

RESUMEN



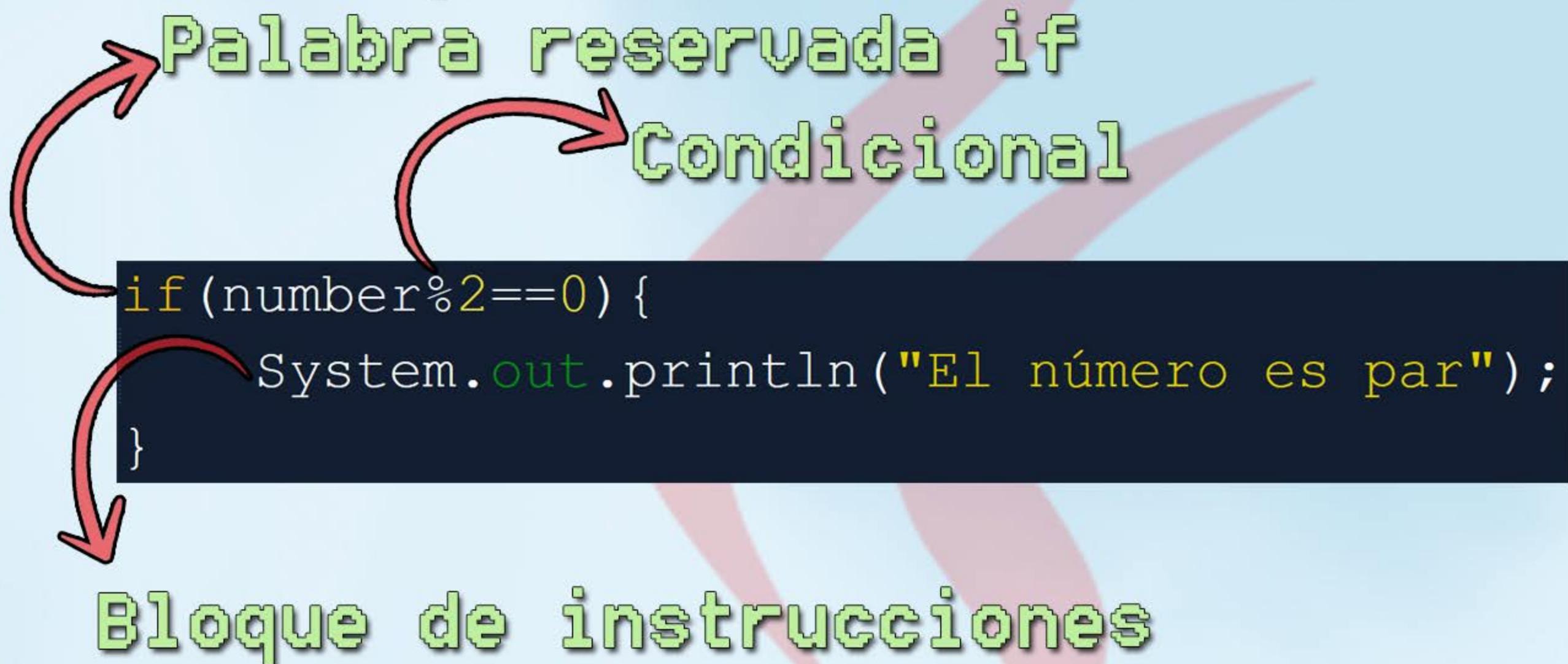
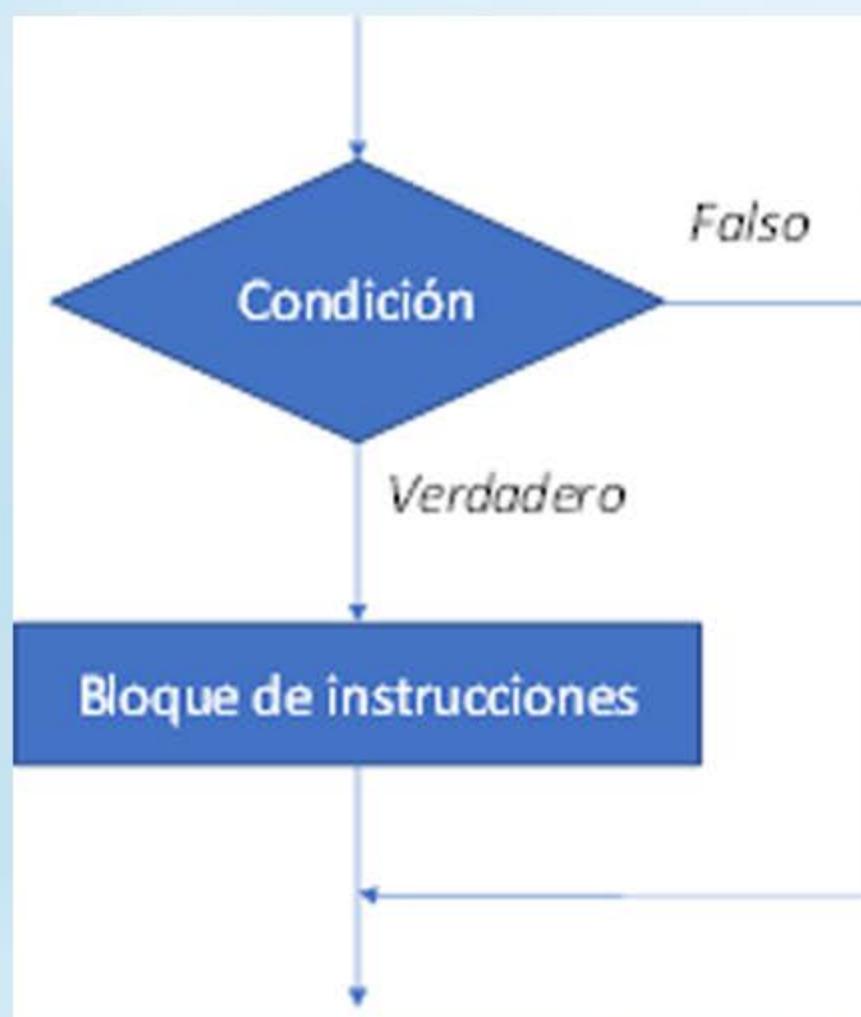
## MÓDULO 2: CONTROL DE FLUJO

