

Desenvolvimento Web

JavaScript para desenvolvimento web



Índice

Introdução	4
Características do Javascript	5
Onde vai o Javascript?	5
Na head	5
Em uma etiqueta	5
Linkar uma página Javascript ao nosso HTML	5
Entrada e saída de dados	5
Prompts()	6
Alert()	6
Alerta normal	6
Alertas de confirmação	6
console.log()	6
Document.write()	7
Estruturas básicas	7
Declarações	7
Funções pré-definidas	8
Comentários	8
Tipos de dados	8
Modo estrito	8
Declaração variável	8
Casting do variável	10
Constantes	10
Exemplos de operações com variáveis	10
Strings	12
Funções de strings	12
Modelos de strings	14
Operadores	14
Operadores de Atribuição	14
Operadores de Incremento e Decremento	15
Operadores Lógicos	16
Negação	16
AND	17
OR	17
Operadores Matemáticos	18
Operadores Relacionais	18
Condições e loops em JavaScript	20
Condicionais	21
Comparadores	21
IF...ELSE	22
SWITCH...CASE	23

Loops	24
O loop dado para FOR	24
WHILE. DO...WHILE	26
Break	28
Arrays	28
Funções da arrays	29
Travessar arrays com for	30
Travessar arrays com for each	31
Travessar arrays com for in	31
Buscando em um array	31
Arrays multidimensionais	32
POO	32
Funções	33
Parâmetros de rest y spread	36
Parâmetros de spread	36
Funções anónimas	36
Callback	36
Funções das setas	39
Eventos	40
Manipuladores de eventos especificados no código HTML	40
Manipuladores de eventos associados ao addEventListener	42
Apêndice do evento Javascript	43
DOM	44
Seleção de clases e etiquetas	45
BOM	46

Introdução

Começamos neste tópico com a terceira parte do padrão **HTML5, javascript**.

Uma página web **HTML5** será composta de três partes bem definidas:

- **O código HTML:** Composto de tags **HTML** que formarão o esqueleto de nossa página WEB.
- **O código CSS:** Com o qual daremos um aspecto gráfico à nossa página WEB.
- **O código Javascript:** Com o qual forneceremos animação e interatividade à nossa página WEB.

Características do Javascript

- Funciona localmente.
- Ela é interpretada, não compilada.
- Ela é responsiva.
- Adiciona interatividade aos websites.
- Proporciona efeitos visuais dinâmicos.

Onde vai o Javascript?

Para "linkar" uma página **javascript** a **html**, você pode colocá-la na **head**, em qualquer tag ou "linkar" a página inteira (similar a como fizemos com o **CSS**);

Na head

Entre `<script></script>` tags.

```
<head>
  <meta charset="UTF-8">
  <title>Ejemplo de HEAD con
javascript</title>
  <script>
    alert("Hola mundo");
  </script>
</head>
```

Nota: Se nossa página vai ter tanto **CSS** quanto **Javascript** na head, é comum colocar o código **CSS** em primeiro lugar, pois é muito comum que o **script** se refira a elementos **CSS** e eles já devem ser carregados na memória quando a execução atinge a parte **javascript**.

Em uma etiqueta

```
<p onClick="alert('Hola
mundo');">Página de prueba Javascript</p>
```

Com o comando acima, emitimos uma mensagem de alerta dizendo "Hola mundo".

Linkar uma página Javascript ao nosso HTML

Neste caso, criamos a página **Javascript** e no **head** de nossa página **HTML** colocamos:

```
<head>
  <meta charset="UTF-8" />
  <title>Ejemplo de página con
Javascript externo</title>
  <script
src="nombre_de_la_página_Javascript.js"><
/script>
</head>
```

Dessa forma, o conteúdo do arquivo .js será anexado à nossa página web.

Entrada e saída de dados

Para a entrada de dados, o javascript tem duas ferramentas:

- O comando `prompt()`
- Entrada de formulário HTML

Prompts()

Com `prompt`, inserimos a variável através de uma janela.

```
var nombre=prompt("Introduce nombre ",
"nombre por defecto");
```

Neste exemplo, o que o usuário entrar na janela que será aberta será armazenado da variável `nombre`.

Eu posso passar dois parâmetros para a função `prompt()`. O primeiro parâmetro é o texto a ser exibido e o segundo é o valor padrão.

```
var edad = prompt("Que edad tienes?:", 18);
```

Importante: Tudo o que for inserido por `prompt()` será sempre considerado como `string`, mesmo que sejam números. Se eu quiser poder operar com esses números, tenho que fazer a conversão com `parseInt`, `parseFloat`... etc.

Para a saída de dados, temos três opções:

- O comando `alert()`
- O comando `console.log()`
- O comando `document.write()`

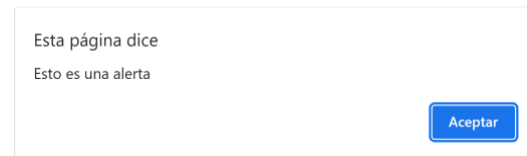
Alert()

O comando `alert()` abrirá uma janela pop-up com um aviso. Há dois tipos de `alert()`:

Alerta normal

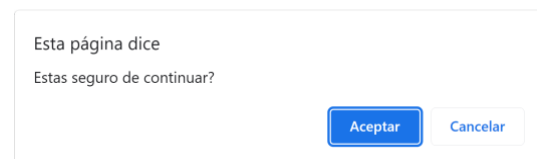
```
alert("Esto es una alerta");
```

O seguinte aparecerá no navegador:



Alertas de confirmação

```
var confirmacion = confirm("Estas
seguro de continuar?");
```

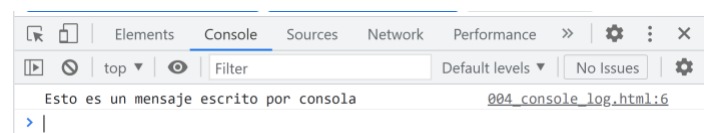


Na `confirmacion` variável eu guardo a resposta do usuário, que será `true` ou `false`, dependendo se ele aceita ou cancela.

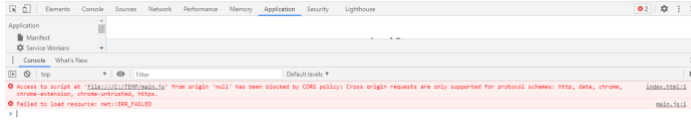
console.log()

Este comando exibirá uma mensagem no console do navegador. Ele não será visível a menos que abramos o console. Na maioria dos navegadores, isto é feito com `F12`.

```
console.log("Esto es un mensaje
escrito por consola");
```



O console nos permite trabalhar com JavaScript instantaneamente, corrigir erros, ver quais variáveis estão dentro, ver o que está acontecendo em nosso código, ver para onde o fluxo do programa está indo e assim por diante.



Document.write()

Com `document.write()`, podemos escrever diretamente no código **HTML** de nossa página.

Um **String** é retornado se o usuário clicar em "OK", o valor de entrada é retornado. Se o usuário clicar em "cancelar", o **NULL** é devolvido. Se o usuário clicar em "aceitar" sem inserir nenhum texto, uma cadeia vazia é devolvida.

```
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var nombre = "Angel";
    var apellido = "García";
    document.write("<h1>Tu nombre
es</h1>" + nombre + " " + apellido);
  </script>
</head>
```

Exemplos de entrada e saída de dados

```
//Salida por ventana emergente
alert("Esto es una ventana
emergente.");

//Entrada de datos
var edad = prompt("Introduzca edad:
");

// Salida de datos leyendo desde una
variable
alert("La edad introducida es: " +
edad);

// Salida de datos por consola
console.log("Ha terminado el
programa");

// Salida de datos al BODY de mi
página
document.write("<h1>Esto es un H1
puesto desde javascript</h1>");
```

Estruturas básicas

Declarações

Chamamos cada "comando" que programamos em **Javascript** de uma declaração. Eles terminam com `","`.

```
alert("Esto es una sentencia ");
document.write("Esta es otra ");
```

Funções pré-definidas

Estes são comandos que vêm por padrão em **Javascript** prontos para serem usados. Eles sempre têm parênteses.

```
alert();  
document.write();
```

Além dessas funções predefinidas, podemos criar nossas próprias funções.

Comentários

Em **Javascript** temos dois tipos de comentários, uma e várias linhas:

```
//      Comentário de uma linha.  
/*...*/ Comentário de várias linhas.
```

```
//      Comentario de una sola línea.  
  
/* Comentario de varias líneas.  
Siempre va entre  
los signos de asterisco y barra  
invertida */
```

Tipos de dados

O **Javascript** tem apenas três tipos básicos:

- **Valores numéricos:** números inteiros ou números em ponto flutuante (com casas decimais).
- **Strings:** caracteres alfanuméricos, ou seja, palavras ou frases.
- **Booleans:** Com dois valores possíveis, **True** ou **False**.

Modo estrito

Em **javascript** temos um modo que, quando ativado, nos permite ser mais restritivos em nosso modo de programação, para isso, no início de nossa página, colocaremos:

'use strict'

Por exemplo, sem este modo ativo, poderíamos definir uma variável com a seguinte sintaxe:

```
num=5;
```

Com o modo estrito ativo, teríamos que definir:

```
var num=5;
```

Além disso, existem certas funcionalidades que só funcionam com o modo estrito ativado.

Declaração variável

Por definição, uma variável é um espaço na memória do computador onde um valor será armazenado, que pode mudar durante a execução do programa.

As variáveis podem ser declaradas de duas maneiras, com **let** e com **var**:

- **Let:** permite declarar variáveis limitando seu escopo ao bloco ou declaração onde elas estão sendo utilizadas. Ou seja, as variáveis declaradas com **let** só podem ser utilizadas a partir de funções, condicionantes ou loops.
- **Var:** define uma variável global ou local (em uma função), independentemente do escopo do bloco. As variáveis declaradas com **var** estarão disponíveis para todos os elementos de código.

