Conversiones

Number()

```
Number(true);       //1
Number(false);      //0
Number(13);         //13
Number("13");       //13
var b;
Number(b);          //Number(undefined) NaN
var b=10;
Number(b);          //10
Number("1.5");      //1.5
Number("01.5");     //1.5
Number("0xA");      //10
Number("");         //0
Number("cindy");    //NaN
Number("123cindy"); //NaN
Number("123.123.123");  //NaN
Number("4.89e7");   //48900000
```

parseInt()

```
parseInt(true);     //NaN
parseInt(false);    //NaN
parseInt(13);       //13
parseInt("13");     //13
var k;              //k is undefined
parseInt(k);        //NaN
var b=10;
parseInt(b);        //10
parseInt("1.5")     //1
parseInt("01.5")    //1
parseInt("0xA");    //10
parseInt("");       //NaN
parseInt("cindy");  //NaN
parseInt("123cindy") //123
parseInt("123.123.123") //123
parseInt("4.89e7")  //4
parseInt("10",16);  //hexadecimal 16
parseInt("10",8);   //octal 8
parseInt("10",10);  //decimal 10
```

parseFloat()

```
parseFloat(true);      //NaN
parseFloat(false);     //NaN
parseFloat(13);        //13
parseFloat("13");      //13
var j;                 //undefined
parseFloat(j);         //NaN
var b=10;
```

```
parseFloat(b)          //10
parseFloat("1.5")      //1.5
parseFloat("01.5")     //1.5
parseFloat("0xA")      //0  only uses decimal system
parseFloat("")         //NaN
parseFloat("cindy");   //NaN
parseFloat("123cindy") //123.123
parseFloat("4.89e7");  //48900000
```

Operadores

Fuente: lenguajejs.com//Documentación de Mozilla

Operador de Asignación

Un operador de asignación asigna un valor a su operando izquierdo basándose en el valor de su operando derecho. El operador de asignación simple es igual (=), que asigna el valor de su operando derecho a su operando izquierdo. Es decir, x = y asigna el valor de y a x.

También hay operadores de asignación compuestos que son una abreviatura de las operaciones enumeradas en la siguiente tabla:

Nombre	Operador abreviado	Significado
Asignación	**x = y**	**x = y**
Asignación de adición	**x += y**	**x = x + y**
Asignación de resta	**x -= y**	**x = x - y**

Asignación de multiplicación	**x *= y**	**x = x * y**
Asignación de división	**x /= y**	**x = x / y**
Asignación de residuo	**x %= y**	**x = x % y**
Asignación de exponenciación	**x **= y**	**x = x ** y**

Un operador de comparación compara sus operandos y devuelve un valor lógico en función de si la comparación es verdadera (true) o falsa (false). Los operandos pueden ser valores numéricos, de cadena, lógicos u objetos. Las cadenas se comparan según el orden lexicográfico estándar, utilizando valores Unicode. En la mayoría de los casos, si los dos operandos no son del mismo tipo, JavaScript intenta convertirlos a un tipo apropiado para la comparación. Este comportamiento generalmente resulta en comparar los operandos numéricamente. Las únicas excepciones a la conversión de tipos dentro de las comparaciones involucran a los operadores === y !==, que realizan comparaciones estrictas de igualdad y desigualdad. Estos operadores no intentan convertir los operandos a tipos compatibles antes de verificar la igualdad. La siguiente tabla describe los operadores de comparación en términos de este código de ejemplo:

```
let var1 = 3;

let var2 = 4;
```

Operador	Descripción	Ejemplos que devuelven true
Igual (==)	Devuelve true si los operandos son iguales.	3 == var1 "3" == var1 3 == '3'
No es igual (!=)	Devuelve true si los operandos *no* son iguales.	var1 != 4 var2 != "3"
Estrictamente igual (===)	Devuelve true si los operandos son iguales y del mismo tipo.	3 === var1
Desigualdad estricta (!==)	Devuelve true si los operandos son del mismo tipo pero no iguales, o son de diferente tipo.	var1 !== "3" 3 !== '3'
Mayor que (>)	Devuelve true si el operando izquierdo es mayor que el operando derecho.	var2 > var1 "12" > 2