*(Flow control structures)*

# IF - ELSE

**BLOQUE IF:** LA ESTRUCTURA EVALUA UNA CONDICIÓN O CONDICIONES Y EN CASO DE QUE SEA VERDADERA (TRUE) EJECUTA UN BLOQUE DE INSTRUCCIONES.

**Sintaxis:**



**ELSE:** PODEMOS ACOMPAÑAR EL IF DE UN BLOQUE ELSE. LAS INSTRUCCIONES DE ESTE BLOQUE SE EJECUTAN SI EL CONDICIONAL DEL “IF” DEVUELVE FALSE .

**Sintaxis:**

```
if (number%2==0) {  
    System.out.println("El número es par");  
}  
else {  
    System.out.println("El número es impar");  
}
```

No podemos poner instrucciones entre el bloque “if” y el “else”. Si el bloque solo tiene una instrucción podemos prescindir de las llaves.

# IF - ELSE II

**BLOQUE ELSE IF:** ADICIONALMENTE PODEMOS INCLUIR “PREGUNTAS” ADICIONALES UTILIZANDO BLOQUES **else if**. PODEMOS USAR TANTOS COMO NECESITEMOS.

## Sintaxis:

```
if (diaDeLaSemana.equals("Lunes")) {  
    // Instrucciones  
}  
  
else if (diaDeLaSemana.equals("Martes")) {  
    // Instrucciones  
}  
  
else if (diaDeLaSemana.equals("Miércoles")) {  
    // Instrucciones  
}  
  
else if (diaDeLaSemana.equals("Jueves")) {  
    // Instrucciones  
}  
  
else if (diaDeLaSemana.equals("Viernes")) {  
    // Instrucciones  
}  
  
else if (diaDeLaSemana.equals("Sábado")) {  
    // Instrucciones  
}  
  
else{  
    // Instrucciones  
}
```

**Comportamiento:**  
USANDO VARIOS BLOQUES **if**, CADA CONDICIONAL SE EVALUA POR SEPARADO. DE ESTA MANERA EVALUAMOS SOLO SI EL BLOQUE ANTERIOR DEVOLVIO **false**.

# BUCLLES

**CONCEPTO:** SON ESTRUCTURAS CICLICAS, PERMITEN EJECUTAR REPETIDAS VECES (ITERAR) UNA SECUENCIA DE INSTRUCCIONES MIENTRAS UNA, O VARIAS, CONDICIONES SE CUMPLAN.

**TIPOS DE BUCLES:** EXISTEN CUATRO TIPOS DE BUCLES EN JAVA, `while`, `do-while`, `for` y `for-each`.

**Sintaxis:**

**Bucle while**

```
while (true) {  
    //Instrucciones  
}
```

**Bucle do-while**

```
do {  
    //Instrucciones  
} while (true);
```

**Bucle for**

```
for (int i = 0; i < 10; i++) {  
    //Instrucciones  
}
```

**Bucle for-each**

```
for (String texto : textArray) {  
    //Instrucciones  
}
```



Si la condición/condiciones en ningún momento se evalúan como false estaremos ante un bucle infinito. Esto es algo que debemos evitar porque bloquearía la ejecución de nuestro programa.