As variáveis têm dois escopos possíveis:

- **Global:** Podemos acessá-los de qualquer lugar em nosso código.
- **Local:** Podemos acessá-los apenas dentro do escopo em que foram definidos. Normalmente funciona. Eles serão acessíveis a partir da própria função ou de funções aninhadas em níveis mais altos do que a função anterior. Ou seja, variáveis locais podem ser vistas de dentro para fora, mas nunca de fora para dentro da função.

```
var num = 5;
document.write(num);    //num vale 5

if (true) {
    var num = 10;
    document.write(num); //num
vale 10
}

document.write(num);    //num vale 10
```

Com **let**:

```
let num = 5;
document.write(num);    //num vale 5

if (true) {
    let num = 10;
    document.write(num); //num
vale 10
}

document.write(num);    //num vale 5
```

let o que realmente faz é criar uma nova variável local no nível do bloco, neste caso dentro do **if**. Fora dela não tem validade.

```
var puntuación;
puntuación=500;

var puntuación=500, record=1000,
jugador= "Angel";
```

Se declararmos uma variável entre aspas, **javascript** entende que se trata de uma **string**, mesmo que seja um número:

```
var num1=5;    //Aquí num1 es un
número (int).
var num2= "5"; //Aquí num2 es un
string.
```

Para modificar uma variável já declarada, não colocamos **var**, apenas o nome e o valor:

```
var num=5; //Declaracion y
asignación
num=10;    //Reasignación
```

O comando **typeof()** nos diz o tipo de uma variável.

Podemos colocar **typeof num;** ou **typeof(num);**

Casting do variável

Se precisarmos converter um `string` para um `number`, temos duas opções:

1- Usando a função `Number()`:

```
var num1=45;
Resultado= num1+Number(num2);
```

2-Usando `parseInt`:

```
alert(parseInt(num2)+23);
```

A função `parseFloat()` também está disponível para converter `strings` em números decimais.

Constantes

Por definição, uma variável é um espaço na memória do computador onde um valor será armazenado, que pode mudar durante a execução do programa. No caso de uma constante, ela é definida como um espaço na memória do computador onde um valor será armazenado, que **NÃO** pode ser alterado durante a execução do programa. Ou seja, se declaramos uma constante e armazenamos um valor dentro dela, este valor não pode ser modificado.

Sintaxe:

```
const nombre="Antonio";
```

Exemplos de operações com variáveis

```
//Podemos trabajar con números, strings y
booleans
// Declaración de variables:
// No puede empezar por un número
// Esto de error:   var 2_cliente;
// Evitar tildes y Ñ

//Variable correctamente declarada
let _numero;

// Inicialización de la variable
_numero = 63;

//Variable declara e inicializada
var numero4 = 647;

// Sepueden declara e inicializar varias
variables a la vez.
var a=3, b=4, c=5;

//Sería lo mismo que poner:
/*
var a=3;
let b=4;
var c=5;
*/

// Javascript es Case sensitive.
Distingue entre mayúsculas/minúsculas
var numero = 10;
var Numero = 20;

console.log("La variable 'numero' vale:"
+ numero);
console.log("La variable 'Numero' vale:"
+ Numero);

//Con variable numéricas podemos operar
matemáticamente

var sumando1 = 35;
```

```
var sumando2 = 45;

var resultado = sumando1+sumando2;

console.log("El resultado de sumar " +
sumando1 + " + "
          + sumando2 + " es " +
resultado);

// Operador modulo o resto de la
división
var resto = 46 % 5;
console.log("El resto de 46/5 es " +
resto);

//Operador incremento
var numeroInicial = 10;
let numeroIncrementado =
++numeroInicial;

//Sería como poner que
numeroIncrementado = numeroInicial +1;
console.log("El operador incremento
sobre numeroInicial daría " +
numeroIncrementado);

//IMPORTANTE: No es lo mismo ++variable
que variable++
var numero5 = 5;

document.write("El número antes del
incremento vale " + numero5++);
document.write("<br>");
document.write("El número después del
incremento vale " + numero5);

document.write("<br>");
document.write("<br>");

// Ahora al revés
let numero6 = 5;

document.write("El número antes del
incremento vale " + ++numero6);
document.write("<br>");
document.write("El número después del
incremento vale " + numero6);
```

```
//Constantes. Como las variables pero no
se puede cambiar su valor
// Se declaran en mayúsculas
const MICONSTANTE = 4765;
//MICONSTANTE = 345; Esto daría error

// Hay que declararlas e inicializarlas
obligatoriamente.
//const MICONSTANTE2; Esto daría error
```

Strings

As strings são definidas como variáveis cujo conteúdo são caracteres alfanuméricos, ou seja, texto. Eles estão sempre entre aspas.

```
//Declaración de strings
//-----
//-----
let nombre = "Mi nombre es Angel";
let nombre2 = 'Mi nombre es Angel';

/* Aunque declaremos números, si lo
hacemos entre comillas,
nos lo tratará como strings */
let edad2 = "43";
console.log(edad2+20);

//Las comillas anidadas siempre alternan,
let nombre3 = 'Mi nombre es "Angel" ';
let nombre4 = "Mi nombre es 'Angel' ";

/* Cuando declaremos numeros con los que
no vamos a operar
matemáticamente lo haremos como strings
*/
let telefono = "666666666";

//Los strings no se suman, se concatenan:
let nombre5 = "Angel";
let espacio = " ";
let apellido = "García";
let nombreCompleto =
nombre+espacio+apellido;

console.log(nombreCompleto);

//Otra forma de hacerlo
console.log(nombre5 + " " +apellido);

console.log(nombre5 + " " +edad2);
```

Funções de strings

JavaScript tem funções predefinidas que nos facilitam o trabalho com **strings**. As mais comumente utilizadas são:

Parseint(): Converte um **STRING** a um número. Outra opção é usar o **Number()**.

toString: Converte um número para **string**:

```
var num=3;
var num2=num.toString();
```

toUpperCase: Converter um string em maiúsculas.

toLowerCase: Converte um String para minúscula.

Length: Calcula a extensão de um texto. Se usado com **arrays** me diz o número de elementos no **array**.

Concat: Para concatenar o texto. Como o +

```
var texto=texto1.concat(" "+texto2);
```

indexOf(): Para procurar um texto dentro de outro texto. Ele me diz se há um texto dentro de outro texto e em que posição ele se encontra. Procura a primeira partida.

Temos também um **LastIndexOf()** que traz à tona a última coincidência.

```
var texto="Esto es el texto";
var busqueda= texto.indexOf("texto");
document.write(busqueda);
```

Neste caso, eu imprimiria **11**.

O método **search()** funciona da mesma maneira:

```
var busca = texto.search("texto");
```

Se não encontrar o texto, ele retornará **-1**.

Match(): Funciona como **indexOf** o **search**, mas, neste caso, ele devolve um array com os resultados. Em princípio, ele retorna a primeira partida. Se eu quiser obter todos os fósforos, tenho que usar a expressão regular:

```
var busca = texto.match(/texto/gi);
```

Substr(): Leve-me do personagem 14, 5 caracteres para cima:

```
var busca = texto.substr(14,5);
```

CharAt(): Para selecionar uma determinada carta de um **string**.

```
var busca = texto.CharAt(44);
```

Isso me mostraria a carta na posição 44.

StartWith(): Para procurar texto no início do **string**. Retorna **true** ou **false**, dependendo se você o encontrou ou não.

Ou seja, se o texto começa com os caracteres com os quais dizemos para começar.

```
var texto = "Hola mundo";  
var busca = texto.StartsWith("Hola");
```

Neste caso, daria **true**.

```
var busca =  
texto.StartsWith("mundo");
```

Neste caso, daria **false**.

endsWith(): Como o anterior, mas olhando se o texto termina com os caracteres que dizemos para ele.

includes(): Procura por uma palavra em um texto. É case sensitive.

replace(): Permite-nos substituir um texto por outro.

```
var busca = texto.replace("Hola",  
"Adios");
```

Procura por **"Hola"** e o substitui por **"Adios"**.

slice(): Eu separo um **string** do personagem que dizemos que seja.

```
var texto = "Hola mundo";  
var busca = texto.slice(5);
```

O **string** é cortado a partir do personagem 5, ou seja, ele me deixa apenas com **"mundo"**.

Também posso lhe dizer a posição inicial e final:

```
var busqueda= texto.slice(5,7);
```

`split()`; Eu tenho o `string` em um `array`.

```
var busqueda= texto.split();
```

Podemos dar-lhe um separador e ele insere as palavras entre os espaços, por exemplo:

```
var busqueda= texto.split(" ");
```

`trim()`: Ele remove os espaços na frente e atrás da `string`.

Modelos de strings

Podemos substituir valores dentro de um `string` sem concatená-los, usando um modelo. Para isso, utilizo aspas invertidas e interpolação variável com `$()`.

Modo normal:

```
var nombre=prompt("Introduce tu nombre: ");
var apellidos=prompt("Introduce tus apellidos: ");
var texto="Tu nombre es "+nombre+" y tus apellidos"+apellidos;
```

Com modelo:

```
var texto= `
<h3>Tu nombre es ${nombre}</h3>
<h3> y tus apellidos
${apellidos}</h3>
`;
```

Operadores

As variáveis por si só são de pouca utilidade. Até agora, só vimos como criar variáveis de diferentes tipos e como exibir seu valor usando a função de `alert()`. Para fazer programas realmente úteis, são necessários outros tipos de ferramentas.

Os operadores permitem manipular o valor das variáveis, realizar operações matemáticas sobre seus valores e comparar diferentes variáveis. Desta forma, os operadores permitem que os programas realizem cálculos complexos e tomem decisões lógicas com base em comparações e outros tipos de condições.

