3: Controlar el flujo del programa

Al igual que una criatura con sentimientos, un programa debe manipular su mundo y tomar decisiones durante su ejecución.

En Java, se manipulan objetos y datos haciendo uso de operadores, y se toman decisiones con la ejecución de sentencias de control. Java se derivó de C++, por lo que la mayoría de esas sentencias y operadores resultarán familiares a los programadores de C y C++. Java también ha añadido algunas mejoras y simplificaciones.

Si uno se encuentra un poco confuso durante este capítulo, acuda al CD ROM multimedia adjunto al libro: *Thinking in C: Foundations for Java and C++*. Contiene conferencias sonoras, diapositivas, ejercicios y soluciones diseñadas específicamente para ayudarle a adquirir familiaridad con la sintaxis de C necesaria para aprender Java.

Utilizar operadores de Java

Un operador toma uno o más parámetros y produce un nuevo valor. Los parámetros se presentan de distinta manera que en las llamadas ordinarias a métodos, pero el efecto es el mismo. Uno debería estar razonablemente cómodo con el concepto general de operador con su experiencia de programación previa. La suma (+), la resta y el menos unario (–), la multiplicación (*), la división (/), y la asignación (=) funcionan todos exactamente igual que en el resto de lenguajes de programación.

Todos los operadores producen un valor a partir de sus operandos. Además, un operador puede variar el valor de un operando. A esto se le llama *efecto lateral*. El uso más común de los operadores que modifican sus operandos es generar el efecto lateral, pero uno debería tener en cuenta que el valor producido solamente podrá ser utilizado en operadores sin efectos laterales.

Casi todos los operadores funcionan únicamente con datos primitivos. Las excepciones las constituyen "=", "==" y "!=", que funcionan con todos los objetos (y son una fuente de confusión para los objetos). Además, la clase **String** soporta "+" y "+=".

Precedencia

La precedencia de los operadores define cómo se evalúa una expresión cuando hay varios operadores en la misma. Java tiene reglas específicas que determinan el orden de evaluación. La más fácil de recordar es que la multiplicación y la división siempre se dan tras la suma y la resta. Los programadores suelen olvidar el resto de reglas de precedencia a menudo, por lo que se deberían usar paréntesis para establecer explícitamente el orden de evaluación. Por ejemplo:

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

tiene un significado diferente que la misma sentencia con una agrupación particular de paréntesis:

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

Asignación

La asignación se lleva a cabo con el operador =. Significa "toma el valor de la parte derecha (denominado a menudo *dvalor*) y cópialo a la parte izquierda (a menudo denominada *ivalor*"). Un ivalor es cualquier constante, variable o expresión que pueda producir un valor, pero un ivalor debe ser una variable única con nombre. (Es decir, debe haber un espacio físico en el que almacenar un valor.) Por ejemplo, es posible asignar un valor constante a una variable (**A = 4;**), pero no se puede asignar nada a un valor constante —no puede ser un ivalor. (No se puede decir **4 = A;**.)

La asignación de tipos primitivos de datos es bastante sencilla y directa. Dado que el dato primitivo alberga el valor actual y no una referencia a un objeto, cuando se asignan primitivas se copian los contenidos de un sitio a otro. Por ejemplo, si se dice **A = B** para datos primitivos, los contenidos de **B** se copian a **A**. Si después se intenta modificar **A**, lógicamente **B** no se verá alterado por esta modificación. Como programador, esto es lo que debería esperarse en la mayoría de situaciones.

Sin embargo, cuando se asignan objetos, las cosas cambian. Siempre que se manipula un objeto, lo que se está manipulando es la referencia, por lo que al hacer una asignación "de un objeto a otro" se está, de hecho, copiando una referencia de un sitio a otro. Esto significa que si se escribe **C = D** siendo ambos objetos, se acaba con que tanto **C** como **D** apuntan al objeto al que originalmente sólo apuntaba **D**. El siguiente ejemplo demuestra esta afirmación.

He aquí el ejemplo:

```
//: c03:Asignacion.java
// La asignación con objetos tiene su truco.

class Numero {
  int i;
}

public class Asignacion {
  public static void main(String[] args) {
    Numero n1 = new Numero();
    Numero n2 = new Numero();
    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 **Número** es sencilla, y sus dos instancias (**n1** y **n2**) se crean dentro del método **main()**. Al valor **i** de cada **Número** se le asigna un valor distinto, y posteriormente se asigna **n2** a **n1**, y se varía **n1**. En muchos lenguajes de programación se esperaría que **n1** y **n2** fuesen independientes, pero dado que se ha asignado una referencia, he aquí la salida que se obtendrá:

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

Al cambiar el objeto **n1** parece que se cambia el objeto **n2** también. Esto ocurre porque, tanto **n1**, como **n2** contienen la misma referencia, que apunta al mismo objeto. (La referencia original que estaba en **n1** que apuntaba al objeto que albergaba el valor 9 fue sobreescrita durante la asignación y, en consecuencia, se perdió; su objeto será eliminado por el recolector de basura.)

A este fenómeno se le suele denominar *uso de alias* y es una manera fundamental que tiene Java de trabajar con los objetos. Pero, ¿qué ocurre si uno no desea que se dé dicho uso de alias en este caso? Uno podría ir más allá con la asignación y decir:

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

Esto mantiene los dos objetos separados en vez de desechar uno y vincular **n1** y **n2** al mismo objeto, pero pronto nos damos cuenta que manipular los campos de dentro de los objetos es complicado y atenta contra los buenos principios de diseño orientado a objetos. Este asunto no es trivial, por lo que se deja para el Apéndice A, dedicado al uso de alias. Mientras tanto, se debe recordar que la asignación de objetos puede traer sorpresas.

Uso de alias durante llamadas a métodos

También puede darse uso de alias cuando se pasa un objeto a un método:

```
//: c03:PasarObjeto.java
// Pasar objetos a métodos puede no ser aquello a lo que uno está
// acostumbrado.
class Carta {
  char c;
}

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

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

En muchos lenguajes de programación el método **f( )** parecería estar haciendo una copia de su argumento **Carta** y dentro del ámbito del método. Pero una vez más, se está pasando una referencia, por lo que la línea:

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

está, de hecho, cambiando el objeto fuera de **f( )**. La salida tiene el aspecto siguiente:

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

El uso de alias y su solución son un aspecto complejo, y aunque uno debe esperar al Apéndice A para tener todas las respuestas, hay que ser consciente de este problema desde este momento, de forma que podamos estar atentos y no caer en la trampa.

Operadores matemáticos

Los operadores matemáticos básicos son los mismos que los disponibles en la mayoría de lenguajes de programación: suma (+), resta (-), división (/), multiplicación (*) y módulo (%, que devuelve el resto de una división entera). La división entera trunca, en vez de redondear, el resultado.

Java también utiliza una notación abreviada para realizar una operación y llevar a cabo una asignación simultáneamente. Este conjunto de operaciones se representa mediante un operador seguido del signo igual, y es consistente con todos los operadores del lenguaje (cuando tenga sentido). Por ejemplo, para añadir 4 a la variable **x** y asignar el resultado a **x** puede usarse: **x+=4**.

El siguiente ejemplo muestra el uso de los operadores matemáticos:

```
//: c03:OperadoresMatematicos.java
// Demuestra los operadores matemáticos
import java.util.*;

public class OperadoresMatematicos {
  // Crear un atajo para ahorrar teclear:
  static void visualizar(String s) {
    System.out.println(s);
  }
  // Atajo para visualizar un string y un entero:
  static void pInt (String s, int i) {
    visualizar(s + " = " + i);
  }
  // Atajo para visualizar una cadena de caracteres y un float:
  static void pFlt(String s, float f) {
    visualizar(s + " = " + f);
  }
```

```
  public static void main(String [] args) {
    // Crear un generador de números aleatorios
    // El generador se alimentará por defecto de la hora actual:
    Random aleatorio = new Random();
    int i, j, k;
    // '%' limita el valor a 99:
    j = aleatorio.nextInt() % 100;
    k = aleatorio.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);
    // Pruebas de números de coma flotante:
    float u,v,w;  // Se aplica también a doubles
    v = aleatorio.nextFloat();
    w = aleatorio.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 funciona también para 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 se verán serán los métodos relacionados con la visualización por pantalla: el método **visualizar( )** imprime un **String**, el método **pInt( )** imprime un **String** seguido de un **int,** y el médodo **pFlt( )** imprime un **String** seguido de un **float**. Por supuesto, en última instancia todos usan **System.out.println( )**.

Para generar números, el programa crea en primer lugar un objeto **Random**. Como no se le pasan parámetros en su creación, Java usa la hora actual como semilla para el generador de números aleatorio. El programa genera un conjunto de números aleatorios de distinto tipo con el objeto **Random** simplemente llamando a distintos métodos: **nextInt( ), nextLong( ), nextFloat( )** o **nextDouble( )**.

Cuando el operador módulo se usa con el resultado de un generador de números aleatorios, limita el resultado a un límite superior del operando menos uno (en este caso 99).

Los operadores unarios de suma y resta

El menos unario (-) y el más unario (+) son los mismos operadores que la resta y la suma binarios. El compilador averigua cuál de los dos usos es el pretendido por la manera de escribir la expresión. Por ejemplo, la sentencia:

```
x = -a;
```

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

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

Pero puede que el lector llegue a confundirse, por lo que es más claro decir:

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

El menos unario genera el valor negativo del valor dado. El más unario proporciona simetría con el menos unario, aunque no tiene ningún efecto.

Autoincremento y Autodecremento

Tanto Java, como C, está lleno de atajos. Éstos pueden simplificar considerablemente el tecleado del código, y aumentar o disminuir su legibilidad.

Dos de los atajos mejores son los operadores de incremento y decremento (que a menudo se llaman operadores de autoincremento y autodecremento). El operador de decremento es -- y significa "disminuir en una unidad". El operador de incremento es ++ y significa "incrementar en 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 el valor de la variable como resultado.

Hay dos versiones de cada tipo de operador, llamadas, a menudo, versiones prefija y postfija. El preincremento quiere decir que el operador ++ aparece antes de la variable o expresión, y el postincremento significa que el operador ++ aparece después de la variable o expresión. De manera análoga, el predecremento quiere decir 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. Para el preincremento y el predecremento (por ejemplo, **++a** o—**a**), la operación se lleva a cabo y se produce el valor. En el caso del postincremento y postdecremento (por ejemplo, **a++** o **a--**) se produce el valor y después se lleva a cabo la operación. Por ejemplo:

```
//: c03:AutoInc.java
// Mostrar el funcionamiento de los operadores ++ y --

public class AutoInc {
  public static void main (String[] args) {
    int i = 1;
    visualizar("i : " + i);
    visualizar("++i : " + ++i); // Pre-incremento
    visualizar("i++ : " + i++); // Post-incremento
```

```
    visualizar("i : " + i);
    visualizar("--i : " + --i); // Pre-decremento
    visualizar("i-- : " + i--); // Post-decremento
    visualizar("i : " + i);
  }
static void visualizar(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
```

Se puede pensar que con la forma prefija se consigue el valor después de que se ha hecho la operación, mientras que con la forma postfija se consigue el valor antes de que la operación se lleve a cabo. Éstos son los únicos operadores (además de los que implican asignación) que tienen efectos laterales. (Es decir, cambian el operando en vez de usarlo simplemente como valor.)

El operador de incremento es una explicación para el propio nombre del lenguaje C++, que significa "un paso después de C". En una de las primeras conferencias sobre Java, Bill Joy (uno de sus creadores), dijo que "Java=C++–" (C más más menos menos), tratando de sugerir que Java es C++ sin las partes duras no necesarias, y por consiguiente, un lenguaje bastante más sencillo. A medida que se progrese en este libro, se verá cómo muchas partes son más simples, y sin embargo, Java no es *mucho* más fácil que C++.

Operadores relacionales

Los operadores relacionales generan un resultado de tipo **boolean**. Evalúan la relación entre los valores de 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 (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), igual que (==) y distinto que (!=). La igualdad y la desigualdad funcionan con todos los tipos de datos predefinidos, pero las otras comparaciones no funcionan con el tipo **boolean**.

Probando la equivalencia de objetos

Los operadores relacionales == y != funcionan con todos los objetos, pero su significado suele confundir al que programa en Java por primera vez. He aquí un ejemplo:

```
//: c03:Equivalencia.java

public class Equivalencia {
  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)** visualizará el resultado de la comparación de tipo lógico. Seguramente la salida debería ser **true** y después **false**, pues ambos objetos **Integer** son el mismo. Pero mientras que los *contenidos* de los objetos son los mismos, las referencias no son las mismas, y los operadores == y != comparan referencias a objetos. Por ello, la salida es, de hecho, **false** y después **true**. Naturalmente, esto sorprende a la gente al principio.

¿Qué ocurre si se desea comparar los contenidos de dos objetos? Es necesario utilizar el método especial **equals( )** que existe para todos los objetos (no tipos primitivos, que funcionan perfectamente con == y !=). He aquí cómo usarlo:

```
//: c03:MetodoComparacion.java
public class MetodoComparacion {
  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 **true**, tal y como se espera. Ah, pero no es así de simple. Si uno crea su propia clase, como ésta:

```
//:c03:MetodoComparacion2.java
class Valor {
  int i;
}

public class MetodoComparacion2 {
  public static void main(String[] args) {
    Valor v1 = new Valor();
    Valor v2 = new Valor();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} ///:~
```

se obtiene como resultado **falso**. Esto se debe a que el comportamiento por defecto de **equals( )** es comparar referencias. Por tanto, a menos que se invalide **equals( )** en la nueva clase no se obtendrá el comportamiento deseado. Desgraciadamente no se mostrarán las invalidaciones hasta el Capítulo

7, pero debemos ser conscientes mientras tanto de la forma en que se comporta **equals( )** podría ahorrar algunos problemas.

La mayoría de clases de la biblioteca Java implementan **equals( )**, de forma que compara los contenidos de los objetos en vez de sus referencias.

Operadores lógicos

Los operadores lógicos AND (&&), OR (||) y NOT(!) producen un valor **lógico (true** o **false)** basado en la relación lógica de sus argumentos. Este ejemplo usa los operadores relacionales y lógicos:

```
//: c03:Logico.java
// Operadores relacionales y lógicos
import java.util.*;

public class Logico {
  public static void main(String[] args) {
    Random aleatorio = new Random();
    int i  = aleatorio.nextInt() % 100;
    int j = aleatorio.nextInt() % 100;
    visualizar("i = " + i);
    visualizar("j = " + j);
    visualizar("i > j es " + (i > j));
    visualizar("i < j es " + (i < j));
    visualizar("i >= j es " + (i >= j));
    visualizar("i <= j es " + (i <= j));
    visualizar("i == j es " + (i == j));
    visualizar("i != j es " + (i != j));

    // Tratar un int como un boolean no es legal en Java
  //! visualizar ("i && j es " + (i && j));
  //! visualizar ("i || j es " + (i || j));
  //! visualizar ("!i es " + !i);

    visualizar("(i<10) && (j<10) es " + ((i < 10) && (j < 10));
    visualizar("(i<10) || (j<10) es " + ((i < 10) || (j < 10));
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
} ///:~
```

Sólo es posible aplicar AND, OR o NOT a valores **boolean**. No se puede construir una expresión lógica con valores que no sean de tipo **boolean**, cosa que sí se puede hacer en C y C++. Se pueden

ver intentos fallidos de hacer esto en las líneas que comienzan por //! en el ejemplo anterior. Sin embargo, las sentencias que vienen a continuación producen valores **lógicos** utilizando comparaciones relacionales, y después se usan operaciones lógicas en los resultados.

El listado de salida tendrá la siguiente apariencia:

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

Obsérvese que un valor **lógico** se convierte automáticamente a formato de texto si se utiliza allí donde se espera un **String**.

Se puede reemplazar la definición **int** en el programa anterior por cualquier otro tipo de datos primitivo, excepto **boolean**. Hay que tener en cuenta, sin embargo, que la comparación de números en coma flotante es muy estricta. Un número que sea diferente por muy poco de otro número sigue siendo "no igual". Un número infinitamente próximo a cero es distinto de cero.

Cortocircuitos

Al manipular los operadores lógicos se puede entrar en un fenómeno de "cortocircuito". Esto significa que la expresión se evaluará únicamente *hasta* que se pueda determinar sin ambigüedad la certeza o falsedad de toda la expresión. Como resultado, podría ocurrir que no sea necesario evaluar todas las partes de la expresión lógica. He aquí un ejemplo que muestra el funcionamiento de los cortocircuitos:

```
//: c03:CortoCircuito.java
// Demuestra el comportamiento de los cortocircuitos con operadores
lógicos.

public class CortoCircuito {
  static boolean prueba1(int val) {
    System.out.println("prueba1(" + val + ")");
    System.out.println("resultado: " + (val < 1));
    return val < 1;
  }
  static boolean prueba2(int val) {
    System.out.println("prueba2(" + val + ")");
    System.out.println("resultado: " + (val < 2));
    return val < 2;
  }
```

