( String value ) {
  eyeColor = value;
}

// sonido
public String getSound() {
  return sound;
}

public void setSound( String value ) {
  sound = value;
}

// Edad
public int getAge() {
  return age;
}

public void setAge( int value ) {
  if ( value > 0 )
  {
    age = value;
  }
  else
    age = 0;
}

public MammalClass() {
  setName("El nombre");
  setEyeColor("Marrón");
  setAge(0);
}

public void speak() {
  JOptionPane.showMessageDialog(null, this.getSound(),
  this.getName() + " Says", 1);
}
```

```
  abstract public void speed();

  private String name, eyeColor, sound;
  private int age;
}
```

Ahora, la definición de `MammalClass` implementa completamente `SoundInterface`. Debido a que la implementación del método `speak()` utiliza el componente `JOptionPane`, que es parte de la biblioteca Swing, debe añadir una sentencia de importación cerca de la parte superior del archivo:

```
import javax.swing.*;
```

Esta sentencia de importación convierte en disponible la biblioteca Swing completa para `MammalClass`. Leerá más sobre sentencias de importación en el apartado "Sentencia import" en la página 6-25.

Puesto que `DogClass` y `ManClass` derivan de `MammalClass`, tienen automáticamente acceso al método `speak()` definido en `MammalClass`. No tienen que implementar específicamente `speak()` por sí mismos. El valor de la variable `sound` que ha sido pasado al método `speak()` es asignado en los constructores de `DogClass` y `ManClass`. He aquí como debe quedar la clase `DogClass`:

```
package oop1;
import javax.swing.*;

public class DogClass extends MammalClass{

  public boolean hasTail() {
    return tail;
  }

  public void setTail(boolean value) {
    tail = value;
  }

  public DogClass() {
    setName("Snoopy");
    setSound("¡Guau!¡Guau!");
    setAge(2);
    setTail(true);
  }

  public void speed() {
    JOptionPane.showMessageDialog(null, "30 mph", "Dog Speed", 1);
  }

  private boolean tail;
}
```

Y así como debe quedar ManClass:

```
package oop1;
import javax.swing.*;

 public class ManClass extends MammalClass {

  public boolean isMarried() {
    return married;
  }

  public void setMarried(boolean value) {
    married = value;
  }

  public ManClass() {
    setName("Steven");
    setEyeColor("Azul");
    setSound("Hello there! I'm " + this.getName() + ".");
    setAge(35);
    setMarried(true);
  }

  public void speed() {
    JOptionPane.showMessageDialog(null, "17 mph", "Man Speed", 1);
  }

  private boolean married;
}
```

Adición de dos nuevos botones

Aunque ha añadido los métodos `speak()` y `speed()` a la aplicación ejemplo, hasta ahora la aplicación no los llama en ningún momento. Para cambiar esto, añada dos botones más a la clase `Marco1.java`:

1. Pulse la pestaña Marco1 en el panel de contenido.
2. Pulse sobre la pestaña Diseño para visualizar el diseñador de interfaces de usuario.
3. Coloque dos botones adicionales en el formulario.
4. Cambie en el Inspector el valor de la propiedad `text` del primer botón a Velocidad, y cambie la propiedad `text` del segundo botón a Habla.

Figura 6.2 Nueva versión de la aplicación ejemplo con los botones Velocidad y Habla añadidos

Pulse la pestaña Fuente para volver al código de `Marco1.java` y añadir el código mostrado en negrita a la definición de la clase:

```
// Crear una referencia para los objetos
DogClass dog;
ManClass man;

//Crear una matriz de SoundInterface
SoundInterface soundList[] = new SoundInterface[2];

//Crear una matriz de Mammal
MammalClass mammalList[] = new MammalClass[2];
```

Ha añadido código que crea dos matrices: una para Mammals y otra para SoundInterfaces.

Añada también código a los manejadores de sucesos Create Dog y Create Man para añadir a las matrices referencias a los objetos hombre y perro:

```
void button1_actionPerformed(ActionEvent e) {
  dog = new DogClass();
  txtfldDogName.setText(dog.getName());
  txtfldDogEyeColor.setText(dog.getEyeColor());
  txtfldDogAge.setText(Integer.toString(dog.getAge()));
  chkboxDog.setSelected(true);
  mammalList[0] = dog;
  soundList[0]  = dog;
}

void button2_actionPerformed(ActionEvent e) {
  man = new ManClass();
  txtfldManName.setText(man.getName());
  txtfldManEyeColor.setText(man.getEyeColor());
  txtfldManAge.setText(Integer.toString(man.getAge()));
  chkboxMan.setSelected(true);
```

```
    mammalList[1] = man;
    soundList[1]  = man;
    }
```

Vuelva al diseñador de interfaces, haga doble clic en el botón Velocidad, y rellene el manejador de sucesos que le presenta JBuilder de la siguiente forma:

```
void button3_actionPerformed(ActionEvent e) {
  for (int i = 0; i <= 1; i++) {
    mammalList[i].speed();
  }
}
```

El código recorre la lista de mamíferos guardada en la matriz (¡Los dos, perro y hombre!) y dice a cada objeto que muestre su velocidad. En el primer recorrido de la lista, el perro muestra su velocidad; en el segundo, el hombre hace otro tanto. Esto es polimorfismo en acción - dos objetos distintos, pero relacionados, recibiendo el mismo mensaje y reaccionando ante él a su propia manera.

El código para el botón Habla es muy similar.

```
void button4_actionPerformed(ActionEvent e) {
  for (int i = 0; i <= 1; i++) {
    soundList[i].speak();
  }
}
```

Seleccione Archivo | Guardar todo para guardar todos los cambios.

Como puede ver, puede tratar a `SoundInterface` como una clase cuando sea conveniente. Observe que la interfaz le proporciona muchos de los beneficios de la herencia múltiple sin su complejidad añadida.

Ejecución de la aplicación

Ya está listo para ejecutar su aplicación modificada. Elija Ejecutar | Ejecutar el Proyecto para recompilar su proyecto y ejecutarlo.

Cuando su aplicación comience a ejecutarse, asegúrese de pulsar los botones Crear Perro y Crear Humano para crear los objetos `dog` y `man` antes de intentar utilizar los botones Velocidad y Habla; si no lo hace, obtendrá una excepción `NullPointerException`.

Una vez que los objetos existan y pulse el botón Velocidad, aparecerá un mensaje informando de la velocidad del primer mamífero de la matriz `mammalList`, el perro. Cuando pulse Aceptar para eliminar el mensaje, aparece un mensaje que informa de la velocidad del segundo mamífero, el hombre. Pulsar el botón Habla produce un comportamiento similar, salvo que los mensajes mostrados son los sonidos que cada mamífero puede hacer.

Paquetes Java

Para facilitar la reutilización de código, Java permite agrupar varias definiciones de clase juntas en un agrupamiento lógico llamado un *paquete*. Si, por ejemplo, crea un grupo de reglas de negocio que modelan los procesos de trabajo de su organización, puede que quiera colocarlos juntos en un paquete. Esto hace más sencillo reutilizar el código que haya creado previamente.

Sentencia import

El lenguaje Java se suministra con muchos paquetes predefinidos. Por ejemplo, el paquete `java.applet` contiene clases para trabajar con applets Java:

```
public class Hello extends java.applet.Applet {
```

Este código se refiere a la clase llamada `Applet` del paquete Java `java.applet`. Puede imaginarse que debe ser muy tedioso tener que repetir el nombre completo de la clase `java.applet.Applet` cada vez que se refiera a esta clase. En su lugar, Java ofrece una alternativa. Puede elegir importar un paquete que vaya a utilizar frecuentemente:

```
import java.applet.*;
```

Esto le ordena al compilador que, en el caso de que vea un nombre de clase que no reconozca, que lo busque en el paquete `java.applet`. Ahora, cuando declare una clase nueva, puede decir:

```
public class Hello extends Applet {
```

Esto es más conciso. El problema se presenta cuando hay dos clases con el mismo nombre definidas en dos paquetes importados diferentes. En este caso, tiene que utilizar el nombre completo.

Declaración de paquetes

Crear sus propios paquetes es casi tan fácil como usarlos. Por ejemplo, si quiere crear un paquete llamado `mi paquete`, simplemente utilizará una sentencia `package` al principio de su archivo:

```
package mi paquete;

public class Hello extends java.applet.Applet {

public void init() {
    add(new java.awt.Label("Hello World Wide Web!"));
  }

} // end class
```

Ahora, cualquier otro programa puede acceder a las clases declaradas en `mi paquete` con la sentencia:

```
import mi paquete.*;
```

Recuerde, este archivo debe estar en un subdirectorio llamado `mi paquete`. Esto permite al compilador Java localizar fácilmente su paquete. El Asistente para Proyectos de JBuilder asignará automáticamente el directorio para igualarlo con el nombre del proyecto. Además, tenga en mente que el directorio base de cualquier paquete que importe debe estar relacionado en la Vía de acceso de origen del JBuilder IDE o en la de su proyecto. Es bueno recordar esto si decide reubicar un paquete en un directorio base diferente.

Para obtener más información sobre trabajo con paquetes en JBuilder, consulte “Paquetes” en *Creación de aplicaciones con JBuilder*.

Capítulo

7

Técnicas de hilos

Los hilos forman parte de todo programa Java. Un hilo es un único flujo de control secuencial dentro de un programa. Tiene un comienzo, una secuencia, y un final. Un hilo no puede correr por sí mismo; se ejecuta dentro de un programa. Si su programa es una única secuencia de ejecución, no necesita configurar un hilo explícitamente, la Máquina Virtual Java (MV) se ocupará de ello.

Uno de los aspectos más poderosos del lenguaje Java es que se pueden programar múltiples hilos de ejecución para que corran simultáneamente en el mismo programa. Por ejemplo, un navegador Web puede descargar un archivo de un sitio, y acceder a otro sitio al mismo tiempo. Si el navegador puede realizar simultáneamente dos tareas, no tendrá que esperar hasta que el archivo haya terminado de descargarse para poder navegar a otro sitio.

La MV Java siempre está ejecutándose en múltiples hilos, llamados *hilos de utilidad*. Por ejemplo, un hilo de utilidad que corre permanente realiza las tareas de liberación de memoria. Otro hilo de utilidad maneja los sucesos del ratón y el teclado. Es posible que su programa bloquee uno de los hilos de la máquina virtual Java (MV). Si su programa aparenta estar bloqueado -sin que le sean enviados sucesos- pruebe a utilizar hilos.

El ciclo de vida de un hilo

Cada hilo tiene un ciclo de vida definido - arranca y para, puede hacer una pausa y esperar que suceda un evento, y puede notificar a otro hilo mientras se está ejecutando. Esta sección introduce en algunos de los aspectos más comunes del ciclo de vida de los hilos.

Personalización del método run()

Utilice el método `run()` para implementar el comportamiento en ejecución del hilo. Este comportamiento puede ser cualquier cosa que una sentencia Java pueda realizar - cálculos, ordenación, animaciones, etc.

Puede elegir entre dos técnicas para implementar el método `run()` para un hilo:

- Crear una subclase de `java.lang.Thread`
- Implementar la interfaz `java.lang.Runnable`

Creación de subclases desde la clase Thread

Si está creando una clase cuyos objetos quiere ejecutar en hilos separados, necesita derivar la clase `java.lang.Thread`. El método `run()` por defecto, perteneciente a la clase `Thread` no hace nada, por lo que su clase necesitará redefinir dicho método. El método `run()` es lo primero que se ejecuta cuando un hilo se inicia.

A modo de ejemplo, la siguiente clase, `CountThread`, crea una subclase desde `Thread` y redefine su método `run()`. En este ejemplo, el método `run()` identifica un hilo e imprime su nombre en la pantalla. El bucle `for` cuenta enteros desde el valor de `start` hasta el valor de `finish`, e imprime cada uno en la pantalla. A continuación, y antes de que el bucle finalice su ejecución, el método imprime una cadena que indica que el hilo ha terminado de ejecutarse.