Operadores de Atribuição

O operador de atribuição é o operador mais utilizado e mais simples. Este operador é utilizado para armazenar um valor específico em uma variável. O símbolo utilizado é `=` (não confundir com o `==` operador, que será discutido mais tarde):

```
var numero1 = 3;
```

À esquerda do operador, o nome de uma variável deve ser sempre indicado. À direita do operador, podem ser indicadas variáveis, valores, condições lógicas, etc.:

```
var numero1 = 3;
var numero2 = 4;

/* Error, la asignación siempre se
realiza a una variable,
por lo que en la izquierda no se
puede indicar un número */
5 = numero1;

// Ahora, la variable numero1 vale 5
numero1 = 5;

// Ahora, la variable numero1 vale 4
numero1 = numero2;
```

Operadores de Incremento e Decremento

Estes dois operadores são válidos somente para variáveis numéricas e são usados para incrementar ou diminuir o valor de uma variável por uma unidade.

```
var numero = 5;
++numero;
alert(numero); // numero = 6
```

O operador de incremento é indicado pelo prefixo `++` no nome da variável.

O resultado é que o valor dessa variável é aumentado em uma unidade.

Portanto, o exemplo acima é equivalente a:

```
var numero = 5;
numero = numero + 1;
alert(numero); // numero = 6
```

Equivalentemente, o operador de decremento (indicado como um prefixo `--` no nome da variável) é usado para diminuir o valor da variável:

```
var numero = 5;
--numero;
alert(numero); // numero = 4
```

O exemplo acima é equivalente a:

```
var numero = 5;
numero = numero - 1;
alert(numero); // numero = 4
```

Os operadores de incremento e decremento não só podem ser especificados como um prefixo para o nome da variável, mas também podem ser usados como um sufixo. Neste caso, seu comportamento é semelhante, mas muito diferente. No exemplo a seguir:

```
var numero = 5;
numero++;
alert(numero); // numero = 6
```

O resultado da execução do script acima é o mesmo que quando se usa o operador `++numero`, por isso pode parecer que é equivalente a indicar o operador `++` antes ou depois do identificador da variável. Entretanto, o exemplo a seguir mostra suas diferenças:

```
var numero1 = 5;
var numero2 = 2;
numero3 = numero1++ + numero2;
// numero3 = 7, numero1 = 6

var numero1 = 5;
var numero2 = 2;
numero3 = ++numero1 + numero2;
// numero3 = 8, numero1 = 6
```

Se o operador `++` é especificado como um prefixo para o identificador da variável, seu valor é incrementado **antes** que qualquer outra operação seja realizada. Se o operador `++` é dado como sufixo ao identificador da variável, seu valor é incrementado **após** a execução da declaração em que aparece.

Portanto, na instrução `numero3 = numero1++ + numero2;`, o valor de `numero1` é incrementada após a realização da operação (primeiro a soma e `numero3` é 7, então aumente o valor de `numero1` e vale 6). No entanto, na instrução `numero3 = ++numero1 + numero2;`, antes de tudo, o valor de `numero1` e então a adição é realizada (primeiro o `numero1` é 6, então a soma é realizada e `numero3` é 8).

Isto é, embora as duas expressões sejam muito semelhantes, o operador de adição e incremento não é o mesmo (`numero++`) que aumenta e soma (`++numero`).

Operadores Lógicos

Os operadores lógicos são essenciais para aplicações complexas, pois são usados para tomar decisões sobre quais instruções o programa deve executar com base em certas condições.

O resultado de qualquer operação utilizando operadores lógicos é sempre um valor lógico ou booleano.

Negação

Um dos operadores lógicos mais comumente utilizados é o operador de negação. É utilizado para obter o valor oposto ao valor da variável:

```
var visible = true;
alert(!visible); // Muestra "false"
y no "true"
```

A negação lógica é obtida através da prefixação do símbolo `!` ao identificador da variável. A operação deste operador está resumida na seguinte tabela:

variable	!variable
true	false
false	true

Se a variável original é uma variável booleana, é muito fácil obter sua negação. Entretanto, o que acontece quando a variável é um número ou uma cadeia de texto?

Para obter a negação neste tipo de variável, ela é primeiramente convertida para um valor booleano:

- Se a variável contém um número, ela se torna **false** se vale 0 e em **true** para qualquer outro número (positivo ou negativo, decimal ou inteiro).
- Se a variável contém uma cadeia de texto, ela é transformada em **false** se o fio estiver vazio (""), e em **true** em qualquer outro caso.

```
var cantidad = 0;
vacio = !cantidad; // vacio = true

cantidad = 2;
vacio = !cantidad; // vacio = false

var mensaje = "";
mensajeVacio = !mensaje; //
mensajeVacio = true

mensaje = "Bienvenido";
mensajeVacio = !mensaje; //
mensajeVacio = false
```

AND

A operação lógica **AND** obtém seu resultado através da combinação de dois valores booleanos. O operador é indicado pelo símbolo **&&** e seu resultado é apenas **true** se os dois operandos são **true**:

variable1	variable2	variable1 && variable2
true	true	true
true	false	false
false	true	false
false	false	false

```
var valor1 = true;
var valor2 = false;
resultado = valor1 && valor2; //
resultado = false

valor1 = true;
valor2 = true;
resultado = valor1 && valor2; //
resultado = true
```

OR

A operação lógica **OR** também combina dois valores booleanos. O operador é indicado pelo símbolo **||** e seu resultado é verdadeiro se um dos dois operandos forem **true**:

variable1	variable2	variable1 variable2
true	true	true
true	false	true
false	true	true
false	false	false

```
var valor1 = true;
var valor2 = false;
resultado = valor1 || valor2; //
resultado = true

valor1 = false;
valor2 = false;
resultado = valor1 || valor2; //
resultado = false
```

Operadores Matemáticos

O JavaScript permite manipulações matemáticas sobre o valor das variáveis numéricas. Os operadores definidos são: adição (+), subtração (-), multiplicação (*) e divisão (/). Exemplo:

```
var numero1 = 10;
var numero2 = 5;

resultado = numero1 / numero2; //
resultado = 2
resultado = 3 + numero1;      //
resultado = 13
resultado = numero2 - 4;      //
resultado = 1
resultado = numero1 * numero2; //
resultado = 50
```

Além dos quatro operadores básicos, o Javascript define outro operador matemático que não é fácil de entender ao estudá-lo pela primeira vez, mas que é muito útil em algumas ocasiões.

Este é o operador do "módulo" (ou o restante da divisão), que calcula o restante da divisão inteira de dois números. Se você dividir por exemplo 10 e 5, a divisão é exata e dá um resultado de 2. O restante dessa divisão é 0, então o módulo de 10 e 5 é igual a 0.

Entretanto, se você dividir 9 e 5, a divisão não é exata, o resultado é 1 e o restante é 4, então o módulo de 9 e 5 é igual a 4.

O operador do módulo em JavaScript é indicado pelo símbolo %, que não deve ser confundido com o cálculo da porcentagem:

```
var numero1 = 10;
var numero2 = 5;
resultado = numero1 % numero2; //
resultado = 0

numero1 = 9;
numero2 = 5;
resultado = numero1 % numero2; //
resultado = 4
```

Os operadores matemáticos também podem ser combinados com o operador de atribuição para abreviar sua notação.:

```
var numero1 = 5;
numero1 += 3; // numero1 = numero1 +
3 = 8
numero1 -= 1; // numero1 = numero1 -
1 = 4
numero1 *= 2; // numero1 = numero1
* 2 = 10
numero1 /= 5; // numero1 = numero1
/ 5 = 1
numero1 %= 4; // numero1 = numero1
% 4 = 1
```

Operadores Relacionais

Os operadores relacionais definidos pelo JavaScript são idênticos aos definidos pela matemática: maiores que (>), menores que (<), maiores ou iguais a (>=), menores ou iguais a (<=), iguais a (==) e não iguais a (!=).

Os operadores que relacionam as variáveis são essenciais para realizar qualquer aplicação complexa, como veremos no próximo capítulo de programação avançada. O resultado de todos esses operadores é sempre um valor booleano:

```
var numero1 = 3;
var numero2 = 5;
resultado = numero1 > numero2; //
resultado = false
resultado = numero1 < numero2; //
resultado = true

numero1 = 5;
numero2 = 5;
resultado = numero1 >= numero2; //
resultado = true
resultado = numero1 <= numero2; //
resultado = true
resultado = numero1 == numero2; //
resultado = true
resultado = numero1 != numero2; //
resultado = false
```

Um cuidado especial deve ser tomado com o operador de igualdade (`==`), pois ele é a fonte da maioria dos erros de programação, mesmo para os usuários que já têm alguma experiência no desenvolvimento de scripts. O `==` operador é usado para comparar o valor de duas variáveis, portanto é muito diferente do `=` operador, que é usado para atribuir um valor a uma variável:

```
// El operador "=" asigna valores
var numero1 = 5;
resultado = numero1 = 3; // numero1
= 3 y resultado = 3

// El operador "==" compara variables
var numero1 = 5;
resultado = numero1 == 3; // numero1
= 5 y resultado = false
```

Os operadores relacionais também podem ser usados com variáveis de corda de texto:

```
var texto1 = "hola";
var texto2 = "hola";
var texto3 = "adios";

resultado = texto1 == texto3; //
resultado = false
resultado = texto1 != texto2; //
resultado = false
resultado = texto3 >= texto2; //
resultado = false
```

Ao utilizar cadeias de texto, os operadores "maior que" (`>`) e "menor que" (`<`) seguem um raciocínio não-intuitivo: letra por letra é comparada a partir da esquerda até que uma diferença seja encontrada entre as duas cadeias de texto. Para determinar se uma letra é maior ou menor que outra, letras maiúsculas são consideradas menos que letras minúsculas e as primeiras letras do alfabeto são menores que as últimas letras (a é menor que b, b é menor que c, A é menor que a, etc.)

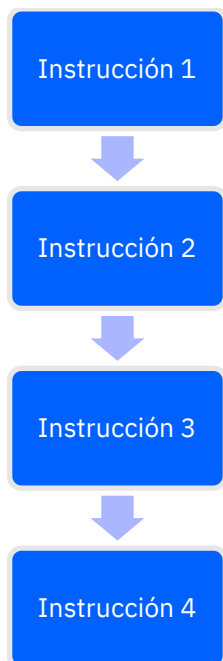
Condições e loops em JavaScript

Em linguagens de programação, as instruções que permitem controlar decisões e loops de execução são chamadas de "Estruturas de Controle". Uma estrutura de controle dirige o fluxo da execução através de uma sequência de instruções, baseada em decisões simples e outros fatores.