# BUCLE WHILE

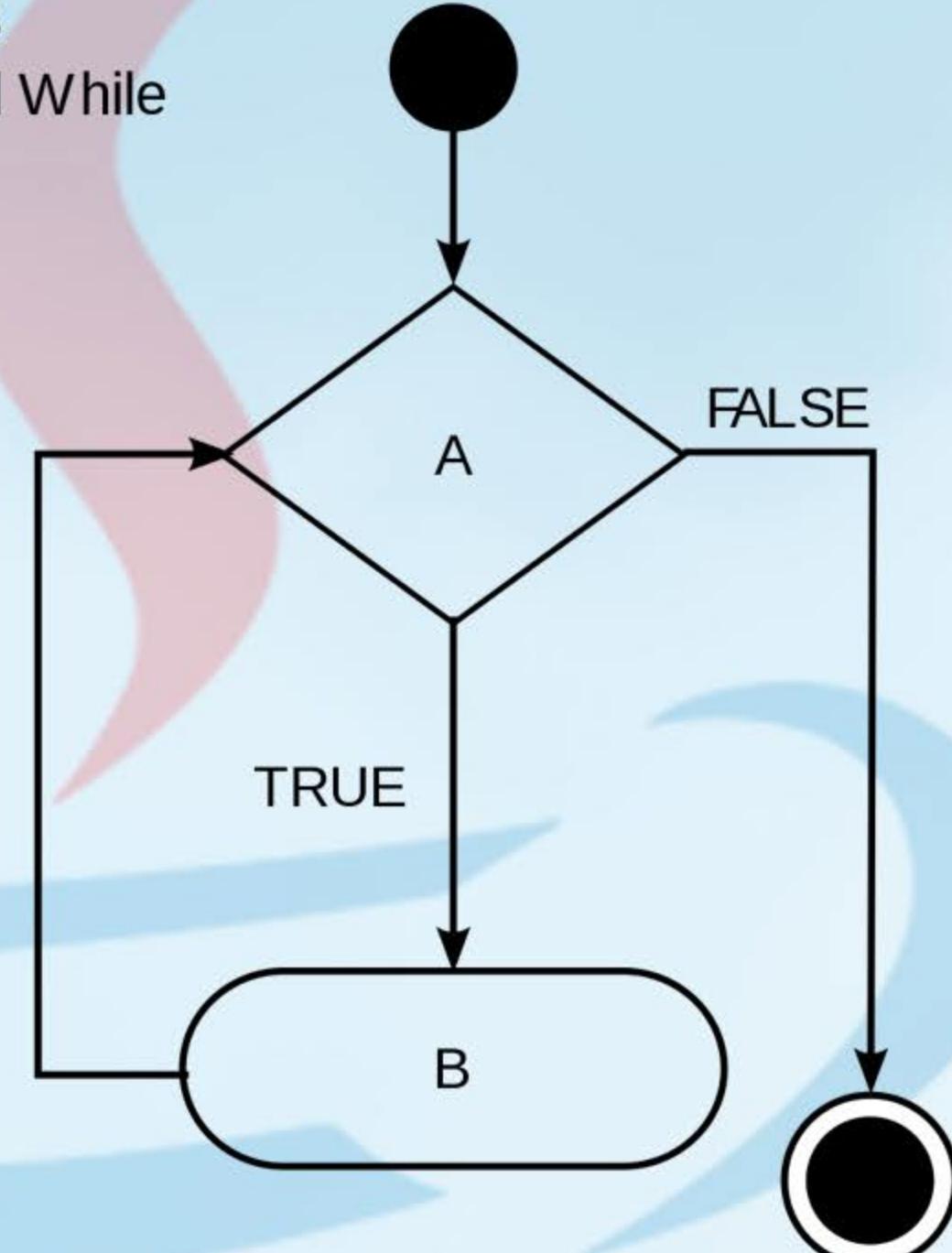
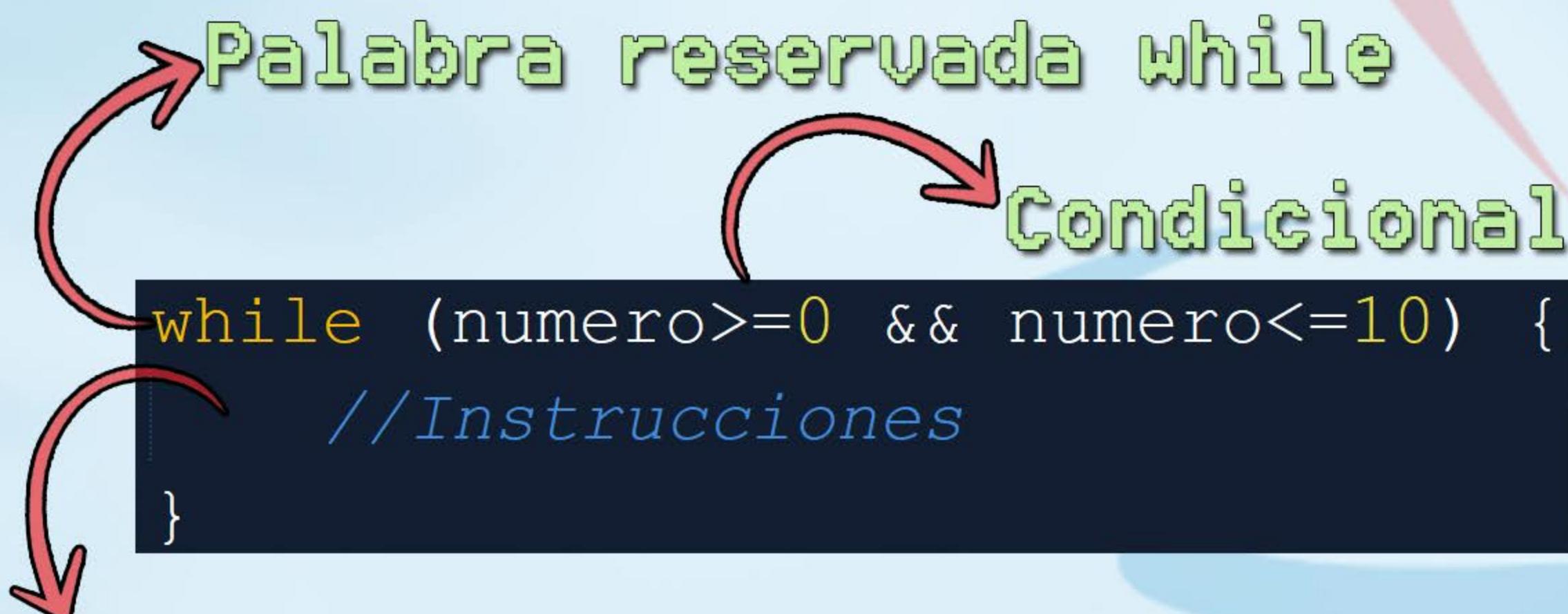
**CONCEPTO:** ES UN CICLO REPETITIVO BASADO EN LOS RESULTADO DE UNA EXPRESION LOGICA. EL PROPOSITO ES REPETIR UN BLOQUE DE CODIGO MIENTRAS UNA CONDICION SE MANTENGA **true**.

