Operadores

3

En el nivel inferior, los datos en Java se manipulan utilizando operadores.

Puesto que Java surgió a partir de C++, la mayoría de estos operadores les serán familiares a casi todos los programadores de C y C++. Java ha añadido también algunas mejoras y ha hecho algunas simplificaciones.

Si está familiarizado con la sintaxis de C o C++, puede pasar rápidamente por este capítulo y el siguiente, buscando aquellos aspectos en los que Java se diferencie de esos lenguajes. Por el contrario, si le asaltan dudas en estos dos capítulos, repase el seminario multimedia *Thinking in C*, que puede descargarlo gratuitamente en *www.MindView.net*. Contiene conferencias, presentaciones, ejercicios y soluciones específicamente diseñados para familiarizarse con los fundamentos necesarios para aprender Java.

Instrucciones simples de impresión

En el capítulo anterior, ya hemos presentado la instrucción de impresión en Java:

```
System.out.println("Un montón de texto que escribir");
```

Puede observar que esto no es sólo un montón de texto que escribir (lo que quiere decir muchos movimientos redundantes de los dedos), sino que también resulta bastante incómodo de leer. La mayoría de los lenguajes, tanto antes como después de Java, han adoptado una técnica mucho más sencilla para expresar esta instrucción de uso tan común.

En el Capítulo 6, *Control de acceso* se presenta el concepto de *importación estática* añadido a Java SE5, y se crea una pequeña biblioteca que permite simplificar la escritura de las instrucciones de impresión. Sin embargo, no es necesario comprender los detalles para comenzar a utilizar dicha biblioteca. Podemos reescribir el programa del último capítulo empleando esa nueva biblioteca:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
  public static void main(String[] args) {
    print("Hello, it's: ");
    print(new Date());
  }
} /* Output: (55% match)
Hello, it's:
Wed Oct 05 14:39:05 MDT 2005
*///:~
```

Los resultados son mucho más limpios. Observe la inserción de la palabra clave **static** en la segunda instrucción **import**.

Para emplear esta biblioteca, es necesario descargar el paquete de código del libro desde *www.MindView.net* o desde alguno de los sitios espejo. Descomprima el árbol de código y añada el directorio raíz de dicho árbol de código a la variable de entorno CLASSPATH de su computadora (más adelante proporcionaremos una introducción completa a la variable de ruta CLASSPATH, pero es muy probable que el lector ya esté acostumbrado a lidiar con esa variable. De hecho, es una de las batallas más comunes que se presentan al intentar programar en Java).

Aunque la utilización de **net.mindview.util.Print** simplifica enormemente la mayor parte del código, su uso no está justificado en todas las ocasiones. Si sólo hay unas pocas instrucciones de impresión en un programa, es preferible no incluir la instrucción **import** y escribir el comando completo **System.out.println( )**.

Ejercicio 1: (1) Escriba un programa que emplee tanto la forma "corta" como la normal de la instrucción de impresión.

Utilización de los operadores Java

Un operador toma uno o más argumentos y genera un nuevo valor. Los argumentos se incluyen de forma distinta a las llamadas normales a métodos, pero el efecto es el mismo. La suma, el operador más unario (+), la resta y el operador menos unario (−), la multiplicación (*), la división (/) y la asignación (=) funcionan de forma bastante similar a cualquier otro lenguaje de programación.

Todos los valores producen un valor a partir de sus operandos. Además, algunos operadores cambian el valor del operando, lo que se denomina *efecto colateral*. El uso más común de los operadores que modifican sus operandos consiste precisamente en crear ese efecto colateral, pero resulta conveniente pensar que el valor generado también está disponible para nuestro uso, al igual que sucede con los operadores que no tienen efectos colaterales.

Casi todos los operadores funcionan únicamente con primitivas. Las excepciones son '=', '==' y '!=', que funcionan con todos los objetos (y son una fuente de confusión con los objetos). Además, la clase **String** soporta '+' y '+='.

Precedencia

La precedencia de los operadores define la forma en que se evalúa una expresión cuando hay presentes varios operadores. Java tiene una serie de reglas específicas que determinan el orden de evaluación. La más fácil de recordar es que la multiplicación y la división se realizan antes que la suma y la resta. Los programadores se olvidan a menudo de las restantes reglas de precedencia, así que conviene utilizar paréntesis para que el orden de evaluación esté indicado de manera explícita. Por ejemplo, observe las instrucciones **(1)** y **(2)**:

```
//: operators/Precedence.java

public class Precedence {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;            // (1)
    int b = x + (y - 2)/(2 + z);        // (2)
    System.out.println("a = " + a + " b = " + b);
  }
} /* Output:
a = 5 b = 1
*///:~
```

Estas instrucciones tienen aspecto similar, pero la salida nos muestra que sus significados son bastante distintos, debido a la utilización de paréntesis.

Observe que la instrucción **System.out.println( )** incluye el operador '+'. En este contexto, '+' significa "concatenación de cadenas de caracteres" y, en caso necesario, "conversión de cadenas de caracteres." Cuando el compilador ve un objeto **String** seguido de '+' seguido de un objeto no **String**, intenta convertirlo a un objeto **String**. Como puede ver en la salida, se ha efectuado correctamente la conversión de **int** a **String** para **a** y **b**.

Asignación

La asignación se realiza con el operador =. Su significado es: "Toma el valor del lado derecho, a menudo denominado *rvalor*, y cópialo en el lado izquierdo (a menudo denominado *lvalor*)". Un rvalor es cualquier constante, variable o expresión que genere un valor, pero un lvalor debe ser una variable determinada, designada mediante su nombre (es decir, debe existir un espacio físico para almacenar el valor). Por ejemplo, podemos asignar un valor constante a una variable:

```
a = 4;
```

pero no podemos asignar nada a un valor constante, ya que una constante no puede ser un lvalor (no podemos escribir **4 = a;**).

La asignación de primitivas es bastante sencilla. Puesto que la primitiva almacena el valor real y no una referencia a cualquier objeto, cuando se asignan primitivas se asigna el contenido de un lugar a otro. Por ejemplo, si escribimos **a = b** para primitivas, el contenido de **b** se copia en **a**. Si a continuación modificamos **a**, el valor de **b** no se verá afectado por esta modificación. Como programador es lo que cabría esperar en la mayoría de las situaciones.

Sin embargo, cuando asignamos objetos, la cosa cambia. Cuando manipulamos un objeto lo que manipulamos es la referencia, así que al asignar " de un objeto a otro", lo que estamos haciendo en la práctica es copiar una referencia de un lugar a otro. Esto significa que si escribimos **c = d** para sendos objetos, lo que al final tendremos es que tanto **c** como **d** apuntan al objeto al que sólo **d** apuntaba originalmente. He aquí un ejemplo que ilustra este comportamiento:

```
//: operators/Assignment.java
// La asignación de objetos tiene su truco.
import static net.mindview.util.Print.*;

class Tank {
  int level;
}

public class Assignment {
  public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    print("1: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1 = t2;
    print("2: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
    t1.level = 27;
    print("3: t1.level: " + t1.level +
          ", t2.level: " + t2.level);
  }
} /* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*///:~
```

La clase **Tank** es simple, y se crean dos instancias de la misma (**t1** y **t2**) dentro de **main( )**. Al campo **level** de cada objeto **Tank** se le asigna un valor distinto, luego se asigna **t2** a **t1**, y después se modifica **t1**. En muchos lenguajes de programación esperaríamos que **t1** y **t2** fueran independientes en todo momento, pero como hemos asignado una referencia, al cambiar el objeto **t1** se modifica también el objeto **t2.** Esto se debe a que tanto **t1** como **t2** contienen la misma referencia, que está apuntada en el mismo objeto (la referencia original contenida en **t1**, que apuntaba al objeto que tenía un valor de 9, fue sobreescrita durante la asignación y se ha perdido a todos los efectos; su objeto será eliminado por el depurador de memoria).

Este fenómeno a menudo se denomina *creación de alias*, y representa una de las características fundamentales del modo en que Java trabaja con los objetos. Pero, ¿qué sucede si no queremos que las dos referencias apunten al final a un mismo objeto? Podemos reescribir la asignación a otro nivel y utilizar:

```
t1.level = t2.level;
```

Esto hace que se mantengan independientes los dos objetos, en lugar de descartar uno de ellos y asociar **t1** y **t2** al mismo objeto. Más adelante veremos que manipular los campos dentro de los objetos resulta bastante confuso y va en contra de los principios de un buen diseño orientado a objetos. Se trata de un tema que no es nada trivial, así que tenga siempre presente que las asignaciones pueden causar sorpresas cuando se manejan objetos.

Ejercicio 2: (1) Cree una clase que contenga un valor **float** y utilícela para ilustrar el fenómeno de la creación de alias.

Creación de alias en las llamadas a métodos

El fenómeno de la creación de alias también puede manifestarse cuando se pasa un objeto a un método:

```
//: operators/PassObject.java
// El paso de objetos a los métodos puede no ser
// lo que cabría esperar.
import static net.mindview.util.Print.*;

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';
    print("1: x.c: " + x.c);
    f(x);
    print("2: x.c: " + x.c);
  }
} /* Output:
1: x.c: a
2: x.c: z
*///:~
```

En muchos lenguajes de programación, el método **f( )** haría una copia de su argumento **Letter** dentro del ámbito del método, pero aquí, una vez más, lo que se está pasando es una referencia, por lo que la línea:

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

lo que está haciendo es cambiar el objeto que está fuera de **f( )**.

El fenómeno de creación de alias y su solución es un tema complejo del que se trata en uno de los suplementos en línea disponibles para este libro. Sin embargo, conviene que lo tenga presente desde ahora, con el fin de detectar posibles errores.

Ejercicio 3: (1) Cree una clase que contenga un valor **float** y utilícela para ilustrar el fenómeno de la creación de alias durante las llamadas a métodos.

Operadores matemáticos

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

Java también utiliza la notación abreviada de C/C++ que realiza una operación y una asignación al mismo tiempo. Este tipo de operación se denota mediante un operador seguido de un signo de igual, y es coherente con todos los operadores del lenguaje (allí donde tenga sentido). Por ejemplo, para sumar 4 a la variable **x** y asignar el resultado a **x**, utilice: **x += 4**.

Este ejemplo muestra el uso de los operadores matemáticos:

```
//: operators/MathOps.java
// Ilustra los operadores matemáticos.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {
  public static void main(String[] args) {
```

```
    // Crea un generador de números aleatorios con una cierta semilla:
    Random rand = new Random(47);
    int i, j, k;
    // Elegir valor entre 1 y 100:
    j = rand.nextInt(100) + 1;
    print("j : " + j);
    k = rand.nextInt(100) + 1;
    print("k : " + k);
    i = j + k;
    print("j + k : " + i);
    i = j - k;
    print("j - k : " + i);
    i = k / j;
    print("k / j : " + i);
    i = k * j;
    print("k * j : " + i);
    i = k % j;
    print("k % j : " + i);
    j %= k;
    print("j %= k : " + j);
    // Pruebas con números en coma flotante:
    float u, v, w; // Se aplica también a los de doble precisión
    v = rand.nextFloat();
    print("v : " + v);
    w = rand.nextFloat();
    print("w : " + w);
    u = v + w;
    print("v + w : " + u);
    u = v - w;
    print("v - w : " + u);
    u = v * w;
    print("v * w : " + u);
    u = v / w;
    print("v / w : " + u);
    // Lo siguiente también funciona para char,
    // byte, short, int, long y double:
    u += v;
    print("u += v : " + u);
    u -= v;
    print("u -= v : " + u);
    u *= v;
    print("u *= v : " + u);
    u /= v;
    print("u /= v : " + u);
  }
} /* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
```

```
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*///:~
```

Para generar números, el programa crea en primer lugar un objeto **Random**. Si se crea un objeto **Random** sin ningún argumento, Java usa la hora actual como semilla para el generador de números aleatorios, y esto generaría una salida diferente en cada ejecución del programa. Sin embargo, en los ejemplos del libro, es importante que la salida que se muestra al final de los ejemplos sea lo más coherente posible, para poder verificarla con herramientas externas. Proporcionando una *semilla* (un valor de inicialización para el generador de números aleatorios que siempre genera la misma secuencia para una determinada semilla) al crear el objeto **Random**, se generarán siempre los mismos números aleatorios en cada ejecución del programa, por lo que la salida se podrá verificar.[1] Para generar una salida más variada, pruebe a eliminar la semilla en los ejemplos del libro.

El programa genera varios números aleatorios de distintos tipos con el objeto **Random** simplemente invocando los métodos **nextInt( )** y **nextFloat( )** (también se pueden invocar **nextLong( )** o **nextDouble( )**). El argumento de **nextInt( )** establece la cota superior para el número generado. La cota superior es cero, lo cual no resulta deseable debido a la posibilidad de una división por cero, por lo que sumamos uno al resultado.

Ejercicio 4: (2) Escriba un programa que calcule la velocidad utilizando una distancia constante y un tiempo constante.

Operadores unarios más y menos

El menos unario (-) y el más unario (+) son los mismos operadores que la suma y la resta binarias. El compilador deduce cuál es el uso que se le quiere dar al operador a partir de la forma en que está escrita la expresión. Por ejemplo, la instrucción:

```
x = -a;
```

tiene un significado obvio. El compilador también podría deducir el uso correcto en:

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

pero esto podría ser algo confuso para el lector, por lo que a veces resulta más claro escribir:

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

El menos unario invierte el signo de los datos. El más unario proporciona una simetría con respecto al menos unario, aunque no tiene ningún efecto.

Autoincremento y autodecremento

Java, como C, dispone de una serie de abreviaturas. Esas abreviaturas pueden hacer que resulte mucho más fácil escribir el código; en cuanto a la lectura, pueden simplificarla o complicarla.

Dos de las abreviaturas más utilizadas son los operadores de incremento y decremento (a menudo denominados operadores de autoincremento y autodecremento). El operador de decremento es -- y significa "disminuir en una unidad". El operador de incremento es ++ y significa "aumentar en una unidad". Por ejemplo, si **a** es un valor **int**, la expresión **++a** es equivalente a **(a = a + 1)**. Los operadores de incremento y decremento no sólo modifican la variable, sino que también generan el valor de la misma como resultado.

Hay dos versiones de cada tipo de operador, a menudo denominadas *prefija* y *postfija*. *Pre-incremento* significa que el operador ++ aparece antes de la variable, mientras que *post-incremento* significa que el operador ++ aparece detrás de la variable. De forma similar, *pre-decremento* quiere decir que el operador −− aparece antes de la variable y *post-decremento* significa que el operador −− aparece detrás de la variable. Para el pre-incremento y el pre-decremento (es decir, ++a o −−a), se realiza primero la operación y luego se genera el valor. Para el post-incremento y el post-decremento (es decir, a++ o −−a), se genera primero el valor y luego se realiza la operación. Por ejemplo:

---

[1] El número 47 se utilizaba como "número mágico" en una universidad en la que estudié, y desde entonces lo utilizo.

