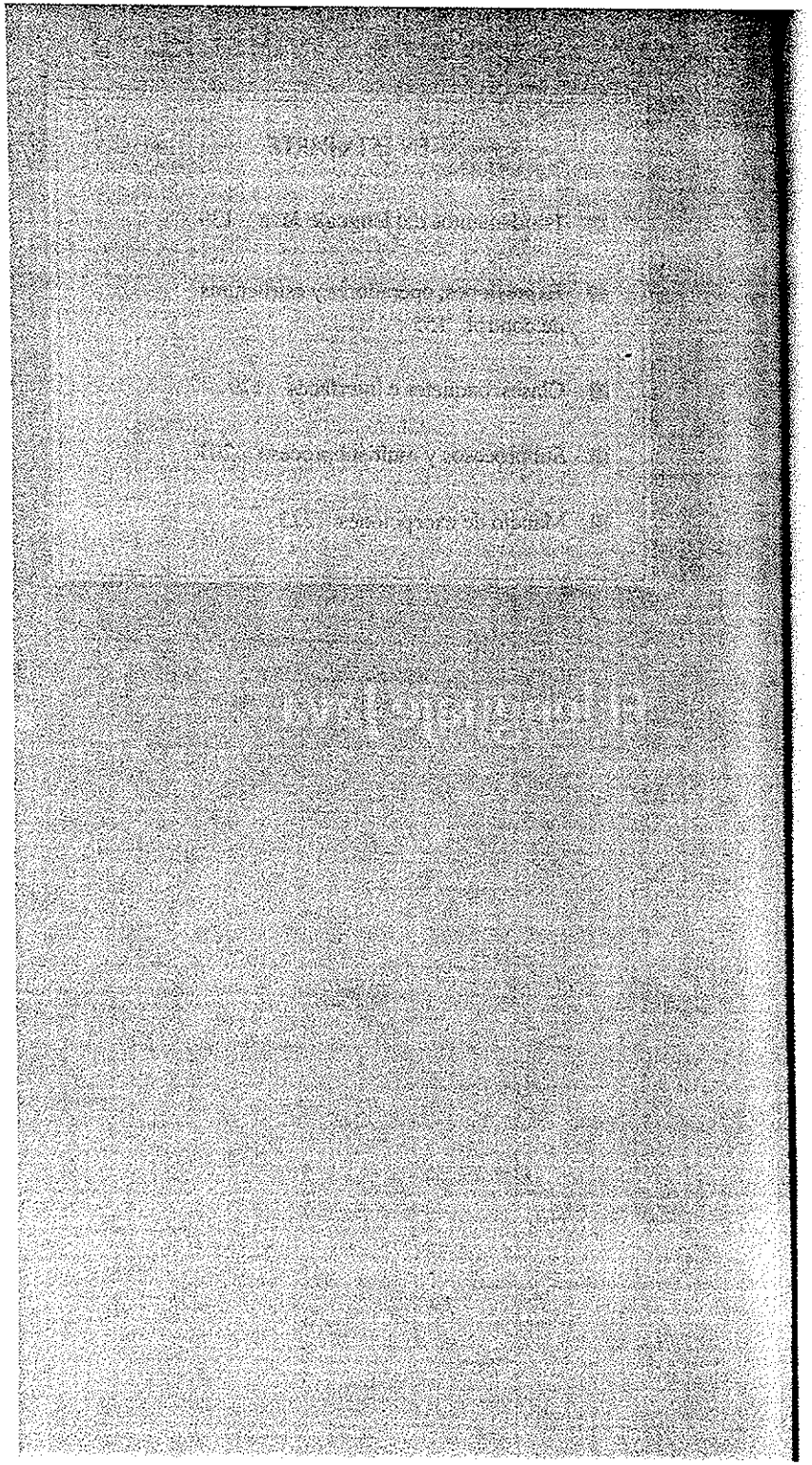


Capítulo 12

Fundamentos del lenguaje Java

por Michael Morrison

Contenido del capítulo

- Hello, World! 139
- Tokens 141
- Tipos de datos 146
- Conversión de tipos de datos 148
- Bloques y ámbito 149
- Matrices 152
- Cadenas 153

Java es un lenguaje orientado a objetos. Esto significa que el lenguaje está basado en el concepto objeto. Aunque es necesario un conocimiento de programación orientada a objetos para un uso práctico de Java, no se requiere comprender los fundamentos del lenguaje Java. Este capítulo se centra en el lenguaje y deja los detalles orientados a objetos de Java para el capítulo 14, "Clases, paquetes e interfaces".

Si ya tiene alguna experiencia con otro lenguaje orientado a objetos como el C++ o el Smalltalk, gran parte de Java le será territorio familiar. En efecto, Java casi puede considerarse un C++ nuevo y modernizado. Debido a que Java se deriva en tal alto grado de C++, se destacarán a lo largo de los próximos capítulos muchas de las similitudes y diferencias entre Java y C++. Además, el apéndice D proporciona una visión más completa de estas diferencias.

Este capítulo cubre lo esencial del lenguaje Java, incluyendo algunos programas de ejemplo para ayudarle a avanzar con seguridad.

Hello, World!

La mejor forma de aprender un lenguaje de programación es lanzarse y ver cómo funciona un programa real. Manteniendo el ejemplo de programación introductorio tradicional, su primer programa será una versión Java del clásico programa "Hello, World!". El listado 12.1 contiene el código fuente de la clase *HelloWorld*, que también se encuentra en el CD-ROM en el archivo *HelloWorld.java*.

Nota

Puede estar pensando que ya dijo hola al mundo con Java en el capítulo 2, "El diseño de Java es flexible y dinámico". El programa Hello World que vio en el capítulo 2 era una miniaplicación Java, es decir, que se ejecutaba dentro de los confines de una página Web e imprimía texto como salida gráfica. El programa HelloWorld de este capítulo es una aplicación Java, lo que significa que se ejecuta dentro del intérprete de Java como un programa autónomo y ofrece salida de texto.

Listado 12.1 La clase HelloWorld

```
class HelloWorld {
  public static void main (String args[]) {
    System.out.println("Hello, World!");
  }
}
```

Si ha compilado el programa con el compilador de Java (*javac*), ya está preparado para ejecutarlo en el intérprete de Java. El compilador de Java coloca la salida ejecutable en un archivo llamado *HelloWorld.class*. Esta convención de denominación puede parecer extraña considerando el hecho de que la mayoría de lenguajes de programación usan la extensión de archivo .EXE para los ejecutables. ¡No ocurre así en Java! Siguiendo la naturaleza orientada a objetos de Java, todos los programas Java se almacenan como clases Java creables y ejecutables como objetos en el entorno de tiempo de ejecución Java. Para ejecutar el programa HelloWorld, escriba *java HelloWorld* en el indicador de comandos. Como habrá supuesto, el programa responde mostrando "Hello, World!" en su pantalla. ¡Felicidades, acaba de escribir y probar su primer programa Java!

HelloWorld es un programa Java muy pequeño. No obstante, ocurren muchas cosas en estas pocas líneas de código. Para comprender lo que está sucediendo, ha de examinar el programa línea a línea. Primero, ha de entender que Java depende mucho de las clases. En efecto, la primera sentencia de HelloWorld le recuerda que éste es una clase, no tan sólo un programa. Además, mirando detenidamente a la sentencia *class*, el nombre de la clase se define como *HelloWorld*. Este nombre lo usa el compilador de Java como el nombre de la clase de salida ejecutable. El compilador de Java crea un archivo de clase ejecutable por cada clase definida en un archivo fuente de Java. Si se define más de una clase en un archivo *.java*, el compilador de Java almacenará cada una de ellas en un archivo *.class* separado. No es estrictamente necesario dar a un archivo fuente el mismo nombre que el del archivo de clase, pero es muy recomendable hacerlo como una norma de estilo.

La clase *HelloWorld* contiene un *método* o función miembro. Por ahora, puede imaginarse esta función como una función de procedimiento normal que da la casualidad que está enlazada a la clase. Los detalles de los métodos están cubiertos en el capítulo 14, "Clases, paquetes e interfaces". El método único de la clase *HelloWorld* se llama *main* y debería serle familiar si ha usado C o C++. El método *main* se define como *public static* con un tipo de retorno *void*. *public* significa que puede llamarse al método desde cualquier lugar dentro o fuera de la clase. *static* significa que el método es el mismo para todas los casos de la clase. El tipo de retorno *type* significa que *main* no devuelve ningún valor.

El método *main* está definido para que tome un solo parámetro, *String args[]*. *args* es una matriz de objetos *String* que representa argumentos de línea de comandos pasados a la clase durante la ejecución. Debido a que HelloWorld no usa ningún argumento de línea de comandos, puede ignorar el parámetro *args*. Aprenderá algo más sobre cadenas más adelante en este capítulo.

Se llama al método *main* cuando se ejecuta la clase *HelloWorld*. *main* se compone de una sola sentencia que imprime el mensaje "Hello, World!" en el flujo de salida estándar, de la forma siguiente:

```
System.out.println("Hello, World!");
```

Al principio, esta sentencia puede parecer un poco confusa por los objetos anidados. Para aclarar las cosas, examine la sentencia de derecha a izquierda. Primero observe que la sentencia acaba con un punto y coma, lo que es sintaxis estándar de Java, tomada de C/C++. Hacia la izquierda, ve la cadena "Hello, World!" entre paréntesis, lo que significa que es un parámetro para una llamada a una función. El método llamado es realmente el *println* del objeto *out*. Este método es similar al método *printf* de C, excepto en que añade automáticamente una *newline* (\n) al final de la cadena. El objeto *out* es una variable miembro del objeto *System* que representa el flujo de salida estándar. Por fin, el objeto *System* es un objeto global en el entorno Java que encapsula la funcionalidad del sistema.

Esto cubre muy bien la clase *HelloWorld*, su primer programa de Java. Si se extravío durante la explicación de la clase *HelloWorld*, no se preocupe demasiado. Se le ha presentado HelloWorld sin ninguna explicación previa del lenguaje Java y con la intención de que se lanzara dentro del código Java. El resto de este capítulo se centra en una descripción más estructurada de los fundamentos del lenguaje Java.

Tokens

Cuando envía un programa Java al compilador de Java, éste analiza el texto y extrae *tokens* individuales. Un token es el elemento más pequeño de un programa significativo para el compilador. Realmente esto es cierto para todos los compiladores, no sólo para el de Java. Estos tokens definen la estructura del lenguaje Java, todos ellos se conocen como el *conjunto de tokens* de Java. Pueden subdividirse en cinco categorías: identificadores, palabras clave, literales, operadores y separadores. El compilador de Java también reconoce y elimina posteriormente los comentarios y los espacios en blanco.

El compilador de Java elimina todos los comentarios y espacios en blancos mientras descompone en tokens el código fuente. Los tokens resultantes se compilan a bytecode Java independiente del sistema, ejecutable desde dentro de un entorno Java interpretado. El bytecode se ajusta al Sistema Virtual Java hipotético, que abstrae las diferencias de procesador a un procesador virtual único. Para obtener más información sobre el Sistema Virtual Java, consulte el capítulo 39, "Java's Virtual Machine, Bytecodes, and More". Tenga presente que un entorno Java interpretado puede ser el interpretador de línea de comandos Java o el programa de navegación apropiado para Java.

Identificadores

Los *identificadores* son tokens que representan nombres, asignables a variables, métodos y clases para identificarlos de forma única ante el compilador y darles nombres

con sentido para el programador. *HelloWorld* es un identificador que asigna este nombre a la clase que reside en el archivo fuente *HelloWorld.java*, desarrollada antes.

Aunque puede ser creativo, al dar nombres a los identificadores de Java hay algunas limitaciones. Todos los identificadores de Java diferencian mayúsculas de minúsculas y deben comenzar con una letra, un subrayado (_) o un símbolo de dólar ($). Las letras incluyen mayúsculas y minúsculas. Los caracteres posteriores del identificador pueden incluir las cifras 0 a 9. La otra única limitación a los nombres de identificadores es que no pueden usarse las palabras clave de Java, que se listan en la sección siguiente. La tabla 12.1 contiene una lista de nombres de identificadores válidos y no válidos.

Tabla 12.1 Identificadores de Java válidos y no válidos

Válido	Inválido
HelloWord	Hello Word
Hi_Mom	Hi_Mom!
heyDude3	3heyDude
tall	short
poundage	#age

El identificador *HelloWorld* es inválido porque contiene un espacio, *Hi_Mom!*, porque contiene un signo de exclamación, *3heyDude* porque comienza con una cifra, *short*, porque es una palabra clave de Java, por último, *#age*, porque comienza con el símbolo *#*.

Además de las restricciones mencionadas en cuanto a la denominación de los identificadores de Java, hay algunas reglas de estilo que debería seguir para hacer que la programación Java sea más fácil y consistente. Es una práctica estándar de Java denominar los identificadores con varias palabras en minúsculas, excepto la primera letra de cada palabra en medio del nombre. Por ejemplo, la variable *toughGuy* está en correcto estilo Java, mientras que *toughguy*, *ToughGuy* y *TOUGHGUY* violan el estilo. Esta regla no está grabada en piedra, sólo es una buena idea a seguir porque la mayoría del código Java que ejecute seguirá este estilo. Otro tema sobre la denominación más crítico tiene que ver con el uso de los caracteres de subrayado y del símbolo de dólar al principio de los nombres de identificadores. Es un poco arriesgado porque muchas bibliotecas C usan la misma convención de denominación para bibliotecas, que pueden importarse en su código Java. Para eliminar el problema potencial de choque de nombres en estos casos, es mejor no usar estos signos al comienzo de sus nombres de identificadores. Un buen uso del carácter de subrayado es la separación de palabras donde normalmente usaría un espacio.

Palabras clave

Las *palabras clave* son identificadores predefinidos reservados por Java para un objetivo determinado y se usan sólo de forma limitada y específica. Java tiene un conjunto de palabras clave más rico que C o C++, por lo que si está aprendiendo Java con conocimientos de C/C++, asegúrese de que presta atención a las palabras clave de Java. Las siguientes palabras clave están reservadas para Java:

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	false	native	this
byvalue	final	new	threadsafe
case	finally	null	throw
catch	float	package	transient
char	for	private	true
class	goto	protected	try
const	if	public	void
continue	implements	return	while
default	import	short	
do	instanceof	static	

Literales

Los elementos de programa que se usan de forma invariable se denominan *literales* o constantes. Los literales pueden ser números, caracteres o cadenas. Los literales numéricos incluyen los enteros, los de coma flotante y los booleanos. Éstos se consideran numéricos por la influencia de C en Java. En C, los valores booleanos para verdadero o falso se representan con un 1 y un 0. Los literales de caracteres siempre se refieren a un único carácter Unicode. Las cadenas, que contienen múltiples caracteres, aún se consideran literales, aunque están implementadas en Java como objetos.

Nota

Por si no está familiarizado con el conjunto de caracteres Unicode, sepa que es un conjunto de caracteres de 16 bits que substituye al conjunto de caracteres ASCII. Debido a que tiene 16 bits, hay suficientes entradas para representar muchos símbolos y caracteres de otros lenguajes. Unicode se está convirtiendo rápidamente en el estándar para los sistemas operativos modernos.

Literales enteros

Los literales enteros son los primarios en la programación Java y vienen en unos pocos formatos diferentes: decimal, hexadecimal y octal. Estos formatos se corresponden con la base del sistema numérico usado por el literal. Los literales decimales (base 10) aparecen como números ordinarios sin ninguna notación especial. Los hexadecimales (base 16) aparecen con un *0x* ó *0X* inicial, similar a C/C++. Los octales (base 8) aparecen con un 0 inicial delante de los dígitos. Por ejemplo, un literal entero para el número decimal 12 se representa en Java como *12* en decimal, como *0xC* en hexadecimal y *014* en octal.

Los literales enteros se almacenan por defecto en el tipo *int*, que es un valor de 32 bits con signo. Si trabaja con números muy grandes, puede forzar que un literal entero se almacene en el tipo *long* añadiendo un *l* ó una *L* al final del número, como en *79L*. El tipo largo es un valor de 64 bits con signo.

Literales de coma flotante

Los literales de coma flotante representan números decimales con partes fraccionarias, como el *3,142*. Pueden expresarse en notación estándar o científica, lo que significa que el número *563,84* también puede expresarse como el *5,6384e2*.

A diferencia de los literales enteros, los de coma flotante son de manera predeterminada del tipo *double*, que es un valor de 64 bits. Tiene la opción de usar el tipo más pequeño *float* de 32 bits si sabe que no se necesitan todos los 64 bits. Esto lo hace añadiendo una *f* o una F al final del número, como en *5,6384e2f*. Si es un rigorista de los detalles, puede expresar explícitamente que quiere el tipo *double* como unidad de almacenaje para su literal, como en *3,142d*. Debido a que el almacenaje predeterminado para números de coma flotante ya es *double*, esto no es necesario.

Literales booleanos

Los literales booleanos son verdaderamente un complemento bienvenido si viene del mundo de C/C++. En C no existe el tipo booleano y, por esto, no existen literales booleanos. Los valores booleanos verdadero y falso se representan con los valores enteros 1 y 0. Java soluciona este problema dando un tipo *boolean* con dos posibles estados: *true* y *false*. No es sorprendente que estos estados estén representados en el lenguaje Java con las palabras clave *true* y *false*.

Los literales booleanos se usan en la programación Java aproximadamente tan a menudo como los literales enteros, porque están presentes en casi todo tipo de estructura de control. Siempre que necesite representar una condición o estado con dos valores posible, lo que necesita es un *boolean*. Aprenderá algo más sobre el tipo *boolean* más adelante en este capítulo. Por ahora, sólo recuerde los dos valores del literal booleanos: *true* y *false*.