Uma parte muito importante de uma estrutura de controle é a "condição". Cada condição é uma expressão que se avalia como true ou false.

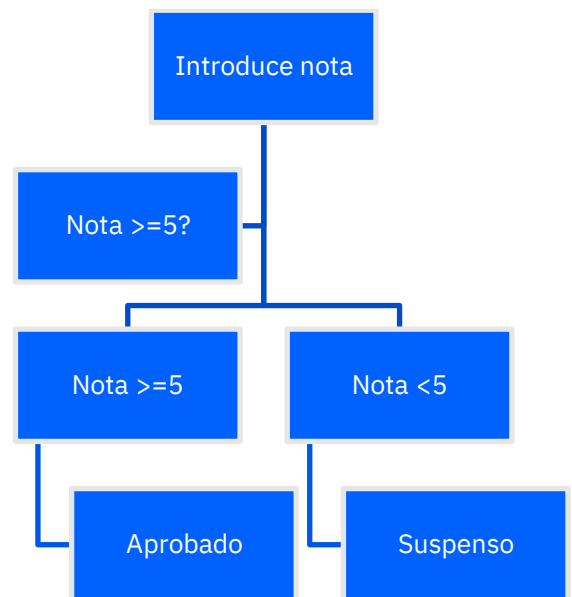
O JavaScript oferece um total de quatro instruções para processar o código de acordo com as condições determinadas pelo programador: `if`, `switch`, `for` e `while`.

Até agora, temos feito a execução de roteiros lineares. Ou seja, executar instruções uma após a outra. Até que uma instrução fosse concluída, a próxima não foi executada:

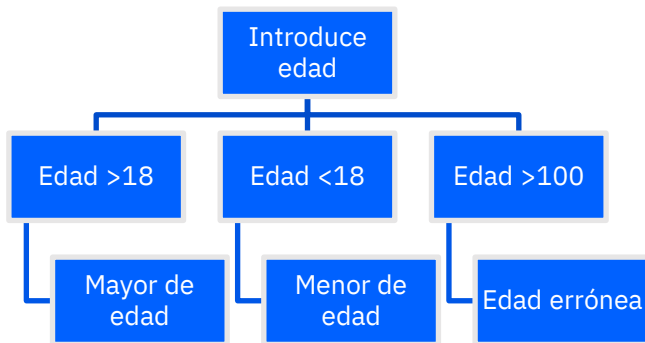


Com os condicionais, modificaremos o fluxo de execução como acharmos conveniente, dependendo se uma condição é cumprida.

O diagrama a seguir representa o fluxo de execução de um simples condicional. A `nota` variável é avaliada e se for igual ou maior que 5, uma série de instruções será executada, caso contrário outras serão executadas, mas nunca as duas, ou uma ou a outra.



Dependendo das necessidades de nosso código, o condicional pode ser tornado tão complicado quanto necessário:



Condicionais

A tomada de decisão na programação é semelhante à tomada de decisão na vida real. Na programação também enfrentamos algumas situações em que queremos que um determinado bloco de código seja executado quando alguma condição é atendida.

As linguagens de programação utilizam declarações condicionais para controlar o fluxo da execução do programa com base em certas condições. Estes são usados para fazer o fluxo de execução avançar e ramificar-se com base em mudanças no estado de um programa.

Comparadores

A fim de poder tomar decisões, é preciso fazer comparações.

Estas comparações são realizadas utilizando operadores de comparação e operadores lógicos:

OPERADOR DE COMPARACIÓN	DESCRIPCIÓN	EJEMPLO
==	Igual que... Comprueba valor.	x==y
===	Estrictamente igual. Comprueba valor y tipo.	x===y
!=	Diferente. Es igual que poner <>	x!=y x<>y
<	Menor que...	x<y
>	Mayor que...	x>y
<=	Menor o igual que...	x<=y
>=	Mayor o igual que...	x>=y

OPERADOR LÓGICO	DESCRIPCIÓN	EJEMPLO
&&	Y lógico	a>b && b<c
	O lógico	a=b a=c
!	NO lógico	x!=y

Javascript funciona com dois tipos de condicionadores, **IF...ELSE** e **SWITCH...CASE**. A seguir, veremos o uso de ambos.

IF...ELSE

Esta declaração é a condição mais simples para a tomada de decisões. É usado para decidir se uma determinada declaração ou bloco de instruções será executado ou não, ou seja, se uma determinada condição for verdadeira, então um bloco de instruções é executado, caso contrário, outras instruções em código serão executadas.

A sintaxe pode assumir várias formas:

Opção 1. Uma única condição a ser verificada:

```
If( Condição a ser cumprida){
Instruções a serem executadas caso a condição seja
cumprida
}else{
Instruções a serem executadas caso a condição não
seja cumprida
}
```

Por exemplo, um carro custa 30.000€. Pergunte ao usuário quanto dinheiro ele tem. Compare os dois valores e emita uma mensagem de alerta dizendo se ele pode ou não comprar o carro.

```
var precio=30000;
var dinero=prompt("Introduce cuanto
dinero tienes: ");

if(dinero>precio){
    alert("Te puedes comprar el
coche");
}else{
    alert("Te vas en autobus");
}
```

Neste exemplo, avaliamos como condição que o dinheiro que você tem deve ser igual ou maior do que o preço do carro para poder comprá-lo.

Opção 2. Mais de uma condição a ser verificada

```
If( Condição a ser cumprida){
Instruções a serem executadas caso a condição seja
cumprida
}else IF( Segunda condição a ser cumprida){
Instruções a serem executadas caso a segunda
condição seja cumprida
}else{
Instruções a serem executadas caso nenhuma
condição seja cumprida
}
```

Você pode colocar quantos `else...if` quiser.

O `else` final não é obrigatório, mas é útil executar instruções no caso de nenhuma das condições anteriores ser atendida.

Vamos acrescentar uma segunda condição ao exemplo anterior. Além de ter o dinheiro, para comprar o carro você deve ter idade legal:

```
var precio = 30000;
var dinero = prompt("Introduce cuánto
dinero tienes: ");
var edad = prompt("Introduce tu edad:
");
if ((precio < dinero) && (edad >=
18)) {
    alert("Te puedes comprar el
coche");
} else if ((precio < dinero) && (edad
< 18)) {
    alert("Tienes el dinero pero no
la edad");
}
else if ((precio > dinero) && (edad
>= 18)) {
    alert("Tienes la edad pero no el
dinero");
}
else if ((precio > dinero) && (edad <
18)) {
    alert("Ni dinero ni edad");
}
```

Se o usuário inserir um valor na janela que não seja um número, teremos um problema, pois devemos lembrar que tudo o que o usuário inserir por prompt é considerado uma **string** e, para verificar se **edad>=18**, precisamos que a **edad** seja um número.

Para descobrir se um valor é um valor numérico, usamos a função **isNaN()**.

```
var num1=prompt("Introduce numero");
var num2=prompt("Introduce numero2");
if(!isNaN(num1)&&!isNaN(num2)){
  alert(parseInt(num1)+parseInt(num2));
}else{
  alert("No has introducido números");
}
```

SWITCH...CASE

Outra opção fornecida pelo Javascript para o uso de condicionadores é **switch...case**.

Qualquer avaliação que possa ser feita com **switch...case** também pode ser feito com **IF... else**.

A sintaxe do switch...case condicional é:

```
switch(Variable_a_comprobar) {
```

```
  case "opcion1" :
```

```
    instrucciones;
```

```
    break;
```

```
  case "opcion2" :
```

```
    instrucciones;
```

```
    break;
```

```
  .....

```

```
  case "opcionN" :
```

```
    instrucciones;
```

```
    break;
```

default :

```
  instrucciones;}
```

Vamos analisar mais de perto a sintaxe deste código, que inclui as palavras-

chave: **switch**, **case**, **break** y **default**:

- Ao lado da palavra-chave '**switch**' colocaremos entre parênteses o nome da variável cujo valor desejamos verificar. Dependendo do valor, uma série de instruções deve ser executada.
- Cada valor contra o qual vamos comparar é incluído após a palavra-chave '**case**' separada por dois pontos (:) e o valor entre aspas.
- Por exemplo, se nosso programa pedisse um número de 1 a 10, e dependendo da escolha do usuário, o programa daria um resultado específico, escreveríamos um '**case**' para cada número: 1, 2, 10.
- Incluímos um '**break**' em cada cláusula de '**case**' para que o programa não pare aí, mas deixe esse ponto e continue após o final da instrução **switch** com as seguintes instruções contidas no programa.
- A última cláusula '**default**' realiza a mesma tarefa que a última cláusula '**else**' que colocamos depois de uma série de '**else if**'. Ou seja, se nenhuma das condições foi cumprida e queremos que algo específico seja executado, colocamos isso nessa cláusula '**default**'.

Em resumo, a instrução '**switch**' atua como um switch em conjunto com os diferentes '**case**' (a tradução da palavra '**switch**' é exatamente isso: interruptor). Imagine uma linha de trem que em um ponto se divide em vários trilhos que vão em diferentes direções. Assim como um mecanismo foi acionado para que o trem continuasse ao longo de um desses trilhos, dependendo de seu destino, a **switch...case** atinge esse efeito, fazendo com que o programa prossiga ao longo do **case** que atende à condição, e portanto, o programa executa as instruções que foram definidas ali.

A partir de uma variável de `edad`, crie um programa que imprima mensagens diferentes, dependendo do valor de `edad`.

```
var edad = 18;
var imprime = "";

switch (edad) {
  case 18:
    imprime = "Acabas de cumplir la mayoría de edad";
    break;
  case 25:
    imprime = "Eres un adulto";
    break;
  case 50:
    imprime = "Eres maduro";
    break;
  default:
    imprime = "Otra edad no contemplada"

    break;
  document.write(imprime);
}
```

É importante observar que `switch...case` avalia as condições concretas de igualdade. Ou seja, em nosso exemplo, verificamos se `edad = 18` ou se `edad = 25` ou `edad = 50`.