```
//: operators/AutoInc.java
// Ilustra los operadores ++ y --.
import static net.mindview.util.Print.*;

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    print("i : " + i);
    print("++i : " + ++i); // Pre-incremento
    print("i++ : " + i++); // Post-incremento
    print("i : " + i);
    print("--i : " + --i); // Pre-decremento
    print("i-- : " + i--); // Post-decremento
    print("i : " + i);
  }
} /* Output:
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
*///:~
```

Puede ver que para la forma prefija, se obtiene el valor después de realizada la operación, mientras que para la forma postfija, se obtiene el valor antes de que la operación se realice. Estos son los únicos operadores, además de los de asignación, que tienen efectos colaterales. Modifican el operando en lugar de simplemente utilizar su valor.

El operador de incremento es, precisamente, una de las explicaciones del por qué del nombre C++, que quiere decir "un paso más allá de C". En una de las primeras presentaciones realizadas acerca de Java, Bill Joy (uno de los creadores de Java), dijo que "Java=C++--" (C más más menos menos), para sugerir que Java es C++ pero sin todas las complejidades innecesarias, por lo que resulta un lenguaje mucho más simple. A medida que vaya avanzando a lo largo del libro, podrá ver que muchas partes son más simples, aunque en algunos otros aspectos Java no resulta mucho más sencillo que C++.

Operadores relacionales

Los operadores relacionales generan un resultado de tipo **boolean**. Evalúan la relación existente entre los valores de los operandos. Una expresión relacional produce el valor **true** si la relación es cierta y **false** si no es cierta. Los operadores relacionales son: menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), equivalente (==) y no equivalente (!=). La equivalencia y la no equivalencia funcionan con todas las primitivas, pero las otras comparaciones no funcionan con el tipo **boolean**. Puesto que los valores **boolean** sólo pueden ser **true** o **false**, las relaciones "mayor que" y "menor que" no tienen sentido.

Comprobación de la equivalencia de objetos

Los operadores relacionales == y != también funcionan con todos los objetos, pero su significado suele confundir a los que comienzan a programar en Java. He aquí un ejemplo:

```
//: operators/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);
  }
```

```
} /* Output:
false
true
*///:~
```

La instrucción **System.out.println(n1 == n2)** imprimirá el resultado de la comparación booleana que contiene. Parece que la salida debería ser "true" y luego "false", dado que ambos objetos **Integer** son iguales, aunque el *contenido* de los objetos son los mismos, las referencias no son iguales. Los operadores == y **!=** comparan referencias a objetos, por lo que la salida realmente es "false" y luego "true". Naturalmente, este comportamiento suele sorprender al principio a los programadores.

¿Qué pasa si queremos comparar si el contenido de los objetos es equivalente? Entonces debemos utilizar el método especial **equals( )** disponible para todos los objetos (no para las primitivas, que funcionan adecuadamente con == y **!=**). He aquí la forma en que se emplea:

```
//: operators/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));
  }
} /* Output:
true
*///:~
```

El resultado es ahora el que esperábamos. Aunque, en realidad, las cosas no son tan sencillas. Si creamos nuestra propia clase, como por ejemplo:

```
//: operators/EqualsMethod2.java
// equals() predeterminado no compara los contenidos.

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));
  }
} /* Output:
false
*///:~
```

los resultados vuelven a confundirnos. El resultado es **false**. Esto se debe a que el comportamiento predeterminado de **equals( )** consiste en comparar referencias. Por tanto, a menos que sustituyamos **equals( )** en nuestra nueva clase, no obtendremos el comportamiento deseado. Lamentablemente, no vamos a aprender a sustituir unos métodos por otros hasta el capítulo dedicado a la *Reutilización de clases*, y no veremos cuál es la forma adecuada de definir **equals( )** hasta el Capítulo 17, *Análisis detallado de los contenedores*, pero mientras tanto tener en cuenta el comportamiento de **equals( )** nos puede ahorrar algunos quebraderos de cabeza.

La mayoría de las clases de biblioteca Java implementan **equals( )** de modo que compare el contenido de los objetos, en lugar de sus referencias.

Ejercicio 5: (2) Cree una clase denominada **Dog** (perro) que contenga dos objetos **String**: **name** (nombre) y **says** (ladrido). En **main( )**, cree dos objetos perro con los nombres "spot" (que ladre diciendo "Ruff!") y "scruffy" (que ladre diciendo, "Wurf!"). Después, muestre sus nombres y el sonido que hacen al ladrar.

Ejercicio 6: (3) Continuando con el Ejercicio 5, cree una nueva referencia **Dog** y asígnela al objeto de nombre "spot". Realice una comparación utilizando == y **equals( )** para todas las referencias.

Operadores lógicos

Cada uno de los operadores lógicos AND (**&&**), OR (||) y NOT (**!**) produce un valor **boolean** igual a **true** o **false** basándose en la relación lógica de sus argumentos. Este ejemplo utiliza los operadores relacionales y lógicos:

```
//: operators/Bool.java
// Operadores relacionales y lógicos.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);
    print("i = " + i);
    print("j = " + j);
    print("i > j is " + (i > j));
    print("i < j is " + (i < j));
    print("i >= j is " + (i >= j));
    print("i <= j is " + (i <= j));
    print("i == j is " + (i == j));
    print("i != j is " + (i != j));
    // Tratar int como boolean no es legal en Java:
//! print("i && j is " + (i && j));
//! print("i || j is " + (i || j));
//! print("!i is " + !i);
    print("(i < 10) && (j < 10) is "
        + ((i < 10) && (j < 10)) );
    print("(i < 10) || (j < 10) is "
        + ((i < 10) || (j < 10)) );
  }
} /* Output:
i = 58
j = 55
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 false
*///:~
```

Sólo podemos aplicar AND, OR o NOT a valores de tipo **boolean**. No podemos emplear un valor que no sea **boolean** como si fuera un valor booleano en una expresión lógica como a diferencia de lo que sucede en C y C++. Puede ver en el ejemplo los intentos fallidos de realizar esto, desactivados mediante marcas de comentarios '//!' (esta sintaxis de comentarios permite la eliminación automática de comentarios para facilitar las pruebas). Sin embargo, las expresiones subsiguientes generan valores de tipo **boolean** utilizando comparaciones relacionales y a continuación realizan operaciones lógicas con los resultados.

Observe que un valor **boolean** se convierte automáticamente a una forma de tipo texto apropiada cuando se les usa en lugares donde lo que se espera es un valor de tipo **String**.

Puede reemplazar la definición de valores **int** en el programa anterior por cualquier otro tipo de dato primitivo excepto **boolean**. Sin embargo, teniendo en cuenta que la comparación de números en coma flotante es muy estricta, un número que difiera de cualquier otro, aunque que sea en un valor pequeñísimo seguirá siendo distinto. Asimismo, cualquier número situado por encima de cero, aunque sea pequeñísimo, seguirá siendo distinto de cero.

Ejercicio 7: (3) Escriba un programa que simule el proceso de lanzar una moneda al aire.

Cortocircuitos

Al tratar con operadores lógicos, nos encontramos con un fenómeno denominado "cortocircuito". Esto quiere decir que la expresión se evaluará únicamente *hasta* que la veracidad o la falsedad de la expresión completa pueda ser determinada de forma no ambigua. Como resultado, puede ser que las últimas partes de una expresión lógica no lleguen a evaluarse. He aquí un ejemplo que ilustra este fenómeno de cortocircuito.