Mayor o igual que (>=)	**Devuelve true si el operando izquierdo es mayor o igual que el operando derecho.**	**var2 >= var1** **var1 >= 3**
Menor que (<)	**Devuelve true si el operando izquierdo es menor que el operando derecho.**	**var1 < var2** **"2" < 12**
Menor o igual (<=)	**Devuelve true si el operando izquierdo es menor o igual que el operando derecho.**	**var1 <= var2** **var2 <= 5**

Un operador aritmético toma valores numéricos (ya sean literales o variables) como sus operandos y devuelve un solo valor numérico. Los operadores aritméticos estándar son suma (+), resta (-), multiplicación (*) y división (/). Estos operadores funcionan como lo hacen en la mayoría de los otros lenguajes de programación cuando se usan con números de punto flotante (en particular, ten en cuenta que la división entre cero produce Infinity). Por ejemplo:

```
1 / 2; // 0.5

1 / 2 == 1.0 / 2.0; // Esto es true
```

Además de las operaciones aritméticas estándar (+, -, *, /), JavaScript proporciona los operadores aritméticos enumerados en la siguiente tabla:

Operador	Descripción	Ejemplo
Residuo (%)	Operador binario. Devuelve el resto entero de dividir los dos operandos.	12 % 5 devuelve 2.
Incremento (++)	Operador unario. Agrega uno a su operando. Si se usa como operador prefijo (++x), devuelve el valor de su operando después de agregar uno; si se usa como operador sufijo (x++), devuelve el valor de su operando antes de agregar uno.	Si x es 3, ++x establece x en 4 y devuelve 4, mientras que x++ devuelve 3 y , solo entonces, establece x en 4.
Decremento (--)	Operador unario. Resta uno de su operando. El valor de retorno es análogo al del operador de incremento.	Si x es 3, entonces --x establece x en 2 y devuelve 2, mientras que x-- devuelve 3 y, solo entonces, establece x en 2.
Negación unaria (-)	Operador unario. Devuelve la negación de su operando.	Si x es 3, entonces -x devuelve -3.
Positivo unario (+)	Operador unario. Intenta convertir el operando en un número, si aún no lo es.	+"3" devuelve 3.

		+true devuelve 1.
Operador de exponenciación (**)	**Calcula la base a la potencia de exponente, es decir, baseexponente**	**2 ** 3 returns 8.** **10 ** -1 returns 0.1.**

Los operadores lógicos se utilizan normalmente con valores booleanos (lógicos); cuando lo son, devuelven un valor booleano. Sin embargo, los operadores && y || en realidad devuelven el valor de uno de los operandos especificados, por lo que si estos operadores se utilizan con valores no booleanos, pueden devolver un valor no booleano. Los operadores lógicos se describen en la siguiente tabla.

Operador	Uso	Descripción
AND Lógico (&&)	**expr1 && expr2**	**Devuelve expr1 si se puede convertir a false; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, && devuelve true si ambos operandos son true; de lo contrario, devuelve false.**
OR lógico (||)	**expr1 || expr2**	**Devuelve expr1 si se puede convertir a true; de lo contrario, devuelve expr2. Por lo tanto, cuando se usa con valores booleanos, || devuelve true si**

		alguno de los operandos es true; si ambos son falsos, devuelve false.
NOT lógico (!)	**!expr**	**Devuelve false si su único operando se puede convertir a true; de lo contrario, devuelve true.**

Ejemplos de expresiones que se pueden convertir a false son aquellos que se evalúan como null, 0, NaN, la cadena vacía ("") o undefined.

El siguiente código muestra ejemplos del operador && (AND lógico).

```
let a1 = true && true; // t && t devuelve true

let a2 = true && false; // t && f devuelve false

let a3 = false && true; // f && t devuelve false
```

El siguiente código muestra ejemplos del operador || (OR lógico).

```
let o1 = true || true; // t || t devuelve true

let o2 = false || true; // f || t devuelve true

let o3 = true || false; // t || f devuelve true

let o4 = false || (3 == 4); // f || f devuelve false
```

El siguiente código muestra ejemplos de el operador ! (NOT lógico).

```
let n1 = !true; // !t devuelve false

let n2 = !false; // !f devuelve true
```

Condicionales

Al hacer un programa necesitaremos establecer **condiciones** o **decisiones**, donde buscamos que el navegador realice una **acción A** si se **cumple** una condición o una **acción B** si **no se cumple**. Este es el primer tipo de estructuras de control que encontraremos. Para ello existen varias estructuras de control:

Estructura de control	Descripción
If	**Condición simple**: Si ocurre algo, haz lo siguiente...
If/else	**Condición con alternativa**: Si ocurre algo, haz esto, sino, haz lo esto otro...
?:	**Operador ternario**: Equivalente a If/else, método abreviado.
Switch	**Estructura para casos específicos**: Similar a varios If/else anidados.