Mas não seríamos capazes de fazer verificações do tipo `edad >= 18`, por exemplo. Para este tipo de verificações, é necessário utilizar um `if...else`.

Loops

Los bucles, son una estructura de programación que nos permite ejecutar instrucciones de forma repetitiva dentro de un bloque de programación.

Os loops são uma estrutura de programação que nos permite executar instruções de forma repetitiva dentro de um bloco de programação.

Além disso, os loops são executados através de uma condição.

Existem dois tipos de loops:

- **Loops determinados:** Aqueles em que se sabe antecipadamente quantas vezes um código vai ser repetido. Um loop deste tipo é um loop "For".
- **Loops indeterminados:** Aqueles em que não se sabe a priori quantas vezes o código será repetido. Dependerá do cumprimento ou não de uma condição. Dois loops deste tipo são "While" e "Do while".

O loop dado para FOR

Um tipo de loop cuja execução dura um determinado número de vezes ou até que sua condição seja avaliada como falsa (false).

Sua sintaxe é:

```
for(inicio ; condición ;
incremento/decremento){

  código a repetir}
```

A estrutura é descrita da seguinte forma:

- **inicio:** Esta expressão indica a condição para o início do ciclo. Geralmente é a declaração de uma variável numérica. Uma variável é geralmente criada com o nome `i`, `j`, `k`... etc.
- **condición:** Esta expressão é a condição que deve ser mantida a fim de continuar a execução do ciclo.
- **incremento/decremento:** Esta expressão muda o valor da variável numérica indicada no **inicio**.
- **código a repetir :** Corpo em loop. O conjunto de instruções que são executadas durante cada iteração do laço (em cada laço).


```
for(var i=0;i<10;i++){
    document.write("Hola"+"<br>");
}

document.write("Ejecución
terminada.");
```

Neste loop **for** criamos uma variável local chamada **i** que inicializamos a zero. A condição é que o loop se repita por tanto tempo quanto **i<10**. A cada volta de loop **i** será incrementado por uma unidade (de modo que na décima volta a condição não será mais cumprida). E a cada volta a palavra **hola** deve ser impressa.

Após o décimo loop, a condição não será mais cumprida, pois **i** me tornarei **10** e nosso programa sairá do loop e continuará com a próxima linha de execução, neste caso, imprimir

Ejecución terminada.

```
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Hola
Ejecución terminada.
```

Outro exemplo. Crie um loop **for** que executa 10 vezes e imprime na tela o valor da variável local em cada uma das curvas.

```
for (i=0 ; i<10 ; i++){
    document.write("En esta vuelta de
bucle I vale " + i);
    document.write("<br>");
}

document.write("Ejecución terminada.");
```

```
En esta vuelta de bucle I vale 0
En esta vuelta de bucle I vale 1
En esta vuelta de bucle I vale 2
En esta vuelta de bucle I vale 3
En esta vuelta de bucle I vale 4
En esta vuelta de bucle I vale 5
En esta vuelta de bucle I vale 6
En esta vuelta de bucle I vale 7
En esta vuelta de bucle I vale 8
En esta vuelta de bucle I vale 9
Ejecución terminada.
```

O principal uso do **for** loop é fazer loop através de **arrays**, com a condição de que o loop se repita tantas vezes quantas tivermos elementos em nosso **array**:

```
var meses = ["Enero", "Febrero",
"Marzo", "Abril", "Mayo", "Junio",
"Julio", "Agosto", "Septiembre",
"Octubre", "Noviembre", "Diciembre"];
for (var i = 0; i < meses.length; i++)
{
    document.write(meses[i] + "<br/>");
}
```

Enero
 Febrero
 Marzo
 Abril
 Mayo
 Junio
 Julio
 Agosto
 Septiembre
 Octubre
 Noviembre
 Diciembre

Vejamos um exemplo de [while](#).

Repita o exemplo dos meses do ano que vimos no [for](#), mas implementando-o com um loop de [while](#):

```
var contador = 0;
var meses = ["Enero", "Febrero",
"Marzo", "Abril", "Mayo", "Junio",
"Julio", "Agosto", "Septiembre",
"Octubre", "Noviembre", "Diciembre"];
while (contador < meses.length) {
  document.write(meses[contador] +
"<br>");
  contador++;
}
```

WHILE. DO...WHILE

É semelhante ao ciclo [for](#) explicado acima. Ela é executada até que a condição seja falsa ([false](#)). A diferença é que a avaliação é feita no final de cada iteração.

Se usarmos [while](#) podemos descobrir que a condição não é atendida mesmo na primeira verificação, então o código nunca seria executado.

No caso de [do...while](#) nós nos certificamos de que o código seja executado pelo menos uma vez.

Sintaxe:

While(Condición a cumprir){

Código a ejecutar mientras se cumpla la condición}

do{ Código a ejecutar mientras se cumpla la condición }

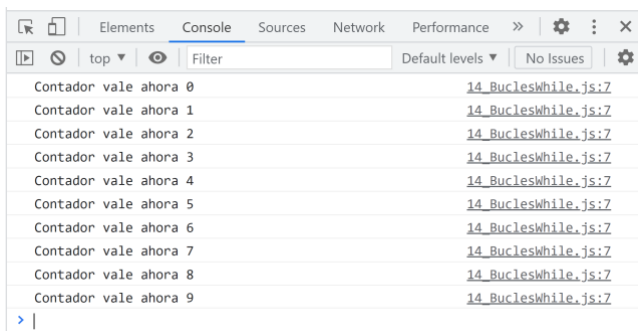
While(Condición a cumplir)

Enero
 Febrero
 Marzo
 Abril
 Mayo
 Junio
 Julio
 Agosto
 Septiembre
 Octubre
 Noviembre
 Diciembre

Otro ejemplo sería la creación de un programa que pide al usuario el número de ejecuciones e imprime en el console cuánto vale una contra-variável, que es incrementada en cada ciclo del loop:

```
let contador = 0;
let ciclos = Number(prompt("Introduce
número de ejecuciones"));

while(contador < ciclos){
  console.log("Contador vale ahora
" + contador);
  contador++;
}
```



Message	File
Contador vale ahora 0	14_BuclesWhile.js:7
Contador vale ahora 1	14_BuclesWhile.js:7
Contador vale ahora 2	14_BuclesWhile.js:7
Contador vale ahora 3	14_BuclesWhile.js:7
Contador vale ahora 4	14_BuclesWhile.js:7
Contador vale ahora 5	14_BuclesWhile.js:7
Contador vale ahora 6	14_BuclesWhile.js:7
Contador vale ahora 7	14_BuclesWhile.js:7
Contador vale ahora 8	14_BuclesWhile.js:7
Contador vale ahora 9	14_BuclesWhile.js:7

Um exemplo de `do...while` é criar um número aleatório entre 0-100 e pedir ao usuário que tente adivinhá-lo.

Cada tentativa incrementará um contador.

```
//Creamos un número aleatorio entre 0
y 1.
// Despues lo multiplicamos por 100
para que esté entre 0-100
//Lo redondeamos para que sea entero
var numero =
parseInt(Math.random()*100);

var numero_introducido;
var contador = 0;

while (numero != numero_introducido)
{
  numero_introducido =
Number(prompt("Introduce número: "));
  contador++;

  if (numero_introducido > numero) {
    alert("Demasiado alto");
  }

  if (numero_introducido < numero) {
    alert("Demasiado bajo");
  }
}

alert("CORRECTO!!!, el número era el
" + numero + ". Has acertado en " +
contador + " intentos.");
```

Break

Permite-nos sair de um loop de execução quando uma condição é cumprida.

No exemplo a seguir, temos um loop que imprimirá os anos de 2000 a 2010. Mas introduzimos uma pausa quando o **año = 2005**:

```
var year = 2000;
while (year < 2010) {
  document.write(year + "<br>");
  if (year == 2005) {
    break;
  }
  year++;
}
```

2000
2001
2002
2003
2004
2005

Neste caso, quando o ano é igual a 2005, ele sai do loop e não executa o resto das instruções do laço.

Arrays

Até agora, vimos como colocamos um único valor dentro de uma variável, seja ela numérica, de texto ou booleana.

Os arrays nos permitem trabalhar com conjuntos de valores e armazená-los em um único endereço de memória.

Em **Javascript**, os valores de **array** NÃO têm que ser do mesmo tipo.

- var Articulos=["zapatilla","camiseta","pantalon","calcetines"];
- var Articulos=new array ["zapatilla","camiseta","pantalon","calcetines"];

Artigos: sapatos, camiseta, calça e meias.

Em **Javascript**, um **array** é declarada das seguintes formas:

```
var
Articulos=["zapatilla","camiseta","pantalón","calcetines"];
```

Ou:

```
var Articulos =new
array("zapatilla","camiseta","pantalón","calcetines");
```

Em ambos os exemplos, criamos um **array** (ou seja, uma coleção de elementos) que denominamos **Artículos** e dentro nós armazenamos os valores: **zapatilla**, **camiseta**, **pantalón** y **calcetines**.

Para trabalhar com os dados contidos em um array, devemos nos referir a eles de acordo com sua posição no array.

Este é um exemplo de acesso a um array:

```
alert(Articulos[1]);           Resultado
: camiseta
```

A posição dos elementos começa a contar a partir de zero, ou seja, em nosso array **Artículos**, o elemento na posição zero é **zapatilla**, na posição 1, temos

camiseta, na posição 2, temos **pantalón** calças e na posição 3 temos **calcetines**.

Como em nosso exemplo pedimos que fosse exibida uma mensagem de alerta com o elemento na posição 1, receberemos **camiseta**.

Se quisermos mudar ou inserir um elemento em um array, o faremos da mesma forma, referindo-nos a sua posição dentro do array:

No exemplo a seguir, vamos criar um array de três elementos e vamos pedir ao usuário que insira um valor por meio do comando **prompt**. Este valor deve ser armazenado na posição 4.

```
var
array1=["objeto1","objeto2","objeto3"];
array1[3]="objeto4";
array1[4]=prompt("Introduce
objeto4");
alert(array1[4]);
```

Funções da arrays

O Javascript nos fornece uma série de funções predefinidas prontas para facilitar o trabalho com arrays.:

- **Length**: Retorna o comprimento do array.
- **push**: Adiciona elementos ao final do array.
- **unshift**: Adiciona elementos ao início do array.
- **pop**: Remove elementos no final do array.
- **shift**: Remove elementos no início do array.

