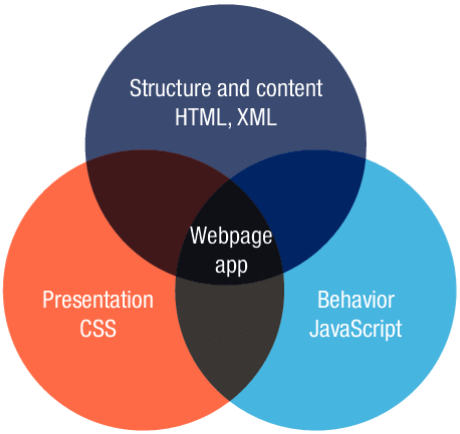
JavaScript. Sintaxe

Introdución

Os sitios web están construídos en base ás tecnoloxías web **HTML**, **CSS** e **JavaScript**. Cada unha destas tecnoloxías ten diferentes responsabilidades.

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:

- 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**.



Outra forma de explicar as funcionalidades de HTML, CSS e JavaScript, usando o símil gramatical coa frase '**Un dinosauro violeta baila**' é:

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

Exemplo calculadora en codepen. Probar a comentar o CSS e o JavaScript e entender a responsabilidade de cada unha das tecnoloxías implicadas no desenvolvemento web: HTML, CSS e JavaScript.

As principais características de JavaScript son:

- é unha linguaxe de programación multiplataforma, de alto nivel e lixeira.
- é unha linguaxe interpretada, non compilada.

- executase no lado cliente (no navegador) permitindo agregar funcionalidade a un sitio web como interactividade co usuario e páxinas web dinámicas, aínda que actualmente hai implementacións, como NodeJS, para o lado servidor,
- é unha linguaxe orientada a obxectos, aínda que está baseada en prototipos e non en clases.
- é unha linguaxe debilmente tipada, con tipado dinámico (non se indica o tipo de datos dunha variable ao declarala e incluso pode cambiarse).

JavaScript permite engadir elementos dinámicos e interactivos ás páxinas web: animacións complexas, botóns nos que se poida facer clic, menús emerxentes, etc. Exemplos básicos do que pode facer JavaScript:

- JavaScript pode cambiar o contido HTML.
- JavaScript pode cambiar os valores dos atributos HTML.
- JavaScript pode cambiar os estilos CSS.
- JavaScript pode ocultar elementos HTML.
- JavaScript pode mostrar elementos HTML.

JavaScript segue a maioría da sintaxe de 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. Realmente o nome de JavaScript decidiuse por razóns comerciais, para atraer, en 1996, a persoas programadoras de Java.

JavaScript foi deseñado de forma que se executara nunha contorna moi limitada que permitira aos usuarios confiar na execución dos scripts. Desta forma, os scripts de JavaScript non poden comunicarse con recursos que non pertenzan ao mesmo dominio desde o que se descargou e tampouco poden cerrar ventás que non fosen abertas por eles. Ademais, os scripts tampouco poden acceder aos arquivos do ordenador do usuario e tampouco poden ler ou modificar as preferencias do navegador.

Actualmente hai librerías e frameworks modernos de JavaScript como **React**, **Angular** ou **Vue**. 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 e tamén aplicacións de escritorio usando a ferramenta Electron.

JavaScript contén unha biblioteca estándar de obxectos como **Array**, **Date** e **Math** e un conxunto básico de elementos da linguaxe como operadores, estruturas de control e declaracións. Ademais:

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

Tamén é interesante a utilización de APIs (*Application Programming Interfaces*) que proporcionan funcionalidades adicionais para usar no código JavaScript. As APIs son bloques de código listos para usar. 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

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. Cando unha persoa cubría un formulario era máis rápido facer a comprobación de erros no cliente en lugar de enviar os datos ao servidor e esperar por 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 13 edición foi publicada en xuño de 2022.

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 creouse 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. Pódese procesar o código JavaScript para garantir o seu funcionamento en navegadores antigos mediante un proceso denominado ***transpilling.*** Hai ferramentas, como Babel, que permiten automatizar este proceso. Non será necesario realizar este proceso durante a fase de desenvolvemento web, xa que se pode garantir o uso da última versión do navegador para comprobar o funcionamento das páxinas.

Soporte nos navegadores

Os navegadores non se adaptan inmediatamente ás novas versións de JavaScript, polo que pode ser un problema usar unha característica moi moderna dado que pode haber partes dos programas que non funcionen en moitos navegadores. Na páxina de *Kangax* pode verse a 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.

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.

