3: Controlando el flujo del programa

Como una criatura consciente, un programa debe manipular su mundo y hacer cambios en el transcurso de su ejecución.

En Java manipulamos objetos y datos mediante el uso de operadores, y elegimos alternativas con las sentencias de control de ejecución. Java es heredado de C++, por lo que muchas de sus sentencias y operadores serán familiares a programadores de C y C++. Java ha añadido además algunas mejoras y simplificaciones.

Utilizando los operadores de Java

Un operador recibe uno o más argumentos y produce un nuevo valor. Los argumentos tienen un formato diferente que las habituales llamadas a métodos, pero el efecto es el mismo. Deberías estar razonablemente cómodo con el concepto general de operador gracias a la experiencia previa programando. Suma (+), resta y signo negativo (-),multiplicación (*), división (/) y asignación (=), todos funciona más o menos igual en cualquier lenguaje de programación.

Todos los operadores producen un valor de sus operandos. Adicionalmente un operador puede cambiar el valor de un operado. Esto se conoce como efecto lateral. El uso más común de los operadores que modifican sus operandos es generar el efecto lateral, pero debemos recordar que el valor generado estará disponible para su uso de la misma forma que en el caso de los operadores sin efecto lateral.

La mayoría de operadores trabajan solamente con tipos primitivos. Las excepciones son ' **=** ', ' **==** ' y ' **!=** ', que trabajan con todos los tipos de objetos (y son una fuente de confusiones en los objetos). Adicionalmente la clase **String** soporta ' **+** ' y ' **+=** '.

Precedencia

La precedencia de operadores define como se evalúa una expresión cuando contiene más de un operador.

Java tiene reglas especificas que determinan el orden de evaluación. La más fácil de recordar es que la multiplicación y la división se evalúan antes que la suma y la resta. A menudo los programadores olvidan el resto de reglas de precedencia, por lo es recomendable usar paréntesis para especificar el orden de evaluación. Por

ejemplo:

```
A = X + Y - 2/2 + Z;
```

tiene un significado muy diferente que la misma sentencia agrupada con paréntesis:

```
A = X + (Y - 2)/(2 + Z);
```

Asignación

La asignación se representa con el operador =. Significa "coger el valor de la parte derecha (a menudo llamado *rvalue*) y copiarlo en la parte izquierda (a menudo llamado *lvalue*). Un rvalue es cualquier constante, variable o expresión que produce un valor, pero el lvalue debe ser una variable. (Esto es, tiene que haber un espacio físico donde almacenar el valor.) Por ejemplo, podemos asignar un valor constante a una variable (**A = 4;**), pero no podemos asignar nada a un valor constante - las constantes no pueden ser lvalue. (No podemos decir **4 = A;.**)

La asignación de tipos primitivos es bastante integra. Como los tipos primitivos contienen el valor actual y no una referencia a un objeto, cuando asignamos tipos primitivos copiamos el contenido de un sitio a otro. Por ejemplo, si ponemos **A = B** cuyos tipos son primitivos, entonces el contenido de **B** se copia en **A** . Si ahora modificamos **A** , obviamente **B** no se verá afectado por esta modificación.

Esto es lo que tenemos que esperar como programadores en la mayoría de situaciones. En cambio cuando asignamos objetos, las cosas cambian. Siempre que manipulamos un objeto lo que estamos manipulando es la referencia al objeto, por lo que cuando asignamos "de un objeto a otro" lo que estamos haciendo es copiar una referencia de un sitio a otro. Esto significa que si decimos **C = D** siendo ambos objetos, acabamos teniendo a **C** y **D** referenciando al objeto que originalmente solo referenciaba **D** . El siguiente ejemplo lo demuestra.

Lo primero que vemos como a parte es una sentencia **package** en **package c03** , que indica el Capítulo 3 de este libro. El primer listado de código de cada capítulo incluirá una sentencia de paquete como esta para establecer el número de capítulo del listado de código en ese capítulo. Como resultado veremos, en el Capítulo 17, que todos los listados del capítulo 3 (excepto aquellos que tengan un nombre de paquete diferente) se situarán automáticamente en un subdirectorio llamado **c03** , los listados del Capítulo 4 estarán en **c04** y así sucesivamente. Todo esto se hará a través del programa **CodePackager.java** visto en el Capítulo 17, y en el

Capítulo 5 se explicará de forma completa el concepto de paquetes. Lo que tenemos que saber en este punto es que, para este libro, las líneas de código del tipo **package c03** se utilizan para establecer el subdirectorio del capítulo para los listados que haya en el mismo.

Para ejecutar el programa debemos asegurarnos que el classpath contiene el directorio raíz donde se instaló el código de este libro. (A partir de este directorio verás los subdirectorios **c02** , **c03** , **c04** , etc.)

Para versiones futuras de Java (1.1.4 y en adelante), cuando el **main( )** está dentro de un fichero con una sentencia **package** , para ejecutar un programa deberemos especificar el nombre completo del paquete antes del nombre del programa. En este caso la línea de comandos es:

```
java c03.Assignment
```

Deberemos recordar esto siempre que ejecutemos una programa que está en un **package** .

Aquí está el ejemplo:

```
//: Assignment.java
// La asignación con objetos es un poco falsa.
package c03;
class Number {
	int i;
}
public class Assignment {
	public static void main(String[] args) {
		Number n1 = new Number();
		Number n2 = new Number();
		n1.i = 9;
		n2.i = 47;
		System.out.println("1: n1.i: " + n1.i +
		", n2.i: " + n2.i);
		n1 = n2;
		System.out.println("2: n1.i: " + n1.i +
		", n2.i: " + n2.i);
		n1.i = 27;
		System.out.println("3: n1.i: " + n1.i +
		", n2.i: " + n2.i);
	}
} ///:~
```

La clase **Number** es sencilla, en el **main( )** creamos dos instancias de esta clase (**n1** y **n2**).Al valor i de cada **Number** se le da un valor diferente, entonces asignamos **n2** a **n1** , y cambiamos **n1** . En muchos lenguajes de programación creeríamos que **n1** y **n2** son independientes todo el tiempo, pero como lo que hemos asignado es una referencia esta es la salida que veremos:

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

¡Cambiando el objeto **n1** parece que el objeto **n2** también cambia! Esto es porque tanto **n1** como **n2** contienen la misma referencia, que está apuntando al mismo objeto. (La referencia que originalmente contenía **n1** y que apuntaba al objeto que contenía el valor 9 ha sido sobrescrita durante la asignación y se ha perdido; este objeto será eliminado por el recolector de basura.)

Frecuentemente se denomina a este fenómeno *alias* , y es esta es fundamentalmente la forma de trabajar de Java con objetos. ¿ Pero y si en este caso no queremos que aparezcan alias?. Podemos adelantarnos a la asignación y decir:

```
n1.i = n2.i;
```

Esto conserva los dos objetos por separado en vez de tirar uno y apuntar **n1** y **n2** al mismo objeto, pero pronto comprenderemos que manipulando los campos de los objetos es desordenado y va en contra de los principios del buen diseño orientado a objetos. Esto no es un tópico trivial, por lo que lo dejaremos para el Capítulo 12, que está dedicado a los alias. En lo sucesivo tendremos en cuenta que la asignación de objetos puede traer sorpresas.

Alias en las llamadas a métodos

Los alias también pueden aparecer cuando pasamos un objeto en un método:

```
//: PassObject.java
// Pasar objetos a métodos puede ser un poco falsa.
class Letter {
	char c;
}
public class PassObject {
	static void f(Letter y) {
```

```
            y.c = 'z';
      }
      public static void main(String[] args) {
            Letter x = new Letter();
            x.c = 'a';
            System.out.println("1: x.c: " + x.c);
            f(x);
            System.out.println("2: x.c: " + x.c);
      }
} ///:~
```

En muchos lenguajes de programación puede parecer que el método **f( )** está haciendo una copia del argumento **Letter** y dentro del alcance del método. Pero una vez más se ha pasado una referencia por lo que la línea

```
y.c = 'z';
```

está actualmente cambiando el objeto fuera de **f( )** . La salida muestra lo siguiente:

```
1: x.c: a
2: x.c: z
```

Los alias y su solución es un tema complejo y, aunque deberemos esperar hasta el Capítulo 12 para todas las respuestas, en este punto debemos tener cuidado con ello para así poder vigilar el peligro.

Operadores matemáticos

Los operadores matemáticos básicos son los mismos que están disponibles en la mayoría de lenguajes de programación: suma (+), resta (-), división (/), multiplicación (*) y modulo (%, produce el resto de una división de enteros). La división de enteros trunca más que redondea el resultado.

