

## EL OPERADOR CONDICIONAL "?:"

Este operador puede considerarse una suerte de abreviatura de la sentencia condicional *if*, y aunque esta sentencia se verá con calma en otra unidad, merece la pena adelantar el uso de *if* para poder entender con mayor facilidad el operador condicional.

La sintaxis básica de la sentencia *if* es

```
if(condicion){
    bloque1: Sentencias que se ejecutan si se cumple la condición
}
else{
    bloque2: Sentencias que se ejecutan si no se cumple la condición
}
```

Se lee e interpreta: si se cumple la condición se ejecutan las sentencias del bloque1, y si no (else) se cumple la condición, se va directamente a ejecutar las sentencias del bloque2. Hay que tener en cuenta que en el caso de que se cumpla la condición y se ejecuten las sentencias de bloque1, una vez ejecutada la última instrucción del bloque1, el programa no ejecuta el bloque2 si no que "salta" al final del *if*.

Bloque: conjunto de sentencias encerradas entre llaves {}

Condición: una expresión de tipo booleano, es decir, que al evaluarse se obtiene true/false

```
class Unidad1{
    public static void main(String[] args){
        int x=2, y=3;
        if(x>y){
            System.out.println("x es mayor que y");
        }else{
            System.out.println("x no es mayor que y");
        }
        System.out.println("FIN PROGRAMA");
    }
}
```

Observa:

- La sentencia *if* es "larga" y para leerla con facilidad se escribe en varias líneas con la estructura anterior. La sintaxis del *if* se verá con más detalle en próximas unidades.
- La instrucción `System.out.println("FIN PROGRAMA");` no pertenece al *if* y siempre se ejecuta independientemente de que la condición se cumpla o no

Cambia los valores de *x* e *y* y asegúrate de entender qué ocurre cuando se cumple la condición y qué ocurre cuando no se cumple.

### Ejercicio U1\_B6\_E1:

Determinar si el valor de una variable *x* es par o impar. Nos ayudamos para averiguarlo del operador %. Un número es par si es divisible entre 2, es decir, si al dividirlo entre dos obtenemos de resto 0.

### Ejercicio U1\_B6\_E2:

Tenemos almacenados dos números enteros almacenados en variables *x* e *y*. Queremos averiguar si *x* es múltiplo de *y*. De nuevo, ayúdate del operador módulo.

### Ejercicio U1\_B6\_E3:

Si el valor de *x* al cuadrado es mayor que 100 aumenta el valor de la variable *y* en 1 y lo imprime, en caso contrario no se hace nada.

## El operador condicional . Un operador ternario.

Los operadores normalmente son binarios (dos operandos) como por ejemplo el operador multiplicación "\*", por ejemplo  $a*b \Rightarrow$  el operando  $*$  tiene dos operandos  $a$  y  $b$ . También hay operadores unarios (un operando) como por ejemplo  $++x \Rightarrow$  el operador  $++$  tiene el operando  $x$ , y también hay un operador ternario (tres operandos), **el operador ?**: Es una especie de abreviatura de la instrucción `if`. El primer programa ejemplo escrito con `if` lo volvemos a escribir ahora usando este operador

```
class Unidad1{
    public static void main(String[] args){
        int x=2, y=3;
        System.out.println(x>y?"x es mayor que y":"x no es mayor que y");
        System.out.println("FIN PROGRAMA");
    }
}
```

Es muy compacto y por eso es difícil de leer y de usar pero tiene ventajas, entre otras:

- internamente genera un código máquina eficiente.
- Forma parte de una expresión que devuelve un valor y esto permite por ejemplo usarlo en el `return` de los métodos (ya veremos esto....)
- etc.

**El operador condicional ?: siempre forma parte de una expresión que devuelve un valor.**

Observa que el operador condicional sintácticamente usa dos símbolos: el `?` y el `:`

Y este operador de dos símbolos siempre forma parte de una expresión con la siguiente sintaxis:

*Operando1?operando2:operando3*

que podemos ver como

*condicion?expresionSiSeCumpleCondicion:expresionSiNOseCumpleCondicion*

Por tanto, todo lo anterior es una expresión, que a su vez contiene tres subexpresiones:

- *operando1* siempre de tipo boolean
- *operando2* y *operando3* de cualquier tipo