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:

- **JavaScript interno:** incluír o código JavaScript dentro da etiqueta <script> na cabeceira (<head>) ou no corpo (<body>) do documento. Exemplo.

```
<script>

  // Código JavaScript

</script>
```

- **JavaScript externo**: incluír o código nun ficheiro externo separado. Na cabeceira (<head>) do ficheiro HTML incluír a 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 utilizan en diferentes 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 correcta. 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 analizar todo o corpo HTML, polo que poderían aparecer erros. Ver vídeo explicativo.

Unha posible solución é usar o evento **DOMContentLoaded**, que é un evento que se produce cando o HTML está completamente cargado e analizado. Pode colocarse o JavaScript de forma que se execute cando se produza este evento, o que evitará este erro:

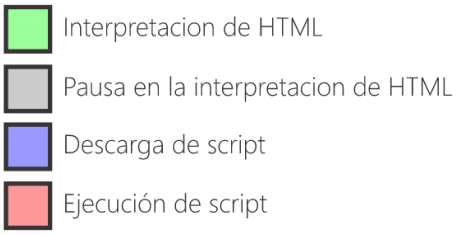
```
<script>
   document.addEventListener("DOMContentLoaded", function() {
     ...
   });
</script>
```

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 despois de procesar todo o HTML. O problema desta solución é que a carga/procesamento do script está bloqueada ata que se cargue o DOM HTML.

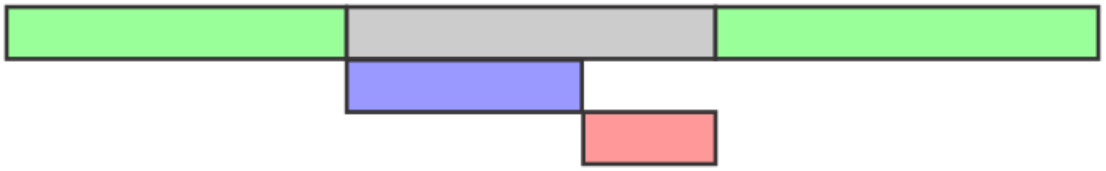
Hoxe en día úsanse os atributos **async** e **defer,** que se explican no seguinte apartado.

async e defer

Cando se carga unha páxina web, as etiquetas vanse cargando na orde de aparición. A continuación indícase con gráficos as diferentes fases de carga dunha páxina web que conteña scripts utilizando a lenda da seguinte imaxe:

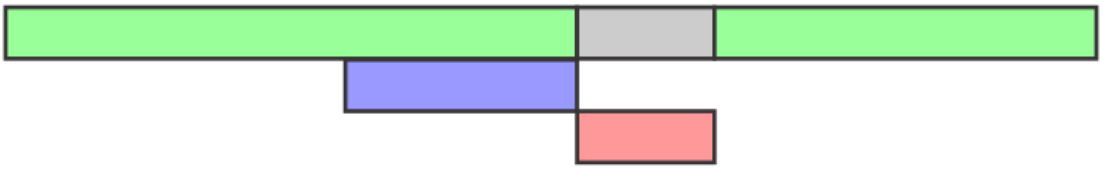


- **<script>**: no proceso de carga dunha páxina web con etiqueta <script> sen máis atributos o HTML vaise interpretando ata que aparece a etiqueta <script>. Neste punto detense a carga do HTML e solicítase o arquivo js (no caso de que sexa externo). Unha vez se obtén o arquivo js, execútase para continuar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

  Aínda así, con esta técnica non hai garantías de que os scripts se executen nunha orde específica, polo que se recomenda usar esta técnica cando os scripts da páxina se executen de forma independente e 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 o caso anterior. Os scripts que aparezan no HTML executaranse todos ao final e na orde en que aparezan. Non se executarán ata que o contido da páxina estea cargado, o que é útil se os script dependen do DOM.

En resumo:

- Tanto async como defer descargan o script nun fío separado, sen bloquear o renderizado da páxina.
- Async executa o script tan pronto como se descargan, bloqueando o renderizado. Non se garante a orde de execución.
- Defer garante que os scripts se executan en orde e despois de cargar todo o código.

Comentarios

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

```
/*
  Comentarios de
  varias liñas
*/

// Comentarios dunha liña
```

Mostrar información

JavaScript permite mostrar ventás modais para pedir información. As funcións que o fan son:

- window.alert([mensaxe]): 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(mensaxe): 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(mensaxe, [Texto por defecto]): 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.

Estas funcións tamén poden escribirse sen *window*, xa que en JavaScript todos os métodos e propiedades do que non se indica de que obxecto son executánse no obxecto *window.*

Se o que quere é mostrarse información para depurar o código utilizaranse os métodos console.log(mensaxe), console.warn(mensaxe) ou console.error(mensaxe) que mostrarán a información na consola do navegador.

