

1. OPERADORES PARA ENTEROS

Los **operadores** que proporciona el lenguaje Java permiten escribir **expresiones** de cierta complejidad.

Estos operadores pueden ser unarios, es decir, los que trabajan con un único operando; o binarios, cuando lo hacen con dos operandos a la vez.

Nosotros ya hemos utilizado algún operador de ambos tipos. Por ejemplo, el operador punto (.) permite acceder a los métodos y propiedades de un objeto.

Por otra parte, el operador + permite realizar la suma aritmética de dos valores numéricos o bien la concatenación de dos cadenas de texto.

Lo importante es recordar que Java es un lenguaje rígido en cuanto a las comprobaciones de tipo; es decir, una expresión en la que aparezca un operador sólo tendrá éxito cuando los operandos pertenezcan a un tipo adecuado para dicho operador.

Los operadores de Java se dividen en categorías según la naturaleza de la operación que llevan a cabo.



En el caso de operadores para enteros podrá encontrar:

- Operadores de asignación: =, +=, -=, *=, /=
- Operadores de comparación: ==, !=, <, >, <=, >=
- Operadores de signo unitarios: -
- Operadores aritméticos: +, -, *, /, %
- Operadores de incremento y decremento: ++, --
- Operadores de desplazamiento a nivel de bit: >>, <<
- Operador negación lógica a nivel de bit: ~
- Operadores AND, OR, XOR a nivel de bit: &, |, ^

Observe el código de la página siguiente. Hemos declarado tres variables de tipo **short**, es decir, que podrán contener valores enteros (en el rango [-32768 a 32767]). A las dos primeras se les asigna el valor **100** y a la tercera se le asigna el resultado de realizar la operación de adición.

Éste es un ejemplo típico en el que puede ver el uso de varios operadores. El más utilizado es el de asignación (=).

Al mismo tiempo, puede ver cómo se utiliza el operador + para realizar la operación de adición o suma.

```
public class Enteros {  
  
    public static void main(String[] args)  
    {  
        short operando1 = 100;  
        short operando2 = 100;  
        short resultado;  
  
        resultado = (operando1 + operando2);  
        System.out.println(resultado);  
    }  
  
}
```

Si intenta compilar este sencillo ejemplo, obtendrá un error. El problema surge cuando se asigna el resultado de la operación suma a una variable **short**.

¿Dónde está el error? Muy sencillo, debe saber las siguientes reglas sobre los operadores para enteros:

- Los operadores de igualdad y comparación siempre producen un resultado **boolean** (true o false).
- Los otros operadores binarios producen un resultado de tipo **int** o de tipo **long** (nunca byte o short). Si alguno de los operandos es un long, entonces el resultado es un long. También será un long si el resultado desborda la capacidad de una variable int. En otro caso, el resultado será un int.

Por lo tanto, el error se ha producido al asignar un valor int a una variable **short**, ya que el resultado de la suma, aunque los operandos sean de tipo **short**, es un valor **int**.

Java introduce dos tipos de **conversiones** entre tipos. Las conversiones pueden forzar a que una expresión de cierto tipo sea de un tipo diferente.

Java puede producir conversiones **implícitas**, es decir, sin escribir código para ello, si el tipo objetivo es mayor (en capacidad) que el tipo inicial.

Por ejemplo, podrá convertir implícitamente un **short** a un **int** sin necesidad de indicarlo.

Sin embargo, si el tipo objetivo es menor que el tipo inicial, entonces debe indicarlo usted **explícitamente** en el código, demostrando “*que sabe lo que hace*”.

Por ejemplo, podríamos convertir explícitamente el resultado de la suma de nuestro ejemplo (un **int**) al tipo **short**, pero deberíamos tener en cuenta que se podría perder información, ya que **short** admite un rango de valores inferior a **int**.

Para ello se utiliza la siguiente sintaxis: *(tipo objetivo) expresión*. Donde el tipo objetivo es el nombre del tipo de datos al que queremos convertir la expresión.

```
resultado = (short) (operando1 + operando2);
```

Lo que acaba de hacer es crear una expresión de conversión explícita (**casting** en inglés). Está indicando que desea convertir el valor de la suma en un valor de tipo **short**.



Recuerde que no se puede cambiar el tipo de datos de una variable posteriormente a su declaración.

Es importante entender que simplemente estamos obteniendo una expresión “*equivalente*” con el tipo de datos que ponemos entre paréntesis.

Si la expresión siguiente al casting fuera una variable, como en **(short)a**, entonces esa expresión se evaluaría como del tipo **short**, pero la variable **a** seguiría siendo del tipo original, ya que no se produciría ningún cambio en ella.

