

OPERADORES

http://manuais.iessanclemente.net/index.php/Elementos_esenciais_da_linguaxe_Java#Operadores

<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

En este boletín analizamos los operadores más básicos y habituales. Dejamos los operadores “menos habituales” para el siguiente boletín.

Operadores aritméticos binarios

Operator	Description
+	Additive operator (also used for String concatenation)
-	Subtraction operator
*	Multiplication operator
/	Division operator
%	Remainder operator

Funcionan como los operadores matemáticos. Te puede resultar novedoso el operador % llamado “el operador módulo” o “operador resto”, que también tiene su equivalente matemático.

Ejemplo: probar los operadores aritméticos básicos

```
class Unidad1{
    public static void main(String[] args){
        int opIzquierdo=4;
        int opDerecho=2;
        int result;

        result=opIzquierdo+opDerecho;
        System.out.println("Ejemplo operador +: " + opIzquierdo + "+" + opDerecho + "=" + result);
        result=opIzquierdo-opDerecho;
        System.out.println("Ejemplo operador -: " + opIzquierdo + "-" + opDerecho + "=" + result);
        result=opIzquierdo*opDerecho;
        System.out.println("Ejemplo operador *: " + opIzquierdo + "*" + opDerecho + "=" + result);
        result=opIzquierdo/opDerecho;
        System.out.println("Ejemplo operador /: " + opIzquierdo + "/" + opDerecho + "=" + result);
        result=opIzquierdo%opDerecho;
        System.out.println("Ejemplo operador %: " + opIzquierdo + "%" + opDerecho + "=" + result);
    }
}
```

Ejemplo: Demostramos que si los dos operandos son enteros el operador / realiza la división entera. Si uno o los dos operadores son reales realiza la división real.

```
class Unidad1{
    public static void main(String[] args){
        System.out.println("4/2 =" + 4/2);
        System.out.println("4.0/2 =" + 4.0/2);
        System.out.println("4/2.0 =" + 4/2.0);
        System.out.println("5/2 =" + 5/2);
        System.out.println("5.0/2 =" + 5.0/2);
    }
}
```

```

    }
    System.out.println("5/2.0 =" + 5/2.0);
}

```

Ejercicio U1_B6_E1

Calcula la IMC de una persona cuyo peso y altura los tenemos almacenados en sus respectivas variables. La fórmula que calcula el IMC es:

$$IMC = \frac{peso(kg)}{altura^2(m)}$$

El programa imprime el IMC del peso y altura almacenados en una tabla de valores IMC. Un ejemplo de salida:

```

L:\Programacion>java Unidad1
peso: 71.5
altura: 1.67
IMC: 25.637348

```

```

TABLA IMC
Delgado: <18.5
Normal: entre 18.5 y 24.9
Sobrepeso: entre 25 y 29.9
Obeso: 30 o más

```

```

L:\Programacion>

```

División por cero

Matemáticamente una división por cero genera una indeterminación o indefinición ya que "no se puede repartir entre cero". En informática a nivel máquina las divisiones se calculan con algoritmos de restas sucesivas que con un divisor cero provocaría un bucle infinito, por tanto, para evitar que la máquina se cuelgue, el procesador no ejecuta el algoritmo y envía un aviso que llamamos "error lógico". Un poco más extensamente explicado en:

https://es.wikipedia.org/wiki/Divisi%C3%B3n_por_cero

Por motivos de estándares y algoritmos en computación "se comunica" el error lógico de dividir por cero de dos formas distintas según la aritmética empleada.

División por cero en aritmética entera.

Se genera una excepción. En el ejemplo la ejecución ya se detiene en la primera instrucción

```

class Unidad1 {
    public static void main(String[] args) {
        System.out.println(2/ 0);
        System.out.println("esto no se imprime");
    }
}

```

comprueba:

- la ejecución se para al hacer 4/0

- que el error no salta al compilar(javac), si no al ejecutar(java).
- si comentamos la primera división por cero observo que el programa “casca” en la siguiente división por 0.