Java también utiliza una notación abreviada para permitir una operación y una asignación al mismo tiempo. Esto se indica con un operador seguido del signo de igual, y esto es consistente con todos los operadores del lenguaje (siempre que tenga sentido). Por ejemplo, para añadir 4 a la variable **x** y asignar el resultado a **x** , utilizaremos **x += 4;**

Este ejemplo muestra como utilizar los operadores matemáticos:

```
//: MathOps.java
// Muestra los operadores matemáticos
import java.util.*;
public class MathOps {
	// Crea una abreviatura para evitar el teclear:
	static void prt(String s) {
		System.out.println(s);
	}
	// abreviatura para escribir un string y un entero:
	static void pInt(String s, int i) {
		prt(s + " = " + i);
	}
	// abreviatura para escribir un string y un float:
	static void pFlt(String s, float f) {
		prt(s + " = " + f);
	}
	public static void main(String[] args) {
		// Crea un generador de números aleatorios,
		// por defecto a partir de la hora actual:
		Random rand = new Random();
		int i, j, k;
		// '%' limita el valor máximo a 99:
		j = rand.nextInt() % 100;
		k = rand.nextInt() % 100;
		pInt("j",j); pInt("k",k);
		i = j + k; pInt("j + k", i);
		i = j - k; pInt("j - k", i);
		i = k / j; pInt("k / j", i);
		i = k * j; pInt("k * j", i);
		i = k % j; pInt("k % j", i);
		j %= k; pInt("j %= k", j);
		// Comprueba números de coma flotante:
		float u,v,w; // se aplica también a dobles
		v = rand.nextFloat();
		w = rand.nextFloat();
		pFlt("v", v); pFlt("w", w);
		u = v + w; pFlt("v + w", u);
		u = v - w; pFlt("v - w", u);
		u = v * w; pFlt("v * w", u);
		u = v / w; pFlt("v / w", u);
		// lo siguiente también funciona con
		// char, byte, short, int, long,
		// y double:
		u += v; pFlt("u += v", u);
		u -= v; pFlt("u -= v", u);
		u *= v; pFlt("u *= v", u);
		u /= v; pFlt("u /= v", u);
	}
} ///:~
```

Lo primero que vemos son una serie de métodos abreviados para la escritura: el método **prt( )** escribe un **String** , el **pInt( )** escribe un **String** seguido de un **entero** y el **pFlt( )** escribe un **String** seguido de un **float** . Por supuesto, todos ellos finalmente acaban utilizando **System.out.println( )** .

Para genera números, el programa primero crea un objeto **Random** . Como no se pasan argumentos en la creación, Java utiliza la hora actual como base para el generador de números aleatorios. El programa genera una serie de números aleatorios de diferente tipos con el objeto **Random** simplemente llamando a diferentes métodos: **nextInt( )** , **nextLong( )** , **nextFloat( )** o **nextDouble( )** .

El operador módulo, cuando se utiliza con el resultado de un generador de números aleatorios, limita el resultado a un limite superior del operando menos uno (99 en este caso).

Operadores de signo positivo y negativo

El signo negativo (-) y el signo positivo (+) son los mismos operadores que para la suma y la resta.

El compilador se imagina que uso se le quiere dar por la forma en que escribimos la expresión. Por ejemplo, la sentencia

```
x = -a;
```

tiene un significado obvio. El compilador es capaz de imaginar:

```
x = a * -b;
```

pero el lector puede confundirse, por lo que es más claro decir:

```
x = a * (-b);
```

El signo negativo produce el negativo del valor. El signo positivo proporciona el simétrico del signo negativo, aunque no es que no hace mucho.

Auto incremento y decremento

Java, como C, está lleno de abreviaturas. Las abreviaturas pueden facilitar mucho

la escritura de código y también facilitar o dificultar su lectura.

Dos de las mejores abreviaturas son los operadores de incremento y decremento (a menudo se refiere a ellos como operadores de auto-incremento y auto-decremento). El operador de decremento es -- y significa "decrementar una unidad." El operador de incremento es ++ y significa "incrementar una unidad." Si **A** es un **entero** , por ejemplo, la expresión **++A** es equivalente a **(A = A + 1)** . Los operadores de incremento y decremento producen como resultado el valor de la variable.

Hay dos versiones de cada uno de los operadores, habitualmente llamadas versiones prefijo y sufijo. El pre-incremento significa que el operador **++** aparece antes de la variable o expresión, y el post-incremento significa que el operador **++** aparece después de la variable o expresión. De forma similar, el pre-decremento significa que el operador **--** aparece antes de la variable o expresión, y el post-decremento significa que el operador **--** aparece después de la variable o expresión. En el pre-incremento o pre- decremento, (pe., **++A** o **--A**), se realiza la operación y luego se produce el valor. En el post-incremento o post-decremento (pe., **A++** o **A--**), el valor se produce y luego se realiza la operación. Como ejemplo:

```
//: AutoInc.java
// Muestra los operadores ++ y --
public class AutoInc {
	public static void main(String[] args) {
		int i = 1;
		prt("i : " + i);
		prt("++i : " + ++i); // Pre-incremento
		prt("i++ : " + i++); // Post-incremento
		prt("i : " + i);
		prt("--i : " + --i); // Pre-decremento
		prt("i-- : " + i--); // Post-decremento
		prt("i : " + i);
	}
	static void prt(String s) {
		System.out.println(s);
	}
} ///:~
```

La salida de este programa es:

```
i : 1
++i : 2
i++ : 2
i : 3
```

```
--i : 2
i-- : 2
i : 1
```

Podemos ver que para el prefijo obtenemos el valor después de que se ha realizado la operación, pero con la forma de sufijo obtenemos el valor antes de que se haya realizado la operación. Estos son los únicos operadores que tienen efectos laterales (distintos que los envueltos en la asignación). (Esto es, no solamente utilizan el valor del operando, sino que lo cambian.)

El operador de incremento es una explicación del nombre C++, implicando "un paso más allá de C." En un discurso inicial de Java, Bill Joy (uno de los creadores), dijo que "Java=C++--" (C más más menos menos), sugiriendo que Java es C++ sin las partes difíciles innecesarias quitadas y por tanto un lenguaje mucho más simple. Según progresemos en este libro veremos que varias partes son más simples, y aún así Java no es ese mucho más fácil que C++.

Operadores relacionales

Los operadores relacionales generan un resultado **boolean** . Evalúan la relación entre los valores y los operandos. Una expresión relacional produce **true** si la relación es verdadera y **false** si la relación es falsa. Los operadores relacionales son menor que (<), menor o igual que (<=), mayor que (>), mayor o igual que (>=), igualdad (==) y desigualdad (!=). La igualdad y la desigualdad funcionan con todos los tipos de datos predefinidos, pero las otras comparaciones no funcionarán con el tipo **boolean** .

Comprobando la equivalencia de un objeto

Los operadores relacionales **==** y **!=** también funcionan con todos los objeto, pero a menudo su significado puede confundir a los programadores principiantes de Java. Aquí hay un ejemplo:

```
//: Equivalence.java
public class Equivalence {
	public static void main(String[] args) {
		Integer n1 = new Integer(47);
		Integer n2 = new Integer(47);
		System.out.println(n1 == n2);
		System.out.println(n1 != n2);
	}
} ///:~
```

La expresión **System.out.println(n1 == n2)** mostrará el resultado **lógico** de la comparación. Seguramente la salida será **cierto** y luego **falso** , ya que los dos

objetos **Integer** son el mismo. Pero mientras el contenido es el mismo, las referencias no lo son y los operadores **==** y != comparan referencias a objetos. Luego la salida es actualmente **falso** y luego **cierto** . Naturalmente, a lo primero esto sorprende a la gente.

¿Y si lo que queremos comparar es la igualdad de los contenidos de un objeto? Deberemos usar el método especial **equals( )** que existe para todos los objetos (no para los tipos primitivos que trabajan bien con **==** y **!=**). Aquí está como se utiliza:

```
//: EqualsMethod.java
public class EqualsMethod {
	public static void main(String[] args) {
		Integer n1 = new Integer(47);
		Integer n2 = new Integer(47);
		System.out.println(n1.equals(n2));
	}
} ///:~
```

El resultado será **cierto** , tal y como esperábamos. Ah, pero no es tan simple como esto. Si creamos nuestra propia clase, como esta:

```
//: EqualsMethod2.java
class Value {
	int i;
}
public class EqualsMethod2 {
	public static void main(String[] args) {
		Value v1 = new Value();
		Value v2 = new Value();
		v1.i = v2.i = 100;
		System.out.println(v1.equals(v2));
	}
} ///:~
```

volvemos a la primera situación: el resultado es **falso** . Esto es porque el comportamiento por defecto de **equals()** es comparar las referencias. Por lo que a menos que *sobre escribamos* **equals()** en nuestra nueva clase no tendremos el comportamiento deseado. Desafortunadamente, no aprenderemos sobre como sobre escribir hasta el Capítulo 7, pero en este momento ser advertidos del comportamiento de **equals()** nos puede evitar algunos fracasos.

La mayoría de las clases de las librerías de Java implementan **equals()** de forma

que compare el contenido de los objetos en vez de sus referencias.

Operadores lógicos

Los operadores lógicos AND (**&&**), OR (**||**) y NOT (**!**) producen el valor **lógico true** o **false** basándose en la relación lógica entre sus argumentos. Este ejemplo utiliza operadores relacionales y lógicos:

```
//: Bool.java
// Operadores relacionales y lógicos
import java.util.*;
public class Bool {
	public static void main(String[] args) {
		Random rand = new Random();
		int i = rand.nextInt() % 100;
		int j = rand.nextInt() % 100;
		prt("i = " + i);
		prt("j = " + j);
		prt("i > j is " + (i > j));
		prt("i < j is " + (i < j));
		prt("i >= j is " + (i >= j));
		prt("i <= j is " + (i <= j));
		prt("i == j is " + (i == j));
		prt("i != j is " + (i != j));
		// Tratar un entero como si fuera un valor lógico
		// no está permitido en Java
		//! prt("i && j is " + (i && j));
		//! prt("i || j is " + (i || j));
		//! prt("!i is " + !i);
		prt("(i < 10) && (j < 10) is "
		+ ((i < 10) && (j < 10)) );
		prt("(i < 10) || (j < 10) is "
		+ ((i < 10) || (j < 10)) );
	}
	static void prt(String s) {
		System.out.println(s);
	}
} ///:~
```

Podemos aplicar AND, OR, o NOT solamente a valores **lógicos** . En una expresión no podemos utilizar un valor **no lógico** como si fuera **lógico** , como hacíamos en C y C++. Podemos ver los intentos fallidos quitando los comentarios marcados con el marcador //!. Las siguientes expresiones, en cambio, producen valores **lógicos** utilizando comparaciones relacionales, y luego utilizando operaciones lógicas con los resultados.