```
//: operators/ShortCircuit.java
// Ilustra el comportamiento de cortocircuito
// al evaluar los operadores lógicos.
import static net.mindview.util.Print.*;

public class ShortCircuit {
  static boolean test1(int val) {
    print("test1(" + val + ")");
    print("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    print("test2(" + val + ")");
    print("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    print("test3(" + val + ")");
    print("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    boolean b = test1(0) && test2(2) && test3(2);
    print("expression is " + b);
  }
} /* Output:
test1(0)
result: true
test2(2)
result: false
expression is false
*///:~
```

Cada una de las comprobaciones realiza una comparación con el argumento y devuelve **true** o **false**. También imprime la información necesaria para que veamos que está siendo invocada. Las pruebas se utilizan en la expresión:

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

Lo natural sería pensar que las tres pruebas llegan a ejecutarse, pero la salida muestra que no es así. La primera de las pruebas produce un resultado **true**, por lo que continúa con la evaluación de la expresión. Sin embargo, la segunda prueba produce un resultado **false**. Dado que esto quiere decir que la expresión completa debe ser **false**, ¿por qué continuar con la evaluación del resto de la expresión? Esa evaluación podría consumir una cantidad considerable de recursos. La razón de que se produzca este tipo de cortocircuito es, de hecho, que podemos mejorar la velocidad del programa si no es necesario evaluar todas las partes de una expresión lógica.

Literales

Normalmente, cuando insertamos un valor literal en un programa, el compilador sabe exactamente qué tipo asignarle. En ocasiones, sin embargo, puede que ese tipo sea ambiguo. Cuando esto sucede, hay que guiar al compilador añadiendo cierta información adicional en la forma de caracteres asociados con el valor literal. El código siguiente muestra estos caracteres:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;

public class Literals {
  public static void main(String[] args) {
    int i1 = 0x2f; // Hexadecimal (minúscula)
    print("i1: " + Integer.toBinaryString(i1));
    int i2 = 0X2F; // Hexadecimal (mayúscula)
    print("i2: " + Integer.toBinaryString(i2));
    int i3 = 0177; // Octal (cero inicial)
    print("i3: " + Integer.toBinaryString(i3));
    char c = 0xffff; // máximo valor hex para char
    print("c: " + Integer.toBinaryString(c));
    byte b = 0x7f; // máximo valor hex para byte
    print("b: " + Integer.toBinaryString(b));
    short s = 0x7fff; // máximo valor hex para short
    print("s: " + Integer.toBinaryString(s));
    long n1 = 200L; // sufijo long
    long n2 = 200l; // sufijo long (pero puede ser confuso)
    long n3 = 200;
    float f1 = 1;
    float f2 = 1F; // sufijo float
    float f3 = 1f; // sufijo float
    double d1 = 1d; // sufijo double
    double d2 = 1D; // sufijo double
    // (Hex y Octal también funcionan con long)
  }
} /* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 111111111111111
*///:~
```

Un carácter situado al final de un valor literal permite establecer su tipo. La **L** mayúscula o minúscula significa **long** (sin embargo, utilizar una **l** minúscula es confuso, porque puede parecerse al número uno). Una **F** mayúscula o minúscula significa **float**. Una **D** mayúscula o minúscula significa **double**.

Los valores hexadecimales (base 16), que funcionan con todos los tipos de datos enteros, se denotan mediante el prefijo **0x** o **0X** seguido de **0-9** o **a-f** en mayúscula o minúscula. Si se intenta inicializar una variable con un valor mayor que el máximo que puede contener (independientemente de la forma numérica del valor), el compilador dará un mensaje de error. Observe, en el código anterior, los valores hexadecimales máximos permitidos para **char**, **byte** y **short**. Si nos excedemos de éstos, el compilador transformará automáticamente el valor a **int** y nos dirá que necesitamos una *proyección hacia abajo* para la asignación (definiremos las proyecciones posteriormente en el capítulo). De esta forma, sabremos que nos hemos pasado del límite permitido.

Los valores octales (base 8) se denotan incluyendo un cero inicial en el número y utilizando sólo los dígitos 0-7.

No existe ninguna representación literal para los números binarios en C, C++ o Java. Sin embargo, a la hora de trabajar con notación hexadecimal y octal, a veces resulta útil mostrar la forma binaria de los resultados. Podemos hacer esto fácilmen-

te con los métodos **static toBinaryString( )** de las clases **Integer** y **Long**. Observe que, cuando se pasan tipos más pequeños a **Integer.toBinaryString( )**, el tipo se convierte automáticamente a **int**.

Ejercicio 8: (2) Demuestre que las notaciones hexadecimal y octal funcionan con los valores long. Utilice **Long.toBinaryString( )** para mostrar los resultados.

Notación exponencial

Los exponentes utilizan una notación que a mí personalmente me resulta extraña:

```
//: operators/Exponents.java
// "e" significa "10 elevado a".

public class Exponents {
  public static void main(String[] args) {
    // 'e' en mayúscula o minúscula funcionan igual:
    float expFloat = 1.39e-43f;
    expFloat = 1.39E-43f;
    System.out.println(expFloat);
    double expDouble = 47e47d; // 'd' es opcional
    double expDouble2 = 47e47; // automáticamente double
    System.out.println(expDouble);
  }
} /* Output:
1.39E-43
4.7E48
*///:~
```

En el campo de las ciencias y de la ingeniería, 'e' hace referencia a la base de los logaritmos naturales, que es aproximadamente 2,718 (en Java hay disponible un valor **double** más preciso, que es **Math.E**). Esto se usa en expresiones de exponenciación, como por ejemplo $1.39 \times e^{-43}$, que significa 1.39×2.718^{-43}. Sin embargo, cuando se inventó el lenguaje de programación FORTRAN, decidieron que **e** significaría "diez elevado a", lo cual es una decisión extraña, ya que FORTRAN fue diseñado para campos de la ciencia y de la ingeniería, así que cabría esperar que sus diseñadores tendrían en cuenta lo confuso de introducir esa ambigüedad[2]. En cualquier caso, esta costumbre fue también introducida en C, C++ y ahora en Java. Por tanto, si el lector está habituado a pensar en **e** como en la base de los logaritmos naturales, tendrá que hacer una traducción mental cuando vea una expresión como **1.39 e-43f** en Java; ya que quiere decir 1.39×10^{-43}.