Sin embargo, debe saber muy bien lo que está haciendo ya que, en caso contrario, podría producir resultados inesperados.

Ejecutando ahora el código obtendremos el resultado **200**. Como el valor **200** está dentro del rango de valores válidos para el tipo **short**, todo va bien.

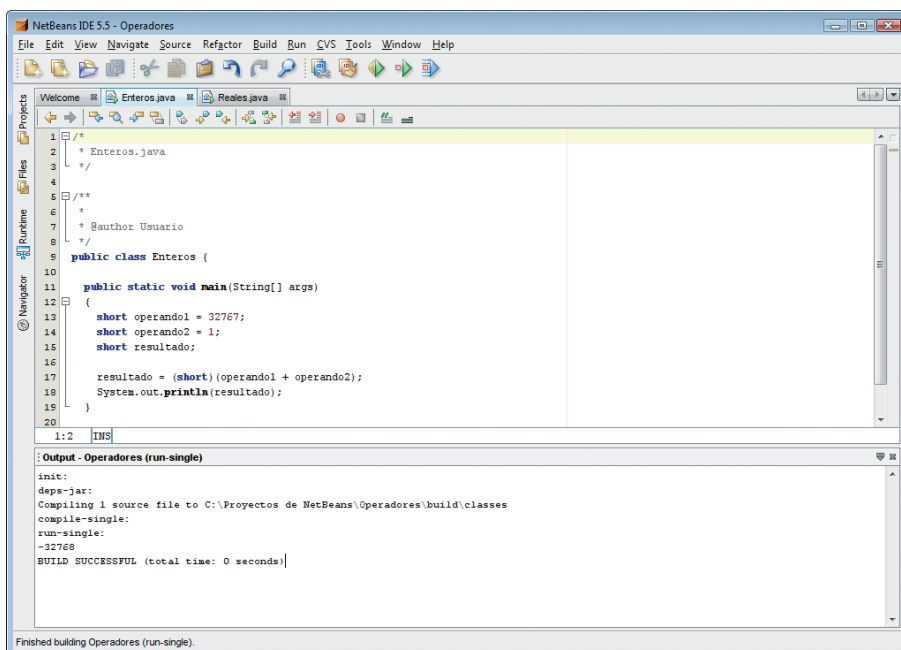
Sin embargo, modifiquemos el valor inicial de los operandos, de forma que el resultado de la operación no sea un valor de tipo **short**, como ve a continuación:

```
short operando1 = 32767;
short operando2 = 1;
```

Observe cómo ahora el resultado de sumar los dos operandos ya no es un valor de tipo **short**, sino que se supera en **1** el valor máximo permitido en este tipo.

Al ejecutar el nuevo código obtendremos el valor negativo **-32768**, pero ningún error.

A continuación veremos lo que ha ocurrido.



2. DESBORDAMIENTO (OVERFLOW Y UNDERFLOW)

Se llama **desbordamiento** u overflow al hecho de que el valor que recoge una variable de cierto tipo de datos supera el rango de valores válidos de ese tipo.

Volviendo al ejemplo que nos ocupa, vemos que el tipo **short** tiene un rango de valores válidos que va desde el valor negativo **-32768** hasta el valor positivo **32767**.

Sin embargo, el resultado de la operación **32767 + 1** es **32768**, que supera este rango.

Cuando esto sucede, Java no produce ningún error sino que empieza a contar desde el número negativo más pequeño, dando resultados inesperados y difíciles de detectar si no se han establecido las correspondientes comprobaciones.

El desbordamiento es un asunto a tener muy en cuenta y se relaciona directamente con la elección de los tipos de datos adecuados.

Usted podría pensar que lo mejor es utilizar siempre los tipos de datos más grandes para no tener este tipo de problemas.

Sin embargo, a mayor capacidad del tipo de datos, también mayor necesidad de memoria es requerida cada vez que declara una variable de ese tipo.

Por ello, su código debe ser lo suficientemente robusto y eficiente para tratar este tipo de situaciones.

Por otra parte, la situación opuesta al overflow es el underflow.

Por ejemplo, si hiciera una operación como **-32768 - 1** con variables de tipo **short**, produciría underflow.

