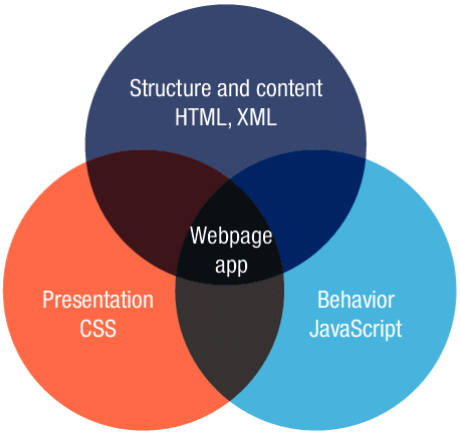
# JavaScript. Sintaxe

# Introdución aos sitios web

Os sitios web están construídos en base ás tecnoloxías web **HTML**, **CSS** e **JavaScript**. Inicialmente as tres tecnoloxías estaban nun mesmo ficheiro HTML, o que complicaba o mantemento da páxina. Hoxe en día estas tecnoloxías están separadas en diferentes ficheiros. Cada unha delas ten diferentes responsabilidades:

- O **HTML** define a **estrutura** e o **contido** da páxina. Almacénase nun ficheiro *.**html**, normalmente nunha carpeta chamada **public.**
- O **CSS** define o **formato** e a **aparencia** usando estilos e definindo a disposición dos elementos na páxina (layout). Estará nun ficheiro .**css** dentro dunha carpeta chamada **styles**.
- O **JavaScript** engade interactividade á páxina e estará en ficheiros .**js** nun directorio chamado **scripts**.




Utilizando un símil gramatical poden explicarse as funcionalidades de HTML, CSS e JavaScript coa frase '**Un dinosauro violeta baila**':

| | | |
|---|---|---|
| Un dinosauro | nome | Contido - HTML |
| violeta | adxectivo | Presentación - CSS |
| baila | verbo | Comportamento, efectos dinámicos - JavaScript |

Utilizando o exemplo desta calculadora en codepen pode probarse a comentar o CSS e o JavaScript para entender a responsabilidade de cada unha das tecnoloxías implicadas no desenvolvemento web: HTML, CSS e JavaScript.

# JavaScript

JavaScript é unha linguaxe de programación que permite engadir dinamismo ao **frontend** ou interface dunha páxina web. Tamén se pode empregar no lado do servidor para interactuar con bases de datos ou co sistema de ficheiros utilizando Node.js.

JavaScript segue a maioría da sintaxe e expresións de Java, convencións de nomenclatura e construcións de control de fluxo básicas. A diferencia de Java, non ten o tipado estático nin a forte verificación de tipos.

A elección do nome de JavaScript, comezando por Java, decidiuse por razóns comerciais, para atraer, en 1996, a persoas programadoras de Java.

As principais características de JavaScript son:

- é unha linguaxe de programación multiplataforma, de alto nivel e lixeira.
- é unha linguaxe **interpretada**, non compilada, o que o fai unha linguaxe de script, de aí o nome JavaScript.
- é unha linguaxe **debilmente tipada**, con tipado dinámico (non se indica o tipo de datos dunha variable ao declarala, podendo incluso cambiarse ao longo da execución).
- é unha linguaxe baseada en **prototipos**, estilo de programación **orientada a obxectos** onde as clases non se definen explicitamente, senón que se derivan de agregar propiedades e métodos a unha instancia doutra clase.
- ten **funcións de primeira clase,** que significa que as funcións son tratadas como variables. É dicir, unha función pode ser pasada como argumento a outra función, pode ser devolta por outra función ou pode ser asignada como valor a unha variable.
- execútase no lado **cliente** (no navegador), permitindo agregar funcionalidade a un sitio web como interactividade coa persoa usuaria (permite crear animacións complexas, botóns sobre os que se pode clicar, menús de popup, …) . Actualmente tamén hai implementacións como NodeJS, para o lado servidor.

JavaScript utilízase para, entre outras cousas:

- **Manipular o DOM** (Document Object Model): cambiar o contido HTML, cambiar os valores dos atributos HTML, cambiar os estilos CSS, ocultar elementos HTML, mostrar elementos HTML, etc.
- **Manexar eventos**, é dicir, é capaz de responder a accións da persoa usuaria como clicks, envío de formularios, arrastrar o rato, pasar co rato por riba dun elemento, … Isto permite, por exemplo, crear menús despregables.
- **Realizar peticións asíncronas** para pedir ou enviar datos permitindo actualizar o contido dunha páxina dinamicamente sen ter que recargala por completo.
- **Crear efectos e animacións**.
- **Manipular información** (filtrar, ordenar, …). JavaScript ten estruturas de datos, como arrays, que permiten o filtrado e ordenado de información de forma fácil.
- **Crear SPAs** (Single Page Applications). Unha SPA e unha única páxina web que permite a actualización dinámica de contido sen ter que recargar toda a páxina. Normalmente créanse utilizando un framework como React, Vue ou Angular aínda que tamén se poden crear con JavaScript puro.

JavaScript contén unha biblioteca estándar de obxectos como **Array**, **Date** e **Math** con funcións para manipulalos de forma fácil e rápida. Ademais, o núcleo de JavaScript pode ser estendido con outros obxectos adicionais:

- JavaScript do lado do **client** proporciona obxectos para controlar un navegador e o seu modelo de obxectos do documento (DOM - *Document Object Model*), permitindo a manipulación do HTML e responder a eventos.
- JavaScript do lado do **servidor** proporciona obxectos para, por exemplo, permitir que unha aplicación se comunique con unha base de datos, manipule arquivos no servidor, ….

Están dispoñibles multitude de APIs (*Application Programming Interfaces*) de JavaScript. As APIs son bloques de código listos para usar que proporcionan funcionalidades adicionais. Xeralmente divídense en dúas categorías:

- APIs do navegador:
  - API do DOM: permite manipular HTML e CSS
  - API de xeolocalización: recupera información xeográfica.
  - Canvas e WebGL: permiten crear gráficos animados en 2D e 3D.
  - APIs de audio e vídeo
- APIs de terceiros:
  - API de Twitter
  - API de Google Maps

JavaScript foi deseñado de forma que se puidese executar nunha contorna moi limitada para permitir ás persoas usuarias confiar na execución dos scripts. Desta forma, os scripts de JavaScript só poden comunicarse con recursos que pertenzan ao mesmo dominio desde o que se descargaron, non poden cerrar ventás que non fosen abertas por eles, non poden acceder aos arquivos do ordenador da persoa usuaria e tampouco poden modificar as preferencias do navegador.

Ademais, cada pestana do navegador ten a súa propia contorna de execución, isto significa que o código executado en cada pestana é independente e non pode interaccionar co código que se execute noutra pestana. Isto é unha boa medida de seguridade para evitar roubos de información entre sitios web.

Actualmente hai librerías e frameworks modernos de JavaScript como **Vue, React** e **Angular**. Estas ferramentas permiten escribir aplicacións web modernas de forma máis rápida e fácil. Todas están baseadas en JavaScript, polo que é necesario ter un bo coñecemento desta linguaxe antes de empezar a aprendelas.

Tamén é posible usar JavaScript para crear aplicacións móbiles nativas usando ferramentas modernas como **React Native** ou **Ionic framework**.

Algunhas das aplicacións de escritorio máis utilizadas están escritas con JavaScript utilizando o framework Electron, como **VS Code**, editor que usaremos durante o curso.

## Un pouco de historia

A principios dos 90, a maioría das conexións a Internet facíanse con módems a unha velocidade máxima de 28,8 kbps. Nesa época as páxinas web comezaron a incluír formularios complexos, polo que xurdiu a necesidade dunha linguaxe de programación que se executara no navegador, xa que era máis rápido facer a comprobación de erros no cliente en lugar de enviar os datos do formulario ao servidor e esperar unha resposta.

JavaScript foi creado en 1995 por Brendan Eich, mentres era enxeñeiro en Netscape, e lanzouse por primeira vez con Netscape 2 a principios de 1996. Uns meses despois, Microsoft lanzou JScript con Internet Explorer 3, que era practicamente idéntico a JavaScript. Pouco despois, Netscape enviou JavaScript a Ecma International, unha organización europea de estándares que resultou na primeira edición do estándar ECMAScript ese ano. O estándar permitiu que todos os navegadores puidesen implementalo.

En 2009, despois de moitas complicacións e desacordos sobre que dirección tiña que levar a linguaxe, lanzouse **ES5** que incluía moitas características novas. Seis anos máis tarde, en xuño de 2015, publicouse a sexta edición (**ES6** ou **ES2015**) do estándar que supuxo unha grande actualización da linguaxe. Dende aquela, cada ano publícase unha nova edición que introduce pequenos cambios. Por exemplo a 15ª edición foi publicada en xuño de 2024.