Variables

Unha variable é un contedor para almacenar un valor. 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 "**lower camel case**": 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 usar 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. Exemplo.
- Evitar 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 usar 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:

  O seu **ámbito** é o seu contexto de execución, que pode ser a **función** que a contén ou **global,** para unha variable declarada fóra de calquera función.

  Poden ser **redeclaradas** e non perderán o seu valor, a menos que se reasigne un novo.

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

foo();
```

  Dado que as declaracións de variables son procesadas antes de que se execute calquera instrución do código, declarar unha variable en calquera sitio é equivalente a declarala ao comezo do programa. Isto é o que se chama "**Hoisting**". Só se aplica o hoisting á declaración e non á inicialización. Isto implica que a variable terá o valor **undefined** ata que se lle asigne un valor.

Código	Interpretación real do código
bla = 2; **var bla;**	**var bla;** bla = 2;

  NOTA: **var** usouse en JavaScript dende 1995 ata 2015. **let** e **const** foron introducidas en 2015 e son as recomendadas hoxe en día.

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

  As variables declaradas con **let** teñen ámbito de bloque. 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.

```
var a = 1;
var b = 2;

if (a === 1) {
  var a = 11; // the scope is global
  let b = 22; // the scope is inside the if-block

  console.log(a);  // 11
  console.log(b);  // 22
}

console.log(a); // 11
console.log(b); // 2
```

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 con un valor por defecto cando se aplica o hosting. Lanzarase unha excepción se se usa unha variable declarada con **let** ou **const** antes de inicializarse:

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

NOTA: en realidade, a 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**.

Cando se creou JavaScript, a única forma de declarar variables era usando a palabra chave **var**. En versións máis modernas de JavaScript, introduciuse a palabra chave **let**, que funciona de forma algo diferente e resolve algúns erros. Actualmente desaconséllase o uso de **var** e recoméndase usar **let** ou **const**.

- 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.

Por convenio, para diferencialas das variables, as constantes declararanse en maiúsculas. Exemplo:

const PI = 3.1415

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

- **Non usar ningunha palabra reservada**. JavaScript permite usar variables non declaradas, que é equivalente a declarala **global** con **var.** En realidade isto non crea unha variable, senón que crea unha nova propiedade no obxecto global. 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

Test your skills: variables.

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 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. 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.). Exemplo: 'Ola mundo!'.

    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.

    ```
    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 representa unha referencia que apunta a ningún obxecto ou dirección inválida.

      NOTA: aínda que **null** é un tipo primitivo, a operación **typeof null,** devolve “object”. Isto é considerado un bug, sen embargo, non pode ser corrixido porque deixarían de funcionar moitos scripts.

    - **Symbol**: (incluiuse en ECMAScript 2015). É un tipo de dato que garante que a súa instancia é única e inmutable. Normalmente úsanse como claves 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: arrays, funcións, valores compostos, etc. Esta distinción é moi importante porque os valores primitivos e os obxectos compórtanse de forma diferente cando son asignados e cando son pasados como parámetro a unha función. Os obxectos en JavaScript poden verse como unha colección de propiedades (clave - valor). Os valores das propiedades poden ser de calquera tipo, incluído outros obxectos. As funcións son como obxectos coa capacidade adicional de que poden ser chamadas.

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

Exemplos usando os tipos de datos máis habituais:

```
let javascriptIsFun = true;
console.log(javascriptIsFun);

console.log(typeof true);
console.log(typeof javascriptIsFun);
console.log(typeof 23);
console.log(typeof 'Jonas');

// pode reasignarse o valor dunha variable e cambiar o tipo de dato -> tipado dinámico
javascriptIsFun = 'YES!';
console.log(typeof javascriptIsFun);

// undefined
let year;
console.log(year);
console.log(typeof year);

// reasignar un valor, non é necesario volver a declarar a variable con let
year = 1991;
console.log(typeof year);

// bug de JavaScript
console.log(typeof null);
```

Conversión de tipos

JavaScript non define explicitamente o tipo de datos das variables. A operación de conversión de tipos permite cambiar o tipo de datos dunha variable, sendo necesario facelo para poder executar certas instrucións. Por exemplo, cando se len datos dende teclado, normalmente lense como tipo String e pode ser necesario convertelos a outro tipo.

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.

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

Cando JavaScript intenta operar cun tipo de dato "incorrecto", intenta antes facer a conversión automaticamente ao tipo de dato correcto. 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
```

Pode consultarse a táboa de conversión de tipos para máis información.

NOTA: o operador "+" serve para concatenar cadeas ou sumar números. Prevalece a concatenación cando a situación o permite.