División por cero en aritmética flotante.

El resultado es un valor especial “infinito” o NaN (not is a number)

```
class Unidad1 {
    public static void main(String[] args) {
        double x= 4.5/0;
        System.out.println(x);
        System.out.println(2.0 / 0);
        System.out.println(2.0 / 0.0);
        System.out.println(-2.0 / 0);
    }
}
```

Conclusión: si no queremos que nuestro programa se pare o funcione mal no podemos dividir por cero de forma que en toda división antes de ejecutarla hay que comprobar que el divisor no vale cero

El operador módulo.

Con el operador módulo obtenemos el resto de la división entera. El operador módulo en java también es aplicable con operandos float/double a los que se le aplica una suerte de división entera pero para simplificar nos restringimos a usarlo con operandos enteros.

Recuerda la división entera

$$D = d \cdot c + r$$

Dividendo		Divisor
<div style="border: 1px solid black; padding: 2px;">Resto</div>		Cociente

17		5	
<div style="border: 1px solid black; padding: 2px;">2</div>		3	17 = 5 · 3 + 2

Ejemplo: demostramos que el operador módulo devuelve el resto de la división entera.

```
class Unidad1{
    public static void main(String[] args){
        System.out.println("10%1="+ 10%1);
        System.out.println("10%2="+ 10%2);
        System.out.println("10%3="+ 10%3);
        System.out.println("10%4="+ 10%4);
        System.out.println("10%5="+ 10%5);
        System.out.println("10%6="+ 10%6);
        System.out.println("10%7="+ 10%7);
        System.out.println("10%8="+ 10%8);
        System.out.println("10%9="+ 10%9);
        System.out.println("10%10="+ 10%10);
        System.out.println("10%11="+ 10%11);
        System.out.println("10%12="+ 10%12);
    }
}
```

Ejemplo: ino se puede dividir por 0!

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

```

        //a=10/0;
        //a=10%0;
    }
}

```

Observa en el siguiente ejemplo el carácter "circular" del operador módulo

```

class Unidad1 {
    public static void main(String[] args) {
        System.out.println(0 % 3);
        System.out.println(1 % 3);
        System.out.println(2 % 3);
        System.out.println(3 % 3);
        System.out.println(4 % 3);
        System.out.println(5 % 3);
        System.out.println(6 % 3);
        System.out.println("etc...");
    }
}

```

con %2 los restos son siempre 0 y 1

con %3 los restos son siempre 0,1 y 2

con %4 los restos son siempre 0,1,2 y 3

etc...

Ejercicio U1_B6_E2. Dados unos segundos iniciales (500000 en el ejemplo) los distribuimos en días, horas, minutos y segundos como en el ejemplo que sigue

```

C:\Users\niinjashyper\programacion>java Unidad1
500000 segundos = 5 días, 18 horas 53 minutos y 20 segundos
C:\Users\niinjashyper\programacion>

```

Operadores unarios

Solo requieren un operador. Son los operadores +, -, ++, -- y !

Ejemplo:

```

class Unidad1{
    public static void main(String[] args) {
        //el + unario no es necesario
        int result = +1;
        System.out.println(result);
        result=-9;
        System.out.println(result);
        result++;
        System.out.println(result);
        result--;
        System.out.println(result);

        result = -result;
        System.out.println(result);

        boolean soyElMejor = false;
        System.out.println(soyElMejor);
        boolean soyUnFracasado=!soyElMejor;
        System.out.println(soyUnFracasado);
    }
}

```

los operadores unarios ++(incremento) --(decremento) y !(negación lógica) los estudiamos a continuación