**NOTA:** nalgúns textos pode verse JavaScript referenciado polo seu estándar **ECMAScript**.

Unha particularidade de JavaScript é que todas as versións son compatibles cara atrás. Isto quere dicir que o código antigo funciona nos navegadores novos, porque JavaScript foi creado sobre a base de que as novas actualizacións non poden facer que a web deixe de funcionar. Non ocorre o mesmo ao revés, é dicir, o código JavaScript actual pode non funcionar nalgún navegador antigo. Debido a isto, existen ferramentas para traducir o código JavaScript a unha versión que garanta o seu funcionamento en navegadores antigos mediante os procesos denominados ***transpilling*** e ***polyfilling.*** Hai ferramentas, como Babel, que permiten automatizar este proceso.

Durante a fase de desenvolvemento web non será necesario utilizar estas ferramentas, xa que se pode garantir o uso da última versión do navegador para comprobar o funcionamento das páxinas. Tan só será necesario usalas na fase de produción.

## Soporte nos navegadores

JavaScript é unha linguaxe en constante evolución. A miúdo aparecen novas propostas que son analizadas e, se se consideran útiles, terminan por incluírse na especificación da linguaxe.

Os navegadores non se adaptan inmediatamente ás novas versións de JavaScript, polo que pode ser un problema usar unha característica moi moderna xa que pode que non estea implementada nalgúns navegadores. Poden encontrarse páxinas en internet que informan da compatibilidade dos diferentes navegadores coas distintas versións de JavaScript. Tamén se pode usar *CanIUse* para buscar a compatibilidade dun elemento concreto de JavaScript, así como tamén dos elementos de HTML5 ou CSS3.

# Agregar JavaScript a unha páxina web

O código JavaScript pode escribirse directamente na consola do navegador, que forma parte das ferramentas de desenvolvemento web. Sen embargo, o habitual é escribir código JavaScript nun ficheiro de texto, ao igual que se escribe o código HTML e CSS.

As instrucións en JavaScript denomínanse sentencias e están separadas por punto e coma (;). Aínda que non é necesario escribir o punto e coma se só hai unha sentencia por liña, é considerado boa práctica poñelo sempre.

Para engadir JavaScript a unha páxina web, tan só é necesario usar o elemento <script> para embeber o código dentro. O elemento <script> pode colocarse no **<head>** ou no **<body>** dependendo de cando se queira cargar o script:

- **JavaScript interno:** incluír o código JavaScript dentro da etiqueta <script> do documento. Exemplo.

```
<script>

  // Código JavaScript

</script>
```

- **JavaScript externo**: incluír o código nun ficheiro externo separado facendo referencia ao código JavaScript. Exemplo.

```
<script src="script.js"></script>
```

  A referencia ao arquivo JavaScript pode ser:
  - unha URL completa. Exemplo.
  - unha ruta absoluta. En realidade é unha ruta relativa no servidor. Exemplo.
  - unha ruta relativa. Exemplo.

  O ficheiro JavaScript ten extensión **.js** e só contén instrucións en JavaScript (non debe incluír etiquetas como <script>). O script comportarase igual que se o código estivese escrito directamente no ficheiro HTML.

  Usar scripts externos é moi útil cando estes ficheiros se reutilizan en varias páxinas web. Ademais, permiten separar o código HTML de JavaScript, permitindo que as páxinas web sexan máis fáciles de manter.

- **Manexadores de eventos en liña**: algunhas veces tamén se encontra código JavaScript mesturado co código HTML. Aínda que é posible mesturar código JavaScript con HTML non se recomenda facelo por ser unha mala práctica:

```
<button onclick="createParagraph()">Click me!</button>
```

# Estratexias para a carga de scripts

Cando se carga unha páxina HTML vanse cargando as etiquetas na orde en que aparecen. Isto tamén se aplica aos scripts, que se van cargando en orden, de arriba abaixo, o que significa que hai que ter coidado coa orde na que se escriben as instrucións, pois poden producirse erros se a orde non é a adecuada.

Se se usa JavaScript para manipular elementos HTML (o DOM), haberá que esperar a que todo o HTML estea cargado antes de executar o script. Se o código JavaScript se carga na cabeceira do documento (<head>), executarase antes de cargar todo o corpo HTML, polo que poderían aparecer erros ao intentar manipular un elemento do DOM que aínda non estivese cargado.

Unha posible solución é usar o evento **DOMContentLoaded**, que é un evento que se produce cando o HTML está completamente cargado e analizado. O seguinte exemplo executará o código JavaScript cando se produza este evento, é dicir, cando o HTML estea completamente cargado:

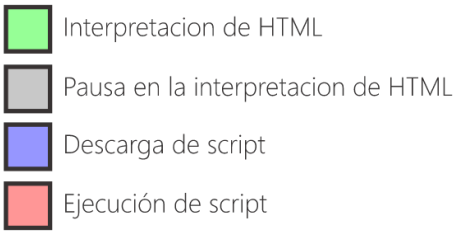
```
<script>
   document.addEventListener("DOMContentLoaded", function() {
     // Código JavaScript
   });
</script>
```

Outra solución, xa pasada de moda, consistía en colocar o script xusto na parte inferior do corpo, antes da etiqueta de cerre </body>, para que se cargara e executara despois de procesar todo o HTML. O problema desta solución é que a carga do script non empeza ata que se cargue o DOM HTML.

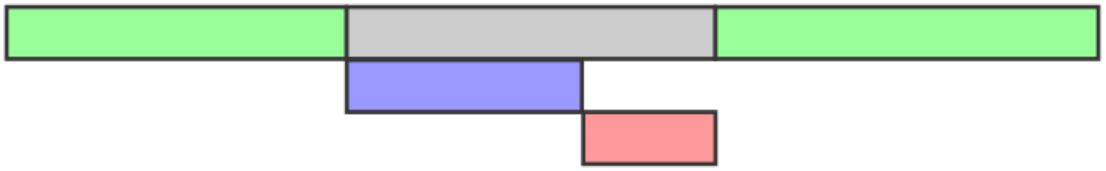
Hoxe en día úsanse os atributos **async** e **defer,** que se explican no seguinte apartado.

## async e defer

Cando se carga unha páxina web, as etiquetas vanse cargando na orde de aparición. As seguintes imaxes indican, en diferentes cores, as fases de carga dunha páxina web que conteña scripts, utilizando a seguinte lenda:

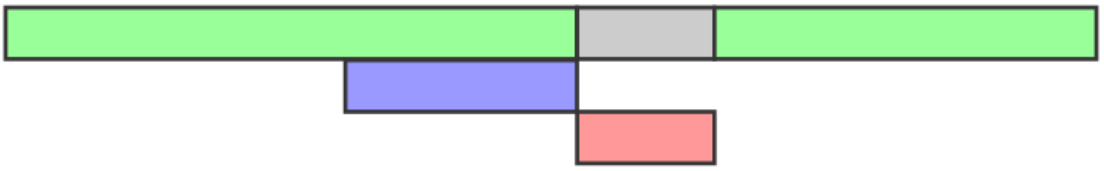


- **<script>**: no proceso de carga dunha páxina web con etiqueta <script>, sen máis atributos, o HTML vaise interpretando ata que aparece a etiqueta <script>. Neste punto detense a carga do HTML e solicítase o arquivo js (no caso de que sexa externo). Unha vez se descarga o arquivo js, este execútase para continuar despois coa interpretación do resto de etiquetas HTML.



- **<script async>**: o atributo async permite que o navegador descargue o arquivo script asincronamente mentres se continúa coa interpretación do código HTML. Unha vez se obtén o arquivo js, detense a interpretación do HTML para executar o script. Despois continuarase coa interpretación do resto de etiquetas HTML.

  Con esta técnica non hai garantías de que os scripts se executen nunha orde específica, polo que só se debe usar **async** cando os scripts da páxina se executen de forma independente, é dicir, que non dependan de ningún outro script.



- **<script defer>**: o atributo defer fai que a execución do script se aprace ata que remate a interpretación do HTML. A descarga do script realízase de forma asíncrona, igual que no caso anterior. Os scripts que aparezan no HTML executaranse todos ao final e na orde de aparición. Non comezarán a súa execución ata que o contido da páxina estea cargado, o que é de utilidade se os script modifican o DOM.

En resumo:

- Tanto **async** como **defer** descargan o script nun fío separado de forma asíncrona, sen bloquear o renderizado da páxina. Por isto, a colocación da etiqueta script debe colocarse no head do documento. Non ten sentido colocala ao final de body.
- **Async** executa o script tan pronto como se descarga, bloqueando o renderizado. Non se garante a orde de execución dos scripts.
- **Defer** garante que os scripts se executan en orde e despois de cargar todo o código.