**Sintaxis en pseudocódigo:**

```
mientras condición hacer  
    instrucciones  
fin mientras
```

```
While (A= TRUE) Do  
    B  
End While
```

**Sintaxis:**



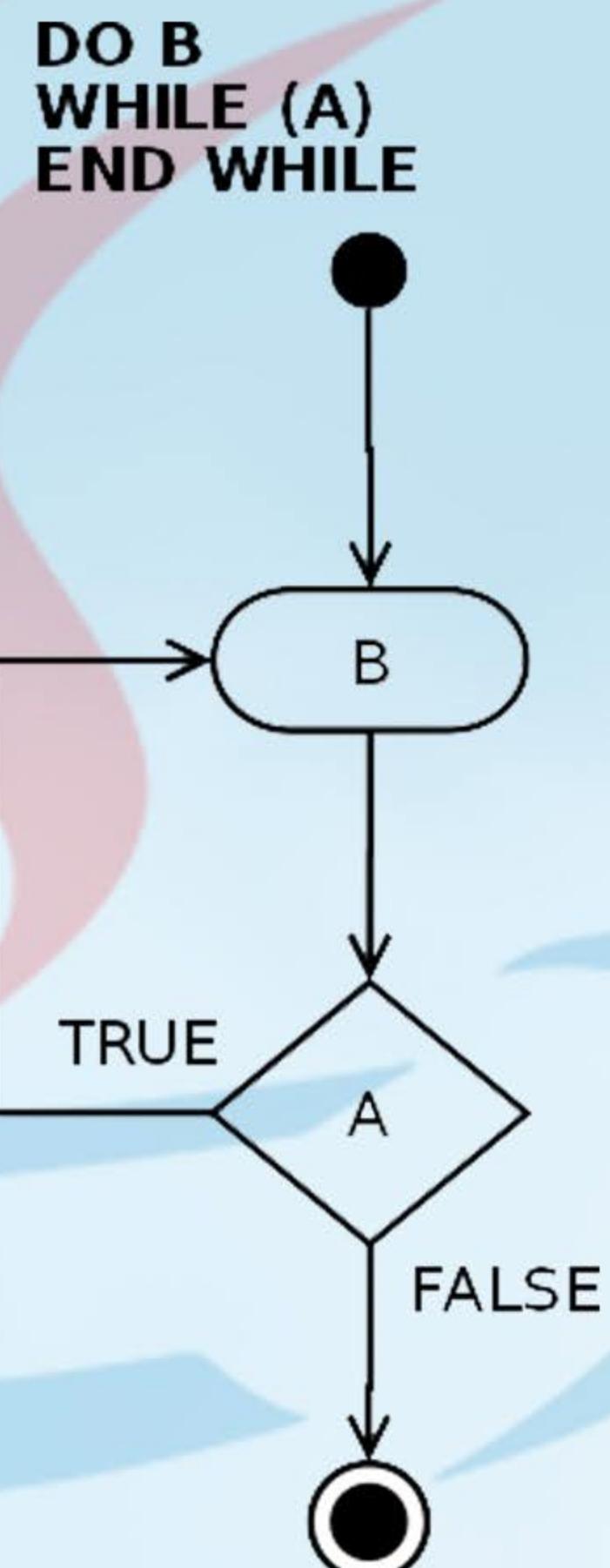
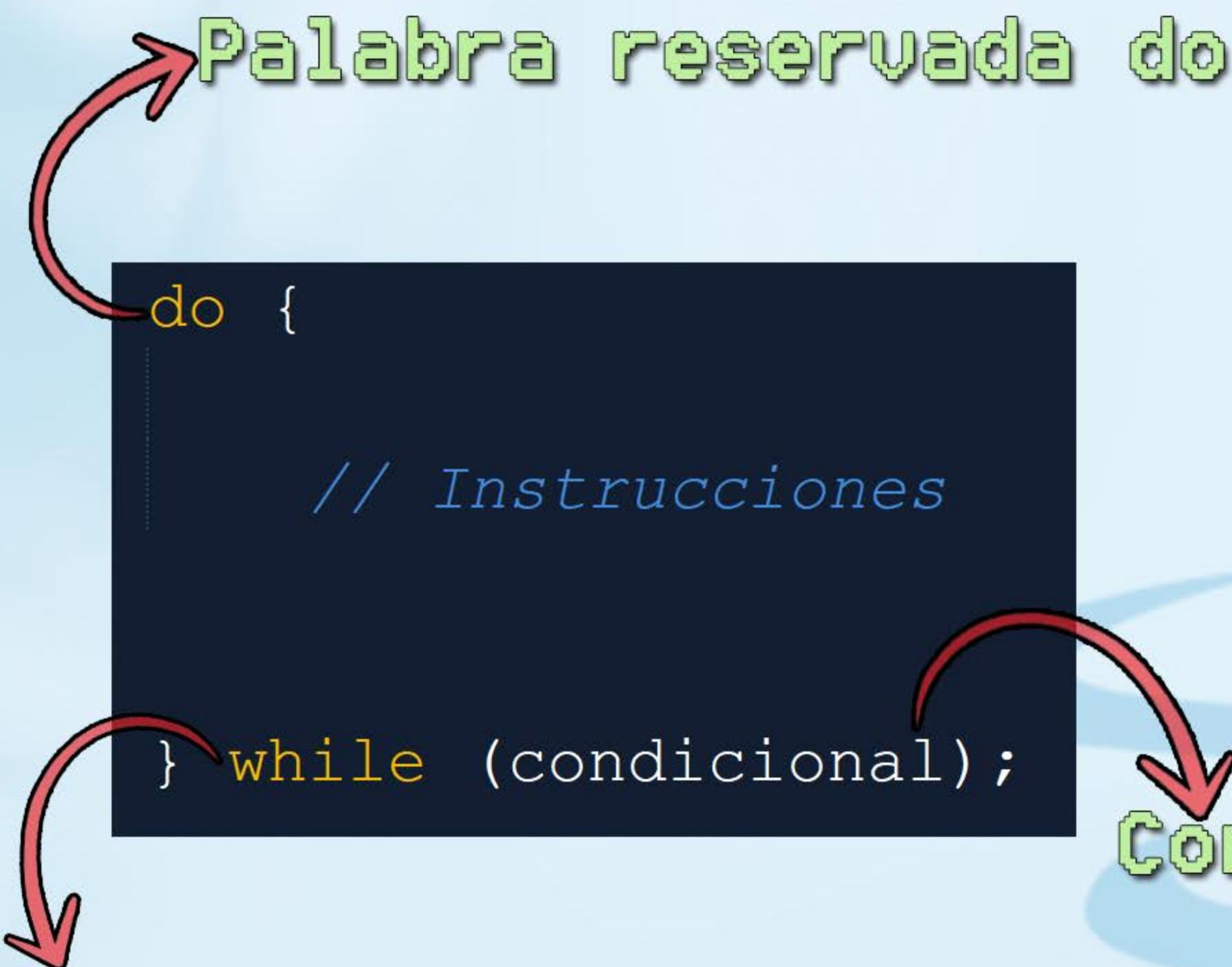
**Bloque de instrucciones**

