INTRODUCCIÓN A LA PROGRAMACIÓN MÓDULO 3

Bucle While / Do...While

El bucle while es uno de los bucles más simples para comenzar a practicar iteraciones o repeticiones de código. Tengamos en consideración nuevamente que los bucles nos permiten "repetir" o iterar múltiples veces sobre un bloque de código dado. Es por esto que los bucles necesitan siempre de 3 partes indispensables para un correcto funcionamiento:

- Variable de control
- Condición
- Edición de Variable de control(aumento, decremento)

La **variable de control** nos va a servir para inicializar el bucle y también para saber exactamente en donde estamos parados cuando corremos el programa. La misma sirve además para controlar que un bucle no sea infinito(En el caso en que no necesitemos uno, que son la mayoría) con la **edición de la variable de control**.

Estructura

```
//variable de control contador
var contador = 0;
while(contador < 5){
        console.log(contador)
        //edicion de la variable de control en aumento
        contador++
}
```

El programa anterior muestra una forma de imprimir en consola el valor de la variable contador, que a su vez es nuestra variable de control la cual hace que el bucle arranque desde el número 0 hasta un valor que discrepe con la condición dada, la cual es que contador sea menor a 5. En cada iteración, al finalizar el bloque estamos aumentando la variable de control contador en una unidad para que la próxima vez que corra su valor sea distinto, de otra forma el mismo sería un bucle infinito y esto causaría que el programa presente errores de ejecución.

Una variante interesante de este bucle **while** es el **do...while** el cual me permite ejecutar un bloque de código una única vez como mínimo y luego dependiendo la condición se iniciará un ciclo de bucle o no.

Estructura

```
//variable de control contador
var contador = false;
do{
        console.log(contador)
        //edicion de la variable de control en aumento
}
while(contador < 5)
```

En el programa anterior vemos un caso particular donde la condición por defecto es falsa. De cualquier manera, el programa ejecutará su cuerpo una primera vez dentro de la sentencia **do** para luego evaluar la condición y determinar que el bucle no debería iniciarse.

Bucle for:

El bucle for es otro de los bucles de los lenguajes de programación. Se trata de un bucle EXACTO, el cual se repite (itera) N veces. El valor N debe ser conocido de antemano por el programador. Es decir, si queremos iterar 20 veces, 300 veces, 1480 veces, N veces.. es conveniente trabajar con este bucle.

La estructura general de un bucle for es la siguiente:

```
for(A ; B ; C){
      // Sentencias a ejecutar repetidas veces.
}
```

Siendo:

A: Inicialización de la variable de control.
B: Condición de continuidad.
C: Incremento ó decremento de la variable de control.

El bloque de llaves { } incluirá todas las sentencias a repetir. La sintaxis entre un lenguaje y otro puede variar, pero la estructura del for y la forma de funcionar en esencia NO cambia.

A) Inicialización de la variable de control:
La variable de control es la variable interna que utiliza el bucle for, la cual se irá incrementando o decrementando hasta alcanzar un valor final, en el cual el bucle finaliza su trabajo.

Por convención, esta variable recibe el nombre “i” (iterator), aunque podría ser cambiado por otro nombre más específico, según el criterio del programador.

La variable de control puede ser inicializada en cualquier valor numérico, según la conveniencia del caso puntual. En muchos casos se la inicializa en 0 ó en 1, pero podría inicializarse incluso con valores negativos.

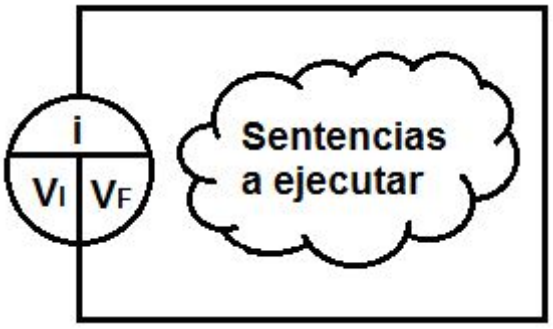
B) Condición de continuidad:
Mientras sea verdadera la condición de continuidad (una expresión lógica simple) el bucle for continuará iterando.