- **Async** e **defer** non son soportados en navegadores antigos. Nestes casos a solución sería colocar o script ao final do body.
- Tanto **async** como **defer** son ignorados se a etiqueta <script> non ten **src.**

# Comentarios

Igual que con calquera linguaxe de programación, é posible escribir comentarios en JavaScript:

```
/*
  Comentarios de
  varias liñas
*/

// Comentarios dunha liña
```

# Modo estrito

O modo estrito, que apareceu na versión 5 de ECMAScript, é un modo especial que se pode activar en JavaScript e fai máis fácil a escritura de código seguro, evitando erros accidentais durante a programación.

Para activar este modo hai que incluír ao comezo do script a seguinte liña:

**'use strict';**

**NOTA:** ten que ser a primeira sentencia do script (os comentarios están permitidos antes, porque JavaScript ignóraos).

Aínda que se pode activar o modo estrito para funcións individuais, recoméndase activalo para todo o código.

A sentencia “use strict” é ignorada por versións antigas de JavaScript, xa que é interpretado como unha cadea de caracteres.

O modo estrito permite escoller unha variante restrinxida de JavaScript, deixando de lado o modo pouco rigoroso. Entre outras cousas, incorpora algúns cambios na semántica normal de JavaScript e na forma de execución, como por exemplo o lanzamento dalgúns erros que doutra forma serían silenciados:

- Obriga a declarar as variables antes de usalas. É dicir, fai imposible crear variables globais por accidente.

```
'use strict';
let mistypedVariable;

mistypeVariable = 17;   // esta liña lanza un ReferenceError debido a
                        // unha erro no nome da variable
```

- Mostra un erro de execución en certas asignacións que, doutra forma, non darían erros. Por exemplo, non permite asignar valores a propiedades de obxectos definidas como de só lectura.

```
'use strict';

undefined = 5;  // asignación a unha variable global de só lectura
                // lanza unha excepción
```

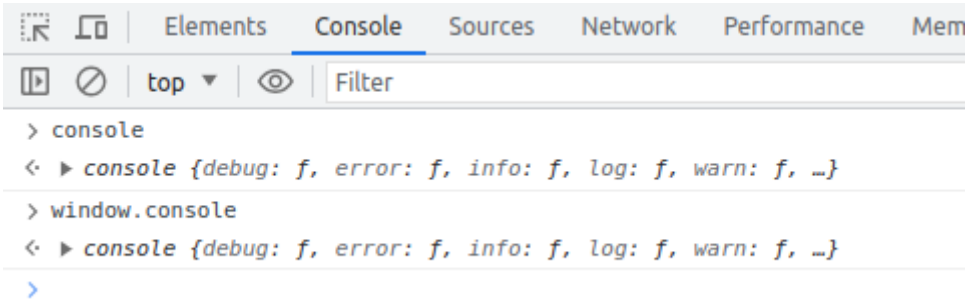
- Mostra un erro de execución cando se usa certa sintaxe que é probable que sexa definida en futuras versións de ECMAScript. O modo estrito converte unha lista de identificadores en palabras reservadas: implements, interface, let, package, private, protected, public, static e yield. No modo estrito non se poden usar estas palabras como nomes de variables.

```
'use strict';
const interface = 'Audio'; // non permite usar a palabra “interface” por se nun futuro
                           // se inclúe no estándar
```

# Mostrar información

JavaScript permite mostrar información por consola utilizando os métodos do obxecto console. O obxecto **console** está accesible a partir da variable global window (que representa a ventá onde se executa o script) ou simplemente como **console**.

Probar a a escribir **console** (ou window.console) na consola do navegador e observar que se mostra información do obxecto console, neste caso os métodos, pois teñen un **f** ao lado:




**NOTA:** En JavaScript existe un **obxecto global** que representa o ámbito global (global scope). Dependendo da contorna onde se execute JavaScript, o obxecto global pode variar. Así, nun navegador o obxecto global é window mentres que en Node.js existe un obxecto global chamado global. Dado que window é o obxecto por defecto nun navegador, pode accederse aos seus métodos e propiedades sen necesidade de explicitar o seu nome (*window*). É dicir, nun navegador, todos os métodos e propiedades dos que non se indica de que obxecto son, execútanse no obxecto *window.*

Aínda que o obxecto console teña moitos métodos, os máis usados son console.log(), console.warn(), console.error() ou console.table() que mostrarán a información na consola do navegador.

Exemplo:

```
console.log(34);
console.log('Mensaxe de texto');

const x = 100;
console.log(x);

console.error('Mensaxe de erro');

console.warn('Warning');

console.table(['apples', 'oranges', 'bananas']);
console.table({ nome: 'Ada', correo: 'ada@mail.com' });
```

Para mostrar un obxecto pode usarse directamente console.log():

```
const someObject = { str: 'Some text', id: 5 };
console.log(someObject);      // {str: 'Some text', id: 5}
```

A información dos obxectos é recuperada cando o obxecto é visto por primeira vez, non cando foi mostrado por pantalla. Observar o comportamento do seguinte código:

```
const obj = {};
console.log(obj);
obj.prop = 123;
```

Os métodos de **console** son usados por persoas desenvolvedoras, non polos usuarios dunha páxina web, xa que estes últimos non soen consultar a consola das ferramentas de desenvolvemento do navegador.

JavaScript tamén permite mostrar información en ventás modais utilizando os seguintes métodos:

- window.alert(): mostra un diálogo modal cunha mensaxe e espera a que a persoa usuaria o cerre. Unha ventá modal evita que a persoa usuaria acceda ao resto da interface do programa ata que o diálogo se cerre. Exemplo.
- window.confirm(): mostra un diálogo modal con unha mensaxe e espera a que a persoa usuaria o cerre ou cancele. O valor devolto é **true** se se pulsou OK e **false** se se pulsou Cancel. Exemplo.
- window.prompt(): mostra un diálogo modal para que a persoa usuaria introduza un texto. O resultado devolto é unha cadea de texto que contén o valor introducido pola persoa usuaria ou o valor null se se pulsa Cancel. Exemplo.

As funcións anteriores tamén poden escribirse sen especificar o obxecto *window*, que é o obxecto global cando JavaScript se executa nun navegador.

# Variables

Unha variable é un contedor para almacenar información. As variables poden almacenar case calquera cousa, non só cadeas e números, tamén poden conter datos complexos e incluso funcións.

Os nomes das variables deben cumprir algunhas recomendacións:

- Por convención recoméndase usar a nomenclatura "**lowerCamelCase**": cando se unan varias palabras para formar o nome dunha variable, a primeira irá en minúscula e as seguintes terán a primeira letra en maiúscula. Exemplos: firstName, lastName, cardNumber, etc.
- O nome dunha variable debe empezar por letra, guión baixo (_) ou dólar ($). O resto dos caracteres poden ser letras, díxitos, símbolo dólar e guión baixo (_).
- Non deben usarse números ao inicio do nome dunha variable.
- JavaScript é sensible a maiúsculas e minúsculas. Por exemplo, as variables **lastName** e **lastname** son dúas variables diferentes.
- Deben evitarse nomes de variables con unha única letra.
- Os nomes das variables debe ser **significativos** do valor que almacenan. Para variables booleanas recoméndase usar nomes como **isGameOver**, que explican correctamente o significado do valor almacenado na variable.
- Non deben usarse palabras reservadas como nomes de variables.

JavaScript é unha linguaxe debilmente tipada. Isto significa que non se indica de que tipo é unha variable ao declarala e incluso o seu tipo pode cambiar durante a execución. Exemplo:

```
let variable;           // declaro variable. Ao non asignar un valor, valerá undefined
variable = "Ola";       // agora o seu valor é 'Ola', por tanto contén unha cadea de texto
variable = 34;          // agora contén un número
variable = [3, 45, 2];  // agora un array
variable = undefined;   // vólvese a asignar o valor especial undefined
```

Para usar unha variable, primeiro hai que creala, ou o que é o mesmo, declarala. JavaScript proporciona catro formas de declarar de variables:

- var: declara unha variable de **ámbito función,** ou **ámbito global** cando a variable é declarada fóra de calquera función. En realidade isto non crea unha nova variable, senón que crea unha nova propiedade no obxecto global.

```
var y = 3;
function foo() {
  var x = 1;

  function bar() {
    var y = 2;
    console.log(x); // 1 (function `bar` closes over `x`)
    x = 11;
    console.log(y); // 2 (`y` is in scope)
  }
```

```
  bar();
  console.log(x); // 11 (`x` is in scope)
  console.log(y); // 3
}

foo();
```

Crear unha variable dentro dun bloque de código non crea un novo ámbito para esa variable, permitindo que sexa referenciada fóra dese bloque:

```
var x = 1;

if (x === 1) {
  var x = 2;
  console.log(x); // Expected output: 2
}

console.log(x); // Expected output: 2
```

O uso de **var** permite **redeclarar** variables e non perderán o seu valor, a menos que se asigne un novo.

```
var a = 1;
var a = 2;
console.log(a); // 2
var a;
console.log(a); // 2; not undefined
```

Recoméndase colocar as declaracións de variables ao inicio do ficheiro, aínda que non é obrigatorio, pois as declaracións de variables son procesadas antes de que se execute calquera instrución do código. Isto é o que se chama "**Hoisting**". Hai que ter en conta que só se aplica o hoisting á declaración e non á inicialización, polo que a variable terá o valor **undefined** ata que se lle asigne un valor.

| Código | Interpretación real do código |
|---|---|
| console.log(bla);  // undefined<br>bla = 2;<br>**var bla;** | **var bla;**<br>console.log(bla);<br>bla = 2; |

**NOTA**: **var** usouse en JavaScript dende 1995 ata 2015 e era a única forma que había para declarar variables. **let** e **const** foron introducidas en 2015. Funcionan de forma algo diferente e resolven algúns erros. Actualmente desaconséllase o uso de **var** e recoméndase usar **let** ou **const**.