**Funcionamiento:** La condición ha de ser una sentencia que devuelva un valor “booleano” (**true** o **false**). También puede contener una variable booleana y el valor de la expresión dependerá de su contenido.

# BUCLE DO-WHILE

**CONCEPTO:** SIMILAR AL BUCLE WHILE, PERO EN ESTE CASO LA CONDICIÓN SE EVALUA AL FINAL DEL BUCLE. NOS ASEGURA QUE LAS INSTRUCCIONES SE EJECUTAN AL MENOS UNA VEZ.

## Sintaxis:



## Consideraciones:

Tanto el while como el do-while no suelen ir asociados a recorrer un conjunto de datos (iterable). Es por ello que se dice de estos bucles que no conocemos cuando acaban, pues estan sujetos a decisiones del usuario.

# BUCLE FOR

**CONCEPTO:** ES UN BUCLE QUE UTILIZA UNA O VARIAS VARIABLES DE CONTROL (INDICE O ITERADOR) QUE NOS PERMITE MANEJAR LAS ITERACIONES. FOR CONSTA DE TRES BLOQUES DIFERENCIADOS:

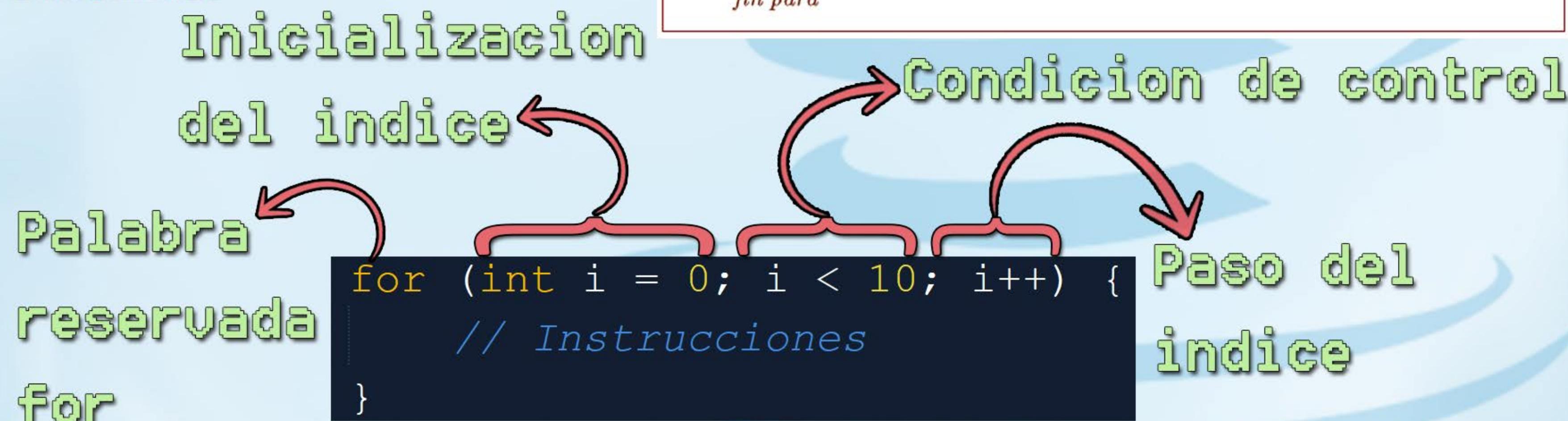
**Inicialización del indice:** Es la parte donde se especifica el valor inicial de la/s variable/s de control.

