



Repaso de JavaScript

DigitalHouse>



**Certified Tech
Developer**

The Ultimate Degree

Índice

1. [Variables](#)
2. [Tipos de datos](#)
3. [Operadores](#)
4. [Funciones](#)
5. [Condicionales](#)
6. [Ciclos](#)

1 | Variables

“

Las **variables** son espacios de memoria en la computadora donde podemos almacenar distintos tipos de datos.



”

Tipos de variables

En JavaScript existen estos tipos de variables:

- let
- const

Para declarar una variable escribimos el tipo y el nombre que le queremos dar a la variable:

```
{  
  let contador;  
  const url;
```

Veamos cada parte con más detalle.

Declaración de una variable

```
let nombreSignificativo;
```



let

La palabra reservada **let** le indica a JavaScript que vamos a **declarar una variable de tipo let**.



Nombre

- Solo puede estar formado por letras, números y los símbolos \$ (pesos) y _ (guion bajo).
- No pueden empezar con un número.
- No deberían contener ñ o caracteres con acentos.



Es una buena práctica que los nombres de las variables usen el formato Camel case, como variableEjemplo en vez de variableejemplo o variable_ejemplo. Recordá que **JavaScript** es un lenguaje que **hace diferencia entre MAYÚSCULAS y minúsculas**.

Asignación de un valor

```
let miApodo = 'Hackerman';
```



Nombre

El nombre que nos va a servir para identificar nuestra variable cuando necesitemos usarla.

Asignación

Le indica a JavaScript que queremos guardar el valor de la derecha en la variable de la izquierda.

Valor

Lo que vamos a guardar en nuestra variable. En este caso, un texto.

Asignación de un valor

La **primera vez** que declaramos una variable es necesaria la palabra reservada **let**.

```
{ } let miApodo = 'Hackerman';
```

Una vez que la variable ya fue declarada, le asignamos valores **sin let**.

```
{ } miApodo = 'El Barto';
```


Declaración con const

Las variables **const** se declaran con la palabra reservada **const**.

```
{} const email = "mi.email@hotmail.com";
```

Las variables declaradas con **const** funcionan igual que las variables **let**, estarán disponibles solo en el bloque de código en el que se hayan declarado.

Al contrario de **let**, una vez que les asignemos un valor, no podremos cambiarlo.

```
email = "mi.otro.email@hotmail.com";  
{}  
// Error de asignación, no se puede cambiar  
// el valor de un const
```

Declaración con let o const

Como dijimos antes, tanto **let** como **const** son accesibles dentro del bloque donde son declaradas.

Por esta razón solo podemos declararlas una vez. Si volvemos a declararlas, JavaScript nos devolverá un error.

```
{  
  let contador = 0;  
  let contador = 1;  
  // Error de re-declaración de la variable  
  
  const email = "mi.email@hotmail.com";  
  const email = "mi.nuevo.email@hotmail.com";  
  // Error de re-declaración de la variable  
}
```

Declaración con var

Existe otra forma de declarar variables que **ya no se utiliza y no lo veremos en la materia**, pero que puede ser que la encuentres en códigos que tengan algunos años.

Estas variables se declaran de una manera similar, con la diferencia que utilizamos la palabra reservada **var**.

```
{ } var contador = 1;
```

La principal diferencia entre **var** y **let** es que **var** será accesible de manera global por todo nuestro código y no estará limitado el acceso a solo el bloque de código donde fue declarado como es el caso de **let**.

Por convención y buenas prácticas es recomendado el uso de **let**.

Los bloques de código son normalmente determinados por las llaves **{ }**.

“

Las **palabras reservadas** como **var**, **let** y **const** solo pueden utilizarse para el propósito que fueron creadas.

No pueden ser utilizadas como:
nombre de variables, funciones,
métodos o identificadores de objetos.