```
  static boolean prueba3(int val) {
    System.out.println("prueba3(" + val + ")");
    System.out.println("resultado: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    if(prueba1(0) && prueba2(2) && prueba3(2))
      System.out.println("La expresión es verdadera");
    else
      System.out.println("La expresión es falsa");
  }
} ///:~
```

Cada test lleva a cabo una comparación con el argumento pasado y devuelve verdadero o falso. También imprime información para mostrar lo que se está invocando. Las comprobaciones se usan en la expresión:

```
if (prueba1(0) && prueba2(2) && prueba3(2))
```

Naturalmente uno podría pensar que se ejecutarían las tres pruebas, pero en la salida se muestra de otra forma:

```
prueba1(0)
resultado: true
prueba2(2)
resultado: false
la expresión es falsa
```

La primera prueba produjo un resultado **verdadero**, de forma que la evaluación de la expresión continúa. Sin embargo, el segundo test produjo un resultado **falso**. Puesto que esto significa que toda la expresión va a ser **falso** ¿por qué continuar evaluando el resto de la expresión? Podría ser costoso. Ésa es precisamente la razón para realizar un cortocircuito; es posible lograr un incremento potencial de rendimiento si no es necesario evaluar todas las partes de la expresión lógica.

Operadores de bit

Los operadores a nivel de bit permiten manipular bits individuales de la misma forma que si fueran tipos de datos primitivos íntegros. Los operadores de bit llevan a cabo álgebra lógica con los bits correspondientes de los dos argumentos, para producir el resultado.

Los operadores a nivel de bit provienen de la orientación a bajo nivel de C, para la manipulación directa del hardware y el establecimiento de los bits de los registros de hardware. Java se diseñó originalmente para ser empotrado en las cajas *set-top* de los televisores, de forma que esta orientación de bajo nivel tenía sentido. Sin embargo, probablemente no se haga mucho uso de estos operadores de nivel de bit.

El operador de bit AND (**&**) produce un uno a la salida si los dos bits de entrada son unos; si no, produce un cero. El operador de bit OR (|) produce un uno en la salida si cualquiera de los bits de

entrada es un uno, y produce un cero sólo si los dos bits de entrada son cero. El operador de bit OR EXCLUSIVO o XOR (^), produce un uno en la salida si uno de los bits de entrada es un uno, pero no ambos. El operador de bit NOT (~, también llamado operador de *complemento a uno*) es un operador unario; toma sólo un argumento. (Todos los demás operadores de bits son operadores binarios.) El operador de bit NOT produce el contrario del bit de entrada —un uno si el bit de entrada es cero y un cero si el bit de entrada es un uno.

Los operadores de bit y lógicos utilizan los mismos caracteres, por lo que ayuda tener algún mecanismo mnemónico para ayudar a recordar su significado: dado que los bits son "pequeños", sólo hay un carácter en los operadores de bits.

Los operadores de bit se pueden combinar con el signo = para unir la operación a una asignación: **&=**, |= y ^= son válidos (dado que ~ es un operador unario, no puede combinarse con el signo =).

El tipo **boolean** se trata como un valor de un bit, por lo que es en cierta medida distinto. Se puede llevar a cabo un AND, OR o XOR de bit, pero no se puede realizar un NOT de bit (se supone que para evitar la confusión con el NOT lógico). Para los datos de tipo **boolean**, los operadores de bit tienen el mismo efecto que los operadores lógicos, excepto en que no tienen capacidad de hacer cortocircuitos. Además, los operadores de bit sobre datos de tipo **boolean** incluyen un operador XOR lógico no incluido bajo la lista de operadores "lógicos". Hay que tratar de evitar los datos de tipo **boolean** en las expresiones de desplazamiento, descritas a continuación.

Operadores de desplazamiento

Los operadores de desplazamiento también manipulan bits. Sólo se pueden utilizar con tipos primitivos enteros. El operador de desplazamiento a la izquierda (<<) provoca que el operando de la izquierda del operador sea desplazado a la izquierda, tantos bits como se especifique tras el operador (insertando ceros en los bits menos significativos). El operador de desplazamiento a la derecha con signo (>>) provoca que el operando de la izquierda del operador sea desplazado a la derecha el número de bits que se especifique tras el operador. El desplazamiento a la derecha con signo >> utiliza la *extensión de signo*: si el valor es positivo se insertan ceros en los bits más significativos; si el valor es negativo, se insertan unos en los bits más significativos. Java también ha incorporado el operador de rotación a la derecha sin signo >>>, que utiliza la *extensión cero*: independientemente del signo, se insertan ceros en los bits más significativos. Este operador no existe ni en C ni en C++.

Si se trata de desplazar un **char**, un **byte** o un **short**, éste será convertido a **int** antes de que el desplazamiento tenga lugar y el resultado será también un **int**. Sólo se utilizarán los cinco bits menos significativos de la parte derecha. Esto evita que se desplace un número de bits mayor al número de bits de un **int**. Si se está trabajando con un **long**, se logrará un resultado **long**. Sólo se usarán los seis bits menos significativos de la parte derecha, por lo que no es posible desplazar más bits que los que hay en un **long**.

Los desplazamientos pueden combinarse con el signo igual (<<= o >>= o >>>=). El ivalor se reemplaza por el ivalor desplazado por el dvalor. Hay un problema, sin embargo, con el desplazamiento sin signo a la derecha combinado con la asignación. Si se utiliza con un **byte** o **short** no se logra el resultado correcto. En vez de esto, los datos son convertidos a **int** y desplazados a la derecha, y

teriormente se truncan al ser asignados de nuevo a sus variables, por lo que en esos casos el resultado suele ser **-1**. El ejemplo siguiente demuestra esto:

```
//: c03:DesplDatosSinSigno.java
// Prueba del desplazamiento a la derecha sin signo.

public class DesplDatosSinSigno {
  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, no se asigna el valor resultante de nuevo a **b**, sino que se imprime directamente para que se dé el comportamiento correcto.

He aquí un ejemplo que demuestra el uso de todos los operadores que involucran a bits:

```
//: c03:ManipulacionBits.java
// Utilizando los operadores de bit.
import java.util.*;

public class ManipulacionBits {
  public static void main(String[] args) {
    Random aleatorio = new Random();
    int i = aleatorio.nextInt();
    int j = aleatorio.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int posmax = 2147483647;
    pBinInt("posmax", posmax);
    int negmax = -2147483648;
    pBinInt("negmax", negmax);
    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 = aleatorio.nextLong();
    long m = aleatorio.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 + ", binario: ");
    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 + ", binario: ");
    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 cadena de caracteres descriptiva. De momento, ignoraremos la implementación de estos métodos.

Se habrá dado cuenta el lector del uso de **System.out.print( )** en vez de **System.out.println( )**. El método **print( )** no finaliza con un salto de línea, por lo que permite ir visualizando una línea por fragmentos.

Además de demostrar el efecto de todos los operadores de bit para **int** y **long**, este ejemplo también muestra los valores mínimo, el máximo, +1 y –1 para **int** y para **long**, por lo que puede verse qué aspecto tienen. Nótese que el bit más significativo representa el signo: 0 significa positivo, y 1 significa negativo. La salida de la porción **int** tiene la apariencia siguiente:

```
-1, int: -1, binario:
  11111111111111111111111111111111
+1, int: 1, binario:
  00000000000000000000000000000001
posmax, int: 2147483647, binario:
  01111111111111111111111111111111
negmax, int: -2147483648, binario:
  10000000000000000000000000000000
i, int: 59081716, binario:
  00000011100001011000001111110100
~i, int: -59081717, binario:
  11111100011110100111110000001011
-i, int: -59081716, binarios:
  11111100011110100111110000001100
j, int: 198850956, binario:
  00001011110110100011100110001100
i & j, int: 58720644, binario:
  00000011100000000000000110000100
i | j, int: 199212028, binario:
  00001011110111111011101111111100
i ^ j, int: 140491384, binario:
  00001000010111111011101001111000
```

```
i << 5, int: 1890614912, binario:
  01110000101100000111111010000000
i >> 5, int: 1846303, binario:
  00000000000111000010110000011111
(~ i) >>5, int: -1846304, binario:
  11111111111000111101001111100000
i >>> 5, int: 1846303, binario:
  00000000000111000010110000011111
(~ i) >>> 5, int: 132371424, binario:
  00000111111000111101001111100000
```

La representación binaria de los números se denomina también *complemento a dos con signo.*

Operador ternario if-else

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