**Condición de control:** El valor final que pueden tomar la/s variable/s de control.

**Paso del indice:** Especificamos como va a afectar el fin del bucle a nuestra/s variable/s de control.

**Sintaxis en pseudocódigo:**

**Sintaxis:**



# BUCLE FOR II

## Sintaxis multiples indices:

```
for (int i = 0, j = 10; i < 10 && j > 0; i++, j--) {  
    // Instrucciones  
}
```

## Los bloques no son obligatorios:

### Sin bloque inicializador

```
int i = 0;  
for ( ; i < 100; i+=2) {  
    // Instrucciones  
}
```

Podemos declarar e inicializar la variable de control fuera del bucle.

### Sin bloque condición

```
int i = 0;  
for ( ; ; i+=2) {  
    // Instrucciones  
    if (i==100) break;  
}
```

Si prescindimos del bloque de la condición y/o del bloque en el que indicamos el paso que debe dar el índice, tendremos que manejar el bucle dentro del bloque de instrucciones (como se muestra en la imagen) o provocaremos un bucle infinito.

### Sin bloque incremento

```
int i = 0;  
for ( ; ; ) {  
    // Instrucciones  
    if (i==100) break;  
    i+=2;  
}
```

```
for ( ; ; ) { }
```

No hay error de compilación, pero es un bucle infinito.

# BUCLE FOR EACH

**CONCEPTO:** SE TRATA DE UNA SIMPLIFICACIÓN DEL BUCLE “`for`” PARA ITERAR SOBRE UN CONJUNTO DE DATOS (ARRAYS O COLECCIONES). CONSTA DE DOS BLOQUES:

**Declaración de variable:** Sirve para crear una variable que tomará el valor de cada uno de los elementos del iterable.

**Referencia al iterable:** Sirve para indicar que conjunto de datos (iterable) vamos a recorrer en el bucle.

**Sintaxis:** **Iterable**

```
String[] nombres = new String[] {"Ana", "Eva", "Sandra"};
```

Declaracion de variable

Referencia al iterable

```
for (String nombre : nombres) {  
    // Vuelta 1 -> nombre = "Ana"  
    // Vuelta 2 -> nombre = "Eva"  
    // Vuelta 3 -> nombre = "Sandra"  
}
```

Equivale a:

```
for (int i = 0; i < nombres.length; i++) {  
    // Vuelta 1 -> i = 0 -> nombres[i] = "Ana"  
    // Vuelta 2 -> i = 1 -> nombres[i] = "Eva"  
    // Vuelta 3 -> i = 2 -> nombres[i] = "Sandra"  
}
```

# BUCLES II

**BUALES ANIDADOS:** ANIDAR UN BUCLE SIGNIFICA METER UN BUCLE DENTRO DE OTRO. POR CADA CICLO DEL BUCLE “PADRE” EL BUCLE “HIJO” REALIZA TODOS SUS CICLOS.

## Sintaxis:

```
for (int i = 0; i < 3; i++) {  
    System.out.println("Hola");  
    for (int j = 0; j < 3; j++) {  
        System.out.println("Adios");  
    }  
}
```

Este ejemplo imprime por consola tres veces la palabra “Hola” y nueve veces la palabra “Adios”

**BREAK:** LA SENTENCIA “break” NOS PERMITE SALIR DE MANERA ABRUPTA DE UN BUCLE.

## Sintaxis:

```
long number = 0;  
while (number < 10) {  
    if (number == 5) {  
        break;  
    }  
    number++;  
}
```

Este bucle daría un total de diez vueltas en condiciones normales. Pero en este caso, al llegar al sexto ciclo, la variable number tendrá un valor de 5, cumpliendo el condicional del “if” y ejecutando la sentencia “break” que provocará que el bucle termine.

# BUCLES III

**CONTINUE:** SENTENCIA QUE SIRVE PARA FORZAR AL BUCLE A QUE COMIENCE EL SIGUIENTE CICLO AUNQUE QUEDEN LINEAS DE INSTRUCCION POR EJECUTAR DENTRO DEL BLOQUE.

## Sintaxis:

```
long number = 0;  
while (number < 10) {  
    number++;  
    if (number%2 == 0) {  
        continue;  
    }  
    System.out.println(number);  
}
```

Este bucle imprime solo los número que son impar. Cuando “number” sea igual a un número par, se mete dentro del bloque “if” y ejecuta la sentencia “continue” forzando el inicio del siguiente ciclo.

**LABEL:** PODEMOS USAR ETIQUETAS QUE PERMITEN IDENTIFICAR EL BUCLE. ESTO ES UTIL EN BUCLES ANIDADOS COMBINADO CON “break” y “continue”.

