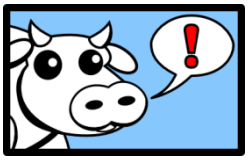
Estructuras de Control en Java

Las estructuras de control son el componente esencial de todo lenguaje de programación, ya que, como su nombre lo indica, son estructuras que permiten **controlar el flujo** de la ejecución del programa.

Antes de comenzar con las estructuras de control en sí, conviene notar que la declaración de las mismas varía un poco con relación a la de otras sentencias.

Las estructuras de control **no utilizan punto y coma (;) al finalizar**, en su lugar **utilizan llaves** ("**{**" y "**}**") que establecen dónde empieza y dónde termina su contenido. A este espacio lo llamaremos **contexto de la estructura de control.**



DEFINICIÓN: 3 – Sintaxis General de las Estructuras de Control

```
estructura(parámetros){
      sentencias;
}
```

Otra particularidad es que las variables que declaremos ***dentro* del contexto** de una estructura de control, sólo existen **dentro de ese contexto**. Cuando termine la ejecución de la estructura de control, todas las variables declaradas dentro de la misma y su contenido **serán eliminadas** de la memoria. **Tampoco es posible acceder a las mismas de forma directa** desde afuera del contexto de la estructura de control, pero es posible recuperar la información de las mismas, como veremos más adelante.

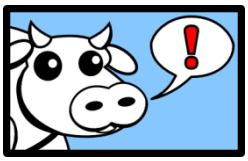
La Condicional (IF/IF-ELSE)

La estructura de control más básica de todas es ***la condicional***, a la cual nos referiremos, de ahora en adelante, como ***if***.

Como repaso, recordemos que ***if*** es una estructura que permite que el programa decida **si ejecuta o no un conjunto de acciones, o decida entre *dos conjuntos* de acciones posibles basándose en el *valor de una condición***, la cual se evalúa al ejecutarse la estructura.
Esto implica que el ***if*** puede declararse de **dos formas diferentes**, que llamaremos **declaración simple** y **declaración completa**.
En cualquiera de los dos casos, el ***if*** evalúa el valor de **una condición** que por lo general tiene una de las siguientes formas:

- **Una comparación lógica entre valores** o **entre el contenido de dos variables**.
- **Una variable** de tipo **`boolean`.**
- **El resultado de la ejecución de un método** que devuelve un valor de tipo **`boolean`.**

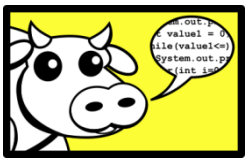
La **declaración simple** se utiliza cuando queremos que el programa decida **si debe ejecutar o no** un conjunto determinado de sentencias. Si la condición resulta **verdadera**, las sentencias dentro del ***if*** se ejecutan, de lo contrario, el programa las ignora y no son ejecutadas.



DEFINICIÓN: 4.1 – Declaración Simple del IF

```
if( condiciónAEvaluar ){
   //sentencias del bloque
}
```

Ejemplo 1: la expresión dentro de las llaves del ***if*** se ejecutará (mostrará en pantalla "**`El valor es 2`**"), porque la condición expresada ("**`num1 == 2`**") chequea si es verdadero o falso que la variable "**`num1`**" almacena el valor "2". En este caso la condición es verdadera, o sea, tiene el valor **`true`**.



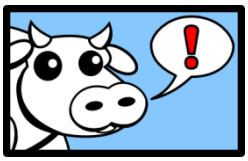
```
int num1 = 2;
if(num1 == 2){
    System.out.println("El valor es 2");
}
```

Ejemplo 2: la expresión dentro de las llaves del ***if*** **no se ejecutará** (no se mostrará en pantalla "**`El valor es 3`**"), porque la condición expresada ("**`num1 == 3`**") chequea si es verdadero o falso que la variable "**`num1`**" almacena el valor "3". En este caso la condición es falsa, o sea, tiene el valor **`false`**.

```
int num1 = 2;
if(num1 == 3){
    System.out.println("El valor es 3");
}
```

Ahora bien, si las condicionales utilizan comparaciones entre valores y variables, ¿cómo se realizan esas comparaciones? Para realizar comparaciones utilizaremos **operadores de comparación** y **operadores lógicos**, los cuales nos permiten expresar estas comparaciones.