Un listado de la salida sería:

```
i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

Fijémonos en que un valor **lógico** se convierte automáticamente al formato texto apropiado si se utiliza donde se espera un **String** .

Podemos cambiar la definición de **entero** en el programa a continuación con cualquier otro tipo de dato primitivo excepto **boolean** . De cualquier manera hay que tener cuidado con los números de coma flotante ya que su comparación es muy estricta. Un número que es la mínima fracción diferente de otro número es aún "no igual." Un número que es el bit más pequeño después del cero sigue sin ser un cero.

Cortocircuitados

Tratar con operadores lógicos con puede llevar a un fenómeno denominado "cortocircuitado." Esto significa que la expresión se evaluará solamente hasta que la falsedad o veracidad de la misma se pueda determinar de forma no ambigua. Como resultado de esto, pueden no evaluarse todas las partes de una expresión lógica. Aquí hay un ejemplo que muestra el cortocircuito:

```
 ShortCircuit.java
// Demuestra el comportamiento del cortocircuito
// con operadores lógicos.
public class ShortCircuit {
	static boolean test1(int val) {
		System.out.println("test1(" + val + ")");
		System.out.println("result: " + (val < 1));
		return val < 1;
	}
	static boolean test2(int val) {
		System.out.println("test2(" + val + ")");
		System.out.println("result: " + (val < 2));
		return val < 2;
	}
```

```
	static boolean test3(int val) {
		System.out.println("test3(" + val + ")");
		System.out.println("result: " + (val < 3));
		return val < 3;
	}
	public static void main(String[] args) {
		if(test1(0) && test2(2) && test3(2))
			System.out.println("expression is true");
		else
			System.out.println("expression is false");
	}
} ///:~
```

Cada comprobación realiza una comparación entre los argumentos y devuelve cierto o falso. También escribe la información para que podamos ver que se le está llamando. Las comprobaciones se realizan en la expresión:

```
if(test1(0) && test2(2) && test3(2))
```

Naturalmente podemos pensar que se ejecutan las tres comprobaciones, pero en cambio la salida nos muestra:

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

La primera comprobación produce un resultado **cierto** , luego la evaluación de la expresión continua. En cambio la segunda comprobación produce un resultado **falso** . Como esto significa que toda la expresión tiene que ser **falso** , ¿por qué continuar evaluando el resto de la expresión? Esto podría ser caro. La razón del cortocircuito, es precisamente esta; podemos tener un incremento potencial del procesamiento si todas las partes de una expresión lógica no necesitan ser evaluadas.

Operadores de bits

Los operadores de bits permiten la manipulación de bits individualmente dentro de un tipo de datos primitivo. Los operadores de bits ejecutan algebra booleana obre los bits correspondientes a sus dos argumentos para producir un resultado.

Los operadores de bits proviene de la orientación de bajo nivel de C, cuando se

manipulaba directamente el *hardware* y se tenían que modificar los bits en los registros *hardware* . Java se diseñó originalmente para programar aparatos de TV por lo que esta orientacó a bajo nivel sigue teniendo sentido. Sin embargo, lo más probable es que no utilice los operadores de bits muy a menudo.

El operador de bits AND (&) produce un uno en el bit de salida si los dos bits de entrada son uno; si no, produce un cero. El operador de bits OR (|) produce un uno en el bit de salida si uno de los bits de entrada es un uno y produce un cero solamente si ambos bits de entrada son cero. El operador OR EXCLUSIVO o XOR (^) produce un uno si uno de los dos bits de entrada es un uno pero no ambos. El operador de bits NOT (~, también llamado operador complemento) es un operador unario, que tiene solamente un argumento. (Todos los demás operadores son operadores binarios). El operador NOT produce el opuesto del bit de entrada - un uno si el bit de entrada es cero, un cero si el bit de entrada es uno.

Los operadores de bits y los operadores lógicos utilizan los mismos caracteres, por lo que resulta útil tener un macanismo mnemónico para ayudar a recordar los significados: como los bits son "pequeños", los operadores de bits solamente tienen un caracter.

Los operadores de bits pueden combinarse con el signo = para unir la operación y la asignación: &=, |= y ^= son expresiones legítimas. (Debido a que ~ es un operador unario, no puede combinarse con el signo =).

El tipo **booleano** se trata como un valor de un bit por lo que es algo diferente. Se pueden realizar operaciones AND, OR y XOR, pero no se puede usar el operador de bits NOT (presumiblemente para evitar la confusión con el NOT lógico). Para los **booleanos** , las operaciones de bits tienen el mismo efecto que los operadores lógicos excepto que en estos no hay "corto-circuito". Además, las operaciones de bits sobre **booleanos** incluye un operador lógico XOR que no está incluido en la lista de operadores "lógicos". Estas avisado del uso de **booleanos** en expresiones de desplazamiento, las cuales se describen a continuación.

Operadores de desplazamiento

Los operadores de desplazamiento tambié manipulan bits. Solamente pueden utilizarse con el tipo primitivo entero. El operador de desplazamiento a la izquierda (<<) da como resultado el desplazamiento a la izquierda del operando de la izquierda del operador el número de bits especificado por el número escrito tras el operador (Se insertan ceros en los bits de órden más bajo). El operador desplazamiento a la derecha con signo (>>) da como resultado el desplazamiento a la derecha del operando a la izquierda del operador desplazamiento el número de bits especificado por el número situado tras el operador. El operador de desplazamiento a la derecha con signo >> utiliza *extensión de signo* : Si el valor es positivo, se insertan ceros en los bits de órden más alto. Si el valor es negativo, se insertan unos en los bits de órden más alto. Java también dispone el desplazamiento a la derecha sin signo >>>, el cual usa *extensión de ceros* : Sea cual sea el signo, se insertan ceros en los bits de los bits de órden más alto. Este

operador no existe en C o en C++.

Si se desplazan **char** , **byte** o **short** se prmueven a **int** antes de que el desplazamiento tenga lugar y el resultado será un **int** . Solamente se utilizarán los cinco bits de órden bajo del lado derecho. Esto impide desplazar más del número de bits que hay en un **int** . Si se opera sobre un **long** , se obtendrá un **long** . Solamente se usarán los seis bits de órden bajo del lado derecho por lo que no se podrán desplazar más que el número de bits en un **long**

Los desplazamientos pueden combinarse con el signo igual (<<= o >>= o >>>=). El "ivalor" (valor de la izquierda) se sustituye por el "ivalor" desplazado por el "rvalor" (valor de la derecha). Existe un problema, sin embargo, con el desplazamiento a la derecha sin signo combinado con la asignación. Si se utiliza con **byte** o **short** no se obtienen resultados correctos. Por el contrario, estos tipos se promocionan a **int** y se desplaza a la derecha, pero se trunca cuando vuelve a asignarse a la variable, por lo que siempre se obtiene **-1** en estos casos. El siguiente ejemplo demuestra esto:

```
//: c03:URShift.java
// Prueba del desplazamiento de bits a la derecha sin signo
public class URShift {
  public static void main(String[] args) {
    int i = -1;
    i >>>= 10;
    System.out.println(i);
    long l = -1;
    l >>>= 10;
    System.out.println(l);
    short s = -1;
    s >>>= 10;
    System.out.println(s);
    byte b = -1;
    b >>>= 10;
    System.out.println(b);
    b = -1;
    System.out.println(b>>>10);
  }
} ///:~
```

En la última línea, el valore resultante no vuelve a asignarse a **b** , sino que se imprime directamente y, por lo tanto, tenemos el comportamiento correcto.

Aquí tenemos un ejemplo que demuestra el uso de todos los operadores que implican bits

```
//: c03:BitManipulation.java
// Uso de los operadores de bits
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);

    int maxneg = -2147483648;
    pBinInt("maxneg", maxneg);
    pBinInt("i", i);
    pBinInt("~i", ~i);
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
```

```
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary: ");
    .System.out.print(" ");
    for(int j = 31; j >= 0; j--)
      if(((1 << j) & i) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary: ");
    System.out.print(" ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
} ///:~
```

Los dos métodos del final, **pBinInt()** y **pBinlong()** toman un **int** o un **long** , respectivamente, y lo imprimen en formato binario junto con una descripción. Puede ignorar la implementación de estas funciones por ahora.

Se habrá dado cuenta del uso de **System.out.print()** en vez de **System.out.println()** . El método **print()** no envía una nueva línea por lo que permite sacar una línea a trozos.

Además de demostrar el efecto de todos los operadores de bits para **int** y **long** , este ejemplo también muestra los valores mínimo, máximo, 1 y -1 para **int** y **long** por lo que se puede ver el aspecto que presentan. Puede notarse lo que representa el bit de órden superior es el signo: 0 significa positivo y 1 significa negativo. La salida de la parte relativa a **int** es:

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: 59081716, binary:
```

```
   00000011100001011000001111110100
~i, int: -59081717, binary:
   11111100011110100111110000001011
-i, int: -59081716, binary:
   11111100011110100111110000001100
j, int: 198850956, binary:
   00001011110110100011100110001100
i & j, int: 58720644, binary:
   00000011100000000000000110000100
i | j, int: 199212028, binary:
   00001011110111111011101111111100
i ^ j, int: 140491384, binary:
   00001000010111111011101001111000
i << 5, int: 1890614912, binary:
   01110000101100000111111010000000
i >> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
   11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
   00000111111000111101001111100000
```

La representación binaria de los números se refiere a la representación en *Complemento a dos* .

