INTRODUCCIÓN A LA PROGRAMACIÓN
MÓDULO 2

Tipo de dato booleano y expresiones lógicas

Los tipos de dato booleanos son aquellos conformados únicamente por 2 posibles valores: verdadero (true) ó falso (false). Un booleano puede ser definido en forma explícita, por ejemplo, asignando un "true" a una variable.

```
var soyMayor = true;
```

(Utilizando booleanos explícitos evitamos muchas veces trabajar con valores de tipo string: "si"/"no", "yes"/"not", "on"/"off" o similares). Un valor booleano declarado en forma explícita JAMAS deberá escribirse entrecomillado ("true"); de lo contrario no estaremos hablando de un booleano, estaremos hablando de una cadena de caracteres: un string. Los booleanos también se encuentran presente en forma IMPLÍCITA cuando trabajamos con expresiones booleanas: Son aquellas expresiones que se resuelven como verdaderas (true) o bien como falsas (false).

Operadores Comparativos

> (Mayor a)
< (Menor a)
>= (Mayor o igual a)
<= (Menor o igual a)
== ("Igual a" ⇒ Utilizado para verificar si 2 valores coinciden)
!= ("Distinto de" ⇒ Utilizado para verificar si 2 valores NO coinciden)
=== ("Idéntico a" ⇒ Utilizado para verificar si 2 valores coinciden y son del mismo tipo de dato)

Ejemplos de expresiones booleanas / lógicas:

A) 5 > 8 (Expresión booleana que devolverá: falso)

B) 5 + 9 > 10 (Expresión la cual se resolverá como verdadera).

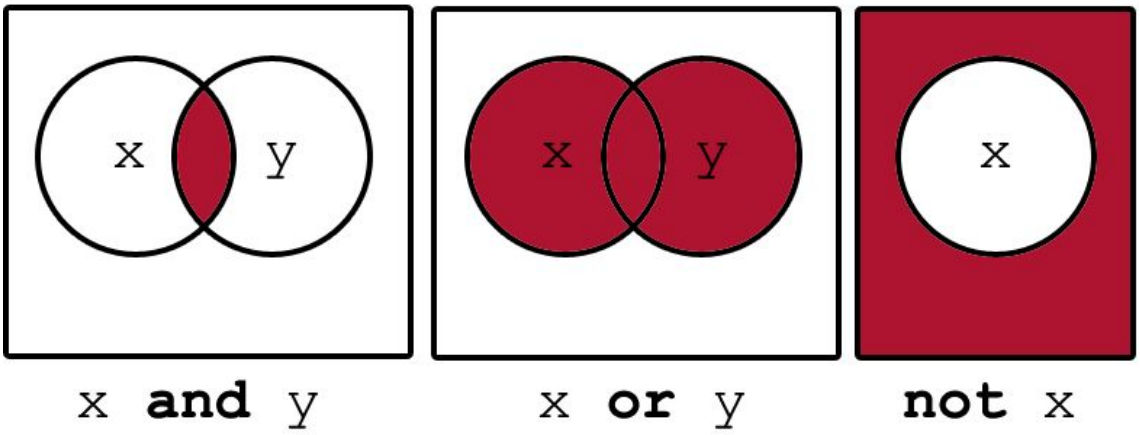
C) edad > 21 (siendo "edad" una variable).

Que la expresión termine resolviéndose como verdadera o como falsa dependerá del valor de la variable "edad" en ese determinado momento.

Operadores Lógicos

Un operador lógico se utiliza para crear expresiones booleanas COMPUESTAS. En programación, tenemos 3 operadores lógicos BASICOS:

Y	(Conjunción, intersección)	⇒ También llamado "AND"
O	(Disyunción, unión)	⇒ También llamado "OR"
NO	(Negación, complemento)	⇒ También llamado "NOT"



La sintaxis de ellos dependerá del lenguaje, aunque en muchos casos se escriben similar a:

Y	⇒	&&
O	⇒	||
NO	⇒	!