Literales de carácter

Los literales de carácter representan un único carácter Unicode y aparecen dentro de un par de comillas simples. De forma similar a C/C++, los caracteres especiales (de control y no imprimibles) se representan con una barra invertida (\) seguida del código del carácter. Un buen ejemplo de un carácter especial es \n, que fuerza la salida de una línea nueva cuando se imprime. La tabla 12.2 muestra los caracteres especiales admitidos por Java.

Tabla 12.2 Caracteres especiales admitidos por Java

Descripción	Representación
Barra invertida	\\
Continuación	\
Retroceso	\b
Retorno de carro	\r
Alimentación de formularios	\f
Tabulador horizontal	\t

(continúa)

Descripción	Representación
Línea nueva	\n
Comillas simples	\'
Comillas dobles	\"
Carácter Unicode	\udddd
Número octal	\ddd

Un ejemplo de un literal de carácter Unicode es *\u0048*, que es una representación hexadecimal del carácter *H*. Este mismo carácter se representa en octal como *\110*.

Literales de cadena

Los literales de cadena representan múltiples caracteres y aparecen dentro de un par de comillas dobles. A diferencia de todos los otros literales descritos, los literales de cadena se implementan en Java con la clase *String*. Esto es muy diferente de la representación de C/C++ de cadenas como una matriz de caracteres.

Cuando Java encuentra una literal de cadena, crea un caso de la clase *String* y define su estado con los caracteres que aparecen dentro de las comillas dobles. Desde una perspectiva del uso, el hecho que Java implemente cadenas como objetos es relativamente poco importante. Sin embargo, vale la pena mencionarlo ahora porque es un recordatorio de que Java está muy orientado al objeto por su naturaleza, mucho más que C++, considerado el estándar actual de programación orientada a objetos.

Operadores

Los operadores, conocidos también como *operandos*, indican una evaluación o computación para ser realizada en objetos de datos u objetos. Estos operandos pueden ser literales, variables o tipos de retorno de funciones. Los operadores admitidos por Java son los siguientes:

```
+     -     *     /     %     &     |
^     ~     &&    ||    !     <     >
<=    >=    <<    >>    >>>   =     ?
++    --    ==    +=    -=    *=    /=
%=    &=    =     ^=    !=    <<=   >>=
>>>=  .     [     ]     (     )
```

La simple observación de estos operadores probablemente no le ayudará mucho en la determinación de cómo usarlos. No se preocupe, aprenderá mucho más sobre operadores y cómo se usan en el siguiente capítulo, "Expresiones, operadores y estructuras de control."

Separadores

Los separadores se usan para informar al compilador de Java de cómo están agrupadas las cosas en el código. Por ejemplo, los elementos de una lista están separados por comas, de forma muy parecida a las listas de elementos de una frase. Sin embar-

go, los separadores de Java van mucho más allá de las comas, como descubrirá en el próximo capítulo. Los separadores admitidos por Java son los siguientes:

```
( ) ; , :
```

Comentarios y espacios en blanco

Antes aprendió que el compilador de Java elimina los comentarios y los espacios en blanco durante la descomposición en tokens del código fuente. Puede preguntarse, ¿qué es lo que cualifica al espacio en blanco y cómo se soportan los comentarios? Primero, los espacios en blanco los forman los espacios, los tabuladores y las alimentaciones de formulario. El compilador de Java elimina todos los casos de éstos, al igual que los comentarios. Éstos pueden definirse de tres formas distintas, tal como se muestra en la tabla 12.3

Tabla 12.3 Tipos de comentarios admitidos por Java

Tipo	Uso
`/* comentario */`	Se ignoran todos los caracteres entre `/*` y `*/`.
`// comentario`	Se ignoran todos los caracteres detrás de `//` hasta el final de la línea.
`/** comentario */`	Lo mismo que `/* */`, excepto que el comentario puede usarse con la herramienta javadoc para crear documentación automática.

El primer tipo de comentario (*/* comentario */*) debería serle familiar si ya ha programado antes en C. El compilador ignora todos los caracteres entre los delimitadores de comentario /* y */. Igualmente, el segundo tipo de comentario (*// comentario*) debería serle familiar si ha usado C++. El compilador ignora todos los caracteres detrás del delimitador de comentarios // hasta el final de la línea. Estos dos tipos de comentarios han sido tomados de C y C++. El tipo de comentario final (*/** comentario */*) funciona como que el tipo de comentario de estilo C, con la ventaja adicional de que puede usarse con la herramienta Java Automatic Documentation, javadoc, para crear documentación automática a partir del código fuente. La herramienta javadoc se describe en el capítulo 37, ///"Java Documentation." Los siguientes son algunos ejemplos de uso de los diversos tipos de comentarios:

```
/* Este es un comentario estilo C. */
// Este es un comentario estilo C++.
/** Este es un comentario estilo javadoc. */
```

Tipos de datos

Uno de los conceptos fundamentales de cualquier lenguaje de programación es el de los tipos de datos. Éstos definen los métodos de almacenamiento disponibles para representar información, junto con la manera como se interpreta ésta. Los tipos de datos están estrechamente ligados al almacenamiento de variables en la memoria, porque el tipo de datos de una variable determina cómo interpreta el compilador el contenido de la memoria. Ya ha saboreado un poco los tipos de datos en la descripción de los tipos de literales.

Para crear una variable en la memoria, debe declararla dando el tipo de variable y el identificador que la identifica de forma única.

La sintaxis de la sentencia de declaración de Java de variables es la siguiente:

```
Tipo Identificador [, Identificador];
```

La sentencia de declaración informa al compilador que reserve memoria para una variable del tipo *Tipo* con el nombre *Identificador*. El *Identificador* entre corchetes opcional indica que puede realizar declaraciones múltiples del mismo tipo separándolas con comas. Al final, como en todas las sentencias Java, la sentencia de declaración finaliza con un punto y coma.

Los tipos de datos Java pueden dividirse en dos categorías: simples y compuestos. Los simples son tipos nucleares que no se derivan de otros tipos. Los enteros, de coma flotante, booleanos y de carácter son todos tipos simples. Los tipos compuestos, por otra parte, se basan en los tipos simples e incluyen las cadenas, las matrices y tanto las clases como las interfaces en general. Aprenderá las matrices más adelante en este capítulo. Las clases y las interfaces están cubiertas en el capítulo 14, "Clases, paquetes e interfaces".

Tipos enteros de datos

Los tipos enteros de datos se usan para representar números enteros con signo. Hay cuatro tipos: *byte*, *short*, *int* y *long*. Cada uno de ellos ocupa una diferente cantidad de espacio en la memoria, como se muestra en la tabla 12.4.

Tabla 12.4 Tipos enteros Java

Tipo	Tamaño
byte	8 bits
short	16 bits
int	32 bits
long	64 bits

Para declarar variables usando los tipos enteros, emplee la sintaxis de declaración descrita previamente con el tipo deseado. Los siguientes son algunos ejemplos de declaración de variables enteras:

```
int i;
short rocketFuel;
long angle, magnitude;
byte red, green, blue;
```

Tipos de datos de coma flotante

Los tipos de datos de coma flotante se usan para representar números con partes fraccionarias. Hay dos tipos de coma flotante: *float* y *double*. El primero reserva almacenamiento para un número de precisión simple de 32 bits y el segundo lo hace para un número de precisión doble de 64 bits.

La declaración de variables de coma flotante es muy similar a la de las variables enteras. Los siguientes son algunos ejemplos de declaraciones de variables de coma flotante:

```
float temperature;
double windSpeed, barometricPressure;
```

Tipo de datos boolean

El tipo de datos *boolean* se usa para almacenar valores con uno de los dos estados: *true* o *false*. Puede imaginar el tipo *boolean* como un valor entero de 1 bit, porque un bit sólo puede tener dos valores: 1 ó 0. Sin embargo, en lugar de usar 1 y 0, usa las palabras clave de Java *true* y *false*, que no son sólo convenciones de Java, sino realmente los únicos valores booleanos legales. Esto significa que no puede usar de forma intercambiable los tipos booleanos y los enteros como en C/C++. Para declarar un valor booleano, use simplemente la declaración de tipo *boolean*:

```
boolean gameOver;
```

Tipo de datos carácter

El tipo de datos carácter se usa para almacenar caracteres Unicode simples. Debido a que el conjunto de caracteres Unicode se compone de valores de 16 bits, el tipo de datos *char* se almacena en un entero sin signo de 16 bits. Crea variables del tipo *char* de la siguiente forma:

```
char firstInitial, lastInitial;
```

Recuerde que el tipo *char* es útil sólo para almacenar caracteres aislados. Si viene de C/C++, puede caer en la tentación de formar una cadena creando una matriz de *chars*. En Java esto no es necesario porque la clase *String* se ocupa de manejar las cadenas, lo cual no significa que no deba nunca crear matrices de caracteres, sólo quiere decir que no debería usar una matriz de caracteres cuando lo que realmente quiere es una cadena. C y C++ no distinguen entre matrices de caracteres y cadenas, pero Java sí.

Conversión de tipos de datos

Inevitablemente habrá momentos en que necesite convertir de un tipo de datos a otro. Este proceso se llama *conversión*. A menudo, la conversión es necesaria cuando una función devuelve un tipo diferente del que necesita para realizar una operación. Por ejemplo, la función miembro *read* del flujo de entrada estándar (*System.in*) devuelve un *int*. Debe convertir este tipo *int* al *char*, antes de almacenarlo, de la forma siguiente:

```
char c = (char)System.in.read();
```

La conversión se lleva a cabo colocando el tipo deseado entre paréntesis, a la izquierda del valor a convertir. La llamada a la función *System.in.read* devuelve un valor *int*, que se convierte en un *char* debido a la conversión (*char*). El valor *char* resultante se almacena en la variable *c char*.

El tamaño del almacenamiento de los tipos que está intentando convertir es muy importante. No todos los tipos se convertirán a otros de forma segura. Para entenderlo considere el resultado de convertir un *long* a un *int*. Un *long* es un valor de 64 bits y un *int* de 32 bits. Cuando se convierte un *long* a un *int*, el compilador corta los 32 bits superiores del valor *long*, de forma que encajen en el *int* de 32 bits. Si los 32 bits superiores del *long* contienen cualquier información útil, esta se perderá y el número cambiará como resultado de la conversión. También puede producirse una pérdida de información en la conversión entre tipos fundamentales diferentes, como entre los números enteros y los de coma flotante. Por ejemplo, en la conversión de un *double* a un *long* se perderá la información fraccionaria, aunque ambos números son valores de 64 bits.

En las conversiones el tipo destino siempre debería ser igual o mayor en tamaño que el tipo fuente. Además, debería prestar mucha atención en la conversión entre tipos fundamentales, como los tipos de coma flotante y el entero. La tabla 12.5 lista las conversiones en las que está garantizado que no se producirá ninguna pérdida de información.

Tabla 12.5 Conversiones cuyos resultados no sufren pérdidas de información

Del tipo	Al 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*

Bloques y ámbito

En Java, el código fuente está dividido en partes separadas por signos de llave de apertura y de cierre ({ y }). Todo lo que está entre las llaves se considera un bloque y existe de forma más o menos independiente de todo lo que está fuera. Los bloques no sólo son importantes en un sentido lógico, se necesitan como parte de la sintaxis del lenguaje Java. Sin llaves, el compilador tendría problemas para determinar dónde acaba una sección de código y dónde comienza la siguiente. Desde un punto de vista puramente estético, sería muy difícil para alguien que leyese su código entenderlo sin las llaves. Por este motivo, no sería muy fácil para el usuario entender su propio código sin las llaves.

Las llaves se usan para agrupar sentencias relacionadas. Puede imaginarse que todo lo que está dentro de llaves concordantes se ejecuta como una sentencia. En efecto, desde un bloque externo es exactamente lo que parece un bloque interno: una sola sentencia. Pero, ¿qué es un bloque externo? Suerte que lo haya preguntado, porque nos lleva a otro punto importante: los bloques pueden ser jerárquicos. Un bloque puede contener uno o más subbloques anidados.

La sangría para identificar los diferentes bloques es característica del estilo de programación Java. Cada vez que introduzca un nuevo bloque debe sangrar su código fuente cierto número de espacios, preferiblemente dos. Cuando deja un bloque debe retroceder, o *desangrar*, dos espacios. Es una convención bastante establecida en muchos lenguajes de programación. Sin embargo, sólo es un estilo y técnicamente no forma parte del lenguaje. El compilador produciría la misma salida incluso si no sangrase nada. La sangría se usa para el programador, no para el compilador; simplemente hace que el código sea más fácil de seguir y entender. A continuación se muestra un ejemplo de sangría adecuada de bloques de Java:

```
for (int i = 0; i < 5; i++) {
  if (i < 3) {
    System.out.println(i);
  }
}
```

A continuación se muestra el mismo código sin sangrías de bloque:

```
for (int i = 0; i < 5; i++) {
if (i < 3) {
System.out.println(i);
}
}
```

El primer listado de código muestra claramente la subdivisión del flujo del programa por medio de las sangrías; es obvio que la sentencia *if* está anidada dentro del bucle *for*. El segundo listado de código, por otra parte, no proporciona señales visuales de la relación entre los bloques de código. No se preocupe si desconoce las sentencias *if* y bucles *for*; aprenderá mucho sobre ellos en el siguiente capítulo, "Expresiones, operadores y estructuras de control".

El concepto de *ámbito* está estrechamente relacionado con el de bloque y es muy importante cuando se trabaja con variables de Java. El ámbito se refiere a cómo las secciones de un programa (*bloques*) afectan el tiempo de vida de las variables.

Toda variable declarada en un programa tiene un ámbito asociado, lo que significa que la variable sólo se usa en esa parte determinada del programa.

El ámbito está determinado por los bloques. Para entenderlos mejor, vuelva a mirar la clase *HelloWorld* en el listado 12.1. La clase *HelloWorld* se compone de dos bloques. El bloque exterior del programa es el que define la clase *HelloWorld*:

```
class HelloWorld {
...
}
```

Los bloques de clase son muy importantes en Java. Casi cualquier cosa de interés es una clase en sí misma o pertenece a una. Por ejemplo, los métodos se definen dentro de las clases a las que pertenecen. Tanto sintácticamente como lógicamente, todo sucede en Java dentro de una clase. Volviendo a *HelloWorld*, el bloque interno define el código dentro del método *main* de la forma siguiente:

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

El bloque interno se considera que está anidado dentro del bloque externo del programa. Cualquier variable definida en el bloque interno es local respecto a ese bloque y no es visible respecto al bloque externo; el ámbito de las variables se define como el bloque interno. Para conseguir una mejor idea del uso del ámbito y de los bloques, eche un vistazo a la clase *HowdyWorld* del listado 12.2.

Listado 12.2 La clase HowdyWorld

```
class HowdyWorld {
  public static void main (String args[]) {
    int i;
  printMessage();
}
  public static void printMessage () {
     int j;
     System.out.println("Howdy, World!");
  }
}
```

(continúa)

La clase *HowdyWorld* contiene dos métodos: *main* y *printMessage*. *main* debería serle familiar por la clase *HelloWorld*, excepto que esta vez declara una variable entera *i* y llama al método *printMessage*. Éste es un nuevo método que declara una variable *j* e imprime el mensaje "Howdy, World!" en el flujo de salida estándar, de forma muy parecida a como lo hacía el método *main* de HelloWorld.

Probablemente ya ha averiguado que HowdyWorld produce básicamente la misma salida que HelloWorld, porque la llamada a *printMessage* provoca que se muestre un único mensaje de texto.

Lo que quizás no puede ver inmediatamente es el ámbito de los enteros definidos en cada método. El entero *i* definido en *main* tiene un ámbito limitado al cuerpo del método *main*. El cuerpo de *main* está definido por las llaves que rodean al método (el bloque de método). De forma parecida, el entero j tiene un ámbito limitado al cuerpo del método *printMessage*. La importancia del ámbito de estas dos variables es que éstas no son visibles más allá de sus ámbitos respectivos; el bloque de clase *HowdyWorld* no sabe nada de los dos enteros. Además, *main* no sabe nada de j y *printMessage* no sabe nada sobre i.

El ámbito se hace más importante cuando comienza a anidar bloques de código dentro de otros bloques. La clase *GoodbyeWorld* que se muestra en el listado 12.3 es un buen ejemplo de variables anidadas dentro de ámbitos distintos.

Listado 12.3 La clase GoodbyeWorld

```
class GoodbyeWorld {
  public static void main (String args[]) {
    int i, j;
    System.out.println("Goodbye, World!");
    for (i = 0; i < 5; i++) {
      int k;
      System.out.println("Bye!");
    }
  }
}
```

Los enteros *i* y *j* tienen ámbitos dentro del cuerpo del método *main*. El entero *k*, sin embargo, tiene un ámbito limitado al bloque del bucle *for*. Debido a que el ámbito de *k* está limitado al bloque del bucle *for*, no puede ser vista desde fuera de ese bloque. Por otra parte, *i* y *j* sí pueden ser vistas dentro del bloque del bucle *for*. Esto significa que la definición de ámbitos tiene un efecto jerárquico de arriba a abajo, las variables definidas en ámbitos exteriores sí pueden ser vistas y usadas dentro de ámbitos anidados, pero las variables definidas en ámbitos anidados están limitadas a esos ámbitos. Por cierto, no se preocupe si no está familiarizado con los bucles *for*, lo aprenderá todo sobre ellos en el siguiente capítulo, "Expresiones, operadores y estructuras de control".

