Capítulo 3. Operadores

A. J. Pérez

Operadores

Noción de operador

Los operadores de Java son caracteres especiales que le indican al compilador que se desea realizar una operación sobre algunos operandos. A los operadores que toman un único operando se les llama operadores *unarios*. A los que aparecen antes del operando se les llama *operadores prefijos* y los que van después se les llama *operadores sufijos*. La mayoría de los operadores actúan con dos operandos y se les llama operadores *binarios infijos*. Incluso hay un operador que toma tres operandos y se le llama *operador ternario*.

Java tiene cuarenta y cuatro operadores incorporados, organizados en cinco categorías:

- Aritméticos.
- Asignación.
- Relacionales.
- Operador if-then-else ternario
- Lógicos.
- Nivel de bit.
- Reserva de memoria.

Operadores aritméticos

Los operadores aritméticos se utilizan para operaciones matemáticas, exactamente de la misma manera como están definidos en Álgebra. Los operandos deben ser de tipo numérico. No se pueden utilizar estos operadores con operandos de tipo **`boolean`**, pero se pueden utilizar con tipos **`char`**.

Operador	Resultado
`+`	Suma
`-`	Resta y cambio de signo
`*`	Producto o multiplicación
`/`	Cociente o división
`%`	Módulo o resto de la división entera

Operadores de cálculo

La *suma*, *resta*, *multiplicación* y *división* se comportan como es de esperar con todos los tipos numéricos. El operador **–** (*unario*) cambia el signo del operando al que precede.

Operador módulo

El *operador módulo*, **%**, devuelve el resto de la división entera realizada sin obtener decimales en el cociente. En Java el operador módulo funciona con tipos en coma flotante además de con tipos enteros.

Incremento y decremento

Los operadores incremento y decremento (**++** y **--**) **son una notación abreviada para añadir o restar 1 de un operando**.

```
//...

x = x + 1;              //equivalentes
x += 1;
++x;

x = x - 1;              //equivalentes
x -= 1;
--x;

//...
```

Estos operadores pueden aparecer en forma de *prefijo*, como lo mostrado hasta ahora, y también en forma de *sufijo* cuando siguen al operando. La diferencia entre las dos formas es importante. **En la forma de *prefijo*, el operando se modifica antes de obtener el valor**. En **la forma sufijo, se obtiene el valor, y a continuación el operando se incrementa o decrementa.**

```
//...

dato1 = 4;
dato2 = ++dato1;          // dato2 vale 5, dato1 vale 5

dato3 = 4;
dato4 = dato3++;         // dato4 vale 4, dato3 vale 5

//...
```

Operadores de asignación

Se utilizan para asignar el valor de una expresión a una variable.

Aritméticos y asignación

Operador	Resultado
+=	Suma y asignación

-=	Resta y asignación
***=**	Producto y asignación
/=	Cociente y asignación
%=	Módulo y asignación

Nivel de bit y asignación

Operador	Resultado
<<=	Desplazamiento a la izquierda y asignación
>>=	Desplazamiento a la derecha y asignación
>>>=	Desplazamiento a la derecha y asignación rellenando con ceros
&=	and sobre bits y asignación
|=	or sobre bits y asignación
^=	xor sobre bits y asignación

Si los dos operandos de una expresión de asignación (el de la izquierda y el de la derecha) son de distinto tipo de datos, el valor de la expresión de la derecha se convertirá al tipo del operando de la izquierda.
Por ejemplo, una expresión de tipo real (float, double) se truncará si se asigna a un entero, o una expresión en de tipo double se redondeará si se asigna a una variable de tipo float.
En Java están permitidas las asignaciones múltiples.