”



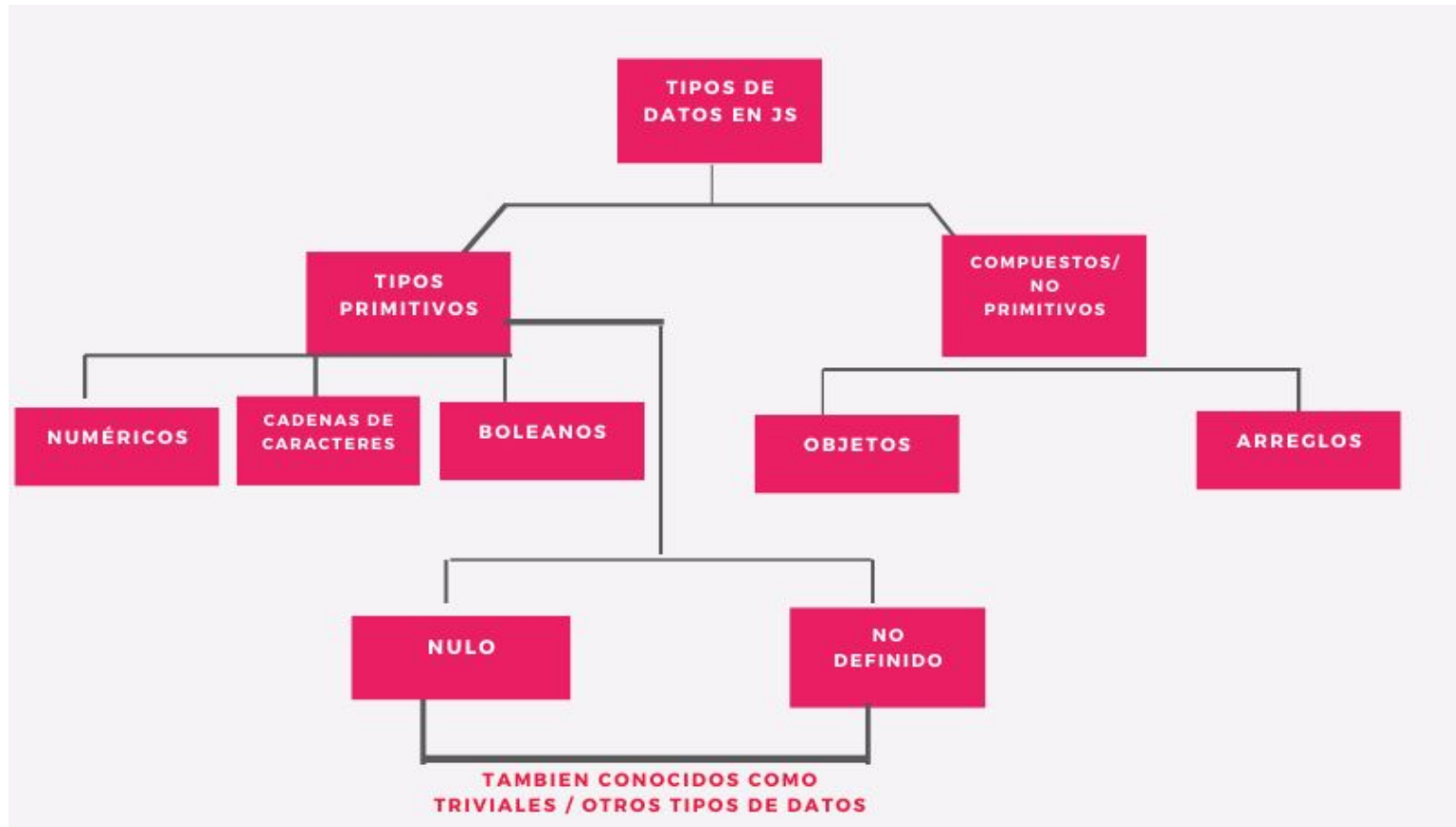
2 | Tipos de datos

“

Los **tipos de datos** le **permiten** a JavaScript **conocer** las **características** y **funcionalidades** que estarán disponibles **para ese dato**.