Aqui está um exemplo destas funções:

```
<html>

<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var articulos = ["balon", "botas",
"camiseta", "pantalon"];
  </script>
</head>

<body>
  <script>
    document.write("<p>El primer articulo
es <strong > "+articulos[0]+"</strong
></p > ");
    document.write("<p>La longitud del
array es " + articulos.length + "</p>");
    document.write("<p>El último articulo
es <strong>" + articulos[articulos.length
- 1] + "</strong></p>");
    articulos.unshift("pelota");
    articulos.push("canasta");
    document.write("<p>Añadimos el
artículo <strong>" + articulos[0] +
"</strong></p>");
    document.write("<p>El primer articulo
ahora es <strong>" + articulos[0] +
"</strong></p>");
    document.write("<p>El último articulo
ahora es <strong>" +
articulos[articulos.length - 1] +
"</strong></p>");
    document.write("<p>La longitud del
array ahora es " + articulos.length +
"</p>");
    document.write(articulos);
    articulos.push(prompt("introduce
nuevo articulo"));
    document.write("<br>");
    document.write(articulos);

  </script>
</body>
</html>
```

Podemos colocar um elemento como **undefined**, para que seu valor seja indefinido (ou desconhecido);

```
articulos[0]= undefined;
```

Para descobrir a posição de um elemento:

```
var
posicion=articulos.indexOf("pantalones");
```

Converter um array em texto:

```
var miString=articulos.join();
```

Converte-o em um **string** separada por vírgula.

Para fazer o contrário, ou seja, para converter um **string** em um **array**, o faremos com o método **split()**. Dentro dos parâmetros, colocaremos o personagem que atuará como um separador.

```
var cadena="nombre1,nombre2,nombre3";
var miArray=cadena.split(",");
```

Para ordenar um array, usaremos o método o método o método **short()**. Para ordenar em ordem inversa o método **reverse()**;

```
cadena.sort();
cadena.reverse();
```

Isto classifica alfabeticamente um array.

Travessar arrays com for

Às vezes, precisaremos percorrer todos os elementos de uma matriz em busca de um elemento ou elementos específicos.

Para atravessar um array, use o loop **FOR**:

```
document.write("<ul>");
var nombres = ["Angel", "Sara",
"Manolo", "Ana"];
for (var i = 0; i < nombres.length;
i++) {
    document.write("<li>" +
nombres[i] + "</li>");
};
document.write("</ul>");
```

Exemplo, encher um array e depois atravessar por ela:

```
var trabajadores = new Array();
var contador = 0;
var persona;

while (persona != "salir") {
    persona = prompt("Introduce nombre");
    trabajadores[contador] = persona;
    contador++;
}
trabajadores.pop();
for (var i = 0; i <
trabajadores.length; i++) {
    document.write(trabajadores[i] +
"<br>");
}
```

Travessar arrays com for each

Há uma modificação do loop **FOR** criado especificamente para facilitar o trabalho com **arrays**, é o loop **foreach**.

Utilizamos a função **foreach** que vai receber o elemento por parâmetro.

```
document.write("<ul>");
var nombres=["Angel", "Sara",
"Manolo", "Ana"];
nombres.forEach((elemento)=>{
  document.write("<li>" + elemento + "</li>");
});
document.write("</ul>");
```

Opcionalmente, a função **foreach** pode receber mais parâmetros.

Um segundo parâmetro pode ser o índice.

```
nombres.forEach(elemento, index) => {
  document.write("<li>" + index + " - "
+ elemento + "</li>");}
```

Travessar arrays com for in

Outra maneira de atravessar um array é com o loop **for in**.

```
for( let nombre in nombres){
  document.write("<li>" + nombres[nom
bre] + "</li>");
};
```

Neste caso, **nombre** é o índice.

Buscando em um array

Para pesquisar um array, podemos usar o método **find()** que tem dentro de si uma função de carga.

```
var
busqueda=nombres.find(function(nombre){
  return nombre==nombre1;
});
```

busqueda armazena o texto que estamos procurando.

Isto pode ser simplificado colocando da seguinte forma:

```
var busqueda=nombres.find(nombre =>
nombre==nombre1;);
```

Da mesma forma, podemos pesquisar o índice com o método **findIndex()**.

```
var busca = nombres.findIndex(nombre => nombre === nombre1);
```

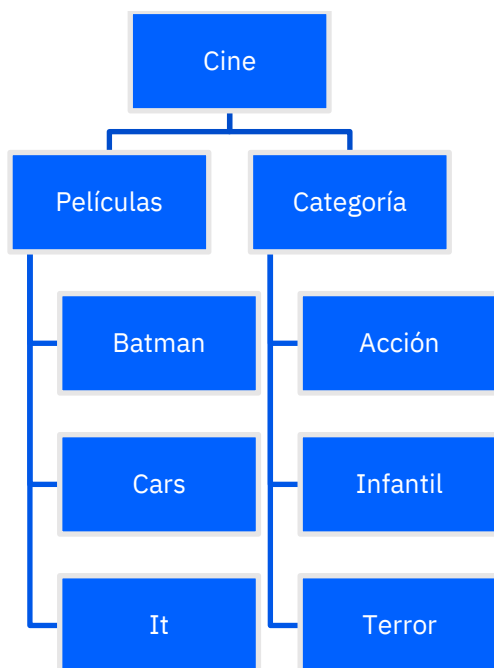
Outro método interessante é `some()` o que nos permite verificar se algum dos elementos da matriz atende a uma determinada condição. Por exemplo, para verificar se há elementos maiores que 40:

```
var numeros = [3, 45, 33, 65, 776];
var busca = numeros.some(numero => numero > 40);
```

Loja `true` ou `false`.

Arrays multidimensionais

Eles não são nada mais do que um array composto de outros arrays. Ou seja, um array que, em vez de ter números ou strings dentro dela, tem outros arrays. Coloquialmente, poderíamos dizer que se trata de um array de arrays.



```
var peliculas = ['Batman', 'Cars', 'It'];
var categoria = ['Acción', 'infantil', 'Terror'];
var cine = [peliculas, categoria];
console.log(cine);
```

Para acessar um elemento, por exemplo, uma categoria de criança, devemos indicar a posição que esse elemento ocupa na primeiro array (no array de containers) seguido pela posição que ocupa no segundo array (o array contido):

```
console.log(cine[0][1]);
```

Neste exemplo, ele imprimiria `Cars`.

Os arrays multidimensionais não são uma ferramenta muito utilizada, mas é útil para que o estudante esteja familiarizado com elas.

POO

Já vimos em tópicos anteriores os princípios da programação orientada a objetos. Bem, o Javascript é uma linguagem de programação orientada a objetos, por isso compartilha os conceitos-chave deste paradigma de programação. Entretanto, o POO em Javascript não é normalmente usado, uma vez que o POO é projetado para outros tipos de programas diferentes das páginas da web. Na programação web, o uso de classes, objetos, atributos, métodos, etc., não faz muito sentido. Mas vale a pena para o estudante se familiarizar com estes conceitos.

Em Javascript, os objetos têm propriedades e métodos.

As propriedades são modificadas com a nomenclatura do ponto e o valor em vírgulas invertidas:

```
nombreDelObjeto.propiedad= "valor ";
Renault.ancha= "2000 ";
Boton.style.width= "500px ";
Document.write();
Windows.alert();
```

Para chamar os métodos:
`nombreDelObjeto.metodo();`

```
Renault.acelera();
```

Exemplo:

```
<body>
  <input type="button" id="boton">
  <input type="button" id="boton2">
  <script>
    var
miBoton=document.getElementById("boton");
    miBoton.style.width="300px";
    miBoton.style.height="300px";

    var
miBoton2=document.getElementById("boton2"
);
    miBoton2.style.width="300px";
    miBoton2.style.height="300px"
;
    miBoton2.focus();
  </script>
</body>
```

Funções

As funções são definidas como um conjunto de instruções prontas para serem chamadas a qualquer momento. Eles só serão executados se nós os chamarmos. Seu objetivo é, portanto, a reutilização do código.

Sintaxe:

Para utilizar uma função, é necessário primeiro defini-la:

```
function nombre_funcion(){

//Código a ejecutar

}
```

E então, em outro ponto da página **HTML**, chamamos a função:

```
nombre_funcion();
```

Com exceção das funções predefinidas do Javascript, é indispensável chamar uma função para que ela funcione.

Este é um exemplo para declarar uma função denominada `suma`. Esta função lerá o valor de duas variáveis e emitirá o valor da soma das duas variáveis.

```
//Declaración de la función
let num1 =5;
let num2 = 10;

function suma(){
  console.log("La suma de "+ num1 +" y
"+ num2 +" es " + (num1+num2));
}

//Llamada a la función
suma();
```

Nesta ocasião, nossa função tomou duas variáveis já instanciadas e inicializadas, mas podemos criar funções que funcionam com valores que devemos passar para elas quando as chamamos. Estas são chamadas funções com parâmetros.

```
//Declaración de la función

function suma2(num1, num2){
  console.log("La suma de "+num1+"
y "+num2+" es "+(num1+num2));
}

//Llamada a la función
suma(3, 7);
```

Não tomamos mais o valor de `num1` e `num2` de duas variáveis pré-existentes. Nós os passamos como um parâmetro para nossa função.

Na declaração da função, colocamos que nossa função, quando chamada, receberá dois parâmetros, `num1` e `num2`. Portanto, na chamada para a função somos obrigados a passar estes parâmetros ou receberemos um erro.

Ao chamar a função, passamos o valor de `3` para o `num1` e `7` para o `num2`.