Ejemplo:

```
//...

x = y = z = 3;            // equivale a x = 3; y = 3; z = 3;


//...
```

Ejemplo de asignaciones en Java:

```
//...

a += 3;            // equivale a a = a + 3;
a *= 3;            // equivale a a = a * 3;
//...
```

En general:

`variable` **`op=`** `expresión`

Equivale a:

`variable = variable` **`op`** `expresión`

En la siguiente tabla vemos más ejemplos de asignaciones:

```
//...
int i = 5, j = 7, x = 2, y = 2, z = 2;
float f = 5.5F, g = -3.25F;
//...
```

Expresión	Expresión equivalente	Valor final
`i += 5`	`i = i + 5`	`10`
`f -= g`	`f = f - g`	`8.75`
`j *= (i - 3)`	`j = j * (i - 3)`	`14`
`f /= 3`	`f = f / 3`	`1.833333`
`i %= (j - 2)`	`i = i % (j - 2)`	`0`
`x *= -2 * (y + z) / 3`	`x = x * (-2 * (y + z) / 3)`	`-4`

Cada uno de los operadores aritméticos tiene una forma asociada, cuando se tiene una asignación tras la operación. Esto sirve para todos los operaciones que se utilizan de la siguiente forma:

`var = var <op> expresión;`

que se puede reescribir como:

`var <op=> expresión;`

un ejemplo:

```
//...
a = a + 4;              //equivalentes
a += 4;

a = a % 2;              //equivalentes
a %= 2;
```

```
//...
```

Operadores relacionales

Para comparar dos valores, Java tiene el siguiente conjunto de operadores relacionales :

Operador	Interpretación
`==`	Igual que...
`!=`	Distinto que...
`>`	Mayor que...
`<`	Menor que...
`>=`	Mayor o igual que...
`<=`	Menor o igual que...

Se pueden comparar operandos enteros, en coma flotante y caracteres para ver cual es mayor o menor que otro. Cada uno de los operadores devuelve como resultado un valor de tipo **`boolean`**.

Operadores lógicos

Todos los *operadores lógicos* evalúan dos valores **`boolean`** para dar como resultado un valor **`boolean`**.

Operador	Interpretación
`&`	AND lógico
`|`	OR lógico
`^`	OR exclusivo lógico
`||`	OR en cortocircuito
`&&`	AND en cortocircuito
`!`	NOT unario lógico
`?:`	if-then-else ternario

Los operadores lógicos, AND, OR y XOR operan sobre valores booleanos según la siguiente tabla. El operador lógico NOT invierte el estado booleano.

Operando A	Operando B	AND `A & B` `A && B`	OR `A | B` `A || B`	XOR `A ^ B`
`false`	`false`	`false`	`false`	`false`
`false`	`true`	`false`	`true`	`true`
`true`	`false`	`false`	`true`	`true`
`true`	`true`	`true`	`true`	`false`

Operando A	NOT `!A`
`false`	`true`
`true`	`false`

Operadores lógicos en cortocircuito

Existen versiones secundarias de los operadores lógicos AND y OR, a los que se llama *operadores lógicos en cortocircuito*. La utilización de estos operadores en una expresión provoca que la evaluación se detenga inmediatamente en el momento en que se conoce el resultado de la expresión que se está evaluando. Así, por ejemplo, este fragmento de código muestra la manera de aprovechar la evaluación lógica en cortocircuito para asegurarse que una operación de división es válida antes de evaluarla.

```
//...

// Si denominador es 0 la segunda parte de la expresión no se evalúa y se
evita dividir por 0
if (denominador != 0 && numero / denominador > 10) {
      //...
}
//...
```

Dado que se utilizó la forma en cortocircuito de AND, **&&**, no se corre el riesgo de provocar una *excepción* (error) en tiempo de ejecución al intentar dividir por cero.

Operador if-then-else ternario

Java incluye el mismo *operador ternario* que C y C++. La forma general es:

<expresión> ? sentencia1 : sentencia2

Donde expresión puede ser cualquier expresión que dé como resultado un valor **boolean**. Si el resultado es **true** entonces se ejecuta `sentencia1`, en caso contrario se ejecuta `sentencia2`. La limitación es que `sentencia1` y `sentencia2` deben devolver el mismo tipo de dato y además no puede ser **void**.

```
//...

// Pide un número y dice si es par o impar

Scanner teclado = new Scanner(System.in);

System.out.print("Teclea un número: ");
int num = teclado.nextInt();

int resto = num % 2;                    // resto de la división entera

// utiliza un if in-line y el operador == de comparación
String mensaje = (resto == 0) ? " Es par": " Es impar";

// muestra
System.out.println(num + mensaje);

//...
```

Operadores a nivel de bit

Los tipos numéricos enteros, **`long`**, **`int`**, **`short`**, **`byte`** y **`char`** tienen un conjunto adicional de operadores que pueden **modificar e inspeccionar los *bits* que componen sus valores**. Estos operadores se resumen a continuación.