```
public class CountThread extends Thread {
   private int start;
   private int finish;

   public CountThread(int from, int to) {
     this.start = from;
     this.finish = to;
   }

   public void run() {
      System.out.println(this.getName()+ " started executing...");
      for (int i = start; i <= finish; i++) {
         System.out.print (i + " ");
      }
   System.out.println(this.getName() + " finished executing.");
   }
}
```

Para probar la clase `CountThread`, puede crear una clase de prueba:

```
public class ThreadTester {
   static public void main(String[] args) {
      CountThread thread1 = new CountThread(1, 10);
      CountThread thread2 = new CountThread(20, 30);
      thread1.start();
```

```
        thread2.start();
    }
}
```

El método `main()` de la aplicación de prueba crea dos objetos `CountThread`: `thread1`, que cuenta de 1 a 10, y `thread2`, que cuenta de 20 a 30. Ambos hilos se inician llamando a sus métodos `start()`. El resultado de esta aplicación de prueba podría parecerse a esto:

```
Thread-0 started executing...
1 2 3 4 5 6 7 8 9 10 Thread-0 finished executing.
Thread-1 started executing...
20 21 22 23 24 25 26 27 28 29 30 Thread-1 finished executing.
```

Observe que el resultado no muestra los nombres de hilos como `thread1` y `thread2`. A menos que le asigne específicamente un nombre a un hilo, Java le dará automáticamente uno en la forma `Thread-n`, donde `n` es un número único, comenzando en 0. Puede asignar un nombre a un hilo en el constructor de clase, o con el método `setName(String)`.

En este ejemplo, `Thread-0` comenzó a ejecutarse primero y finalizó primero. Sin embargo, podría haber comenzado primero y terminado último, o comenzar parcialmente y ser interrumpido por `Thread-1`. Esto es debido a que en Java no está garantizada la ejecución de los hilos en ninguna secuencia en particular. De hecho, cada vez que ejecute `ThreadTester`, podría obtener un resultado diferente. Básicamente, el proceso de programar los hilos es controlado por el planificador de hilos de Java, y no por el programador. Para obtener más información, consulte "Prioridad de los hilos" en la página 7-7.

Implementación de la interfaz Runnable

Si quiere que los objetos de una clase existente se ejecuten en sus propios hilos, puede implementar la interfaz `java.lang.Runnable`. Esta interfaz añade el soporte de hilos a las clases que no heredan desde la clase `Thread`. Proporciona un solo método, `run()`, que tiene que implementar en la clase que se esté definiendo en ese momento.

Nota Si su clase deriva de otra que no sea `Thread`, por ejemplo, `Applet`, deberá utilizar la interfaz `Runnable` (ejecutable) para crear hilos.

Para crear una nueva clase `CountThread` que implemente la interfaz `Runnable`, necesita cambiar la definición de clase de `CountThread`. El código de la definición de clase con los cambios señalados en negrita es el siguiente:

```
public class CountThread implements Runnable {
```

También tendrá que cambiar la forma de obtención del nombre del hilo. Puesto que no está instanciando la clase `CountThread`, no puede llamar al método `getName()` de la superclase de `CountThread`, en este caso,`java.lang.Object`. Este método no está disponible. En su lugar, necesita utilizar el método `Thread.currentThread()`, el cual devuelve el nombre del hilo en un formato ligeramente diferente al del método `getName()`.

La clase completa, con los cambios señalados en negrita, tendría este aspecto:

```
public class CountThread implements Runnable {
   private int start;
   private int finish;

   public CountThread(int from, int to) {
      this.start = from;
      this.finish = to;
   }

   public void run() {
      System.out.println(Thread.currentThread() + " started executing...");
      for (int i=start; i <= finish; i++) {
         System.out.print (i + " ");
      }
   System.out.println(Thread.currentThread() + " finished executing.");
   }
}
```

La aplicación de prueba necesitaría cambiar la forma en que se crean sus objetos. En lugar de instanciar `CountThread`, la aplicación necesita crear un objeto `Runnable` desde la nueva clase y pasárselo a uno de los constructores de hilos. El código, con los cambios resaltados en negrita, quedaría así:

```
public class ThreadTester {
   static public void main(String[] args) {
      CountThreadRun thread1 = new CountThreadRun(1, 10);
      new Thread(thread1).start();
      CountThreadRun thread2 = new CountThreadRun(20, 30);
      new Thread(thread2).start();
  }
}
```

El resultado de este programa sería este:

```
Thread[Thread-0,5,main] started executing...
1 2 3 4 5 6 7 8 9 10 Thread[Thread-0,5,main] finished executing.
Thread[Thread-1,5,main] started executing...
20 21 22 23 24 25 26 27 28 29 30 Thread[Thread-1,5,main] finished executing.
```

`Thread-0` es el nombre del hilo, `5` es la prioridad que se le ha dado al hilo cuando ha sido creado, y `main` es el `ThreadGroup` (grupo de hilos) por defecto al que el hilo ha sido asignado. (La prioridad y el grupo son asignados por la máquina virtual Java (MV) si no se especifican.)

Consulte "Prioridad de los hilos" en la página 7-7
"Grupos de hilos" en la página 7-9

Definición de un hilo

La clase `Thread` proporciona siete constructores. Estos constructores combinan los siguientes tres parámetros de diversas maneras:

- Un objeto `Runnable` cuyo método `run()` se ejecutará dentro del hilo.
- Un objeto `String` para identificar el hilo.
- Un objeto `ThreadGroup` al que asignar el hilo. La clase `ThreadGroup` organiza grupos de hilos relacionados.

Constructor	Descripción
`Thread()`	Asigna un nuevo objeto `Thread`.
`Thread(Runnable target)`	Asigna un nuevo objeto `Thread` de forma que su objeto ejecutable sea `destino`.
`Thread(Runnable target, String name)`	Asigna un nuevo objeto `Thread` de manera que tiene como objeto ejecutable a `target` y, como nombre, el `nombre` especificado.
`Thread(String name)`	Asigna un nuevo objeto `Thread` que tiene por nombre el `nombre` especificado.
`Thread(ThreadGroup group, Runnable target)`	Asigna un nuevo objeto `Thread` de forma que pertenezca al grupo referenciado por `grupo` y tenga a `destino` como su objeto ejecutable.
`Thread(ThreadGroup group, Runnable target, String name)`	Asigna un nuevo objeto `Thread` que tiene como objeto ejecutable a `target`; por nombre, el `nombre` especificado, y pertenece al grupo de hilos referenciado por `grupo`.
`Thread(ThreadGroup group, String name)`	Asigna un nuevo objeto `Thread` que pertenece al grupo de hilos referenciado por `grupo` y tiene por nombre el `nombre` especificado.

Si quiere asociar un estado a un hilo, utilice un objeto `ThreadLocal` cuando lo cree. Esta clase permite que cada hilo tenga su propia copia inicializada independientemente de una variable estática privada, por ejemplo un identificador de usuario o de transacción.

Inicio de un hilo

Para iniciar un hilo, llame al método `start()`. Este método crea los recursos del sistema necesarios para correr el hilo, lo planifica, y llama al método `run()`.

Cuando el método `start()` vuelve, el hilo está corriendo y está en un estado ejecutable. Debido a que la mayoría de los ordenadores tienen una sola CPU, la MV Java debe programar los hilos. Para obtener más información, consulte "Prioridad de los hilos" en la página 7-7.

Hacer un hilo no ejecutable

Para poner un hilo en estado no ejecutable, utilice una de las siguientes técnicas:

- Un método `sleep()`: estos métodos le permiten especificar un número concreto de segundos y nanosegundos de parada.
- El método `wait()`: este método hace que el hilo actual espere a que se dé una condición específica para ser usado.
- Bloqueo del hilo en entrada o salida.

Cuando el hilo está no ejecutable, el hilo no corre, incluso aunque el procesador esté disponible. Para salir del estado no ejecutable, debe cumplirse la condición de entrada a dicho estado. Por ejemplo, si utilizó el método `sleep()`, debe haber pasado el número de segundos especificado. Si utilizó el método `wait()`, otro objeto debe indicarle al hilo en espera (con `notify()` o `notifyAll()`) el cambio en la condición. Si un hilo está bloqueado por entrada o salida, la entrada o salida deben terminar.

También puede utilizar el método `join()` para hacer que un hilo espere a que termine otro hilo que se está ejecutando. Se llama a este método para el hilo al que se está esperando. Puede especificar un tiempo de espera para un hilo, pasándole al método un parámetro en milisegundos. El método `join()` espera hasta que el tiempo ha pasado o el hilo haya finalizado. Este método trabaja en conjunción con el método `isAlive()` - `isAlive()` devuelve `true` si el hilo ha sido iniciado y no parado.

Observe que los métodos `suspend()` y `resume()` han sido desaconsejados. El método `suspend()` tiene tendencia a los bloqueos. Si el hilo de destino está bloqueando un monitor que protege un recurso crítico del sistema cuando es suspendido, ningún hilo puede acceder a ese recurso hasta que el hilo destino sea reanudado. Un monitor es un objeto Java utilizado para

verificar que sólo un hilo a la vez está ejecutando los métodos sincronizados del objeto. Para obtener más información, consulte "Sincronización de hilos" en la página 7-8.

Parada de un hilo

Ya no se puede parar un hilo con el método `stop()`. Este método ha sido desaconsejado porque no es seguro. Parar un hilo (con stop()) produce el desbloqueo de todos los monitores que ha bloqueado. Si un objeto previamente protegido por uno de estos monitores está en un estado incoherente, otros hilos verán a ese objeto como incoherente. Esto puede hacer que su programa sufra daños.

Para parar un hilo, finalice el método `run()` con un bucle finito.

Si desea más información, consulte el tema "Why are Thread.stop, Thread.suspend, Thread.resume and runtime.runFinalizersOnExit Deprecated?" en la *Java 2 SDK, Standard Edition Documentation*

Prioridad de los hilos

Cuando se crea un hilo Java, hereda su prioridad del hilo que lo creó. Se puede definir la prioridad de un hilo utilizando el método `setPriority()`. Las prioridades de los hilos se representan como valores enteros que abarcan desde `MIN_PRIORITY` hasta `MAX_PRIORITY` (constantes de la clase `Thread`). Se ejecuta el hilo con la prioridad más alta.

Cuando un hilo se para, cede el paso, o se vuelve no ejecutable, se ejecuta otro con una prioridad inferior. Si dos hilos de la misma prioridad están esperando, el programador de Java elegirá por turno uno de ellos para ejecutarse. El hilo correrá hasta que:

- Un hilo de mayor prioridad se vuelva ejecutable.
- El hilo ceda el paso, mediante el uso del método `yield()` o cuando su método `run()` finalice.
- Su tiempo adjudicado haya expirado. Esto sólo es aplicable a sistemas que soporten time slicing (reparto de tiempos).

Este tipo de programación se basa en un algoritmo de planificación llamado *programación fija de prioridades.* Los hilos se ejecutan en base a su prioridad comparada con otros hilos. El hilo con la mayor prioridad será el que se esté ejecutando siempre.

Reparto de tiempos

Algunos sistemas operativos utilizan un mecanismo de programación conocido como *reparto de tiempos.* El reparto de tiempos divide la CPU en porciones de tiempo. El sistema proporciona tiempo para ejecutarse a los hilos de mayor prioridad hasta que uno o más de ellos terminan o hasta que uno de prioridad mayor pasa a estado ejecutable. Puesto que el reparto de tiempos no es admitido por todos los sistemas operativos, su programa no debe depender de un mecanismo de planificación de reparto de tiempos.

Sincronización de hilos

Uno de los problemas centrales de la programación multihilo es manejar situaciones en las que más de un hilo tiene acceso a la misma estructura de datos. Por ejemplo, si un hilo estuviera intentando actualizar los elementos de una lista, mientras otro está simultáneamente intentando clasificarla, su programa puede bloquearse o producir resultados incorrectos. Para evitar este problema, debe utilizar la *sincronización de hilos.*

La forma más sencilla de evitar que dos objetos accedan al mismo método al mismo tiempo es requerirle a un hilo que consiga un bloqueo. Cuando un hilo mantiene un bloqueo, otro hilo que también lo necesita tiene que esperar hasta que el primer hilo libera su bloqueo. Para mantener un método *thread-safe* (libre de problemas con los hilos), utilice la palabra clave `synchronized` cuando declare métodos que sólo pueden ser ejecutados por un hilo a la vez. Advierta que también puede sincronizar un objeto.

Por ejemplo, si crea un método `swap()` que intercambia valores utilizando una variable local y crea dos hilos diferentes para ejecutar el método, su programa podría producir resultados incorrectos. El primer hilo, debido a las restricciones del planificador Java, sólo podría haber ejecutado la primera mitad del método. A continuación, el segundo hilo podría ser capaz de ejecutar el método completo, pero utilizando valores incorrectos (puesto que el primer hilo no completó la operación). Entonces el primer hilo vuelve para finalizar el método. En este caso, todo indicará que el intercambio de valores no ha tenido lugar. Para evitar que esto suceda, utilice la palabra clave `synchronized` en la declaración de su método.

Como norma general, cualquier método que modifique una propiedad de un objeto debería declararse como `synchronized`.

Grupos de hilos

Todos los hilos Java son miembros de un *grupo de hilos.* Un grupo de hilos reúne múltiples hilos en un solo objeto, y manipula a todos esos hilos a la vez. Los grupos de hilos son implementados por la clase `java.lang.ThreadGroup`.

El sistema de ejecución coloca a cada hilo en un grupo en el momento de su construcción. El hilo es colocado en un grupo por defecto o en uno que se haya especificado cuando el hilo se creó. No se puede cambiar un hilo a otro grupo una vez que ha sido creado.

Si creó un hilo sin especificar un nombre de grupo en su constructor, el sistema pone al nuevo hilo en el mismo grupo que el hilo que lo ha creado. Habitualmente, los hilos sin especificar son colocados en el grupo `main`. Sin embargo, si creó un hilo en un applet, el nuevo hilo podría ser puesto en otro grupo que `main`, dependiendo del navegador o del visor en el que el applet esté corriendo.

Si construye un hilo con un `ThreadGroup`, el grupo puede ser:

- Un nombre de su propia creación.
- Un grupo creado por el sistema Java.
- Un grupo creado por la aplicación en la que está corriendo su applet.

Para obtener el nombre del grupo del que su hilo forma parte, utilice el método `getThreadGroup()`. Una vez que sepa el grupo de un hilo, puede determinar qué otros hilos están en el grupo y manipularlos todos a la vez.

Capítulo

8

Serialización

La *serialización de un objeto* es el proceso de guardar un objeto completo en disco u otro medio de almacenamiento, listo para ser recuperado en cualquier momento. El proceso de recuperar un objeto es conocido como *paralelización*. En esta sección aprenderá por qué es útil la serialización y cómo Java implementa la serialización y la paralelización.

De un objeto que ha sido serializado se dice que es *persistente*. La mayoría de los objetos en memoria son *transitorios*, lo que significa que se borran cuando se pierde su referencia o el ordenador se apaga. Los objetos persistentes existen mientras haya una copia de ellos almacenada en un disco, cinta o ROM.

¿Por qué serializar?

Tradicionalmente, guardar datos en disco u otro dispositivo de almacenamiento requería definir un formato especial de datos, escribir una serie de funciones para escribir y leer dicho formato y crear una equivalencia entre el formato del archivo y el formato de los datos. Las funciones para leer y escribir datos o bien eran simples y carentes de flexibilidad, o bien eran complejas y difíciles de crear y mantener.

Java está completamente basado en objetos y en programación orientada a objetos, por lo que proporciona un mecanismo de almacenamiento para ellos mediante la serialización. Utilizando la manera de hacer las cosas de Java ya no tendrá que preocuparse con detalles de formatos de archivo y de entrada/salida (E/S). En vez de eso, puede concentrarse en resolver las tareas que importan diseñando e implementando objetos. Si, por ejemplo, crea una clase persistente y posteriormente le añade nuevos campos, no tiene que preocuparse por modificar rutinas que lean y escriban los datos.

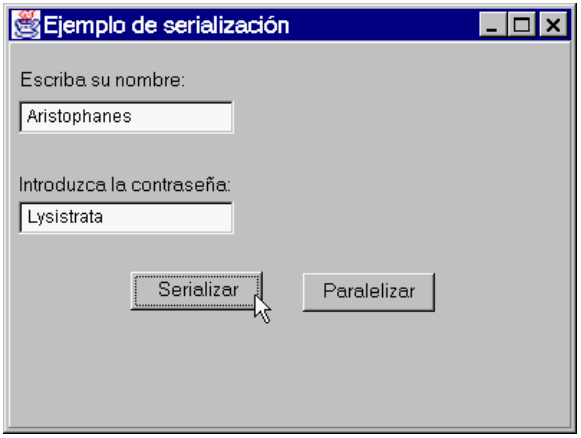
Todos los campos de un objeto serializado serán escritos y recuperados automáticamente.

Serialización en Java

La serialización apareció por primera vez como una característica de JDK 1.1. El soporte de Java para la serialización se compone de la interfaz `Serializable`, la clase `ObjectOutputStream`, la clase `ObjectInputStream` y varias clases e interfaces de apoyo. Examinaremos los tres elementos y mostraremos una aplicación que puede guardar la información del usuario en disco y leerla de nuevo.

Supongamos que desea guardar información de un usuario concreto, tal y como se muestra aquí.

Figura 8.1 Guardar un nombre y contraseña de usuario



Después de que el usuario introduzca su nombre y contraseña en los campos apropiados, la aplicación debe salvar información sobre este usuario en disco. Por supuesto, este es un ejemplo muy simple, pero es fácil de imaginar lo que hay que hacer para guardar datos de las preferencias del usuario sobre aplicaciones, el último documento abierto y acciones similares.

Utilizando JBuilder, puede diseñar una interfaz de usuario como la que se muestra arriba. Si necesita ayuda con esta tarea, consulte *Diseño de aplicaciones con JBuilder*. Al campo de texto Nombre llámelo `textFieldName` y al campo contraseña `passwordFieldName`. Además de las dos etiquetas que ve, añada una tercera cerca de la parte inferior del marco y llámela `labelOutput`.

Uso de la interfaz Serializable

Cree una nueva clase que represente al usuario actual. Debe tener propiedades que representen el nombre y la contraseña del usuario.

Para crear la nueva clase:

1. Seleccione Archivo | Nueva clase para iniciar el Asistente para clases.
2. En la sección Información de la clase especifique como nombre de la nueva clase UserInfo. Deje sin cambios los campos restantes.
3. En la sección Opciones, seleccione sólo las opciones Pública y Generar constructor por defecto, desactivando todas las demás.
4. Pulse Aceptar.

El Asistente para clases crea el nuevo archivo de clase y lo añade al proyecto. Modifique el código generado para que quede como sigue:

```
package serialize;

public class UserInfo implements java.io.Serializable {
    private String userName = "";
    private String userPassword = "";

  public UserInfo() {
  }

  public String getUserName() {
    return userName;
  }

  public void setUserName(String s) {
    userName = s;
  }

  public String getUserPassword() {
    return userPassword;
  }

  public void setUserPassword(String s) {
    userPassword = s;
  }
}
```

Ha añadido una variable que almacena el nombre del usuario y otra para su contraseña. También ha añadido métodos de acceso para ambos campos.

Observe que la clase `UserInfo` implementa la interfaz `java.io.Serializable`. Se conoce a `Serializable` como una *interfaz de etiquetado* porque no especifica métodos para su implementación, sino que simplemente "etiqueta" sus objetos como pertenecientes a un tipo en particular.

Cualquier objeto que se pretenda serializar debe implementar la interfaz `Serializable`. Esto es crucial porque, de otra forma, las técnicas que se utilizan más adelante en este capítulo no funcionarán. Si, por ejemplo, intenta serializar un objeto que no implementa esta interfaz, se lanzará una excepción `NotSerializableException`.

Este sería el momento en el que debería importar el paquete `java.io` para que su aplicación tenga acceso a las interfaces y clases de entrada y salida que necesita para escribir y leer objetos. Añada esta importante sentencia a las que están en la parte superior del marco de su clase:

```
import java.io.*
```

Uso de flujos de salida

Antes de serializar el objeto `UserInfo`, debe instanciarlo y configurarlo con los valores que el usuario introduzca en los campos de texto. Cuando el usuario introduce información en los campos y pulsa el botón Serializar, los valores que introdujo se almacenan en la instancia del objeto `UserInfo`:

```
void jButton1_actionPerformed(ActionEvent e) {
    UserInfo user = new UserInfo();               // instancia un objeto de
usuario
    user.setUserName(textFieldName.getText());
    user.setUserPassword(textFieldPassword.getText());
}
```

Si está utilizando el diseñador de interfaces de usuario (UI) de JBuilder, haga doble clic en el botón Serializar y JBuilder iniciará automáticamente el código de suceso `jButton1_actionPerformed()`. Instancie un objeto usuario y añada al manejador de sucesos llamadas a los métodos `user.setUserName()` y `user.setUserPassword()`.

A continuación, abra un `FileOutputStream` para el archivo que contendrá los datos serializados. En este ejemplo, el archivo se llamará `C:\userInfo.ser`. Añada este código al botón Serializar del manejador de sucesos:

```
try {
  FileOutputStream file = new FileOutputStream("c:\userInfo.ser");
```

Cree un `ObjectOutputStream` que serializará el objeto y lo enviará al `FileOutputStream`, añadiendo este código al manejador de sucesos:

```
ObjectOutputStream out = new ObjectOutputStream(file);
```

Ya está listo para enviar el objeto `UserInfo` al archivo. Esto se hace llamando al método `writeObject()` de ObjectOutputStream. Llame al método `flush()` para vaciar el búfer de salida y asegurarse de que el objeto se escribe realmente en el archivo.

```
out.writeObject(u);
out.flush();
```

Cierre el flujo de salida para liberar los recursos, tales como los descriptores de archivos, utilizados por el flujo.

```
    out.close();
}
```

Añada código al manejador que captura una excepción `IOException` si hubiera problemas escribiendo en el archivo o si el objeto no soporta la interfaz *Serializable*.

```
catch (java.io.IOException IOE) {
      labelOutput.setText("IOException");
}
```

Este es el aspecto completo que debe tener el manejador de sucesos del botón Serializar:

```
void jButton1_actionPerformed(ActionEvent e) {
    UserInfo user = new UserInfo();
    user.setUserName(textFieldName.getText());
    user.setUserPassword(textFieldPassword.getText());
            try {
       FileOutputStream file = new FileOutputStream("c:\userInfo.ser");
       ObjectOutputStream out = new ObjectOutputStream(file);
       out.writeObject(user);
       out.flush();
    }
    catch (java.io.IOException IOE) {
       labelOutput.setText("IOException");
    }
    finally {
  out.close();
    }
}
```

Ahora, compile el proyecto y ejecútelo. Introduzca valores en los campos Nombre y Contraseña y pulse el botón Serializar. Puede verificar que el objeto se ha escrito abriéndolo con un editor de texto. (¡No intente modificarlo, o el archivo probablemente resultará dañado!) Observe que un objeto serializado contiene una mezcla de texto ASCII y datos binarios:

Figura 8.2 El objeto serializado

Métodos ObjectOutputStream

La clase `ObjectOutputStream` contiene varios métodos útiles para escribir datos en un flujo. La escritura no está restringida a objetos. Con llamadas a `writeInt()`, `writeFloat()`, `writeDouble()`, etc., podrá escribir cualquiera de los tipos fundamentales en un flujo. Si quiere escribir más de un objeto o tipo fundamental en el mismo flujo, debe hacerlo llamando repetidamente a estos métodos contra el mismo objeto `ObjectOutputStream`. Cuando haga esto, no obstante, deberá leer los objetos *en el mismo orden*.

Uso de flujos de entrada

Ya ha escrito el objeto en el disco. ¿Cómo lo recupera? Una vez que el usuario hace clic en el botón Paralelizar, querrá leer los datos del disco en un nuevo objeto.

Puede comenzar el proceso creando un nuevo objeto `FileInputStream` para leer del archivo que acaba de escribir. Si está utilizando JBuilder, haga doble clic en el botón Paralelizar del Diseñador de interfaces de usuario (UI); en el manejador de sucesos que crea JBuilder, añada el código resaltado:

```
void jButton2_actionPerformed(ActionEvent e) {
            try {
   FileInputStream file = new FileInputStream("c:\userInfo.ser");
```

A continuación, cree un `ObjectInputStream`, que le proporciona la capacidad de leer objetos de ese archivo.

```
ObjectInputStream input = new ObjectInputStream(file);
```

A continuación, llame al método `ObjectInputStream.readObject()` para leer el primer objeto del archivo. `readObject()` devuelve un tipo `Object`, por lo que será preciso convertirlo en el tipo apropiado `(UserInfo)`.