Observe que no es necesario utilizar el carácter sufijo cuando el compilador puede deducir el tipo apropiado. Con:

```
long n3 = 200;
```

no existe ninguna ambigüedad, por lo que una **L** después del 200 sería superfluo. Sin embargo, con:

```
float f4 = 1e-43f; // 10 elevado a
```

el compilador normalmente considera los números exponenciales como de tipo **double**, por lo que sin la **f** final, nos daría un error en el que nos informaría de que hay que usar una proyección para convertir el valor **double** a **float**.

Ejercicio 9: (1) Visualice los números más grande y más pequeño que se pueden representar con la notación exponencial en el tipo **float** y en el tipo **double**.

---

[2] John Kirkham escribe: "Comencé a escribir programas informáticos en 1962 en FORTRAN II en un IBM 1620. Por aquel entonces y a lo largo de las décadas de 1960 y 1970, FORTRAN era un lenguaje donde todo se escribía en mayúsculas. Probablemente, la razón era que muchos de los dispositivos de entrada eran antiguas unidades de teletipo que utilizaban el código Baudot de cinco bits que no disponía de minúsculas. La "E" en la notación exponencial era también mayúscula y no se confundía nunca con la base de los logaritmos naturales "e" que siempre se escribe en minúscula. La "E" simplemente quería decir exponencial, que era la base para el sistema de numeración que se estaba utilizando, que normalmente era 10. En aquella época, los programadores también empleaban los números octales. Aunque nunca vi que nadie lo utilizara, si yo hubiera visto un número octal en notación exponencial, habría considerado que estaba en base 8. La primera vez que vi un exponencial utilizando una "e" fue a finales de la década de 1970 y también a mí me pareció confuso; el problema surgió cuando empezaron a utilizarse minúsculas en FORTAN, no al principio. De hecho, disponíamos de funciones que podían usarse cuando se quisiera emplear la base de los logaritmos naturales, pero todas esas funciones se escribían en mayúsculas.

Operadores bit a bit

Los operadores bit a bit permiten manipular bits individuales en un tipo de datos entero primitivo. Para generar el resultado, los operadores bit a bit realizan operaciones de álgebra booleana con los bits correspondientes de los dos argumentos.

Los operadores bit a bit proceden de la orientación a bajo nivel del lenguaje C, en el que a menudo se manipula el hardware directamente y es preciso configurar los bits de los registros hardware. Java se diseñó originalmente para integrarlo en codificadores para televisión, por lo que esta orientación a bajo nivel seguía teniendo sentido. Sin embargo, lo más probable es que no utilicemos demasiado esos operadores bit a bit en nuestros programas.

El operador bit a bit AND (**&**) genera un uno en el bit de salida si ambos bits de entrada son iguales a uno; en caso contrario, genera un cero. El operador OR bit a bit (|) genera un uno en el bit de salida si alguno de los bits de entrada es un uno y genera cero sólo si ambos bits de entrada son cero. El operador bit a bit EXCLUSIVE OR o XOR (^) genera un uno en el bit de salida si uno de los dos bits de entrada es un uno pero no ambos. El operador bit a bit NOT (~, también denominado operador de *complemento a uno*) es un operador unario, que sólo admite un argumento (todos los demás operadores bit a bit son operadores binarios). El operador bit a bit NOT genera el opuesto al bit de entrada, es uno si el bit de entrada es cero y es cero si el bit de entrada es uno.

Los operadores bit a bit y los operadores lógicos utilizan los mismos caracteres, por lo que resulta útil recurrir a un truco mnemónico para recordar cuál es el significado correcto. Como los bits son "pequeños" sólo se utiliza un carácter en los operadores bit a bit.

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

El tipo **boolean** se trata como un valor de un único bit, por lo que es algo distinto de los otros tipos primitivos. Se puede realizar una operación AND, OR o XOR bit a bit, pero no se puede realizar una operación NOT bit a bit (presumiblemente, para evitar la confusión con la operación lógica NOT). Para los valores booleanos, los operadores bit a bit tienen el mismo efecto que los operadores lógicos, salvo porque no se aplica la regla de cortocircuito. Asimismo, las operaciones bit a bit con valores booleanos incluyen un operador lógico XOR que no forma parte de la lista de operadores "lógicos". No se pueden emplear valores booleanos en expresiones de desplazamiento, las cuales vamos a describir a continuación.

Ejercicio 10: (3) Escriba un programa con dos valores constantes, uno en el que haya unos y ceros binarios alternados, con un cero en el dígito menos significativo, y el segundo con un valor también alternado pero con un uno en el dígito menos significativo (consejo: lo más fácil es usar constantes hexadecimales para esto). Tome estos dos valores y combínelos de todas las formas posibles utilizando los operadores bit a bit, y visualice los resultados utilizando **Integer.toBinaryString( )**.

Operadores de desplazamiento

Los operadores de desplazamiento también sirven para manipular bits. Sólo se les puede utilizar con tipos primitivos enteros. El operador de desplazamiento a la izquierda (<<) genera como resultado el operando situado a la izquierda del operador después de desplazarlo hacia la izquierda el número de bits especificado a la derecha del operador (insertando ceros en los bits de menor peso). El operador de desplazamiento a la derecha con signo (>>) genera como resultado el operando situado a la izquierda del operador después de desplazarlo hacia la derecha el número de bits especificado a la derecha del operador. El desplazamiento a la derecha con signo >> utiliza lo que se denomina *extensión de signo:* si el valor es positivo, se insertan ceros en los bits de mayor peso; si el valor es negativo, se insertan unos en los bits de mayor peso. Java ha añadido también un desplazamiento a la derecha sin signo >>>, que utiliza lo que denomina *extensión con ceros:* independientemente del signo, se insertan ceros en los bits de mayor peso. Este operador no existe ni en C ni C++.

Si se desplaza un valor de tipo **char**, **byte** o **short**, será convertido a **int** antes de que el desplazamiento tenga lugar y el resultado será de tipo **int**. Sólo se utilizarán los bits de menor peso del lado derecho; esto evita que se realicen desplazamientos con un número de posiciones superior al número de bits de un valor **int**. Si se está operando con un valor **long**, se obtendrá un resultado de tipo **long** y sólo se emplearán los seis bits de menor peso del lado derecho, para así no poder desplazar más posiciones que el número de bits de un valor **long**.