- let: declara unha variable con **ámbito de bloque** (espacio delimitado por { }).

  Unha variable declarada dentro dun bloque { } non pode ser accedida dende fóra do bloque. **Ollo** ao declarar variables, por exemplo, dentro dun bloque if, pois non se poderán usar fóra do bloque.

```
let x = 1;

if (x === 1) {
  let x = 2;
  console.log(x);   // Expected output: 2
}

console.log(x); // Expected output: 1
```

  As variables definidas con let non poden ser redeclaradas:

```
let foo;
let foo; // SyntaxError thrown.
```

  Ás variables declaradas con **let** e **const** tamén se lles aplica o **hoisting**, mais ao contrario que **var,** non son inicializadas a *undefined* cando se aplica o hoisting. En realidade, unha variable declarada con let inicialízase cando se executa a liña de código onde se declara. Se non se lle asigna ningún valor, será inicializada a **undefined**.

  Algunhas persoas prefiren ver let e const como **non-hoisting** xa que non está permitido utilizar estas variables antes da súa declaración. Lanzarase unha excepción se se usa unha variable declarada con **let** ou **const** antes de declararse.

```
console.log(bar); // undefined
console.log(foo); // ReferenceError
var bar;
let foo;
```

```
console.log(bar); // undefined
var bar;
let foo;
console.log(foo); // undefined
```

- const: declara unha constante de só lectura e **ámbito de bloque**, como **let**. As constantes almacenan valores que non cambian.

  Ademais, as constantes:

  - non poden ser redeclaradas.
  - deben inicializarse ao ser declaradas.
  - non se pode cambiar o seu valor.
  - se referencian un obxecto ou array, as súas propiedades poden cambiarse.

Exemplo:

```
const PI = 3.1415

console.log (PI);
```

En realidade as constantes almacenan unha **referencia** de só lectura a un valor. Isto non significa que o valor sexa inmutable, senón que o é a referencia. Polo tanto, no caso de apuntar a un obxecto ou un array, as súas propiedades ou elementos poden cambiar. Exemplo:

```
const numeros = [1, 2];
console.log(numeros);
numeros[2] = 3;
console.log(numeros);

const person = { name: 'Ada' };
console.log(person);

person.name = 'Ánxeles';
console.log(person);
```

- **Non usar ningunha palabra reservada**. JavaScript permite usar variables non declaradas, que é equivalente a declarala **global** (con **var).** Non se recomenda usar esta opción.

```
foo = 'f';     // In non-strict mode, assumes you want to create a
               // property named `foo` on the global object

console.log(foo);
```

```
function x() {
  y = 1; // En modo estrito lanza un erro de tipo "ReferenceError" ('use strict')
         // en modo non estrito crea unha variable global
}

x();

console.log(y); // Imprime "1" (en modo non estrito)
```

Recomendacións:

- Declarar as variables ao comezo do ficheiro ou ao comezo das funcións.
- Evitar, na medida do posible, variables globais.
- Usar let para declarar variables. Se é posible, usar constantes.
- Inicializar variables ao declaralas.

**Resumo:**

| **Declaración variables** | **Ámbito** |
|---|---|
| var | función ou global |
| let ou const | bloque de código |
| - | global |

**Exercicio:** Test your skills: variables.

Pode afondarse no estudo das diferentes variables e como funciona JavaScript na seguinte ligazón: JavaScript execution context — lexical environment and block scope (part 3)

# Tipos de datos

JavaScript é unha "**linguaxe de tipado dinámico**", o que significa que, a diferencia doutras linguaxes, cando se declara unha variable non é necesario especificar o tipo de datos que almacenará. JavaScript determina automaticamente o tipo de dato dunha variable cando se lle asigna un valor. Ademais, tamén é posible que o tipo dunha variable cambie.

O último estándar ECMAScript define 8 tipos de datos, 7 tipos primitivos e object:

- Sete tipos de datos primitivos:
  - **Booleano**: tipo de datos lóxico que permite almacenar ***true*** ou ***false***.
  - **Number**: tipo de dato numérico. En JavaScript só hai un tipo de dato numérico que engloba os números enteiros e decimais positivos e negativos. En realidade os números en JavaScript usan o formato de punto flotante de 64 bits de dobre precisión (IEEE 754). Exemplo: 42 ou 3.14159. (O carácter para a coma decimal é '.').

    **NOTA**: **NaN** é un número especial que representa un número inválido: "Not-A-Number". Aparece cando o resultado dunha operación aritmética non pode expresarse como un número. É o único valor en JavaScript que non é igual a si mesmo.

  - **BigInt**: incluíuse en ES2020 para representar números máis grandes dos que é posible representar usando o tipo Number. Para crear un BigInt engádese **n** ao final dun número enteiro ou chámase ao construtor.

    ```
    const x = BigInt(Number.MAX_SAFE_INTEGER);    // 9007199254740991n
    console.log(x + 1n);
    ```

  - **String**: secuencia de caracteres usada para representar texto. As cadeas deben escribirse entre aspas simples ou dobres, aínda que se recomenda usar sempre a mesma decisión en todo o código.

Poden usarse aspas dentro dunha cadea, sempre que sexan diferentes ás usadas para delimitala. En caso contrario, haberá que usar o **carácter de escape **, que indica que o que aparece a continuación sexa tratado como texto e non como parte do código.

Tamén é posible delimitar cadeas co acento grave (`). Exemplo: `Ola mundo!`

```
const bigmouth = 'I\'ve got no right to take my place...';
let answer1 = "It's alright";
```

- **undefined**: é o valor que se asigna automaticamente a unha variable que só foi declarada e non inicializada. Exemplo: **let firstName;**
- **null**: palabra chave que denota ausencia de valor ou "baleiro". En programación, **null** indica que unha variable non apunta a ningún obxecto.

  **NOTA**: aínda que **null** é un tipo primitivo, a operación **typeof null,** devolve "object". Isto pode ser considerado un bug, sen embargo, non se pode corrixir porque deixarían de funcionar moitos scripts. Explicación typeof null.
- **Symbol**: (incluíuse en ECMAScript 2015). É un tipo de dato que garante que a súa instancia é única e inmutable. Normalmente úsanse como chaves de propiedades únicas dun obxecto, para que sexan distintas ás de calquera outro obxecto.

- e **Object**: todos os valores que non son dun tipo primitivo son considerados obxectos: datas (Date), arrays, mapas (maps), conxuntos (sets), funcións, etc. Os obxectos tamén son denominados tipos por referencia (reference types), pois cando se asigna un obxecto a unha variable, a variable almacena a referencia ao obxecto.

  Os obxectos en JavaScript poden verse como unha colección de propiedades (chave - valor) e métodos. Os valores das propiedades poden ser de calquera tipo, incluído outros obxectos. As funcións son obxectos coa capacidade adicional de que poden ser chamadas.

Todos os tipos primitivos, excepto **null** e **undefined**, teñen o seu correspondente obxecto envolvente (wrapper), que proporciona métodos útiles para traballar cos valores primitivos. Así, por exemplo, o obxecto String ten métodos como toLowerCase() que, aplicado directamente a unha cadea, devolve a cadea convertida a minúsculas.

Pode comprobarse o tipo dunha variable usando **typeof**, que devolve unha cadea indicando o tipo do operando, sen avalialo. Exemplo.