```
exp-booleana ? valor0 : valor1
```

Si el resultado de la evaluación exp-boolean es **true**, se evalúa *valor0* y su resultado se convierte en el valor producido por el operador. Si exp-booleana es **false**, se evalúa *valor1* y su resultado se convierte en el valor producido por el operador.

Por supuesto, podría usarse una sentencia **if-else** ordinaria (descrita más adelante), pero el operador ternario es mucho más breve. Aunque C (del que es originario este operador) se enorgullece de ser un lenguaje sencillo, y podría haberse introducido el operador ternario en parte por eficiencia, deberíamos ser cautelosos a la hora de usarlo cotidianamente —es fácil producir código ilegible.

El operador condicional puede usarse por sus efectos laterales o por el valor que produce, pero en general se desea el valor, puesto que es éste el que hace al operador distinto del **if-else**. He aquí un ejemplo:

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

Este código, como puede observarse, es más compacto que el necesario para escribirlo sin el operador ternario:

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

La segunda forma es más sencilla de entender, y no requiere de muchas más pulsaciones. Por tanto, hay que asegurarse de evaluar las razones a la hora de elegir el operador ternario.

El operador coma

La coma se usa en C y C++ no sólo como un separador en las listas de parámetros a funciones, sino también como operador para evaluación secuencial. El único lugar en que se usa el *operador* coma en Java es en los bucles **for**, que serán descritos más adelante en este capítulo.

El operador de String +

Hay un uso especial en Java de un operador: el operador + puede utilizarse para concatenar cadenas de caracteres, como ya se ha visto. Parece un uso natural del + incluso aunque no encaje con la manera tradicional de usar el +. Esta capacidad parecía una buena idea en C++, por lo que se añadió la *sobrecarga de operadores* a C++, para permitir al programador de C++ añadir significados a casi todos los operadores. Por desgracia, la sobrecarga de operadores combinada con algunas otras restricciones de C++, parece convertirse en un aspecto bastante complicado para que los programadores la usen al diseñar sus clases. Aunque la sobrecarga de operadores habría sido mucho más fácil de implementar en Java que en C++, se seguía considerando que se trataba de un aspecto demasiado complicado, por lo que los programadores de Java no pueden implementar sus propios operadores sobrecargados como pueden hacer los programadores de C++.

El uso del + de **String** tiene algún comportamiento interesante. Si una expresión comienza con un **String**, entonces todos los operandos que le sigan deben ser de tipo **String** (recuerde que el compilador convertirá una secuencia de caracteres entre comas 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 a **x**, **y** y **z** en sus representaciones **String** en vez de sumarlas. Mientras que si se escribe:

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

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

Pequeños fallos frecuentes al usar operadores

Uno de los errores frecuentes al utilizar operadores es intentar no utilizar paréntesis cuando se tiene la más mínima duda sobre cómo se evaluará una expresión. Esto sigue ocurriendo también en Java.

Un error extremadamente frecuente en C y C++ es éste:

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

El programador estaba intentando probar una equivalencia (==) en vez de hacer una asignación. En C y C++ el resultado de esta asignación siempre será **true** si **y** es distinta de cero, y probablemente se entrará en un bucle infinito. En Java, el resultado de esta expresión no es un **boolean**, y el compilador espera un **boolean** pero no convertirá el **int** en **boolean**, por lo que dará el conveniente error en tiempo de compilación, y capturará el problema antes de que se intente siquiera ejecutar el programa. De esta forma, esta trampa jamás puede ocurrir en Java. (El único momento en que no se obtendrá un error en tiempo 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 utilizar los operadores de bit AND y OR, en vez de sus versiones lógicas. Los AND y OR de bit utilizan uno de los caracteres (**&** o **|**) y los AND y OR lógicos utilizan dos (**&&** y **||**). Como ocurre con el = y el ==, es fácil escribir sólo uno de los caracteres en vez de ambos. En Java, el compilador vuelve a evitar esto porque no los permite utilizar con operadores incorrectos.

Operadores de conversión

La palabra *conversión* se utiliza con el sentido de "convertir[1] a un molde". Java convertirá automáticamente un tipo de datos en otro cuando sea adecuado. Por ejemplo, si se asigna un valor entero a una variable de coma flotante, el compilador convertirá automáticamente el **int** en **float**. La conversión permite llevar a cabo estas conversiones de tipos de forma explícita, o forzarlas cuando no se diesen por defecto.

Para llevar a cabo una conversión, se pone el tipo de datos deseado (incluidos todos los modificadores) entre paréntesis a la izquierda de cualquier valor. He aquí un ejemplo:

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

Como puede verse, es posible llevar a cabo una conversión, tanto con un valor numérico, como con una variable. En las dos conversiones mostradas, la conversión es innecesaria, dado que el compilador convertirá un valor **int** en **long** cuando sea necesario. No obstante, se permite usar conversiones innecesarias para hacer el código más limpio. En otras situaciones, puede ser esencial una conversión para lograr que el código compile.

En C y C++, las conversiones pueden conllevar quebraderos de cabeza. En Java, la conversión de tipos es segura, con la excepción de que al llevar a cabo una de las denominadas *conversiones reductoras* (es decir, cuando se va de un tipo de datos que puede mantener más información a otro que no puede contener tanta) se corre el riesgo de perder información. En estos casos, el compilador fuerza a hacer una conversión explícita, diciendo, de hecho, "esto puede ser algo peligroso de hacer

[1] N. del Traductor: Casting se traduce aquí por convertir.

—si quieres que lo haga de todas formas, tiene que hacer la conversión de forma explícita". Con una *conversión extensora* no es necesaria una conversión explícita porque el nuevo tipo es capaz de albergar la información del viejo tipo sin que se pierda nunca ningún bit.

Java permite convertir cualquier tipo primitivo en cualquier otro tipo, excepto **boolean**, que no permite ninguna conversión. Los tipos clase no permiten ninguna conversión. Para convertir una a otra debe utilizar métodos especiales (**String** es un caso especial y se verá más adelante en este libro que los objetos pueden convertirse en una *familia* de tipos; un **Roble** puede convertirse en **Árbol** y viceversa, pero esto no puede hacerse con un tipo foráneo como **Roca**.)

Literales

Generalmente al insertar un valor literal en un programa, el compilador sabe exactamente de qué tipo hacerlo. Sin embargo, en ocasiones, el tipo es ambiguo. Cuando ocurre esto es necesario guiar al compilador añadiendo alguna información extra en forma de caracteres asociados con el valor literal. El código siguiente muestra estos caracteres:

```
//: c03:Literales.java

class Literales {
  char c = 0xffff; // Carácter máximo valor hexadecimal
  byte b = 0x7f; // Máximo byte valor hexadecimal
  short s = 0x7fff; // Máximo short valor hexadecimal
  int i1 = 0x2f; // Hexadecimal (minúsculas)
  int i2 = 0X2F; // Hexadecimal (mayúsculas)
  int i3 = 0177; // Octal (Cero delantero)
  // Hex y Oct también funcionan con long.
  long n1 = 200L; // sufijo long
  long n2 = 200l; // sufijo long
  long n3 = 200;
  //! long l6(200); // prohibido
  float f1 = 1;
  float f2 = 1F; // sufijo float
  float f3 = 1f; // sufijo float
  float f4 = 1e-45f; // 10 elevado a -45
  float f5 = 1e+9f; // sufijo float
  double d1 = 1d; // sufijo double
  double d2 = 1D; // sufijo double
  double d3 = 47e47d; // 10 elevado a 47
  } ///:~
```

La base 16 (hexadecimal), que funciona con todos los tipos de datos enteros, se representa mediante un **0x** o **0X** delanteros, seguidos de 0–9 y a–f, tanto en mayúsculas como en minúsculas. Si se trata de inicializar una variable con un valor mayor que el que puede albergar (independientemente de la forma numérica del valor), el compilador emitirá un mensaje de error. Fíjese en el código anterior, los valores hexadecimales máximos posibles para **char**, **byte** y **short**. Si se excede de éstos, el compi-

lador generará un valor **int** automáticamente e informará de la necesidad de hacer una conversión reductora para llevar a cabo la asignación. Se sabrá que se ha traspasado la línea.

La base 8 (octal) se indica mediante un cero delantero en el número, y dígitos de 0 a 7. No hay representación literal de números binarios en C, C++ o Java.

El tipo de un valor literal lo establece un carácter arrastrado por éste. Sea en mayúsculas o minúsculas, **L** significa **long**, **F** significa **float**, y **D** significa **double**.

Los exponentes usan una notación que yo a veces encuentro bastante desconcertante: **1,39 e-47f**. En ciencias e ingeniería, la "e" se refiere a la base de los logaritmos naturales, aproximadamente 2,718. (Hay un valor **double** mucho más preciso en Java, denominado **Math.E**.) Éste se usa en expresión exponencial, como 1,39 e^{-47}, que quiere decir 1,39 x $2{,}718^{-47}$. Sin embargo, cuando se inventó *Fortran* se decidió que la **e** querría indicar "diez elevado a la potencia" lo cual es una mala decisión, pues *Fortran* fue diseñado para ciencias e ingeniería y podría pensarse que los diseñadores deben ser conscientes de que se ha introducido semejante ambigüedad[2]. En cualquier caso, esta costumbre siguió en C y C++, y ahora en Java. Por tanto, si uno está habituado a pensar que **e** es la base de los logaritmos naturales, tendrá que hacer una traslación mental al ver una expresión como **1,39 e^{-47f}** en Java; significa **1,39 * 10^{-47}**.

Nótese que no es necesario utilizar el carácter final cuando el compilador puede averiguar el tipo apropiado. Con