Los desplazamientos se pueden combinar con el signo igual (<<= o >>= o >>>=). El lvalor se sustituye por el lvalor desplazado de acuerdo con lo que el rvalor marque. Existe un problema, sin embargo, con el desplazamiento a la derecha sin

signo combinado con la asignación. Si se usa con valores de tipo **byte** o **short**, no se obtienen los resultados correctos. En lugar de ello, se transforman a **int** y luego se desplazan a la derecha, pero a continuación se truncan al volver a asignar los valores a sus variables, por lo que se obtiene **−1** en esos casos. El siguiente ejemplo ilustra esta situación:

```
//: operators/URShift.java
// Prueba del desplazamiento a la derecha sin signo.
import static net.mindview.util.Print.*;

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    print(Integer.toBinaryString(i));
    i >>>= 10;
    print(Integer.toBinaryString(i));
    long l = -1;
    print(Long.toBinaryString(l));
    l >>>= 10;
    print(Long.toBinaryString(l));
    short s = -1;
    print(Integer.toBinaryString(s));
    s >>>= 10;
    print(Integer.toBinaryString(s));
    byte b = -1;
    print(Integer.toBinaryString(b));
    b >>>= 10;
    print(Integer.toBinaryString(b));
    b = -1;
    print(Integer.toBinaryString(b));
    print(Integer.toBinaryString(b>>>10));
  }
} /* Output:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*///:~
```

En el último desplazamiento, el valor resultante no se asigna de nuevo a **b**, sino que se imprime directamente, obteniéndose el comportamiento correcto.

He aquí un ejemplo que ilustra todos los operadores para el manejo de bits:

```
//: operators/BitManipulation.java
// Uso de los operadores bit a bit.
import java.util.*;
import static net.mindview.util.Print.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt();
    int j = rand.nextInt();
    printBinaryInt("-1", -1);
    printBinaryInt("+1", +1);
```

```
    int maxpos = 2147483647;
    printBinaryInt("maxpos", maxpos);
    int maxneg = -2147483648;
    printBinaryInt("maxneg", maxneg);
    printBinaryInt("i", i);
    printBinaryInt("~i", ~i);
    printBinaryInt("-i", -i);
    printBinaryInt("j", j);
    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
    printBinaryInt("i ^ j", i ^ j);
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(-i) >>> 5", (-i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("maxpos", ll);
    long lln = -9223372036854775808L;
    printBinaryLong("maxneg", lln);
    printBinaryLong("l", l);
    printBinaryLong("~l", ~l);
    printBinaryLong("-l", -l);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(-l) >>> 5", (-l) >>> 5);
  }
  static void printBinaryInt(String s, int i) {
    print(s + ", int: " + i + ", binary:\n    " +
      Integer.toBinaryString(i));
  }
  static void printBinaryLong(String s, long l) {
    print(s + ", long: " + l + ", binary:\n    " +
      Long.toBinaryString(l));
  }
} /* Output:
-1, int: -1, binary:
    11111111111111111111111111111111
+1, int: 1, binary:
    1
maxpos, int: 2147483647, binary:
    1111111111111111111111111111111
maxneg, int: -2147483648, binary:
    10000000000000000000000000000000
i, int: -1172028779, binary:
    10111010001001000100001010010101
-i, int: 1172028778, binary:
    01000101110110111011110101101010
```

```
-i, int: 1172028779, binary:
   10001011101101111011110101101011
j, int: 1717241110, binary:
   11001100101101100000010100010110
i & j, int: 570425364, binary:
   10001000000000000000000000010100
i | j, int: -25213033, binary:
   11111110011111111010000111010111
i ^ j, int: -595638397, binary:
   11011100011111111010000111010011
i << 5, int: 1149784736, binary:
   10001001000100000101001010100000
i >> 5, int: -36625900, binary:
   11111101111010001001000010000010100
(~i) >> 5, int: 36625899, binary:
   10001011101101110111101011
i >>> 5, int: 97591828, binary:
   10111010001001000010000010100
(-i) >>> 5, int: 36625899, binary:
   10001011101101110111101011
...
*///:~
```

Los dos métodos del final, **printBinaryInt( )** y **printBinaryLong( )**, toman un valor **int** o **long**, respectivamente, y lo imprimen en formato binario junto con una cadena de caracteres descriptiva. Además de demostrar el efecto de todos los operadores bit a bit para valores **int** y **long**, este ejemplo también muestra los valores mínimo, máximo, +1 y −1 para **int** y **long** para que vea el aspecto que tienen. Observe que el bit más alto representa el signo: 0 significa positivo y 1 significa negativo. En el ejemplo se muestra la salida de la parte correspondiente a los valores **int**.

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

Ejercicio 11: (3) Comience con un número que tenga un uno binario en la posición más significativa (consejo: utilice una constante hexadecimal). Emplee el operador de desplazamiento a la derecha con signo, desplace el valor a través de todas sus posiciones binarias, mostrando cada vez el resultado con **Integer.toBinaryString( )**.

Ejercicio 12: (3) Comience con un número cuyos dígitos binarios sean todos iguales a uno. A continuación desplácelo a la izquierda y utilice el operador de desplazamiento a la derecha sin signo para desplazarlo a través de todas sus posiciones binarias, visualizando los resultados con **Integer.toBinaryString( )**.

Ejercicio 13: (1) Escriba un método que muestre valores **char** en formato binario. Ejecútelo utilizando varios caracteres diferentes.

Operador ternario if-else

El operador *ternario*, también llamado operador *condicional* resulta inusual porque tiene tres operandos. Realmente se trata de un operador, porque genera un valor a diferencia de la instrucción **if-else** ordinaria, que veremos en la siguiente sección del capítulo. La expresión tiene la forma:

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

Si *exp-booleana* se evalúa como **true**, se evalúa *valor0* y el resultado será el valor generado por el operador. Si *exp-booleana* es **false**, se evalúa *valor1* y su resultado pasará a ser el valor generado por el operador.

Por supuesto, podría utilizarse una instrucción **if-else** ordinaria (que se describe más adelante), pero el operador ternario es mucho más compacto. Aunque C (donde se originó este operador) se enorgullece de ser un lenguaje compacto, y el operador ternario puede que se haya introducido en parte por razones de eficiencia, conviene tener cuidado a la hora de emplearlo de forma cotidiana, ya que el código resultante puede llegar a ser poco legible.