Y como toda expresión, una vez calculada devuelve un valor

Ejemplo: una variante del ejemplo anterior de forma que ahora hacemos que el operador condicional devuelve un entero

```
class Unidad1{
    public static void main(String[] args){
        int x=2, y=3;
        System.out.println("el mayor es "+ (x>y?x:y));
    }
}
```

podemos guardar el valor que devuelve la expresión en una variable para visualizar mejor la devolución.

```
class Unidad1{
    public static void main(String[] args){
        int x=2, y=3;
        int z=x>y?x:y;
        System.out.println("el mayor es "+ z);
    }
}
```

Respecto al if encontramos dos diferencias muy importantes:

- El if es muy elástico, no tiene únicamente “la forma” del ejemplo, como veremos más adelante. Por el contrario, el operador ?: tiene una sintaxis limitada al formato anterior.
- El if no devuelve un valor pero el operador ?: como todo operador, forma parte de una expresión que devuelve un valor al ser evaluada, concretamente el operador ?: devuelve el valor de evaluar *operando2* si la condición es cierta o bien *operando3* si no es cierta.

#### **Ejercicio U1\_B6\_E4:**

Repite Ejercicio U1\_B6\_E1 con el operador condicional

#### **Ejercicio U1\_B6\_E5:**

Repite Ejercicio U1\_B6\_E2 con el operador condicional

#### **Ejercicio U1\_B6\_E6:**

Repite Ejercicio U1\_B6\_E3 con el operador condicional

## **OPERADORES A NIVEL DE BIT**

Se usan poco. Sólo en contextos muy determinados, como por ejemplo en programas que trabajan en algoritmos de encriptación o en programas que es prioritario la eficiencia de ejecución. Nosotros les echamos un vistazo porque nos hace ser conscientes de que por debajo de nuestro código de alto nivel siempre hay bits

Los operadores de nivel de bit trabajan con valores enteros, y permiten hacer cambios en los bits de dichos enteros. Los operadores a nivel de bit son:

&      and a nivel de bit  
 |      or a nivel de bit  
 ^      xor a nivel de bit  
 <<    desplazamiento a la izquierda, rellenando con ceros a la derecha  
 >>    desplazamiento a la derecha, rellenando con el bit de signo por la izquierda  
 >>>   desplazamiento a la derecha rellenando con ceros por la izquierda

### **Operadores de desplazamiento.**

Permiten desplazar una o varias posiciones los bits de un valor hacia la derecha o hacia la izquierda, esto nos permite jugar con la aritmética binaria para conseguir un determinado efecto. Por ejemplo, desplazar hacia la izquierda en aritmética binaria es equivalente a multiplicar por dos

Imagina que tengo una variable entera *a* inicializada a 1. Internamente se codifica con 32 bits

```
int a=1;
```

00000000000000000000000000000001 valor de 1 en binario con 32 bits

Si desplazo todos los bits hacia la izquierda, despreciando al mismo tiempo el primero de la izquierda e introduciendo un cero por la derecha estoy haciendo un desplazamiento de un bit a la izquierda quedando los siguientes 32 bits

00000000000000000000000000000010

Los bits anteriores representan un 2, es decir, he multiplicado por 2 el valor inicial ( $2*1=2$ ). ¿Y cómo se "indica" este desplazamiento desde el código java? con el operador de desplazamiento a la izquierda <<

```
a=a<<1; //el 1 quiere decir que desplaza 1 bit
```

Observa como sucesivos desplazamientos implican ir multiplicando por dos el valor anterior

[illegible]

Ejemplo: comprobar cómo desplazando a nivel de bit voy multiplicando por dos

```
class Unidad1{
    public static void main(String[] args){
        int a=1;
        System.out.println(a);
        a=a<<1;
        System.out.println(a);
        a=a<<1;
        System.out.println(a);
        a=a<<1;
        System.out.println(a);
        a=a<<1;
        System.out.println(a);
        a=a<<1;
        System.out.println(a);
    }
}
```

como bien habrás presentado, si << desplaza a la izquierda y por tanto multiplica por 2, >> desplaza a la derecha y divide por 2.