```
long n3 = 200;
```

no hay ambigüedad, por lo que una **L** tras el 200 sería superflua. Sin embargo, con

```
float f4 = 1e-47f; // 10 elevado a
```

el compilador, normalmente, tomará los números exponenciales como *double*, de forma que sin la **f** arrastrada dará un error indicando que es necesario hacer una conversión de **double** en un **float**.

Promoción

Al hacer operaciones matemáticas o de bit sobre tipos de datos primitivos, se descubrirá que si son más pequeños que un **int** (es decir, **char**, **byte**, o **short**), estos valores se promocionarán a **int** antes de hacer las operaciones, y el valor resultante será de tipo **int**. Por tanto, si se desea asignar el valor devuelto, de nuevo al tipo de menor tamaño, será necesario utilizar una conversión. (Y dado

[2] John Kirkham escribe: "Empecé a trabajar con computadores en 1962 utilizando FORTRAN II en un IBM 1620. En ese tiempo, y a través de los años sesenta y setenta, FORTRAN era un lenguaje todo en mayúsculas. Esto empezó probablemente porque muchos de los primeros dispositivos de entrada eran viejas unidades de teletipo que utilizaban código Baudot de 5 bits, que no tenía capacidad de empleo de minúsculas. La 'E' para la notación exponencial era también siempre mayúscula y nunca se confundía con la base de los logaritmos naturales 'e', que siempre era minúscula. La 'E' simplemente quería decir siempre exponencial, que era la base del sistema de numeración utilizado —generalmente 10. En ese momento se comenzó a extender entre los programadores el sistema octal. Aunque yo nunca lo vi usar, si hubiera visto un número octal en notación exponencial, habría considerado que tenía base 8. La primera vez que recuerdo ver un exponencial utilizando una 'e' minúscula fue al final de los setenta, y lo encontré bastante confuso. El problema aumentó cuando la 'e' se introdujo en FORTRAN, a diferencia de sus principios. De hecho, nosotros teníamos funciones para usar cuando realmente se quería usar la base logaritmica natural, pero todas ellas eran en mayúsculas".

que se está haciendo una asignación, de nuevo hacia un tipo más pequeño, se podría estar perdiendo información.) En general, el tipo de datos de mayor tamaño en una expresión será el que determine el tamaño del resultado de esa expresión; si se multiplica un **float** y un **double**, el resultado será **double**; si se suman un **int** y un **long**, el resultado será **long**.

Java no tiene "sizeof"

En C y C++, el operador **sizeof( )** satisface una necesidad específica: nos dice el número de bits asignados a elementos de datos. La necesidad más apremiante de **sizeof( )** en C y C++ es la portabilidad. Distintos tipos de datos podrían tener distintos tamaños en distintas máquinas, por lo que el programador debe averiguar cómo de grandes son estos tipos de datos, al llevar a cabo operaciones sensibles al tamaño. Por ejemplo, un computador podría almacenar enteros en 32 bits, mientras que otro podría almacenar enteros como 16 bits. Los programas podrían almacenar enteros con valores más grandes en la primera de las máquinas. Como podría imaginarse, la portabilidad es un gran quebradero de cabeza para los programadores de C y C++.

Java no necesita un operador **sizeof( )** para este propósito porque todos los tipos de datos tienen los mismos tamaños en todas las máquinas. No es necesario pensar en la portabilidad a este nivel —está intrínsecamente diseñada en el propio lenguaje.

Volver a hablar acerca de la precedencia

Tras oír quejas en uno de mis seminarios, relativas a la complejidad de recordar la precedencia de los operadores uno de mis alumnos sugirió un recurso mnemónico que es simultáneamente un comentario (en inglés): "Ulcer Addicts Really Like C A lot."

Mnemónico	Tipo de operador	Operador
Ulcer	Unario	+ – ++ –
Addicts	Aritméticos (y de desplazamiento)	* / % + – << >>
Really	Relacional	> < >= <= == !=
Like	Lógicos (y de bit)	&& || & | ^
C	Condicional (ternario)	**A > B ? X : Y**
A Lot	Asignación	= (y asignaciones compuestas como *=)

Por supuesto, con los operadores de desplazamiento y de bit distribuidos por toda la tabla, el recurso mnemónico no es perfecto, pero funciona para las operacione de no bit.

Un compendio de operadores

El ejemplo siguiente muestra qué tipos de datos primitivos pueden usarse como operadores particulares. Básicamente, es el mismo ejemplo repetido una y otra vez, pero usando distintos tipos de datos primitivos. El fichero se compilará sin error porque las líneas que causarían errores están marcadas como comentarios con un //!.

```
//: c03:TodosOperadores.java
// Prueba todos los operadores con
// todos los tipos de datos para probar
// cuáles son aprobados por el compilador de Java.

class TodosOperadores {
  // Para aceptar los resultados de un test booleano:
  void f(boolean b) {}
  void pruebaBool(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 bit:
    //! 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;
    // Conversión:
    //! 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 pruebaChar(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 bit:
    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;
    // Conversión:
    //! 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 pruebaByte(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);
    // Operadores de bit:
    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;
    // Conversión:
    //! 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 pruebaShort(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 bit:
    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;
    // Conversión:
    //! 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 pruebaInt(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 bit:
    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;
    // Conversión:
    //! 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 pruebaLong(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 bit:
    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;
    // Conversión:
    //! 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 pruebaFloat(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 bit:
  //! 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;
  // Conversión:
  //! 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 pruebaDouble(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 bit:
    //! 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;
    // Conversión:
    //! 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;
  }
} ///:~
```

Fíjese que **boolean** es bastante limitado. Se le pueden asignar los valores **true** y **false**, y se puede comprobar su validez o falsedad, pero no se pueden sumar valores lógicos o llevar a cabo ningún otro tipo de operación sobre ellos.

En **char**, **byte** y **short** se puede ver el efecto de promoción con los operadores aritméticos. Cada operación aritmética que se haga con estos tipos genera como resultado un **int**, que debe ser explícitamente convertido para volver al tipo original (una conversión reductora que podría implicar pérdida de información) para volver a ser asignado a ese tipo. Con los valores **int**, sin embargo, no es necesaria ninguna conversión, porque todo es ya un **int**. Aunque no hay que relajarse pensando que todo está ya a salvo. Si se multiplican dos **valores de tipo int** lo suficientemente grandes, se desbordará el resultado. Esto se demuestra en el siguiente ejemplo:

```
//: c03:Desbordamiento.java
// ¡Sorpresa! Java permite desbordamientos.
public class Desbordamiento {
  public static void main(String[] args) {
    int grande = 0x7fffffff; // Valor entero máximo
    visualizar("grande = " + grande);
    int mayor = grande * 4;
    visualizar("mayor = " + mayor);
  }
  static void visualizar(String s) {
    System.out.println(s);
  }
} ///:~
```

La salida de esto es:

```
grande = 2147483647
mayor = -4
```

y no se recibe ningún error ni advertencia proveniente del compilador, ni excepciones en tiempo de ejecución. Java es bueno, pero no tanto.

La asignaciones compuestas no requieren conversiones para **char**, **byte** o **short**, incluso aunque estén llevando a cabo promociones que tienen los mismos resultados que los operadores aritméticos directos. Por otro lado, la falta de conversión, definitivamente, simplifica el código.

Se puede ver que, con la excepción de **boolean**, cualquier tipo primitivo puede convertirse a otro tipo primitivo. De nuevo, debemos ser conscientes del efecto de la conversión reductora cuando se hace una conversión a un tipo menor. Si no, se podría perder información sin saberlo durante la conversión.

Control de ejecución

Java utiliza todas las sentencias de control de ejecución de C, de forma que si se ha programado con C o C++, la mayoría de lo que se ha visto será familiar. La mayoría de los lenguajes procedurales tienen algún tipo de sentencia de control, y casi siempre hay solapamiento entre lenguajes. En Java, las palabras clave incluyen **if-else**, **while**, **do-while**, **for**, y una sentencia de selección denominada **switch**. Java, sin embargo, no soporta el siempre perjudicial **goto** (lo que podría seguir siendo la manera más expeditiva de solventar cierto tipo de problemas). Todavía se puede hacer un salto del estilo del "goto", pero es mucho más limitado que un **goto** típico.

True y false

Todas las sentencias condicionales utilizan la certeza o falsedad de una expresión de condición para determinar el cauce de ejecución. Un ejemplo de una expresión condicional es **A == B**. Ésta hace uso del operador condicional == para ver si el valor de **A** es equivalente al valor de **B**. La expresión devuelve **true** o **false**. Cualquiera de los operadores relacionales vistos anteriormente en este capítulo puede usarse para producir una sentencia condicional. Fíjese que Java no permite utilizar un número como un **boolean**, incluso aunque está permitido en C y C++ (donde todo lo distinto de cero es verdadero, y cero es falso). Si se quiere usar un valor que no sea lógico en una conducción lógica, como **if(a)**, primero es necesario convertirlo a un valor **boolean** utilizando 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, por lo que puede usarse **if** de dos formas:

```
if(expresión condicional)
  sentencia
```

o

```
if(expresión condicional)
  sentencia
else
  sentencia
```

La expresión condicional debe producir un resultado **boolean**. La *sentencia* equivale bien a una sentencia simple acabada en un punto y coma, o a una sentencia compuesta, que es un conjunto de sentencias simples encerradas entre llaves. Cada vez que se use la palabra *sentencia*, siempre implicará que ésta puede ser simple o compuesta.

He aquí un método **prueba( )** como ejemplo de **if-else**. Se trata de un método que indica si un número dicho en un acertijo es mayor, menor o equivalente al número solución:

```
//: c03:IfElse.java
public class IfElse {
  static int prueba(int intento, int solucion) {
  int resultado = 0;
  if(intento > solucion)
      resultado = +1;
  else if(intento < solucion)
      resultado = -1;
  else
      resultado = 0; // Coincidir
  return resultado;
  }
  public static void main(String[] args) {
  System.out.println(prueba(10, 5));
  System.out.println(prueba(5, 10));
  System.out.println(prueba(5, 5));
  }
} ///:~
```

Es frecuente alinear el cuerpo de una sentencia de control de flujo, de forma que el lector pueda determinar fácilmente dónde empieza y dónde acaba.

return

La palabra clave **return** tiene dos propósitos: especifica qué valor devolverá un método (si no tiene un valor de retorno **void**), y hace que el valor se devuelva inmediatamente. El método **prueba( )** puede reescribirse para sacar ventaja de esto:

```
//: c03:IfElse2.java
public class IfElse2 {
  static int prueba(int intento, int solucionar) {
  int resultado = 0;
  if(intento > solucionar)
  return +1;
  else if(intento < solucionar)
return -1;
else
  return 0; // Coincidir
```

```
  }
  public static void main(String[] args) {
  System.out.println(prueba(10, 5));
  System.out.println(prueba(5, 10));
  System.out.println(prueba(5, 5));
  }
} ///:~
```

No hay necesidad de **else** porque el método no continuará ejecutándose una vez que se ejecute el **return**.

Iteración

Las sentencias **while**, **do-while** y **for** son para el control de bucles, y en ocasiones se clasifican como *sentencias de iteración*. Se repite una *sentencia* hasta que la expresión *Condicional* controladora se evalúe a falsa. La forma de un bucle **while** es:

while(Expresión-*Condicional*)
sentencia

La *expresión condicional* se evalúa al comienzo de cada interación del bucle, y de nuevo antes de cada iteración subsiguiente de la *sentencia*.

He aquí un ejemplo sencillo que genera números aleatorios hasta que se dé una condición determinada:

```
//: c03:PruebaWhile.java
// Muestra el funcionamiento del bucle while.

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

Este ejemplo usa el método **estático random( )** de la biblioteca **Math**, que genera un valor **double** entre 0 y 1. (Incluye el 0, pero no el 1.) La expresión condicional para el **while** dice "siga haciendo este bucle hasta que el número sea 0,99 o mayor". Cada vez que se ejecute este programa, se logrará un listado de números de distinto tamaño.

do-while

La forma del **do-while** es

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

La única diferencia entre **while** y **do-while** es que la sentencia del **do-while** se ejecuta siempre, al menos, una vez, incluso aunque la expresión se evalúe como falsa la primera vez. En un **while**, si la condicional es falsa la primera vez, la sentencia no se ejecuta nunca. En la práctica, **do-while** es menos común que **while**.

for

Un bucle **for** lleva a cabo la inicialización antes de la primera iteración. Después, lleva a cabo la comprobación condicional y, al final de cada iteración, hace algún tipo de "paso". La forma del bucle **for** es:

```
for(inicialización; Expresión condicional; paso)
sentencia
```

Cualquiera de las expresiones *inicialización*, *expresión condicional* o *paso* puede estar vacía. Dicha expresión se evalúa antes de cada iteración, y en cuanto el resultado sea falso, la ejecución continuará en la línea siguiente a la sentencia **for**. Al final de cada iteración se ejecuta *paso*.

Los bucles **for** suelen utilizarse para crear contadores:

```
//: c03:ListaCaracteres.java
// Muestra el funcionamiento del bucle "for" listando
// todos los caracteres ASCII.

public class ListaCaracteres {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )   // Limpiar pantalla ANSI
      System.out.println(
        "valor: " + (int)c +
        " caracter: " + c);
  }
} ///:~
```

Fíjese en que la variable **c** está definida en el punto en que se usa, dentro de la expresión de control del bucle **for**, en vez de al principio del bloque delimitado por la llave de apertura. El ámbito de **c** es la expresión controlada por el **for**.

Los lenguajes procedurales tradicionales como C requieren que todas las variables se definan al principio de un bloque, de forma que cuando el compilador cree un bloque, pueda asignar espacio para esas variables. En Java y C++ es posible diseminar las declaraciones de variables a lo largo del bloque, definiéndolas en el momento en que son necesarias. Esto permite un estilo de codificación más natural y hace que el código sea más fácil de entender.

Se puede definir múltiples variables dentro de una sentencia **for**, pero deben ser del mismo tipo:

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

La definición **int** de la sentencia **for** cubre tanto a **i** como a **j**. La habilidad de definir variables en expresiones de control se limita al bucle **for**. No se puede utilizar este enfoque con cualquiera de las otras sentencias de selección o iteración.

El operador coma

Anteriormente en este capítulo, dije que el *operador* coma (no el *separador* coma, que se usa para separar definiciones y parámetros de funciones) sólo tiene un uso en Java: en la expresión de control de un bucle **for**. Tanto en la inicialización como en las porciones de "paso" de las expresiones de control, se tiene determinado número de sentencias separadas por comas, y estas sentencias se evaluarán secuencialmente. El fragmento de bloque previo utiliza dicha capacidad. He aquí otro ejemplo:

```
//: c03:OperadorComa.java
public class OperadorComa {
     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);
     }
   }
} ///:~
```

He aquí la salida:

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

Se puede ver que tanto en la inicialización, como en las porciones de "paso" se evalúan las sentencias en orden secuencial. Además, la porción de inicialización puede tener cualquier número de definiciones *de un tipo*.

break y continue

Dentro del cuerpo de cualquier sentencia de iteración también se puede controlar el flujo del bucle utilizando **break** y **continue**. **Break** sale del bucle sin ejecutar el resto de las sentencias del bucle. **Continue** detiene la ejecución de la iteración actual y vuelve al principio del bucle para comenzar la siguiente iteración.

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

```
//: c03:BreakYContinue.java
// Muestra el funcionamiento de las palabras clave break y continue.
public class BreakYContinue {
     public static void main(String[] args) {
for(int i = 0; i < 100; i++) {
if(i == 74) break; // Sale del bucle for
  if(i % 9 != 0) continue; // Siguiente iteración
  System.out.println(i);
}
  int i = 0;
  // Un "bucle infinito":
  while(true) {
    i++;
    int j = i * 27;
  if(j == 1269) break; // Sale del bucle
  if(i % 10 != 0) continue; // Parte superior del bucle
  System.out.println(i);
     }
   }
} ///:~
```

En el bucle **for** el valor de **i** nunca llega a 100 porque la sentencia **break** rompe el bucle cuando **i** vale 74. Normalmente, el **break** sólo se utilizaría de esta manera si no se supiera cuándo va a darse la condición de terminación. La sentencia **continue** hace que la ejecución vuelva a la parte superior del bucle de iteración (incrementando por consiguiente la **i**) siempre que **i** no sea totalmente divisible por 9. Cuando lo es, se imprime el valor.

La segunda porción muestra un "bucle infinito" que debería, en teoría, continuar para siempre. Sin embargo, dentro del bucle hay una sentencia **break** que romperá el bucle y saldrá de él. Además, se verá que la sentencia **continue** vuelve a la parte de arriba del bucle sin completar el resto.