Operadores de incremento/decremento

Pueden aplicarse de dos formas: prefijo(antes del operando) y postfijo (después del operando). Por ejemplo, incrementamos la variable result

```
++result; //modo prefijo  
result++; //modo postfijo
```

y similarmente se puede decrementar

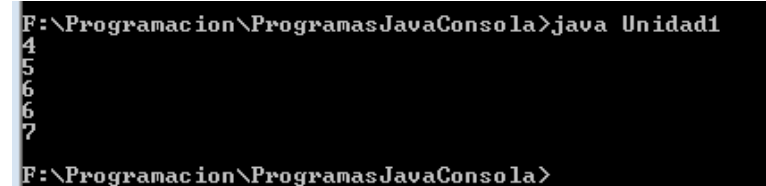
```
--result; //modo prefijo  
result--; //modo postfijo
```

```
++result; //modo prefijo  
result++; //modo postfijo
```

En ambos caso el valor de result se incrementa en uno.

Si lo único que hacemos es incrementar/decrementar un operador el efecto del modo prefijo y postfijo es el mismo. La diferencia entre uno y otro se puede producir cuando el incremento/decremento se hace dentro de una expresión más compleja.

```
class Unidad1{  
    public static void main(String[] args){  
        int i = 3;  
        i++;//equivalente a ++i  
        System.out.println(i);  
        ++i;//equivalente a i++  
        System.out.println(i);  
        System.out.println(++i); //NO equivalente a i++  
        System.out.println(i++); //NO equivalente a ++i  
        System.out.println(i);  
    }  
}
```



```
F:\Programacion\ProgramasJavaConsola>java Unidad1  
4  
5  
6  
7  
F:\Programacion\ProgramasJavaConsola>
```

Observa que justo antes de ejecutarse la instrucción

```
System.out.println(++i);
```

i vale 5, luego, cuando se ejecuta `System.out.println(++i);` ocurren dos cosas y por este orden:

1. Debido al modo prefijo, antes de utilizar el valor de i se incrementa y pasa a valer 6
2. Se imprime el valor de i que es actualmente 6

A continuación se ejecuta

```
System.out.println(i++);
```

y Ocurre:

1. Debido al modo postfijo, antes de incrementar el valor de i se calcula la expresión con el valor "viejo" de i. La expresión consiste en una única variable de la que se toma su valor que es 6 y se imprime.
2. Se incrementa el valor de i pasando a valer 7

Es importante observar que igual que `2+3` es una expresión y por tanto al evaluarse devuelve un valor, `++x` o `x++` también son expresiones y por lo tanto se evalúan y devuelven un valor. Lo peculiar de estas expresiones es que además de devolver un valor realizan un incremento en la variable y según sea el operador prefijo o postfijo el valor devuelto se calcula con el valor incrementado o sin incrementar. Se entiende esto mejor en jshell

con prefijo el valor se calcula con valor de x incrementado. Observa en el jshell que si escribo `++x` lo que me devuelve es el valor de la expresión `++x`

```
jshell> int x=5
x ==> 5

jshell> ++x
$6 ==> 6

jshell> x
x ==> 6

jshell>
```

Lo mismo en modo postfijo: ahora el jshell me devuelve un valor de la expresión `x++` que no coincide con el valor de `++x` del ejemplo anterior porque para evaluar la expresión ahora NO SE INCREMENTÓ PREVIAMENTE el valor de `x`.

```
jshell> int x=5
x ==> 5

jshell> x++
$9 ==> 5

jshell> x
x ==> 6

jshell>
```

un ejemplo lioso:
esto hace lo que esperábamos

```
jshell> int x=10
x ==> 10

jshell> x=++x
x ==> 11

jshell>
```

pero esto ...

```
jshell> x=10
x ==> 10

jshell> x=x++
x ==> 10

jshell> x=x++
x ==> 10

jshell>
```

`x++` es una expresión que hace dos cosas:

1. devuelve un valor, que el `x` antes de incrementar, o sea, 10. Este valor que devuelve la expresión imagina que se almacena temporalmente en alguna zona de la memoria.
2. Luego efectivamente incrementa `x` que pasa a valer 11

Hay un instante que `x` vale 11, pero queda pendiente resolver la asignación del operador `=`, este operador asigna el valor de la expresión de la derecha, a la variable de la izquierda. ¡El valor de la expresión era "10" y por tanto ya no vale 11 `x`!