*en la calculadora windows:*

**lsh(left shift):** desplazamiento a la izquierda es equivalente al `<<` java

rsh(right shift): desplazamiento a la derecha es equivalente al >> java

después de pulsar lsh/rsh pulsas el número de bits a desplazar (1,2,3, ...) y luego pulsas en =

por ejemplo

2lsh1= nos da 4

en la calculadora google chrome es similar y es la que vamos a usar

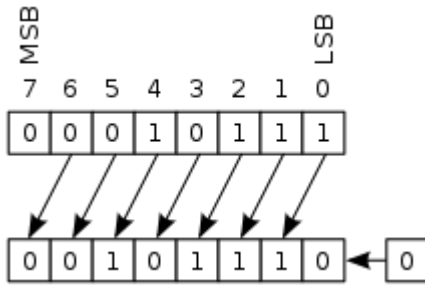


<< para desplazar a la izquierda

A>> y L>> para desplazar a la derecha. La diferencia entre las dos últimas es como rellena lo que vemos a continuación

## con qué bits se rellena al desplazar a la izquierda

por cada desplazamiento se introduce un bit 0 por la derecha. Comprueba esta afirmación con calculadora. Gráficamente, con sólo 8 bits para simplificar lo reflejamos en el siguiente dibujo

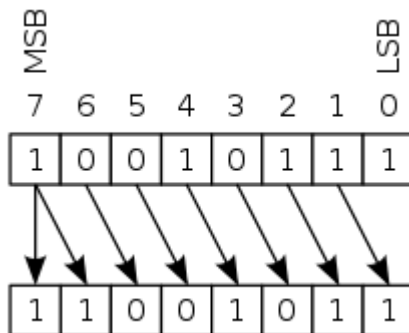


### con qué bits se rellena al desplazar a la derecha

Dos posibilidades:

- rellena con el bit de signo por la izquierda, es decir, con 0 si el número a desplazar es positivo y unos si es negativo. En Java se hace con `>>` y calculadora chrome con `A>>`

En este dibujo se rellena con bit de signo



Comprueba esto con la calculadora desplazando - 4 a la derecha y comprueba que se obtiene un resultado equivalente con JShell

```
jshell> -4>>1
$1 ==> -2
jshell>
```

```
class Unidad1{
public static void main(String[] args){
    System.out.println(-4 >>1);
}
```

```
}
```

- rellenando con un 0 por la izquierda (siempre con 0, sin tener en cuenta signo). esta forma de rellenar se consigue con `>>>` en Java o con `L>>` en calculadora chrome

Ejemplo:

`-4 >>>1`

genera un número positivo ya que introduce un 0 por la izquierda y pasa el número de negativo a positivo

```
jshell> -4>>>1
$2 ==> 2147483646
```

comprueba esto MISMO con calculadora chrome (con `L>>`)









### **Convertir un entero en un String que visualice sus bits.**

Cuando a `println()` se le pasa un entero lo imprime, por defecto, en base 10. Si quiero imprimirlo en base 2 lo que podemos hacer es pasar el entero a un método especial que lo convierte en un String que visualiza los bits del entero.

ya entenderás más adelante que:

- Integer es una clase.
- `toBinaryString()` es un método de la clase Integer que le pasas como argumento un entero y devuelve un String que contiene los 0s y 1s correspondientes a la representación en binario de dicho entero

```
class Unidad1{
    public static void main(String[] args){
        int x=15;
        String xEnBinario=Integer.toBinaryString(x);
        System.out.println("x en base 10: "+x);
        System.out.println("x en base 2: "+xEnBinario);
    }
}
```

`toBinaryString()` no devuelve los números en complemento a 2 con sus 32 bits, si no que devuelve números en binario puro. Usar números negativos es posible pero confuso y no merece la pena usarlos. Con `toBinaryString()` usaremos sólo valores positivos y no olvides que devuelve un número en binario puro, sin bit de signo.

```
jshell> Integer.toBinaryString(2)
$30 ==> "10"

jshell> Integer.toBinaryString(4)
$31 ==> "100"

jshell>
```

### **Convertir un String que contiene un número binario en un entero**

La clase Integer tiene el método `parseInt()` al que se indican dos argumentos:

- la cadena de bits dentro de un String, en el ejemplo "1111". Es un número en binario puro sin bit de signo, por tanto es el 15 en decimal
- y la base en la que se tiene que interpretar, "2" para nuestro objetivo

```
class Unidad1{
    public static void main(String[] args){
        int x = Integer.parseInt( "1111",2);
        System.out.println("En base 10 \"1111\" es: "+x);
    }
}
```