Ya vimos, en los ejemplos anteriores, uno de estos comparadores: el **comparador de igualdad** ("**==**"). En la siguiente tala se detallan los otros comparadores existentes en el lenguaje:



Operadores de Comparación:

En todos los casos, es posible comparar entre sí el valor de variables o valores "sueltos" en cualquier combinación: variables con variables, variables con valores, o valores con valores.

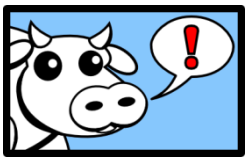
Operador	Descripción
`var1 == var2` `var1 == valor`	**Igualdad** Devuelve **`True`** si los elementos comparados son **iguales.**
`var1 != var2` `var1 != valor`	**Desigualdad** Devuelve **`True`** si los elementos comparados son **distintos.**
`var1 > var2` `var1 > valor`	**Mayor Que** Devuelve **`True`** si el primero de los elementos comparados es **mayor que el otro.**
`var1 < var2` `var1 < valor`	**Menor Que** Devuelve **`True`** si el primero de los elementos comparados es **menor que el otro.**
`var1 >= var2` `var1 >= valor`	**Mayor o Igual Que** Devuelve **`True`** si el primero de los elementos comparados es **mayor o es igual que el otro.**
`var1 <= var2` `var1 <= valor`	**Menor o Igual Que** Devuelve **`True`** si el primero de los elementos comparados es **menor o es igual que el otro.**

Operadores Lógicos:

Permiten combinar o transformar comparaciones lógicas. Salvo la negación, que trabaja sobre una sola expresión, el resto Permite combinar dos expresiones o variables de tipo **`boolean`** en una sola.

Operador	Descripción
`!(comp1)`	**Negación Lógica** Niega el valor de la expresión o variable de tipo **`boolean`** ubicada entre los paréntesis. Equivalente a la negación de lógica proposicional.
`comp1 && comp2`	**Conjunción Lógica** Devuelve **`true`** **solo si las dos** tienen valor **`true`** al mismo tiempo. Equivalente al operador "Y" de lógica proposicional.
`comp1 || comp2`	**Disyunción Lógica Simple** Devuelve **`true`** si **al menos una** tiene valor **`true`**. Equivalente al operador "O" de lógica proposicional.
`comp1 ^ comp2`	**Disyunción Lógica Exclusiva** Devuelve **`true`** si **las dos expresiones** tienen valor lógico **opuesto entre sí**. Equivalente al operador "XOR" de lógica proposicional.

Otra forma de trabajar con ***if*** es con la **declaración completa**. Esta forma permite que el algoritmo elija **un camino entre dos caminos posibles**, dependiendo del valor de la **condición** establecida. Esto se logra añadiendo un nuevo bloque de llaves encabezado por la palabra clave ***else.***

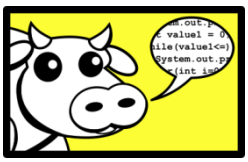


DEFINICIÓN: 4.2 – Declaración Completa del IF

```
if( condiciónAEvaluar ){
        //sentencias_si_verdadero;
 }
else {
        //sentencias_si_falso;
 }
```

Si el chequeo de la condición resulta verdadero (**`true`**), se ejecuta el primer bloque de sentencias. Si resulta falso (**`false`**), se ejecuta el ***segundo boque de sentencias***, el que está dentro del contexto de **`else`**.

Ejemplo 3:



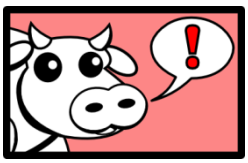
Ejemplo: funcionamiento de un bloque IF

```
int num1 = 2;
if(num1 == 3){
        System.out.println("El valor es 3");
        }
else{
        System.out.println("El valor es 2");
        }
```

El resultado de este código al ejecutarse sería:

```
El valor es 2
```

La expresión del primer bloque ***no se ejecutará*** (es decir, no se mostrará en pantalla el texto "**`El valor es 3`**"), porque la condición expresada ("**`num1 == 3`**") chequea si es verdadero o falso que la variable "**`num1`**" almacena el valor "3". En este caso la condición **es falsa** (o sea, tiene el valor **`false`**). Por este motivo, solo se ejecutará la expresión dentro del bloque **`else`**.