Operador ternario if-else

Este operador es inusual porque tiene tres operandos. Es un verdadero operador porque produce un valor, a diferencia de la sentencia if-else que se verá en la siguiente sección de este capítulo. La expresión es de la forma:

```
boolean-exp ? value0 : value1
```

Si evaluar *boolean-exp* resulta **true** , se calcula *value0* y su resultado es el resultado del operador. Si *boolean-exp* es **false** , se calcula *value1* y su resultado es el resultado producido por el operador.

Naturalmente, se puede usar un **if-else** típico (descrito más adelante), pero el operador ternario es mucho más breve. Aunque en C (de donde este operador es originario) presume de ser un lenguaje breve y el operador ternario podría haber sido introducido parcialmente por razones de eficiencia, debería ser cauteloso a la hora en usarlo habitualmente - es fácil escribir codigo ilegible.

El operador condicional puede utilizarse por sus efectos secundarios o por el valor

que genera pero, en general, lo que se desea es un valor puesto que es esto lo que diferencia al operador de la sentencia **if-else** . Aquí tenemos un ejemplo:

```
static int ternary(int i) {
   return i < 10 ? i * 100 : i * 10;
}
```

Puede comprobarse que este código es más compacto que

```
static int alternative(int i) {
if (i < 10)
   return i * 100;
else
   return i * 10;
}
```

La segunda forma es más fácil de entender y no requiere teclear mucho más. Asegurese de considerar bien sus razones cuando elija utilizar el operador ternario

El operador punto y coma (;)

El punto y coma usado en C y C++ no solamente es un separador de listas de argumentos, sino también se usa como operador de evaluación secuencial. El único sitio donde se usa el *operador* en Java es en lo bucles **for** que se describen más tarde en este capítulo

El operador + de String

Tenemos en Java un uso especial de un operador: El operador + puede utilizarse para concatenar cadenas de caracteres, como ya ha visto. Parece ser un uso natural de + aunque no se adecúe a la forma tradicionalen la que se usa +. Esta capacidad pareció una buena idea en C++, por lo que se añadió la *sobrecarga de operadores* en C++ para permitir a los programadores de C++ añadir significado a casi cualquier operador. Desafortunadamente, la sobrecarga de operadores combinada con algunas otras restricciones de C++ hace que esta característica sea bastante complicada para los programadores cuando intentan incorporarla a sus clases. Aunque la sobrecarga de operadores podría haber sido mucho más simple de implementar en Java que en C++, esta característica sigue siendo considerada muy compleja por lo que los programadores de Java no pueden implementar sus propios operadores sobrecargados como pueden hacerlo los programadores de C++.

El uso de **String +** tiene un comportamiento interesante. Si una expresión

comienza con un **String** , entonces todos los operadores que siguen deben ser **String** (Recuerde que el compilador convertirá una secuencia de caracteres entre comillas simples en un **String**):

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

Aquí, el compilador Java convertirá **x** , **y** y **z** en sus representaciones **String** en lugar de sumarlas antes. Si dice:

```
System.out.println(x + sString);
```

Java convertirá **x** en un **String** .

Trampas comunes en el uso de operadores

Una de los problemas que se presentan cuando se usan operadores es intentar evaluar sin paréntesis una expresión en la que se tiene una pequeña duda en como será evaluada. Esto sigue siendo cierto en Java.

```
while(x = y) {
.// ....
}
```

El programador intentaba comprobar la equivalencia (==) en lugar de hacer una asignación. En C y C++, el resultado de esta asignación siempre será **true** si **y** no es cero por lo que, probablemente, entrará en un bucle infinito. En Java, el resultado de esta expresión no es un **booleano** , que es lo que espera el compilador que no lo convertirá en **int** , por lo que, convenientemente, le dará un error en tiempo de compilación antes incluso de ejecutar el programa. Por lo tanto, no se encontrará con esta trampa en Java. (El único caso en el que no se obtendrá un error de compilación es cuando **x** e **y** sean **boolean** , en cuyo caso **x = y** es una expresión legal y, en el caso anterior, probablemente un error).

Un problema similar en C y C++ es usar los operadores de bits AND y OR en lugar de sus verisones lógicas. Los operadores de bits usan un caracter (**&** o **|**)) mientras que los operadores lógicos AND y OR usan dos (**&&** y **||**). Lo mismo que con = y ==, es fácil teclear sólo un caracter en lugar de dos. En Java, el compilador, de nuevo, previene esto porque no le deja usar despreocupadamente

usar un tipo donde no corresponde,

Moldeado de operadores

La palabra *moldeado* (del inglés *cast*)se usa en el sentido de "ajustar a un molde". Java cambiará automáticamenteun tipo de dato a otro cuando estime apropiado. Por ejemplo, si asigna un valor entero a una variable en punto-flotante, el compilador convertirá automáticamente el **int** en **float** . El moldeado le permite realizar esta conversión explícitamente o forzarla cuando no vaya a suceder.

Para realizar un moldeado hay que poner el tipo de dato deseado (incluidos todos los modificadores) entre paréntesis a la izquierda de cualquier valor. Aquí tenemos un ejemplo:

```
void casts() {
   int i = 200;
   long l = (long)i;
   long l2 = (long)200;
}
```

Como puede ver, es posible ralizar un moldeado sobre valores numéricos tanto como sobre variables. En los casos aquí mostrados, sin embargo, el moldeado es completamente superfluo puesto que el compilador promocionará automáticamente un valor **int** a **long** cuando sea necesario. Pero puede hacer este cast superfluo para asegurarse o hacer el código más claro. En otras situaciones, un cast puede ser esencial para compilar el código.

En C y C++, el moldeado puede provocar dolores de cabeza. En Java, el moldeado seguro con la excepción hecha de la *conversión a menor* (esto es, cuando va desde un tipo de dato que puede contener más información a otro que no puede con tanta) en la que se corre el rieso de perder información. Aquí, el compilador le fuerza a realizar un moldeado. En efecto, diciendo "esto puede ser algo peligroso de hacer - si usted quiere que lo haga debe hacer un moldeado explícito". Con una *conversión a mayor* un moldeado explícito no es necesario porque el nuevo tipo podrá contener mayor información que el tipo anterior por lo que no habrá pérdida de información.

Java permite moldear cualquier tipo primitivo en otro tipo primitivo, escepto el **boolean** , al cual no le está permitido ningún moldeado. Los tipos de clase no permiten moldeado. Para convertir de uno a otro debe haber métodos especiales. (**String** es un caso especial y encontrará más tarde en este libro que los objetos pueden moldearse dentro de una *familia* de tipos; un **Roble** puede moldearse a un **Árbol** pero no hay un tipo extraño como una **Roca**)

Literales

Normalmente, cuando inserta un valor literal en un programa, el cokpilador sabe exactamente de que tipo hacerlo. A veces, por el contrario, el tipo es ambiguo. Cuando esto ocurre, debe guiar al compilador añadiendo información extra en forma de caracteres asociados con el valor literal. El siguiente código muestra esos caracteres:

```
//: c03:Literals.java
class Literals {
   char c = 0xffff; // valor hex char máximo
   byte b = 0x7f; // valor hex byte máximo
   short s = 0x7fff; // valor hex short máximo
   int i1 = 0x2f; // Hexadecimal (minúsculas)
   int i2 = 0X2F; // Hexadecimal (mayúsculas)
   int i3 = 0177; // Octal (cero inicial)
   // Hex and Oct también funcionan con long.
   long n1 = 200L; // sufijo long
   long n2 = 200l; // sufijo suffix
   long n3 = 200;
   //! long l6(200); // no permitido
   float f1 = 1;
   float f2 = 1F; // sufijo float
   float f3 = 1f; // sufijo float
   float f4 = 1e-45f; // potencia de 10
   float f5 = 1e+9f; // sufijo float
   double d1 = 1d; // sufijo double
   double d2 = 1D; // sufijo double
   double d3 = 47e47d; // potencia de 10
} ///:~
```

El hexadecimal (base 16), que funciona con todos los tipos de datos enteros, se denota por un **0x** o **0X** inicial seguido por =-9 y a-f ya se en minúsculas o en mayúsculas. Si trata de inicializar una variable con un valor mayor del que puede contener (sin importar la forma numérica del valor), el compilador mostrará un mensaje de error. Notar en el código anterior el valor hexadecimal máximo posible para **char** , **byte** y **short** . Si se exceden estos valores, el compilador automáticamente convertirá este valor en **int** y le indicará que tendrá que realizar un moldeado a menor en la asignación. Usted lo sabrá porque se le indicará la línea.