O bien en otros lenguajes se escriben en inglés: AND – OR - NOT

Ejemplos:

- (5 + 2 > 4) AND (2 < 10)
- (1 != 1) OR (2 + 2 == 4)
- (edad > 21) OR (edad == 30) (Siendo “edad” una variable)
- NOT (8 < 3)

Los 3 primeros ejemplos son expresiones booleanas COMPUESTAS, formadas por 2 sub-expresiones. El último ejemplo tiene una única expresión, a diferencia de los primeros 3. Para comprender la manera en que funcionan los operadores lógicos debemos saber que cada operador lógico trabaja con una TABLA DE VERDAD asociada.

Operadores lógicos “Y” (AND) -- “O” (OR)

Expresion1	Expresion2	AND	OR

true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Siendo EXPRESION1 y EXPRESION2 cualquier expresión booleana.

Por lo cual, volviendo a los ejemplos anteriores:

(5 + 2 > 4) AND (2 < 10) ⇒ true AND false ⇒ false
(1 != 1) OR (2 + 2 == 4) ⇒ false OR true ⇒ true
(edad > 21) OR (edad == 30) ⇒ El valor de retorno dependerá del valor de "edad"

Operador lógico "NO"

Los operadores lógicos AND y OR son operadores BINARIOS: Requieren que existan 2 operandos asociados. En cambio, la negación (NOT) es un operador lógico UNARIO (Trabaja con solo 1 operando asociado). El operador NOT invierte el valor de verdad de un término / expresión:

Todo aquello verdadero se transforma en falso.
Todo aquello falso se transforma en verdadero.

NOT	EXPRESION
false	true
true	false

Volviendo al ejemplo anterior:
NOT (8 < 3) ⇒ NOT false ⇒ true

Precedencia de los operadores lógicos

Al igual que los operadores aritméticos, los operadores lógicos tienen su precedencia:

- Mayor precedencia: NOT
- Precedencia intermedia: AND
- Menor precendencia: OR

Ejemplo:

```
NOT (5 == 5) OR (4 > 2) AND (1 != 1)
```

Se resolverá en el siguiente orden:

```
NOT true OR true AND false
false OR true AND false
false OR false
false
```

Por otra parte, el uso de paréntesis en la expresión adulterará la precedencia natural de los operadores: Primero deberá resolverse la expresión entre ()

Ejemplo:

```
NOT ( (5 == 5) OR (4 > 2) ) AND (1 != 1)
```

Se resolverá en el siguiente orden:

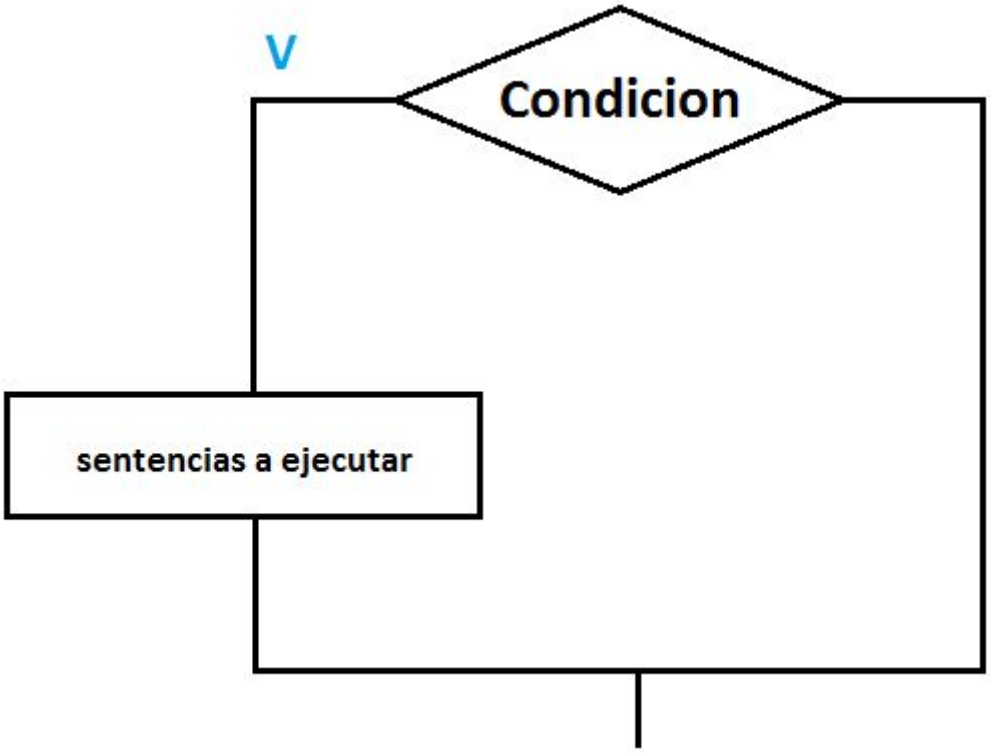
```
NOT (true OR true) AND false
NOT (true) AND false
false AND false
false
```