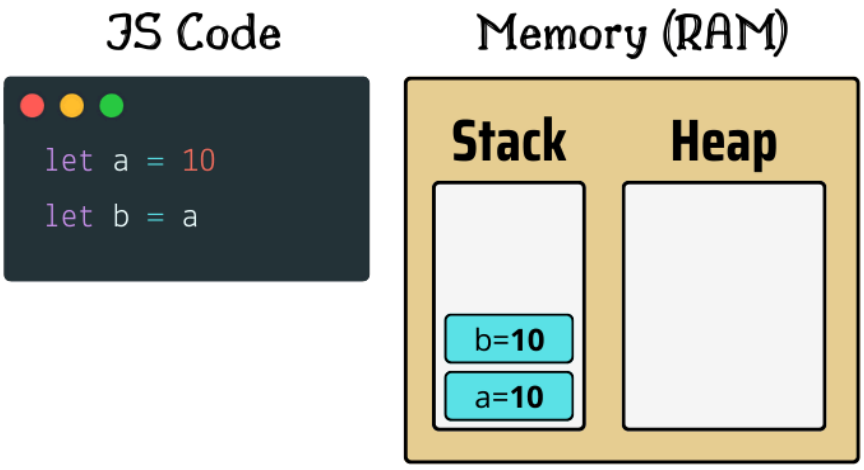
Exemplos:

```
console.log('Ada', typeof 'Ada');
console.log(23, typeof 23);
console.log(8.5, typeof 8.5);
console.log(true, typeof true);
console.log(null, typeof null);
```

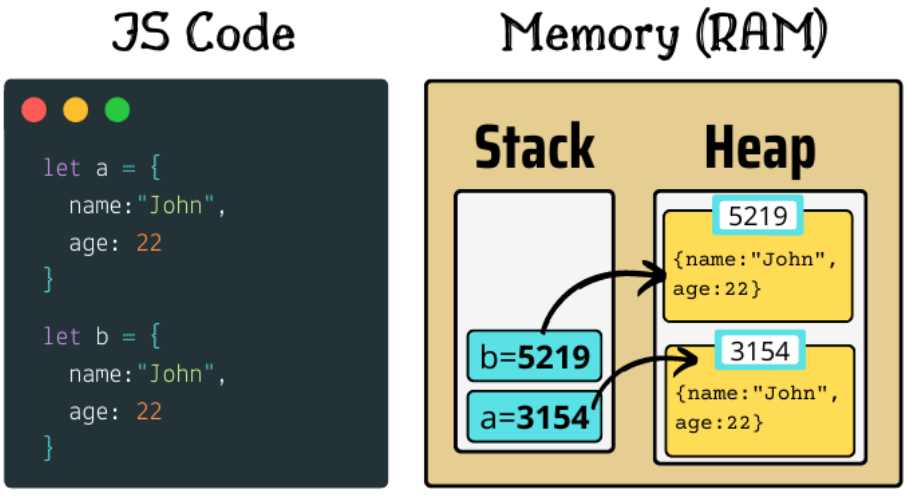
```
console.log(undefined, typeof undefined);
console.log(Symbol('id'), typeof Symbol('id'));
console.log(9007199254740991n, typeof 9007199254740991n);

console.log([1, 2, 3, 4], typeof [1, 2, 3, 4]);
console.log({ name: 'Ada' }, typeof { name: 'Ada' });

function sayHello() {
  console.log('Hello');
}

console.log(sayHello, typeof sayHello);  // function
```

Os tipos primitivos e os obxectos tamén se diferencian por como se almacenan en memoria. Así, mentres os tipos primitivos se almacenan directamente na pila (stack), os obxectos almacénanse no montículo (heap) ao que se accede por referencia.

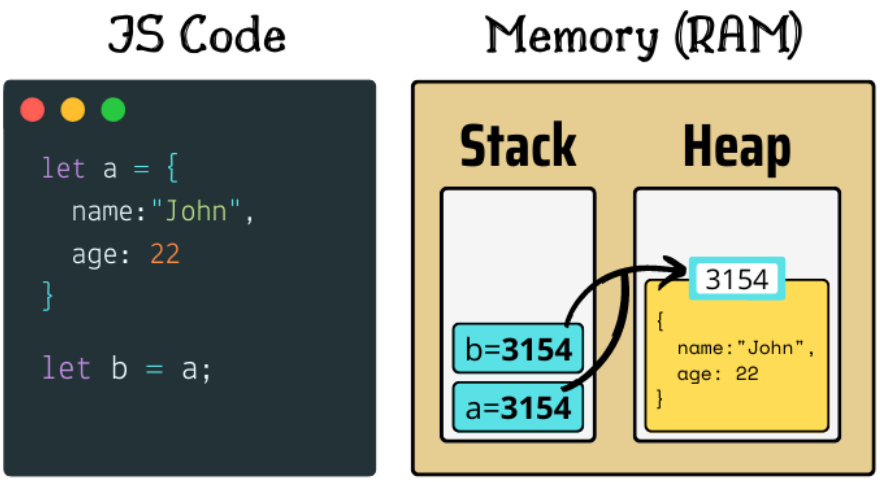
Almacenamento de tipos primitivos en memoria:




Almacenamento de obxectos en memoria:



Ao asignar un obxecto a outro, o que realmente se asigna é a dirección de memoria:




Exemplo:

```
let a = {
    name:"John",
    age:22
};

let b = a;

b.name = "Mark";

console.log('a => ', a);
console.log('b => ', b);
```

# Conversión de tipos

JavaScript non define explicitamente o tipo de dato dunha variable, aínda que internamente toda variable ten un tipo de datos. A conversión de tipos significa cambiar os datos dun tipo a outro. Isto pode suceder de forma implícita (automaticamente) ou de forma explícita (manual).

## Conversión explícita

JavaScript ofrece funcións para cambiar o tipo de dato dunha variable de forma explícita ou manual.

Aínda que existen varios tipos de datos en JavaScript, só se poden facer conversións a **cadeas, números** e **booleanos**:

- **Conversión a cadeas**

  String(value) permite converter valores a cadeas:

  - unha cadea é devolta igual
  - **undefined** convértese en "undefined".
  - **true** convértese en **"true"; false** convértese en **"false"**
  - os números son convertidos usando o mesmo algoritmo que **toString()**.

```
console.log('cadea', String('cadea'));
console.log(undefined, String(undefined));
console.log(true, String(true));
console.log(1, String(1));
```

  O método **toString()** de Number fai o mesmo. Exemplo:

```
let numero = 15;
numero = numero.toString();
console.log(numero, typeof numero);
```

  **NOTA:** ¿Como é posible que se x é un tipo primitivo, non é un obxecto, teña un método **toString()**?. En realidade JavaScript envolve o tipo primitivo nun obxecto do tipo apropiado de forma temporal cando se utiliza con algún método.

- **Conversión a números:**

  Number() converte un string, ou outro valor, a un número:

  - os números son devoltos igual.
  - **undefined** convértese en **NaN**.
  - **null** convértese en **0**
  - **true** convértese en **1; false** convértese en 0
  - as cadeas son parseadas a números.
    - Se falla a conversión, devolve **NaN.**
    - Os espazos ao inicio e ao final son eliminados.
    - Cadeas baleiras son convertidas a **0.**

```
console.log(3.14, Number(3.14));
console.log(undefined, Number(undefined));
console.log(null, Number(null));
console.log(true, Number(true));
console.log(false, Number(false));
console.log('3.14', Number('3.14')); // 3.14
console.log(' ', Number(' ')); // 0
console.log('99 88', Number('99 88')); // NaN
console.log('0x11', Number('0x11')); // 17
```

É habitual ter que converter cadeas a números, por exemplo cando se recolle un valor dun campo input dun formulario. O valor recollido será de tipo String e pode ser necesario convertelo a número.

Tamén pode converterse unha cadea a enteiro usando a función Number.parseInt() (igual á función global parseInt()). Esta función toma como segundo argumento a base. Aínda que a base é opcional, o seu valor por defecto non sempre é 10, polo que se recomenda explicitalo. Se o primeiro carácter non pode ser convertido a número, devolve NaN:

```
parseInt('123', 10); // 123
parseInt("3.65")      // Devolve 3
console.log(parseInt('1a'));  // 1
console.log(parseInt('a1'));  // NaN
```

**NOTA**: a diferencia de Number(), parseInt() só converte strings.

Para converter un número decimal utilizarase parseFloat():

```
parseFloat("3.65");   // 3.65
```

Tamén é posible converter unha cadea a un número utilizando o operador unario "+":

```
let numero = '100';
numero = +numero;
console.log(numero, typeof numero);
```

- **Conversión a booleanos**

Para converter outros tipos de datos a booleanos, pode usarse Boolean():

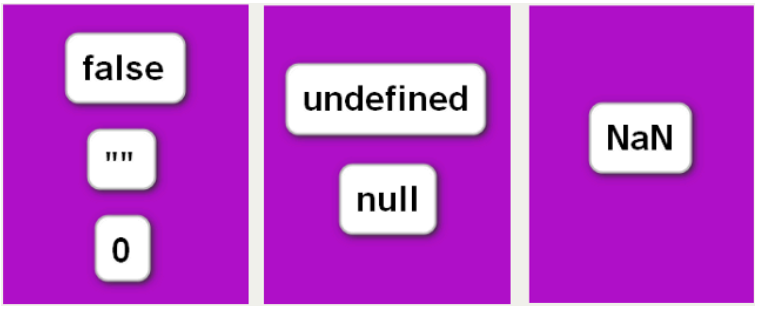
- Os booleanos son devoltos igual.
- **undefined** convértese en **false**
- **null** convértese en **false**
- **0** convértese en **false**
- **NaN** convértese en **false**
- **""** convértese en **false**