El octal (base 8) se denota por un cero inicial en el número y los dígitos0-7. No hay representación literal para números binarios en C, C++ o Java

Un caracter final después de un valor literal establece su tipo. Una **L** mayúsucula o minúscula significa **long** , una **F** mayúscula o minúscula signigfica **float** y una **D**

mayúscula o minúscula significa **double** .

Los exponentes usan una notación que siempre he encontrado consternadora: **1.39e-47f** . En ciencia e ingeniería, 'e' se refiere a la base de los logaritmos naturales, aproximadamente 2.718 (Un valor **double** más preciso está disponible en Java como **Math.E**). Este se usa en expresiones de exponencicion tales como 1.39 x e ^-47, que significa 1.39 x 2.718 ^-47. Sin embargo, cuando se inventó FORTRAN se decidió que **e** seignificaba naturalmente "potencia de diez", lo que es una curiosa decisión puesto que FORTRAN se diseño para la ciencia y la ingeniería y uno tendría que pensar que los diseñadores deberían haber sido sensible a tal ambigüedad[1]. En cualquier caso, esta decisión se siguió en C, C++ y en Java. Por tanto, si está acostumbrado a pensar en **e** como en la base de los logaritmos naturales, debe hacer una traducción mental cuando use una expresión como **1.39e-47f** en Java donde significa 1.39 x 10^-47.

Note que no es necesario usar el carácter final cuando el compilador puede saber el tipo apropiado. Con

```
long n3 = 200;
```

no hay ambigüedad por lo que la **L** después del 200 es superflua. Sin embargo, con

```
float f4 = 1e-47f; //potencia de 10
```

el compilador normalmente hace toma los números exponenciales como dobles, por lo que la **f** final dará un error diciendo que debe usar un moldeado para convertir el **double** en **float** .

Promoción

Descubrirá que si realiza cualquier operación matemática o de bits sobre tipos de datos primitivos que son menores que un **int** (Esto es, **char** , **byte** o **short**), esos valores se promocionaran a **int** antes de realizar las operaciones y el valor resultante será de tipo **int** . Por lo tanto, si se desea volver al tipo menor, debe utlizar un moldeado (Y, puesto que está asignando a un tipo menor, puede perder información). En general, el tipo de dato más grande en una expresión es el que determina el tamaño del resultado de la expresión. Si multiplpica un **float** y un **double** , el resultado será un **double** , si añade un **int** y un **long** , el resultado será un **long** .

Java no tiene "sizeof"

En C y C++, el operator **sizeof()** satisface una necesidad específica: le dice en número de bytes ocupados por cada elemento de datos. La necesidad más respetable de **sizeof()** en C y C++ es la portabilidada. Diferentes tipos de datos pueden ser de diferentes tamaños en máquinas diferentes por lo que el programador debe saber como son de grandes esos tipos cuando realiza operaciones que son sensibles al tamaño. Por ejemplo, un ordenador puede almacenar enteros en 32 bits mientras que otro puede almacenar enteros como 16 bits. Los programas pueden utilizar valores enteros más grandes en la primera máquina. Como puede imaginar, la portabilidad es un enorme dolor de cabeza para los programadores C y C++.

Java no necesita el operador **sizeof()** para este propósito porque todos los tipos de datos son del mismo tamaño en todas las máquinas. No necesita pensar en la portabilidad en este nivel - está diseñado dentro del lenguaje.

Revisión de la precedencia

Al oir quejarme sobre la complejidad de recordar la precedencia de operadores durante uno de mis seminarios, un estudiante sugirió una regla mnemónica que es, a la vez, un comentario: "Ulcer Addicts Really Like C A Lot".

Mnemónico	Tipo de operador	Operadores
Ulcer	Unario	+ - ++ --
Addicts	Aritméticos (y desplazamientos)	* / % + - << >>
Really	Relacional	> < >= <= == !=
Like	Lógicas (y de bits)	&& || & | ^
C	Condicionales (ternarios)	A > B ? X : Y
A Lot	Asignación	= (y las compuestas como *=)

Naturalmente, con los operadores de desplazamiento y de bits distribuidos por la tabla, no se tiene un mnemónico perfecto, pero para operaciones que no sean con bits funciona.

Un compendio de operadores

El siguiente ejemplo muestra como pueden usarse los tipos de datos primitivos con operadores particulares. Básicamente, es el mismo ejemplo repetido una y otra vez pero usando tipos primitivos diferentes. El fichero se compilará sin error porque las líneas erroneas están comentadas con un //!

```
//: c03:AllOps.java
// Prueba todos los operadores sobre todos los
```

```
// tipos de datos primitivos para mostrar
// cuales son los aceptados por el compilador Java
class AllOps {
   // Aceptar los resultados de un test booleano
   void f(boolean b) {}
   void boolTest(boolean x, boolean y) {
      // Operadores Aritméticos:
      //! x = x * y;
      //! x = x / y;
      //! x = x % y;
      //! x = x + y;
      //! x = x - y;
      //! x++;
      //! x--;
      //! x = +y;
      //! x = -y;
      // Relacionales y lógicos:
      //! f(x > y);
      //! f(x >= y);
      //! f(x < y);
      //! f(x <= y);
      f(x == y);
      f(x != y);
      f(!y);
      x = x && y;
      x = x || y;
      // Operadores de bits:
      //! x = ~y;
      x = x & y;
      x = x | y;
      x = x ^ y;
      //! x = x << 1;
      //! x = x >> 1;
      //! x = x >>> 1;
      // Asignación compuesta:
      //! x += y;
      //! x -= y;
      //! x *= y;
      //! x /= y;
      //! x %= y;
      //! x <<= 1;
      //! x >>= 1;
      //! x >>>= 1;
      x &= y;
      x ^= y;
      x |= y;
      // Moldeado:
      //! char c = (char)x;
      //! byte B = (byte)x;
      //! short s = (short)x;
```

```
   //! int i = (int)x;
   //! long l = (long)x;
   //! float f = (float)x;
   //! double d = (double)x;
}
void charTest(char x, char y) {
   // Operadores aritméticos:
   x = (char)(x * y);
   x = (char)(x / y);
   x = (char)(x % y);
   x = (char)(x + y);
   x = (char)(x - y);
   x++;
   x--;
   x = (char)+y;
   x = (char)-y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Operadores de bits:
   x= (char)~y;
   x = (char)(x & y);
   x = (char)(x | y);
   x = (char)(x ^ y);
   x = (char)(x << 1);
   x = (char)(x >> 1);
   x = (char)(x >>> 1);
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   x <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   byte B = (byte)x;
   short s = (short)x;
```

```
   int i = (int)x;
   long l = (long)x;
   float f = (float)x;
   double d = (double)x;
}
void byteTest(byte x, byte y) {
   // Operadores Aritméticos:
   x = (byte)(x* y);
   x = (byte)(x / y);
   x = (byte)(x % y);
   x = (byte)(x + y);
   x = (byte)(x - y);
   x++;
   x--;
   x = (byte)+ y;
   x = (byte)- y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Bitwise operators:
   x = (byte)~y;
   x = (byte)(x & y);
   x = (byte)(x | y);
   x = (byte)(x ^ y);
   x = (byte)(x << 1);
   x = (byte)(x >> 1);
   x = (byte)(x >>> 1);
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   x <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   char c = (char)x;
   short s = (short)x;
```

```
   int i = (int)x;
   long l = (long)x;
   float f = (float)x;
   double d = (double)x;
}
void shortTest(short x, short y) {
   // Operadores Aritméticos:
   x = (short)(x * y);
   x = (short)(x / y);
   x = (short)(x % y);
   x = (short)(x + y);
   x = (short)(x - y);
   x++;
   x--;
   x = (short)+y;
   x = (short)-y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Operadores de bits:
   x = (short)~y;
   x = (short)(x & y);
   x = (short)(x | y);
   x = (short)(x ^ y);
   x = (short)(x << 1);
   x = (short)(x >> 1);
   x = (short)(x >>> 1);
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   x <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   char c = (char)x;
   byte B = (byte)x;
```

```
   int i = (int)x;
   long l = (long)x;
   float f = (float)x;
   double d = (double)x;
}
void intTest(int x, int y) {
   // Operadores Aritméticos:
   x = x * y;
   x = x / y;
   x = x % y;
   x = x + y;
   x = x - y;
   x++;
   x--;
   x = +y;
   x = -y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Operadores de bits:
   x = ~y;
   x = x & y;
   x = x | y;
   x = x ^ y;
   x = x << 1;
   x = x >> 1;
   x = x >>> 1;
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   x <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   char c = (char)x;
   byte B = (byte)x;
```

```
   short s = (short)x;
   long l = (long)x;
   float f = (float)x;
   double d = (double)x;
}
void longTest(long x, long y) {
   // Operadores Aritméticos:
   x = x * y;
   x = x / y;
   x = x % y;
   x = x + y;
   x = x - y;
   x++;
   x--;
   x = +y;
   x = -y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Operadores de bits:
   x = ~y;
   x = x & y;
   x = x | y;
   x = x ^ y;
   x = x << 1;
   x = x >> 1;
   x = x >>> 1;
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   x <<= 1;
   x >>= 1;
   x >>>= 1;
   x &= y;
   x ^= y;
   x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   char c = (char)x;
   byte B = (byte)x;
```

```
   short s = (short)x;
   int i = (int)x;
   float f = (float)x;
   double d = (double)x;
}
void floatTest(float x, float y) {
   // Operadores Aritméticos:
   x = x * y;
   x = x / y;
   x = x % y;
   x = x + y;
   x = x - y;
   x++;
   x--;
   x = +y;
   x = -y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Operadores de bits:
   //! x = ~y;
   //! x = x & y;
   //! x = x | y;
   //! x = x ^ y;
   //! x = x << 1;
   //! x = x >> 1;
   //! x = x >>> 1;
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   //! x <<= 1;
   //! x >>= 1;
   //! x >>>= 1;
   //! x &= y;
   //! x ^= y;
   //! x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   char c = (char)x;
   byte B = (byte)x;
```

```
   short s = (short)x;
   int i = (int)x;
   long l = (long)x;
   double d = (double)x;
}
void doubleTest(double x, double y) {
   // Operadores Aritméticos:
   x = x * y;
   x = x / y;
   x = x % y;
   x = x + y;
   x = x - y;
   x++;
   x--;
   x = +y;
   x = -y;
   // Relacionales y lógicos:
   f(x > y);
   f(x >= y);
   f(x < y);
   f(x <= y);
   f(x == y);
   f(x != y);
   //! f(!x);
   //! f(x && y);
   //! f(x || y);
   // Operadores de bits:
   //! x = ~y;
   //! x = x & y;
   //! x = x | y;
   //! x = x ^ y;
   //! x = x << 1;
   //! x = x >> 1;
   //! x = x >>> 1;
   // Asignación compuesta:
   x += y;
   x -= y;
   x *= y;
   x /= y;
   x %= y;
   //! x <<= 1;
   //! x >>= 1;
   //! x >>>= 1;
   //! x & = y;
   //! x ^= y;
   //! x |= y;
   // Moldeado:
   //! boolean b = (boolean)x;
   char c = (char)x;
   byte B = (byte)x;
```

```
      short s = (short)x;
      int i = (int)x;
      long l = (long)x;
      float f = (float)x;
   }
} ///:~
```