Ejercicio U1_B6_E3

Escribe código equivalente al siguiente sin utilizar el operador unario `++`, es decir incrementando de la forma `x=x+1`

```
class Unidad1{
    public static void main(String[] args){
        int x=3;
```

```

    int y;
    y=x++*2;
    System.out.println("x: " + x+ " y:"+y);
}
}

```

Ejercicio U1_B6_E4

Escribe código equivalente al siguiente sin utilizar el operador ++

```

class Unidad1{
    public static void main(String[] args){
        int x=3;
        int y;
        y=++x*2;
        System.out.println("x: " + x+ " y:"+y);
    }
}

```

sobre el operador --

El operador -- funciona de forma similar al ++ pero decrementando en lugar de incrementando

Sobre el operador + con String.

El operador + se utiliza para hacer sumas aritméticas, pero además, cuando los operandos son cadenas de caracteres, el operador + funciona "concatenando cadenas".

Ejecuta el siguiente ejemplo en el que + concatena strings

```

class Unidad1{
    public static void main(String[] args){
        String miString;
        miString="Hola" + " y " + "adios";
        System.out.println(miString);
        //o haciendo directamente la concatenación en el println
        System.out.println("Jamones" + " y " + "salchichones");
        //o combinando en println constantes y variables
        System.out.println("Tengo dos cosas que decirte: " + miString);
        //y muchas otras combinaciones ...
        String advertencia="Por segunda vez, tengo dos cosas que decirte: " + miString;
        System.out.println(advertencia);
    }
}

```

El operador + suma aritméticamente en aquellas expresiones en que sus operandos son números

Por ejemplo: 2+5

y concatena strings si sus operandos son strings (recuerda que un string es un conjunto de caracteres encerrado entre comillas dobles)

Por ejemplo: "2"+"5"

Ejecuta el siguiente ejemplo para salir de toda duda:

```

class Unidad1{
    public static void main(String[] args){
        int miNumero=2+5;
        String miString="2"+"5";
        System.out.println(miNumero);
        System.out.println(miString);
    }
}

```

¿Es posible mezclar "+" con números y strings?, y si es posible, ¿Cuál es su efecto?.

Al detectar un operando string y otro número, java convierte automáticamente el número en un string y concatena ambos operadores (no hace operación aritmética).

Ejecuta el siguiente ejemplo:

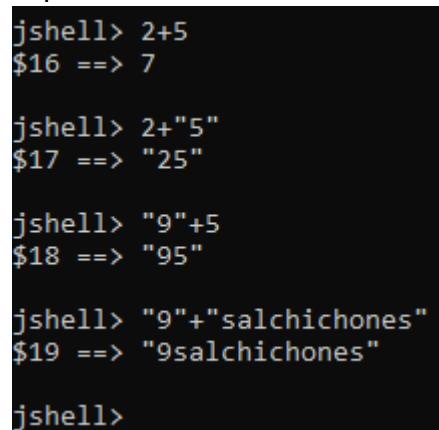
```
class Unidad1{
    public static void main(String[] args){
        //Como java convierte 2+"5" en un string "25" no se puede asignar un string a un entero
        // int miNumero=2+"5";
        String miString1=2+"5";
        System.out.println(miString1);
        String miString2="9"+5;
        System.out.println(miString2);
        String miString3= 9+ " salchichones";
        System.out.println(miString3);
    }
}
```

Observa que con una expresión como 2+"5", ya que java detecta un operador string, convierte el numerico a string y concatena ambos obteniendo el nuevo string "25"

Obtendremos el mismo resultado, ya que las consideraciones son las mismas, si hubiéramos escrito las expresiones directamente en el println(), ya que el println recibiría un string

```
class Unidad1{
    public static void main(String[] args){
        System.out.println(2+"5");
        System.out.println("9"+5);
        System.out.println(9+"salchichones");
    }
}
```

podemos probar lo anterior también en jshell, al ver inmediatamente el valor que devuelve la expresión se entiende todo de maravilla



```
jshell> 2+5
$16 ==> 7

jshell> 2+"5"
$17 ==> "25"

jshell> "9"+5
$18 ==> "95"

jshell> "9"+"salchichones"
$19 ==> "9salchichones"

jshell>
```