Habitualmente, la expresión es del tipo:
i < N ó i <= N

Siendo N, el valor a alcanzar.
NOTA: En el caso que la expresión sea: i <= N, habrá una iteración adicional.

C) Incremento ó decremento de la variable de control:
La variable de control "i" debe incrementar o decrementar por cada iteración (repetición) del bucle.

Habitualmente, la variable de control se incrementa de a una unidad (de uno en uno), aunque también existe la posibilidad que esta variable decremente (bucle for descendente). Una forma de representar un bucle for en un diagrama es la siguiente:



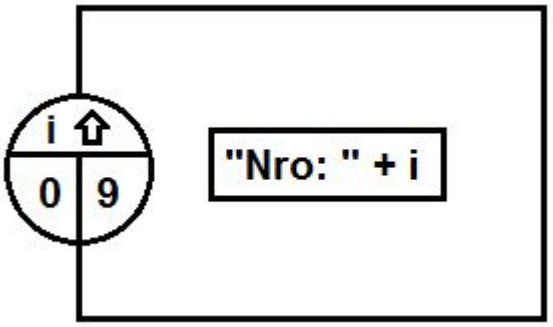
Siendo:

- "Vi" el valor en el cual se inicializa "i".
- "Vf" el valor a alcanzar para finalizar.

Un primer ejemplo:

```
for(i = 0; i <= 4; i++){
    console.log(i);
}
//salida de consola
//0
//1
//2
//3
```

DESDE i = 0 MIENTRAS i <= 9 INCREMENTO i EN 1
IMPRIMIR i

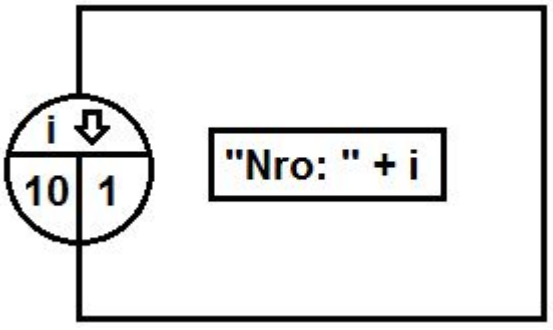


Este ejemplo del bucle for imprimirá en pantalla la variable de control “i” 10 veces.
(Se imprimirán en pantalla los números del 0 al 9 inclusive). Se trata de un bucle for ASCENDENTE: Se inicializa la variable de control en cero, se pide que continúe iterando hasta llegar a 9, con la variable de control incrementándose de a 1 (i++).

Un segundo ejemplo:

```
for(i = 10; i >= 1; i--){
    console.log(i);
}
```

DESDE i =10 MIENTRAS i >= 1 DECREMENTO i EN 1
IMPRIMIR i



Este ejemplo del bucle for imprimirá en pantalla la variable “i” 10 veces (Se imprimirán en pantalla los números del 10 al 1 inclusive). Se trata de un bucle for DESCENDENTE: Se inicializa la variable de control en 10, se pide que continúe iterando hasta llegar a 1, con la variable de control decrementándose de a 1 (i--).

¿Puede un bucle for generar infinitas iteraciones? SI

Suele darse cuando el programador decide, dada cierta circunstancia, re-asignar un valor a la variable “i” dentro del bloque del for.

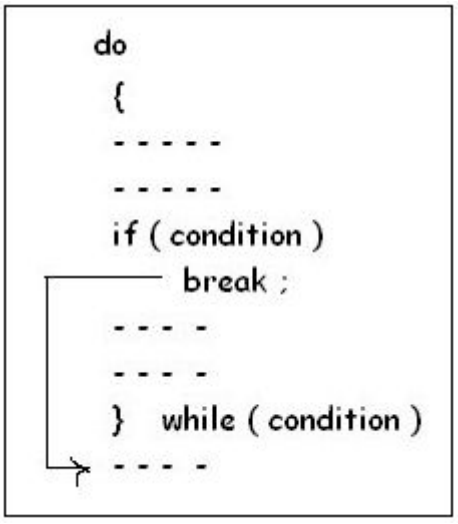
Esta acción siempre es peligrosa (aunque válida para usos avanzados), dado que altera la lógica natural del bucle: El incremento / decremento de la variable de control la realiza el mismo bucle; si el programador re-asigna el valor de dicha variable “por cuenta propia”, podría generarse un error en la lógica que genere un bucle infinito.