**NOTA:** En JavaScript só hai 5 valores que cando se converten a boolean dan o resultado **false.** Estes valores son: **undefined, null, 0, NaN** e **"" (cadea baleira)**. O resto de valores darán **true** cando se convertan a boolean.

```
Boolean(0);    //devolve false
Boolean('');    //devolve false
Boolean(undefined);     //devolve false
Boolean(null);     //devolve false
Boolean(NaN);     //devolve false
```

```
Boolean(234);     //devolve true
Boolean('hello');     //devolve true
Boolean(' ');      //devolve true
```

JavaScript ten 6 valores que representan falsidade:




Só os valores de cada caixa son comparables entre si. Cando se comparan con un valor doutra caixa dan falso

```
console.log(false == ""); // true
console.log(false == 0); // true
console.log(false == undefined); // false
console.log("" == null); // false
```

Sen embargo, moitas veces é máis interesante utilizar a igualdade estrita, que compara o valor e o tipo:

```
console.log(false === ""); // false
console.log(false === 0); // false
```

Normalmente non se fai a conversión explícita de booleanos, senón que é JavaScript o que a fai automaticamente cando ten que avaliar unha condición e a variable non é booleana. Exemplo:

```
let money = 0;
if (money) {
    console.log('Non gastes todo!');
} else {
    console.log('Deberías buscar un traballo');
}

// comprobar se unha variable está inicializada
let height;
if (height) {
    console.log('Height está inicializada');
} else {
    console.log('Height NON está inicializada');
}
// ¿Que pasa se se inicializa height a 0?
```

# Conversión implícita

Habitualmente os operadores e funcións converten automaticamente os valores que se lles pasan ao tipo de datos apropiado para poder realizar a operación indicada. A isto chámase ***type coercion***. Isto soe suceder cando se opera con valores de diferente tipo de datos, como **1 == null**, **2/'5'** ou nalgúns contextos concretos **if (value) {...}**.

```
let numero = 5;
console.log(numero);    // sería equivalente a console.log(numero.toString());
```

Cando JavaScript necesita operar cun tipo de dato "incorrecto", intenta antes facer a conversión automaticamente ao tipo de dato correcto, en lugar de lanzar erros.

Así por exemplo, o operador "+" serve para concatenar cadeas ou sumar números:

- Se un dos operandos é unha cadea, converte o outro a cadea e fai a concatenación
- En caso contrario, os operandos son convertidos a números e faise a suma aritmética.

Exemplo:

```
console.log(5 + null);     // devolve 5          porque null é convertido a 0
console.log("5" + null);   // devolve '5null'    porque null é convertido a 'null'
console.log("5" + 2);      // devolve '52'       porque 2 é convertido a '2'
console.log("5" - 2);      // devolve 3          porque '5' é convertido a 5
console.log("5" * "2");    // devolve 10         porque '5' e '2' son convertidos a 5 e 2
```

Estas conversións implícitas ocorren moi a miúdo, aínda que non sexamos conscientes. Pasa habitualmente cando se aplican operadores a valores de diferentes tipos. Pode consultarse a táboa de conversión de tipos para máis información.

**Exercicio:** Comprobar o resultado das seguintes instrucións:

```
console.log("Ana is " + 18 + " years old");
console.log(1 + "5");
console.log("23" - "10" - 3);
console.log(12 / "6");
console.log("number" + 15 + 3);
console.log(15 + 3 + "number");

let n = '1' + 1;
n = n - 1;
console.log(n);

console.log('10' - '4' - '3' - 2 + '5');
console.log("false" == false);
```

**NOTA:** hai que entender a conversión automática de tipos para evitar cometer erros.

# Operadores

Os operadores permiten combinar variables e valores formulando expresións complexas para crear novos valores.

Existen moitos tipos de operadores: de asignación, aritméticos, de comparación, lóxicos, etc.

**Operadores de asignación:** asignan un valor ao operando do lado esquerdo baseándose no valor do lado dereito.

| **Operador** | **Exemplo** | **Equivalencia** |
|---|---|---|
| = | x = y; | x = y; |
| += | x += y; | x = x + y; |
| -= | x -= y; | x = x - y; |
| *= | x *= y; | x = x * y; |
| /= | x /= y; | x = x / y; |
| %= | x %= y; | x = x % y; |
| **= | x **= y | x = x ** y; |

**Operadores aritméticos:** serven para realizar operacións aritméticas.

| **Operador** | **Nome** | **Exemplo** |
|---|---|---|
| + | Suma | 6 + 9 |
| - | Resta | 20 - 15 |
| * | Multiplicación | 3 * 7 |
| / | División | 10 / 5 |
| % | Resto | 8 % 3 |
| ** | Expoñente | 5 ** 2 (equivalente a $5^2$) |
| ++ | Incremento | y = ++x;  // (x = x + 1; y = x)<br>y = x++;  // y = x; x = x + 1; |
| - - | Decremento | y = - -x;  // x = x - 1; y = x;<br>y = x- -;  // y = x; x = x - 1; |
| + | suma unaria: converte un valor a número | +"3" returns 3.<br>+true returns 1. |

A precedencia das operacións en JavaScript é a mesma que en matemáticas: as multiplicacións e divisións fanse primeiro, despois fanse as sumas e restas. Pode alterarse este funcionamento co uso de parénteses.

**Operadores de cadeas:**

| Operador | Nome | Exemplo |
|---|---|---|
| + | Concatenar cadeas | let text1 = 'John';<br>let text2 = 'Doe';<br>let text3 = text1 + ' ' + text2; |
| += | Concatenar cadeas | let text1 = 'What a very ';<br>text1 += 'nice day'; |

**NOTA:** se un dos operandos do operador + é unha cadea, faise a concatenación de cadeas. Exemplo. En caso contrario, os operandos son convertidos a números.

**Operadores de comparación:** comparan os operandos entre si e devolven un valor booleano baseado na comparación:

- Se os operandos son cadeas, a comparación faise en base á orde alfabética, baseándose nos valores UTF-16.
- Se os operandos non son do mesmo tipo, JavaScript convérteos ao tipo apropiado para facer a comparación, resultando a maioría das veces na conversión a números.

  **NOTA:** cando se utilizan os operadores === ou !== non se fai ningunha conversión.

| Operador | Nome | Exemplo |
|---|---|---|
| === | Igualdade **estrita**.<br>Compara o valor e o tipo de dato sen facer conversión automática de tipos | 5 === 5<br>5 === '5' |
| !== | Desigualdade estrita<br>Compara os elementos e o tipo. | 5 !== '5'<br>5 !== 5 |
| == | Igualdade.<br>Fai conversión automática de tipos. | '1' == 1<br>0 == false |
| != | Desigualdade.<br>Fai conversión automática de tipos | '1' != 2<br>1 != false |
| < | Menor que | 10 < 6 |
| > | Maior que | 10 > 20 |
| <= | Menor igual que | 3 <= 2 |
| >= | Maior igual que | 5 >= 4 |
| ? | operador ternario | (age < 18) ? 'Too young':'Old enough'<br>Exemplo. |

**NOTA:** recoméndase usar a igualdade estrita (===/!==) xa que esta fai a comprobación do valor e o tipo de dato, resultando en menos erros no código. No caso de que haxa que facer conversión de tipos, haberá que facelo manualmente.

Cando se comparan diferentes tipos de datos poden producirse resultados inesperados:

```
console.log(2 < 12); // true
console.log(2 < "12"); // true
console.log("2" < "12"); // false alfabeticamente
console.log("2" > "12"); // true alfabeticamente
console.log("2" == "12"); // false
console.log(2 < "John"); // false - NaN
console.log(2 == "John"); // false
```

**NOTA:** Para asegurar un resultado correcto recoméndase facer a conversión de tipos explicitamente antes da comparación.

Pode consultarse Implicit type coercion in comparison (JavaScript) | by Tinkal | Medium para máis información.