Nota sobre la comparación de valores de tipo *String:*

Los comparadores **`"=="`** **`y`** **`"!="`** **no permiten comparar valores de tipo *String***, o sea, comparar si dos cadenas de texto son idénticas entre ellas. En su lugar usaremos el método **`.equals()`**

`Ej:`

Supongamos dos variables de tipo String ("texto1" y "texto2") y una cadena de texto cualquiera, la comparación entre ellas se realiza de la siguiente manera:

```
texto1.equals(texto2);
```

Otra manera, si quiero comparar texto1 a un texto cualquiera, sería:

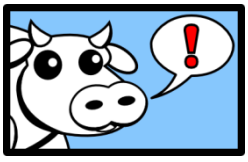
```
texto1.equals("texto cualquiera");
```

El Bucle FOR

Otra forma de controlar el flujo de un algoritmo es mediante **bucles:** estructuras de control que ejecutan **de forma repetitiva** una serie de sentencias, mientras **se cumpla una condición**.

En el bucle FOR, esta condición toma la forma de una expresión que contiene un **contador** con un **tope** y una **regla de avance**. Cada vez que el código contenido en el bucle se ejecuta en su totalidad, ese contador **se modifica** de acuerdo a la regla de avance y se lo compara con el valor del tope. Si el contador **no iguala** el valor tope, el bucle **se ejecuta una vez más**. Por el contrario, si el contador **iguala al tope,** el bucle **termina su ejecución**.

En código, esto se expresa de esta manera:



DEFINICIÓN: 4.3 – Declaración del Bucle FOR

```
for(int contador = 0; contador < 10; contador++){

 //sentencias del bucle

 }
```

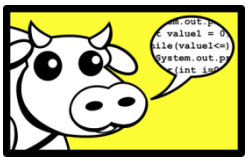
Los componentes del bucle se declaran **entre los paréntesis** luego de la palabra clave ***`for`*, separados por comas,** y en este ejemplo son los siguientes:

`int contador = 0` → Es la variable que cumple rol del **contador** del bucle. Como es natural en un contador, esta variable es de tipo **`int`**, ya que debe manejar ***solamente*** **valores numéricos enteros.** En este caso, la variable **`contador`** se inicia en 0, pero puede iniciarse en otro valor, según haga falta.

`contador < 10` → Esta expresión cumple el rol de **tope** del bucle. Como puede observarse, es una comparación lógica, igual a las utilizadas en la estructura IF y que, por lo tanto, devuelve un valor de tipo **`boolean`**. Para que el bucle continúe su ejecución, esta comparación tiene que resultar en **`true`** (en este caso, para que eso suceda el valor de la variable **`contador`** debe ser menor a 10).

`contador++` → Esta expresión establece la **regla de avance** del bucle. En este caso, indica que tras cada ejecución completa del bucle, el valor de la variable **`contador`** debe aumentar en 1. También se pueden utilizar como reglas de avance expresiones como estas:

- **`contador--`** Indica que la variable **`contador`** debe disminuir en 1.
- **`contador = contador + 2`** Indica que la variable **`contador`** debe aumentar en 2 (NOTA: puede usarse **cualquier otro número**).
- **`contador = contador - 2`** Indica que la variable **`contador`** debe disminuir en 2 (NOTA: puede usarse **cualquier otro número**).



Ejemplo: Funcionamiento de un Bucle FOR

```
for(int contador=0; contador<10; contador++){

        System.out.println("El contador vale:" + contador);

}
```

Resultado:

```
El contador vale:0
El contador vale:1
El contador vale:2
El contador vale:3
El contador vale:4
El contador vale:5
El contador vale:6
El contador vale:7
El contador vale:8
El contador vale:9
```

En la primera ejecución del bucle, se define la variable interna **`contador`**, como ya vimos, y se establece su valor en 0. Por supuesto, **0 es menor que 10**, por lo tanto, **el bucle puede continuar su ejecución**.

Así que se ejecuta **el código dentro del bloque del bucle**. En este caso, indica que se debe mostrar en pantalla el texto **`"El contador vale:"`**, seguido del **valor de la variable `contador` en ese momento**, que, como ya dijimos, es 0. Por lo tanto, nos muestra en pantalla **`"El contador vale:0"`**.