Otro caso de bucle for infinito podría ser un bucle con “i” inicializada en cero, la cual debe alcanzar cierto valor X MAYOR a cero para finalizar, pero que DECREMENTA en lugar de incrementar (Un error grave en la lógica del programador, lo cual debe ser corregido).

Sentencias break y continue:

Break

La sentencia break se utiliza para interrumpir un bucle antes de lo esperado.
Suele escribirse en muchos lenguajes: **break;**



Esta sentencia es válida dentro de un bucle (cualquiera de ellos); en cambio NO es válida fuera de un bucle, lo cual generará errores en el programa. ¿Por qué interrumpir un bucle antes de tiempo? Puede darse una situación en la cual el bucle no debe finalizar “naturalmente”, sino forzar su interrupción antes, dada cierta circunstancia.

Es decir, el uso de break dentro de un bucle siempre estará condicionado:

```
if(condición){
    break; //Se termina la iteración
}
```

Esta sentencia es muy útil cuando se trabaja con arreglos, uno de los próximos temas.

Continue:
A diferencia de la sentencia break, continue interrumpe solamente la iteración ACTUAL, y NO todo el bucle: Al ejecutarse un continue, se procederá con la siguiente iteración. Continue siempre es válido dentro de un bucle, y nunca fuera de ellos.
Su uso dentro del bucle estará condicionado:

```
if(condición){
    continue; //Se continúa con el siguiente paso dentro del bucle si es que hubiera alguno
}
```

Por ejemplo, si quisiéramos imprimir en pantalla los números en pantalla desde el 1 al 30, excepto el 13, la sentencia continue resulta sumamente útil:

```
for(i = 1; i <= 30; i++){
    if(i == 13){
        continue;
    }
    console.log(i);
}
```

El valor numérico de “i” siempre se imprime en pantalla, excepto cuando “i” alcanza el valor “13”: En ese caso se ejecutará la sentencia continue; por lo cual finalizará en ese mismo momento la iteración actual, pasando inmediatamente a iterar con i = 14.