El operador condicional es diferente de **if-else** porque genera un valor. He aquí un ejemplo donde se comparan ambas estructuras:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
  static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
  }
  static int standardIfElse(int i) {
    if(i < 10)
      return i * 100;
    else
      return i * 10;
  }
  public static void main(String[] args) {
    print(ternary(9));
    print(ternary(10));
    print(standardIfElse(9));
    print(standardIfElse(10));
  }
} /* Output:
900
100
900
100
*///:~
```

Puede ver que el código de **ternary( )** es más compacto de lo que sería si no dispusiéramos del operador ternario; la versión sin operador ternario se encuentra en **standardIfElse( )**. Sin embargo, **standardIfElse( )** es más fácil de comprender y además exige escribir muchos caracteres más. Así que asegúrese de ponderar bien las razones a la hora de elegir el operador ternario; normalmente, puede convenir utilizarlo cuando se quiera configurar una variable con uno de dos valores posibles.

Operadores + y += para String

Existe un uso especial de un operador en Java: los operadores + y += pueden usarse para concatenar cadenas, como ya hemos visto. Parece un uso bastante natural de estos operadores, aún cuando no encaje demasiado bien con la forma tradicional en que dichos operadores se emplean.

Esta capacidad le pareció adecuada a los diseñadores de C++, por lo que se añadió a C++ un mecanismo de *sobrecarga de operadores* para que los programadores C++ pudieran añadir nuevos significados casi a cualquier operador. Lamentablemente, la sobrecarga de operadores combinada con alguna de las otras restricciones de C++, resulta una característica excesivamente complicada para que los programadores la incluyan en el diseño de sus clases. Aunque la sobrecarga de operadores habría sido mucho más fácil de implementar en Java de lo que lo era en C++ (como se ha demostrado en el lenguaje C#, que *sí* que dispone de un sencillo mecanismo de sobrecarga de operadores), se consideraba que esta característica seguía siendo demasiado compleja, por lo que los programadores Java no pueden implementar sus propios operadores sobrecargados, a diferencia de los programadores de C++ y C#.

El uso de los operadores para valores **String** presenta ciertos comportamientos interesantes. Si una expresión comienza con un valor **String**, todos los operandos que siguen también tendrán que ser cadenas de caracteres (recuerde que el compilador transforma automáticamente a **String** toda secuencia de caracteres encerrada entre comillas dobles).

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
  public static void main(String[] args) {
    int x = 0, y = 1, z = 2;
    String s = "x, y, z ";
    print(s + x + y + z);
```

```
    print(x + " " + s); // Convierte x a String
    s += "(summed) = "; // Operador de concatenación
    print(s + (x + y + z));
    print("" + x); // Abreviatura de Integer.toString()
  }
} /* Output:
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
*///:~
```

Observe que la salida de la primera instrucción de impresión es '**o12**' en lugar de sólo '**3**', que es lo que obtendría si se estuvieran sumando los valores enteros. Esto es porque el compilador Java convierte **x**, **y** y **z** a su representación **String** y concatena esas cadenas de caracteres, en lugar de efectuar primero la suma. La segunda instrucción de impresión convierte la variable inicial a **String**, por lo que la conversión a cadena no depende de qué es lo que haya primero. Por último, podemos ver el uso del operador += para añadir una cadena de caracteres a **s**, y el uso de paréntesis para controlar el orden de evaluación de la expresión, de modo que los valores enteros se sumen realmente antes de la visualización.

Observe el último ejemplo de **main( )**: en ocasiones, se encontrará en los programas un valor **String** vacío seguido de + y una primitiva, como forma de realizar la conversión sin necesidad de invocar el método explícito más engorroso, (**Integer.toString( )**, en este caso).

Errores comunes a la hora de utilizar operadores

Uno de los errores que se pueden producir a la hora de emplear operadores es el de tratar de no incluir los paréntesis cuando no se está del todo seguro acerca de la forma que se evaluará una expresión. Esto, que vale para muchos lenguajes también es cierto para Java.

Un error extremadamente común en C y C++ sería el siguiente:

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

El programador estaba intentando, claramente, comprobar la equivalencia (==) en lugar de hacer una asignación. En C y C++ el resultado de esta asignación será siempre **true** si **y** es distinto de cero, por lo que probablemente se produzca un bucle infinito. En Java, el resultado de esta expresión no es de tipo **boolean**, pero el compilador espera un valor **boolean** y no realizará ninguna conversión a partir de un valor **int**, por lo que dará un error en tiempo de compilación y detectará el problema antes de que ni siquiera intentemos ejecutar el programa. Por tanto, este error nunca puede producirse en Java (la única posibilidad de que no se tenga un error de tiempo de compilación, es cuando **x** e **y** son de tipo **boolean**, en cuyo caso **x = y** es una expresión legal, aunque en el ejemplo anterior probablemente su uso se deba a un error).

Un problema similar en C y C++ consiste en utilizar los operadores bit a bit AND y OR en lugar de las versiones lógicas. Los operadores bit a bit AND y OR utilizan uno de los caracteres (**&** o |) mientras que los operadores lógicos AND y OR utilizan dos (**&&** y ||). Al igual que con = y ==, resulta fácil confundirse y escribir sólo uno de los caracteres en lugar de dos. En Java, el compilador vuelve a evitar este tipo de error, porque no permite emplear un determinado tipo de datos en un lugar donde no sea correcto hacerlo.

Operadores de proyección

La palabra *proyección* (*cast*) hace referencia a la conversión explícita de datos de un tipo a otro. Java cambiará automáticamente un tipo de datos a otro cada vez que sea apropiado. Por ejemplo, si se asigna un valor entero a una variable de coma flotante, el compilador convertirá automáticamente el valor **int** a **float**. El mecanismo de conversión nos permite realizar esta conversión de manera explícita, o incluso forzarla en situaciones donde normalmente no tendría lugar.

Para realizar una proyección, coloque el tipo de datos deseado entre paréntesis a la izquierda del valor que haya que convertir, como en el siguiente ejemplo:

```
//: operators/Casting.java

public class Casting {
  public static void main(String[] args) {
    int i = 200;
    long lng = (long)i;
    lng = i; // "Ensanchamiento," por  lo que no se requiere conversión
    long lng2 = (long)200;
    lng2 = 200;
    // Una "conversión de estrechamiento ":
    i = (int)lng2; // Proyección requerida
  }
} ///:~
```

Como podemos ver, resulta posible aplicar una proyección de tipo tanto a los valores numéricos como a las variables. Observe que se pueden también introducir proyecciones superfluas, por ejemplo, el compilador promocionará automáticamente un valor **int** a **long** cuando sea necesario. Sin embargo, podemos utilizar esas proyecciones superfluas con el fin de resaltar la operación o de clarificar el código. En otras situaciones, puede que la proyección de tipo sea esencial para que el código llegue a compilarse.

En C y C++, las operaciones de proyección de tipos pueden provocar algunos dolores de cabeza. En Java, la proyección de tipos resulta siempre segura, con la excepción de que, cuando se realiza una de las denominadas *conversiones de estrechamiento* (es decir, cuando se pasa de un tipo de datos que puede albergar más información a otro que no permite albergar tanta), se corre el riesgo de perder información. En estos casos, el compilador nos obliga a emplear una proyección, como diciéndonos: "Esta conversión puede ser peligrosa, si quieres que lo haga de todos modos, haz que esa proyección sea explícita". Con una *conversión de ensanchamiento*, no hace falta una proyección explícita, porque el nuevo tipo permitirá albergar con creces la información del tipo anterior, de modo que nunca se puede perder información.

Java permite proyectar cualquier tipo primitivo a cualquier otro, excepto en el caso de **boolean**, que no permite efectuar ningún tipo de proyección. Los tipos de clase tampoco permiten efectuar proyecciones: para convertir uno de estos tipos en otro, deben existir métodos especiales (posteriormente, veremos que los objetos pueden proyectarse dentro de una *familia* de tipos; un **Olmo** puede proyectarse sobre un **Árbol** y viceversa, pero no sobre un tipo externo como pueda ser **Roca**).