Aunque no tiene que ver exactamente con lo que estamos viendo, observar que a println() le puedo pasar una diversidad de argumentos, no solo string, por ejemplo un número o un carácter

```
System.out.println(2);
System.out.println(2+3);
System.out.println('a');
```

En estos casos a println() "no llega" un string pero funciona correctamente ya que veremos en unidades posteriores que println() es un método "sobrecargado" que admite una diversidad de tipos de argumentos.

PRECEDENCIA OPERADORES

<https://riptutorial.com/es/java/example/9207/precedencia-del-operador>

El concepto es el mismo que en matemáticas, y la conclusión la misma, EN CASO DE DUDA UTILIZA PARÉNTESIS.

La precedencia indica cual es el orden de ejecución de los operadores cuando existen varios. Por ejemplo en la expresión

$$20-16/2+2*2$$

1º: $16/2 \Rightarrow 20-8+2*2$ (el / tiene el mismo nivel de precedencia que el * pero está más a la izquierda)

2º: $2*2 \Rightarrow 20-8+4$

3º: $20-8 \Rightarrow 12+4$ (el - tiene el mismo nivel de precedencia que el + pero está más a la izquierda)

4º: $12+4 \Rightarrow 16$

```
jshell> 20-16/2+2*2
$4 ==> 16
jshell>
```

Ejemplo: la multiplicación tiene más precedencia que la suma

Ejemplo:

modificando el ejercicio de IMC, comprueba que diferente son los resultados

```
float imc=peso/(altura*altura);
```

de

```
float imc=peso/altura*altura;
```

si te fijas esta segunda versión además no tiene sentido ya que matemáticamente al simplificar ocurre que

```
imc=peso
```

en java más o menos igual con aritmética flotante (salvo con aquellas cantidades que acarreen algo de imprecisión). Si los operandos son enteros, la división es entera y entonces obtenemos otro resultado

```
jshell> 10.0/3.0*3.0
$8 ==> 10.0
jshell> 10/3*3
$9 ==> 9
jshell>
```

Expresiones que mezclan strings y enteros que contienen el operador +

Sabemos que el operador + tiene las funciones de sumar números y concatenar Strings. Si mezclamos números y Strings con varios + la precedencia de izquierda a derecha es muy importante.

```
class Unidad1{
    public static void main(String[] args){
        System.out.println(2+3+"hola");
        System.out.println("hola"+2+3);
    }
}
```

Y observa el resultado

```
5hola
hola23
```

Es decir, java obtuvo:

- de la expresión `2+3+"hola"` el string `5hola`. Parece que sumo 2 y 3, convierto 5 a string y lo concatené a hola
- y de expresión `"hola"+2+3`, el string `hola23`. Es decir no hizo suma alguna en este caso

¿Por qué ocurre esto?

Tiene que ver con "precedencia de operadores". Al haber varios + su precedencia se aplica de izquierda a derecha, quiere esto decir que:

- `2+3+"hola"` se interpreta como `(2+3)+"hola"`. Por lo tanto el primer + que se evalúa es `(2+3)`, como ambos operadores son enteros se interpreta como un + aritmético y genera el valor 5. Finalmente nos quedaría evaluar `5+"hola"`, como uno de los operandos es un string el primer operando se promociona a string y se realiza una concatenación.
- `"hola"+2+3` se interpreta como `("hola"+2)+3`. Por tanto se evalúa el primer + y como uno de los operandos es un string se produce la concatenación `"hola2"`. Queda por evaluar `"hola2"+3`, de nuevo un operador es un string y se produce concatenación: `"hola23"`.