Estructuras de Control Condicionales

Son aquellas que nos permiten tomar decisiones acerca de qué sentencias (órdenes) debe ejecutar el programa, en base a una condición / expresión booleana.

Condicional if

El condicional if nos permite tomar una decisión en base a una condición.
Cuando la condición sea evaluada como verdadera, se ejecutará un bloque con un conjunto de sentencias asociado al if.



Su sintaxis en muchos lenguajes será similar a:

```
if(condición){
    //conjunto de sentencias a ejecutar.
}
```

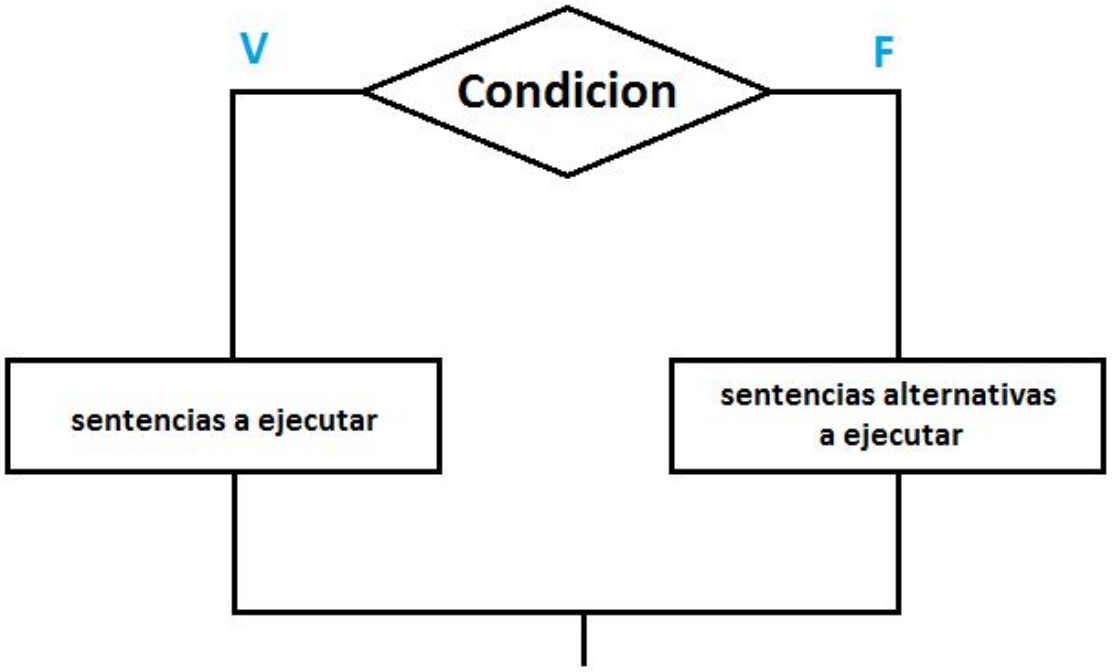
Ejemplo 1:

```
if(5 > 4){
    console.log('Condicion verdadera');
}
```

SI 5 > 4 ENTONCES IMPRIMIR “Condición verdadera”. Ejemplo 2:

```
if(2 == 3){
    console.log('Condicion verdadera');
}
```

SI 2 == 3 ENTONCES IMPRIMIR “Condición verdadera”. En el ejemplo 2, el bloque de sentencias asociado al if NO se ejecutará, ya que la condición resulta ser falsa. Aquí es donde en muchos casos nos será útil un camino alternativo de ejecución: **ELSE**
(“de lo contrario”), por donde siempre entraremos cuando la condición del if resulte ser falsa.