Por más razones que la visibilidad, es importante que preste atención al ámbito de las variables cuando las declare. Además de determinar la visibilidad de las variables, el ámbito también define su tiempo de vida. Esto significa que las variables se destruyen realmente cuando la ejecución del programa abandona su ámbito. Vol-

viendo a mirar el ejemplo GoodByeWorld, se reserva almacenamiento para los enteros *i* y *j* cuando la ejecución del programa entra en el método *main*. Cuando se entra al bloque del bucle *for*, se reserva almacenamiento para el entero k. Cuando la ejecución del programa abandona el bloque del bucle *for*, se libera la memoria para **k** y se destruye la variable. De forma similar, cuando la ejecución del programa abandona *main*, todas las variables de su ámbito se liberan y destruyen (*i* y *j*). El concepto de tiempo de vida de una variable y ámbito se hace incluso más importante cuando comienza a tratar las clases. Recibirá una buena dosis de esto en el capítulo 14, "Clases, paquetes e interfaces."

Matrices

Una matriz es una construcción que proporciona almacenaje a una lista de elementos del mismo tipo. Los elementos de una matriz pueden ser de tipo de datos simple o compuesto. Las matrices también pueden ser multidimensionales. Las matrices Java se declaran con corchetes cuadrados ([ ]). A continuación algunos ejemplos de matrices en Java:

```
int numbers[ ];
char[ ] letters;
long grid[ ][ ];
```

Si está familiarizado con matrices en otro lenguaje, puede sorprenderle la ausencia de un número entre los corchetes cuadrados indicando el número de elementos de la matriz. Java no le deja indicar el tamaño de una matriz vacía cuando la declara. Siempre ha de definir el tamaño de una matriz de forma explícita con el operador *new* o asignando una lista de elementos a la matriz, cuando se elabore. El operador *new* se describe en el capítulo siguiente, "Expresiones, operadores y estructuras de Control".

Nota

Puede que parezca complicado el tener que definir explícitamente el tamaño de una matriz con el operador new. Pero Java no tiene punteros como C o C++ y, por esto, no le deja apuntar simplemente en una matriz y crear nuevos elementos. Manejando de esta forma la gestión de la memoria, se han evitado en el lenguaje Java los problemas de comprobación de límites comunes a C y C++.

Otra cosa extraña que puede observar en las matrices Java es la situación opcional de los corchetes cuadrados en la declaración de la matriz. Se le permite colocar los corchetes cuadrados tanto después del tipo de variable como después del identificador.

A continuación se muestra un par de ejemplos de matrices que han sido declaradas y definidas con un tamaño determinado usando el operador *new* o asignando una lista de elementos en la declaración de la matriz:

```
char alphabet[ ] = new char[26];
int primes = {7, 11, 13};
```

Java también admite las estructuras más complejas para el almacenamiento de listas de elementos, las pilas y las tablas de hashing. A diferencia de las matrices, estas estructuras están implementadas en Java como clases. Recibirá un curso intensivo sobre estos otros mecanismos de almacenamiento en el capítulo 19, "El paquete de utilidades".

Cadenas

En Java, las cadenas se tratan con una clase especial llamada *String*. Incluso las cadenas de literales se gestionan internamente por medio de una instancia de la clase *String*. Una instancia de una clase es simplemente un objeto que ha sido creado en base a la descripción de la clase. Este método de manejar cadenas es muy diferente al de los lenguajes tipo C y C++, donde las cadenas se representan simplemente como una matriz de caracteres. Las siguientes son cadenas declaradas usando la clase *String* de Java:

String message;

String name = "Mr. Blonde";

En este punto es tan importante conocer la clase *String* tanto por dentro como por fuera. Aprenderá todos los detalles de la clase *String* en el capítulo 18, "El paquete del lenguaje".

Resumen

En este capítulo ha revisado los componentes centrales del lenguaje Java, y habrá comprendido por qué Java se ha vuelto tan popular en un tiempo relativamente corto. Con grandes mejoras respecto a las debilidades de los lenguajes C y C++, los estándares del lenguaje del sector, Java cobrará importancia sin duda alguna en un futuro próximo. Los elementos del lenguaje cubiertos en este capítulo son sólo la punta del iceberg respecto a las ventajas de la programación en Java.

Ahora que ya está armado con los fundamentos del lenguaje Java, se espera que esté preparado para seguir adelante y profundizar en el lenguaje Java. El siguiente capítulo, "Expresiones, operadores y estructuras de control" cubre exactamente lo que su título sugiere. En él aprenderá cómo trabajar con y manipular gran parte de la información que ha aprendido en este capítulo. Al hacerlo, será capaz de comenzar a escribir programas que hagan algo más que mostrar mensajes "monos" en la pantalla.

Capítulo 13

Expresiones, operadores y estructuras de control

por Michael Morrison

Contenido del capítulo

- Expresiones y operadores 155
- Estructuras de control 168

En el capítulo anterior aprendió los componentes básicos de un programa Java. Este capítulo se centra en cómo usar estos componentes para hacer cosas útiles. Los tipos de datos son interesantes, pero sin expresiones y operadores no puede hacer mucho con ellos. Incluso las expresiones y los operadores solos son algo limitados. Añádales las estructuras de control y podrá hacer algunas cosas interesantes.

Este capítulo cubre todas estas cosas y reúne muchas de las piezas que faltaban del rompecabezas de programación de Java. No sólo aumentará en gran manera su conocimiento del lenguaje Java, también aprenderá lo necesario para escribir algunos programas más interesantes.

Expresiones y operadores

Si ha creado las variables, querrá realizar algo con ellas. Los operadores le permiten realizar una evaluación o una computación de objetos de datos u objetos. Los operadores aplicados a variables y a literales forman las *expresiones*. Una expresión puede imaginarse como una ecuación programática, más formalmente, es una secuencia de uno o más objetos de datos (operandos) y cero o más operadores que dan un resultado. Un ejemplo de una expresión es el siguiente:

```
x = y / 3;
```

En esta expresión, *x* e *y* son variables, *3* es un literal y *= y /* son operadores. Esta expresión dice que la variable *y* se divide por *3* usando el operador de división *(/)* y el resultado se almacena en *x*, usando el operador de asignación *(=)*. Observe que la expresión se describió de derecha a izquierda. Esta es la técnica estándar para descomponer y entender expresiones en Java y también en la mayoría de los otros lenguajes de programación. Esta evaluación de derecha a izquierda de expresiones no es sólo una técnica para su propia comprensión de éstas, es cómo el mismo compilador las analiza para generar código.

Precedencia de operadores

Incluso con el análisis de derecha a izquierda de las expresiones por el compilador, aún son muchas las veces en las que el resultado de una expresión sería indeterminado sin algunas otras reglas. La siguiente expresión muestra el problema:

```
x = 2 * 5 + 12 / 4
```

Usando estrictamente la evaluación de derecha a izquierda, primero se lleva a cabo la operación de división *12 / 4*, que da un resultado de *3*. Después se ejecuta la operación de adición *5 + 3*, lo que le da un resultado de *8*. Después se realiza la operación de multiplicación *2 * 8*, lo que le da un resultado de *16*. Finalmente se trata la operación de asignación *x = 16*, en cuyo caso el número 16 se asigna a la variable *x*.

Si tiene alguna experiencia con precedencia de operadores de otro lenguaje, ya puede estar cuestionándose la evaluación de esta expresión y por buenas razones juzgará que ¡está mal! El problema es que el uso de la simple evaluación de expresiones de derecha a izquierda puede producir resultados inconsistentes, dependiendo del orden de los operadores. La solución a este problema está en la precedencia de operadores, que determina el orden en el que se evalúan los operadores. Todo operador Java tiene una precedencia asociada. A continuación se muestra un listado de todos los operadores Java desde la máxima a la mínima precedencia:

```
.       [ ]     ()
++      --      !       ~
*       /       %
+       -
<<      >>      >>>
<       >       <=      >=
==      !=
&
^
&&
||
?:
=
```

En esta lista de operadores, todos los que están en una fila determinada tienen la misma precedencia. El nivel de precedencia de cada fila disminuye de arriba a abajo, es decir, el operador *[ ]* tiene una precedencia superior que el *, pero la misma que el *( )*.

La evaluación de la expresión aún va de derecha a izquierda, pero sólo cuando se tratan operadores que tengan la misma precedencia. En caso contrario, los operadores con una precedencia superior se evalúan antes que los operadores con una inferior. Sabiendo esto, reconsidere la misma ecuación:

```
x = 2 * 5 + 12 / 4
```

Antes de usar la evaluación de derecha a izquierda de la expresión, compruebe primero si cualquiera de los operadores tiene una precedencia diferente. ¡En efecto, así es! Los operadores de multiplicación *(*)* y división *(/)* tienen la precedencia más alta, seguidos por el operador de adición *(+)* y, finalmente, por el de asignación *(=)*. Debido a que los operadores de multiplicación y división comparten la misma precedencia, evalúelos de derecha a izquierda. Al hacerlo, realiza la operación de división *12 / 4*, primero, resultando un *3*. Después realice la operación de multiplicación *2 * 5*, que da un *10*. Después de realizar estas dos operaciones, la expresión se ve así:

```
x = 10 + 3;
```

Debido a que el operador de adición tiene una precedencia superior al de asignación, realice a continuación la adición *10 + 3*, que da un *13*. Por último, se procesa la operación de asignación *x = 13*, que da el número *13* y que se asigna a la variable *x*. Como puede ver, al evaluar la expresión usando la precedencia de operadores se produce un resultado completamente diferente.

Sólo para acabar de entenderlo, considere otra expresión que usa paréntesis con el objetivo de agrupar:

```
x = 2 * (11 - 7);
```

Sin los paréntesis que agrupan, realizaría primero la multiplicación y después la sustracción. Sin embargo, según la lista precedente, el operador *( )* antecede a los otros operadores. Por esto, se lleva a cabo primero la sustracción *11 - 7*, que da *4* y la siguiente expresión:

```
x = 2 * 4;
```

El resto de la expresión se resuelve fácilmente con una multiplicación y una asignación para dar un resultado de *8* a la variable *x*.

Operadores enteros

Hay tres tipos de operaciones que pueden realizarse sobre enteros: unaria, binaria y relacional. Los operadores unarios actúan sólo sobre números enteros simples y los binarios, sobre pares de números enteros. Ambos operadores devuelven resultados enteros. Los operadores relacionales, por otra parte, actúan sobre dos números enteros pero devuelven un resultado booleano en lugar de un entero.

Los operadores unarios y binarios devuelven típicamente un tipo *int*. Todas las operaciones que impliquen los tipos *byte*, *short* e *int* siempre dan como resultado un *int*. La única excepción a esta regla es si uno de los operandos es un **long**, en cuyo caso el resultado de la operación también será un tipo *long*.

Unarios

Los operadores enteros unarios actúan sobre un solo entero. La tabla 13.1 lista los operadores enteros unarios.

Tabla 13.1 Los operadores enteros unarios

Descripción	Operador
Incremento	++
Decremento	--
Negación	-
Complemento a nivel de bits	~

Los operadores de incremento y decremento (++ y --) aumentan y disminuyen las variables enteras en una unidad. Parecidos a sus complementos en C y C++, estos

operadores pueden usarse tanto en forma prefijo como sufijo. Un operador de prefijo tiene lugar antes de la evaluación de la expresión en la que está y uno de sufijo tiene lugar después de que la expresión se ha evaluado. Los operadores unarios de prefijo se sitúan inmediatamente antes de la variable y los de sufijo, inmediatamente después. A continuación vea un ejemplo de cada tipo de operador:

```
y = ++x;
z = x--;
```

En el primer ejemplo, *x* se *incrementa en prefijo*, lo que significa que se incrementa antes de ser asignada a *y*. En el segundo ejemplo, *x* se *decrementa en sufijo*, lo que significa que se decrementa después de ser asignada a *z*. En el último caso, se asigna a *z* el valor de *x* antes de que *x* se decremente. El listado 13.1 contiene el programa *IncDec*, que usa ambos tipos de operadores. Por favor, observe que el programa *IncDec* está implementado realmente en la clase Java *IncDec*. Esto es consecuencia de la estructura orientada a objetos de Java, que requiere que los programas se implementen como clases. Así pues, cuando vea una referencia a un programa Java, tenga en cuenta que se está refiriendo realmente a una clase Java.

Listado 13.1 La clase IncDec

```
class IncDec {
  public static void main (String args[]) {
    int x = 8, y = 13;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("++x = " + ++x);
    System.out.println("y++ = " + y++);
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

El programa *IncDec* da los siguientes resultados:

```
x = 8
y = 13
++x = 9
y++ = 13
x = 9
y = 14
```

El operador entero unario de negación (-) se usa para cambiar el signo de un valor entero. Este operador es tan simple como suena, como se muestra en el siguiente ejemplo:

```
x = 8;
y = -x;
```

En este ejemplo, se asigna a *x* el valor literal *8* y después se realiza la negación y se asigna a *y*. El valor resultante de *y* es *-8*. Para ver este código en un programa Java real, mire el programa Negation en el listado 13.2.

Listado 13.2 La clase Negation

```
class Negation {
  public static void main (String args[]) {
    int x = 8;
    System.out.println("x = " + x);
```

(continúa)


```
    int y = -x;
    System.out.println("y = " + y);
  }
}
```

El último operador entero unario Java es el operador de complemento a nivel de bit (~), que realiza una negación a nivel de bit de un valor entero. La *negación a nivel de bit* significa que se conmuta cada bit del número. En otras palabras, todos los ceros binarios se convierten en unos y todos los unos binarios se convierten en ceros. Mire un ejemplo muy similar al del operador de negación:

```
x = 8;
y = ~x;
```

En este ejemplo se asigna otra vez a *x* el valor literal *8*, pero se complementa a nivel de bit antes de ser asignado a *y*. ¿Qué significa esto? Bien, sin entrar en detalle de cómo se almacenan en la memoria los enteros, significa que a todos los bits de la variable *x* se invierten, produciendo un resultado decimal de *-9*. Este resultado tiene que ver con el hecho de que los números negativos se almacenan en la memoria usando un método conocido como el *complemento a dos* (vea la siguiente nota). Si tiene problemas para creerlo, inténtelo por sí mismo con el programa *BitwiseComplement* que se muestra en el listado 13.3.

Nota

Los números enteros se almacenan en la memoria como series de bits binarios que pueden tener, cada uno, un valor de 0 ó 1. Un número se considera negativo si el bit de orden superior del número está a 1. Debido a que un complemento a nivel de bits conmuta todos los bits de un número, incluyendo el de orden superior, se invierte el signo de un número.

Listado 13.3 La clase BitwiseComplement

```
class BitwiseComplement {
  public static void main (String args[]) {
    int x = 8;
    System.out.println("x = " + x);
    int y = ~x;
    System.out.println("y = " + y);
  }
}
```

Binarios

Los operadores enteros binarios actúan sobre pares de enteros. La tabla 13.2 lista los operadores enteros binarios.

Tabla 13.2 Los operadores enteros binarios

Descripción	Operador
Adición	+
Sustracción	-

(continúa)

Tabla 13.2 Continúa

Descripción	Operador
Multiplicación	*
División	/
Módulo	%
AND a nivel de bits	&
OR a nivel de bits	|
XOR a nivel de bits	^
Desplazamiento a la izquierda	<<
Desplazamiento a la derecha	>>
Desplazamiento a la derecha con relleno de ceros	>>>

Los operadores de adición, sustracción, multiplicación y división (+, -, *, /) hacen todos lo que espera de ellos. Una cosa importante es observar cómo funciona el operador de división; debido a que está tratando con operandos enteros, el operador de división devuelve un divisor entero. En los casos en los que la división da un resto, el operador de módulo *(%)* puede usarse para obtener el valor del resto. El listado 13.4 contiene el programa *Arithmetic*, que muestra cómo funcionan los operadores aritméticos enteros binarios básicos.

Listado 13.4 La clase Arithmetic

```
class Arithmetic {
  public static void main (String args[]) {
    int x = 17, y = 5;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("x + y = " + (x + y));
    System.out.println("x - y = " + (x - y));
    System.out.println("x * y = " + (x * y));
    System.out.println("x / y = " + (x / y));
    System.out.println("x % y = " + (x % y));
  }
}
```

Vea los resultados del programa *Arithmetic*:

```
x = 17
y = 5
x + y = 22
x - y = 12
x * y = 85
x / y = 3
x % y = 2
```

Estos resultados no deberían sorprenderle mucho. Observe sólo que la operación de división *x / y*, se reduce a *17 / 5*, que da el resultado *3*. Observe también que la operación de módulo *x% y*, que se reduce a *17 % 5*, acaba con un resultado de 2, que es el resto de la división entera.

Matemáticamente, una división por cero da un resultado infinito. Debido a que representar números infinitos es un gran problema para los computadores, la división o operaciones de módulo por cero dan un error. Para ser mas detallados, se lanza una excepción en tiempo de ejecución. Aprenderá mucho más sobre excepciones en el capítulo 16, "Manejo de excepciones".

Los operadores AND, OR y XOR a nivel de bits (*&*, | y ^) actúan sobre los bits individuales de un entero. Estos operadores a veces son útiles cuando se emplea un entero como un campo de bits, por ejemplo cuando se usa un entero para representar un grupo de indicadores binarios. Un *int* es capaz de representar 32 indicadores diferentes, porque se almacena en 32 bits. El listado 13.5 contiene el programa *Bitwise*, que muestra cómo usar los operadores enteros a nivel de bits.

Listado 13.5 La clase Bitwise

```
class Bitwise {
  public static void main (String args[]) {
    int x = 5, y = 6;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("x & y = " + (x & y));
    System.out.println("x | y = " + (x | y));
    System.out.println("x ^ y = " + (x ^ y));
  }
}
```

El resultado de ejecutar *Bitwise* es el siguiente:

```
x = 5
y = 6
x & y = 4
x | y = 7
x ^ y = 3
```

Para entender este resultado, primero ha de comprender los equivalentes binarios de cada número decimal. En *Bitwise*, las variables *x* e *y* se definen como *5* y *6*, lo que corresponde a los números binarios *0101* y *0110*. La operación AND a nivel de bits compara cada bit de cada número para ver si son iguales. Pone el bit resultante a 1 si ambos bits comparados son 1, y 0 en caso contrario. El resultado de la operación AND a nivel de bits con estos dos números es *0100* en binario o *4* en decimal. La misma lógica se utiliza para los otros dos operadores, excepto que las reglas de comparación de bits son diferentes. El operador OR a nivel de bits pone el bit resultante a *1* si cualquiera de los bits que se comparan es 1. Con estos números, el resultado es *0111* en binario o 7 en decimal. Por último, el operador XOR a nivel de bits pone el bit resultante a 1 si exactamente uno y sólo uno de los bits que se compara es 1 y en caso contrario lo pone a 0. Con estos números, el resultado es *0011* binario o *3* decimal.

Los operadores de desplazamiento a la izquierda, desplazamiento a la derecha y desplazamiento a la derecha con relleno de ceros (<<, >> y >>>) desplazan los bits individuales de un entero en una cantidad entera determinada. Los siguientes son algunos ejemplos de cómo se usan estos operadores:

```
x << 3;
y >> 7;
z >>> 2;
```

En el primer ejemplo, los bits individuales de una variable *x* entera se desplazan tres posiciones hacia la izquierda. En el segundo ejemplo, los bits de *y* se desplazan siete lugares a la izquierda. El tercer ejemplo muestra cómo se desplaza *z* dos posiciones a la derecha, con ceros desplazados en las dos posiciones más a la izquierda. Para ver los operadores en un programa real, mire *Shift* en el listado 13.6.

Listado 13.6 La clase Shift

```
class Shift {
  public static void main (String args[]) {
    int x = 7;
    System.out.println("x = " + x);
    System.out.println("x >> 2 = " + (x >> 2));
    System.out.println("x << 1 = " + (x << 1));
    System.out.println("x >>> 1 = " + (x >>> 1));
  }
}
```

A continuación se muestra la salida de *Shift*:

```
x = 7
x >> 2 = 1
x << 1 = 14
x >>> 1 = 3
```

El número que se desplaza en este caso es el decimal 7, que se representa como *0111* en binario. La primera operación de desplazamiento a la derecha desplaza los bits dos posiciones, lo que da como resultado el número binario *0001*, o decimal 1. La siguiente operación, un desplazamiento a la izquierda, desplaza los bits una posición, lo que da como resultado el número binario *1110*, o *14* decimal. Por fin, la última operación es un desplazamiento a la derecha con relleno con ceros, que desplaza los bits una posición, lo que da como resultado el número binario *0011*, o *3* decimal. Muy simple, ¿eh? ¡Y probablemente pensaba que era difícil trabajar con enteros a nivel de bits!

En función de estos ejemplos, puede estar preguntándose cuál es la diferencia entre los operadores de desplazamiento a la derecha (>>) y los de desplazamiento a la derecha con relleno de ceros (>>>). El operador de desplazamiento a la derecha parece desplazar ceros en los bits más a la izquierda, igual que el operador de desplazamiento a la derecha con relleno de ceros, ¿no es así? Bien, cuando se tratan números positivos, no hay diferencia entre los dos operadores, ambos desplazan ceros en los bits superiores de un número. La diferencia surge cuando comienza a desplazar números negativos. Recuerde que los números negativos tienen el bit de más alto nivel a 1. El operador de desplazamiento a la derecha preserva el bit de orden más alto y desplaza efectivamente los 31 bits inferiores. Este comportamiento hace que para números negativos se produzcan resultados similares a los de los números positivos. Es decir, -8 desplazado a la derecha en una posición dará -4. Por otra parte, el operador de desplazamiento a la derecha con relleno con ceros desplaza ceros en los bits superiores, incluyendo al de orden más alto. Cuando se aplica este desplazamiento a números negativos, el bit de orden más alto se convierte en 0 y el número se convierte en positivo.

Relacionales

El último grupo de operadores enteros son los relacionales, que operan sobre enteros pero devuelven un tipo booleanos. La tabla 13.3 lista los operadores enteros relacionales.

Tabla 13.3 Operadores enteros relacionales

Descripción	Operador
Menor que	<
Mayor que	>
Menor que o igual que	<=
Mayor que o igual que	>=
Igual que	==
Desigual que	!=

Estos operadores realizan comparaciones entre enteros. El listado 13.7 contiene el programa *Relational*, que muestra el uso de los operadores relacionales con enteros.

Listado 13.7 La clase Relational

```
class Relational {
  public static void main (String args[]) {
    int x = 7, y = 11, z = 11;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("z = " + z);
    System.out.println("x < y = " + (x < y));
    System.out.println("x > z = " + (x > z));
    System.out.println("y <= z = " + (y <= z));
    System.out.println("x >= y = " + (x >= y));
    System.out.println("y == z = " + (y == z));
    System.out.println("x != y = " + (x != z));
  }
}
```

A continuación se muestra el resultado de ejecutar *Relational*:

```
x = 7
y = 11
z = 11
x < y = true
x > z = false
y <= z = true
x >= y = false
y == z = true
x != y = true
```

Como puede ver, el método *println* es suficientemente inteligente para imprimir correctamente los resultados booleanos de *true* y *false*.

Operadores de coma flotante

De forma similar a los operadores enteros, hay tres tipos de operaciones que pueden realizarse sobre números de coma flotante: unaria, binaria y relacional. Los operadores unarios actúan sólo sobre números de coma flotante simples y los binarios sobre pares de números de coma flotante. Ambos operadores devuelven resultados en números de coma flotante. Sin embargo, los operadores relacionales actúan sobre dos números de coma flotante pero devuelven un resultado booleanos.

Los operadores unarios y binarios de coma flotante devuelven un tipo *float* si ambos operandos son de este tipo. Sin embargo, si uno o ambos operandos es del tipo *double*, el resultado de la operación es de tipo *double*.

Unarios

Los operadores de coma flotante unarios actúan sobre un solo número de coma flotante. La tabla 13.4 lista los operadores de coma flotante unarios.

Tabla 13.4 Los operadores de coma flotante unarios

Descripción	Operador
Incremento	++
Decremento	--

Como puede ver, los dos únicos operadores de coma flotante son los de incremento y decremento. Estos dos operadores, respectivamente, suman y restan 1.0 a su operando de coma flotante.

Binarios

Los operadores binarios de coma flotante actúan sobre pares de números de coma flotante. La tabla 13.5 lista los operadores de coma flotante binarios.

Tabla 13.5 Los operadores de coma flotante binarios

Descripción	Operador
Adición	+
Sustracción	-
Multiplicación	*
División	/
Módulo	%

Los operadores de coma flotante son las cuatro operaciones binarias tradicionales (+, -, *, /) más el operador de módulo (%). Puede estar preguntándose cómo encaja aquí el operador de módulo, considerando que su uso como operador entero se basaba en la división entera. Si recuerda, el operador de módulo entero devolvía el resto de una división entera de dos operandos. Pero una división de coma flotante nunca da un resto, entonces ¿qué hace un módulo de coma flotante? Devuelve el

equivalente de coma flotante de una división entera. Lo que significa que la división se lleva a cabo con ambos operandos de coma flotante, pero el divisor resultante se trata como un entero, resultando un resto de coma flotante. El listado 13.8 contiene el programa *FloatMath*, que muestra como funciona el operador de módulo de coma flotante junto con otros operadores de coma flotante binarios.

Listado 13.8 La clase FloatMath

```
class FloatMath {
  public static void main (String args[]) {
    float x = 23.5F, y = 7.3F;
    System.out.println("x = " + x);
    System.out.println("y = " + y);
    System.out.println("x + y = " + (x + y));
    System.out.println("x - y = " + (x - y));
    System.out.println("x * y = " + (x * y));
    System.out.println("x / y = " + (x / y));
    System.out.println("x % y = " + (x % y));
  }
}
```

A continuación se muestra la salida del programa *FloatMath*:

```
x = 23.5
y = 7.3
x + y = 30.8
x - y = 16.2
x * y = 171.55
x / y = 3.21918
x % y = 1.6
```

Las cuatro primeras operaciones se han realizado sin ninguna duda como esperaba, tomando los dos operandos de coma flotante y dando un resultado de coma flotante. La operación de módulo final determinó que *7,3* divide a *23,5* una cantidad entera de *3* veces, dejando un resto resultante de *1,6*.

Relacionales

Los operadores de coma flotante relacionales comparan dos operandos de coma flotante, dando un resultado booleano. Los operadores relacionales de coma flotante son los mismos que los enteros listados en la tabla 13.3, excepto que operan sobre números de coma flotante.

Los operadores booleanos

Los operadores booleanos actúan sobre tipos booleanos y devuelven un resultado Booleano. Estos operadores se listan en la tabla 13.6.

Tabla 13.6 Los operadores booleanos

Descripción	Operador
Evaluación AND	&
Evaluación OR	|

(continúa)

Tabla 13.6 Continúa

Descripción	Operador
Evaluación XOR	^
AND lógico	&&
OR lógico	||
Negación	!
Igual que	==
No igual que	!=
Condicional	?:

Los operadores de evaluación (&, | y ^) evalúan ambos lados de una expresión antes de determinar el resultado. Los operadores lógicos (&& y | |) evitan la evaluación del lado derecho de la expresión si no es necesaria. Para entender mejor la diferencia entre estos operadores, analice las siguientes expresiones:

```
boolean result = isValid & (Count > 10);
boolean result = isValid && (Count > 10);
```

La primera expresión usa el operador de evaluación AND (*&*) para realizar una asignación. En este caso, siempre se evalúan los dos lados de la expresión, independientemente de los valores de las variables implicadas. En el segundo ejemplo, se usa el operador AND lógico (*&&*), que comprueba primero el valor booleano *isValid*; si es *false*, se ignora la parte derecha de la expresión y se realiza la asignación. Esto es más eficiente porque un valor *false* en la parte izquierda de la expresión proporciona suficiente información para determinar el resultado *false*.

Aunque los operadores lógicos son más eficientes, aún puede haber momentos en los que quiera usar los operadores de evaluación para garantizar que se evalúa la expresión completa. El siguiente código muestra lo necesario que es el operador de evaluación AND para la evaluación completa de una expresión:

```
while ((++x < 10) && (++y < 15)) {
  System.out.println(x);
  System.out.println(y);
}
```

En este ejemplo, se evalúa la segunda expresión (*++y > 15*) después del último paso por el bucle, debido al operador de evaluación AND. Si se hubiese usado el operador AND lógico, no se habría evaluado la segunda expresión *y*, y no se habría incrementado después de la última pasada.

Los operadores booleanos de negación, igual que y no igual que (*!*, *==* y *!=*) se comportan de la forma que podría esperar. El operador de negación conmuta el valor de un booleano de *false* a *true* o de *true* a *false*, dependiendo del valor original. El operador *igual que* determina simplemente si dos valores booleanos son iguales (ambos *true* o *false*). De forma parecida, el operador *no igual que* determina si dos operandos booleanos son desiguales.

El operador booleano condicional (*?:*) es el más raro de los operadores booleanos, por lo que vale la pena considerarlo por un momento. Este operador también se conoce como el *operador ternario* porque toma tres elementos: una condición y dos expresiones. La sintaxis del operador condicional es la siguiente:


```
Condition ? Expression1 : Expression2
```

La *Condition*, que ella misma es booleana, se evalúa primero para determinar si es *true* o *false*. Si la *Condition* se evalúa con un resultado *true*, se evalúa la *Expression1*. Si resulta que la *Condition* es *false*, se evalúa la *Expresión2*. Para cogerle mejor el truco al operador condicional, analice el programa *Conditional* del listado 13.9.

Listado 13.9 La clase Conditional

```
class Conditional {
  public static void main (String args[]) {
    int x = 0;
    boolean isEven = false;
    System.out.println("x = " + x);
    x = isEven ? 4 : 7;
    System.out.println("x = " + x);
  }
}
```

Vea los resultados del programa Conditional:

```
x = 0
x = 7
```

Primero se asigna un valor de *0* a la variable *x*. A la variable booleana *isEven* se le asigna una valor de *false*. Se comprueba el valor de *isEven*, usando el operador condicional. Debido a que es *false*, se usa la segunda expresión de condicional, que da un resultado de *7*, que se asigna a *x*.

Operadores de cadenas

Junto con los números enteros, los de coma flotante y los booleanos, también pueden manipularse las cadenas con los operadores. Realmente, sólo hay un operador de cadenas: el operador de concatenación (+). Este operador de cadenas funciona de forma muy similar al operador de adición de números, agrega cadenas. Este operador se muestra en el programa *Concatenation* mostrado en el listado 13.10.

Listado 13.10 La clase Concatenation

```
class Concatenation {
  public static void main (String args[]) {
    String firstHalf = "What " + "did ";
    String secondHalf = "you " + "say?";
    System.out.println(firstHalf + secondHalf);
  }
}
```

A continuación se muestra el resultado de *Concatenation*:

```
What did you say?
```

En el programa de *Concatenation*, se concatenan las cadenas de literales para realizar asignaciones a las dos variables, *firstHalf* y *secondHalf*, en el momento de su creación. Después se concatenan las dos variables de cadena dentro de la llamada al método *println*.

Operadores de asignación

Un grupo final de operadores aún no estudiado es el de los operadores de asignación. Éstos funcionan realmente con todos los tipos de datos fundamentales. La tabla 13.7 lista estos operadores.

Tabla 13.7 Los operadores de asignación

Descripción	Operador
Simple	=
Adición	+=
Sustracción	-=
Multiplicación	*=
División	/=
Módulo	%=
AND	&=
OR	|=
XOR	^=

Con la excepción del operador de asignación simple (=), los operadores de asignación funcionan exactamente igual que sus colegas de no asignación, excepto que el valor resultante se almacena en el operando de la parte izquierda de la expresión. Vea los siguientes ejemplos:

```
x += 6;
x *= (y - 3);
```

En el primer ejemplo, se suman *x* y *6*, y el resultado se almacena en x. En el segundo, se resta *3* de *y*, y el resultado se multiplica por *x*. El resultado final se almacena en *x*.

Estructuras de control

Aunque es muy útil realizar operaciones con datos, ya es hora de abordar el tema del control del flujo del programa, que se determina a partir de dos tipos de construcciones: las ramas y los bucles. Éstas le permiten ejecutar selectivamente una parte de un programa en lugar de otra; los bucles le ofrecen un medio de repetir ciertas partes de un programa. Juntos le proporcionan un medio potente de controlar la lógica y la ejecución de su código.

Ramas

Sin ramas ni bucles, el código Java se ejecuta de forma secuencial, como se muestra en la figura 13.1.

En esta figura, cada sentencia se ejecuta secuencialmente. ¿Qué sucede si no quiere que se ejecuten siempre absolutamente todas las sentencias? En este caso use una rama. La figura 13.2 muestra cómo una rama condicional da más opciones al flujo de su código.

Fig. 13.1
Un programa que se ejecuta secuencialmente

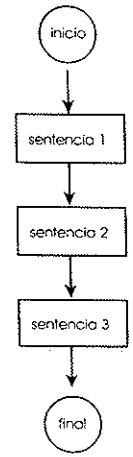
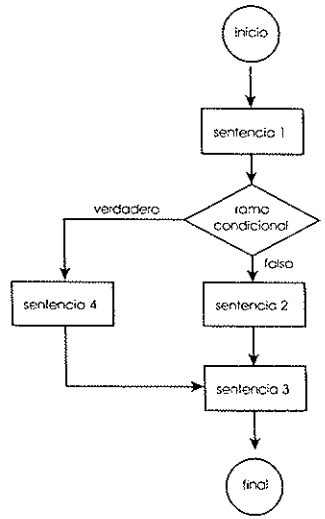


Fig. 13.2
Un programa que se ejecuta con una rama



Al añadir una rama, ha dado al código dos rutas opcionales, en función del resultado de una expresión condicional. El concepto de rama puede parecer trivial, pero sería difícil, por no decir imposible, escribir programas útiles sin ellas. Java admite dos tipos de ramas: las *if-else* y las *switch*.

if-else

Es la usada más comúnmente en la programación Java. Sirve para seleccionar condicionalmente uno de dos posibles resultados. La sintaxis de la sentencia *if-else* es la siguiente:

```
if (Condición)
  Sentencia1
else
  Sentencia2
```

Si la *Condición* booleana se evalúa como *true*, se ejecuta la *Sentencia1*. Asimismo, si la *Condición* se evalúa como *false*, se ejecuta la *Sentencia2*. El siguiente ejemplo debería aclararlo mejor:

```
if (isTired)
  timeToEat = true;
else
  timeToEat = false;
```

Si la variable booleana isTired es *true*, se ejecuta la primera sentencia y se pone *timeToEat* en *true*. En caso contrario, se ejecuta la segunda sentencia y se pone *timeToEat* en *false*. Puede haber observado que la rama *if-else* funciona de forma muy parecida al operador condicional (*?:*) que vio anteriormente. En efecto, puede imaginar la rama *if-else* como una versión expandida del operador condicional. Una diferencia significativa entre las dos es que en la rama *if-else* puede incluir sentencias compuestas .

Nota

Las *sentencias compuestas* son bloques de código rodeados por señales de llave {}, que aparecen como una sentencia única, o simple, frente a un bloque exterior de código.

Si tiene una sola sentencia que necesita ejecutar de forma condicional, puede olvidarse de la parte *else* de la rama, como se muestra en el ejemplo siguiente:

```
if (isThirsty)
  pourADrink = true;
```

Por otra parte, si necesita más de dos resultados condicionales, puede encadenar una serie de ramas *if-else* que se usan para conmutar entre resultados distintos:

```
if (x == 0)
  y = 5;
else if (x == 2)
  y = 25;
else if (x >= 3)
  y = 125;
```

En este ejemplo se realizan tres comparaciones distintas, cada una con su propia sentencia, que se ejecuta en caso de un resultado condicional *true*. Sin embargo, observe que las ramas *if-else* subsiguientes están efectivamente anidadas dentro de la rama primera. Esto garantiza que como mínimo se ejecutará una sentencia.

El último tema importante a analizar respecto a las ramas *if-else* es el de las sentencias compuestas. Tal como se ha mencionado antes, una sentencia compuesta es un bloque de código rodeado por llaves que aparece como una sola sentencia frente a un bloque exterior. El siguiente es un ejemplo de una sentencia compuesta usada con una rama *if*:

```
if (performCalc) {
   x += y * 5;
   y -= 10;
   z = (x - 3) / y;
}
```


A veces, cuando se anidan ramas *if-else*, es necesario usar llaves para distinguir qué sentencias van con cada rama. El siguiente ejemplo muestra el problema:

```
if (x != 0)
  if (y < 10)
    z = 5;
else
  z = 7;
```

En este ejemplo, el estilo de sangrado indica que la rama *else* pertenece al primer (exterior) *if*. Sin embargo, debido a que no se ha indicado ninguna agrupación, el compilador de Java asume que el *else* va con el *if* interior. Para obtener los resultados deseados, ha de modificar el código de la siguiente manera:

```
if (x != 0) {
  if (y < 10)
    z = 5;
}
else
  z = 7;
```

Al añadir las llaves se informa al compilador que el *if* interior forma parte de una sentencia compuesta y, lo que es más importante, esconde completamente la rama *else* al *if* interior. Basándose en lo que ha aprendido de la descripción de los bloques y el ámbito en el capítulo anterior, puede ver que el código dentro del *if* interior no tiene forma de acceder al código fuera de su ámbito, incluyendo la rama *else*.

El listado 13.11 contiene el código fuente de la clase *IfElseName*, que usa mucho de lo que ha aprendido hasta este momento.

Listado 13.11 La clase IfElseName

```
class IfElseName {
  public static void main (String args[]) {
    char firstInitial = (char)-1;
    System.out.println("Enter your first initial:");
    try {
     firstInitial = (char)System.in.read();
    }
    catch (Exception e) {
     System.out.println("Error: " + e.toString());
    }
    if (firstInitial == -1)
     System.out.println("Now what kind of name is that?");
    else if (firstInitial == 'j')
     System.out.println("Your name must be Jules!");
    else if (firstInitial == 'v')
     System.out.println("Your name must be Vincent!");
    else if (firstInitial == 'z')
     System.out.println("Your name must be Zed!");
    else
     System.out.println("I can't figure out your name!");
  }
```

Cuando escriba la letra *v* en respuesta al mensaje de entrada, *IfElseName* genera los siguientes resultados:

```
Your name must be Vicente!
```

Supongo que en la clase *IfElseName* habrá descubierto el método *read* y se estará preguntando qué es. El método *read* lee simplemente un carácter del flujo de entrada estándar (*System.in*), que es típicamente el teclado. Observe que se usa una conversión de tipo porque *read* devuelve un tipo *int*. Una vez recuperado con éxito el carácter de entrada, se usa una sucesión de ramas *if-else* para determinar la salida apropiada. Si no se produce ninguna concordancia, se ejecuta la rama *else* final, que notifica al usuario que su nombre no pudo determinarse. Observe que el valor leído se comprueba si es igual a -1. El método *read* devuelve -1 si ha alcanzado el final del flujo de entrada.

Nota

Puede haber advertido que la llamada al método read de IfElseName está encerrada dentro de una cláusula try-catch. Esta cláusula es parte del soporte de Java al manejo de excepciones y se usa en este caso para capturar los errores que se encuentren mientras se lee la entrada del usuario. Aprenderá más cosas sobre excepciones y la cláusula try-catch en el capítulo 16, "Manejo de excepciones".

Switch

Parecida a la rama *if-else*, la rama *switch* está diseñada específicamente para conmutar condicionalmente entre múltiples resultados. La sintaxis de la sentencia *switch* es la siguiente:

```
switch (Expresión) {
  case Constante1:
    ListasSentencia1
  case Constante2:
    listaSentencia2
...
  default:
    ListaSentenciaPredeterminada
}
```

La rama *switch* evalúa y compara *Expresión* con todas las constantes *case* y bifurca la ejecución del programa a la lista de sentencias con un *case* concordante. Si no hay ninguna constante *case* que concuerde con *Expresión*, el programa se bifurca a la *ListaSentenciaPredeterminada*, si se ha dado alguna (la *ListaSentenciaPredeterminada* es opcional). Quizás se pregunte qué es una lista de sentencias. Simplemente es una serie, o *lista*, de sentencias. A diferencia de la rama *if-else*, que dirige el flujo del programa a una sentencia única o compuesta, la rama *switch* dirige el flujo a una lista se sentencias.

Cuando la ejecución del programa entra en una lista de sentencias case, continúa desde ahí de forma secuencial. Para entenderlo mejor vea el listado 13.12, que contiene una versión *switch* del programa de nombres que ha desarrollado antes con las ramas *if-else*.

Listado 13.12 La clase SwitchName1

```
class SwitchName1 {
  public static void main (String args[]) {
    char firstInitial = (char)-1;
    System.out.println("Enter your first initial:");
    try {
     firstInitial = (char)System.in.read();
    }
    catch (Exception e) {
     System.out.println("Error: " + e.toString());
    }
    switch(firstInitial) {
      case (char)-1:
       System.out.println("Now what kind of name is that?");
      case 'j':
       System.out.println("Your name must be Jules!");
      case 'v':
       System.out.println("Your name must be Vincent!");
      case 'z':
       System.out.println("Your name must be Zed!");
      default:
       System.out.println("I can't figure out your name!");
    }
  }
}
```

(continúa)

Cuando escriba la letra *v* en respuesta al mensaje de entrada, *SwitchName1* presenta los siguientes resultados:

```
Your name must be Vincent!
Your name must be Zed!
I can't figure out your name!
```

¡Eh!, ¿qué está pasando? Definitivamente esta salida no tiene buen aspecto. El problema se encuentra en la forma que la rama *switch* controla el flujo del programa. La rama switch comparó la v introducida con la sentencia *case* correcta, como se muestra en la primera cadena impresa. Sin embargo, el programa continuó ejecutando las sentencias *case* desde ese punto hacia adelante, que no es lo que quería. La solución al problema se encuentra en la sentencia *break*. Ésta fuerza a que un programa interrumpa el bloque de código que éste ejecutando en ese momento. Vea la nueva versión del programa en el listado 13.13, con las sentencias *break* añadidas en los lugares apropiados.

Listado 13.13 La clase SwitchName2

```
class SwitchName2 {
  public static void main (String args[]) {
    char firstInitial = (char)-1;
    System.out.println("Enter your first initial:");
    try {
     firstInitial = (char)System.in.read();
    }
    catch (Exception e) {
     System.out.println("Error: " + e.toString());
    }
    switch(firstInitial) {
      case (char)-1:
       System.out.println("Now what kind of name is that?");
       break;
```

(continúa)

```
        case 'j':
         System.out.println("Your name must be Jules!");
          break;
        case 'v':
         System.out.println("Your name must be Vincent!");
          break;
        case 'z':
         System.out.println("Your name must be Zed!");
          break;
        default:
         System.out.println("I can't figure out your name!");
       }
     }
   }
```

Cuando ejecute *SwitchName2* e introduzca v, obtendrá la siguiente salida:

```
Your name must be Vincent!
```

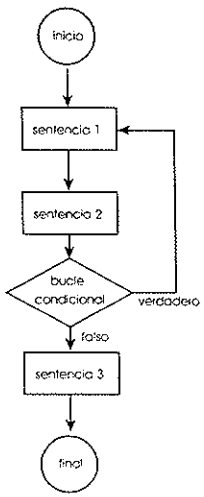
¡Esto está mucho mejor! Puede ver que al colocar las sentencias *break* después de cada sentencia *case* se evitó que el programa continuase por las sentencias case siguientes. Aunque la mayoría de las veces usará sentencias *break* de esta forma, puede haber situaciones en las que quiera que el programa continúe de una sentencia *case* a la siguiente.

Bucles

Cuando se trata del flujo de un programa, las ramas sólo le cuentan la mitad de la historia; los bucles le cuentan la otra mitad. En pocas palabras, los bucles le permiten ejecutar un código de forma repetida. Hay tres tipos de bucles en Java: *for*, *while* y *do-while*.

Al igual que las ramas alteran el flujo secuencial de los programas, también lo hacen los bucles. La figura 13.3 muestra cómo un bucle altera el flujo secuencial de un programa Java.

Fig. 13.3
Un programa que se ejecuta con un bucle.



for

El bucle *for* proporciona un medio de repetir una sección de código un número determinado de veces. El bucle *for* se estructura de forma que se repita una sección de código hasta que se alcance cierto límite. La sintaxis de la sentencia *for* es la siguiente:

```
for (ExpresiónInicialización; CondiciónBucle; ExpresiónPaso)
  Sentencia
```

El bucle *for* repite la *Sentencia* el número de veces determinado por la *ExpresiónInicialización*, la *CondiciónBucle* y la *ExpresiónPaso*. La *ExpresiónInicialización* se usa para inicializar una variable de control del bucle. La *CondiciónBucle* compara la variable de control del bucle con algún valor límite. Por último, la *ExpresiónPaso* indica cómo debería ser modificada la variable de control del bucle antes de la siguiente iteración. El siguiente ejemplo muestra cómo puede usarse un bucle *for* para imprimir números del uno al diez:

```
for (int i = 1; i < 11; i++)
  System.out.println(i);
```

Primero, se declara *i* como un entero. El hecho de que *i* se declare dentro del cuerpo del bucle *for* puede parecerle extraño en este punto. No desespere, es completamente legal. i se inicializa a *1* en la parte de *ExpresiónInicialización* del bucle *for*. Después se evalúa la expresión condicional $i < 11$ para ver si el bucle debe continuar. En este punto, *i* aún vale *1*, de forma que la *CondiciónBucle* se evalúa como *true* y se ejecuta la *Sentencia* (se imprime el valor de *i* en la salida estándar). Se incrementa i en la parte *ExpresiónPaso* del bucle *for* y el proceso vuelve a comenzar con la evaluación de la *CondiciónBucle*. Esto continúa hasta que se evalúa *CondiciónBucle* como *false*, lo que se produce cuando *x* es igual a *11* (diez iteraciones más tarde).

El listado 13.14 muestra el programa *ForCount*, que muestra cómo usar un bucle *for* para contar una determinada cantidad de números introducida por el usuario.

Listado 13.14 La clase ForCount

```
class ForCount {
  public static void main (String args[]) {
    char input = (char)-1;
    int numToCount;
    System.out.println("Enter a number to count to between 0 and 10:");
    try {
     input = (char)System.in.read();
    }
    catch (Exception e) {
     System.out.println("Error: " + e.toString());
    }
    numToCount = Character.digit(input, 10);
    if ((numToCount > 0) && (numToCount < 10)) {
     for (int i = 1; i <= numToCount; i++)
      System.out.println(i);
    }
    else
     System.out.println("That number was not between 0 and 10!");
  }
}
```

Cuando se ejecuta el programa y se introduce el número *4*, se produce la siguiente salida:

```
1
2
3
4
```

Primero, *ForCount* indica al usuario que introduzca un número entre cero y diez. Se lee un carácter del teclado usando el método *read* y el resultado se almacena en la variable de carácter *input*. El método estático *digit* de la clase *Character* se usa para convertir el carácter a su representación entera de base 10. Este valor se almacena en la variable entera *numToCount*. Se comprueba esta variable para asegurarse de que está en el rango de cero a diez. Si es así, se ejecuta el bucle *for* que cuenta de *1* a *numToCount*, imprimiendo mientras tanto cada número. Si *numToCount* está fuera del rango válido, se imprime un mensaje de error.

Antes de seguir adelante veamos un pequeño problema con *ForCount* que puede habérsele pasado por alto. Ejecútelo e intente escribir un número mayor que nueve. ¿Qué sucede con el mensaje de error? El problema es que *ForCount* sólo toma el primer carácter que ve de la entrada. Así, si escribe *300*, sólo obtendrá el *3* y creerá que todo está bien. Por ahora no necesita preocuparse en arreglar este problema, ya que se resolverá cuando aprenda más sobre entradas y salidas en el capítulo 20, "El paquete de E/S".

while

Al igual que el bucle *for*, *while* tiene una condición de bucle que controla la ejecución de la sentencia de bucle. A diferencia del bucle *for*, *while* no tiene inicialización ni expresiones de pasos. La sintaxis de la sentencia *while* es:

```
while (CondiciónBucle)
  Sentencia
```

Si se evalúa como *true* la *CondiciónBucle* booleana, se ejecuta la *Sentencia* y el proceso vuelve a comenzar. Es importante entender que no hay ninguna expresión de paso, como en el bucle *for*. Esto significa que la *CondiciónBucle* debe ser afectada de alguna forma por el código de la *Sentencia* o si no se repetirá el bucle infinitamente, lo que es malo, porque un bucle infinito provoca que el programa nunca salga, lo que acapara el tiempo de proceso y puede acabar colgando el sistema.

Otro aspecto importante del bucle *while* es que su *CondiciónBucle* se produce antes del cuerpo de la *Sentencia* del bucle. Esto significa que si inicialmente la *CondiciónBucle* se evalúa como false, no se ejecutará nunca la *Sentencia*. Puede parecerle trivial, pero es, en efecto, la única cosa que diferencia el bucle *while* del *do-while*, que se describe en la sección siguiente.

Para entender mejor cómo funciona el bucle *while*, vea el listado 13.15, que muestra cómo funciona un programa de contar usando un bucle *while*.

Listado 13.15 La clase WhileCount

```
class WhileCount {
 public static void main (String args[]) {
   char input = (char)-1;
   int  numToCount;
   System.out.println("Enter a number to count to between 0 and 10:");
```

(continúa)

Listado 13.15 continúa

```
   try {
     input = (char)System.in.read();
    }
    catch (Exception e) {
     System.out.println("Error: " + e.toString());
    }
   numToCount = Character.digit(input, 10);
   if ((numToCount > 0) && (numToCount < 10)) {
     int i = 1;
     while (i <= numToCount) {
       System.out.println(i);
        i++;
      }
    }
    else
     System.out.println("That number was not between 0 and 10!");
  }
}
```

Sin discusión, *WhileCount* no demuestra el mejor uso de un bucle *while*. Los bucles que implican contar deberían implementarse siempre con bucles *for*. Sin embargo, ver cómo puede hacer que un bucle *while* imite a un bucle *for* puede darle una visión de las diferencias estructurales entre ambos.

Debido a que los bucles *while* no tienen ningún tipo de expresión de inicialización, primero tiene que declarar e inicializar la variable *i* a *1*. Después, la condición del bucle *while* se establece como *i* <= *numToCount*. Dentro de la sentencia *while* compuesta, puede ver una llamada al método *println*, que produce la salida del valor de i. Por último, se incrementa *i* y se reinicia la ejecución del programa en la condición del bucle *while*.

do-while

El bucle *do-while* es muy semejante al *while*, como puede ver en la sintaxis siguiente:

```
do
  Sentencia
while (Condición);
```

La diferencia principal entre el bucle *do-while* y el *while* es que la *CondiciónBucle* se evalúa después de que se haya ejecutado la *Sentencia*. Esta diferencia es importante porque a veces deseará que el código *Sentencia* se ejecute como mínimo una vez, independientemente de la *CondiciónBucle*.

La *Sentencia* se ejecuta inicialmente y desde ese momento se va ejecutando mientras se evalúe la *CondiciónBucle* como *true*. Al igual que el bucle *while*, deber tener cuidado con el bucle *do-while* para evitar crear un bucle infinito, que se produce cuando la *CondiciónBucle* permanece siempre true, indefinidamente. El siguiente ejemplo muestra un bucle *do-while* infinito muy obvio:

```
do
  System.out.println ("I'm stuck!");
while (true);
```

Debido a que la *CondiciónBucle* siempre es cierta, se imprime siempre el mensaje *I'm Stuck!* o, como mínimo, hasta que pulse <Ctrl>+<C> e interrumpa el programa.

break y continue

Ya ha visto cómo funciona la sentencia *break* en relación con la rama *switch*. La sentencia *break* también es útil cuando se trata de bucles. Puede usarla para saltar fuera de un bucle y hacer efectivamente un bypass de la condición de bucle. El listado 13.6 muestra como puede ayudarle la sentencia *break* para salir fuera el problema de bucle infinito mostrado antes.

Listado 13.16 La clase BreakLoop