## Sintaxis:

```
BucleA:  
for (int i = 0; i < 10; i++) {  
    BucleB:  
    for (int j = 0; j < 10; j++) {  
        if (i < 5) continue BucleA;  
        else break BucleA;  
    }  
}
```

# OPERADORES

## ASIGNACION:

**ASIGNACION:** Permite asignar valor a una variable.

```
int numero = 5; String palabra = "word";
```

## ARITMETICOS: PERMITE REALIZAR CALCULOS.

**+** **SUMA:** Suma dos valores.

```
int n = 2 + 4; → n=6 int n2 = n + n; → n2=12
```

**-** **RESTA:** Resta dos valores.

```
int n = 2 - 4; → n=-2 int n2 = n - n; → n2=-4
```

**\*** **MULTIPLICACION:** Multiplica dos valores.

```
double n = 3.5 * 2; → n=9.0
```

**/** **DIVISION:** Divide un valor entre el otro.

```
long n = 100 / 2; → n=50L
```

**% RESTO:** Devuelve el resto (módulo) de un número dividido por el otro.

```
byte n = 100 % 2; → n=0
```

**+= ASIGNACION 2.0:** Realiza la operación sobre el valor actual.

```
num += 10;
```

```
num -= 10;
```

```
num *= 10;
```

```
num /= 10;
```

```
num %= 10;
```

Similar a  
Forma abreviada

```
num = num + 10;
```

```
num = num - 10;
```

```
num = num * 10;
```

```
num = num / 10;
```

```
num = num % 10;
```

# OPERADORES II

**RELACIONALES:** PERMITEN COMPARAR EXPRESIONES LOGICAS, DATOS NUMERICOS Y CARACTERES.

**== IGUAL QUE:** Comprueba si los datos son iguales.

`false == false` → **true**    `5 == 4` → **false**

**!= DISTINTO QUE:** Comprueba si los datos son diferentes.

`false != false` → **false**    `5 != 4` → **true**

**< MENOR QUE:** Comprueba si el primer dato es menor al segundo.

`'a' < 'b'` → **true**    `10 < 8` → **false**

**> MAYOR QUE:** Comprueba si el primer dato es mayor al segundo.

`'a' > 'b'` → **false**    `10 > 8` → **true**

**MENOR O IGUAL QUE:** Comprueba si el primer dato es menor

**≤ o igual al segundo.**

`10 <= 8` → **false**    `8 <= 8` → **true**

**MAYOR O IGUAL QUE:** Comprueba si el primer dato es mayor

**≥ o igual al segundo.**

`10 >= 8` → **true**    `8 >= 8` → **true**

# OPERADORES III

**LOGICOS:** PERMITEN AGRUPAR EXPRESIONES LOGICAS.

! **NOT (Negación):** Invierte el valor de la expresión.

!  $(8 == 8)$  → **false**      ! **false** → **true**

& **AND (Producto lógico):** Une dos expresiones. Se deben cumplir ambas condiciones.  $(4 == 4) \& (5 > 10)$  → **false**

&& **AND (Con cortocircuito):** Similar al anterior, pero si la primera condición es **false**, la segunda no se evalua.  
 $(10 == 4) \&\& (5 < 10)$  → **false** ( $5 < 10$ ) no se procesa

| **OR (Suma lógica):** Une dos condiciones. Con que una cumpla la condición el resultado será **true**.  $(4 == 4) | (5 > 10)$  → **true**

|| **OR (Con cortocircuito):** Similar al anterior, pero si la primera condición es **true**, la segunda no se evalua.  
 $(4 == 4) || (5 > 10)$  → **true** ( $5 > 10$ ) no se procesa

^ **XOR (Suma lógica exclusiva):** Une dos expresiones. Deben dar resultados distintos para obtener **true**.

$(4 == 4) ^ (5 > 10)$  → **true**      **true** ^ **true** → **false**

**Se pueden mezclar:**

**true** ^ **true** &&  $5 == 5$  |  $10 > 8$  &  $6 == 10$  || **false**

# OPERADORES IV

**A NIVEL BITS:** PERMITEN REALIZAR MODIFICACIONES EN DATOS ALTERANDO SU VALOR BINARIO.

~ **COMPLEMENTO UNARIO:** Invierte los bits del valor.

```
int n = ~6; → n=-7 char let = (char) ~'Ñ'; → let=N
```

>> **DESPLAZAMIENTO DERECHA:** Desplaza los bits hacia la derecha la cantidad indicada. short n = 8>>1; → n=4

<< **DESPLAZAMIENTO IZQUIERDA:** Desplaza los bits hacia la izquierda la cantidad indicada. short n = 8<<1; → n=16

>>> **DESPLAZAMIENTO DERECHA SIN SIGNO:** Desplaza los bits hacia la derecha la cantidad indicada sin signo.

```
char let = 'ñ'>>>2; → let=<
```

| **OPERACIONES LOGICAS:** Realizan operaciones lógicas entre los & valores. | (OR), ^ (XOR) y & (AND).

& int n = 10 & 5; → n=0

```
int n = 10 | 5; → n=15
```

```
int n = 10 ^ 5; → n=15
```

**ASIGNACION 3.0:** Realiza la operación sobre el valor actual.

<<= n <<= 2;

n = n << 2;

|= n |= 2;

n = n | 2;

>>= n >>= 2;

n = n >> 2;

&= n &= 2;

n = n & 2;

>>>= n >>>= 2;

n = n >>> 2;

^= n ^= 2;

n = n ^ 2;

# OPERADORES V

**UNARIOS:** FORMA SIMPLIFICADA PARA INCREMENTAR O DECREMENTAR UN NUMERO EN UNA UNIDAD.

**INCREMENTAL:** Se incrementa la variable en uno. Dos tipos:

**Postincremental:** Se utiliza el valor y se incrementa.

++

```
int a = 0;  
int b = a++;
```

→ b=0 y a=1

**Preincremental:** Se incrementa el valor y luego se utiliza.

```
int a = 10;  
int b = ++a;
```

→ b=11 y a=11

**DECREMENTAL:** Se decrementa la variable en uno. Dos tipos:

--

**Postdecremental:**

```
int a = 5;  
int b = a--;
```

→ b=5  
a=4

**Predecremental:**

```
int a = 5;  
int b = --a;
```

→ b=4  
a=4

**OPERADOR TERNARIO:** ES UN `if/else` ABREVIADO.

**Sintaxis:**

Expresión condicional



True

```
String resultado = (num%2==0) ? "Par" : "Impar";
```



Solo puede usarse el operador condicional cuando interviene devolución de datos.