Além das funções que criamos de acordo com nossas necessidades, há funções predefinidas em [Javascript](#) prontas para uso, como, por exemplo, a função `Date()` que mostra o tempo do sistema.

```
<html>
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    function fecha() {
      document.write(Date());
    }
  </script>
</head>

<body>
  <h1>Hola es día:
    <script>fecha();</script>
  </h1>
</body>
</html>
```

Escreva uma função que pede o nome do usuário e compõe uma mensagem de boas-vindas com esse nome:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var nombre;
    function pedir_nombre() {
      nombre = prompt("Introduce
nombre");
      document.write(nombre);
    }
  </script>
</head>

<body>
  <p>Hola
  <script>pedir_nombre();</script> ,
cómo estás?
  </p>
</body>
</html>
```

Criar uma função que acrescenta dois números inseridos pelo usuário:

```
<html>
<head>
  <meta charset="utf-8">
  <title>Documento sin título</title>
  <script>
    var num1;
    var num2;
    function suma(num1, num2) {
      num1 = prompt("Introduce número
1: ");
      num2 = prompt("Introduce número 1
:");
      document.write("La suma de " +
num1 + " y " + num2 + " es= " +
(Number(num1) + Number(num2)));
    }
  </script>
</head>

<body>
  <script>
    suma(num1, num2);
  </script>
</body>
</html>
```

Parâmetros de rest y spread

Quando não soubermos exatamente o número de parâmetros que podem atingir nossa função, colocaremos três pontos para nos referirmos ao restante dos parâmetros possíveis. Estes parâmetros serão armazenados em um **array** com o nome do meu parâmetro.

```
function numeros(num1,
num2,...otros){
    document.write(num1 + " , " +
num2+"<br>");
    document.write(otros);
}

numeros(2,3,4,54,332);
```

Neste exemplo **otros** vale **4,54,332**. É um array.

Se não tivesse os três pontos, valeria **4**.

Parâmetros de spread

Similarmente, no exemplo a seguir, criamos um array com dois elementos e a passamos para a função como parâmetro. Se não usarmos os três pontos, o que estamos dizendo é que **num1=[1,10]** e **num2=3** e **otros=[4,54,332]**. Mas se colocarmos os três pontos nele, o que eu recebo é **num1=1**, **num2=10** y **otros=[3,4,54,332]**

```
function numeros(num1,
num2,...otros){
    document.write(num1 + " , " +
num2+"<br>");
    document.write(otros);
}
var arrayNumeros=[1,10];
numeros(...arrayNumeros,3,4,54,332);
```

Funções anônimas

As funções anônimas são aquelas que não têm nome e podem ser armazenadas em uma variável.

```
var miFuncion=function(nombre);
return "El nombre es " + nombre;
```

Para chamá-la:

```
miFuncion("Angel");
```

Callback

Os **Callbacks** em JavaScript são, como seu nome indica, **chamadas de retorno**, significa que quando eu invoco uma função passando-a como parâmetro outra função (a **callback**) esta função de parâmetro é executada quando a função principal tiver terminado de ser executada. Ou quando estamos interessados em...

Este conceito, que parece muito complicado, é na verdade muito simples de entender se utilizarmos exemplos práticos, por exemplo:

```
function funcionPrincipal(callback){
    alert('hago algo y llamo al
callback avisando que terminé');
    callback();
}

funcionPrincipal(function(){
    alert('termino de hacer algo');
});
```

Aqui vemos como a função **funcionPrincipal** é executada ao receber um argumento que é outra função e que é executado após ter terminado seu trabalho chamando a **callback**.

Mas não se preocupe se você não tiver entendido, vamos fazer mais alguns exemplos para ajudá-lo a tomar o jeito, é realmente muito simples.

Seguindo o caso anterior, podemos até fazer o **função callback** receber **argumentos** enviado da função principal...

```
function funcionPrincipal(callback){
    alert('hago algo y llamo al
callback avisando que terminé');
    callback('Miguel');
}

funcionPrincipal(function(nombre){
    alert('me llamo ' + nombre);
});
```

Com isto podemos ver o poder que esta forma de programação pode ter, mas agora vamos imaginar que queremos encadear diferentes funções com **callbacks**, o que é muito simples, seguindo a mesma lógica:

```
function funcionPrincipal(callback1,
callback2, callback3){
    //código de la función principal
    callback1();
    //más código de la función
principal
    callback2();
    //más código de la función
principal
    callback3();
    //más código si fuera necesario
}

funcionPrincipal(
    function(){
        alert('primer callback');
    },
    function(){
        alert('segundo callback');
    },
    function(){
        alert('tercer callback');
    }
);
```

Aqui vemos como ao invocar a função principal, as três funções são passadas como argumentos, que são executados, que poderiam receber parâmetros... como vimos acima, mas desta forma o código é um pouco sujo, pois não podemos ver claramente as funções.

Assim, podemos declarar as funções a serem enviadas como argumentos separados, o que nos permite utilizá-las também em outras partes do código, como vemos no exemplo a seguir:

```
function funcionPrincipal(callback1,
callback2, callback3){
    //código de la función principal
    callback1();
    //más código de la función
principal
    callback2();
    //más código de la función
principal
    callback3();
    //más código si fuera necesario
}

function callback1(){
    alert('primer callback');
}

function callback2(){
    alert('segundo callback');
}

function callback3(){
    alert('tercer callback');
}

funcionPrincipal(callback1,
callback2, callback3);
```

Podemos ver como todas as funções são declaradas e depois passadas como argumentos para a **função principal**, a fim de usá-las dentro da função principal. Mas embora isso seja suficiente para outra entrada, vamos dar uma olhada no uso de outra função JavaScript chamada `setInterval()`, esta função é usada para retardar ações e necessita de dois parâmetros para funções; uma função que invoca a função que vamos usar e um valor numérico que diz em milissegundos o atraso e a atualização da função invocada.

Isto parece uma bagunça, vamos vê-lo em um código onde mostraremos um `alert`, um relógio e um texto, e onde o relógio será modificado a cada segundo e o texto levará 3 segundos para aparecer:

```
<html>
<head>
    <meta charset="utf-8">
    <title>Documento sin título</title>
</head>

<body>
    <h3 id="demo"></h3>
    <h3 id="demo2"></h3>
    <script>
        function funcionPrincipal(callback1,
callback2, callback3){
            //código de la función principal
            callback1();
            //más código de la función
principal
            var miVar =
setInterval(function(){ callback2() },
1000);
            //más código de la función
principal
            var miVar2 =
setInterval(function(){ callback3() },
3000);
            //más código si fuera necesario
        }

        function callback1(){
            alert('primer callback');
        }

        function callback2(){
            var d = new Date();
            var t =
d.toLocaleTimeString();
            document.getElementById("demo
").innerHTML = t;
        }

        function callback3(){
            document.getElementById("demo
2").innerHTML = 'Esto es el callback3';
        }
    </script>
</body>
</html>
```

```

    funcionPrincipal(callback1,
callback2, callback3);
  </script>
</body>
</html>

```

O exemplo é essencialmente o mesmo, mudando as funcionalidades das funções de `callback`, mas adicionamos a função `setInterval()` para que sejam adiados e até atualizados, aproveitando a forma de fazer um relógio em javascript...

Há uma sintaxe para as funções de `callback` chamadas funções de seta, onde a palavra `function` é omitida e substituída por uma seta colocada após o nome da função:

```

funcionPrincipal(function(nombre){
    alert('me llamo ' + nombre);
});

```

Ficaria assim:

```

funcionPrincipal(nombre =>{
    alert('me llamo ' + nombre);
});

```

Funções das setas

As funções de setas são uma maneira de definir as funções de `callback` de uma forma muito mais clara e limpa. Basta substituir a palavra `function` por uma seta. Se você tem um parâmetro, não precisa colocar o parêntese, se você tem dois você tem.

```

function sumame(num1, num2,
sumaYmuestra, sumaPorDos) {
    var suma = num1 + num2;

    sumaYmuestra(suma);
    sumaPorDos(suma);

    return suma;
}

sumame(5, 7, dato => {
    console.log('La suma es ', dato);
},
    dato => {
        console.log('La suma de dos
es ', (dato * 2));

    });

```

Eventos

Os eventos são a forma que temos em [Javascript](#) para controlar as ações dos visitantes e definir o comportamento da página quando eles ocorrem. Quando um usuário visita uma página web e interage com ela, são produzidos eventos e com [Javascript](#) podemos definir o que queremos que aconteça quando os eventos ocorrem.

Para entender os eventos, precisamos conhecer alguns conceitos básicos:

- **Evento:** Algo que acontece. Geralmente os eventos ocorrem quando o usuário interage com o documento, mas podem ser causados por situações fora do controle do usuário, tais como uma imagem que não pode ser carregada porque não está disponível.
- **Tipo de evento:** Este é o tipo de evento que ocorreu, por exemplo, um clique em um botão ou a apresentação de um formulário. Cada tipo de elemento de página oferece diferentes tipos de eventos [Javascript](#).
- **Manipulador de eventos:** é o comportamento que podemos atribuir em resposta a um evento. Ela é especificada por meio de uma função [Javascript](#), que está associada a um tipo específico de evento. Uma vez que o manipulador estiver associado a um tipo de evento em um elemento da página, toda vez que esse tipo de evento ocorrer naquele elemento específico, o manipulador do evento associado será executado..

Com o [javascript](#) podemos definir o que acontece quando um evento ocorre, como um usuário clicando em um botão, editando um campo de texto ou saindo da página.

A manipulação de eventos é o cavalo de batalha para fazer páginas interativas, pois com elas podemos responder às ações dos usuários.

Para definir as ações que queremos realizar quando um evento ocorre, utilizamos manipuladores de eventos. Há muitos tipos de eventos nos quais podemos associar manipuladores de eventos, para muitos tipos de ações de usuários.

Em [Javascript](#), podemos definir eventos de duas maneiras diferentes. Uma forma está no próprio código HTML, usando atributos dos elementos (tags) aos quais queremos associar os manipuladores de eventos. Outra maneira, um pouco mais avançada, é usar os próprios objetos DOM. Vamos dar uma olhada nos dois sentidos abaixo.

Manipuladores de eventos especificados no código HTML

O manipulador de eventos pode ser colocado na tag HTML do elemento da página que queremos responder às ações do usuário. Para isso, usamos atributos especiais nas tags HTML, que são prefixadas com "[on](#)" seguido do tipo de evento. Por exemplo, o manipulador associado com o atributo "[onclick](#)" é executado quando ocorre um clique em uma tag.

Vamos dar um exemplo do manipulador de eventos [onclick](#). Já sabemos que ele é usado para descrever ações que queremos executar quando um clique é feito. Então, se quisermos que uma função seja executada quando um botão é clicado, escrevemos o manipulador [onclick](#) na etiqueta `<INPUT type=button>` desse botão.

Algo como isto:

```
<INPUT type="button" value="pulsame"
onclick="miFunción()"></INPUT>
```

Um novo atributo é colocado na etiqueta que tem o mesmo nome do evento, neste caso, `onclick`. O atributo é compatível com as declarações `JavaScript` que queremos que sejam executadas quando o evento ocorrer.