(Por consiguiente la impresión se da en el segundo bucle sólo cuando el valor de **i** es divisible por 10.) La salida es:

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

El valor 0 se imprime porque 0 % 9 da 0.

Una segunda forma de hacer un bucle infinito es escribir **for(;;)**. El compilador trata tanto a **while (true),** como a **for(;;)** de la misma manera, de forma que cualquiera que se use en cada caso, no es más que una cuestión de gusto.

El infame "goto"

La palabra clave **goto** ha estado presente en los lenguajes de programación desde los comienzos. Sin duda, el **goto** era la génesis del control de los programas en el lenguaje ensamblador: "Si se da la condición A, entonces saltar aquí, sino, saltar ahí." Si se lee el código ensamblador generado al final por cualquier compilador, se verá que el control del programa contiene muchos saltos. Sin embargo, un **goto** es un salto al nivel de código fuente, y eso es lo que le ha traído tan mala reputación. Si un programa siempre salta de un punto a otro, ¿no hay forma de reorganizarlo de manera que el flujo de control no dé tantos saltos? **goto** cayó en desgracia con la publicación del famoso artículo "El Goto considerado dañino"[3], de Edsger Dijkstra, y desde entonces, la prohibición del goto ha sido un deporte popular, con los partidarios de la palabra clave repudiada buscando guarida.

Como es típico en situaciones como ésta, el terreno imparcial es el más fructífero. El problema no es el uso del **goto**, sino el uso excesivo de **goto** —en raras ocasiones el **goto** es de hecho la mejor manera de estructurar el flujo del programa.

Aunque **goto** es una palabra reservada en Java, no se utiliza en el lenguaje; Java no tiene **goto**. Sin embargo, tiene algo que se parece un poco a un salto atado vinculado a las palabras clave **break** y **continue**. No es un salto sino más bien una forma de romper una sentencia de iteración. El motivo por el que aparece muy a menudo en discusiones relacionadas con el **goto,** es que utiliza el mismo mecanismo: una etiqueta.

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

```
etiqueta1:
```

El *único* sitio en el que una etiqueta es útil en Java es justo antes de una sentencia de iteración. Y eso significa *justo* antes —no hace ningún bien poner cualquier otra sentencia entre la etiqueta y la iteración. Y la única razón para poner una etiqueta antes de una iteración es si se va a anidar otra iteración o un "switch" dentro. Eso es porque las palabras **break** y **continue** únicamente interrumpirán normalmente al bucle actual, pero cuando se usan con una etiqueta, interrumpirán a los bucles hasta donde exista la etiqueta:

```
etiqueta1:
iteracion-externa {
      iteracion-interna {
  //...
```

[3] Nota del traductor: "Goto considered harmful".

```
      break; //1
      //...
      continue; //2
      //...
      continue etiqueta1; //3
      //...
      break etiqueta1; //4
    }
}
```

En el caso 1, el **break** rompe la iteración interna, pasando a la iteración exterior. En el caso 2, el **continue** hace volver al principio de la iteración interna. Pero en el caso 3, el **continue etiqueta1** rompe tanto la iteración interna, *como* la externa, retrocediendo hasta **etiqueta1**. Posteriormente, de hecho, continúa la iteración, pero empezando en la iteración exterior. En el caso 4, el **break etiqueta1** también rompe el bucle haciendo volver hasta **etiqueta1**, pero no vuelve a entrar en la iteración. De hecho, rompe ambas iteraciones.

He aquí un ejemplo de utilización de bucles **for**:

```
//: c03:ForEtiquetado.java
// El bucle "for etiquetado" de Java.

public class ForEtiquetado {
public static void main(String[] args) {
      int i = 0;
      externo: // Aquí no puede haber sentencias
      for(; true ;) { // bucle infinito
      interno: // Aquí no puede haber sentencias
      for(; i < 10; i++) {
  visualizar("i = " + i);
  if(i == 2) {
  visualizar("continuar");
    continue;
          }
  if(i == 3) {
  visualizar("salir");
  i++; // En caso contrario i
  // no se incrementa nunca.
  break;
          }
  if(i == 7) {
  visualizar("continuar el externo");
  i++; // En caso contrario i
  // no se incrementa nunca.
  continue externo;
          }
```

```
    if(i == 8) {
    visualizar("salir externo");
    break externo;
            }
    for(int k = 0; k < 5; k++) {
    if(k == 3) {
    prt("continuar el interno");
    continue interno;
              }
            }
          }
       }
    // Aquí no se puede hacer break o continue
    // a etiquetas
    }
    static void visualizar(String s) {
    System.out.println(s);
    }
} ///:~
```

Este ejemplo usa el método **visualizar( )** que ha sido definido en los otros ejemplos.

Nótese que **break** sale del bucle **for**, y que la expresión de incremento no se da hasta acabar de pasar por el bucle **for**. Dado que **break** se salta la expresión e incremento, el incremento se da directamente en el caso de **i==3**. La sentencia **continuar externo** en el caso de **i==7** va también a la parte superior del bucle, y se salta también el incremento, por lo que también se incrementa directamente.

He aquí la salida:

```
i = 0
continuar el interno
i = 1
continuar el interno
i = 2
continuar
i = 3
salir
i = 4
continuar el interno
i = 5
continuar el interno
i = 6
continuar el interno
i = 7
continuar el externo
```

```
i = 8
salir externo
```

Si no fuera por la sentencia **break externo**, no habría manera de salir del bucle externo desde el bucle interno, dado que **break**, por sí misma puede romper únicamente el bucle más interno. (Y lo mismo ocurre con **continue**.)

Por supuesto, en los casos en los que salir de un bucle implique también salir del método, uno puede usar simplemente un **return**.

He aquí una demostración de sentencias etiquetadas **break** y **continue** con bucles **while**:

```
//: c03:WhileEtiquetado.java
// El bucle "while etiquetado" de Java.

public class WhileEtiquetado {
  public static void main(String[] args) {
  int i = 0;
  externo:
  while(true) {
    visualizar("Bucle while externo");
    while(true) {
      i++;
    visualizar("i = " + i);
      if(i == 1) {
    visualizar("continuar");
        continue;
}
  if(i == 3) {
    visualizar("Continuar externo");
    continue externo;
}
  if(i == 5) {
    visualizar("salir");
    break;
}
  if(i == 7) {
    visualizar("break externo");
    break externo;
          }
       }
     }
   }
  static void visualizar(String s) {
  System.out.println(s);
   }
} ///:~
```

Las mismas reglas son ciertas para **while**:

1. Un **continue** sin más va hasta el comienzo del bucle más interno, y continúa.
2. Un **continue** etiquetado va a la etiqueta, y vuelve a entrar en el bucle situado justo después de la etiqueta.
3. Un **break** "abandona" el bucle.
4. Un **break** etiquetado abandona el final del bucle marcado por la etiqueta.

La salida de este método lo deja claro:

```
Bucle while externo
i = 1
continuar
i = 2
i = 3
continuar externo
Bucle while externo
i = 4
i = 5
salir
Bucle while externo
i = 6
i = 7
salir externo
```

Es importante recordar que la *única* razón para usar etiquetas en Java es cuando se tienen bucles anidados, y se quiere utilizar sentencias **break** o **continue** a través de más de un nivel de anidamiento.

En el artículo "El goto considerado dañino" de Dijkstra, se ponen objeciones a las etiquetas, no al goto en sí. Dijkstra observó que el número de errores tiende a incrementarse con el número de etiquetas que haya en un programa. Las etiquetas y las sentencias goto hacen difícil el análisis estático, puesto que introducen ciclos en el grafo de ejecución de los programas. Fíjese que las etiquetas de Java no tienen este problema, pues están limitadas a su ubicación, y no pueden ser utilizadas para transferir el control de forma directa. También es interesante tener en cuenta que éste es el caso en el que una característica de un lenguaje se convierte en más interesante, simplemente restringiendo el poder de la propia sentencia.

switch

La orden **switch** suele clasificarse como *sentencia de selección*. La sentencia **switch** selecciona de entre fragmentos de código basados en el valor de una expresión entera. Es de la forma:

```
switch (selector-entero) {
  case valor-entero1 : sentencia; break;
```

```
  case valor-entero2 : sentencia; break;
  case valor-entero3 : sentencia; break;
  case valor-entero4 : sentencia; break;
  case valor-entero5 : sentencia; break;
        // ...
  default : sentencia;
}
```

El *selector entero* es una expresión que produce un valor entero. El **switch** compara el resultado de *selector entero* con cada *valor entero*. Si encuentra un valor que coincida, ejecuta la sentencia (simple o compuesta) correspondiente. Si no encuentra ninguna coincidencia, ejecuta la *sentencia* **default**.