Ejemplo: La instrucción `System.out.println(4+1+"-"+4+1);`

¿Mostrará la cadena `41 - 41` o la cadena `5 -41`?

muestra `5- 41` ya que todos los operadores son + la precedencia es izquierda a derecha y ocurre:

1. se suma 4 y 1, ambos son numéricos y su resultado es 5. Nos quedaría la siguiente expresión "intermedia" camino al cálculo final `5+"-"+4+1`
2. ahora el + tiene un operador numérico(el 5) y otro string(el "-") y por tanto no va a funcionar aritméticamente si no que crea la cadena `"5-"`. Nos quedaría la siguiente expresión "intermedia" camino al cálculo final `"5-"+4+1`
3. Ahora siempre vamos a tener que hay un operador es string y ya siempre se hacen concatenaciones, el siguiente paso intermedio sería `"5-4"+1`
4. y finalmente al ejecutar el último cálculo nos queda `"5-41"`

Ejercicio U1_B6_E5

Comprueba si `++x*y` es equivalente o no a `(++x)*y`. Idem respecto a `++(x*y)`

Al probar estas situaciones debes también detectar un error de compilación en la última expresión

Se podrían buscar ejemplos todavía más rebuscados pero no merece la pena ya que son situaciones que se pueden simplificar con el uso de paréntesis. **En caso de duda usa paréntesis!**

OPERADORES LÓGICOS Y RELACIONALES

Los dos tienen en común que el resultado de la operación es true/false.

Operadores relacionales

Permiten comparar normalmente números y devuelven true o false. Similar a matemáticas.

Operador	Significado
==	igual
!=	distinto
>	mayor
>=	mayor o igual
<	menor
<=	menor o igual

Ejemplo operadores relacionales:

```

class Unidad1{
    public static void main(String[] args){
        int x=8;
        int y=5;
        boolean compara=(x<y);
        System.out.println("x<y es "+compara);
        compara=(x>y);
        System.out.println("x>y es "+compara);
        compara=(x==y);
        System.out.println("x==y es "+compara);
        compara=(x!=y);
        System.out.println("x!=y es "+compara);
        compara=(x<=y);
        System.out.println("x<=y es "+compara);
        compara=(x>=y);
        System.out.println("x>=y es "+compara);
    }
}

```

Si te fijas los operadores relacionales tienen la típica estructura

operando1 operador operando2, ej. 5<2

Observa que los valores de operando1 y operando2 tiene que tener sentido para la operación:

```
jshell> 5<8
$16 ==> true

jshell> true<true
Error:
bad operand types for binary operator '<'
first type: boolean
second type: boolean
true<true
^-----^

jshell> "hola">"adios"
Error:
bad operand types for binary operator '>'
first type: java.lang.String
second type: java.lang.String
"hola">"adios"
```

true<true no tiene sentido. Una expresión con un operador relacional devuelve true/false pero sus operandos no pueden valer true/false

"hola">"adios" si podría tener sentido(orden alfabético) pero no se puede hacer así en Java. Los operandos de los operadores relacionales tienen que ser de tipo primitivo y String es un objeto.

Los operandos pueden ser expresiones complejas pero finalmente esa expresión se evalúa y tiene un valor de tipo primitivo.

Ejemplo: 'A' +1 es una expresión int que vale 66 (el ASCII de A es 65)

```
jshell> 'A'+1>65
$21 ==> true
```

```
jshell>
```

```
jshell> 4*5>15+4
$20 ==> true
```

```
jshell>
```

Operadores lógicos

Permiten relacionar varias expresiones lógicas, para obtener finalmente un valor lógico.

El operador and (&&)

el operador and, en java &&, se conoce también por Sí lógico, Y lógico y multiplicación lógica. Se utiliza para reflejar en el código la siguiente situación del mundo real

"Tengo dos afirmaciones que pueden ser ciertas o no. Quiero que en su conjunto, todo sea cierto si las dos afirmaciones son ciertas". Ejemplo

Los perros tienen columna vertebral y (and) se reproducen true and true = true

Los mejillones tienen columna vertebral y(and) se reproducen false and true= false

El operador or (||)

el operador or, en java ||, se conoce también como O lógico y suma lógica.