”



Numéricos (number)

```
{  
  let edad = 35; // número entero  
  let precio = 150.65; // decimales  
  let malaDivision = "35" / 2; // NaN - Not a Number, no es  
    un número aunque es un valor de "tipo" number  
}
```



Como JavaScript está escrito en inglés, usaremos un punto para separar los decimales.

Cadenas de caracteres (string)

```
{  
  let nombre = 'Mamá Luchetti'; // comillas simples  
  let ocupacion = "Master of the sopas"; // comillas  
    dobles tienen el mismo resultado  
}
```

Lógicos o booleanos (boolean)

```
{  
  let laCharlaEstaReCopada = true;  
  let hayAsadoAlFinal = false;  
}
```


Undefined (valor sin definir)

Indica la ausencia de valor.

Las variables tienen un valor indefinido hasta que les asignamos uno.

```
{}
```

```
let saludo; // undefined, no tiene valor  
saludo = "¡Hola!"; // Ahora si tiene un valor
```

Null (valor nulo)

Lo asignamos nosotros para indicar un valor vacío o desconocido.

```
{}
```

```
let temperatura = null; // No llegó un dato, algo falló
```

Array

Los arrays nos permiten generar una colección de datos ordenados. Cada dato de un array ocupa una posición numerada conocida como **índice**. La **primera posición** de un array es **siempre 0**.

```
{ } let pelisFavoritas = ['Star Wars', 'Kill Bill', 'Alien'];
```



0 1 2

Para acceder a un elemento puntual de un array, nombramos al array y, **dentro de los corchetes**, escribimos el índice al cual queremos acceder.

```
{ } pelisFavoritas[2];  
// accedemos a la película Alien, el índice 2 del array
```

Objeto literal

Un **objeto** es una estructura de datos que puede contener **propiedades** y **métodos**.

Para crearlo usamos llave de apertura y de cierre `{}`.

```
{} let auto = {  
    patente : 'AC 134 DD',  
};
```

PROPIEDAD

Definimos el nombre de la **propiedad** del objeto.

DOS PUNTOS

Separa el nombre de la propiedad de su valor.

VALOR

Puede ser cualquier **tipo de dato** que conocemos.

Métodos de un objeto

Una propiedad puede almacenar cualquier tipo de dato. Si una propiedad almacena una **función**, diremos que es un **método** del objeto. Con una estructura similar a la de las funciones expresadas, vemos que se crean mediante el nombre del método, seguido de una función anónima.

```
{  
  let tenista = {  
    nombre: 'Roger',  
    edad: 38,  
    activo: true,  
    saludar: function() {  
      return '¡Hola! Me llamo Roger';  
    }  
  };  
}
```

Ejecución de un método de un objeto

Para ejecutar un método de un objeto usamos la notación `objeto.metodo()`.

La palabra reservada **this** hace referencia al objeto en sí donde estamos parados. Es decir, el objeto en sí donde escribimos la palabra.

Con la notación `this.propiedad` accedemos al valor de cada propiedad interna de ese objeto.

```
{  
  let tenista = {  
    nombre: 'Roger',  
    apellido: 'Federer',  
    saludar: function() {  
      return '¡Hola! Me llamo ' + this.nombre;  
    }  
  };  
  console.log(tenista.saludar()); // ¡Hola! Me llamo Roger
```

NaN (Not-A-Number)

Indica que el valor pasado no es un número.

```
{}  
let nombre = "Esteban"  
console.log("Esteban"-1)
```

```
PS C:\Users\esteb\Desktop> node nan.js  
NaN
```

3 | Operadores

“

Los **operadores** nos permiten **manipular el valor** de las variables, realizar operaciones y comparar sus valores.