Todos los tipos enteros se representan mediante números binarios de anchura variable. Por ejemplo, el valor de tipo **`byte`** 42 en binario es `00101010`. **Los operadores a nivel de bit operan independientemente sobre cada uno de los bits de un valor**.

Operador	Resultado	Operador	Resultado
~	NOT unario a nivel de bit		
&	AND a nivel de bit	**&=**	AND a nivel de bit y asignación
|	OR a nivel de bit	**|=**	OR a nivel de bit y asignación
^	OR exclusivo a nivel de bit	**^=**	OR exclusivo a nivel de bit y asignación
>>	Desplazamiento a la derecha	**>>=**	Desplazamiento a la derecha y asignación
>>>	Desplazamiento a la derecha rellenando con ceros	**>>>=**	Desplazamiento a la derecha rellenando con ceros y asignación
<<	Desplazamiento a la izquierda	**<<=**	Desplazamiento a la izquierda y asignación

NOT

El operador NOT *unario*, **~**, **invierte todos los bits de su operando; es lo que se llama complemento a 1**.

Por ejemplo:
El número 42 tomándolo de tipo **byte**, tiene el patrón de bits `00101010`; después de aplicar el operador NOT se convierte en `11010101` correspondiente a `-43` en *complemento a 2* que es como se representan los enteros negativos en Java. **Invertir a nivel de bits un entero positivo siempre resulta el correspondiente negativo más uno**.

```
//...

byte dato1 = 42;                // 42  en binario 00101010
byte dato1 = ~42;               // -43 en C2      11010101

//...
```

AND

El operador AND, **&**, **combina los *bits* de dos valores enteros de manera que se obtiene un 1 si ambos operandos son 1, obteniendo 0 en cualquier otro caso.**

```
//...
```

```
byte dato1 = 42;                 //42 en binario 00101010
byte dato2 = (byte) dato1 & 15;  //15 en binario 00001111 &
                                 //resultado(10) 00001010

//...
```

OR

El operador OR, **|**, **combina los bits de dos valores enteros de manera que se obtiene un 1 si cualquiera de los operandos es un 1**.

```
//...

byte dato1 = 42;                 //42 en binario 00101010
byte dato2 = (byte) dato1 | 15;  //15 en binario 00001111 |
                                 //resultado(47) 00101111
//...
```

XOR

El operador XOR, **^**, **combina los bits de de dos valores enteros de manera que se obtiene un 1 si cualquiera de los operandos es un 1, pero no ambos, y 0 en caso contrario**.

```
//...

byte dato1 = 42;                 //42 en binario 00101010
byte dato2 = (byte) dato1 ^ 15;  //15 en binario 00001111 ^
                                 //resultado(37) 00100101
//...
```

La tabla siguiente muestra cómo actúan cada operador a nivel de bit sobre cada combinación de bits de los operandos.

Operando A	Operando B	AND A & B	OR A | B	XOR A ^ B
0	**0**	**0**	**0**	**0**
0	**1**	**0**	**1**	**1**
1	**0**	**0**	**1**	**1**
1	**1**	**1**	**1**	**0**

Operando A	NOT ~A
0	1
1	0

Desplazamiento a la izquierda

El operador de desplazamiento a la izquierda, **<<**, **mueve hacia la izquierda todos los bits del operando de la izquierda un número de posiciones de *bit* especificado en el operando de la derecha**. Al realizarse el desplazamiento **se pierden por el extremo izquierdo operando el número de bits desplazados y se rellena con ceros por la derecha** el mismo número de *bits*.
En este ejemplo se desplaza el valor 8 a la izquierda dos posiciones de *bit*, obteniendo como resultado que variable sea 32.

```
//...

byte dato1 = 8;                 //8 en binario  00000100    <<2
dato1 = dato1 << 2;             //resultado(32) 00010000

//...
```

Desplazamiento a la derecha

El operador de desplazamiento a la derecha, **>>**, **mueve hacia la derecha todos los bits del operando de la izquierda un número de posiciones de bit especificado por el operando de la derecha**.
En este ejemplo se desplaza el valor 32 a la derecha dos posiciones de *bit*, obteniendo como resultado que variable sea 8.

```
//...

byte dato1 = 32;                //32 en binario 00010000   >>2
dato1 = dato1 >> 2;             //resultado(8)  00000100

//...
```

Cuando un dato tiene bits que se desplazan fuera por la parte izquierda o derecha de una palabra, esos bits se pierden.