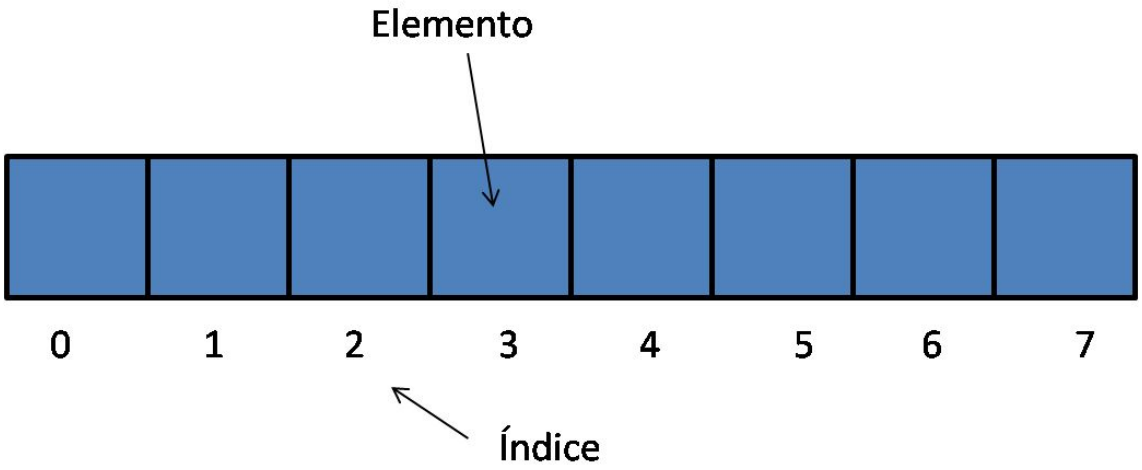
Labels

Los labels son identificadores de sentencias en Javascript. Nos permiten darle un “puntero” a una línea o bloque de código determinado. Puntualmente son utilizados para diferenciar capas de bucles, dado que de esta forma podemos desde una iteración interna continuar ó parar la ejecución de una iteración externa. Los labels si bien son strings, no se escriben con el mismo formato, es decir entre comillas, sino como una variable, pero estos no se declaran como tal ni toman valores con el operador genérico de asignación(=). Ejemplo :

```
//Label para un bucle externo
externo :
        for(var i = 0; i<5;i++){
                //Label para un bucle interno
                Interno :
                for(var i = 0; i<5;i++){
                        break externo;
                }
        }
```

Arreglos / Vectores

Un arreglo es una estructura formada por múltiples datos correlativos que se guardan en memoria. Todo arreglo contendrá N elementos (posiciones), con determinados valores.



Cuando un arreglo es UNIDIMENSIONAL (una tira de valores) se lo conoce con el nombre específico de VECTOR. En cambio, si el arreglo es BIDIMENSIONAL (una tabla de valores) se lo conoce como MATRIZ.

Características de los arreglos:

- Dependiendo del lenguaje de programación, los arreglos contendrán datos del mismo tipo o bien datos heterogéneos (números + strings + booleanos, etc).
- En JavaScript, a diferencia de otros lenguajes, los arrays admiten valores heterogéneos (es decir, que pueden contener valores de distinto tipo: números, strings, booleanos, etc).
- Todo elemento / posición de un arreglo tiene un índice numérico.
  Naturalmente los índices comienzan a contarse desde CERO.

- En algunos lenguajes los vectores son ESTÁTICOS: Contendrán N cantidad fija de elementos: No será posible agregar nuevos elementos en tiempo de ejecución.

- En JavaScript, a diferencia de otros lenguajes, los arrays son estructuras dinámicas ya que no tienen una cantidad de elementos predefinida. Esto implica que durante la ejecución del programa es posible crear nuevos elementos, como también remover los existentes. Concluyendo: Los arrays dinámicos se pueden REDIMENSIONAR.

En cambio en otros lenguajes (Ej: JavaScript, PHP) los arreglos son
DINÁMICOS: Siempre será posible seguir agregando nuevos elementos en tiempo de ejecución, como también remover los existentes. Decimos entonces que los arreglos dinámicos son REDIMENSIONABLES: En cada instante durante la ejecución del programa la cantidad de elementos puede ser variable.

Véase: La imagen de ejemplo anterior muestra un arreglo (vector) de 7 elementos con valores numéricos. Observe que el último índice es 6 (y NO 7) dado que los índices empiezan a contarse desde CERO.

Tipos de vectores

Dependiendo el lenguaje en el que estemos programando vamos a encontrar una gran gama de posibles tipos de vectores a nuestra disposición. A continuación nombramos los tipos de vectores con los que contamos en Javascript :

- Vector indexado secuencial simple
- Vector indexado secuencial multidimensional (Matriz)
- Vector indexado asociativo simple
- Vector indexado asociativo multidimensional (Matriz)
-

Dependiendo también el lenguaje, los distintos tipos de vectores pueden estar contenidos en la misma estructura de datos o formar parte de otra distinta. En Javascript podemos diferenciar a los vectores secuenciales de los asociativos por su tipo de dato. Observemos el siguiente ejemplo :

```
//Vector Secuencial
var personas = ["Horacio","Pablo","Aylen"];
console.log(personas);

//>Array(3)
//0 : "Horacio"
//1 : "Pablo"
//2 : "Aylen"
```

Como podemos apreciar, este vector automáticamente va asignando el índice de sus valores de manera secuencial.

```
//Vector Asociativo
var persona = {
```

```
    nombre : "Horacio",
    edad : 28,
    esProfe : true
}
console.log(persona);
```

En este caso nosotros somos los que asociamos un valor cualquiera a un índice específico.