En el ejemplo que estamos siguiendo, lo lógico sería utilizar el tipo **int** para la variable suma y disponer, de esta forma, de un rango de valores mucho mayor.

```
public class Enteros {

    public static void main(String[] args)
    {
        short operando1 = 32767;
        short operando2 = 10;
        int resultado;

        resultado = (operando1 + operando2);
        System.out.println(resultado);
    }

}
```

3. OPERADORES PARA REALES

Java permite trabajar con números reales, es decir, con aquellos que tienen una parte entera y otra decimal. Para ello, el lenguaje aporta dos tipos de datos: **float** y **double**.

Trabajar con números reales es más difícil que hacerlo con enteros ya que, al realizar operaciones, pueden darse resultados erróneos debido a redondeos o pérdidas de información.

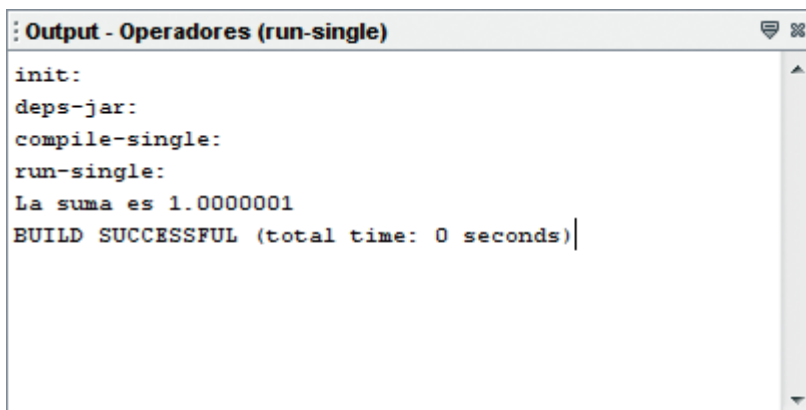
Los valores **float** y **double** son siempre valores aproximados y no exactos como los enteros, ya que dependen de la exactitud con la que se han almacenado.

```
public class Reales {  
  
    public static void main(String[] args)  
    {  
        float suma = 0;  
        for (byte i=0; i<10; i++)  
        {  
            suma = suma + (float)0.1;  
        }  
        System.out.println("La suma es " + suma);  
    }  
  
}
```

Estudie el código de este archivo. Observe que se declara una variable local de tipo **float** que se inicializa al valor **0** (recuerde que todas las variables locales deben inicializarse).

Después se utiliza un *bucle*, tema que estudiaremos más tarde, de forma que se suma 10 veces la cantidad 0.1, es decir, que al final del bucle la variable **suma** debería tener el valor **0.1 * 10 = 1.0**.

Si ejecuta este archivo, comprobará que el resultado es ligeramente distinto al esperado: **1.0000001**.



```
Output - Operadores (run-single)  
init:  
deps-jar:  
compile-single:  
run-single:  
La suma es 1.0000001  
BUILD SUCCESSFUL (total time: 0 seconds)
```

Este ejemplo nos debe servir para trabajar de forma muy precavida con los números reales, es decir de tipo **float** o de tipo **double**.

Por ejemplo, no es adecuado realizar una comparación exacta entre dos números reales ya que el resultado puede verse afectado por la pérdida de exactitud.

Cuando desee comparar dos números reales, lo mejor es comparar la diferencia de ambos respecto a una constante que determine la exactitud de la operación.



En nuestro ejemplo, podría decir que la expresión `suma == 1.0` es cierta si la diferencia entre ambos es menor de **0.0000001**.

Los símbolos utilizados para los operadores que trabajan con reales son los mismos que para los que trabajan con enteros.

Operadores con números reales:

- Operadores de asignación.
- Operadores de comparación.
- Operadores de signo unitarios.
- Operadores aritméticos.
- Operadores incremento y decremento.

Por otra parte, debe tener en cuenta las siguientes reglas:

- El resultado de una operación en la que aparece al menos un valor **double** es siempre del tipo **double**.
- En una operación binaria en la que aparece un entero y un número real, el resultado será del tipo de datos del número real.
- Las operaciones binarias que involucren dos números **float** será un **float**.
- Un valor literal que incluye decimales es, por defecto, un **double**.

La última regla es la que ha requerido de la conversión explícita del valor **1.0** en **float**: `suma = suma + (float)0.1;`

Si no lo hubiéramos hecho, el compilador habría detectado un error al no poder convertir implícitamente un valor **double** (el resultado de la operación) en un valor **float** (el tipo de la variable **suma**).

4. OPERADOR DE CONCATENACIÓN

Un operador especial es el signo **+** ya que puede actuar tanto con valores numéricos como con valores de texto.

Cuando los dos operandos son numéricos, entonces se realiza la suma aritmética de ambos, mientras que, cuando uno o ambos son cadenas de texto, entonces se realiza lo que se llama la concatenación de las cadenas.