Es de notar que **boolean** es bastante limitado. Se le pueden asignar valores **true** y **false** y se puede comprobar su verdad o su falsedad pero no se pueden sumar booleabos o realizar ningún otro tipo de operación con él.

En **char** , **byte** y **short** puede efectuarse la promoción con los operadores aritméticos. Cada operación aritmética sobre cualquiera de esos tipos da un resultado **int** , que debe ser moldeado explícitamente al tipo original. (Una conversión a un tipo menor que podría producri una pérdida de información) para asignárselo a ese tipo. Con los valores **int** , sin embargo, no necesita moldeado porque ya es todo **int** . Por el contrario, no hay que dormirse pensando que todo es seguro. Si multiplica dos **int** s que sean lo suficientemente grandes, se tendrá un desbordamiento como resultado. El siguiente ejemplo muestra esto:

```
//: c03:Overflow.java
// Surprise! Java lets you overflow.
public class Overflow {
      public static void main(String[] args) {
      int big = 0x7fffffff; // valor entero máximo
      prt("big = " + big);
      int bigger = big * 4;
      prt("bigger = " + bigger);
   }
   static void prt(String s) {
      System.out.println(s);
   }
} ///:~
```

El resultado de salida es

```
big = 2147483647
bigger = -4
```

y no se obtienen errores o mensajes de advertencia del compilador ni excepciones en ejecución. Java es bueno. pero no es *así* de bueno.

Las asignaciones compuestas *no* requieren moldeado para **char** , **byte** o **short**

aunque ellos realicen promociones que tienen el mismo resultado que las operaciones aritméticas directas. Por otro lado, la falta de moldeado simplifica el código.

Como puede ver, con la excepción de **boolean** , cualquier tipo primitivo puede amoldarse a cualquier otro tipo primitivo. De nuevo, debe darse cuenta del efecto de la conversión a un tipo menor cuando se moldea a un tipo más pequeño, de otra forma perderá información sin percatarse durante el moldeado.

Control de ejecución

Java utiliza todas las sentencias de control de ejecución de C. De tal modo que si has programado en C o C++ la mayor parte de lo que vas a ver te será familiar. La mayoría de los lenguajes de programación de procedimiento poseen alguna clase de sentencia de control. Muchas veces éstas sentencias se repiten en los distintos lenguajes. En Java el conjunto de palabras clave incluye al **if-else, while, do-while, for** y una sentencia de selección llamada **switch.** Java en cambio no posee el nunca bien ponderado **goto** (el cual, sin embargo, puede ser la forma más rápida de resolver ciertos tipos de problemas). Aún es posible utilizar saltos al estilo goto, pero de una manera mucho más restringida.

true y false

Todas las sentencias condicionales utilizan el valor de verdad o falsedad devuelto por una expresión condicional. Un ejemplo de expresión condicional es **A == B.** Esta expresión usa el operador condicional **==** para determinar si el valor de **A** es equivalente al valor de **B.** La expresión devuelve **true** o **false.** Cualquiera de los operadores relacionales que has visto previamente en este capítulo puede usarse para producir una sentencia condicional. Ten en cuenta que, aunque permitido en C y C++ (donde un valor distinto de cero es falso y un valor igual a cero es verdadero), en Java no es legal utilizar un número como **boolean.** Si quieres usar un valor no booleano en una comprobación booleana, como por ejemplo **if(a),** debes primero convertirla a un valor booleano usando una expresión condicional como **if(a != 0).**

if-else

La sentencia **if-else** es, probablemente, la manera más básica de controlar el flujo de un programa. El **else** es opcional, así que es posible escribir el **if** de dos formas:

```
if(Expresión booleana)
   sentencia
```

o

```
if(Expresión booleana)
   sentencia
else
   sentencia
```

El condicional debe producir un valor booleano. *Sentencia* significa bien una sentencia simple terminada por un punto y coma, o una serie de sentencias encerradas entre llaves. De aquí en más siempre que el término *"sentencia"* sea usado, implicará que la sentencia puede ser simple o compuesta.

Como ejemplo de **if-else,** el método **test()** te mostrará si una estimación está por encima, debajo o es igual que el número buscado:

```
//: c03:IfElse.java
 public class IfElse {
   static int test(int testval, int target) {
      int result = 0;
         if(testval > target)
           result = +1;
         else if(testval < target)
           result = -1;
         else
           result = 0; // Match
         return result;
   }
   public static void main(String[] args) {
      System.out.println(test(10, 5));
      System.out.println(test(5, 10));
      System.out.println(test(5, 5));
   }
} ///:~
```

Por convención, el cuerpo de las sentencias de control de flujo se indenta de manera tal que quien lee el programa pueda determinar fácilmente cuál es el comienzo y el fin de cada una.

return

La palabra reservada **return** tiene dos fines: indicar cuál es el valor que devolverá un método (siempre y cuando el método no sea **void**) y provocar que ese valor sea devuelto en forma inmediata. El método **test()** presentado más arriba puede volver a escribirse para aprovechar esta característica:

```
//: c03:IfElse2.java
```

```
public class IfElse2 {
   static int test(int testval, int target) {
      int result = 0;
      if(testval > target)
         return +1;
      else if(testval < target)
        return -1;
      else
        return 0; // Match
   }
   public static void main(String[] args) {
      System.out.println(test(10, 5));
      System.out.println(test(5, 10));
      System.out.println(test(5, 5));
   }
} ///:~
```

No hay necesidad de un **else** porque el método no continúa luego de ejecutar un **return.**

Iteración

while, do-while y **for** controlan los ciclos y a menudo son clasificadas como *sentencias de iteración.* Una *sentencia* se repite hasta que la *expresión booleana* que la controla se evalúa como falsa. La forma de un ciclo **while** es

```
while(Expresión booleana)
   sentencia
```

La *expresión booleana* se evalúa antes de cada ejecución del ciclo (incluyendo a la primera).

Este es un simple ejemplo que genera números aleatorios hasta que cierta condición se cumpla:

```
//: c03:WhileTest.java
public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

Aquí se usa el método estático **random()** de la biblioteca **Math,** el cual genera un valor de tipo **double** comprendido entre 0 y 1 (incluyendo al 0 pero no al 1). La expresión condicional para el **while** indica lo siguiente: "siga ejecutando este ciclo hasta que el número sea 0.99 o mayor". Cada vez que corras este programa obtendrás una lista de números de diferente longitud.

do-while

La forma para un **do-while** es

```
do
   sentencia
while(Expresión booleana);
```

La única diferencia entre un **while** y un **do-while** es que la sentencia del **do-while** siempre se ejecuta al menos una vez, aún cuando la expresión se evalúe como falsa la primera vez. En un **while,** el hecho de que el condicional sea falso la primera vez que se evalúa provoca que la sentencia nunca llegue a ejecutarse. En la práctica un **do-while** es menos frecuente que un **while.**

for

Un ciclo **for** realiza una inicialización antes de la primera iteración. Luego realiza una comprobación condicional y, al final de cada iterración, cierta forma de "dar un paso" hacia la próxima. La forma para un cliclo **for** es:

```
for(inicializacion;expresión booleana;paso)
   sentencia
```

Cualquiera de las expresiones de *inicialización, expresión booleana* o *paso* puede estar vacía. La expresión booleana se comprueba antes de cada iteración y ni bien ésta evalúe como falso la ejecución continúa en la línea siguiente a la sentencia del **for.** Al final de cada ciclo se ejecuta el *paso.*

los ciclos **for** se utilizan normalmente para tareas en las que es necesario contar:

```
//: c03:ListCharacters.java
public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen
```

```
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

Observa que la variable **c** se define en el punto en que la misma se utiliza, dentro de la expresión de control del ciclo **for,** y no al comienzo del bloque indicado por la llave de apertura. El ámbito de **c** es la expresión controlada por el **for.**

Los lenguajes de procedimiento como C requieren que todas las variables se definan al comienzo de un bloque, de tal manera que cuando el compilador cree un bloque pueda reservar espacio para estas variables. En Java y C++ se puede distribuir las declaraciones de variables en todas las partes del bloque, definiéndolas en el lugar en que se las precisa. Esto facilita un estilo de codificación más natural y hace que el código sea más fácil de entender.