”

De asignación

Asignan el valor de la derecha en la variable de la izquierda.

```
{ } let edad = 35; // Asigna el número 35 a edad
```

Aritméticos

Nos permiten hacer operaciones matemáticas, devuelven el resultado de la operación. Siempre devolverán el resultado numérico.

```
{ } 10 + 15 // Suma → 25  
10 - 15 // Resta → -5  
10 * 15 // Multiplicación → 150  
15 / 10 // División → 1.5
```

Aritméticos (continuación)

Nos permiten hacer operaciones matemáticas, devuelven el resultado de la operación.

```
{ } 15++ // Incremento, es igual a 15 + 1 → 16  
15-- // Decremento, es igual a 15 - 1 → 14
```

```
{ } 15 % 5 // Módulo, el resto de dividir 15 entre 5 → 0  
15 % 2 // Módulo, el resto de dividir 15 entre 2 → 1
```

El operador de módulo % nos devuelve el resto de una división.

| | | | | | | | |
|----|--|---|--|----|--|---|--|
| 15 | | 5 | | 15 | | 2 | |
| | | | | | | | |
| 0 | | 3 | | 1 | | 7 | |

De concatenación

Sirven para unir cadenas de texto. Devuelven otra cadena de texto.

```
{}  
let nombre = 'Teodoro';  
let apellido = 'García';  
let nombreCompleto = 'Me llamo ' + nombre + ' ' + apellido;
```



Template literals

Existe otra forma de armar strings a partir de variables, y es con los template literals utilizando backtick en lugar de comillas y las variables entre llaves a continuación del símbolo \$. En este ejemplo lo escribiríamos así: `Me llamo \${nombre} \${apellido}`

Si mezclamos otros tipos de datos, estos se convierten a cadenas de texto.

```
{}  
let fila = 'M';  
let asiento = 7;  
let ubicacion = fila + asiento; // 'M7' como string
```

De comparación simple

Comparan dos valores, devuelven verdadero o falso.

```
{ } 10 == 15 // Igualdad → false  
10 != 15 // Desigualdad → true
```

De comparación estricta

Comparan el valor y el tipo de dato también.

```
{ } 10 === "10" // Igualdad estricta → false  
10 !== 15 // Desigualdad estricta → true
```

En el primer caso, el valor es 10 en ambos ejemplos, pero los tipos de datos son number y string. Como estamos comparando que ambos —valor y tipo de dato— sean iguales, el resultado es false.

De comparación (continuación)

Comparan dos valores, devuelven verdadero o falso.

```
{}  
15 > 15 // Mayor que → false  
15 >= 15 // Mayor o igual que → true  
10 < 15 // Menor que → true  
10 <= 15 // Menor o igual que → true
```