Como no hay más código dentro del bloque del bucle, pasamos al último comando del bucle: **`contador++`**, que le **suma 1 al valor actual de la variable**, o sea 0 + 1, por lo que **`contador`** ahora contiene el valor 1.

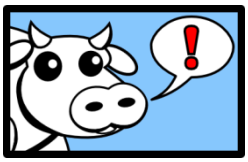
Este proceso se repite hasta que contador alcance el valor de 9. En ese punto, al ejecutarse el contenido del bloque y pasar al comando **`contador++`**, esta variable alcanza el valor de 10. Este valor rompe la condición, ya que **10 no es estrictamente menor que 10**. Por lo tanto, no se ejecuta el contenido del bloque y se sale del bucle.

Existe otra forma del bucle **`for`**, especialmente adaptada para trabajar con estructuras de datos multidimensionales como son las listas, pero la veremos más adelante, cuando estemos trabajando con ese tipo de estructuras.

El Bucle WHILE

A diferencia del bucle **`for`**, este bucle no establece un contador con un tope de ejecuciones, sino que chequea activamente una condición lógica como las usadas por el **`if`**. Esto implica que, dependiendo del valor de esa condición, el bucle **`while`** puede **no ejecutarse para nada,** o **ejecutarse indefinidamente**. Esta particularidad es muy útil en algunos casos, pero puede devenir en un ***bucle infinito*** si no tenemos cuidado.

El bucle **`while`** es considerablemente más sencillo de declarar que el bucle **`for`**:



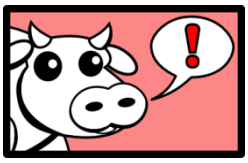
DEFINICIÓN: 4.3 – Declaración del Bucle WHILE

```
while( condición ){

    //sentencias del bucle

}
```

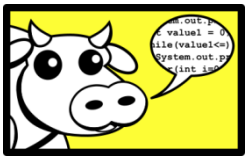
Como se puede apreciar, su sintaxis es muy semejante a la que utiliza el **`if`**. En este caso, la condición no solo controla el acceso a la estructura, sino que también controla la ejecución de una nueva iteración[2] del bucle. Mientras la condición establecida entre paréntesis devuelva un resultado **`True`**, el bucle va a continuar ejecutándose. Esto implica que el código no va a avanzar más allá del bucle mientras esto suceda. Por este motivo, el bloque de sentencias del **`while`** debe incluir alguna forma de control sobre el estado de esa condición, para evitar que el mismo quede sin control y caiga en un bucle infinito.



Bucles Infinitos:

Son bucles en los que no existe control sobre el valor de la condición que regula el funcionamiento del mismo, por lo que el bucle se ejecuta de forma repetitiva. Esto impide el avance de la ejecución del código, paraliza la aplicación, y consume recursos de cómputo en la computadora por encima de lo normal.

Veamos un ejemplo:



Ejemplo: Funcionamiento de un Bucle WHILE

```
int contador=0;

while( contador < 10 ){

        System.out.println("El contador vale:" + contador);
        contador++;

}
```

[2] Iteración: una secuencia completa en una serie repetitiva de acciones, o sea, en este caso es una "vuelta" del bucle.

Resultado:

```
El contador vale:0
El contador vale:1
El contador vale:2
El contador vale:3
El contador vale:4
El contador vale:5
El contador vale:6
El contador vale:7
El contador vale:8
El contador vale:9
```

Este ejemplo se diseñó para que entregue un resultado idéntico al que vimos con el bucle **`for`**. Nótese que definimos, necesariamente, la variable **`contador`** por fuera del bucle. La condición que establecimos en el bucle en este caso es idéntica que la usada para establecer el tope del for.

El funcionamiento sería de esta manera: al entrar en el bucle, se chequea el valor de la variable **`contador`**. Como el valor 0 es menor que 10, el bucle puede iniciarse y se ejecutan las sentencias del mismo: se muestra el mensaje en pantalla y se suma 1 al valor de la variable **`contador`**.

Este proceso se repite hasta que **`contador`** llega al valor 9. En esa iteración, luego de mostrar el mensaje, **`contador`** va alcanzar el valor 10. Eso significa que al chequear nuevamente la condición, esta va a devolver un valor **`False`** como ya vimos antes, y el bucle se cierra sin ejecutar su contenido.