### **Ejercicio U1\_B6\_E8:**

A la operación "and" también se le llama multiplicación lógica. Observa la diferencia entre multiplicación lógica y aritmética

```
class Unidad1{
    public static void main(String[] args){
        int x = 4;
        int y =5;
        System.out.println("x:"+ x + " y:"+ y);
        System.out.println("Multiplicación lógica: "+ (x&y));
        System.out.println("Multiplicación aritmética: "+ (x*y));
    }
}
```

Mejora la salida utilizando `Integer.toBinaryString()` para obtener algo parecido a lo siguiente

```
E:\Programacion>java Unidad1
x:4 y:5
Multiplicación lógica: 100 and 101 = 100
Multiplicación aritmética: 100 * 101 = 10100
E:\Programacion>
```

### Trabajando a nivel de bit con máscara.

Al trabajar a nivel de bit al segundo operando se le suele llamar "máscara" cuando lo que pretende es modificar u observar de alguna manera el valor del primer operando. Por ejemplo con el operador & podemos aplicar una máscara al primer operando para:

- determinar el estado de uno o varios bits
- extraer bits
- apagar un bit

y con el operado | y máscara podemos

- encender bits
- copiar bits si se combina | con el operador desplazamiento.

Veremos los ejemplos más sencillos que son "encender" y "apagar" un bit

### Ejercicio U1\_B6\_E9: poner un bit a 1 "encender un bit"

numeroAModificarBit | máscara

La máscara será un número cuyos bits son todos 0 excepto el de la posición que queremos colocar un 1 en el número a modificar

Ejemplo: queremos "encender" en 10000001 el 6º bit a 1

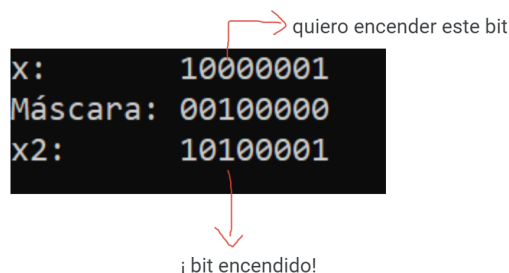
Lo de "6º bit" se refiere empezando por la derecha, también podíamos decir el bit 5 si pensamos que el primer bit por la derecha es el 0

10000001 | 00100000 = 10100001

Cómo estamos utilizando int el ejemplo anterior realmente deberíamos escribirlo usando 32 bits complemento a 2, no en binario puro pero lo hacemos así para abreviar

fíjate que en el segundo operando ponemos todos los bit a cero menos el bit 6º y esto lo podemos conseguir bien escribiendo el número entero correspondiente(32 en el ejemplo) o mejor, generando dicho número o con la instrucción 1<<5 (si desplazo 00...00000001 5 veces pongo el 1 en el 6º bit)

Se pide: demostrar lo anterior en un ejemplo java



```
x:      10000001
Máscara: 00100000
x2:      10100001
```

quiero encender este bit

¡ bit encendido!

### Ejercicio U1\_B6\_E10: colocar a 0 un bit "apagar un bit".

Por ejemplo vamos a poner a 0 el 3ºbit de 01010101

01010101 & 11111011 == 01010001

fíjate que en el segundo operando ponemos todos a unos menos el 3º bit. Este segundo operando en java debemos escribirlo como literal 0B11111011 o simplemente generándolo con  $\sim(1<<2)$   
 Demuestra lo anterior en un ejemplo java

## Operadores de asignación

Se puede combinar el operador de asignación con otros operadores para obtener un efecto idéntico al de aplicar los dos operadores por separado.

=      Asignación  
 +=     Suma y asignación  
 -=     Resta y asignación  
 \*=     Producto y asignación  
 /=     División y asignación  
 %=     Resto de la división entera y asignación  
 <<=    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

Ejemplo:

```
class Unidad1{
    public static void main(String[] args){

        int x1=0;
        x1=x1+3;
        System.out.println("x1:"+x1);

        int x2=0;
        x2+=3;
        System.out.println("x2:"+x2);

    }
}
```

## Ejercicio U1\_B6\_E11:

De forma similar al ejemplo comprueba el funcionamiento de %=, >>= y |= produciendo la siguiente salida

```
E:\Programacion>javac Unidad1.java
E:\Programacion>java Unidad1
Valor inicial de i: 10
Valor de i tras i%=3: 1
Valor de i tras i>>=1: 0
Valor de i tras i|=1: 1
E:\Programacion>
```