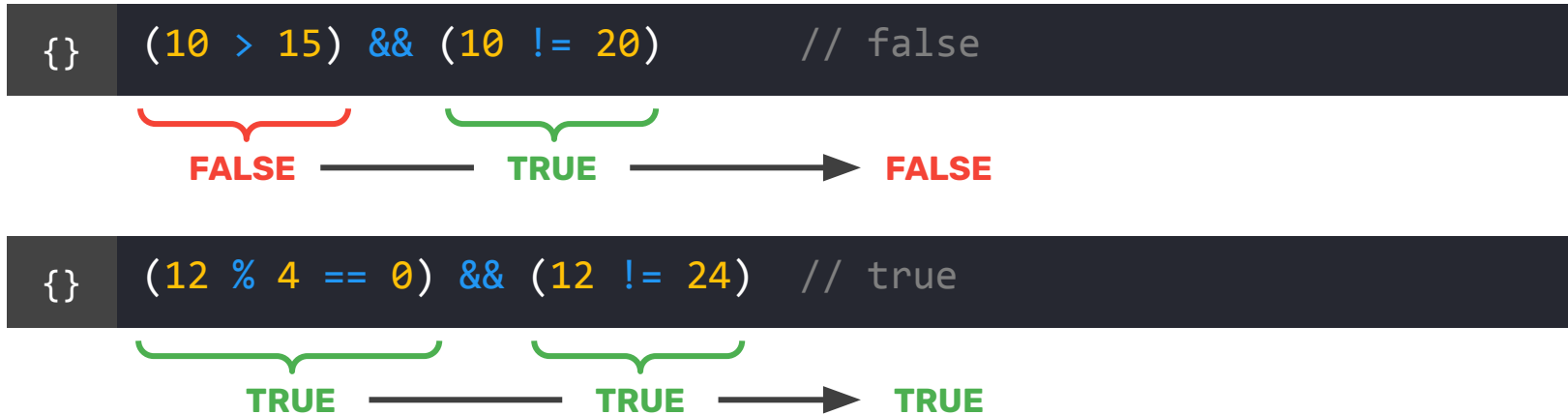


Siempre debemos escribir el símbolo mayor (>) o menor (<) antes que el igual (>= o <=). Si lo hacemos al revés (= > o =<), JavaScript lee primero el operador de asignación = y luego no sabe qué hacer con el mayor (>) o el menor (<).

Lógicos

Permiten combinar valores booleanos, el resultado también devuelve un booleano. Existen tres operadores y (**and**), o (**or**), negación (**not**).

AND (&&) → **todos** los valores deben evaluar como **true** para que el resultado sea true.



OR (||) → **al menos un** valor debe evaluar como **true** para que el resultado sea true.

```
{ } (10 > 15) || (10 != 20) // true
```


FALSE — TRUE → TRUE

```
{ } (12 % 5 == 0) && (12 != 12) // false
```


FALSE — FALSE → FALSE

NOT (!) → **niega la condición**. Si era true, será false y viceversa.

```
{ } !false // true  
{ } !(20 > 15) // false
```

4 | Funciones



Una función es un **bloque de código** que nos permite **agrupar una funcionalidad** para usarla todas las veces que necesitemos.

Normalmente, realiza una **tarea específica** y **retorna** un valor como resultado.



Estructura básica

```
{  
  function sumar (a, b) {  
    return a + b;  
  }  
}
```

- **Palabra reservada:** Usamos la palabra **function** para indicarle a JavaScript que vamos a escribir una función.
- **Nombre de la función:** Definimos un nombre para referirnos a nuestra función al momento de querer **invocarla**.
- **Parámetros:** Escribimos los paréntesis y, dentro de ellos, los parámetros de la función. Si hay más de uno, los separamos usando comas `,`. Si la función no lleva parámetros, igual debemos escribir los paréntesis sin nada adentro `()`.
- **Cuerpo:** Entre las llaves de apertura y de cierre escribimos la lógica de nuestra función, es decir, el código que queremos que se ejecute cada vez que la invoquemos.
- **El retorno:** Es muy común, a la hora de escribir una función, que queramos devolver al exterior el resultado del proceso que estamos ejecutando. Para eso utilizamos la palabra reservada `return` seguida de lo que queramos retornar.

Funciones declaradas

Son aquellas que se declaran usando la **estructura básica**. Pueden recibir un **nombre**, escrito a continuación de la palabra reservada **function**, a través del cual podremos invocarla.

Las funciones con nombre son funciones nombradas.

```
{  
  function saludar(nombre) {  
    return 'Hola' + nombre;  
  }  
}
```

Funciones expresadas

Son aquellas que **se asignan como valor** de una variable. En este caso, la función en sí no tiene nombre, es una **función anónima**.

Para invocarla podremos usar el nombre de la variable que declaremos.

```
{  
  let triplicar = function (numero) {  
    return numero * 3;  
  }  
}
```

Scope local

En el momento en que declaramos una variable dentro de una función, esta pasa a tener alcance local. Es decir, esa variable vive únicamente dentro de esa función.