**Exercicio:** comproba o resultado dos seguintes comandos.

```
console.log("2" == 2);

let a = 4, b = 5, c = "5";

console.log("a = " + a + ", b = " + b + ", c = " + c);
console.log("'b == c' -> " + (b == c));
console.log("'b === c' -> " + (b === c));
console.log("'b != c' -> " + (b != c));
console.log("'b !== c' -> " + (b !== c));
console.log("'a == b' -> " + (a == b));
console.log("'a != b' -> " + (a != b));
```

Ollo cando se recollen valores de teclado ou dun formulario. Hai que ter en conta o tipo de dato recollido e facer a conversión, en caso necesario, antes de facer unha comparación:

```
// Faise a conversión a number porque prompt devolve un String
let favourite = Number(prompt('Cal é o teu número favorito?'));

if (favourite === 5) {
    console.log('O 5 é un bo número');
} else {
     console.log('Non escolliches o 5');
}
```

**Operadores a nivel de bit:** tratan os operandos como un conxunto de bits (ceros e uns). Así o número 9 ten a representación 1001. Os operadores a nivel de bit realizan a operación usando a representación binaria pero devolven valores numéricos.

| Operador | Nome | Exemplo | Resultado |
|---|---|---|---|
| & | AND | 5 & 1 | 0001 |
| \| | OR | 5 \| 1 | 0101 |
| ~ | NOT | ~5 | 1010 |
| ^ | XOR | 5 ^ 1 | 0100 |
| << | desprazamento esquerda | 5 << 1 | 1010 |
| >> | desprazamento dereita | 5 >> 1 | 0010 |
| >>> | desprazamento dereita sen signo | 5 >>> 1 | 0010 |

**Operadores lóxicos:** úsanse normalmente con valores booleanos e, neste caso, devolven un valor booleano.

Hoxe en día os operadores  &&, || e ?? devolven o valor dun dos operandos, que pode non ser booleano. Por iso o nome máis apropiado para eles é “operadores de selección de valor”.

- And lóxico (&&): devolve o valor do primeiro operando (de esquerda a dereita) que sexa avaliado como falso, ou o valor do último operando, se todos son true.

```
result = "" && "foo"; // result is assigned "" (empty string)
result = 2 && 0; // result is assigned 0
result = "foo" && 4; // result is assigned 4
```

- O operador || devolve o primeiro valor que pode converterse a true, ou o valor do último operando:

```
"Cat" || "Dog"; // true || true returns "Cat"
false || "Cat"; // false || true returns "Cat"
"" || false; // false || false returns false
false || ""; // false || false returns ""
```

- O operador de negación lóxica (!) devolve false se o valor é true e viceversa.

```
!true; // !t returns false
!false; // !f returns true
!""; // !f returns true
!"Cat"; // !t returns false
```

Os operadores && e || usan lóxica de curtocircuíto, o que significa que algunhas veces non será necesario executar o segundo operando para saber o resultado da operación. Unha das utilidades deste funcionamento é facer unha comprobación antes de realizar unha operación e evitar así erros ou resultados non desexados:

```
const posts = [];
console.log(posts[0]); // mostra undefined, que non é desexable

// mellor sería usar a seguinte opción
posts.length > 0 && console.log(posts[0]);
```

**Exercicios:** comproba o resultado dos seguintes comandos.

```
console.log("'false && false' -> " + (false && false));
console.log("'false && true' -> " + (false && true));
console.log("'true && false' -> " + (true && false));
console.log("'true && true' -> " + (true && true));

console.log("'false || false' -> " + (false || false));
console.log("'false || true' -> " + (false || true));
console.log("'true || false' -> " + (true || false));
console.log("'true || true' -> " + (true || true));

console.log("'!false' -> " + !false);
```

Cando os operadores lóxicos se usan con valores non booleanos poden devolver un valor non booleano:

# Operadores avanzados

## Asignación lóxica AND (&&=)

Este operador só avalía o valor da parte dereita se a parte esquerda é avaliada a true. Este operador funciona por curtocircuíto. A operación **x &&= y** é equivalente a **x && (x = y).**

```
let x = 0;
let y = 1;

console.log((x &&= 0), x); // x = 0
console.log((x &&= 1), x); // x = 0
console.log((y &&= 1), y); // y = 1
console.log((y &&= 0), y); // y = 0
```

## Asignación lóxica OR (||=)

Este operador asigna o valor da parte dereita só se a parte esquerda é falsa. Este operador funciona por curtocircuíto. A operación **x ||= y** é equivalente a **x || (x = y)**.

Exemplo:

```
let x = 0;
let y = 1;

console.log((x ||= 0), x); // x = 0
console.log((x ||= 1), x); // x = 1
console.log((y ||= 1), y); // y = 1
console.log((y ||= 0), y); // y = 1
```

Exemplo:

```
let titulo = '';

titulo ||= 'título baleiro';
console.log(titulo);  // título baleiro
```

## Operador de coalescencia nula (??)

O operador de coalescencia nula (??) é un operador lóxico que devolve o operando da parte dereita cando o operando da parte esquerda é **null** ou **undefined**. En caso contrario, devolve o operando da parte esquerda.

Ao igual que OR e AND, funciona por curtocircuíto, é dicir, o operando da parte dereita só se avalía en caso necesario.

```
const foo = null ?? 'default string';
console.log(foo);  // "default string"
```

Este operador é útil para asignar un valor por defecto a unha variable. Antes, para asignar un valor por defecto a unha variable usábase o operador ||, sen embargo isto non funciona para valores por defecto falsos (0, ‘’, false, NaN):

```
let foo;

 // asigna o valor por defecto “Hello” se a variable non está inicializada
const someDummyText = foo || 'Hello!';  // 'Hello!'
console.log(someDummyText);
```

```
// Observar que pasa cando as variables están inicializadas a 0 ou “”
const count = 0;
const text = "";

// A pesar de ter as variables inicializadas, os valores 0 e “” devolven falso
const qty = count || 42;
const message = text || "hi!";
console.log(qty);     // 42 e non 0
console.log(message); // "hi!" e non ""
```

```
// o operador de coalescencia nula mantén o valor de inicialización da variable
const count = 0;
const text = '';

// o operador de coalescencia nula mantén o valor de inicialización da variable
const qty2 = count ?? 42;
const message2 = text ?? 'hi!';
console.log(qty2); // 0
console.log(message2); // ""
```

## Asignación coalescencia nula (??=)

O operador de asignación lóxica nula (x ??= y) só avalía o operando da parte dereita e asígnao á parte esquerda, se a parte esquerda é **null** ou **undefined**.

Exemplo:

```
let duracion = 125;
let speed;

duracion ??= 100;
console.log(duracion); // 125

speed ??= 25;
console.log(speed);  // 25
```

Ao igual que OR e AND, este operador funciona por **curtocircuíto**, é dicir, o operando da parte dereita só se avalía en caso necesario.

**x ??= y** é equivalente a **x ?? (x = y)**

## Encadeamento opcional (?.)

O operador de encadeamento opcional permite acceder a unha propiedade ou función dun obxecto. Se o obxecto é **null** ou **undefined**, devolve undefined, en lugar de lanzar un erro.

```
const adventurer = {
  name: 'Alice',
  cat: {
    name: 'Dinah'
  }
};

const dogName = adventurer.dog?.name;
console.log(dogName); // expected output: undefined

console.log(adventurer.someNonExistentMethod?.()); // expected output: undefined
```

## Precedencia de operadores

A precedencia dos operadores determina a orde na cal son avaliados respecto aos outros.

Considérese a expresión **a OP1 b OP2 c**, onde a, b e c son operandos e OP1 e OP2 son operadores. Se OP1 e OP2 teñen diferente nivel de precedencia, o que teña a precedencia máis alta executarase antes. Por exemplo:

**console.log((3 + 10 * 2);  // mostra 23**

Se OP1 e OP2 teñen a mesma precedencia, hai que ter en conta a asociatividade:

- asociatividade de esquerda a dereita: avaliarase **(a OP1 b) OP2 c**.
- asociatividade de dereita a esquerda: avaliarase **a OP1 (b OP2 c)**. Por exemplo, o operador de asignación é de asociatividade de dereita a esquerda, entón:

  **a = b = 5; // é igual a escribir a = (b = 5);**

Pode consultarse a precedencia e a asociatividade na seguinte táboa.

Exercicios para practicar: Test your skills: Math.

# Estruturas de control

As estruturas de control serven para facer comprobacións e controlar o fluxo de execución dos programas, tomando decisións en función do resultado de avaliación dunha expresión. Por exemplo, se o número de vidas nun xogo chegou a 0, o xogo debe rematar.

JavaScript ten un conxunto de estruturas de control similares a outras linguaxes da familia de C.

if … else

Permite decidir que instrucións executar en función dunha condición.

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

As chaves **{ }** só son obrigatorias cando hai varias instrucións pertencentes a unha ramificación.