```
class BreakLoop {
  public static void main (String args[]) {
    int i = 0;
    do {
     System.out.println("I'm stuck!");
      i++;
      if (i > 100)
        break;
     }
    while (true);
   }
 }
```

En *Breakloop*, se crea aparentemente un bucle *do-while* infinito, poniendo la condición de bucle a *true*. Sin embargo, se usa la sentencia *break* para salir del bucle cuando *i* se incremente a más de *100*.

Otra sentencia útil que funciona de forma similar a la sentencia *break* es *continue*: a diferencia de *break*, sólo es útil cuando se trabaja con bucles y no tiene ninguna verdadera aplicación en la rama *switch*. La sentencia *continue* funciona igual que *break* en lo que respecta a que salta fuera de la iteración del momento de un bucle, se diferencia con *continue* en que la ejecución del programa se devuelve a la condición de prueba del bucle. Recuerde, *break* salta totalmente fuera de un bucle. Use *break* cuando quiera saltar fuera y terminar un bucle, y use *continue* cuando quiera saltar inmediatamente a la próxima iteración del bucle.

Resumen

Hemos avanzado mucho terreno en este capítulo. Comenzó aprendiendo expresiones y después fue directamente a los operadores, aprendiendo como funcionan y como afectan a cada tipo de datos. No se arrepentirá del tiempo gastado trabajando con los operadores de este capítulo, están en el núcleo de casi toda expresión matemática o expresión lógica de Java.

De los operadores pasó a las estructuras de control, aprendiendo los diversos tipos de ramas y bucles, que proporcionan los medios de alterar el flujo de los programas Java y tan importantes como los operadores en todo el reino de la programación Java.

Con todos estos conceptos firmemente establecidos en su mente, está preparado para profundizar un poco más en Java. Siguiente parada: ¡programación orientada a objetos con clases, paquetes e interfaces!

Capítulo 14

Clases, paquetes e interfaces

por Michael Morrison

Contenido del capítulo

- Introducción a la programación orientada a objetos 179
- Clases 186
- Creación de objetos 194
- Destrucción de objetos 196
- Paquete 196
- Interfaces 198

Hasta el momento ha logrado evitar el tema de la programación orientada a objetos y cómo se relaciona con Java. Este capítulo quiere remediar este problema. Comienza con una discusión básica sobre la programación orientada a objetos en general. Con estos conocimientos bien asentados, puede circular por el resto del capítulo, que cubre los elementos específicos del lenguaje Java proporcionando el soporte para la programación orientada a objetos, es decir, las clases, los paquetes y las interfaces.

Puede considerar este capítulo como el aprendizaje del lenguaje Java. Las clases son el componente nuclear último del lenguaje Java que ha de aprender para ser un programador de Java competente. Una vez que tenga un conocimiento sólido de las clases y como funcionan en Java, estará preparado para escribir algunos programas Java importantes. Así pues, ¿a qué espera? ¡Adelante!

Introducción a la programación orientada a objetos

Dondequiera que en los últimos cinco años haya estado, cerca de la sección de informática de una librería o si ha cogido una revista de programación, sin duda habrá visto la programación orientada a objetos rodeada de un halo de secretismo. Es la tecnología de programación más popular, pero menos comprendida, de los últimos tiempos. Esta tecnología gira alrededor del concepto de *objeto*.

Puede estar preguntándose qué es lo que sucede con los objetos y la tecnología orientada a objetos. ¿Es algo de lo que tendría que preocuparse?, y, si es así, ¿por qué? Si examina cuidadosamente el secretismo que rodea todo el tema de la orien-

tación a objetos, encontrará una tecnología potente que proporciona muchas ventajas para el diseño de software. El problema es que los conceptos orientados a objetos pueden ser difíciles de comprender. Y no puede aprovechar las ventajas del diseño orientado a objetos si no entiende completamente lo que son. Debido a esto, usualmente se desarrolla con el tiempo y la práctica una comprensión completa de la teoría que está detrás de la programación orientada a objetos.

Una gran confusión acerca de la tecnología orientada a objetos entre los que realizan desarrollos ha llenado de confusión a los usuarios de computadores en general. ¿Cuántos productos ha visto que dicen ser orientados a objetos? Actualmente, considerando el hecho de que la orientación a objetos es un tema de diseño de software, ¿qué puede significar posiblemente esta frase para un consumidor de software? De muchas formas, "orientado a objetos" se ha convertido para el sector de software en lo que "nuevo y mejorado" es para el sector de la limpieza de hogares. La verdad es que el mundo real ya es orientado a objetos, lo que no es una sorpresa para nadie. La importancia de la tecnología orientada a objetos es que permite que los programadores diseñen software de una forma muy parecida a como perciben el mundo real.

Ahora que ha aclarado algunos de los malentendidos que rodean al tema de la orientación a objetos, intente apartarlos y piense en lo que puede significar el término orientado a objetos para el diseño de software. Esta introducción pone los cimientos para la comprensión de cómo el diseño orientado a objetos puede hacer que la programación sea más rápida, fácil y fiable. Y todo comienza con el objeto. Aunque este capítulo se centra en el fondo en Java, esta sección de introducción a la orientación a objetos es válida realmente para todos los lenguajes de este tipo.

Objetos

Los objetos son conjuntos de software de datos y de procedimientos que actúan sobre esos datos. Los procedimientos se conocen también como métodos. La mezcla de datos y métodos proporciona un medio más exacto de representar objetos del mundo real en software. Sin los objetos, el modelado de un problema del mundo real en software necesita realizar un salto lógico significativo. Por otra parte, los objetos permiten que los programadores solucionen problemas del mundo real en el terreno del software de una forma más fácil y lógica.

Como su nombre indica, los objetos ocupan el corazón de la tecnología orientada a objetos. Para entender las ventajas de los objetos de software, piense en las características comunes de los objetos del mundo real. Los leones, los coches y las calculadoras comparten todos dos características comunes: estado y comportamiento. Por ejemplo, el estado de un león puede incluir el color, el peso y si está cansado o hambriento. Además, los leones tienen ciertos comportamientos, como rugir, dormir y cazar. El estado de un coche incluye la velocidad del momento, el tipo de transmisión, si tiene tracción en dos o en cuatro ruedas, si las luces están encendidas y la marcha del momento, entre otras cosas. Los comportamientos de un coche incluyen girar, frenar y acelerar.

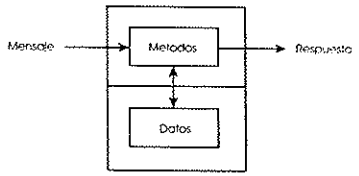
Al igual que los objetos del mundo real, los objetos de software también tienen estas dos características comunes (estado y comportamiento). Para ponerlo en términos de programación, el estado de un objeto está determinado por sus datos y su comportamiento está definido por sus métodos. Al realizar esta conexión entre objetos del mundo real y objetos de software, comienza a ver como los objetos ayudan a

hacer de puente sobre el espacio que hay entre el mundo real y el mundo de software dentro de su computador.

Debido a que los objetos de software están modelados según objetos del mundo real, puede representar más fácilmente los objetos del mundo real en programas orientados a objetos. Podría usar el objeto león para representar un león real en un zoo de software interactivo. De forma similar, los objetos coche serían muy útiles en un juego de carreras. Sin embargo, no es necesario que siempre piense que los objetos de software modelan objetos del mundo real físico; los objetos de software pueden ser igual de útiles en el modelado de conceptos abstractos. Por ejemplo, un subproceso es un objeto usado en los sistemas de software con multisubprocesos, que representa un flujo de ejecución de un programa. Aprenderá mucho más sobre subprocesos y cómo se usan en Java en el siguiente capítulo, "Subprocesos y multisubproceso".

La figura 14.1 muestra la visualización de un objeto de software, incluyendo los componentes primarios y cómo se relacionan.

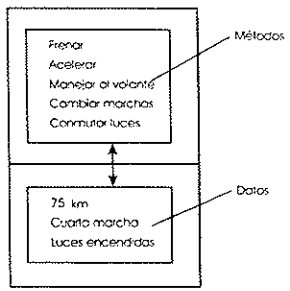
Fig. 14.1
Un objeto de software.



El objeto de software de la figura 14.1 muestra claramente los dos componentes primarios de un objeto: los datos y los métodos. La figura también muestra algún tipo de comunicación, o acceso, entre los datos y los métodos. Además, muestra cómo se envían mensajes a través de los métodos, lo que resulta en respuestas del objeto. Aprenderá más cosas sobre mensajes y respuestas un poco más adelante en este capítulo.

Los datos y los métodos dentro de un objeto expresan todo lo que un objeto representa (estado) junto con todo lo que puede hacer (comportamiento). Un objeto de software que modele un coche del mundo real tendría variables (datos) que indican el estado actual del coche: está viajando a 75 km., circula con la cuarta marcha y las luces están encendidas. El objeto coche de software también tendría métodos que le permitirían frenar, acelerar, manejar el volante, cambiar marchas y encender y apagar las luces. La figura 14.2 muestra como podría ser un objeto coche de software.

Fig. 14.2
Un objeto coche de software.



En las Figuras 14.1 y 14.2, probablemente, ha observado la línea que separa los métodos de los datos dentro de un objeto. Esta línea es un poco equívoca porque los métodos tienen acceso total a los datos interiores de un objeto. La línea está ahí para mostrar la diferencia entre la visibilidad de los métodos y la de los datos respecto al exterior. En este sentido, la visibilidad de un objeto se refiere a qué partes de un objeto tiene acceso otro objeto. Debido a que los datos de un objeto, por defecto, son invisibles o inaccesibles a otros objetos, toda interacción entre objetos se ha de realizar por medio de los métodos. Esta ocultación de los datos dentro de un objeto se denomina encapsulación.

Encapsulación

La *encapsulación* es el proceso de empaquetar los datos de un objeto junto con sus métodos. Una gran ventaja de la encapsulación es la ocultación de los detalles de implementación frente a otros objetos. Esto significa que la parte interna de un objeto tiene una visibilidad más limitada que la parte externa, lo que origina una protección de la parte interna frente a accesos externos no deseados.

La parte externa de un objeto se conoce frecuentemente como la *interfaz* del objeto, debido a que actúa como una interfaz del objeto con el resto del programa. Debido a que los otros objetos sólo se pueden comunicar con el objeto a través de su interfaz, la parte interna del objeto está protegida contra la manipulación exterior. Y como un programa exterior no tiene ningún acceso a la implementación interna de un objeto, ésta puede cambiar en cualquier momento sin afectar a otras partes del programa.

Por lo tanto, la encapsulación proporciona dos ventajas principales a los programadores:

- Ocultación de la implementación
- Modularidad

La *ocultación de la implementación* se refiere a la protección de la implementación interna de un objeto. Un objeto se compone de una interfaz pública y de una sección privada que puede ser una combinacion de datos internos y métodos. Los datos internos y los métodos son las secciones del objeto oculto. La ventaja principal estriba en que estas secciones pueden cambiar sin afectar a otras partes del programa.

La *modularidad* significa que un objeto puede mantenerse independientemente de otros objetos. Debido a que el código fuente de las secciones internas de un objeto se mantiene separado de la interfaz, es libre de realizar modificaciones con la confianza de que su objeto no causará problemas. Esto hace que sea más fácil distribuir objetos por todo el sistema.

Mensajes

Un objeto que actúa solo raras veces es muy útil; la mayoría de los objetos necesitan a otros objetos para hacer casi todas las cosas. Por ejemplo, un objeto coche es bastante inútil por sí mismo sin otras interacciones. Sin embargo, añada un objeto conductor ¡y las cosas se ponen más interesantes! Así pues, está bastante claro que los objetos necesitan algún tipo de mecanismo de comunicación para interaccionar entre sí.

Los objetos de software interaccionan y se comunican entre sí por medio de mensajes. Cuando el objeto conductor quiere que el objeto coche acelere, envía al objeto coche un mensaje. Si quiere imaginarse los mensajes de una forma más literal, imagínese dos personas como objetos, cuando una quiere que la otra se acerque, le envía un mensaje. Más exactamente, puede decir a la otra persona "Ven aquí, por favor". Este es un mensaje en un sentido muy literal. Los mensajes de software son algo diferentes en forma, pero no en teoría: le dicen a un objeto lo que ha de hacer.

Muchas veces el objeto receptor necesita, junto con el mensaje, más información, para saber exactamente qué ha de hacer. Cuando el conductor le dice al coche que acelere, el coche ha de saber cuánto. Esta información se pasa junto al mensaje como *parámetros del mensaje*.

De esta explicación puede ver que los mensajes se componen de tres cosas:

1. El objeto que recibe el mensaje (el coche)
2. El nombre de la acción a realizar (acelerar)
3. Cualquier parámetro que necesite el método (75 km)

Estos tres componentes proporcionan la información suficiente para describir totalmente un mensaje para un objeto. Cualquier interacción con un objeto se maneja pasando un mensaje, es decir, que los objetos de cualquier lugar en un sistema pueden comunicarse con otros objetos sólo a través de mensajes.

Así pues, no se confunda, entienda que un "pase de mensajes" es otra forma de decir una "llamada a métodos". Cuando un objeto envía un mensaje a otro objeto, realmente está llamando a un método del objeto. Los parámetros del mensaje son realmente los parámetros para el método. En programación orientada a objetos, los mensajes y los métodos son sinónimos.

Debido a que todo lo que puede hacer un objeto se expresa por medio de sus métodos (interfaz), el traspaso de mensajes admite todas las interacciones posibles entre objetos. En efecto, las interfaces posibilitan que los objetos envíen y reciban mensajes entre sí, incluso si residen en diferentes lugares de una red. Los objetos en este entorno se denominan *objetos distribuidos*. Java está diseñado específicamente para admitir objetos distribuidos.

Clases

A lo largo de toda esta explicación sobre programación orientada a objetos, sólo ha tocado el concepto de objeto ya existente en el sistema. Puede estar preguntándose cómo aparecen los objetos en el sistema la primera vez. Esta pregunta nos lleva a la estructura fundamental de la programación orientada a objetos: la clase. Una *clase* es una plantilla o un prototipo que define un tipo de objeto, es respecto a un objeto lo que un plano original a una casa. Pueden construirse muchas casas con un sólo plano original; el plano traza las líneas generales de la construcción de una casa. Las clases funcionan de la misma manera, excepto en que trazan las líneas generales de la construcción de un objeto.

En el mundo real hay a menudo muchos objetos del mismo tipo. Siguiendo la analogía de la casa, hay muchas casas diferentes en todo el mundo, pero todas comparten características comunes. En términos de orientación a objetos, diría que su casa es un caso particular de la clase de objetos conocida como casa. Todas tienen estados y comportamientos en común que las definen como casas. Cuando un construc-

tor comienza a construir un nuevo barrio de casas, normalmente las construye a partir de un conjunto de planos originales. No sería tan eficiente crear un nuevo plano para cada casa, especialmente cuando hay tantas similitudes compartidas entre ellas. Lo mismo ocurre con el desarrollo de software orientado a objetos; ¿por qué volver a escribir toneladas de código cuando puede volver a utilizar un código que soluciona problemas similares?

En la programación orientada a objetos, como en la construcción, también es común tener muchos objetos del mismo tipo que comparten características similares. Y, al igual que los planos de casas similares, puede crear planos para objetos que compartan ciertas características. Todo esto se reduce a que las clases son planos de software para objetos.

Por ejemplo, la clase coche discutida antes contendría diversas variables que representan el estado del coche, junto con implementaciones para los métodos que permiten al conductor controlar el coche. Las variables de estado del coche permanecen ocultas bajo la interfaz. Cada caso (caso del objeto) de la clase coche tiene un nuevo conjunto de variables de estado. Esto nos lleva a otro punto importante: cuando se crea un caso de un objeto a partir de una clase, las variables declaradas por esa clase se asignan en la memoria. Después, las variables se modifican mediante los métodos del objeto. Los casos de la misma clase comparten las implementaciones de métodos pero tienen sus propios datos de objeto.

Mientras que los objetos proporcionan las ventajas de la modularidad y la ocultación de información, las clases proporcionan la ventaja de la reutilización. Al igual que el constructor, que reutiliza un plano para una casa, el que desarrolla software reutiliza las clases para un objeto. Los programadores de software pueden usar una clase una y otra vez para crear muchos objetos. Cada uno de estos objetos tiene sus propios datos pero comparte una sola implementación de método.

Herencia

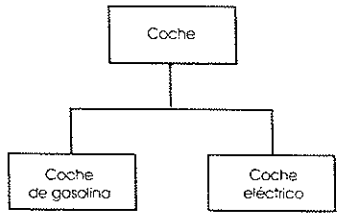
Entonces, ¿qué ocurre si quiere un objeto muy similar a otro que ya tiene, pero con algunas características más? Sólo tiene que heredar una nueva clase a partir de la clase del objeto similar. La herencia es el proceso de crear una nueva clase con las características de una ya existente, junto con características adicionales específicas de la nueva clase. La herencia proporciona un mecanismo potente y natural de organización y estructuración de programas.

Hasta este momento, la discusión sobre las clases se ha limitado a los datos y los métodos que la constituyen. Basándonos en este conocimiento, todas las clases se construyen a partir de la nada definiendo todos los datos y todos los métodos asociados. La herencia proporciona un medio de crear clases basadas en otras clases. Cuando una clase se basa en otra, hereda todas las propiedades de ésta, incluyendo sus datos y métodos. La clase que recibe la herencia se denomina la subclase (clase subordinada) y la clase que proporciona la información a heredar se denomina la superclase (clase superior).

Usando el ejemplo del coche, podrían heredarse clases subordinadas a partir de la clase coche para los coches de gasolina y para los eléctricos. Ambas clases de coche nuevas comparten características de "coche" comunes, pero también añaden unas pocas características propias. El coche de gasolina podría añadir, entre otras cosas, un depósito y un tapón para la gasolina, mientras que el coche eléctrico podría añadir una batería y un enchufe para recargarla. Cada subclase hereda información

de estado (en la forma de declaraciones de variables) de la superclase. La figura 14.3 muestra la clase superior coche con las clases subordinadas coche de gasolina y eléctrico.

Fig. 14.3
Objetos coche heredados



La sola herencia del estado y de los comportamientos de una superclase no sería tan importante para una subclase. La verdadera potencia de la herencia es la capacidad de heredar propiedades y añadir *nuevas*; las subclases pueden agregar variables y métodos a los heredados de la superclase. Recuerde, el coche eléctrico añadió la batería y el enchufe para recargarla. Además, las subclases tienen la capacidad de substituir los métodos heredados y proporcionarles implementaciones diferentes. Por ejemplo, el coche de gasolina probablemente podrá correr mucho más rápido que el eléctrico. El método de aceleración del coche de gasolina podría reflejar esta diferencia.

La herencia de clases está diseñada para permitir la máxima flexibilidad posible. Puede crear árboles de herencia tan profundos como sea necesario para llevar a cabo su diseño. Un árbol de herencia, o jerarquía de clases, se parece mucho a un árbol genealógico; muestra las relaciones entre clases. A diferencia de un árbol genealógico, las clases de un árbol de herencia se vuelven más específicas cuando se desciende por el árbol. Las clases coche de la figura 14.3 son un buen ejemplo de árbol de herencia.

Mediante la herencia ha aprendido cómo las subclases pueden permitir datos y métodos especializados, además de los comunes proporcionados por la superclase. Esto da a los programadores la posibilidad de reutilizar muchas veces el código de la superclase, ahorrando de esta forma un esfuerzo de codificación adicional y, por tanto, eliminando errores potenciales.

Una última observación a realizar respecto a la herencia: es posible y a veces útil crear superclases que actúen sólo como plantillas para subclases con más usos. En este caso, la superclase sólo sirve como abstracción de la funcionalidad de la clase común compartida por las subclases. Por este motivo, estos tipos de superclases se conocen como clases abstractas. Una clase abstracta no puede generar casos, lo que significa que no pueden crearse objetos a partir de ella, ya que hay partes de ella que han sido específicamente desimplementadas. De forma más concreta, estas partes están compuestas de métodos que aún han de implementarse, métodos abstractos.

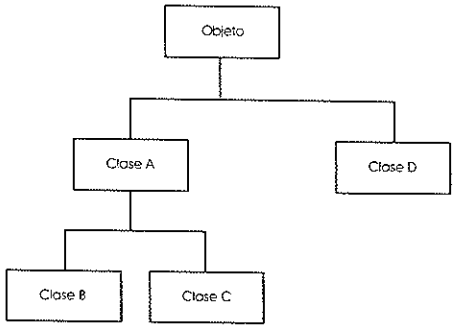
Usando otra vez el ejemplo del coche, el método de acelerar no puede definirse realmente hasta que se conozca la capacidad de aceleración del coche. Por supuesto, cómo acelera un coche lo determina el tipo de motor que tenga. Debido a que el tipo de motor es desconocido en la superclase coche, el método acelerar se ha podido definir pero se ha dejado sin implementar, lo que hace que tanto el método acelerar como la superclase coche sean abstractos. Las clases subordinadas coche de gasoli-

na y eléctrico implementarían el método acelerar para reflejar la capacidad de aceleración de sus motores respectivos.

Clases

Sin duda alguna probablemente ya ha completado actualmente la fase de introducción y está preparado para continuar con la forma que funcionan las clases en Java. Bien, ¡no espere más! En Java, todas las clases son subordinadas de una superclase llamada *Object*. La figura 14.4 muestra a qué se parece la jerarquía de clases Java respecto a la superclase *Object*.

Fig. 14.4
Clases derivadas de la superclase Object.



Como puede ver, todas las clases se abren en abanico a partir de la clase base *Object*. En Java, *Object* sirve de superclase de todas las clases derivadas, incluyendo las clases que componen el API Java.

Declaración de clases

La sintaxis de la declaración de clases en Java es la siguiente:

```
class Identificador {
  CuerpoClase
}
```

El *Identificador* indica el nombre de una nueva clase, que se deriva de forma predeterminada de *Object*. Las llaves rodean el cuerpo de la clase, *CuerpoClase*. Por ejemplo, observe a la declaración de una clase *Alien*, que podría usarse en un juego del espacio:

```
class Alien {
  Color color;
  int  energy;
  int  aggression;
}
```

El estado del objeto *Alien* está definido por tres miembros de datos, que representan el color, la energía y la agresividad del alien. Es importante observar que la clase *Alien* deriva inherentemente de *Object*. Hasta este momento, la clase *Alien* no es muy útil; necesita algunos métodos. La sintaxis básica para declarar métodos de una clase es la siguiente:


```
TipoRetorno Identificador(Parámetros) {
  CuerpoMétodo
}
```

TipoRetorno indica el tipo de datos que devuelve el método, *Identificador* indica el nombre del método y *Parámetros* indica los parámetros del método, si hay. Al igual que los cuerpos de las clases, el cuerpo de un método, *CuerpoMétodo*, está encerrado entre llaves. Recuerde que en términos de diseño orientado a objetos un método es sinónimo de mensaje y el tipo de retorno es la respuesta del objeto al mensaje. A continuación se muestra una declaración de método del método *morph*, que sería útil en la clase *Alien* porque a algunos aliens les gusta cambiar de forma:

```
void morph(int aggression) {
  if (aggression < 10) {
    // morph into a smaller size
  }
  else if (aggression < 20) {
    // morph into a medium size
  }
  else {
    // morph into a giant size
  }
}
```

Se pasa un entero al método *morph* como único parámetro, *aggression*. Este valor se usa para determinar la dosis de morfina a aplicar al alien. Como puede ver, esta dosis varía según su agresividad.

Si hace que el método *morph* sea un miembro de la clase *Alien*, es muy evidente que el parámetro *aggression* no es necesario. Esto es debido a que *aggression* ya es una variable miembro de *Alien*, a la cual tienen acceso todos los métodos de clase. La clase *Alien* con el método morph añadido se ve de esta forma:

```
class Alien {
  Color color;
  int  energy;
  int  aggression;
  void morph() {
    if (aggression < 10) {
      // morph into a smaller size
    }
    else if (aggression < 20) {
      // morph into a medium size
    }
    else {
      // morph into a giant size
    }
  }
}
```

Derivación de clases

Hasta este momento, la explicación sobre la declaración de una clase se ha limitado a la creación de nuevas clases derivadas inherentemente de *Object*. Derivar todas sus clases de *Object* no es muy buena idea, debido a que tendrá que redefinir los

datos y métodos de cada clase. La forma de derivar clases a partir de clases distintas a *Object* es usando la palabra clave *extends*. Vea la sintaxis para derivar una clase usando extends:

```
class Identificador extends ClaseSup {
  CuerpoClase
}
```

Identificador se refiere al nombre de la clase recientemente derivada, *ClaseSup*, al nombre de la clase de la que está derivando y *CuerpoClase* es el cuerpo de la nueva clase.

Usando la clase *Alien* como base de un ejemplo de derivación, ¿qué pasaría si tuviese una clase *Enemy* que definiese la información sobre todos los enemigos? Sin duda querrá derivar la clase *Alien* de la *Enemy*. A continuación se muestra la clase *Alien* derivada de *Enemy* usando la palabra clave *extends*:

```
class Alien extends Enemy {
  Color color;
  int  energy;
  int  aggression;
  void morph() {
    if (aggression < 10) {
      // morph into a smaller size
    }
    else if (aggression < 20) {
      // morph into a medium size
    }
    else {
      // morph into a giant size
    }
  }
}
```

Esta declaración asume que la declaración de la clase *Enemy* está disponible fácilmente en el mismo paquete que *Alien*. En realidad, probablemente derivará desde clases de muchos lugares distintos. Para derivar una clase a partir de una superclase externa, primero debe importarla usando la sentencia *import*.

Nota

Llegará a los paquetes un poco más adelante en este capítulo. Por ahora imagínese sólo que un paquete es un grupo de clases relacionadas.

Si tuviese que importar la clase *Enemy*, lo haría de la forma siguiente:

```
import Enemy;
```

Sustitución de métodos

A veces es útil sustituir métodos en clases derivadas. Por ejemplo, si la clase *Enemy* tuviese un método *move*, querrá que el movimiento variase en función del tipo de enemigo. Algunos tipos de enemigos pueden volar alrededor con formas determinadas, mientras que otros enemigos pueden arrastrarse de forma aleatoria. Para permitir que la clase *Alien* exhiba su propio movimiento, debería substituir el método *move* con una versión específica del movimiento de un alien. La clase *Enemy* se parecería a algo así:


```
class Enemy {
...
void move() {
   // move the enemy
  }
}
```

De forma parecida, la clase *Alien* con la sustitución del método *move* se parecería a algo así:

```
class Alien {
  Color color;
  int  energy;
  int  aggression;
  void move() {
    // move the alien
  }
  void morph() {
    if (aggression < 10) {
      // morph into a smaller size
    }
    else if (aggression < 20) {
      // morph into a medium size
    }
    else {
      // morph into a giant size
    }
  }
}
```

Cuando elabora un caso de la clase *Alien* y llama al método *move*, se ejecuta el nuevo método *move* de *Alien* en lugar del método *move* sustituido original de *Enemy*. La sustitución de un método es un uso simple pero potente del diseño orientado a objetos.

Sobrecarga de métodos

Otra potente técnica orientada a objetos es la sobrecarga de métodos. Esto le permite especificar diferentes tipos de información (parámetros) a enviar a un método. Para sobrecargar un método, declara otra versión con el mismo nombre pero con diferentes parámetros.

Por ejemplo, el método *move* de la clase *Alien* podría tener dos versiones diferentes: un movimiento general y uno para moverse a un sitio concreto. La versión general es la que ya ha definido, que mueve el alien en función de su estado actual. A continuación se muestra la declaración de esta versión:

```
void move() {
  // move the alien
}
```

Para posibilitar que el alien se mueva a un sitio determinado, sobrecarga el método *move* con una versión que toma los parámetros *x* e *y*, que indican el sitio a moverse. La versión sobrecargada de *move* es la siguiente:

```
void move(int x, int y) {
  // move the alien to position x,y
}
```

Observe que la única diferencia entre los dos métodos es la lista de parámetros; el primer *move* no tiene parámetros mientras que el segundo tiene dos enteros.

Puede estar preguntándose cómo sabe el compilador a qué método se llama en un programa, cuando ambos tienen el mismo nombre. El compilador mantiene los parámetros de cada método junto con el nombre. Cuando se encuentra una llamada a un método en un programa, el compilador comprueba el nombre y los parámetros para determinar a qué método sobrecargado llamará. En este caso, las llamadas a los métodos *move* son fácilmente distinguibles por la ausencia o presencia de parámetros enteros.

Modificadores de acceso

El acceso a las variables y métodos de las clases Java se lleva a cabo por medio de modificadores de acceso. Éstos definen varios niveles de acceso entre los miembros de una clase y el mundo exterior (los otros objetos). Los modificadores de acceso se declaran inmediatamente antes del tipo de una variable miembro o del tipo de retorno de un método. Hay cuatro modificadores de acceso: *predeterminado*, *public*, *protected* y *private*.

Los modificadores de acceso no sólo afectan la visibilidad de los miembros de una clase, también la de las clases mismas. Sin embargo, la visibilidad de una clase está estrechamente ligada con los paquetes, que se tratarán más adelante en este capítulo.

Predeterminado

El modificador de acceso predeterminado indica que sólo las clases del mismo paquete pueden tener acceso a las variables y métodos de una clase. Así pues, los miembros de una clase con acceso predeterminado tienen una visibilidad limitada a las otras clases dentro del mismo paquete. No hay una verdadera palabra clave para declarar el modificador de acceso predeterminado; se aplica de manera predeterminada en ausencia de un modificador de acceso. Por ejemplo, los miembros de la clase *Alien* tenían todos acceso predeterminado, porque no se indicaron modificadores de acceso. A continuación se muestran ejemplos de una variable y un método miembros con acceso predeterminado:

```
long length;
void getLength() {
  return length;
}
```

Observe que ni la variable ni el método miembros dan un modificador de acceso, por tanto, toman implícitamente el modificador de acceso predeterminado.

public

El modificador de acceso *public* indica que las variables y métodos de una clase son accesibles, tanto dentro como fuera de la clase. Esto significa que los miembros de la clase public tienen una visibilidad global y pueden ser accesibles para cualquier otro objeto. Vea algunos ejemplos de variables miembro *public*:

```
public int count;
public boolean isActive;
```


protected

El modificador de acceso *protected* indica que los miembros de una clase sólo son accesibles a métodos de la clase y sus subclases. Esto significa que los miembros de una clase *protected* tienen una visibilidad limitada a las subclases. Éstos son algunos ejemplos de una variable y de un método *protected*:

```
protected char middleInitial;
protected char getMiddleInitial() {
  return middleInitial;
}
```

private

Por último, el modificador de acceso *private*, que es el más restrictivo, indica que los miembros de una clase sólo son accesibles a la clase en la que están definidos, es decir que ninguna otra clase tiene acceso a los miembros de una clase *private*, incluso las subclases. Vea algunos ejemplos de variables miembro *private*:

```
private String firstName;
private double howBigIsIt;
```

El modificador static

Hay veces que necesita una variable o método comunes para todos los objetos de una clase determinada. El modificador *static* indica que una variable o un método es el mismo para todos los objetos de una clase determinada.

Normalmente se asignan nuevas variables para cada caso de una clase. Cuando una variable se declara como *static*, sólo se asigna una vez, independientemente de cuantos casos se realicen del objeto. El resultado es que todos los casos de objetos comparten el mismo caso de la variable *static*. De forma similar, un método *static* es el que su implementación es exactamente la misma para todos los objetos de una clase determinada. Esto significa que los métodos *static* sólo tienen acceso a variables static.

A continuación se muestran ejemplos de una variable miembro static y de un método *static*:

```
static int refCount;
static int getRefCount() {
  return refCount;
}
```

Un efecto secundario ventajoso de los miembros *static* es que puede accederse a ellos sin tener que crear un caso de una clase. ¿Recuerda el método *System.out.println* usado en el último capítulo? ¿Recuerda que alguna vez haya creado casos de un objeto *System*? Por supuesto que no. out es una variable miembro *static* de la clase *System*, lo que significa que puede acceder a ella sin tener que crear casos realmente de un objeto *System*.

El modificador final

Otro modificador útil para el control del uso de miembros de una clase es el *final*. Éste indica que una variable tiene un valor constante o que un método de una subclase no puede sustituirse. Para imaginarse de forma literal el modificador *final*, significa que un miembro de una clase es la última versión permitida de la clase.

A continuación se muestran algunos ejemplos de variables miembros *final*:

```
final public int numDollars = 25;
final boolean amIBroke = false;
```

Si viene del mundo de C++, las variables *final* pueden sonarle algo familiares. En efecto, las variables *final* de Java son muy parecidas a las variables const de C++: siempre han de inicializarse en el momento de la declaración y su valor no puede cambiarse en ningún momento posterior.

El modificador synchronized

El modificador *synchronized* se usa para indicar que un método está a prueba de subprocesos. Esto significa básicamente que sólo se permite cada vez una única ruta de ejecución a un método *synchronized*. En un entorno de multisubprocesos como Java, es posible tener muchas rutas de ejecución diferentes que se ejecutan a través del mismo código. El modificador *synchronized* cambia esta regla sólo dejando que cada vez acceda a un método un único subproceso, obligando a que los otros esperen su turno. Si los conceptos de multiproceso y ruta de acceso le son totalmente nuevos, no se preocupe; están analizados detalladamente en el capítulo siguiente, "Subprocesos y multisubprocesos".

El modificador native

El modificador *native* se usa para identificar métodos que tienen implementaciones nativas. Este modificador informa al compilador de Java que una implementación de un método es un archivo C externo. Por este motivo las declaraciones de método *native* parecen diferentes a los otros métodos Java, no tienen cuerpo. Vea un ejemplo de una declaración de método *native*:

```
native int calcTotal();
```

Observe que esta declaración de método acaba simplemente con un punto y coma; no hay llaves que contengan código Java. Esto es debido a que los métodos *native* se implementan en código C, que reside en archivos fuente C externos. Para profundizar en los métodos *native*, consulte el capítulo 38, ///"Native Methods and Libraries".

Clases y métodos abstractos

En la introducción a la orientación a objetos al principio de este capítulo, conoció las clases y métodos abstractos. Para recapitular, una clase abstracta es una clase parcialmente implementada que sólo persigue la comodidad en el diseño. Las clases abstractas se componen de uno o más métodos abstractos, que son métodos declarados pero sin cuerpo (no implementados).

La clase *Enemy* descrita antes es un candidato ideal para convertirse en una clase abstracta. Verdaderamente nunca querrá un objeto enemigo, ya que es demasiado general, sin embargo, sirve para un objetivo muy lógico: ser una superclase para clases de enemigos más específicas, como la *Alien*. Para convertir *Enemy* en una clase abstracta, use la palabra clave *abstract*, de la siguiente forma:

```
abstract class Enemy {
  abstract void move();
  abstract void move(int x, int y);
}
```


Observe el uso de la palabra clave *abstract* antes de la declaración de clase de *Enemy*. Esto informa al compilador que la clase *Enemy* es abstracta. Observe también que los dos métodos *move* están declarados como abstractos; ya que no está claro cómo mover un enemigo genérico, los métodos *move* de *Enemy* se han dejado sin implementar (abstractos).

Debería ser consciente de algunas limitaciones en el uso de *abstract*. Primera, no puede hacer abstractos los métodos de creación. (Aprenderá estos métodos en la siguiente sección que cubre la creación de objetos). Segunda, no puede hacer abstractos los métodos estáticos. Esto proviene del hecho de que los métodos estáticos se declaran para todas las clases, así que no hay forma de suministrar una implementación derivada de un método estático abstracto. Por último, no se le permite hacer abstractos los métodos privados. A primera vista esta limitación puede parecer un poco difícil, pero piense lo que significa. Cuando deriva una clase de una superclase con métodos abstractos, debe sustituir e implementar todos los métodos abstractos o no será capaz de crear casos a partir de su nueva clase y ella misma permanecerá abstracta. Ahora considere que las clases derivadas no pueden ver miembros privados de su superclase, incluidos los métodos, consecuentemente no será capaz de sustituir e implementar métodos abstractos privados de la superclase, es decir, no podrá implementar clases (no abstractas) a partir de ella. Si está limitado a derivar sólo nuevas clases abstractas, ¡no podrá conseguir mucho!

Conversión de tipos

Aunque la conversión entre diferentes tipos de datos ya se discutió en el capítulo 12, "Fundamentos del lenguaje Java", la introducción de las clases aporta algunos nuevos aspectos a la conversión. La conversión entre clases puede subdividirse en tres situaciones diferentes:

- Conversión de una subclase a una superclase
- Conversión de una superclase a una subclase
- Conversión entre hermanos

En el caso de convertir de una subclase a una superclase, puede realizar la conversión implícita o explícitamente. La conversión implícita significa simplemente que no ha de hacer nada, mientras que la explícita quiere decir que ha de proporcionar el tipo de clase, entre paréntesis, al igual que en la conversión entre tipos de datos fundamentales. La conversión de subclase a superclase es totalmente fiable, debido a que las subclases contienen información que las liga a sus superclases. En el caso de convertir de una superclase a una subclase, se le requiere que lo haga explícitamente. Esta conversión no es totalmente fiable, porque el compilador no tiene forma de saber si la clase a la que se convierte es una subclase de la superclase en cuestión. Por último, la conversión entre hermanos no está permitida en Java. Si todas estas conversiones suenan un poco a confusión, mire el siguiente ejemplo:

```
Double d1 = new Double(5.238);
Number n = d1;
Double d2 = (Double)n;
Long l = d1;  // this won't work!
```

En este ejemplo, se crean y asignan uno al otro objetos envolventes del tipo de datos. Si no está familiarizado con las clases envolventes de tipos de datos, no se preocupe, las aprenderá en el capítulo 18, "El paquete del lenguaje". Por ahora, todo lo que tiene que saber es que las clases hermanas *Double* y *Long* derivan de la clase *Number*. En este ejemplo, después que se ha creado el objeto Double d1, se asigna éste a un objeto *Number*. Este es un ejemplo de conversión implícita de una subclase a una superclase, lo que es completamente legal. Después se asigna a otro objeto *Double*, *d2*, el valor de un objeto *Number*. Esta vez, se requiere una conversión explícita porque está convirtiendo de una superclase a una subclase, de lo que no hay garantías de que sea fiable. Por último, se asigna a un objeto *Long* el valor de un objeto *Double*. Esto es una conversión entre hermanos y no está permitida en Java; producirá un error del compilador.

Creación de objetos

Aunque la mayoría del trabajo de diseño en programación orientada a objetos consiste en la creación de clases, realmente no se beneficiará de este trabajo hasta que cree casos (objetos) de estas clases. Para usar una clase en un programa, debe crear primero un caso.

El método creación

Antes de entrar en detalles sobre cómo crear un objeto, hay un método importante que es necesario conocer: el método creación. Cuando cree un objeto, típicamente querrá inicializar sus variables miembro. El método creación es un método especial, implementable en todas sus clases, que le permite inicializar variables y ejecutar cualquier otra operación, cuando se crea un objeto a partir de la clase. Al método creación se le da siempre el mismo nombre que a la clase.

El listado 14.1. contiene el código fuente completo de la clase *Alien*, que contiene dos métodos creación.

Listado 14.1 La clase Alien

```
class Alien extends Enemy {
  protected Color color;
  protected int  energy;
  protected int  aggression;
  public Alien() {
    color = Color.green;
energy = 100;
    aggression = 15;
  }
  public Alien(Color c, int e, int a) {
    color = c;
    energy = e;
    aggression = a;
  }
  public void move() {
    // move the alien
  }
  public void move(int x, int y) {
```

(continúa)


```
    // move the alien to the position x,y
  }
  public void morph() {
    if (aggression < 10) {
      // morph into a smaller size
    }
    else if (aggression < 20) {
      // morph into a medium size
    }
    else {
      // morph into a giant size
    }
  }
}
```

La clase *Alien* usa la sobrecarga de métodos para proporcionar dos métodos creación diferentes. El primero no usa parámetros e inicializa las variables miembro a sus valores predeterminados. El segundo toma el color, la energía y la agresividad del alien e inicializa con ellos las variables miembro. Además de contener los nuevos métodos creación, esta versión de *Alien* usa modificadores de acceso para asignar explícitamente niveles de acceso a cada variable y método miembros. Es un buen hábito a adoptar.

Esta versión de la clase *Alien* se encuentra en el archivo fuente *Enemy1.java* en el CD-ROM, que también incluye la clase *Enemy*. Tenga en cuenta que estas clases son sólo clases de ejemplo con poca funcionalidad. Sin embargo, son buenos ejemplos de diseño de clases Java y pueden compilarse a clases Java.

El operador new

Para crear un caso de una clase, declare una variable objeto y use el operador *new*. Cuando se trata con objetos, una declaración meramente dice a qué tipo de objeto representa una variable. El objeto no se crea realmente hasta que se usa el operador *new*. A continuación se muestran dos ejemplos de uso del operador new para crear casos de la clase *Alien*:

```
Alien anAlien = new Alien();
Alien anotherAlien;
anotherAlien = new Alien(Color.red, 56, 24);
```

En el primer ejemplo, se declara la variable *anAlien* y se crea el objeto usando el operador **new** con una asignación directamente en la declaración. En el segundo ejemplo, se declara primero la variable *anotherAlien* y después se crea y asigna el objeto en una sentencia separada.

Nota

Si tiene alguna experiencia en C++, sin duda reconocerá el operador new. Aunque el operador new de Java funciona casi como su colega de C++, recuerde que *siempre* ha de usar el operador new para crear objetos en Java. Esto contrasta con la versión C++ de new, que sólo se usa cuando trabaja con punteros de objetos. Debido a que Java no admite punteros, se ha de usar siempre el operador new para crear nuevos objetos.

Destrucción de objetos

Cuando un objeto cae fuera del ámbito, se elimina de la memoria o se suprime. De forma parecida al método creación, que se llama cuando se crea un objeto, Java proporciona la capacidad de definir un método destrucción, que se llama cuando se suprime un objeto. A diferencia del método creación, que toma el nombre de la clase, el método destrucción se llama *finalize*. El método *finalize* proporciona un buen lugar para realizar cualquier tipo de limpieza de objetos y se define como

```
void finalize() {
  // cleanup
}
```

Un ejemplo de limpieza realizado típicamente por los objetos de Java es el cierre de archivos. Vale la pena tener en cuenta que no está garantizado que Java llame al método *finalize*, tan pronto como un objeto caiga fuera de ámbito, pues Java suprime objetos como parte de su recogida de basura del sistema, que se produce a intervalos no regulares; como no se suprime realmente un objeto hasta que Java realiza la recogida de basura, tampoco se llama hasta entonces al método *finalize* del objeto.

Paquete

Java proporciona un medio potente de agrupar clases e interfaces en una sola unidad: los paquetes. Las interfaces las aprenderá un poco más adelante en este capítulo. Dicho en pocas palabras, los paquetes son grupos de clases e interfaces relacionados, que proporcionan un mecanismo adecuado para gestionar un gran grupo de clases e interfaces, evitando conflictos de denominación potenciales. El mismo API de Java está implementado como un grupo de paquetes.

Como ejemplo, las clases *Alien* y *Enemy* desarrolladas antes encajarían perfectamente en un paquete *Enemy*, junto con otros objetos de enemigos. Al colocar las clases en un paquete, permite que se beneficien del modificador de acceso predeterminado, que proporciona acceso a las clases del mismo paquete a la información de las otras clases.

Declaración de paquetes

La sintaxis de la sentencia *package* es la siguiente:

```
package Identificador1;
```

Esta sentencia ha de colocarse al principio de la unidad de compilación (archivo fuente), antes de cualquier declaración de clase. Toda clase situada en una unidad de compilación que tenga una sentencia de paquete se considera parte de este paquete. Además, puede esparcir clases entre unidades de compilación separadas; asegúrese sólo de que en cada una incluye una sentencia de paquete.

Los paquetes pueden estar anidados dentro de otros paquetes. En este caso, el interpretador Java espera que la estructura de directorios que contiene las clases ejecutables concuerde con la jerarquía de paquetes.

Importación de paquetes

Cuando llega el momento de usar clases fuera del paquete en el que está trabajando, debe usar la sentencia *import*. Ésta le posibilita importar clases desde otros paquetes, hacia una unidad de compilación. Puede importar clases solas o paquetes enteros de clases de una vez, si lo desea. Vea la sintaxis de la sentencia *import*:

```
import Identificador;
```

Identificador es el nombre de la clase o paquete de clases que está importando. Volviendo a la clase *Alien*, la variable miembro *color* es un caso del objeto *Color*, que forma parte de la biblioteca de clases AWT de Java. Para que el compilador entienda este tipo de variable miembro, debe importar la clase *Color*, que se logra con cualquiera de las sentencias siguientes:

```
import java.awt.Color;
import java.awt.*;
```

La primera sentencia importa la clase específica *Color*, situada en el paquete *Java.awt*. La segunda importa todas las clases del paquete Java.awt. Observe que la siguiente sentencia no funciona:

```
import java.*;
```

Ello ocurre porque no puede importar paquetes anidados con la especificación *. Esto sólo funciona en la importación de todas clases de un paquete determinado, lo que es muy útil.

Hay otra forma de importar objetos desde otro paquete: la referencia explícita al paquete. Al hacer referencia explícita al nombre del paquete cada vez que use un objeto, puede evitar el uso de una sentencia *import*. Al usar esta técnica, la declaración de la variable miembro color de *Alien* sería la siguiente:

```
java.awt.Color color;
```

Generalmente no se requiere hacer referencia explícita al nombre del paquete de una clase externa; usualmente sólo sirve para confundir el nombre de la clase y hacer que el código de más difícil lectura. La excepción a esta regla se produce cuando dos paquetes tienen clases con el mismo nombre. En este caso se le pide que use de forma explícita el nombre del paquete con los nombres de las clases.

Visibilidad de las clases

Anteriormente, en este capítulo, aprendió los modificadores de acceso, que afectan la visibilidad de las clases y de los miembros de las clases. Debido a que la visibilidad de los miembros de las clases se determina en relación a éstas, probablemente se estará preguntando sobre lo que significa la visibilidad para una clase. La visibilidad de una clase se determina en relación a los paquetes.

Por ejemplo, una clase *public* es visible respecto a las clases de otros paquetes. Realmente, *public* es el único modificador de acceso explícito permitido para las clases. Sin el modificador de acceso *public*, las clases, de manera predeterminada, son visibles respecto a las otras clases de un paquete pero no lo son respecto a las clases fuera del paquete.

Interfaces

La última parada de este vertiginoso viaje orientado a objetos por Java es la de las interfaces. Una interfaz es un prototipo para una clase y es útil desde el punto de vista de un diseño lógico. Esta descripción de una interfaz puede parecer algo familiar... ¿Recuerda las clases abstractas?

Anteriormente, en este capítulo, aprendió que una clase abstracta es una clase que se ha dejado parcialmente sin implementar debido a métodos abstractos, que, ellos mismos, están sin implementar. Las interfaces son clases abstractas que se dejan completamente sin implementar. En este caso completamente sin implementar significa que *ningún* método de la clase ha sido implementado. Además, los datos miembros de la interfaz están limitados a las variables finales estáticas, lo que significa que son constantes.

Las ventajas de usar interfaces son en gran parte las mismas que las de usar clases abstractas, pues proporcionan un medio de definir los protocolos de una clase sin preocuparse de los detalles de implementación. Esta ventaja aparentemente sencilla puede hacer más fáciles de gestionar grandes proyectos: una vez diseñadas las interfaces, pueden desarrollarse las clases sin preocuparse por la comunicación entre ellas.

Otro uso importante de las interfaces es la capacidad que tiene una clase de implementar múltiples interfaces. Esto es un giro en el concepto de la herencia múltiple, que C++ admite, pero no Java. La herencia múltiple le permite derivar una clase de múltiples clases superiores. Aunque es potente, la herencia múltiple es una función compleja y, a menudo, truculenta de C++, de la que los diseñadores de Java decidieron prescindir. Su trabajo fue permitir que las clases de Java pudiesen implementar múltiples interfaces.

La principal diferencia entre heredar múltiples interfaces y la auténtica herencia múltiple es que el enfoque de interfaces sólo le permite heredar descripciones de métodos, no implementaciones. Así pues, si una clase implementa múltiples interfaces, está obligada a proporcionar toda la funcionalidad a los métodos definidos en las interfaces. Aunque esto es, ciertamente, más limitante que la herencia múltiple, aún es una función muy útil. Es esta función de las interfaces la que las separa de las clases abstractas.

Declaración de interfaces

La sintaxis para la creación de interfaces es:

```
interface Identificador {
  CuerpoInterfaz
}
```

Identificador es el nombre de la interfaz y *CuerpoInterfaz* se refiere a los métodos abstractos y las variables finales abstractas que componen la interfaz. Debido a que se asume que todos los métodos de una interfaz son abstractos, no es necesario usar la palabra clave *abstract*.

Implementación de interfaces

Debido a que una interfaz es un prototipo, o plantilla, de una clase, debe implementar una interfaz para llegar a una clase utilizable. Para implementar una interfaz, use la palabra clave *implements*. La sintaxis para la implementación de una clase a partir de una interfaz es:

```
class Identificador implements Interfaz {
  CuerpoClase
}
```

Identificador se refiere al nombre de la nueva clase, *Interfaz* es el nombre de la interfaz que está implementando y *CuerpoClase* es el nuevo cuerpo de la clase. El listado 14.2 contiene el código fuente de *Enemy2.java*, que incluye una versión interfaz de *Enemy*, junto con una clase *Alien* que implementa la interfaz.

Listado 14.2 La interfaz Enemy y la clase Alien

```
package Enemy;
import java.awt.Color;
interface Enemy {
  abstract public void move();
  abstract public void move(int x, int y);
}
class Alien implements Enemy {
  protected Color color;
  protected int  energy;
  protected int  aggression;
  public Alien() {
    color = Color.green;
    energy = 100;
    aggression = 15;
  }
  public Alien(Color c, int e, int a) {
    color = c;
    energy = e;
    aggression = a;
  }
  public void move() {
    // move the alien
  }
  public void move(int x, int y) {
    // move the alien to the position x,y
  }
  public void morph() {
    if (aggression < 10) {
      // morph into a smaller size
    }
    else if (aggression < 20) {
      // morph into a medium size
    }
    else {
      // morph into a giant size
    }
  }
}
```

Resumen

Este capítulo ha analizado lo más básico de la programación orientada a objetos, junto con las construcciones Java específicas que le permitirán llevar a cabo conceptos orientados a objetos: clases, paquetes e interfaces. Ha aprendido las ventajas de usar clases, junto con la forma de implementar objetos a partir de ellas. Se ha cubierto el mecanismo de comunicación entre objetos (métodos). También ha aprendido cómo la herencia proporciona un medio potente de volver a utilizar código y crear diseños modulares. Después ha aprendido cómo los paquetes le permiten agrupar lógicamente clases similares, haciendo más fácil el manejo de grandes conjuntos de clases. Por último, ha visto cómo las interfaces proporcionan una plantilla para derivar nuevas clases de una forma estructurada.

Ahora ya está preparado para conocer por funciones más avanzadas del lenguaje Java, como los subprocesos y el multisubproceso. El siguiente capítulo le explica el camino.

Capítulo 15

Subprocesos y multisubproceso

por Charles L. Perkins

Contenido del capítulo

- Subprocesos: lo que son y por qué son necesarios 202
- El problema del paralelismo 204
- Forma de pensamiento en multisubproceso 205
- Creación y uso de subprocesos 211
- Saber cuándo se ha parado un subproceso 215
- Planificación de subprocesos 216

Este capítulo trata de los subprocesos, lo que son y cómo pueden hacer que sus miniaplicaciones funcionen mejor con otras miniaplicaciones y con el sistema Java en general. Explicamos cómo "pensar en multiproceso", cómo proteger a sus métodos y variables de conflictos de subprocesos no intencionados, cómo crear, iniciar y parar subprocesos y clases con subprocesos, y cómo funciona el planificador en Java.

Primero, comencemos por saber por qué son necesarios los subprocesos.

Los subprocesos son una invención relativamente reciente en el mundo de la ciencia de los computadores. Aunque los procesos, su padre mayor, han estado corriendo por ahí desde hace décadas, sólo hace poco que los subprocesos se han aceptado en la corriente principal. Lo raro de esto es que son extremadamente valiosos, y los programas escritos con ellos son notablemente mejores, incluso para el usuario esporádico. En efecto, algunos de los mejores y hercúleos esfuerzos individuales, a lo largo de los años, han implicado la implementación a mano de una utilidad parecida a los subprocesos, para dar a un programa una apariencia más amigable a sus usuarios.

Imagínese que está usando su editor de textos favorito en un gran archivo. Cuando se inicia, ¿ha de examinar todo el archivo antes que le deje editarlo? ¿Necesita hacer una copia del archivo? Si el archivo es inmenso, esto puede ser una pesadilla. ¿No sería más fácil para el programa mostrarle la primera página, permitiéndole que comenzase a editar y que de alguna manera (en segundo plano) completase las tareas más lentas necesarias para la inicialización? Los subprocesos permiten exactamente esta clase de paralelismo dentro del programa.

Quizás, el mejor ejemplo de subproceso (o de falta de él) es el programa de navegación Web. ¿Puede cargar su programa de navegación un número indefinido de archi-