Condicional If

Quizás, el más conocido de estos mecanismos de estructura de control es el **if** (*condicional*). Con él podemos indicar en el programa que se tome un camino sólo si se cumple la **condición** que establezcamos:

```
let nota = 7;

console.log("He realizado mi examen.");

// Condición (si nota es mayor o igual a 5)
if (nota >= 5) {
  console.log("¡Estoy aprobado!");
}
```

En este caso, como el valor de **nota** es superior a 5, nos aparecerá en la consola el mensaje «¡Estoy aprobado!». Sin embargo, si modificamos en la primera línea el valor de **nota** a un valor inferior a 5, no nos aparecerá ese mensaje.

Condicional If / else

Pero se puede dar el caso que queramos establecer una alternativa a una condición. Para eso utilizamos el **if** seguido de un **else**. Con esto podemos establecer una acción A si se cumple la condición, y una acción B si no se cumple.

Vamos a modificar el ejemplo anterior para mostrar también un mensaje cuando estamos suspendidos, pero en este caso, en lugar de mostrar el mensaje directamente con un **console.log** vamos a guardar ese texto en una nueva variable **calificacion**:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Condición
if (nota < 5) {
  // Acción A (nota es menor que 5)
  calificacion = "suspendido";
} else {
  // Acción B: Cualquier otro caso a A (nota es mayor o igual que 5)
  calificacion = "aprobado";
}
```

```
console.log("Estoy", calificacion);
```

Nuevamente, en este ejemplo comprobaremos que podemos conseguir que se muestre el mensaje **Estoy aprobado** o **Estoy suspendido** dependiendo del valor que tenga la variable **nota**. La diferencia con el ejemplo anterior es que creamos una nueva variable que contendrá un valor determinado dependiendo de la condición del **If**.

Por último, el **console.log** del final, muestra el contenido de la variable **calificacion**, independientemente de que sea el primer caso o el segundo.

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Condición
if (nota < 5) {
  // Acción A (nota es menor que 5)
  calificacion = "suspendido";
}
if (nota >= 5) {
  // Acción B (nota es mayor o igual que 5)
  calificacion = "aprobado";
}

console.log("Estoy", calificacion);
```

Este nuevo ejemplo, es equivalente al ejemplo anterior. Si nos fijamos bien, la única diferencia respecto al anterior es que estamos realizando dos **if** independientes: uno para comprobar si está suspendido y otro para comprobar si está aprobado.

Pero aunque son equivalentes, no son exactamente iguales, ya que en el ejemplo que vimos anteriormente sólo existe **un if**, y por lo tanto, sólo se realiza una comprobación. En este ejemplo que vemos ahora, se realizan **dos if**, y por lo tanto, dos comprobaciones.

> En este caso se trata de algo insignificante, pero es importante darse cuenta de que el primer ejemplo estaría realizando menos tareas para conseguir un mismo resultado, ergo, el primer ejemplo sería más eficiente.

Operador ternario

El **operador ternario** es una alternativa de condicional **if/else** de una forma mucho más corta y, en muchos casos, más legible. Vamos a reescribir el ejemplo anterior utilizando este operador:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Operador ternario: (condición ? verdadero : falso)
let calificacion = nota < 5 ? "suspendido" : "aprobado";

console.log("Estoy", calificacion);
```

Este ejemplo hace exactamente lo mismo que el ejemplo anterior. La idea del operador ternario es que podemos condensar mucho código y tener un if en una sola línea. Obviamente, es una opción que sólo se recomienda utilizar cuando son **if** muy pequeños.

Condicional If múltiple

Es posible que necesitemos crear un condicional múltiple con más de 2 condiciones, por ejemplo, para establecer la calificación específica. Para ello, podemos anidar varios **if/else** uno dentro de otro, de la siguiente forma:

```
let nota = 7;
console.log("He realizado mi examen.");

// Condición
if (nota < 5) {
  calificacion = "Insuficiente";
} else if (nota < 6) {
  calificación = "Suficiente";
} else if (nota < 8) {
  calificacion = "Bien";
} else if (nota <= 9) {
  calificacion = "Notable";
} else {
  calificacion = "Sobresaliente";
}

console.log("He obtenido un", calificacion);
```

Sin embargo, anidar de esta forma varios **if** suele ser muy poco legible y produce un código algo feo.

Condicional Switch

La estructura de control **switch** permite definir casos específicos a realizar en el caso de que la variable expuesta como condición sea igual a los valores que se especifican a continuación mediante los **case**. No obstante, hay varias puntualizaciones que aclarar sobre este ejemplo:

```
let nota = 7;
console.log("He realizado mi examen. Mi resultado es el siguiente:");

// Nota: Este ejemplo NO es equivalente al ejemplo anterior (leer abajo)
switch (nota) {
  case 10:
calificacion = "Sobresaliente";
    break;
  case 9:
  case 8:
    calificacion = "Notable";
    break;
  case 7:
  case 6:
    calificacion = "Bien";
    break;
  case 5:
    calificacion = "Suficiente";
    break;
  case 4:
  case 3:
  case 2:
  case 1:
  case 0:
    calificacion = "Insuficiente";
    break;
  default:
    // Cualquier otro caso
    calificacion = "Nota errónea";
    break;
}

console.log("He obtenido un", calificacion);
```

En primer lugar, el ejemplo anterior **no es exactamente equivalente al anterior**. Este ejemplo funcionaría si sólo permitimos notas que sean **números enteros**, es decir, números del 0 al 10, sin decimales. En el caso de que nota tuviera por ejemplo, el valor **7.5**, mostraría **Nota errónea**.

> El ejemplo de los **if múltiples** si controla casos de números decimales porque establecemos comparaciones de rangos con mayor o menor, cosa que con el **switch** no se puede hacer. El **switch** está indicado para utilizar sólo con **casos con valores concretos y específicos**.

En segundo lugar, observa que al final de cada caso es necesario indicar un **break** para salir del **switch**. En el caso que no sea haga, el programa saltará al siguiente caso, aunque no se cumpla la condición específica.

Una de las principales ventajas de la programación es la posibilidad de crear **bucles y repeticiones** para tareas específicas, y que no tengamos que realizarlas varias veces de forma manual. Existen muchas formas de realizar bucles, vamos a ver los más basicos, similares en otros lenguajes de programación:

Bucles

Tipo de bucle	Descripción
while	Bucles simples.
for	Bucles clásicos por excelencia.
do..while	Bucles simples que se realizan siempre como mínimo una vez.
for..in	Bucles sobre posiciones de un array.

for..of	Bucles sobre elementos de un array. Los veremos más adelante.
Array functions	Bucles específicos sobre arrays. Los veremos más adelante.

Antes de comenzar a ver que tipos de bucles existen en Javascript, es necesario conocer algunos conceptos básicos de los bucles:

- **Condición**: Al igual que en los **if**, en los bucles se va a evaluar una condición para saber si se debe repetir el bucle o finalizarlo. Generalmente, si la condición es verdadera, se repite. Si es falsa, se finaliza.
- **Iteración**: Cada repetición de un bucle se denomina iteración. Por ejemplo, si un bucle repite una acción 10 veces, se dice que tiene 10 iteraciones.
- **Contador**: Muchas veces, los bucles tienen una variable que se denomina contador, porque cuenta el número de repeticiones que ha hecho, para finalizar desde que llegue a un número concreto. Dicha variable hay que inicializarla (*crearla y darle un valor*) antes de comenzar el bucle.
- **Incremento**: Cada vez que terminemos un bucle se suele realizar el incremento (*o decremento*) de una variable, generalmente la denominada variable contador.
- **Bucle infinito**: Es lo que ocurre si en un bucle se nos olvida incrementar la variable contador o escribimos una condición que nunca se puede dar. El bucle se queda eternamente repitiéndose y el programa se queda «colgado».

Bucle while

El bucle **while** es uno de los bucles más simples que podemos crear. Vamos a repasar el siguiente ejemplo y todas sus partes, para luego repasar que ocurre en cada iteración del bucle:

```
let i = 0; // Inicialización de la variable contador
```

```
// Condición: Mientras la variable contador sea menor de 5
while (i < 5) {
  console.log("Valor de i:", i);

  i = i + 1; // Incrementamos el valor de i
}
```

Veamos qué es lo que ocurre a la hora de hacer funcionar ese código:

- Antes de entrar en el bucle **while**, se inicializa la variable **i** a **0**.
- Antes de realizar la primera **iteración** del bucle, comprobamos la **condición**.
- Si la condición es **verdadera**, hacemos lo que está dentro del bucle.
- Mostramos por pantalla el valor de **i** y luego incrementamos el valor actual de **i** en **1**.
- Volvemos al inicio del bucle para hacer una **nueva iteración**. Comprobamos de nuevo la **condición** del bucle.
- Cuando la condición sea **falsa**, salimos del bucle y continuamos el programa.

Una muestra paso a paso de las iteraciones de este primer ejemplo:

Iteración del bucle	Valor de i	Descripción	Incremento
Antes del bucle	i = undefined	Antes de comenzar el programa.	
Iteración #1	i = 0	¿(0 < 5)? Verdadero. Mostramos 0 por pantalla.	i = 0 + 1
Iteración #2	i = 1	¿(1 < 5)? Verdadero. Mostramos 1 por pantalla.	i = 1 + 1

Iteración #3	i = 2	¿(2 < 5)? Verdadero. Mostramos 2 por pantalla.	i = 2 + 1
Iteración #4	i = 3	¿(3 < 5)? Verdadero. Mostramos 3 por pantalla.	i = 3 + 1
Iteración #5	i = 4	¿(4 < 5)? Verdadero. Mostramos 4 por pantalla.	i = 4 + 1
Iteración #6	i = 5	¿(5 < 5)? Falso. Salimos del bucle.	

El bucle **while** es muy simple, pero requiere no olvidarse accidentalmente de la inicialización y el incremento (*además de la condición*), por lo que el bucle **for** resulta más interesante, ya que para hacer un bucle de este tipo hay que escribir previamente siempre estos tres factores.

> La operación **i = i + 1** es lo que se suele llamar un incremento de una variable. Es muy común simplificarla como **i++**, que hace exactamente lo mismo: aumenta en 1 su valor.

Bucle do-while

El bucle **do-while** siempre se ejecuta al menos una vez, la comprobación del cumplimiento de la condición se da al final del bucle.

```
let i = 0; // Inicialización de la variable contador

// Condición: Mientras la variable contador sea menor de 5
do{
  console.log("Valor de i:", i);

  i = i + 1; // Incrementamos el valor de i
} while (i < 5);
```

Bucle for

El bucle **for** es quizás uno de los más utilizados en el mundo de la programación. En Javascript se utiliza exactamente igual que en otros lenguajes como Java o C/C++. Veamos el ejemplo anterior utilizando un bucle for:

```
// for (inicialización; condición; incremento)
for (let i = 0; i < 5; i++) {
  console.log("Valor de i:", i);
}
```

Como vemos, la sintaxis de un **bucle for** es mucho más compacta y rápida de escribir que la de un **bucle while**. La primera vez puede parecer algo confusa, pero es mucho más práctica porque te obliga a escribir la **inicialización**, la **condición** y el **incremento** antes del propio bucle, y eso hace que no te olvides de estos tres puntos fundamentales.

> En programación es muy habitual empezar a contar desde **cero**. Mientras que en la vida real se contaría **desde 1 hasta 10**, en programación se contaría **desde 0 hasta 9**.

Incremento múltiple

Aunque no suele ser habitual, es posible añadir varias inicializaciones o incrementos en un bucle for separando por comas. En el siguiente ejemplo además de aumentar el valor de una variable **i**, inicializamos una variable con el valor **5** y lo vamos decrementando:

```
for (let i = 0, j = 5; i < 5; i++, j--) {
  console.log("Valor de i y j:", i, j);
}
```

Si **i++** aumenta en 1 el valor de **i** en cada iteración, lo que hace **j--** es disminuir en 1 el valor de **j** en cada iteración.