Truncamiento y redondeo

Cuando se realizan conversiones de estrechamiento, es necesario prestar atención a los problemas de truncamiento y redondeo. Por ejemplo, si efectuamos una proyección de un valor de coma flotante sobre un valor entero, ¿qué es lo que haría Java? Por ejemplo, si tenemos el valor 29,7 y lo proyectamos sobre un **int**, ¿el valor resultante será 30 o 29? La respuesta a esta pregunta puede verse en el siguiente ejemplo:

```
//: operators/CastingNumbers.java
// ¿Qué ocurre cuando se proyecta un valor float
// o double sobre un valor entero?
import static net.mindview.util.Print.*;

public class CastingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    print("(int)above: " + (int)above);
    print("(int)below: " + (int)below);
    print("(int)fabove: " + (int)fabove);
    print("(int)fbelow: " + (int)fbelow);
  }
} /* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*///:~
```

Así que la respuesta es que al efectuar la proyección de **float** o **double** a un valor entero, siempre se trunca el correspondiente número. Si quisiéramos que el resultado se redondeara habría que utilizar los métodos **round( )** de **java.lang.Math**:

```
//: operators/RoundingNumbers.java
// Redondeo de valores float y double.
import static net.mindview.util.Print.*;

public class RoundingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    print("Math.round(above): " + Math.round(above));
    print("Math.round(below): " + Math.round(below));
    print("Math.round(fabove): " + Math.round(fabove));
    print("Math.round(fbelow): " + Math.round(fbelow));
  }
} /* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*///:~
```

Puesto que **round( )** es parte de **java.lang**, no hace falta ninguna instrucción adicional de importación para utilizarlo.

Promoción

Cuando comience a programar en Java, descubrirá que si hace operaciones matemáticas o bit a bit con tipos de datos primitivos más pequeños que **int** (es decir, **char**, **byte** o **short**), dichos valores serán promocionados a **int** antes de realizar las operaciones, y el valor resultante será de tipo **int**. Por tanto, si se quiere asignar el resultado de nuevo al tipo más pequeño, es necesario emplear una proyección (y, como estamos realizando una asignación a un tipo de menor tamaño, perderemos información). En general, el tipo de datos de mayor tamaño dentro de una expresión es el que determina el tamaño del resultado de esa expresión, si se multiplica un valor **float** por otro **double**, el resultado será **double**; si se suman un valor **int** y uno **long**, el resultado será **long**.

Java no tiene operador "sizeof"

En C y C++, el operador **sizeof( )** nos dice el número de bytes asignado a un elemento de datos. La razón más importante para el uso de **sizeof( )** en C y C++ es la portabilidad. Los diferentes tipos de datos pueden tener diferentes tamaños en distintas máquinas, por lo que el programador debe averiguar el tamaño de esos tipos a la hora de realizar operaciones que sean sensibles al tamaño. Por ejemplo, una computadora puede almacenar los enteros en 32 bits, mientras que otras podrían almacenarlos en 16 bits. Los programas podrían, así, almacenar valores de mayor tamaño en variables de tipo entero en la primera máquina. Como puede imaginarse, la portabilidad es un verdadero 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 el mismo tamaño en todas las máquinas. No es necesario que tengamos en cuenta la portabilidad en este nivel, ya que esa portabilidad forma parte del propio diseño del lenguaje.

Compedio de operadores

El siguiente ejemplo muestra qué tipos de datos primitivos pueden utilizarse con determinados operadores concretos. Básicamente, se trata del mismo ejemplo repetido una y otra vez pero empleando diferentes tipos de datos primitivos. El archivo se compilará sin errores porque las líneas que los incluyen están desactivas mediante comentarios de tipo **//!**.

```
//: operators/AllOps.java
// Comprueba todos los operadores con todos los tipos de datos primitivos
```

```
// para mostrar cuáles son aceptables por el compilador Java.

public class AllOps {
  // Para 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 bit a 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;
    // Proyección:
    //! 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 bit a 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;
    // Proyección:
    //! boolean bl = (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);
    // Operadores bit a 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;
    // Proyección:
    //! boolean bl = (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 bit a 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;
    // Proyección:
    //! boolean bl = (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 bit a 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;
  // Proyección:
  //! boolean bl = (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 bit a 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;
    // Proyección:
    //! boolean bl = (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 bit a 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;
    // Proyección:
    //! boolean bl = (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 bit a 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;
    // Proyección:
    //! boolean bl = (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;
  }
} ///:~
```

Observe que **boolean** es bastante limitado. A una variable de este tipo se le pueden asignar los valores **true** y **false**, y se puede comprobar si el valor es verdadero o falso, pero no se pueden sumar valores booleanos ni realizar ningún otro tipo de operación con ellos.

En **char**, **byte** y **short**, puede ver el efecto de la promoción con los operadores aritméticos. Toda operación aritmética sobre cualquiera de estos tipos genera un resultado **int**, que después debe ser proyectado explícitamente al tipo original (una conversión de estrechamiento que puede perder información) para realizar la asignación a dicho tipo. Sin embargo, con los valo-

res **int** no es necesaria una proyección, porque todo es ya de tipo **int**. Sin embargo, no se crea que todas las operaciones son seguras. Si se multiplican dos valores **int** que sean lo suficientemente grandes, se producirá un desbordamiento en el resultado, como se ilustra en el siguiente ejemplo:

```
//: operators/Overflow.java
// ¡Sorpresa! Java permite los desbordamientos.

public class Overflow {
  public static void main(String[] args) {
    int big = Integer.MAX_VALUE;
    System.out.println("big = " + big);
    int bigger = big * 4;
    System.out.println("bigger = " + bigger);
  }
} /* Output:
big = 2147483647
bigger = -4
*///:-
```

No se obtiene ningún tipo de error o advertencia por parte del compilador, y tampoco se genera ninguna excepción en tiempo de ejecución. El lenguaje Java es muy bueno, aunque no hasta ese punto.

Las asignaciones compuestas *no* requieren proyecciones para **char**, **byte** o **short**, aún cuando estén realizando promociones que provocan los mismos resultados que las operaciones aritméticas directas. Esto resulta quizá algo sorprendente pero, por otro lado, la posibilidad de no incluir la proyección simplifica el código.

Como puede ver, con la excepción de **boolean**, podemos proyectar cualquier tipo primitivo sobre cualquier otro tipo primitivo. De nuevo, recalquemos que es preciso tener en cuenta los efectos de las conversiones de estrechamiento a la hora de realizar proyecciones sobre tipos de menor tamaño; en caso contrario, podríamos perder información inadvertidamente durante la proyección.

Ejercicio 14: (3) Escriba un método que tome dos argumentos de tipo **String** y utilice todas las comparaciones **boolean** para comparar las dos cadenas de caracteres e imprimir los resultados. Para las comparaciones == y **!=**, realice también la prueba con **equals( )**. En **main( )**, invoque el método que haya escrito, utilizando varios objetos **String** diferentes.

Resumen

Si tiene experiencia con algún lenguaje que emplee una sintaxis similar a la de C, podrá ver que los operadores de Java son tan similares que la curva aprendizaje es prácticamente nula. Si este capítulo le ha resultado difícil, asegúrese de echar un vistazo a la presentación multimedia *Thinking in C*, disponible en *www.MindView.net*.

Puede encontrar las soluciones a los ejercicios seleccionados en el documento electrónico *The Thinking in Java Annotated Solution Guide*, que está disponible para la venta en *www.MindView.net*.