Se utiliza para verificar que de dos afirmaciones al menos una es cierta, puede ser cualquiera de ellas o las dos

Los perros tienen columna vertebral o (or) se reproducen true and true = true (es cierto)

Los mejillones tienen columna vertebral o (or) se reproducen false and true = true (es cierto)

El operador not (!)

En java se escribe ! . Se conoce también como negación lógica.

Sirve para controlar si lo inverso o negado de algo es cierto

not (los mejillones tienen columna vertebral) not(false) =true es cierto que los mejillones no tienen columna vertebral!

Tabla de funcionamiento de and y or

Operando 1	Operando 2	and	or
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Tabla de funcionamiento de not

Operando	not Operando
True	False
False	True

Ejemplo de operadores lógicos:

```
jshell> true && true
$22 ==> true

jshell> true && false
$23 ==> false

jshell> boolean a= true
a ==> true

jshell> boolean b=false
b ==> false

jshell> a&&b
$26 ==> false

jshell> a!=b
$27 ==> true

jshell> a==b
$28 ==> false
```

observa que false==false tiene sentido lógico pero qué false<=false no

```
jshell> false==false
$1 ==> true

jshell> false<=false
Error:
  bad operand types for binary operator '<='
    first type:  boolean
    second type: boolean
false<=false
^-----^

jshell>
```

Por último por descuido, es típico confundir
 = operador de asignación
 == operador lógico
 en el último caso me confundí en true=false

```
jshell> boolean compara
compara ==> false

jshell> boolean a=true
a ==> true

jshell> boolean b=false
b ==> false

jshell> compara=true==false
compara ==> false

jshell> compara=true=false
Error:
  unexpected type
    required: variable
    found:    value
compara=true=false
^--^

jshell>
```

pero ojo por que hay en casos que el compilador no detecta error

```
jshell> compara=a=b
compara ==> false

jshell>
```

mi intención era comparar si a y b son iguales y el resultado dárselo a compara pero hice otra cosa:
 a=b => asignar el valor de la variable b a la variable a
 ese valor se asigna a compara
 observa que el operador de asignación tiene una asociatividad por la derecha en lugar de por la izquierda

BLUEJ. Podemos probar expresiones relativas a este boletín en codepad

```

2+3
5 (int)
2+3.0
5.0 (double)
2+3.0f
5.0 (float)
2+"hola"
Error: unclosed string literal
2+"hola"
"2hola" (String)
5+1+"hola"+5+1
"6hola51" (String)
2|1
3 (int)
2||1
Error: bad operand types for binary operator '|'
first type: int
second type: int
(2<3)|(1<2)
true (boolean)

```

Con lo que vimos hasta ahora no puedes entender

```

2|1
3 (int)

```

aquí "|" esta funcionando como operador "or" a nivel de bit lo que veremos en el próximo boletín.

Ejercicio U1_B6_E6

prueba las siguientes expresiones `true&&true` `true&&false` `false&&true` `false&&false`, de forma que nuestro programa sea capaz de imprimir la tabla del AND con un aspecto similar al siguiente:

```

C:\Users\Pilt>java Unidad1
TABLA Operador AND

x      y      resultado
-----
true   true   true
true   false  false
false  true    false
false  false  false

```

lógicamente, la columna resultado debes calcularla, no hagas directamente al impresión del resultado

Ejercicio U1_B6_E7

Crea un programa que demuestre que la instrucción

```
int z = x++%5;
```

produce un valor de Z diferente que la instrucción

```
int z = ++x%5;
```

podría tener una salida similar al siguiente para un valor de x inicial de 10

```

C:\Users\donlo>java Unidad1
Inicialmente x vale: 10
Despues de x++%5, z vale:0 y x vale:11
Despues de ++x%5, z vale:1 y x vale:11
C:\Users\donlo>

```