Es posible definir varias variables dentro de una sentencia **for** , pero éstas deben ser del mismo tipo:

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
     i++, j++)
 /* cuerpo del ciclo for */
```

La definición de **int** dentro de la sentencia **for** abarca tanto a **i** como a **j.** La posibilidad de definir variables en la expresión de control estás limitada al ciclo **for.** No es posible usar esta estrategia en ningún otro tipo de sentencia de selección o iteración.

El operador coma

Previamente en este capítulo indiqué que el *operador coma* (no el *separador coma,* que se utiliza para separar definiciones o argumentos de funciones) posee un único uso en Java: en la expresión de control de un ciclo **for.** Es posible tener varias sentencias separadas por comas tanto en la porción de inicialización como en la porción de paso de la expresión de control, y esas sentencias serán evaluadas secuencialmente. El ejemplo de código anterior utiliza esta capacidad. Aquí hay otro ejemplo:

```
//: c03:CommaOperator.java
public class CommaOperator {
  public static void main(String[] args) {
```

```
    for(int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      System.out.println("i= " + i + " j= " + j);
    }
  }
} ///:~
```

La salida sería

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

Puede verse que tanto en la porción de inicialización como en la porción de paso las sentencias se evalúan en orden secuencial. Además, la porción de inicialización puede tener cualquier cantidad de definiciones *de un tipo.*

break y continue

Dentro del cuerpo de cualquiera de las sentencias de iteración también es posible controlar el flujo del loop usando **break** y **continue. break** sale del ciclo sin ejecutar el resto de las sentencias del mismo. **continue** detiene la ejecución de la iteración actual y vuelve al comienzo del ciclo para comenzar la siguiente iteración.

Este programa muestra ejemplos de **break** y **continue** dentro de ciclos **while:**

```
//: c03:BreakAndContinue.java
public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
    }
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
    }
  }
```

```
} ///:~
```

En el ciclo **for** el valor de **i** nunca llega a 100 ya que la sentencia **break** corta la ejecución del ciclo cuando **i** vale 74. Normalmente se usaría un **break** como este sólo si no se conoce el momento en que una condición de fin va a ocurrir. La sentencia **continue** provoca que la ejecución vuelva al principio del ciclo de iteración (incrementando **i**) siempre y cuando **i** no sea divisible entre 9. Cuando lo es, su valor se imprime.

La segunda parte muestra un "ciclo infinito", el cual, en teoría, continuaría para siempre. Sin embargo, dentro del ciclo existe una sentencia **break** que finalizará el ciclo. Además puede verse que el **continue** vuelve al comienzo del ciclo sin completar el resto (así es que la impresión en el segundo ciclo ocurre solamente cuando el valor de **i** es divisible entre 10). La salida es:

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

Se imprime el valor cero porque 0 % 9 produce 0.

Una segunda forma de ciclo infinito es **for(;;).** El compilador trata tanto a **while (true)** como a **for(;;)** de la misma manera, así es que el uso de uno u otro es simplemente una cuestión de gustos.

El infame "goto"

La palabra reservada **goto** ha estado presente desde el principio en los lenguajes de programación. Es más, **goto** fue el génesis del control de programas en lenguaje ensamblador: "Si se da la condición A, entonces saltar hacia aquí, de otra forma saltar hacia allí". Si se lee el código de ensamblador que en última instancia es generado por virtualmente cualquier compilador se verá que el control del programa posee muchos saltos. Sin embargo, un **goto** es un salto al nivel de código fuente, y eso es lo que le valió su desprestigio. Si un programa tiene siempre que saltar de un punto a otro ¿no hay alguna manera de reorganizar el

código de tal forma que el flujo de control no sea tan saltarín? el **goto** tuvo al fin su desaprobación general con la publicación del famoso artículo de Edsger Dijkstra "Goto considerado dañino", y desde ese entonces el "pegarle" al goto ha sido un deporte popular, con los partidarios de la palabra clave maldecida corriendo a buscar refugio.

Como es típico en casos como éste, lo mejor resulta un punto intermedio. El problema no es el uso del **goto,** sino más bien el abuso del mismo. En contadas ocasiones el **goto** es, en realidad, la mejor manera de estructurar el control de flujo.

Aún cuando **goto** es una palabra reservada en Java, la misma no se utiliza en el lenguaje: Java no posee **goto.** Sin embargo sí tiene algo que se parece bastante a un salto y que se usa conjuntamente con las palabras clave **break** y **continue.** No es un salto, sino más bien una manera de salirse de una sentencia de iteración. La razón por la cual aparece normalmente con el tratamiento del **goto** es que usa el mismo mecanismo que éste: una etiqueta.

Una etiqueta es un identificador seguido de dos puntos, como este:

```
etiqueta1:
```

El *único* lugar en donde una etiqueta es útil en Java es inmediatamente antes de una sentencia de iteración.Y esto significa *justo* antes, no es bueno agregar ninguna otra sentencia entre la etiqueta y la iteración. Y la única razón de poner una etiqueta delante de una iteración es que se vaya a anidar otra iteración o un **switch** dentro de ésta. Esto es porque las sentencias de **break** y **continue** normalmente interrumpen sólo el ciclo actual, pero cuando son usadas con una etiqueta las mismas interrumpen todos los ciclos hasta el nivel en que existe la etiqueta:

```
label1:
outer-iteration { i
   nner-iteration {
     // ...
     break; // 1
     //...
     continue; // 2
     //...
     continue label1; // 3
     //...
     break label1; // 4
 }
}
```

En el caso 1, el **break** finaliza la iteración interna, y el programa continúa en la iteración externa. En el caso 2, el **continue** vuelve al comienzo de la iteración interna. En el caso 3, sin embargo, el **continue label1** interrumpe *tanto* la iteración interna *como* la interna externa y vuelve directamente a **label1.** A partir de ese momento continúa con el ciclo, pero partiendo desde la iteración externa. En el caso 4, el **break label1** también interrumpe hasta **label1,** pero no vuelve a entrar en la iteración. En realidad interrumpe por completo ambas iteraciones.

Aquí hay un ejemplo utilizando ciclos **for:**

```
//: c03:LabeledFor.java
public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        prt("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue
```

```
    // to labels here
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Aquí se usa el método **prt()** definido en otros ejemplos

Observa que **break** interrumpe el ciclo **for,** y que la expresión de incremento no ocurre hasta el final del pasaje por el ciclo **for.** Ya que el **break** se saltea la expresión de incremento, el incremento se realiza directamente en el caso de **i == 3.** La sentencia **continue outer** en el caso de **i == 7** también vuelve al comienzo del ciclo y también se saltea el incremento, así es que también se incrementa directamente.

Esta es la salida

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
```

Si no fuera por la sentencia **break outer** no habría manera de salir del loop externo desde dentro del loop interno, ya que **break** por sí mismo sólo puede terminar sólo el ciclo más interno (lo mismo se aplica a **continue**)

Por supuesto que en los casos donde el terminar un ciclo también finalizaría la ejecución del método es posible utilizar un simple **return.**

Esta es una demostración del uso de **break** y **continue** con etiquetas usados en conjunción con ciclos **while:**

```
//: c03:LabeledWhile.java
// From 'Thinking in Java, 2nd ed.' by Bruce Eckel
// www.BruceEckel.com. See copyright notice in CopyRight.txt.
// Java's "labeled while" loop.

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      prt("Outer while loop");
      while(true) {
        i++;
        prt("i = " + i);
        if(i == 1) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("continue outer");
          continue outer;
        }
        if(i == 5) {
          prt("break");
          break;
        }
        if(i == 7) {
          prt("break outer");
          break outer;
        }
      }
    }
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

Las mismas reglas se aplican al **while:**

1. Un **continue** simple va al comienzo del ciclo más interno y continúa.
2. Un **continue** etiquetado vuelve a entrar al ciclo inmediatamente después de la etiqueta.
3. Un **break** continúa al final del ciclo.
4. Un **break** etiquetado continúa al final del ciclo marcado por la etiqueta.