Ejemplo 3:

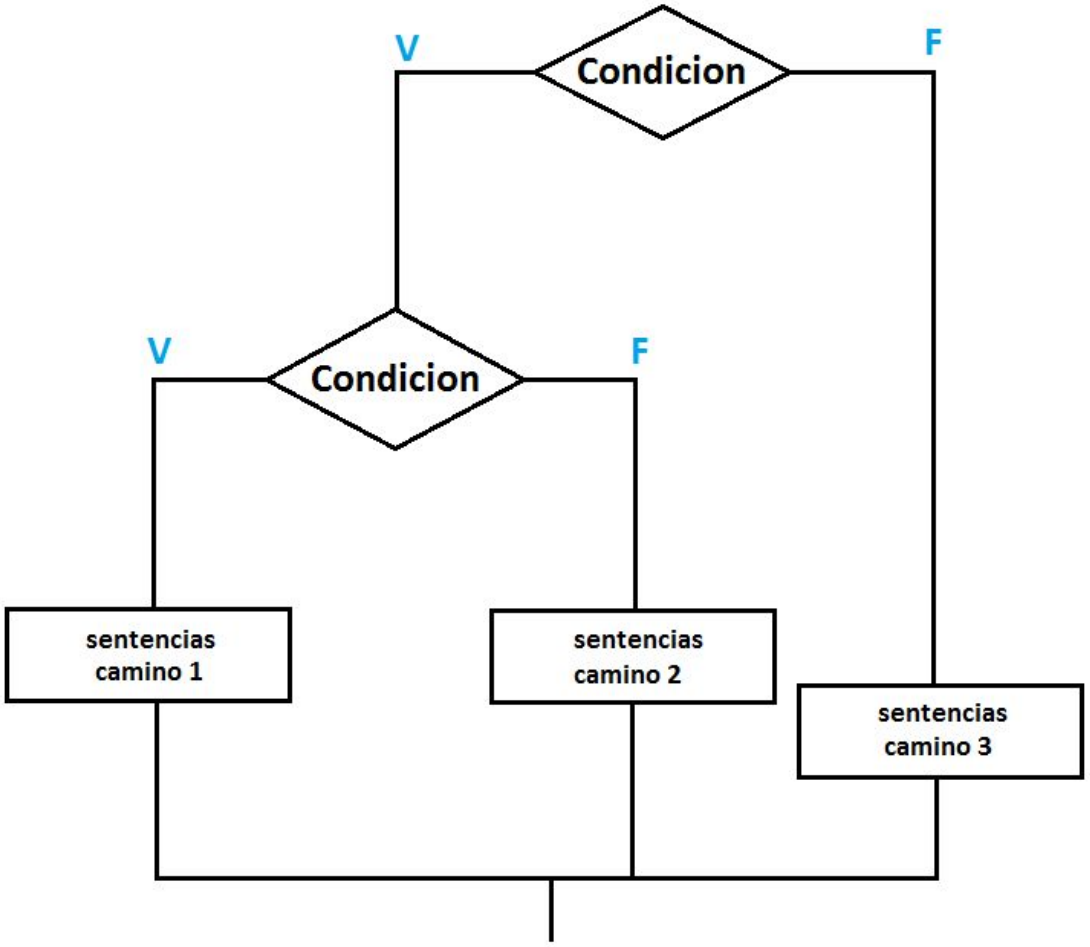
```
if(2 == 3){
    console.log('Condicion verdadera');
}
else{
    console.log('Condicion falsa);
}
```