Si quisiéramos hacer uso de la variable por fuera de la función, no vamos a poder, dado que fuera del **scope** donde fue declarada, esa variable no existe.

```
function saludar() {  
    // todo el código que escribamos dentro  
    // de nuestra función, tiene scope local  
}  
// No podremos acceder desde afuera a ese scope
```




Scope global

En el momento en que declaramos una variable **fuera** de cualquier función, la misma pasa a tener **alcance global**.

Es decir, podemos hacer uso de ella desde cualquier lugar del código en el que nos encontremos, inclusive dentro de una función, y acceder a su valor.

```
{  
  // todo el código que escribamos fuera  
  // de las funciones es global  
  function miFuncion() {  
    // Desde adentro de las funciones  
    // Tenemos acceso a las variables globales  
  }  
}
```



Arrow Function - Estructura básica

Pensemos en una función simple que podríamos programar de la manera habitual: una suma de dos números.

```
{ } function sumar (a, b) { return a + b }
```

Ahora veamos la versión reducida de esa misma función, al transformarla en una función arrow.

```
{ } let sumar = (a, b) => a + b;
```

Nombre de una función arrow

Las funciones arrow **son siempre anónimas**. Es decir, que no tienen **nombre** como las funciones normales.

```
{ } (a, b) => a + b;
```

Si queremos nombrarlas, es necesario escribirlas como una función expresada. Es decir, asignarla como valor de una variable.

```
{ } let sumar = (a, b) => a + b;
```

De ahora en más podremos llamar a nuestra función por su nuevo nombre.

Parámetros de una función arrow

Usamos paréntesis para indicar los **parámetros**. Si nuestra función no recibe parámetros, debemos escribirlos igual.

```
{ } let sumar = (a, b) => a + b;
```

Una particularidad de este tipo de funciones es que si recibe un único parámetro, podemos prescindir de los paréntesis.

```
{ } let doble = a => a * 2;
```

La flecha de una función arrow

La usamos para indicarle a JavaScript que vamos a escribir una función —reemplaza a la palabra reservada `function`—.

```
{ } let sumar = (a, b) => a + b;
```

Lo que está a la izquierda de la flecha será la entrada de la función —los parámetros— y lo que está a la derecha, la lógica —y el posible retorno—.

Cuerpo de una función arrow

Como ya vimos, si la función tiene una sola línea de código, y esta misma es la que hay que **retornar**, no hacen falta las llaves ni la palabra reservada return.

```
{} let sumar = (a, b) => a + b;
```

De lo contrario, vamos a necesitar utilizar ambas. Eso normalmente pasa cuando tenemos más de una línea de código en nuestra función.

```
{} let esMultiplo = (a, b) => {  
    let resto = a % b; // Obtenemos el resto de la div.  
    return resto == 0; // Si el resto es 0, es múltiplo  
};
```

5 | Condicionales

“

Nos permiten **evaluar condiciones** y realizar diferentes acciones según el resultado de esas evaluaciones.



”

Condicional **simple**

Versión más básica del `if`. Establece una condición y un bloque de código a ejecutar en caso de que sea verdadera.

```
{  
  if (condición) {  
    // código a ejecutar si la condición es verdadera  
  }  
}
```

Condicional con bloque **else**

Igual al ejemplo anterior, pero agrega un bloque de código a ejecutar en caso de que la condición sea falsa.

Es importante tener en cuenta que el bloque `else` es opcional.

```
if (condición) {  
    // código a ejecutar si la condición es verdadera  
} else {  
    // código a ejecutar si la condición es falsa  
}
```

Condicional con bloque **else if**

Igual que el ejemplo anterior, pero agrega un `if` adicional. Es decir, otra condición que puede evaluarse en caso de que la primera sea falsa.