La salida del método deja todo esto en claro:

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
```

Es importante recordar que la *única razón* para usar etiquetas en Java es el tener ciclos anidados para los cuales se desea ejecutar un **break** o **continue** a través de más de un nivel de anidación.

En el artículo de Dijkstra "Goto considerado dañino", el autor objetó específicamente las etiquetas, no el goto. Puntualizó que la cantidad de errores de programación parecía incremenentarse en relación con el número de etiquetas en un programa. Las etiquetas y los gotos hacen que un programa sea difícil de analizar en forma estática, ya que introducen ciclos en el grafo de ejecución del programa. Observa que las etiquetas de Java no sufren de ese problema, ya que están restringidas en su ubicación y no pueden ser utilizadas para transferir el ontrol de una manera arbitraria. También es interesante considerar que este es un caso en el que un lenguaje de programación se hace más útil por medio de restringir el poder de una sentencia.

switch

El **switch** normalmente se clasifica como una *sentencia de selección.* La sentencia **switch** realiza una selección de entre varias secciones de código basada en el valor de una *expresión entera* (en este contexto, *expresión entera* no significa necesariamente de tipo **int,** sino cualquier tipo que represente un número que no sea de punto flotante como por ejemplo **short** y **byte**). Su forma es:

```
switch(selector entero) {
 case valor entero 1 : sentencia; break;
 case valor entero 2 : sentencia; break;
 case valor entero 3 : sentencia; break;
```

```
 case valor entero 4 : sentencia; break;
 case valor entero 5 : sentencia; break;
 // ...
 default: sentencia;
}
```

El *selector entero* es una expresión que produce un valor entero. El **switch** compara el resultado del *selector entero* con cada uno de los *valores enteros.* Si encuentra una coincidencia entonces la correspondiente *sentencia* (simple o compuesta) se ejecuta. De no ocurrir ninguna coincidencia se ejecuta la *sentencia* marcada como **default.**

En la definición anterior puede apreciarse que cada **case** finaliza con un **break,** lo cual causa que la ejecución salte al final del cuerpo del **switch.** Esta es la forma convencional de contruir la sentencia **switch,** no obstante, el **break** es opcional. Si el **break** falta, se ejecuta el código de los siguientes **case,** hasta que se encuentre un **break.** Si bien normalmente este no es el comportamiento que se busca, puede llegar a ser útil para un programador experimentado. Hay que tener en cuenta que la última sentencia, la que sigue al **default** no lleva un **break** ya que la ejecución del programa de todas formas seguirá exactamente en donde hubiese saltado el **break.** Puedes usar un **break** de todas formas sin causar ningún daño, si es que lo consideras importante debido a una cuestión de estilo.

La sentencia **switch** es una manera limpia de implementar una selección de entre varios cursos de ejecución diferentes, pero requiere un selector que se evalúe dando un valor entero como por ejemplo **int** o **char.** Si quieres usar, por ejemplo, una cadena de caracteres o un valor de punto flotante como selector no puedes hacerlo con una sentencia **switch.** Para tipos no enteros se debe usar una serie de sentencias **if.**

Este es un ejemplo en el que se generan letras al azar y se determina si éstas son vocales o consonantes:

```
//: c03:VowelsAndConsonants.java
public class VowelsAndConsonants {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
                System.out.println("vowel");
```

```
                  break;
      case 'y':
      case 'w':
                  System.out.println(
                    "Sometimes a vowel");
                  break;
      default:
                  System.out.println("consonant");
      }
    }
  }
} ///:~
```

Como **Math.random()** genera un valor comprendido entre 0 y 1, sólo se necesita multiplicarlo por el límite superior del rango de números que se desea producir (26 para las letras del alfabeto) y sumar el valor del límite inferior.

Aunque pareciera que aquí se está usando un **switch** con un caracter, en realidad el **switch** está usando el valor entero del carácter. Los caracteres encerrados entre apóstrofos en las sentencias **case** también producen valores enteros que son utilizados para realizar las comparaciones.

Observa cómo los **cases** pueden apilarse uno sobre el otro para brindar múltiples coincidencias para un solo segmento de código. También deberías tener en cuenta que es esencial agregar una sentencia **break** al final de un **case** particular, en caso contrario el flujo del programa simplemente continuará con el proceso del siguiente **case.**

Detalles de cálculo

La sentencia

```
char c = (char)(Math.random() * 26 + 'a');
```

merece una mirada más atenta. **Math.random()** produce un **double,** así que el valor de 26 se convierte a **double** para realizar la multiplicación, la cual también produce un **double.** Esto significa que **'a'** debe convertirse a **double** a fin de realizar la suma. El resultado **double** se vuelve a convertir en **char** utilizando un moldeado (cast).

¿Qué es lo que produce un moldeado a **char** ? O sea, si se tiene un valor de 29.7 y se lo moldea a **char** ¿el resultado es 30 o 29? La respuesta a esta pregunta puede averiguarse utilizando este ejemplo:

```
//: c03:CastingNumbers.java
public class CastingNumbers {
  public static void main(String[] args) {
    double
      above = 0.7,
      below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println(
      "(int)above: " + (int)above);
    System.out.println(
      "(int)below: " + (int)below);
    System.out.println(
      "(char)('a' + above): " +
      (char)('a' + above));
    System.out.println(
      "(char)('a' + below): " +
      (char)('a' + below));
  }
} ///:~
```

La salida es:

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

Así que la respuesta es que el moldeado de

`float`

o

`double`

a un valor entero siempre trunca.

Una segunda pregunta concerniente a **Math.random().** ¿El mismo produce un valor comprendodo entre 0 a 1, incluyendo o excluyendo el valor 1? En lenguaje

matemático, ¿es (0,1), [0,1], (0,1] o [0,1)? (el corchete significa "inclusive", mientras que el peréntesis significa "excluyendo"). Otra vez un programa de prueba puede darnos la respuesta

```
//: c03:RandomBounds.java
public class RandomBounds {
  static void usage() {
    System.err.println("Usage: \n\t" +
      "RandomBounds lower\n\t" +
      "RandomBounds upper");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length != 1) usage();
    if(args[0].equals("lower")) {
      while(Math.random() != 0.0)
        ; // Keep trying
      System.out.println("Produced 0.0!");
    }
    else if(args[0].equals("upper")) {
      while(Math.random() != 1.0)
        ; // Keep trying
      System.out.println("Produced 1.0!");
    }
    else
      usage();
  }
} ///:~
```

Para ejecutar el programa simplemente se ingresa la línea de comandos

```
java RandomBounds lower
```

o

```
java RandomBounds upper
```

En ambos casos te verás forzado a interrumpir el programa en forma manual, así que *parecería* que **Math.random()** nunca produce ni 0.0 ni 1.0. Sin embargo, aquí es cuando este experimento puede resultar engañoso. Si consideras que hay 2^62 valores de doble precisión entre 0 y 1, la posibilidad de conseguir cualquiera

de ellos experimentalmente excedería el tiempo de vida de una computadora, y también el del experimentador. Sucede que 0.0 *está* incluído en la salida de **Math.random().** O sea que en lenguaje matemático, es [0,1).

Chuck Allison escribe: La cantidad total de números en un sistema de números de punto flotante es **2(M-m+1)^(p-1)+1,** donde **b** es la base (normalmente 2), **p** es la precisión (dígitos en la mantisa), **M** es el exponente más grande y **m** es el exponente más pequeño. IEEE 754 utiliza: **M=1023, m=-1022, p=53, b=2,** así que el número total es **2(1023 + 1022 + 1)2^52 = 2((2^10 - 1) + (2^10 - 1))2^52 = (2^10 -1)2^54 = 2^64 - 2^54.** La mitad de estos números (correspondientes a exponentes en el rango [-1022,0]) son menores que 1 (aproximadamente 2^62 valores están en el rango [0,1). Ver mi ensayo en http://www.freshsources.com/1995006a.htm

Resumen

Este capítulo concluye el estudio de las características fundamentales que aparecen en la mayoría de los lenguajes de programación: cálculo, precedencia de operadores, moldeado de tipos, selección e iteración. Ahora estás listo para comenzar a dar pasos que te lleven a estar más cerca del mundo de la programación orientada a objetos. El siguiente capítulo cubre aspectos importantes de la inicialización y purgado de objetos, seguido por el concepto esencial del ocultamiento de la implementación.

Ejercicios

Las soluciones a ejercicios seleccionados puede hallarse en el documento electrónico *The Thinking in Java Annotated Solution Guide,* disponible por un precio accesible en *www.BruceEckel.com*

1. Hay dos expresiones que se encuentran en la sección "precedencia" de este capítulo. Pon esas expresiones en un programa y demuestra que ellas producen diferentes resultados
2. Pon los métodos ternary() y alternative() en un programa.
3. Pon los métodos test() y test2() de las secciones tituladas "if-else" y "return" en un programa.
4. Escribe un programa que imprima los valores que van del 1 al 100.
5. Modifica el ejercicio 4 de tal manera que el programa finalize mediente una sentencia break al llegar al n'umero 47. Luego cámbialo para utilizar return.
6. Escribe una función que reciba dos argumentos de tipo String y que utilice todas las comparaciones booleanas para comparar esas dos cadenas e imprima los resultados. Para las comparaciones == y != también utilizar la comprobación equals(). En el método main () llama a tu función con diferentes objetos de tipo String
7. Escribe un programa que genere 25 valores int aleatorios. Para cada valor, usa una sentencia if-then-else para clasificarlo como mayor, menor o igual que un segundo valor generado aleatoriamente.

8. Modifica el ejercicio 7 de tal manera que tu código quede rodeado por un ciclo while "infinito". El programa se ejecutará entonces hasta que lo interrumpas desde el teclado (normalmente presionando la combinación control-C).
9. Escribe un programa que utilice dos ciclos anidados y el operador módulo (%) para detectar e imprimir números primos (números enteros que no son divisibles por ningún otro número, con excepción de sí mismos y el 1).
10. Crea una sentencia switch que imprima un mensaje por cada case, y pon el switch dentro de un ciclo que recorra cada case. Pon un break luego de cada case, ejecuta el programa, luego quita los breaks y observa qué sucede.