**NOTA**: os valores **false**, **undefined**, **null**, **0**, **NaN** ou **cadea baleira (")** devolverán o valor **false** ao ser avaliados nunha sentencia condicional.

while

Executa as instrucións repetidamente mentres a condición sexa certa.

```
while (condicion)
  sentencia
```

do … while

Bucle que executa unha sentencia ata que a condición sexa falsa. A condición avalíase despois de executar a sentencia, polo que esta se executa polo menos unha vez.

```
do
  sentencia
while (condicion)
```

for

Permite executar un bucle varias veces. A inicialización só se executa a primeira vez. A continuación analiza a condición e, se se cumpre, executa as sentencias. Despois, actualiza o índice e retorna á comprobación da condición. Cando non se cumpra a condición, o bucle remata.

```
for ([inicializacion]; [condicion]; [final-expression])
  sentencia
```

Na inicialización normalmente utilízase unha variable que fai de contador. Cando esta variable se declara con **let** o seu ámbito é o bloque do bucle. Exemplo:

```
for (let i = 0; i < 9; i++) {
  console.log(i);
}

console.log(i); // Erro
```

for...of: permite iterar sobre os **valores** almacenados en obxectos iterables: arrays, cadeas, Map, Set, NodeList (e outras coleccións do DOM), obxecto arguments, etc.

```
for (variable of iterable) {
    sentencia
}
```

Cada iteración crea unha nova variable e asígnaselle un valor da secuencia. Pode ser declarada con const, let ou var. Se se utiliza **const**, a variable non pode ser reasignada dentro do corpo do bucle.

Exemplo:

```
const iterable = [10, 20, 30];

for (const value of iterable) {
  console.log(value);     // logs 10, 20, 30
}
```

for...in: bucle para iterar sobre as **propiedades** enumerables de tipo cadea dun obxecto (ignora as propiedades symbol).

```
for (variable in object) {
    sentencia
}
```

Exemplo:

```
const object = { a: 1, b: 2, c: 3 };

for (const property in object) {
  console.log(property);   // logs a, b, c
}
```

Os índices dun array son propiedades do obxecto array. O bucle **for…in** recorrerá todos os índices do obxecto array e o resto de propiedades do obxecto, en caso de que as tivese.

```
const iterable = [10, 20, 30];

for (const value in iterable) {
  console.log(value); // logs 0, 1, 2
}
```

No caso de arrays con elementos baleiros, o bucle **for…in** non iterará sobre os elementos baleiros, sen embargo o bucle **for…of** si.

```
const array1 = ["a", "b", , "c"];

for (let element in array1)
    console.log(element);   // logs 0, 1, 3

for (let variable of array1)
    console.log(variable);  // logs a, b, undefined, c
```

Para recorrer un array recoméndase usar un bucle **for**, **Array.prototype.forEach()** ou o bucle **for...of**.

switch: avalía unha expresión e executa as sentencias despois da primeira cláusula “case” que concorde co valor da expresión (**igualdade estrita ===**), ata encontrar unha sentencia **break**. Saltarase á cláusula **default,** opcional, se ningunha das cláusulas “case” concorda

coa expresión. Se non se engade a instrución **break**, a execución continuará na seguinte liña.

```
switch (expression) {
  case value1: // expression === value1
    // Sentencias a executar cando a expresión e value1 coinciden
    [break;]
  case value2:
    // Sentencias a executar cando a expresión e value2 coinciden
    [break;]
  ...
  case valueN:
    // Sentencias a executar cando a expresión e valueN coinciden
    [break;]
  [default:
    // Sentencias a executar cando a expresión non coincide con ningún valor
    [break;]]
}
```

Exemplo.

Existen dúas instrucións que modifican a iteración dos bucles:

- **continue:** fai que a iteración do bucle actual remate e empeza a seguinte. Nun bucle while a seguinte sentencia a executar é a condición e nun bucle for, actualiza a expresión.
- **break**: fai que o bucle, ou switch actual, remate. A seguinte sentencia a executar é a que vai a continuación do bucle ou switch.

try…catch: esta sentencia está composta dun bloque **try** e un bloque **catch**, **finally** ou ambos. Primeiro execútase o código do bloque **try** e no caso de que lance unha excepción, executarase o código do bloque **catch**. O código do bloque **finally** sempre será executado ao final.

```
try {
  tryStatements
} catch (exceptionVar) {
  catchStatements
} finally {
  finallyStatements
}
```

Exemplo:

```
try {
  nonExistentFunction();
} catch (error) {
  console.error(error);
  // Expected output: ReferenceError: nonExistentFunction is not defined
}
```

**Exercicios:**

**NOTA:** para evitar solicitar os datos ao usuario, nestes exercicios recoméndase inicializar directamente as variables.

1. Crea unha variable que almacene un día da semana de luns a domingo. En función do valor da variable mostra unha mensaxe indicando se o día é laborable ou non.
2. Crea 3 variables e inicialízaas con 3 números diferentes. Mostra por pantalla o maior dos 3 números.
3. Escribe os números pares do 0 ao 30, incluídos.
4. Escribe as potencias de 2, dende $2^0$ ata $2^{20}$. Para cada potencia debe saír un texto similar a “2 elevado a 0 = 1”.
5. Inicializa unha variable a un número, calcula o seu factorial e mostra a resultado por consola. (5! = 5*4*3*2*1).
6. Cálculo do IMC (índice de masa corporal). IMC = peso (kg) / [estatura (m)]$^2$
    a. Almacena en variables o peso e altura de dúas persoas.
    b. Calcula o IMC das dúas persoas.
    c. Indica que persoa ten o maior IMC cunha cadea similar a: 'O IMC (25.3) da primeira persoa é maior que o da segunda persoa (22.5)!'

# Funcións

As funcións son un dos bloques de construción fundamentais en JavaScript. Unha función é un anaco de código deseñado para realizar unha tarefa.

As funcións son importantes por diversos motivos:

- Axudan a estruturar os programas para facer o seu código máis comprensible e máis fácil de modificar.
- Permiten repetir a execución dun bloque de código sen ter que volver a escribilo.

A definición dunha función, tamén chamada declaración de función, consta das seguintes partes básicas:

- O nome da función.
- Lista de parámetros entre parénteses e separados por comas.

  Os parámetros pásanse por valor, polo que o cambio do valor dun parámetro dentro da función non afecta ao resto do código do programa. Se se pasa un obxecto, pásase a referencia ao obxecto (a súa dirección de memoria), polo que os cambios nas propiedades vense fóra da función. Os arrays tamén son obxectos, polo que os cambios dos seus valores veranse fóra da función.

  Dentro da función, os parámetros compórtanse como variables locais.

  **NOTA:** Se se chama a unha función con menos parámetros dos declarados, o valor dos parámetros non pasados será undefined. Dende ES6, poden definirse valores por defecto para os parámetros.

- Chaves de inicio e fin { }.
- O corpo da función pode ter tantas sentencias como sexan necesarias e pode declarar as súas propias variables, que son locais á función.
- Cando se chega a unha sentencia **return**, a función remata a súa execución. A sentencia return pode ser usada para devolver un valor. Se non se usa unha sentencia return, JavaScript devolve **undefined**.

  **NOTA:** As sentencias que vaian despois de return non se executarán.

Sintaxe:

```
function nomeFuncion(parametro1, parametro2=valorPorDefecto, …) {
    // instrucións
    return valor; // Se a función devolve un valor, engadir a sentencia return.
}
```

Sintaxe da chamada a unha función:

```
valorRetornado = nomeFuncion(parametro1, parametro2, …);
```

Exemplo:

```
function logger() {
  console.log('Mensaxe');
}

logger();

function sumar(operando1, operando2) { // función con parámetros e devolve un valor
  return operando1 + operando2;
}
console.log(sumar(5, 2));
```

É importante **diferenciar a definición da función da chamada á función**. Cando se define unha función, esta non se executa. Na definición, unicamente se lle asigna un nome e especifícase que facer cando se chama á función. Para que a función se execute haberá que invocala ou chamala usando o operador **(),** ademais de pasarlle os parámetros necesarios. No seguinte exemplo, **toCelsius** refírese á declaración da función, mentres que **toCelsius()** refírese ao resultado da función:

```
function toCelsius(fahrenheit) {
  return (5 / 9) * (fahrenheit - 32);
}
console.log(toCelsius);
console.log(toCelsius(68));
```

JavaScript tamén aplica o **hoisting** á declaración de funcións, levando a declaración completa da función ao inicio do ámbito actual. Desta forma, o código que aparece á esquerda na seguinte táboa funcionará xa que JavaScript procesará a declaración da

función ao inicio, tal e como se mostra no código da dereita, que é equivalente ao do lado esquerdo.

| | |
|---|---|
| console.log(square(5)); // 25<br><br>function square(n) {<br>  return n * n;<br>} | // All function declarations are effectively at the top of the scope<br>function square(n) {<br>  return n * n;<br>}<br><br>console.log(square(5)); // 25 |

**Exercicio:**

1. Crea unha función que reciba como parámetro un prezo e unha porcentaxe de desconto. A función debe calcular o prezo final aplicado o desconto e devolver este valor.

# Referencias

Para a elaboración deste material utilizáronse, entre outros, os recursos que se enumeran a continuación:


- JavaScript | MDN
- JavaScript Guide
    - Grammar and types
- JavaScript Tutorial
- let - JavaScript | MDN
- Desarrollo Web en Entorno Cliente | materials