Observese en la definición anterior que cada **case** acaba con **break**, lo que causa que la ejecución salte al final del cuerpo de la sentencia **switch**. Ésta es la forma convencional de construir una sentencia **switch**, pero el **break** es opcional. Si no se pone, se ejecutará el código de las sentencias "case" siguientes, hasta encontrar un **break**. Aunque este comportamiento no suele ser el deseado, puede ser útil para un programador experimentado. Hay que tener en cuenta que la última sentencia, la que sigue a **default**, no tiene **break** porque la ejecución llega hasta donde le hubiera llevado el **break**. Se podría poner un **break** al final de la sentencia **default** sin que ello causara ningún daño, si alguien lo considerara importante por razones de estilo.

La sentencia **switch** es una forma limpia de implementar una selección de múltiples caminos (por ejemplo, seleccionar un camino de entre cierto número de caminos de ejecución diferentes), pero requiere de un selector que se evalúe a un valor como **int** o **char**. Si se desea utilizar, por ejemplo, una cadena de caracteres o un número de coma flotante como selector, no se podrá utilizar una sentencia **switch**. En el caso de tipos no enteros, es necesario utilizar una serie de sentencias **if**.

He aquí un ejemplo que crea letras al azar y determina si se trata de vocales o consonantes:

```
//: c03:VocalesYConsonantes.java
// Demuestra el funcionamiento de la sentencia switch.

public class VocalesYConsonantes {
  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("vocal");
                break;
      case 'y':
```

```
        case 'w':
                    System.out.println(
                      "A veces una vocal");
                    break;
        default:
                    System.out.println("consonante");
        }
      }
   }
} ///:~
```

Dado que **Math.random( )** genera un valor entre 0 y 1, sólo es necesario multiplicarlo por el límite superior del rango de números que se desea producir (26 para las letras del alfabeto) y añadir un desplazamiento para establecer el límite inferior.

Aunque aquí parece que se está haciendo un **switch** con un carácter, esta sentencia está usando, de hecho, el valor entero del carácter. Los caracteres entre comillas simples de las sentencias **case** también producen valores enteros que se usan para las comparaciones.

Fíjese cómo las sentencias **case** podrían "apilarse" unas sobre otras para proporcionar varias coincidencias para un fragmento de código particular. También habría que ser conscientes de que es esencial poner la sentencia **break** al final de un caso particular, de otra manera, el control simplemente irá descendiendo, pasando a ejecutar el **case** siguiente.

Detalles de cálculo

La sentencia

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

merece una mirada más detallada. **Math.random( )** produce un **double**, por lo que se convierte el valor 26 a **double** para llevar a cabo la multiplicación, que también produce un **double**. Esto significa que debe convertirse la **'a'** a **double** para llevar a cabo la suma. El resultado **double** se vuelve a convertir en **char** con un **molde**.

¿Qué es lo que hace la conversión a **char**? Es decir, si se tiene el valor 29,7 y se convierte a **char,** ¿cómo se sabe si el valor resultante es 30 o 29? La respuesta a esta pregunta se puede ver en este ejemplo:

```
//: c03:ConvertirNumeros.java
// ¿Qué ocurre cuando se convierte un float
// o un double a un valor entero?

public class ConvertirNumeros {
  public static void main(String[] args) {
    double
      encima = 0.7,
      debajo = 0.4;
```

```
      System.out.println("encima: " + encima);
      System.out.println("debajo: " + debajo);
      System.out.println(
        "(int)encima: " + (int)encima);
      System.out.println(
        "(int)debajo: " + (int)debajo);
      System.out.println(
        "(char)('a' + encima): " +
        (char)('a' + encima));
      System.out.println(
        "(char)('a' + debajo): " +
        (char)('a' + debajo));
   }
} ///:~
```

La salida es:

```
encima: 0.7
debajo: 0.4
(int)encima: 0
(int)debajo: 0
(char)('a' + encima) = a
(char)('a' + debajo) = a
```

Por lo que la respuesta es que si se hace una conversión de un **float** o un **double** a un valor entero lo truncará.

Hay una segunda cuestión que concierne a **Math.random( )**. ¿Produce un valor de cero a uno, incluyendo o excluyendo al valor '1'? En el lingo matemático ¿es (0, 1) o [0, 1], o (0, 1] o [0, 1)? (El corchete significa "incluye" mientras que el paréntesis significa "excluye".) De nuevo, la solución la puede proporcionar un programa de prueba:

```
//: c03:LimitesAleatorios.java
// ¿Produce Math.random()0.0 y 1.0?

public class LimitesAleatorios {
   static void uso() {
      System.out.println("Utilizacion: \n\t" +
        "LimitesAleatorios inferior\n\t" +
        "LimitesAleatorios superior");
      System.exit(1);
   }
   public static void main(String[] args) {
      if(args.length != 1) uso();
      if(args[0].equals("inferior")) {
        while(Math.random() != 0.0)
          ; // Seguir intentándolo
```

```
        System.out.println("Produjo 0.0!");
      }
      else if(args[0].equals("superior")) {
        while(Math.random() != 1.0)
          ; // Seguir intentandolo
        System.out.println("Produjo 1.0!");
      }
      else
        uso();
    }
} ///:~
```

Para ejecutar el programa, se teclea una línea de comandos como:

```
java LimitesAleatorios inferior
```

o

```
java LimitesAleatorios superior
```

En ambos casos nos vemos forzados a romper el programa manualmente, de forma que da la sensación de que **Math.random( )** nunca produce ni 0,0 ni 1,0. Pero éste es el punto en el que un experimento así puede defraudar. Si se considera[4] que hay al menos 2^{62} fracciones **double** distintas entre 0 y 1, la probabilidad de alcanzar cualquier valor experimentalmente podría superar el tiempo de vida de un computador o incluso el de la persona que realiza la prueba. Resulta que 0,0 *está* incluido en la salida de **Math.random( )**. O, en el lingo de las matemáticas es [0, 1).

Resumen

Este capítulo concluye el estudio de los aspectos fundamentales que aparecen en la mayoría de los lenguajes de programación: cálculo, precedencia de operadores, conversión de tipos, y selección e iteración. Ahora estamos listos para empezar a dar pasos y acercarse al mundo de la programación

---

[4] Chuck Allison escribe: "El número total de números en el sistema de números en coma flotante es **2(M–m+1) b^(p–1) +1**, donde **b** es la base (generalmente 2), **p** es la precisión (dígitos de la mantisa), **M** es el exponente mayor, y **m** es el exponente menor. IEEE 754 utiliza:

M = 1023, m = -1022, p = 53, b = 2

por lo que el número total de números 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 (los correspondientes a los exponentes del rango [-1022, 1] son menores a 1 en magnitud (tanto positivos como negativos), por lo que 1/4 de esa expresión, o 2^62 – 2^52+1 (aproximadamente 2^62) está en el rango [0, 1). Véase mi artículo en

*http://www.freshsources.com/*1995006.htm (final del texto).

orientada a objetos. El siguiente capítulo cubrirá los aspectos importantes de la inicialización y limpieza de objetos, seguido del esencial concepto de ocultación de información en el capítulo siguiente.

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com.*

1. Hay dos expresiones en la sección denominada "precedencia" de este capítulo. Poner estas expresiones en un programa y demostrar que producen resultados diferentes.
2. Poner los métodos temario( ) y alternativo( ) en un programa que funcione.
3. Poner los métodos prueba( ) y prueba2( ) de las secciones "if-else" y "return" en un programa que funcione.
4. Escribir un programa que imprima valores de 1 a 100.
5. Modificar el Ejercicio 4, de forma que el programa exista utilizando la palabra clave break en el valor 47. Intentar hacerlo usando return en vez de break.
6. Escribir una función que reciba como parámetros dos cadenas de texto, y use todas las comparaciones lógicas para comparar ambas cadenas e imprimir los resultados. Para el caso de == y !=, llevar a cabo también las pruebas de equals( ). En main( ), llamar a la función con varios objetos String distintos.
7. Escribir un programa que genere 25 valores enteros al azar. Para cada valor, utilizar una sentencia if-then-else para clasificarlo como mayor, menor o igual que un segundo valor generado al azar.
8. Modificar el Ejercicio 7, de forma que el código esté dentro de un bucle while "infinito". Después se ejecutará hasta que se interrumpa desde el teclado (generalmente presionando Control-C).
9. Escribir un programa que use dos bucles for anidados y el operador módulo (%) para detectar e imprimir números primos (números enteros que no son divisibles por otro número que no sean ellos mismos o 1).
10. Crear una sentencia switch que escriba un mensaje en cada caso, e introducirla en un bucle for que pruebe cada caso. Poner un break después de cada caso y probarlo. A continuación, quitar las sentencias break y ver qué ocurre.