Por ejemplo la operación “*Hasta*” + “*luego*” produce una tercera cadena “*Hasta luego*” (compruebe cómo en la primera cadena existe un espacio en blanco al final).

Las cadenas de texto deben aparecer siempre entre comillas dobles.

```
public class Puerta {  
  
    private String color;  
    private String modelo;  
    private double precio;  
  
    public Puerta() {  
        this.color = "Blanco";  
        this.modelo = "Interna";  
        this.precio = 0.0;  
    }  
  
    public Puerta(String color, String modelo,  
                  double precio)  
    {  
        this.color = color;  
        this.modelo = modelo;  
        this.precio = precio;  
    }  
}
```

En este caso se ha definido una clase **Puerta** que consta de tres propiedades: **color**, **modelo** y **precio**.

Además, se han escrito dos constructores para la clase: el primero se llama sin parámetros, por lo que toma los valores por defecto; el segundo utiliza parámetros, por lo que se puede indicar el valor de dichas propiedades.

Fíjese que se puede crear más de un constructor para una misma clase siempre que puedan diferenciarse entre ellos, es decir, que la lista de parámetros sea distinta.

Todas las clases de Java poseen un método heredado de la clase **Object** que permite devolver la representación, en forma de cadena de texto, de un objeto. Este método se llama **toString**.

En la mayoría de los casos, usted deseará sobrescribir este método para la clase que está escribiendo, ya que, por defecto, lo único que hace es devolver el nombre de la clase.


```
public String toString() {
    return ("Color: " + color + ", " + "Modelo: " + modelo
        + ", " + "Precio: " + precio);
}
```

Estamos sobrescribiendo dicho método para que devuelva una representación en forma de cadena de texto de un objeto **Puerta**. Para ello, utilizamos el operador de concatenación.

Por lo tanto, al aplicar el método **toString** a un objeto de la clase **Puerta**, obtendremos una cadena de texto del estilo de:

“Color: Rojo, Modelo: MP021, Precio: 125.48”

Este método puede servir para aportar la descripción de la puerta en un momento dado. Lo importante es darse cuenta de cómo se va construyendo la cadena final a partir de otras cadenas.

Así, la operación **"Color: " + color** proporciona una cadena en la que el valor de la propiedad **color** se inserta al final de la primera cadena. De la misma forma, ocurre con el resto de la expresión.

Observe cómo no importa el tipo de datos de las propiedades ya que se convierte implícitamente a texto al realizar la concatenación.

Ahora utilicemos dicho método:

```
public class NewClass {

    public static void main(String[] args)
    {
        Puerta p1 = new Puerta();
        Puerta p2 = new Puerta("Gris", "P025", 100.5);
        System.out.println(p1.toString());
        System.out.println(p2.toString());
    }
}
```

Creamos un objeto **p1** de la clase **Puerta**, utilizando el constructor por defecto y otro objeto, **p2**, indicando el valor exacto de sus propiedades.

Observe lo que conseguimos: si deseamos especificar un determinado color, modelo y precio, utilizaremos este constructor; en caso contrario, utilizaremos el constructor por defecto.

Seguidamente imprimiremos en pantalla el resultado de invocar el método **toString**.

Recuerde que el espacio en blanco es un carácter más que deberá insertar en sus cadenas si desea que la concatenación muestre separación entre ellas.

```
Output - Puerta (run-single)
init:
deps-jar:
compile-single:
run-single:
Color: Blanco, Modelo: Interna, Precio: 0.0
Color: Gris, Modelo: P025, Precio: 100.5
BUILD SUCCESSFUL (total time: 0 seconds)
```



PRECEDENCIA DE LOS OPERADORES

La precedencia de los operadores determina el orden en el que se evalúan las expresiones, lo que repercute en el resultado obtenido.

Por ejemplo, en la expresión **4 + 5 * 2**, el resultado que obtenemos es **14**, ya que se realiza primero la multiplicación y después, al resultado, la suma. De esta forma, la expresión se evalúa en los siguientes pasos: **4 + 5 * 2 = 4 + 10 = 14**.

Esto es así porque el operador ***** tiene mayor precedencia que el operador **+**. A continuación se detalla la precedencia de los operadores, de mayor a menor:

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

Siempre puede utilizar los paréntesis para cambiar el orden de evaluación. Por ejemplo, la expresión **(4 + 5) * 2** dará como resultado **18**. Además, ante dos operadores de la misma precedencia, el orden de evaluación se realiza de izquierda a derecha.