¿Cómo crear un vector?
Los arreglos son referenciados por una variable.
La forma de crearlo dependerá del lenguaje, pero siempre será muy similar al siguiente ejemplo:

```
var amigos = [ "Pablo","Diego","Maria","Ana","Emilio" ];
//(Usaremos este ejemplo de aquí en adelante).
```

Creamos un arreglo (vector) de 5 elementos, con valores del mismo tipo (strings). Como se ha dicho anteriormente, los índices de los elementos naturalmente comienzan desde cero, Es decir que el primer elemento (“Pablo”) tendrá índice CERO. “Diego” tendrá índice UNO. “Emilio” tendrá índice 4. Otra forma de crear este vector es la siguiente:

```
var amigos = [ ];
amigos[0] = "Pablo";
amigos[1] = "Diego";
amigos[2] = "Maria";
amigos[3] = "Ana";
amigos[4] = "Emilio";
```

Inicialmente dejamos a nuestro vector sin elementos.
Luego, se asigna posición por posición cada valor del vector, indicando entre corchetes, los índices de cada elemento.

En muchísimos lenguajes se accede a cada elemento del arreglo escribiendo su índice entre corchetes [ ].

Ejemplo:

```
console.log(amigos[2]);
```

Imprimirá en pantalla el valor del 3er. Elemento à “María”.

Para el caso de los vectores asociativos es exactamente lo mismo. Tener en cuenta que si el índice que elegimos es un string, el mismo debe tener el formato correcto dentro de los corchetes. Ejemplo :

```
var persona = {
      nombre : "Horacio",
      edad : 28,
      esProfe : true
}
console.log(persona["nombre"])//"Horacio"
```

En muchos lenguajes la diferencia entre vectores puede presentar también distintas posibilidades de sintaxis para acceder a sus valores internos. En el caso de Javascript el mismo tiene dos tipos de notaciones :

- Notación de punto
- Notación de corchete(La que venimos ejemplificando anteriormente)

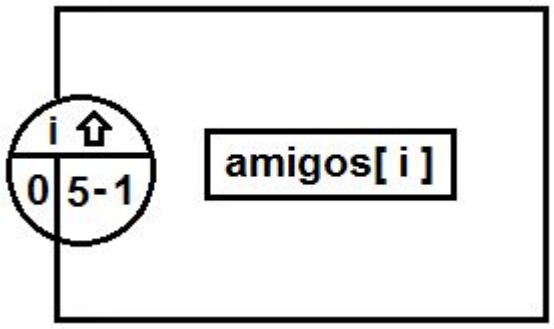
La notación de punto resulta mucho más práctica a la hora de programar ya que requiere que escribamos menos caracteres, pero siempre tenemos que tener en cuenta que ÚNICAMENTE se puede usar si y sólo si el índice es de tipo String, de lo contrario arrojará un error, como observamos en el siguiente ejemplo :

```
var persona = {
      nombre : "Horacio",
      edad : 28,
      esProfe : true,
     0 : false
}
console.log(persona.nombre)//"Horacio"
console.log(persona.0) //Error : Unexpected Number
```

¿Cómo recorrer un vector?

Para acceder a cada uno de los valores de un vector, será necesario recorrer toda la estructura usando un bucle. El bucle más adecuado es el bucle FOR.

```
for(i = 0; i < 5; i++){
    console.log(amigos[i]);
}
```



En el ejemplo anterior, se imprimirán todos los valores del vector de amigos.
Dado que el primer índice del vector es CERO, será conveniente inicializar la variable “i” en CERO.

Con la condición: i < 5
Llegamos a obtener el último elemento del vector (el cual tiene índice 4), pero NO se alcanza al elemento con índice 5 (el cual NO existe).

En el caso de los lenguajes con arreglos dinámicos, no es posible escribir en nuestro código la cantidad de elementos de la forma: i < 5 ya que esa cantidad puede variar constantemente: En esos casos disponemos de funciones de los distintos lenguajes que nos proveen la cantidad de elementos de un arreglo.