SI 2 == 3 ENTONCES
IMPRIMIR “Condición verdadera”
DE LO CONTRARIO
IMPRIMIR “Condición falsa”

Anidamiento de condicionales if

Un condicional if puede escribirse dentro del bloque de otro if (sea en su camino principal o en su camino alternativo), de manera de generar múltiples caminos de ejecución. Mediante el anidamiento de condicionales el código comienza a “ramificarse”: Se generarán varias bifurcaciones. Un simple if con su respectivo else nos proveen 2 posibles caminos de ejecución; en cambio, anidando condicionales if podemos generar 3 o más bifurcaciones.

Ejemplo:




```
if(condición1){
    if(condición2){
        // camino 1
    }else{
        // camino 2
    }
}else{
    // camino 3
}
```

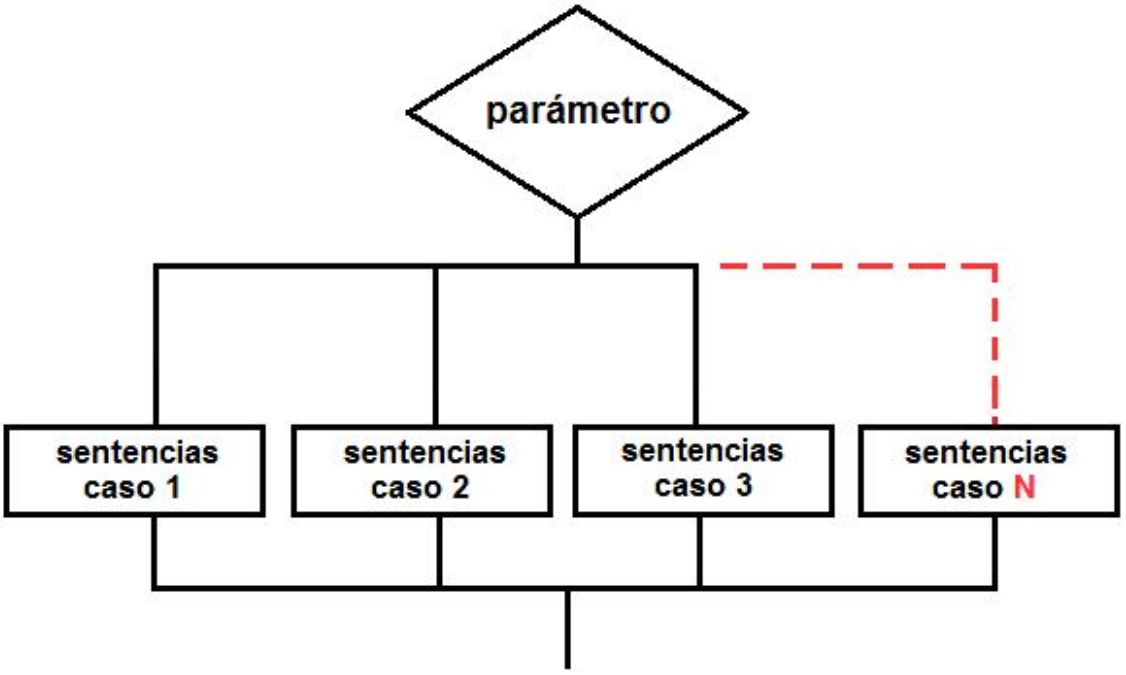
SI condicion1 ENTONCES
SI condicion2 ENTONCES
Sentencias camino 1
DE LO CONTRARIO
Sentencias camino 2
DE LO CONTRARIO
Sentencias camino 3

Con una estructura similar a esta, podemos averiguar por ejemplo, si el valor de una variable es: mayor que cero, menor que cero, o cero (3 alternativas).

El anidamiento de if en forma desmedida generará muchos caminos de ejecución, por lo cual la complejidad del programa puede verse claramente afectada. En esos casos, siempre es recomendable TABULAR el código, dejando el if anidado con mayor margen izquierdo, de manera de hacerlo más legible y facilitar su comprensión (Idem el código anterior).