```
UserInfo user = (UserInfo)input.readObject();
```

Cuando haya realizado la lectura, recuerde cerrar el `ObjectInputStream`. Así liberará todos los recursos asociados a él, tales como los descriptores de archivos.

```
input.close();
```

Finalmente, puede utilizar el objeto `user` como utilizaría cualquier otro objeto de la clase `UserInfo`. En este caso, se va a mostrar el nombre y la contraseña en el tercer campo que añadió al cuadro de diálogo:

```
labelOutput.setText("Name is " + user.getUserName() +
   ", password is: " +
   user.getUserPassword());
```

La lectura de un archivo podría causar una `IOException`, así que tiene que gestionar esta excepción. También podría producirse una `StreamCorruptedException` (una subclase de `IOException`) si el archivo ha sufrido algún tipo de daños:

```
    catch (java.io.IOException IOE) {
      labelOutput.setText("IOException");
    }
```

Hay otra excepción que tendrá que gestionar. El método `readObject()` puede lanzar una `ClassNotFoundException`. Esta excepción puede ocurrir si intenta leer un objeto para el que no tiene implementación. Por ejemplo, si el objeto lo escribió otro programa, o ha renombrado la clase `UserInfo` desde que se escribió el archivo, obtendrá una excepción `ClassNotFoundException`.

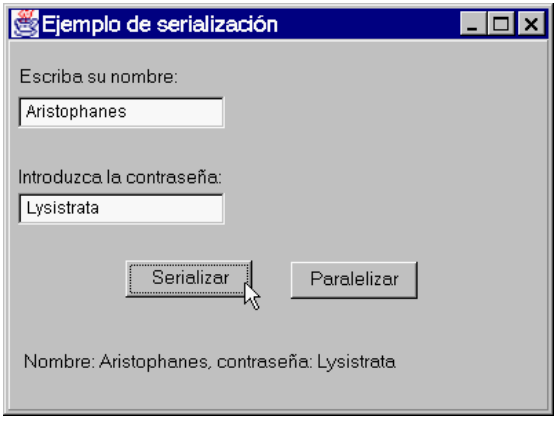
```
    catch (ClassNotFoundException cnfe) {
      labelOutput.setText("ClassNotFoundException");
    }
  }
```

He aquí el manejador de sucesos del botón Paralelizar en su totalidad:

```
  void jButton2_actionPerformed(ActionEvent e) {
             try {
    FileInputStream file = new FileInputStream("c:\userInfo.ser");
    ObjectInputStream input = new ObjectInputStream(file);
    UserInfo user = (UserInfo)input.readObject();
    input.close();
    labelOutput.setText("Name is " + user.getUserName() +
    ", password is: " +
    user.getUserPassword());
    }
   catch (java.io.IOException IOE) {
     labelOutput.setText("IOException");
   }
   catch (ClassNotFoundException cnfe) {
     labelOutput.setText("ClassNotFoundException");
   }
  }
```

Cuando compile y ejecute su proyecto, introduzca valores de Nombre y Contraseña y pulse el botón Serializar para almacenar la información en su disco. Después, pulse el botón Paralelizar para recuperar el objeto `UserInfo` serializado.

Figura 8.3 El objeto recuperado



Métodos de ObjectInputStream

`ObjectInputStream` también tiene métodos tales como `readDouble()`, `readFloat()`, etcétera, que son los equivalentes de `writeDouble()`, `writeFloat()`, y similares. Debe llamar a todos los métodos en secuencia de la misma forma que los objetos se escribieron en el flujo.

Lectura y escritura de flujos de objetos

Tal vez se haya preguntado qué sucede cuando un objeto que está serializando contiene un campo que remite a otro objeto, en lugar de a un tipo primitivo. En este caso, tanto el objeto base como el secundario se escribirán en el flujo. No deje de tener en cuenta, sin embargo, que ambos objetos escritos en el flujo necesitan implementar la interfaz `Serializable`. Si no, se lanzará una `NotSerializableException` cuando se llame al método `writeObject()`.

Recuerde que la serialización de objetos puede crear problemas potenciales de seguridad. En el ejemplo anterior, escribimos una contraseña en un objeto serializado. Aunque esta técnica puede ser aceptable en algunas circunstancias, tenga en mente el tema de la seguridad cuando decida serializar un objeto.

Finalmente, si desea crear un objeto persistente, pero no quiere utilizar el mecanismo de serialización por defecto, la interfaz `Serializable` documenta dos métodos, `writeObject()` y `readObject()`, que puede implementar para llevar a cabo una serialización a medida. La interfaz `Externalizable` también proporciona un mecanismo similar. Consulte la documentación JDK para obtener información sobre estas técnicas.

Capítulo

9

Introducción a la máquina virtual de Java

Este capítulo proporciona una introducción a la máquina virtual de Java (MVJ). Aunque es importante familiarizarse con información básica relativa a la MVJ, a menos que pretenda utilizar programación avanzada en Java, la máquina virtual no es algo de lo que deba preocuparse. Este capítulo se incluye únicamente a efectos informativos.

Antes de examinar la máquina virtual de Java, se explicarán algunos términos utilizados en este capítulo. En primer lugar, la máquina virtual de Java (MVJ) es el entorno en el que se ejecuta un programa Java. La especificación de la MVJ define, en esencia, un ordenador abstracto, y especifica las instrucciones que éste puede ejecutar. Estas instrucciones se denominan *bytecodes*. Generalmente hablando, los bytecodes de Java son a la MVJ lo que un conjunto de instrucciones es a una CPU (unidad central de proceso). Un bytecode es una instrucción de un byte de longitud generada por el compilador de Java y ejecutada por el intérprete de Java. Cuando el compilador compila un archivo .java, produce una serie de bytecodes que almacena en un archivo .class. El intérprete de Java puede entonces ejecutar los bytecodes almacenados en el archivo .class.

Otros términos que se utilizan en este capítulos son los de aplicaciones y applets Java. En ocasiones, resulta apropiado distinguir entre una aplicación Java y un applet Java. Sin embargo, en algunas secciones de este capítulo, esa distinción no es apropiada. En estos casos, se utilizará la palabra *aplicaciones* para referirse tanto a aplicaciones como a applets de Java.

Es importante aclarar aquí en qué consiste realmente Java. Java es algo más que un lenguaje de ordenador; es un *entorno* de ordenador. Esto es así porque Java se compone de dos elementos principales independientes,

cada uno de los cuales constituye una parte esencial de Java: el entorno de diseño de Java (el lenguaje Java propiamente dicho) y el entorno de ejecución de Java (la MVJ). Esta interpretación de la palabra *Java* es más técnica.

Curiosamente, la acepción más popular de la palabra *Java* es la que se refiere al entorno de ejecución, no al lenguaje. Cuando se dice algo como "esta máquina puede ejecutar Java", lo que realmente significa es que la máquina admite el entorno de ejecución de Java (JRE); más concretamente, que implementa una máquina virtual de Java.

Se debería hacer una distinción entre la *especificación de la máquina virtual de Java* y una *implementación* de la máquina virtual de Java. La especificación JVM es un documento (disponible en la página web de Sun) que define cómo implementar una MVJ. Cuando una implementación de la MVJ sigue correctamente la especificación, lo que hace, básicamente, es asegurar que las aplicaciones (o los applets) de Java pueden ejecutarse en esa implementación con los mismos resultados que producen cuando se ejecutan en cualquier otra implementación de la MVJ. La especificación de la MVJ garantiza que los programas Java podrán ejecutarse en cualquier otra plataforma.

La especificación de la MVJ es independiente de la plataforma, ya que puede implementarse en cualquiera. Sin embargo, observe que una implementación específica de la MVJ sí depende de la plataforma. Esto se debe a que la implementación de la MVJ es la única parte de Java que interactúa directamente con el sistema operativo del ordenador. Como cada sistema operativo es diferente, cualquier implementación específica de la MVJ debe saber cómo interactuar con el sistema operativo específico para la que está diseñada.

El hecho de que los programas Java se ejecuten bajo una implementación de la MVJ garantiza un entorno de ejecución predecible, ya que todas las implementaciones de la MVJ se ajustan a la especificación MVJ. Incluso aunque existen diferentes implementaciones de la MVJ, todas ellas deben cumplir ciertos requisitos para garantizar la portabilidad. En otras palabras, las diferencias entre las diversas implementaciones no deben afectar a la portabilidad.

La MVJ es la responsable de realizar las siguientes funciones:

- Asignar memoria para los objetos que se crean.
- Realizar la labor de liberación de memoria no utilizada.
- Manejar operaciones con los registros y la pila.
- Realizar llamadas al sistema de la máquina anfitriona para utilizar determinadas funciones, tales como las de acceso a dispositivos
- Supervisar la seguridad de las aplicaciones de Java.

Durante el resto del capítulo, se tratará la última función: seguridad.

Seguridad en la máquina virtual de Java

Una de las funciones más importantes de la MVJ es la de supervisar la seguridad de las aplicaciones Java. La MVJ utiliza un mecanismo específico para imponer ciertas restricciones de seguridad en las aplicaciones Java. Este mecanismo (o modelo de seguridad) realiza las siguientes funciones:

- Determina hasta qué punto el código en ejecución es "de confianza" y le asigna el nivel de acceso apropiado.
- Se asegura de que los bytecodes no realizan operaciones ilegales.
- Verifica que todos los bytecodes se generan correctamente.

En los siguientes apartados se describe cómo atiende Java estas labores de seguridad.

El modelo de seguridad

En este apartado, se examinarán algunos de los diferentes elementos del modelo de seguridad de Java. En particular, se examinarán las funciones del Verificador de Java (Java Verifier), el Administrador de seguridad, el paquete `java.security` y el Cargador de clases. Éstos son algunos de los componentes que hacen que las aplicaciones Java sean seguras.

El Verificador de Java

Cada vez que se carga una clase, primero debe seguir un proceso de verificación. La función principal de este proceso de verificación consiste en garantizar que ningún bytecode de la clase viola las especificaciones de la MVJ. Algunos ejemplos de bytecodes incorrectos son los errores de tipo así como el desbordamiento o subdesbordamiento de operaciones aritméticas. El proceso de verificación lo realiza el Verificador de Java, y consta de las siguientes cuatro etapas:

1. Verificar la estructura de los archivos de clases.
2. Realizar verificaciones relativas al sistema.
3. Validar bytecodes.
4. Realizar comprobaciones de tipos y accesos durante la ejecución.

La primera etapa del verificador se ocupa de comprobar la estructura del archivo de clases. Todos los archivos de clases comparten una estructura común; por ejemplo, deben empezar siempre con lo que se conoce como el número *mágico*, cuyo valor es `0xCAFEBABE`. En esta etapa, el verificador también comprueba que la agrupación de constantes no esté dañada (la agrupación de constantes es donde se almacenan cadenas y números).

Además, el verificador se asegura de que no existen bytes adicionales al final del archivo de clases.

La segunda etapa realiza verificaciones relacionadas con el sistema. Esto implica verificar la validez de todas las referencias a la agrupación de constantes, y asegurarse de que esas clases se derivan en subclases correctamente.

La tercera etapa se ocupa de validar los bytecodes. Esta etapa es la más significativa y compleja de todo el proceso de verificación. Validar un bytecode significa comprobar que su tipo es válido y sus argumentos tienen el número y el tipo apropiados. El verificador también comprueba que las llamadas a métodos reciben el tipo y número correcto de argumentos, y que cada función externa devuelve el tipo adecuado.

La etapa final es donde se realizan las comprobaciones relativas a la ejecución. En esta etapa, se cargan las clases con referencias externas, y se comprueban sus métodos. La comprobación de métodos implica comprobar que en las llamadas a métodos la firma coincide con la de los métodos en las clases externas. El verificador también supervisa los intentos de acceso que realizan las clases actualmente cargadas para asegurarse de que no violan ninguna restricción de acceso. También se realizan comprobaciones de acceso sobre las variables para garantizar que no se acceda de forma ilegal a variables de tipo `private` o `protected`.

De este exhaustivo proceso de verificación, se infiere la importancia del verificador de Java en el modelo de seguridad. También es importante observar que el proceso de verificación debe realizarse en el ámbito del verificador, y no en el del compilador, ya que cualquier compilador se puede programar para generar bytecodes de Java. Es evidente que confiar al compilador el proceso de verificación puede resultar peligroso, ya que el compilador puede programarse de modo que omita las verificaciones. Este punto ilustra por qué es necesaria la MVJ.

Si desea más información sobre el verificador de Java, consulte *Java Virtual Machine Specification*.

El Administrador de seguridad y el paquete java.security

Una de las clases definidas en el paquete `java.lang` es la clase `SecurityManager`. Esta clase comprueba los criterios de seguridad en las aplicaciones de Java para determinar si la aplicación en ejecución tiene permiso para realizar ciertas operaciones peligrosas. La función principal de la metodología de seguridad consiste en determinar los derechos de acceso. En Java 1.1, la clase `SecurityManager` era únicamente responsable de establecer las políticas de seguridad, pero en Java 2 y versiones posteriores se consigue un modelo de seguridad mucho más robusto y detallado por medio del nuevo paquete `java.security`. La clase `SecurityManager` dispone de varios métodos que empiezan por “check”. En Java 1.1, la implementación por defecto de esos métodos “check” consistía en lanzar

una excepción SecurityException. Desde Java 2, la implementación por defecto de la mayoría de los métodos "check" llama a SecurityManager.checkPermission(), y la implementación por defecto de este método llama, a su vez, a java.security.AccessController.checkPermission(). AccessController es el responsable del algoritmo real para comprobar permisos.

La clase SecurityManager contiene varios métodos que se utilizan para comprobar si una determinada operación está permitida. Los métodos checkRead() y checkWrite(), por ejemplo, comprueban si el llamante del método tiene derecho a ejecutar una operación de lectura o escritura, respectivamente, en un archivo especificado. Esto lo realizan mediante una llamada a checkPermission(), el cual, a su vez, llama a AccessController.checkPermission(). Muchos de los métodos del JDK utilizan SecurityManager antes de ejecutar operaciones peligrosas. El JDK hace esto por razones de compatibilidad; SecurityManager ya existía en versiones anteriores del JDK, cuando había un modelo de seguridad mucho más limitado. En las aplicaciones, puede ser necesario llamar a AccessController.checkPermission() directamente, en vez de utilizar la clase SecurityManager (la cual llama al mismo método indirectamente, de todas formas).

El método estático System.setSecurityManager() se puede utilizar para cargar el administrador de seguridad por defecto en el entorno. Desde ese momento, siempre que una aplicación Java necesita ejecutar una operación peligrosa, puede consultar con el objeto SecurityManager que está cargado en el entorno.

La manera en la que las aplicaciones Java utilizan la clase SecurityManager es la misma en la mayor parte de los casos. En primer lugar, se crea una instancia de SecurityManager, utilizando un argumento especial en la línea de comandos al iniciar la aplicación ("-Djava.security.manager"), o bien en el código, de forma similar al siguiente:

```
SecurityManager security = System.getSecurityManager();
```

El método System.getSecurityManager() devuelve una instancia del objeto SecurityManager cargado actualmente. Si no se ha definido ningún objeto SecurityManager mediante el método System.setSecurityManager(), System.getSecurityManager() devuelve **null**; en cualquier otro caso, devuelve una instancia del objeto SecurityManager que se cargó en el entorno. Supongamos ahora que la aplicación necesita comprobar si puede leer un archivo. Lo hará de la siguiente manera:

```
if (security != null) {
  security.checkRead (fileName);
}
```

La sentencia if comprueba primero si existe el objeto SecurityManager y, a continuación, realiza la llamada al método checkRead(). Si checkRead() no

permite la operación, se lanza una excepción `SecurityException` y la operación no se ejecuta; en caso contrario, se realiza con normalidad.

Habitualmente, suele haber un administrador de seguridad cargado cuando se está ejecutando un applet, ya que la mayoría de los navegadores que admiten Java utilizan uno automáticamente. Una aplicación (no un applet), por el contrario, no utiliza automáticamente un administrador de seguridad, a menos que se cargue alguno en el entorno mediante el método `System.setSecurityManager()`, o desde la línea de comandos cuando se inicia la aplicación. Para utilizar los mismos criterios de seguridad tanto para una aplicación como para un applet, debe asegurarse de que el administrador de seguridad está cargado.

Para especificar su propia metodología de seguridad, deberá trabajar con las clases del paquete `java.security`. Entre las clases más importantes de este paquete, se incluyen `Policy`, `Permission` y `AccessController`. No debería derivar `SecurityManager` en subclases, excepto como último recurso y con grandes precauciones. Una descripción en profundidad del paquete de `seguridad` queda fuera del propósito de este documento. La metodología de seguridad estándar debería bastar para la mayoría de los desarrolladores que se inician en Java. Si está interesado en temas de seguridad más avanzados, o simplemente desea más información del paquete `java.security`, consulte el documento “Security Architecture” de la documentación del JDK.

El cargador de clases

El cargador de clases funciona junto con el administrador de seguridad para supervisar la seguridad de las aplicaciones Java. Las funciones principales del cargador de clases se resumen a continuación:

- Determina si la clase que se está intentando cargar ya está cargada
- Carga archivos de clases en la máquina virtual
- Determina los permisos asignados a la clase cargada de acuerdo con la metodología de seguridad
- Proporciona información sobre clases cargadas en el administrador de seguridad
- Determina la vía de acceso desde la que se debería cargar la clase (las clases System se cargan siempre desde la vía de acceso especificada por `BOOTCLASSPATH`)

Cada instancia de una clase se asocia con un objeto cargador de clases, que es una instancia de una subclase de la clase abstracta `java.lang.ClassLoader`. La carga de clases se produce automáticamente al instanciar una clase. Es posible crear un cargador de clases personalizado derivando en subclases la clase `ClassLoader` o una de sus subclases; aunque, en la mayoría de los casos, no es necesario. Si necesita más información sobre el mecanismo de carga de clases, consulte la

documentación para `java.lang.ClassLoader` así como el documento "Security Architecture" de la documentación del JDK.

Hasta aquí, hemos visto como el verificador de Java, `SecurityManager` y el cargador de clases trabajan para garantizar la seguridad de las aplicaciones Java. Además de estos, existen otros mecanismos no descritos en este capítulo, tales como los incluidos en el paquete `java.security`, que contribuyen a la seguridad de las aplicaciones Java. Asimismo, existe una medida de seguridad integrada en el propio lenguaje Java, pero su descripción queda fuera del ámbito de este capítulo.

Compiladores "justo a tiempo" (JIT)

Resulta apropiado incluir una breve descripción de los compiladores "justo a tiempo" (JIT) en este capítulo. Los compiladores JIT traducen bytecodes de Java en instrucciones nativas de máquina que se pueden ejecutar directamente en la CPU. De esta forma, el rendimiento de las aplicaciones Java se incrementa enormemente. Pero, si se ejecutan instrucciones nativas en lugar de bytecodes, ¿qué ocurre con el proceso de verificación mencionado anteriormente? Realmente, el proceso de verificación no cambia, ya que el verificador de Java sigue verificando los bytecodes antes de que se traduzcan.

Capítulo

10

Utilización de la Interfaz nativa de Java (JNI)

Este capítulo explica cómo invocar métodos nativos en aplicaciones Java utilizando el Interfaz de Método Nativo Java (JNI). Comienza explicando cómo trabaja la JNI; a continuación, se explica la palabra clave `native` y cómo cualquier método Java puede convertirse en un método nativo. Finalmente, examina la herramienta de JDK, **javah**, que se utiliza para generar archivos de cabecera C para clases Java.

Aunque el código Java está diseñado para ejecutarse en múltiples plataformas, hay algunas situaciones en las que puede no ser suficiente por sí mismo. Por ejemplo:

- La biblioteca de clases estándar de Java no admite características específicas de su plataforma requeridas por su aplicación.
- Supongamos que quiere acceder a una biblioteca existente de otro lenguaje y hacerla accesible a su código Java.
- Tiene código que quiere implementar en un programa de bajo nivel, como ensamblador, y hacer que su aplicación Java lo llame.

La Interfaz Nativa de Java es una interfaz de programación multi-plataforma estándar incluido en el JDK. Le permite escribir programas Java que pueden operar con aplicaciones y bibliotecas escritas en otros lenguajes de programación, como C, C++, y ensamblador.

Utilizando JNI, puede escribir *métodos nativos Java* para crear, examinar, y actualizar objetos Java (incluyendo matrices y cadenas), llamar a métodos Java, capturar y lanzar excepciones, cargar clases y obtener información de ellas, y realizar comprobaciones en ejecución.

Además, puede utilizar la Invocación API para incluir la Maquina Virtual Java en sus aplicaciones nativas, y utilizar el puntero de la interfaz JNI

para acceder a las prestaciones de MV. Esto le permite hacer que aplicaciones existentes interpreten Java sin tener que enlazar con el código fuente de la máquina virtual (MV).

Cómo funciona JNI

Para conseguir la principal meta de Java, la independencia de plataforma, Sun no estandarizó su implementación de la Máquina virtual de Java; en otras palabras, Sun no quiere especificar rígidamente la arquitectura interna de MVJ, sino permitir a los fabricantes tener sus propias implementaciones de la MVJ. Esto no impide que Java sea independiente de la plataforma, porque cada implementación de la MVJ debe cumplir con ciertos estándares necesarios para conseguir la independencia de plataforma (tales como la estructura estándar de un archivo .class).

El único problema de este planteamiento consiste en que acceder a bibliotecas nativas desde aplicaciones Java se vuelve difícil, porque los sistemas de ejecución difieren entre las distintas implementaciones de MVJ. Por esta razón, Sun propuso la JNI como una manera estándar de acceder a bibliotecas nativas desde aplicaciones Java.

La forma en que se accede a los métodos nativos desde aplicaciones Java cambió en JDK 1.1. El sistema antiguo permitía a una clase Java acceder directamente a métodos en una biblioteca nativa. La nueva implementación utiliza JNI como una capa intermedia entre una clase Java y una biblioteca nativa. En lugar de que MVJ tenga que hacer llamadas directas a métodos nativos, MVJ utiliza un puntero a la JNI que hace las llamadas reales. De esta forma, aunque las implementaciones de MVJ sean diferentes, la capa que utilizan para acceder a los métodos nativos (JNI) es siempre la misma.

Utilización de la palabra clave native

Hacer métodos Java nativos es muy fácil. Lo que sigue es un sumario de los pasos necesarios:

1. Elimine el cuerpo del método.
2. Añada un punto y coma al final de la firma del método.
3. Añada la palabra clave `native` al principio de la firma del método.
4. Incluya el cuerpo del método en una biblioteca nativa que se vaya a cargar en la ejecución.

Por ejemplo, supongamos que el método siguiente existe en una clase Java:

```
public void nativeMethod () {
  //el cuerpo del método
}
```

Así es como el método se convierte en nativo:

```
public native void nativeMethod ();
```

Ahora que ha declarado el método como nativo, su implementación real se incluirá en una biblioteca nativa. Es tarea de la clase, de la que el método es un miembro, invocar a la biblioteca de manera que su implementación esté disponible globalmente. La forma más fácil de hacer que la clase invoque a la biblioteca es añadirle lo siguiente:

```
static
{
  System.loadLibrary (nombreDeBiblioteca);
}
```

Un bloque de código `estático` siempre se ejecuta una vez que se carga la clase por primera vez. Se puede incluir cualquier cosa en un bloque de código `estático`. No obstante, cargar bibliotecas es su uso más frecuente. Si, por alguna razón, la carga de la biblioteca falla, se activará una excepción `UnsatisfiedLineError` cuando se llame a un método de esa biblioteca. MVJ añadirá la extensión correcta a su nombre (.dll en Windows, y .so en UNIX). No tiene que especificarla en el nombre de la biblioteca.

Uso de la herramienta javah

JDK proporciona una herramienta llamada `javah` que se utiliza para generar archivos de cabecera C para clases Java. A continuación se expone la sintaxis general para utilizar `javah`:

```
javah [opciones] nombredeClase
```

`className` representa el nombre de la clase (sin la extensión) para la que se desea generar un archivo de cabecera C. Se puede especificar más de una clase en la línea de comandos. Para cada clase, `javah` añade un archivo .h en el directorio por defecto de clases. Para ubicar los archivos .h en otro directorio, utilice la opción `-o`. Si una clase está en un paquete, debe especificar el paquete junto con el nombre de la clase.

Por ejemplo, para generar un archivo de cabecera para la clase `Miclase` del paquete `Mipaquete`, haga lo siguiente:

```
javah Mipaquete.Miclase
```

El archivo de cabecera generado incluirá el nombre del paquete (`Mipaquete_Miclase.h`).

A continuación se ofrece una lista de algunas de las opciones de javah:

Opción	Descripción
-jni	Crea un archivo de cabecera JNI
-verbose	Muestra información sobre el avance
-version	Muestra la versión de javah
-o nombreDirectorio	Guarda el archivo .h en el directorio especificado
-classpath viadeAcceso	Redefine la vía de acceso de clases por defecto

El contenido del archivo .h generado por javah incluye todos los prototipos de funciones para los métodos nativos de la clase. Los prototipos se modifican para permitir al ejecutor de Java encontrar e invocar los métodos nativos. Esta modificación implica, básicamente, cambiar el nombre del método de acuerdo a una convención de denominación establecida para la invocación de métodos nativos. El nombre modificado adjunta el prefijo Java_ a los nombres de método y clase. Así, si tiene un método nativo llamado nativeMethod en una clase llamada Miclase, el nombre que aparece en el archivo Miclase.h es Java_Miclase_nativeMethod.

Para obtener más información sobre JNI, consulte:

- Java Native Interface en http://java.sun.com/j2se/1.3/docs/guide/jni/
- Java Native Interface Specification en http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html
- *El Tutorial de Java*, "Trail: Java Native Interface" en http://java.sun.com/docs/books/tutorial/native1.1/index.html

También están disponibles los siguientes libros sobre la Interfaz Nativa de Java:

- *The Java Native Interface: Programmer's Guide and Specification (Java Series)* de Sheng Liang
  - amazon.com
  - fatbrain.com
- *Essential Jni: Java Native Interface (Essential Java)*, de Rob Gordon
  - amazon.com
  - fatbrain.com

Capítulo

11

Guía de referencia rápida del lenguaje Java

Compatibilidad con plataformas de Java 2

La Plataforma Java 2 está disponible en varias ediciones utilizadas para propósitos diversos. Puesto que Java es un lenguaje que se puede ejecutar en cualquier parte y sobre cualquier plataforma, se utiliza en una amplia variedad de entornos y se han preparado de él distintas ediciones: Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE), y Java 2 Micro Edition (J2ME). En algunos casos, tales como en el desarrollo de aplicaciones para empresa, se utiliza un conjunto mayor de paquetes. En otros, como en los productos de electrónica de consumo, sólo se utiliza una pequeña porción del lenguaje. Cada edición contiene un Kit de Desarrollo de Software Java 2 (SDK), utilizado para desarrollar aplicaciones, y un Entorno de Ejecución Java 2 (JRE) utilizado para ejecutar aplicaciones.

Tabla 11.1 Ediciones de la plataforma Java 2

Compatibilidad con plataformas Java 2	Abreviatura	Descripción
Standard Edition	J2SE	Contiene las clases que son el núcleo del lenguaje Java.
Enterprise Edition	J2EE	Contiene las clases de J2SE y otras adicionales para el desarrollo de aplicaciones para empresa.
Micro Edition	J2ME	Contiene un subconjunto de las clases de J2SE, y se utiliza en productos de informática de consumo.

Bibliotecas de clase Java

Java, como la mayoría de los lenguajes de programación, se apoya de forma significativa en bibliotecas preexistentes para soportar determinadas funcionalidades. En el lenguaje Java, estos grupos de clases relacionadas, llamadas paquetes, varían según la edición de Java. Cada edición se utiliza para propósitos específicos, tales como aplicaciones, aplicaciones empresariales, y productos de consumo.

La Plataforma Java 2, Edición Estándar (J2SE) proporciona a los desarrolladores un entorno de desarrollo multi-plataforma seguro, estable y rico en prestaciones. Esta edición de Java soporta características fundamentales, tales como conectividad con bases de datos, diseño de interfaces de usuario, entrada/salida y programación en red, e incluye paquetes fundamentales del lenguaje Java. Algunos de estos paquetes de J2SE se relacionan en la tabla siguiente.

Tabla 11.2 Paquetes J2SE

Paquete	Nombre de Paquete	Descripción
Lenguaje	`java.lang`	Clases que contienen el núcleo principal del lenguaje Java.
Utilidades	`java.util`	Soporte para utilidades de estructuras de datos.
E/S	`java.io`	Soporte para diversos tipos de entrada/salida.
Texto	`java.text`	Soporte para el manejo localizado de texto, fechas, números y mensajes.
Matemáticas	`java.math`	Clases para realizar operaciones aritméticas con enteros y coma flotante.
AWT	`java.awt`	Diseño de interfaz de usuario y gestión de sucesos.
Swing	`javax.swing`	Clases para la creación de componentes ligeros 100% Java que se comportan de forma similar en todas las plataformas.
Javax	`javax`	Extensiones al lenguaje Java.
Applet	`java.applet`	Clases para la creación de applets.
Beans	`java.beans`	Clases para desarrollar JavaBeans.
Reflexión	`java.lang.reflect`	Clases utilizadas para obtener información de clases en tiempo de ejecución.
SQL	`java.sql`	Soporte para procesar y acceder a datos en bases de datos.
RMI	`java.rmi`	Soporte para programación distribuida.
Trabajo en red	`java.net`	Clases que soportan el desarrollo de aplicaciones en red.
Seguridad	`java.security`	Soporte para seguridad criptográfica.

Palabras clave de Java

Estas tablas cubren los siguientes tipos de palabras clave:

- Tipos y términos devueltos y de datos
- Paquetes, clases, miembros e interfaces
- Modificadores de acceso
- Bucles y controles de flujo
- Tratamiento de excepciones
- Reservadas

Tipos y términos devueltos y de datos

Tipos y términos	Palabra clave	Uso
Tipos numéricos	`byte`	un byte, entero.
	`double`	8 bytes, precisión doble.
	`float`	4 bytes, precisión única.
	`int`	4 bytes, entero.
	`long`	8 bytes, entero.
	`short`	2 bytes, entero.
	`strictfp`	(sugerido) Método o clase para utilizar la precisión estándar en cálculos intermedios de coma flotante.
Otros tipos	`boolean`	Valores Booleanos.
	`char`	16 bits, un caracter.
Términos devueltos	`return`	Abandona el bloque de código actual con los valores de retorno resultantes.
	`void`	Tipo devuelto cuando no se necesita valor devuelto.

Paquetes, clases, miembros e interfaces

Palabra clave	Uso
`abstract`	Este método o clase debe ampliarse para ser utilizado.
`class`	Declara una clase Java.
`extends`	Crea una subclase. Da acceso a clases a miembros públicos y protegidos de otras clases. Permite que una iterfaz de herencia a otra.
`final`	Esta clase no se puede amplicar.
`implements`	En una definición de clase, implementa y define una interfaz.
`import`	Hace que todas las clases de la clase o paquetes importados sean visibles en el programa actual.
`instanceof`	Comprueba la herencia de un objeto.

Palabra clave	Uso
`interface`	Abstrae una interfaz de clase de su implementación (dice lo que hace no cómo lo hace).
`native`	Un enlace a una biblioteca nativa proporciona el cuerpo de este método.
`new`	Instancia una clase.
`package`	Declara el nombre de un paquete para todas las clases definidas en los archivos fuente con la misma declaración de paquete.
`static`	El miembro se encuentra disponible para toda la clase, no sólo para un objeto.
`super`	Dentro de una subclase, hace referencia a la superclase.
`synchronized`	Convierte un bloque de códi en seguro para hilos.
`this`	Hace referencia al objeto actual.
`transient`	El valor de esta variable no persiste cuando se almacena el objeto. .
`volatile`	El valor de esta variable puede cambiar inesperadamente.

Modificadores de acceso

Palabra clave	Uso
`package`	Nivel de acceso por defecto. No lo use explícitamente. No puede ser subclase de otro paquete.
`private`	Acceso limitado a la propia clase de los miembros.
`protected`	Acceso limitado al paquete de la clase de los miembros.
`public`	Clase: accesible desde cualquier parte. Subclase: accesible mientras la clase sea accesible.

Bucles y controles de flujo

Tipo de sentencia	Palabra clave
Sentencias de selección	`if`
	`else`
	`switch`
	`case`
Sentencia Breakout	`break`
Sentencia Fallback	`default`
Sentencias de iteración	`for`
	`do`
	`while`
	`continue`

Tratamiento de excepciones

Palabra clave	Uso
`throw`	Transfiere el control del método al manejador de excepciones.
`throws`	Ofrece una lista de las excepciones que un método puede lanzar.
`try`	Sentencia de apertura del manejador de excepciones.
`catch`	Captura la excepción.
`finally`	Ejecuta su código antes de terminar el programa.

Reservadas

Palabra clave	Uso
`const`	Reservada para uso posterior.
`goto`	Reservada para uso posterior.

Conversión y modificación de tipos de datos

Una única operación puede alterar el tipo de datos de un objeto o variable cuando se necesita un tipo diferente. Las conversiones de ampliación (desde una clase o tipo de datos más pequeño a otro mayor) pueden ser implícitas, pero es aconsejable hacer las conversiones explícitamente. Las conversiones de reducción deben hacerse explícitamente. Los programadores inexpertos deberían evitar las conversiones de reducción; pueden ser una poderosa fuente de errores y confusiones.

Para conversiones de reducción, ponga el tipo *al* que quiere convertir entre paréntesis justo antes de la variable a convertir: `(int)x`. Así es como aparece en el contexto, en donde `x` es la variable a convertir, `float` el tipo de datos original, `int` el tipo de datos de destino, e `y` es la variable que almacena el nuevo valor:

```
float x = 1.00; //declara x como float
int y = (int)x; //declara x como entero y le asigna el valor del entero y
```

Se asume que el valor de `x` cabe en un tipo `int`. Observe que los valores decimales de `x` se pierden en la conversión. Java redondea a la baja hasta el número entero más cercano.

Recuerde que las secuencias Unicode pueden representar números, letras, símbolos o caracteres no imprimibles, tales como saltos de línea o tabuladores. Si desea más información sobre Unicode, consulte http://www.unicode.org/.

Este apartado contiene tablas de las siguientes conversiones:

- De primitivo a primitivo.
- De primitivo a cadena.
- De primitivo a referencia.
- De cadena a primitivo.
- De referencia a primitivo.
- De referencia a referencia.

De primitivo a primitivo

Java no admite modificaciones de tipo desde o hacia valores `booleanos`. Para trabajar con la tipificación lógica estricta de Java, se debe asignar un valor equivalente adecuado a la variable y convertirla. 0 y 1 se utilizan con frecuencia para representar valores `false` y `true`.

Sintaxis	Comentarios
De otro tipo primitivo `p` **A** `Boolean t`: `t = p != 0;`	Otros tipos primitivos incluye: `byte`, `short`, `char`, `int`, `long`, `double`, `float`.
De `Boolean t` **A** `byte b`: `b = (byte)(t ? 1 : 0);`	
De `Boolean t` **A** `int, long, double,` **or** `float m`: m = t ? 1 : 0;	
De `Boolean t` **A** `short s` s = (short) (t ? 1 : 0);	
De `Boolean t` **A** `byte b`: b = (byte) (t?1:0);	
De `Boolean t` **A** `char c`: c = (char) (t?'1':'0');	Se pueden omitir las comillas simples, pero su uso es recomendable.
De `short, char, int, long, double,` o `float n` **A** `byte b`: b = (byte)n;	
De `byte b` **A** `short, int, long, double,` o `float n`: n = b;	
De `byte b` **A** `char c`: c = (char)b;	

De primitivo a cadena

Los datos primitivos son convertibles a otros tipos de datos; los tipos de referencia son objetos inmutables. La conversión desde o hacia tipos de referencia es arriesgada.

Java no admite modificaciones de tipo desde o hacia valores `booleanos`. Para trabajar con la tipificación lógica estricta de Java, se debe asignar un valor equivalente adecuado a la variable y convertirla. 0 y 1 se utilizan con frecuencia para representar valores `false` y `true`.

Sintaxis	Comentarios
De `Boolean t` **A** `String gg`: `gg = t ? "true" : "false";`	

De `byte b`
A `String gg`:
`gg = Integer.toString(b);`
o bien
`gg = String.valueOf(b);`
	Los siguientes pueden sustituirse por `toString`, cuando resulte apropiado:
	`toBinaryString`
	`toOctalString`
	`toHexString`
	Donde está utilizando una base distinta a 10 o 2 (como 8):
	`gg = Integer.toString(b, 7);`

De `short` o `int n`
A `String gg`:
`gg = Integer.toString(n);`
o bien
`gg = String.valueOf(n);`
	Los siguientes pueden sustituirse por `toString`, cuando resulte apropiado:
	`toBinaryString`
	`toOctalString`
	`toHexString`
	Donde está utilizando una base distinta a 10 (como 8):
	`gg = Integer.toString(n, 7);`

De `char c` **A** `String gg`: `gg = String.valueOf(c);`	

De `long n`
A `String gg`:
`g = Long.toString(n);`
o bien
`gg = String.valueOf(n);`
	Los siguientes pueden sustituirse por `toString`, cuando resulte apropiado:
	`toBinaryString`
	`toOctalString`
	`toHexString`
	Donde está utilizando una base distinta a 10 o 2 (como 8):
	`gg = Integer.toString(n, 7);`

Sintaxis	Comentarios

De `float f`
A `String gg`:
gg = Float.toString(f);
o bien
gg = String.valueOf(f);
Para protección decimal o notación científica, vea la siguiente columna.
	Estas conversiones protegen más datos.
	Doble precisión:
	java.text.DecimalFormat df2
	= new java.text.DecimalFormat("###,##0.00");
	gg = df2.format(f);
	Notación científica (protege los exponentes) (JDK 1.2.x y posteriores):
	java.text.DecimalFormat de
	= new java.text.DecimalFormat("0.000000E00");
	gg = de.format(f);

De `double d`
A `String gg`:
gg = Double.toString(d);
o bien
gg = String.valueOf(d);
Para protección decimal o notación científica, vea la siguiente columna.
	Estas conversiones protegen más datos.
	Doble precisión:
	java.text.DecimalFormat df2
	= new java.text.DecimalFormat("###,##0.00");
	gg = df2.format(d);
	Notación científica (JDK 1.2.x y posteriores):
	java.text.DecimalFormat de
	= new java.text.DecimalFormat("0.000000E00");
	gg = de.format(d);

De primitivo a referencia

Java proporciona clases que equivalen a tipos de datos primitivos y métodos que facilitan las conversiones.

Recuerde que los tipos de datos primitivos pueden ser convertidos a otros tipos de datos; los tipos de referencia son objetos inmutables. La conversión desde o hacia tipos de referencia es arriesgada.

Java no admite modificaciones de tipo desde o hacia valores `booleanos`. Para trabajar con la tipificación lógica estricta de Java, se debe asignar un valor equivalente adecuado a la variable y convertirla. 0 y 1 se utilizan con frecuencia para representar valores `false` y `true`.

Sintaxis	Comentarios
De `Boolean t` **A** `Boolean tt`: tt = new Boolean(t);	
De tipo Primitivo `p` (distinto de `boolean`) **A** `Boolean tt` tt = new Boolean(p != 0); Para `char`, vea la siguiente columna.	Para `char c`, ponga comillas simples encerrando el cero: tt = new Boolean(c != '0');
De `Boolean t` **A** `Character cc`: cc = new Character(t ? '1' : '0');	
De `byte b` **A** `Character cc`: cc = new Character((char) b);	
De `char c` **A** `Character cc`: cc = new Character(c);	
De `short, int, long, float,` o `double n` **A** `Character cc`: cc = new Character((char)n);	
De `Boolean t` **A** `Integer ii`: ii = new Integer(t ? 1 : 0);	
De `byte b` **A** `Integer ii`: ii = new Integer(b);	
De`short, char,` o `int n` **A** `Integer ii`: ii = new Integer(n);	
De `long, float,` o `double f` **A** `Integer ii`: ii = new Integer((int) f);	
De `Boolean t` **A** `Long nn`: nn = new Long(t ? 1 : 0);	
De `byte b` **A** `Long nn`: nn = new Long(b);	

Sintaxis	Comentarios
De `short, char, int,` o `long s` **A** `Long nn`: nn = new Long(s);	
De `float, double f` **A** `Long nn`: nn = new Long((long)f);	
De `Boolean t` **A** `Float ff`: ff = new Float(t ? 1 : 0);	
De `byte b` **A** `Float ff`: ff = new Float(b);	
De `short, char, int, long, float,` o `double n` **A** `Float ff`: ff = new Float(n);	
De `Boolean t` **A** `Double dd`: dd = new Double(t ? 1 : 0);	
De `byte b` **A** `Double dd`: dd = new Double(b);	
De `short, char, int, long, float,` o `double n` **A** `Double dd`: dd = new Double(n);	

De cadena a primitivo

Recuerde que los tipos de datos primitivos pueden ser convertidos a otros tipos de datos; los tipos de referencia son objetos inmutables. La conversión desde o hacia tipos de referencia es arriesgada.

Java no admite modificaciones de tipo desde o hacia valores `booleanos`. Para trabajar con la tipificación lógica estricta de Java, se debe asignar un valor equivalente adecuado a la variable y convertirla. Los números 0 y 1, las cadenas "true" y "false", o valores igualmente intuitivos se utilizan aquí para representar los valores `true` y `false`.

Sintaxis	Comentarios
De `String gg` **A** `Boolean t`: `t = new Boolean(gg.trim()).booleanValue();`	**Advertencia:** `t` sólo será `true` cuando el valor de `gg` sea "`true`" (no distingue entre mayúsculas y minúsculas); si la cadena es "1", "yes", o cualquier otra afirmación, esta conversión devolverá un valor `false`.
De `String gg` **A** `byte b`: `try { b = (byte)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... }`	**Nota:** Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco. En el caso de bases distintas de 10, como por ejemplo 8. `try { b = (byte)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... }`
De `String gg` **A** `short s`: `try { s = (short)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... }`	**Nota:** Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco. En el caso de bases distintas de 10, como por ejemplo 8: `try { s = (short)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... }`
De `String gg` **A** `char c`: `try { c = (char)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... }`	**Nota:** Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco. En el caso de bases distintas de 10, como por ejemplo 8: `try { c = (char)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... }`

Sintaxis	Comentarios

De `String gg`
A `int i`
 	
try {
i = Integer.parseInt(gg.trim());
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	En el caso de bases distintas de 10, como por ejemplo 8:
	
	try {
	i = Integer.parseInt(gg.trim(), 7);
	}
	catch (NumberFormatException e) {
	...
	}

De `String gg`
A `long n`:
 	
try {
n = Long.parseLong(gg.trim());
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.

De `String gg`
A `float f`:
 	
try {
f = Float.valueOf(gg.trim()).floatValue;
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	Para JDK 1.2.x o posterior:
	
	try {
	f = Float.parseFloat(gg.trim());
	}
	catch (NumberFormatException e) {
	...
	}

De `String gg`
A `double d`:
 	
try {
d = Double.valueOf(gg.trim()).doubleValue;
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	Para JDK 1.2.x o posterior:
	
	try {
	d = Double.parseDouble(gg.trim());
	}
	catch (NumberFormatException e) {
	...
	}

De referencia a primitivo

Java proporciona clases que equivalen a los tipos de datos primitivos. Esta tabla muestra cómo convertir una variable de una de estas clases a un tipo de datos primitivo para una única operación.

Para convertir de un tipo de referencia a uno primitivo, primero debe obtener el valor de la referencia como primitivo y, entonces, convertir el primitivo.

Los datos primitivos son convertibles a otros tipos de datos; los tipos de referencia son objetos inmutables. La conversión desde o hacia tipos de referencia es arriesgada.

Java no admite modificaciones de tipo desde o hacia valores `booleanos`. Para trabajar con la tipificación lógica estricta de Java, se debe asignar un valor equivalente adecuado a la variable y convertirla. 0 y 1 se utilizan con frecuencia para representar valores `false` y `true`.

Sintaxis	Comentarios
De `Boolean tt` **A** `Boolean t`: t = tt.booleanValue();	
De `Boolean tt` **A** `byte b`: b = (byte)(tt.booleanValue() ? 1 : 0);	
De `Boolean tt` **A** `short s`: s = (short)(tt.booleanValue() ? 1 : 0);	
De `Boolean tt` **A** `char c`: c = (char)(tt.booleanValue() ? '1' : '0');	Puede omitir las comillas simples, pero se recomienda su uso.
De `Boolean tt` **A** `int, long, float,` o `double n`: n = tt.booleanValue() ? 1 : 0);	
De `Character cc` **A** `Boolean t`: t = cc.charValue() != 0;	
De `Character cc` **A** `byte b`: b = (byte)cc.charValue();	
De `Character cc` **A** `short s`: s = (short)cc.charValue();	
De `Character cc` **A** `char, int, long, float` o `double n`: n = cc.charValue();	
De `Integer ii` **A** `Boolean t`: t = ii.intValue() != 0;	
De `Integer ii` **A** `byte b`: b = ii.byteValue();	

Sintaxis	Comentarios
De `Integer, Long, Float,` o `Double nn` **A** `short s`: `s = nn.shortValue();`	
De `Integer, Long, Float` o `Double nn` **A** `char c`: `c = (char)nn.intValue();`	
De `Integer, Long, Float` o `Double nn` **A** `int i`: `i = nn.intValue();`	
De `Integer ii` **A** `long n`: `n = ii.longValue();`	
De `Long, Float` o `Double dd` **A** `long n`: `n = dd.longValue();`	
De `Integer, Long, Float` o `Double nn` **A** `float f`: `f = nn.floatValue();`	
De `Integer, Long, Float` o `Double nn` **A** `double d`: `d = nn.doubleValue();`	

De referencia a referencia

Java proporciona clases que equivalen a los tipos de datos primitivos. Esta tabla muestra cómo convertir una variable de una de estas clases a otra para una única operación.

Nota Para conversiones de clase a clase válidas, aparte de las mostradas aquí, las conversiones de ampliación son implícitas. Las conversiones de ampliación utilizan esta sintaxis:

```
castFromObjectName = (CastToObjectClass)castToObjectName;
```

Debe convertir entre clases que estén en la misma jerarquía de herencia. Si convierte un objeto a una clase incompatible, lanzará una excepción `ClassCastException`.

Los tipos de referencia son objetos inmutables. La conversión entre tipos de referencia es arriesgada.

Sintaxis	Comentarios
De `String gg` **A** `Boolean tt`: tt = new Boolean(gg.trim());	**Nota:** Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco. Alternativa: tt = Boolean.valueOf(gg.trim());
De `String gg` **A** `Character cc`: cc = new Character(g.charAt(<index>);	

De `String gg`
A `Integer ii`:
 	
try {
ii = new Integer(gg.trim());
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	Alternativa:
	
	try {
	ii = Integer.valueOf(gg.trim());
	}
	catch (NumberFormatException e) {
	...
	}

De `String gg`
A `Long nn`:
 	
try {
nn = new Long(gg.trim());
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	Alternativa:
	
	try {
	nn = Long.valueOf(gg.trim());
	}
	catch (NumberFormatException e) {
	...
	}

De `String gg`
A `Float ff`:
 	
try {
ff = new Float(gg.trim());
}
catch (NumberFormatException e) {
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	Alternativa:
	
	try {
	ff = Float.valueOf(gg.trim());
	}
	catch
	...
	}

De `String gg`
A `Double dd`:
 	
try {
dd = new Double(gg.trim());
}
catch
...
}
	Nota: Si el valor de `gg` es null, `trim()` lanzará una `NullPointerException`. Si no utiliza `trim()`, asegúrese de que no queda ningún espacio en blanco.
	Alternativa:
	
	try {
	dd = Double.valueOf(gg.trim());
	}
	catch (NumberFormatException e) {
	...
	}

Sintaxis	Comentarios
De `Boolean tt` **A** `Character cc`: cc = new Character(tt.booleanValue() ?'1':'0');	
De `Boolean tt` **A** `Integer ii`: ii = new Integer(tt.booleanValue() ? 1 : 0);	
De `Boolean tt` **A** `Long nn`: nn = new Long(tt.booleanValue() ? 1 : 0);	
De `Boolean tt` **A** `Float ff`: ff = new Float(tt.booleanValue() ? 1 : 0);	
De `Boolean tt` **A** `Double dd`: dd = new Double(tt.booleanValue() ? 1 : 0);	
De `Character cc` **A** `Boolean tt`: tt = new Boolean(cc.charValue() != '0');	
De `Character cc` **A** `Integer ii`: ii = new Integer(cc.charValue());	
De `Character cc` **A** `Long nn`: nn = new Long(cc.charValue());	
De cualquier clase `rr` **A** `String gg`: gg = rr.toString();	

De `Float ff`
A `String gg`:
gg = ff.toString();
	Estas variaciones protegen más datos.
	Doble precisión:
	java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00");
	gg = df2.format(ff.floatValue());
	Notación científica (JDK 1.2.x en adelante):
	java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00");
	gg = de.format(ff.floatValue());

Sintaxis	Comentarios

De `Double dd`
A `String gg`:
gg = dd.toString();
	Estas variaciones protegen más datos.
	Doble precisión:
	java.text.DecimalFormat df2
	= new java.text.DecimalFormat("###,##0.00");
	gg = df2.format(dd.doubleValue());
	Notación científica (JDK 1.2.x en adelante):
	java.text.DecimalFormat de
	= new
	java.text.DecimalFormat("0.0000000000E00");
	gg = de.format(dd.doubleValue());

De `Integer ii` **A** `Boolean tt`: tt = new Boolean(ii.intValue() != 0);	
De `Integer ii` **A** `Character cc`: cc = new Character((char)ii.intValue());	
De `Integer ii` **A** `Long nn`: nn = new Long(ii.intValue());	
De `Integer ii` **A** `Float ff`: ff = new Float(ii.intValue());	
De `Integer ii` **A** `Double dd`: dd = new Double(ii.intValue());	
De `Long nn` **A** `Boolean tt`: tt = new Boolean(nn.longValue() != 0);	
De `Long nn` **A** `Character cc`: cc = new Character((char)nn.intValue());	**Nota:** Algunos valores Unicode pueden ser presentados como caracteres no imprimibles. Consulte `http://`www.unicode.org/
De `Long nn` **A** `Integer ii`: ii = new Integer(nn.intValue());	
De `Long nn` **A** `Float ff`: ff = new Float(nn.longValue());	

Sintaxis	Comentarios
De `Long nn` **A** `Double dd`: dd = new Double(nn.longValue());	
De `Float ff` **A** `Boolean tt`: tt = new Boolean(ff.floatValue() != 0);	
De `Float ff` **A** `Character cc`: cc = new Character((char)ff.intValue());	**Nota:** algunos valores Unicode pueden ser presentados como caracteres no imprimibles. Consulte `http://`www.unicode.org/
De `Float ff` **A** `Integer ii`: ii = new Integer(ff.intValue());	
De `Float ff` **A** `Long nn`: nn = new Long(ff.longValue());	
De `Float ff` **A** `Double dd`: dd = new Double(ff.floatValue());	
De `Double dd` **A** `Boolean tt`: tt = new Boolean(dd.doubleValue() != 0);	
De `Double dd` **A** `Character cc`: cc = new Character((char)dd.intValue());	**Nota:** algunos valores Unicode pueden ser presentados como caracteres no imprimibles. Consulte `http://`www.unicode.org/
De `Double dd` **A** `Integer ii`: ii = new Integer(dd.intValue());	
De `Double dd` **A** `Long nn`: nn = new Long(dd.longValue());	
De `Double dd` **A** `Float ff`: ff = new Float(dd.floatValue());	

Secuencias de escape

Un carácter octal se representa por una secuencia de tres dígitos octales, y un carácter Unicode se representa por una secuencia de cuatro dígitos hexadecimales. Los caracteres octales van precedidos de la marca de escape estándar, \, y los caracteres Unicode van precedidos por \u. Por ejemplo, el número decimal 57 se representa con el código octal \071 y con la secuencia Unicode \u0039. Las secuencias Unicode pueden representar números, letras, símbolos o caracteres no imprimibles, tales como retornos de línea o tabulaciones. Si desea más información sobre Unicode, consulte http://www.unicode.org/.

Carácter	Secuencia de Escape
Barra invertida	\\
Retroceso	\b
Retorno de carro	\r
Comilla doble	\"
Salto de página	\f
Tabulación horizontal	\t
Nueva línea	\n
Carácter octal	\DDD
Comilla simple	\'
Carácter Unicode	\uHHHH

Operadores

Este apartado incluye lo siguiente:

- Operadores básicos
- Operadores aritméticos
- Operadores lógicos
- Operadores de asignación
- Operadores de comparación
- Operadores bit a bit
- Operador ternario

Operadores básicos

Operador	Operando	Comportamiento
.	objeto miembro	Accede a un miembro del objeto.
(<type>)	tipo de datos	Convierte una variable a un tipo de datos diferente. [1]
+	cadena	Une cadenas (concatenador).
	número	Suma.

Operador	Operando	Comportamiento
-	número	El menos unario[2] (invierte el signo del número).
	número	Resta.
!	boolean	El operador NOT `booleano`.
&	entero, booleano	El operador AND, tanto `booleano` como bit a bit (entero). Cuando aparece doble (`&&`), es el AND condicional `booleano`.
=	La mayoría de los elementos con variables	Asigna un elemento a otro (por ejemplo, un valor a una variable, o una clase a una instancia). Puede combinarse con otros operadores para realizar otra operación y asignar el valor resultante. Por ejemplo, `+=` añade el valor de la izquierda a la derecha y después asigna el nuevo valor al lado derecho de la expresión.

1. Es importante distinguir entre un operador y un delimitador. Los paréntesis se utilizan con los argumentos (por ejemplo), como delimitadores que delimitan los argumentos de la sentencia. Se utilizan alrededor de un tipo de datos como operadores que cambian el tipo de datos de una variable al indicado entre los paréntesis.
2. Un *operador unario* afecta a un solo operando, un *operador binario* afecta a dos operandos, y un *operador ternario* afecta a tres operandos.

Operadores aritméticos

Operador	Prec.	Asoc.	Definición
`++/--`	1	Derecha	Auto-incremento/decremento: Añade uno a o resta uno de su único operando. Si el valor de `i` es 4, `++i` es 5. Un `pre-incremento` (`++i`) incrementa el valor en uno y asigna el nuevo valor a la variable. Un `post-incremento` (`i++`) incrementa el valor, pero deja la variable con el valor original.
`+/-`	2	Derecha	Más/menos unario: asigna o cambia el valor positivo/negativo de un número.
*	4	Izquierda	Multiplicación.
/	4	Izquierda	División.
%	4	Izquierda	Modulus: Divide el primer operando por el segundo y devuelve el resto (no el resultado).
`+/-`	5	Izquierda	Suma/resta

Operadores lógicos

Operador	Prec.	Asoc.	Definición
`!`	2	Derecha	El `NOT` booleano (unario) Cambia `true` a `false` o `false` a `true`. Debido a su baja precedencia, puede necesitar utilizar paréntesis encerrando esta sentencia.
`&`	9	Izquierda	Evaluación `AND` (binaria) Da como resultado `true` sólo si ambos operandos son `true`. Siempre evalúa ambos operandos.
`^`	10	Izquierda	Evaluación `XOR` (binaria) Da como resultado `true` solo si ambos operandos son `true`. Evalua ambos operandos.
`|`	11	Izquierda	Evaluación `OR` (binaria) Devuelve `true` si uno o ambos operandos son `true`. Evalúa ambos operandos.
`&&`	12	Izquierda	`AND` condicional (binario) Da como resultado `true` sólo si ambos operandos son `true`. Llamado "condicional" porque sólo evalúa el segundo operando si el primero es `true`.
`||`	13	Izquierda	`OR` condicional (binario) Devuelve `true` si uno o ambos operandos son `true`; devuelve `false` si ambos son `false`. No evalúa el segundo operador si el primero es `true`.

Operadores de asignación

Operador	Prec.	Asoc.	Definición
`=`	15	Derecha	Asigna el valor de la derecha a la variable de la izquierda.
`+=`	15	Derecha	Añade el valor de la derecha al valor de la variable de la izquierda; asigna el nuevo valor a la variable original.
`-=`	15	Derecha	Resta el valor de la derecha del valor de la izquierda; asigna el nuevo valor a la variable original.
`*=`	15	Derecha	Multiplica el valor de la derecha con el valor de la variable de la izquierda; asigna el nuevo valor a la variable original.
`/=`	15	Derecha	Divide el valor de la derecha por el valor de la variable de la izquierda; asigna el nuevo valor a la variable original.

Operadores de comparación

Operador	Prec.	Asoc.	Definición
<	7	Izquierda	Menor que
>	7	Izquierda	Mayor que
<=	7	Izquierda	Menor o igual que
>=	7	Izquierda	Mayor o igual que
==	8	Izquierda	Igual a
!=	8	Izquierda	No igual a

Operadores bit a bit

Nota Un entero con signo es uno cuyo primer bit de la izquierda se utiliza para indicar el signo positivo o negativo del entero: el bit es 1 si el entero es negativo, 0 si es positivo. En Java, los enteros siempre tienen signo, mientras que en C/C++ tienen signo sólo por defecto. In Java, los operadores de desplazamiento preservan el bit de signo, de forma que el bit de signo se duplica y, a continuación, se desplaza. Por ejemplo, el desplazar 10010011 a la derecha 1 bit es 11001001.

Operador	Prec.	Asoc.	Definición
~	2	Derecha	NOT bit a bit Invierte cada bit del operando, de forma que cada 0 se convierte en 1 y viceversa.
<<	6	Izquierda	Desplazamiento a la izquierda con signo Desplaza los bits del operando de la izquierda hacia la izquierda el número de dígitos especificado en el operando de la derecha, añadiendo ceros por la derecha. Los bits de orden superior se pierden.
>>	6	Izquierda	Desplazamiento a la derecha con signo Desplaza hacia la derecha los bits del operando de la izquierda el número de dígitos especificado a la derecha. Si el operando de la izquierda es negativo, se completa con ceros a la izquierda; si es positivo, con unos. De esta forma se preserva el signo original.
>>>	6	Izquierda	Desplazamiento a la derecha rellenando con ceros Desplaza a la derecha, pero siempre rellena con ceros.
&	9	Izquierda	AND bit a bit Pueden utilizarse con = para asignar el valor.
|	10	Izquierda	OR bit a bit Pueden utilizarse con = para asignar el valor.

Operador	Prec.	Asoc.	Definición
^	11	Izquierda	XOR bit a bit Pueden utilizarse con = para asignar el valor.
<<=	15	Izquierda	Desplazamiento a la izquierda con asignación
>>=	15	Izquierda	Desplazamiento a la derecha con asignación
>>>=	15	Izquierda	Desplazamiento a la derecha con asignación y relleno a ceros

Operador ternario

El operador ternario ?: realiza una operación if-then-else muy sencilla en una sola sentencia. Si el primer término es verdadero, evalúa el segundo; si el segundo término es falso, utiliza el tercero. La sintaxis es como sigue:

```
<expresión 1> ? <expresión 2> : <expresión 3>;
```

Por ejemplo:

```
int x = 3, y = 4, max;
max = (x > y) ? x : y;
```

Esta operación asigna a max el valor de la que sea mayor, x o y.

Capítulo

12

Cómo aprender más sobre Java

Existe gran cantidad de recursos sobre el lenguaje Java. La sede web de Sun Microsystems, `http://java.sun.com`, es un buen lugar para buscar más información; encontrará muchos enlaces interesantes. Si es usted nuevo en Java, le resultará especialmente conveniente consultar en tutorial en línea sobre Java de Sun en `http://java.sun.com/docs/books/tutorial/`.

Glosarios en línea

Para obtener rápidamente definiciones de los términos Java, consulte uno de los glosarios en línea de Sun:

- *Glosario de Java* en HTML de Sun Microsystem: `http://java.sun.com/docs/glossary.nonjava.html#top`
- *Glosario de Java* en HTML de Sun Microsystem: `http://java.sun.com/docs/glossary.html`

Libros

Existen muchos libros excelentes sobre la programación en Java. Para ver una lista de los títulos sobre Java en la red de desarrolladores de Borland, diríjase a `http://bdn.borland.com/books/java/0,1427,c|3,00.html`.

Sun también publica sus series de libros Java. Consulte los títulos en `http://java.sun.com/docs/books/`. Encontrará libros para todos los niveles.

Además puede encontrar libros sobre Java en su tienda de libros preferida; debe simplemente escribir `libros sobre Java` en su portal web de búsqueda preferido para encontrar una lista de libros recomendados por programadores de Java o que están ofreciendo los editores.

Índice

Símbolos

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

V

W