```
<INPUT type="button" value="pulsar"
onclick="alert('Botón pulsado')">
</INPUT>
```

Dado o código acima, clicar no botão resultará em uma mensagem de alerta com o texto "`Botón pulsado`".

Cada elemento de página tem sua própria lista de eventos suportados, vamos ver outro exemplo de tratamento de eventos, desta vez em um menu suspenso, onde definimos um comportamento quando alteramos o valor selecionado.

```
<SELECT
onchange="window.alert('Cambiaste la
selección')">
  <OPTION value="opcion1">Opcion 1
  <OPTION value="opcion2">Opcion 2
</SELECT>
```

Dentro dos manipuladores de eventos podemos colocar quantas instruções desejarmos, mas sempre separadas por ponto-e-vírgula.

É comum colocar apenas uma instrução, e se você quiser colocar mais de uma, você geralmente cria uma função com todas as instruções e dentro do manipulador você coloca apenas uma instrução, que é a chamada de função.

Vamos ver como várias instruções seriam colocadas em um manipulador.

```
<input type=button value=Pulsar
  onclick="x=30; window.alert(x);
window.document.bgColor = 'red'">
```

São instruções muito simples como atribuir `x` o valor 30, fazer uma janela de alerta com o valor `x` e mudar a cor do fundo para vermelho.

No entanto, tantas instruções colocadas em um manipulador são um pouco confusas, teria sido melhor criar uma função:

```
<script>
  function ejecutaEventoOnclick(){
    var x = 30;
    window.alert(x);
    window.document.bgColor = 'red';
  }
</script>

<FORM>
  <input type="button" value="Pulsar"
onclick="ejecutaEventoOnclick()">
</FORM>
```

É sempre uma boa ideia fazer com que um código possa ser mantido e colocar várias instruções em um atributo `onclick` não é uma boa ideia.

Manipuladores de eventos associados ao addEventListener

A segunda maneira de associar os manipuladores de eventos aos elementos da página é através do método `addEventListener()`. Esta é uma forma ligeiramente mais avançada, mas ainda melhora a capacidade de manutenção do código, pois permite uma melhor separação entre o código de funcionalidade e o código de conteúdo.

O código `HTML` deve ser usado simplesmente para definir o conteúdo de nossa página. Se tivermos instruções `Javascript` dentro de tags, colocando atributos tais como `"onclick"` ou `"onchange"` o que estamos fazendo é colocar código a partir da funcionalidade dentro do código `HTML` que não é recomendado. Portanto, a técnica que vamos conhecer agora é ainda mais apropriada, pois nos permitirá escrever o código da funcionalidade (os eventos `javascript`) sem bagunçar o código `HTML`.

Para associar um evento a um elemento da página, precisamos fazer duas etapas:

- - Acesse o objeto sobre o qual queremos definir o evento. Para isso, temos que acessar o `DOM` para localizar o objeto apropriado, que representa a etiqueta na qual queremos associar o manipulador do evento.
- - Sobre o objeto `DOM`, aplicamos `addEventListener()`, indicando o tipo de evento e a função do manipulador.

Por exemplo, temos este elemento de página:

```
<input type="button" id="miBoton" value="Pulsa click">
```

A maneira mais conveniente de acessar um elemento de página e recuperar o objeto `DOM` associado a essa tag é usar o identificador (atributo `"id"`). Neste caso, o identificador é `"miBoton"`. Para acessar esse elemento, usamos o método `getElementById()` do objeto `document`, enviando o identificador.

```
var miBoton = document.getElementById('miBoton');
```

Agora temos o objeto `DOM` associado a esse botão na variável `"miBoton"`. Sobre este objeto, podemos agora invocar o método `addEventListener()`. Dois parâmetros devem ser passados para este método. O primeiro é o tipo de evento que queremos detectar e o segundo é a função do manipulador de eventos que queremos executar quando o evento ocorrer.

```
miBoton.addEventListener('click', function() { alert('Has hecho clic!!') })
```

Clicando no botão, será exibida a mensagem de alerta.

Vamos ver um segundo exemplo, em uma imagem na qual manipulador para o tipo de evento que vamos associar um `"mouseover"`, que é produzido quando o usuário coloca o ponteiro do mouse sobre um elemento.

Temos uma imagem, na qual colocamos um atributo de `id` para chegar a ela.

```

```

Agora associamos o manipulador do evento do tipo "mouseover" com este código Javascript.

```
var miImagen =  
document.getElementById('imagen');  
miImagen.addEventListener('mouseover',  
function() {  
    alert('Has pasado el ratón encima de  
la imagen')  
})
```

Apêndice do evento Javascript

A tabela a seguir resume os eventos mais importantes definidos pelo JavaScript:

Evento	Descrição	Elementos para os quais é definido
onblur	Desmarque o elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Desmarque um item que tenha sido modificado	<input>, <select>, <textarea>
onclick	Clique e solte o mouse	Todos os elementos
ondblclick	Clique duas vezes sucessivamente com o mouse	Todos os elementos
onfocus	Selecione um item	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pressione uma tecla (sem soltar)	Elementos do formulário e <body>
onkeypress	Pressione uma tecla	Elementos do formulário e <body>

onkeyup	Solte uma tecla pressionada	Elementos do formulário e <code><body></code>
onload	A página foi completamente carregada	<code><body></code>
onmousedown	Pressione (sem soltar) um botão do mouse	Todos os elementos
onmousemove	Movendo o mouse	Todos os elementos
onmouseout	O mouse "deixa" o elemento (para sobre outro elemento)	Todos os elementos
onmouseover	O mouse "entra" no elemento (para sobre o elemento)	Todos os elementos
onmouseup	Solte o botão que foi pressionado sobre o mouse	Todos os elementos
onreset	Inicialize o formulário (apague todos os seus dados)	<code><form></code>

onresize	O tamanho da janela do navegador foi modificado	<code><body></code>
onselect	Selecione um texto	<code><input></code> , <code><textarea></code>
onsubmit	Enviar o formulário	<code><form></code>
onunload	A página é abandonada (por exemplo, ao fechar o navegador)	<code><body></code>

DOM

O objetivo do [JavaScript](#) é interagir com o código [HTML](#), para isso temos que ser muito claros sobre o que é o **DOM**. (Document Object Model). Ou seja, a árvore de tags que compõe nosso HTML.

Assumindo que temos um elemento [HTML](#) com um `id=caja` podemos nos referir a ela como:

```
var
cajas=document.getElementById("caja").innerHTML;
```

Desta forma, carregamos na variável `cajas` o valor do elemento com `id=caja`.

Também podemos modificar as propriedades ou o valor do elemento com `id=caja`.

```
var
cajas=document.getElementById("caja");
cajas.innerHTML;
cajas.style.background="red";
```

Outra maneira de selecionar um elemento de nossa página é com `querySelector`.

```
var
cajas=document.querySelector("#caja");
```

Desta forma, também seleciono o elemento com `id=caja`.

Se eu quiser selecionar um item cujo `class=caja`:

```
var
cajas=document.querySelector(".caja");
```

Podemos, portanto, ver como podemos, a partir de `Javascript`, para selecionar e modificar elementos de `HTML` através do uso do `DOM`.

Desta forma, se eu selecionar o elemento, é apenas para selecionar o nome da etiqueta,

```
var
cajas=document.querySelector("caja");
```

com `#` para se referir a seu `ID`

```
var
cajas=document.querySelector("#caja");
```

e com um ponto para se referir a sua classe.

```
var
cajas=document.querySelector(".caja");
```

Seleção de classes e etiquetas

Acabamos de ver como selecionar elementos específicos por seu rótulo, nome ou classe.

Para selecionar todos os elementos de um tipo:

```
var
todos_los_div=document.getElementsByTagName('div');
```

Com isso selecionei todos os elementos do tipo `div` e os coloquei em um `array`. Agora podíamos extrair o conteúdo de um elemento deste `array`:

```
var
contenido=todos_los_div[2].textContent();
console.log(contenido);
```

Isso me mostraria o que está no elemento `[2]` de `array`.

Poderíamos também ter usado o `innerHTML`:

```
var
contenido=todos_los_div[2].innerHTML();
```

A única diferença é que com `innerHTML()`, se eu quiser, posso agregar um novo valor ao conteúdo.

Também podemos selecionar elementos por sua `.class` tag com:

```
document.getElementsByClassName();
```

BOM

O **BOM** (Browser Object Model) contém uma infinidade de propriedades que nos permitem trabalhar com o navegador.

Da mesma forma que no **DOM** podemos selecionar objetos do documento, com o **BOM** podemos selecionar objetos do navegador e modificá-los. Por exemplo, podemos selecionar o:

```
console.log(window.innerHeight);  
console.log(window.innerWidth);
```

Não existem padrões oficiais para o modelo de objeto do navegador (**BOM**), por isso não é amplamente utilizado, mas é interessante para o estudante ao menos saber sobre sua existência e funcionamento.

Mais coisas que podemos fazer:

```
console.log(screen.Width);
```

 Semelhante ao
acima
Puxe para cima o URL carregado e mais
dados:

```
console.log(window.location);
```

Remova o Href:

```
console.log(window.location.href);
```

Abra uma nova aba no navegador:

```
window.open(URL_de_destino);
```