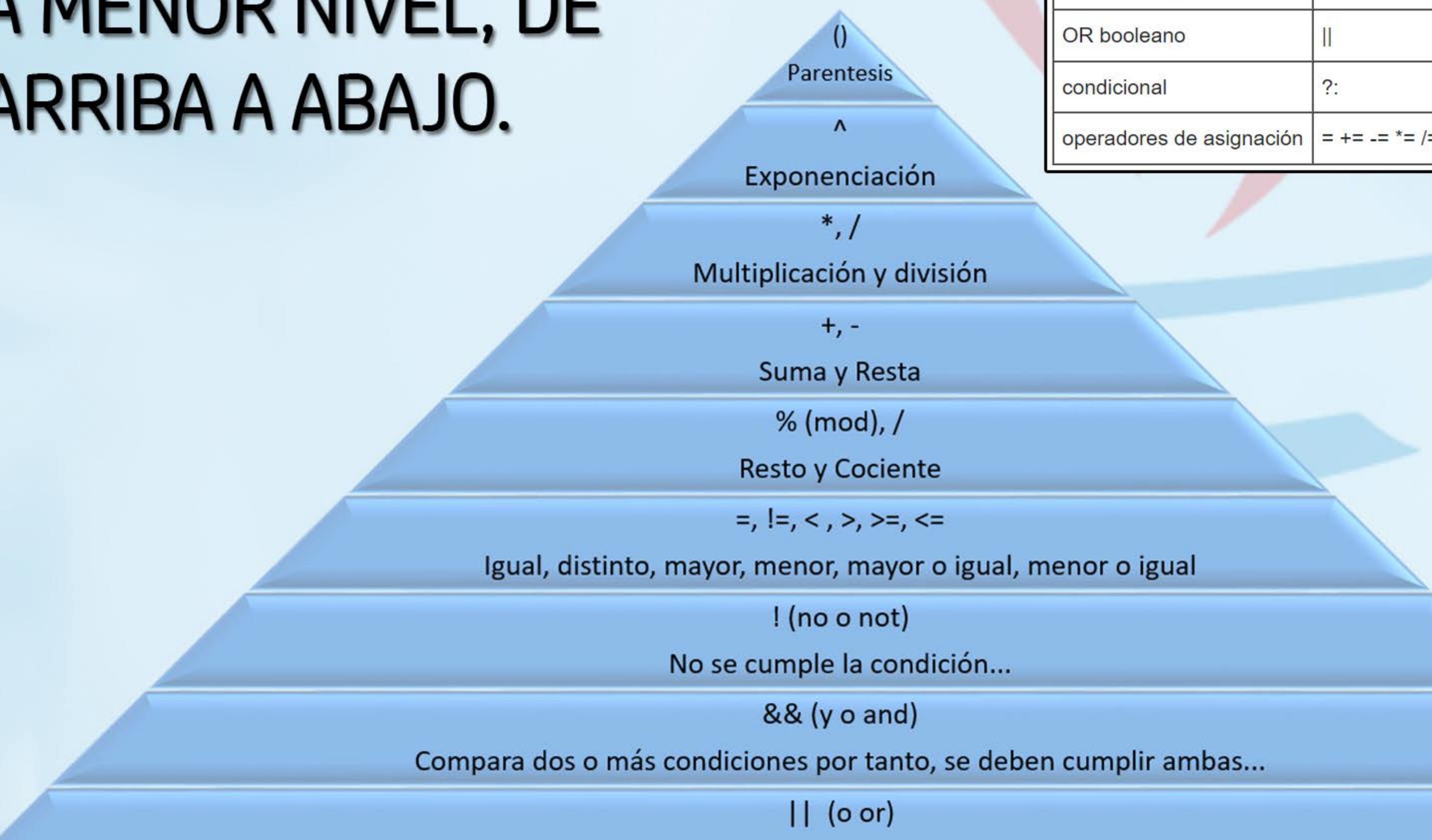
False



# OPERADORES VI

## PRECEDENCIA DE OPERADORES:

ES UNA REGLA QUE ESTABLECE QUE OPERADORES SE DEBEN EJECUTAR PRIMERO. EN LAS IMAGENES PODEMOS VER ESE NIVEL DE PRECEDENCIA, ORDENADOS DE MAYOR NIVEL A MENOR NIVEL, DE ARRIBA A ABAJO.



Descripción	Operadores
operadores posfijos	op++ op--
operadores unarios	++op --op +op -op ~ !
multiplicación y división	* / %
suma y resta	+ -
desplazamiento	<< >> >>>
operadores relacionales	< > <= >=
equivalencia	== !=
operador AND	&
operador XOR	^
operador OR	
AND booleano	&&
OR booleano	
condicional	?:
operadores de asignación	= += -= *= /= %= &= ^=  = <=>= >>>=