Las funciones de los distintos lenguajes tienen nombres muy similares a:

- size
- sizeof
- count
- length

Por lo cual la condición de continuidad del bucle for será similar a:

I < amigos.size
I < amigos.length ⇒ Notación de JavaScript.
I < count(amigos) ⇒ Notación de PHP
I < length(amigos)

Ejemplo :

```
console.log(["peras","manzanas","bananas"].length) // 3
```

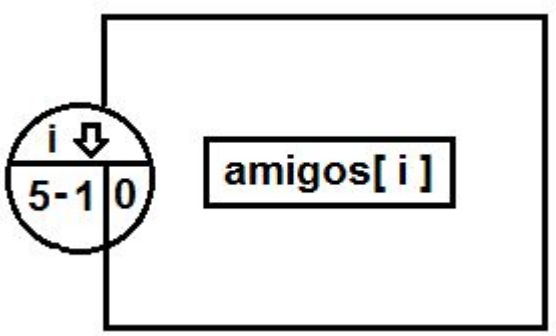
Interrupción del recorrido

En el caso de NO querer recorrer la estructura hasta el último elemento, podemos interrumpir dicho recorrido mediante la sentencia break.
En base a nuestro ejemplo del vector de amigos:
Si quisiéramos averiguar si una persona en particular es nuestro amigo:
Recorremos el vector buscando a una determinada persona, y si la encontramos, claro está que no tiene sentido seguir recorriendo hasta el final del vector; por lo cual podemos en ese momento ejecutar la instrucción break que aborte el bucle.

Recorrido de un vector en reversa:

Si se desea recorrer la estructura de atrás hacia delante, podemos armar un bucle descendente que resuelva esta situación.

```
for(i = amigos.length - 1; i >= 0; i--){
    console.log(amigos[i]);
}
```



Se inicializará la variable de control en el último índice (amigos.length - 1).
La variable de control decrementa, ya que la condición de continuidad indica que debe llegarse hasta CERO (el cual será el índice del primer elemento).

Cada lenguaje además suele contar con su propia implementación de bucles para recorrer vectores asociativos ya que no podemos usar ninguno de los bucles anteriores porque solamente podríamos aumentar o disminuir un número. En Javascript contamos con una "variante" del bucle for, el for...in. Observemos el siguiente ejemplo :

```
 var persona = {
      nombre : "Horacio",
      edad : 28,
      esProfe : true
}

for(var indice in persona){
   console.log("En el indice "+indice+" tenemos el valor
"+persona[indice])
}

//En el indice nombre tenemos el valor Horacio
//En el indice edad tenemos el valor 28
//En el indice esProfe tenemos el valor true
```

El mismo nos permite declarar una variable “de control” la cual va a representar cada uno de los índices de nuestro vector salvo que en este caso no necesitamos de ninguna condición, el bucle sabe cuando tiene que dejar de iterar. Por otro lado, nótese el uso de corchetes para acceder a los elementos del vector. Esto se debe a que no podríamos escribirlo de la siguiente forma :

```
...
console.log(persona.indice) //undefined
...
```

Ya que de esa forma el intérprete pensaría que queremos obtener el valor de un índice LLAMADO “índice” el cual no se encuentra presente dentro de nuestro vector.

Resetear los elementos de un Array :

Si queremos borrar todo el contenido de un array, porque nos interesa cargarle nuevos valores, perdiendo los anteriores, entonces la sentencia a ejecutar es:

amigos.length = 0;

Le asignamos un valor CERO a la propiedad length, lo cual conlleva a indicarle al array que ya no posee NINGUN elemento.
A partir de este momento, podrán ingresarse los nuevos valores:

amigos[0] = ‘dario’;
amigos[1] = ‘marcelo’;
amigos[2] = ‘Claudia’;

Anexo JavaScript:

Observar un vector en consola:

```
Console.table(["Hola Mundo!", "foo", "bar"]);
```

Creación de bloques de texto:

```
var texto = document.createElement("p");
```

La función anterior nos crea un bloque de texto vacío al cual le podemos modificar su contenido.

Asignar contenido a un bloque de texto :

```
var texto = document.createElement("p");
```

```
texto.innerText = "Hola Mundo!";
```

Agregar bloques de texto al documento:

```
var texto = document.createElement("p");
texto.innerText = "Hola Mundo!";
document.appendChild(texto)
```

Modificar el color de fuente de bloques de texto :

```
var texto = document.createElement("p");
Texto.style.color = "red";
```