Podemos agregar todos los bloques `else if` que queramos, solo uno podrá ser verdadero. De lo contrario, entrará en acción el bloque `else`, si existe.

```
if (condición) {  
    // código a ejecutar si la condición es verdadera  
} else if (otra condición) {  
    // código a ejecutar si la otra condición es verdadera  
} else {  
    // código a ejecutar si todas las condiciones son falsas  
}
```


6 | Ciclos

“

Los **ciclos** nos permiten **repetir instrucciones** de manera sencilla. Podemos hacerlo una determinada **cantidad de veces** o mientras se cumpla una **condición**.



”

Ciclo FOR

Consta de **tres partes** que definimos dentro de los paréntesis. En conjunto, nos permiten determinar de qué manera se van a realizar las **repeticiones** y definir las **instrucciones** que queremos que se lleven a cabo en cada una de ellas.

```
{  
  for (inicio; condición ; modificador) {  
    //código que se ejecutará en cada repetición  
  }  
}
```

- **Inicio:** antes de arrancar el ciclo, se establece el valor inicial de nuestro contador.
- **Condición:** antes de ejecutar el código en cada vuelta, se pregunta si la condición resulta verdadera o falsa. Si es **verdadera**, continúa con nuestras instrucciones. Si es **falsa**, detiene el ciclo.
- **Modificador —incremento o decremento—:** luego de ejecutar nuestras instrucciones, se modifica nuestro contador de la manera que hayamos especificado. En este caso se le suma 1, pero podemos hacer la cuenta que queramos.

Estructura básica FOR

En este ejemplo vamos a contar desde 1 hasta 5 inclusive.

```
{  
  for (let vuelta = 1; vuelta <= 5; vuelta++) {  
    console.log('Dando la vuelta número ' + vuelta);  
  };  
}
```



```
Dando la vuelta número 1  
Dando la vuelta número 2  
Dando la vuelta número 3  
Dando la vuelta número 4  
Dando la vuelta número 5
```

Ciclo WHILE

Tiene una estructura similar a la de los condicionales **if/else**, palabra reservada + condición entre paréntesis. Sin embargo, el **while Loop** reevalúa esa condición repetidas veces y **ejecuta su bloque de código** hasta que la condición **deja de ser verdadera**.

```
while (condicion) {  
    //código que se ejecutará en cada repetición  
}  
    // Hace algo para que la condición eventualmente se deje  
    de cumplir  
}
```

Estructura básica WHILE

Tomando el ejemplo utilizado con el for, veamos como sería utilizando while.

```
{  
  let vuelta = 1  
  while(vuelta <= 5) {  
    console.log('Dando la vuelta número ' + vuelta);  
    vuelta++ //al final de cada vuelta sumara 1 a vuelta  
  };  
}
```



```
Dando la vuelta número 1  
Dando la vuelta número 2  
Dando la vuelta número 3  
Dando la vuelta número 4  
Dando la vuelta número 5
```

Estructura básica WHILE (cont.)

Es importante generar el contador al comenzar para evitar caer en lo que se conoce como **loop infinito**.

```
{}  
    let vuelta = 1  
    while(vuelta <= 5) {  
        console.log('Dando la vuelta número ' + vuelta);  
        vuelta++  
    };
```

El **loop infinito** sucede cuando nuestra condición es constantemente verdadera, lo que resulta en ejecutar nuestro código eternamente. Esto puede causar varios problemas, siendo el más importante que trabe todo nuestro programa.

Ciclo DO WHILE

A diferencia del ciclo **while**, la condición en este caso se verifica al finalizar el bloque de código. Por lo tanto, sin importar lo que se resuelva, las acciones se realizarán al menos una vez.

```
{  
  let vuelta = 5  
  do{  
    console.log('Dando la vuelta número ' + vuelta);  
    vuelta++ //Se suma 1 a vuelta por lo tanto vuelta = 6  
  } while(vuelta <= 5); //al vuelta ser 6 la condición retorna  
  false y se termina el bloque de código
```

Fuera de esto, el ciclo **do while** es idéntico en funcionalidad al ciclo **while**.

DigitalHouse>