Condicional múltiple

El condicional múltiple es muy útil en aquellos casos en que hayan muchos caminos de ejecución posibles, y no querramos escribir un conjunto de ifs anidados. En casi todo lenguaje de programación el condicional múltiple es conocido como: switch – case.



Su sintaxis en todo lenguaje es similar a:

```
switch(parametro){
            case 1:
                //sentencias caso 1
            break;
            case 2:
                //sentencias caso 2
            break
            case 3:
                //sentencias caso 3
```

```
            break
            default:
                //sentencias caso default
            break
}
```

Veamos como funciona:

El switch – case evalúa el valor de un **PARÁMETRO** (variables, expresiones o bien el valor de retorno de una función), buscando coincidencias con alguno de los casos existentes: cuando encuentra una coincidencia, ejecuta el bloque de código asociado a ese caso. Si NO encuentra ninguna coincidencia: en PRINCIPIO nada se ejecutará.

Un switch-case puede usarse cuando los valores del parámetro son PREDECIBLES y existe una cantidad FINITA de posibles valores: El programador debe conocer de antemano los posibles valores del parámetro, de manera de poder escribirlos como un posible case dentro del condicional múltiple (cantidad finita de cases, caminos).

Ejemplos de parámetros para un switch - case:

- El estado civil de una persona (soltero,divorciado,viudo,casado)
- El sexo de una persona (M,F)
- El resultado de un partido (victoria,empate,derrota)
- Los colores del arco iris (rojo, azul, violeta, etc)
- Puntos ganados en un partido (3,1 ó 0)
- Grupos sanguíneos.
- Provincias de Argentina.
- Etc

El switch-case es buena herramienta cuando existen mínimamente 3 opciones, ya que con 2 opciones se podría trabajar tranquilamente con un simple if y con su else.

Fin de un case:
El final de un case específico en todo lenguaje se señala de alguna manera, en muchos lenguajes se realiza esta acción escribiendo “break;” o un similar.

Caso por defecto:
Un switch-case puede tener OPCIONALMENTE un case “DEFAULT” el cual será el camino de ejecución elegido si NO existe un case específico acorde al valor del parámetro (no hubieron coincidencias entre el parámetro y algún case). El case default, en caso de estar presente, siempre es el último case del switch.

Un switch-case es más fácil de leer y mantener que los ifs anidados, ya que existen distintos caminos de ejecución todos al mismo nivel jerárquico.

Estructuras de Control Repetitivas:

Son aquellas que nos permiten ejecutar sentencias repetidas veces, sin caer en el recurso del copy-paste de código (una horrible práctica de programación). Un ejemplo: Imprimir un mensaje en pantalla muchas veces. En programación existen los BUCLES (o LOOPS, en inglés), los cuales nos permitirán repetir la ejecución de un grupo de sentencias. Cada una de las repeticiones de un bucle se las conoce con el nombre de CICLO o ITERACION.

En todo lenguaje existen 3 bucles básicos:
En este módulo veremos el bucle while y el bucle do – while.

Bucle while

Permite ejecutar un bloque de código MIENTRAS sea verdadera su condición asociada. El bucle while, al igual que un if, tiene una condición asociada que se evaluará como verdadera o bien como falsa.

Su sintaxis en muchos lenguajes también resulta bastante similar :

```
while(condición){
   //sentencias a ejecutar
}
```

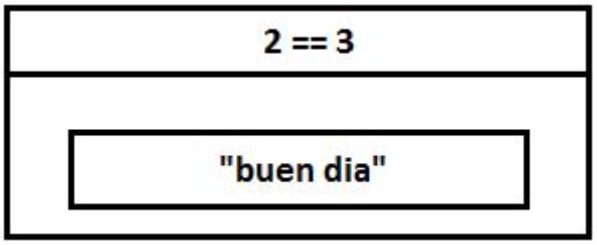
MIENTRAS condición
sentencias a ejecutar

MIENTRAS sea verdadera la condición, el bucle seguirá ejecutando (iterando).
Cuando la condición se vuelva falsa, el bucle se detendrá y se procederá a ejecutar el resto del programa.