Este ejemplo desplaza el valor 35 a la derecha los dos bits inferiores por la derecha, y el resultado vuelve a ser que el valor de a sea 8.

```
//...

byte dato1 = 35;                //35 en binario 00010011   >>2
dato1 = dato1 >> 2;             //resultado(8)  00000100

//...
```

Desplazamiento a la derecha sin signo

El operador de desplazamiento a la derecha rellenando con ceros, **>>>**, **desplaza introduciendo ceros en los bits más significativos, a lo que se llama desplazamiento sin signo**.

En este ejemplo, `dato1` se establece a `-1`, lo que supone -en `complemento a 2`- que los 32 *bits* sean `1`. Si el valor de `dato1` se desplaza 24 *bits* a la derecha, se rellenan los 24 *bits* superiores con `0`, **ignorando el bit de signo normal**; la variable pasa a valer `255`.

```
//...

int dato1 = -1;                 //-1 en C2 11111111111111111111111111111111
dato1 = dato1 >>> 24;           //255      00000000000000000000000011111111

//...
```

Esto es especialmente útil en la creación de máscaras para operaciones en el procesado de imágenes.

Todos los operadores binarios a nivel de bit tienen una forma similar a la de los operadores algebraicos, que hacen una asignación automática del resultado al operador de la izquierda.

Por ejemplo:

```
//...

int dato1 = 32;
int dato2 = 40;

dato1 = dato1 >> 4;           //son equivalentes
dato1 >>= 4;

dato1 = dato1 | dato2;       //son equivalentes
dato1 |= dato2;

//...
```

Precedencia de operadores

La tabla siguiente muestra el orden de todas las posibles operaciones en Java, desde la precedencia más alta a la más baja.

La más alta					
`( )`					Modificador precedencia.
`.`	`[ ]`				Acceso unario.
`++`	`--`				Postfijo unario.
`++`	`--`	`~`	`!`	`(tipo)`	Prefijo unario.
`*`	`/`	`%`			Aritmético binario.
`+`	`-`				Aritmético binario.
`>>`	`>>>`	`<<`			Desplazamiento bits binario.

>	>=	<	<=			Relacional binario.
==	!=					Relacional binario.
&						Lógico binario.
^						Lógico binario.
|						Lógico binario.
&&						Lógico cortocircuito binario.
||						Lógico cortocircuito binario.
?:						Condicional ternario.
=	+=	-=	*=	/=	%=	Asignación binario.

Ejercicios

1. (*) Escribe un programa simple que muestre la "tabla de verdad" de los operadores lógicos utilizando expresiones donden intervengan los operadores adecuados. Al ejecutar se debe conseguir una apariencia semejante a:

```
AND cortocircuito (&&)
    true  && true:  true
    true  && false: false
    false && true:  false
    false && false: false

OR cortocircuito (||)
    true  || true:  true
    true  || false: true
    false || true:  true
    false || false: false

AND lógico (&)
    true  & true:  true
    true  & false: false
    false & true:  false
    false & false: false

OR lógico (|)
    true  | true: true
    true  | false: true
    false | true: true
    false | false: false

OR exclusivo lógico (^)
    true  ^ true:  false
    true  ^ false: true
    false ^ true:  true
    false ^ false: false

NOT lógico (!)
    !true:  false
    !false: true
```

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- CARRERES, J. *Manual de Java*. [en línea] http://www.oocities.org/collegepark/quad/8901/indice.html
- DEITEL, P. J. y DEITEL, H. M. *Cómo programar en Java.* Ed.Pearson Educación, 7ª edición. Méjico, 2008. ISBN: 978-970-26-1190-5
- http://puntocomnoesunlenguaje.blogspot.com.es/2012/04/operadores.html