¿Qué ocurriría si la condición NUNCA se vuelve falsa?
Esto en programación lo llamamos bucle infinito, un error grave en la lógica del programa, el cual NO permitirá que el programa pueda finalizar.

¿Qué ocurriría si la condición es falsa desde un principio?
De ocurrir esto entonces el bucle NO se ejecutará, ni siquiera 1 vez.

Ejemplo:




```
while(2 == 3){
    console.log('Buen dia');
}
```

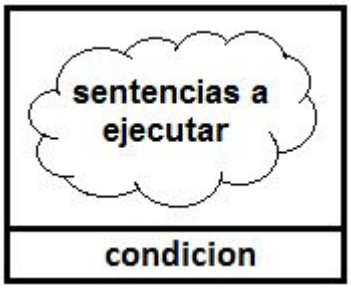
MIENTRAS 2 == 3
IMPRIMIR “Buen dia”

Este bucle tiene la orden de imprimir un mensaje MIENTRAS que 2 sea igual a 3, lo cual CLARAMENTE es falso.

Entonces:
¿Cuántas veces se imprimirá el mensaje en pantalla? NINGUNA vez.

Bucle do - while

El bucle do – while es similar al bucle while, con la diferencia de que primero ejecuta, y luego evalúa si es cierta o falsa la condición de continuidad.

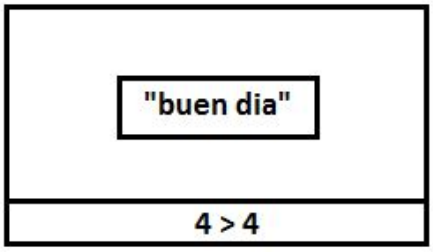



```
do{
    //sentencias a ejecutar
}
while(condición);
```

HACER
sentencias a ejecutar
MIENTRAS condición

Este bucle itera (HACE) MIENTRAS su condición sea verdadera. Y como se comentaba antes: primero HACE, y luego evalúa la condición. ¿Cuál es la consecuencia de que primero haga y luego evalúe la condición? Esto GARANTIZA al menos 1 iteración (POR MÁS que la condición sea FALSA desde un principio.

Ejemplo:



Este bucle imprimirá el mensaje 1 vez en pantalla, aunque la condición del bucle sea CLARAMENTE falsa. Al detectar el bucle que la condición es falsa desde un principio, NO se procederá a ejecutar una 2da. Iteración: Se continuará con la ejecución del resto del programa.

En muchos casos NO es un comportamiento deseado que el bucle itere al menos 1 vez, en muchos casos no se desea ejecutar nada si la condición del bucle es falsa desde un principio: En esos casos se procederá a trabajar con el bucle while.

Anexo JavaScript:

Ventana de confirmación: **confirm**

```
var confirmacion = confirm("Esta seguro que desea
continuar?");
console.log(confirmacion);
```

Como podemos apreciar, la función de confirm al terminar de ejecutarse se “transforma” en un booleano y su valor va a depender de lo que haya elegido el usuario.

Ventana de ingreso de datos: **prompt**

```
var nombre = prompt("Ingrese por favor su nombre de usuario");
console.log("Bienvenido "+nombre);
```

En este caso, la función prompt puede “transformarse” en un string representando lo que sea que haya escrito el usuario o NULL en caso de que haya cancelado la operación.

Función que verifica valores NO numéricos: **isNaN** *(devuelve un booleano)*

```
console.log(isNaN(1)) // false
```

```
console.log(isNaN("1")) // true
```

Función que convierte un valor a tipo numérico: **Number**

```
var edad = "28";
var cumple = edad + 1;
console.log(cumple) // "281"

var edad = "28";
var cumple = Number(edad) + 1;
console.log(cumple) // 29
```

Función que devuelve el tipo de dato de una variable / valor: **typeof**

```
console.log(typeof ""); // string

var edad = 28;
console.log(typeof edad);//number
```