Estas conversións implícitas ocorren moi a miúdo, aínda que non sexamos conscientes diso. Pasa habitualmente cando se aplican operadores a valores de diferentes tipos. 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.

Conversión explícita

A pesar de que é posible implementar manualmente a conversión de tipos, raras veces se fai, xa que habitualmente JavaScript fai a conversión automaticamente. Para facelo utilízanse funcións proporcionadas pola linguaxe.

Aínda que hai varios tipos de datos en JavaScript, a conversión explícita só se aplica a **cadeas, números** e **booleanos**:

- **Converter cadeas a números:**

  Number() converte un string ou outro valor a un número. Number é un obxecto primitivo envolvente que permite manipular valores numéricos. O obxecto Number ten constantes e métodos para traballar con números. O principal uso do obxecto Number(valor) é converter un string, ou outro valor, a un de tipo numérico. Exemplo:

  ```
  Number('3.14');   // retorna o número 3.14
  Number(' ');      // retorna 0
  Number('99 88'); // NaN
  ```

  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:

  ```
  parseInt('123', 10); // 123
  parseInt(3.65)      // Devolve 3
  ```

  A función parseInt() devolve o NaN se a cadea non se pode converter nun número.

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

  ```
  parseInt(false);      // devolve NaN
  Number(false);      // devolve 0
  ```

- **Converter números a cadeas**

  String() permite converter números a cadeas. Exemplo.

```
String(123);       // devolve 123
String(100 + 23);  // devolve 123
```

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

```
(123).toString();
(100 + 23).toString();
```

- **Converter booleanos a números**

  Number() tamén se usa para converter booleanos a números.

```
Number(false)     // devolve 0
Number(true)      // devolve 1
```

- **Converter booleanos a Strings**

  String() permite converter booleanos a Strings.

```
String(false)      // devolve 'false'
String(true)       // devolve 'true'
```

- **Conversión a booleanos**

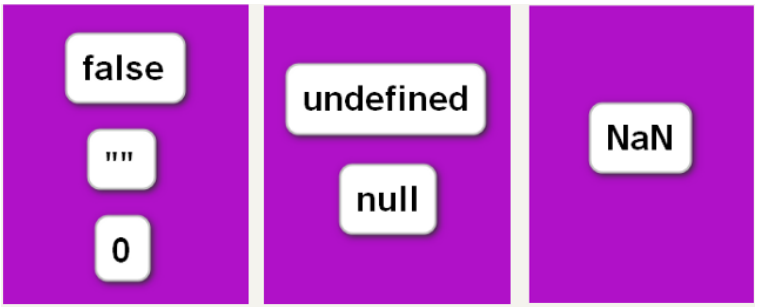
  Para converter outros tipos de datos a booleanos, pode usarse Boolean().

  En JavaScript só hai 5 valores que cando se converten a boolean dan o resultado **false.** Estes valores son: **0, “” (cadea baleira)**, **undefined, null** e **NaN**. 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
```

É dicir, JavaScript ten 6 valores que representan falsidade:



Só os valores de cada caixa son comparables entre sí. 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
```

NOTA: normalmente non se fai a conversión explícita de booleanos, senón que JavaScript faina 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 definida
let height;
if (height) {
    console.log('Height está definida');
} else {
    console.log('Height NON está definida');
}
// ¿Que pasa se se inicializa height a 0?
```

Arrays

Un array é unha variable de tipo especial que almacena unha lista de valores, sendo posible acceder a cada un destes valores de forma individual. Non é necesario que todos os valores almacenados nun array sexan do mesmo tipo.

Exemplos de creación de arrays:

```
const list = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];

const randomNumbers = ['tree', 795, [0, 1, 2]];      // array multidimensional

const people = [];
people[0]= "Ana";
people[1]= "Erea";
people[2]= "Navia";

// Outra forma, aínda que menos habitual, de crear Arrays
const anos = new Array(1991, 1984, 2008, 2020);
```

NOTA: recoméndase que os nomes de arrays sexan en plural ou que fagan referencia a un conxunto de valores.

Os elementos do array están numerados, empezando por 0. O número fai referencia ao índice do elemento no array.

Operacións básicas con arrays:

- Acceder a un elemento do array:

  ```
  const list = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
  const elemento =  list[0]

  // arrays multidimensionais
  const randomNumbers = ['tree', 795, [0, 1, 2]];
  const numero = randomNumbers[2][2];
  ```

- Modificar un elemento do array: list[0] = 'tahini';

  Fixarse que aínda que a variable se declara como constante (**const**), poden modificarse os seus elementos.

- Lonxitude: propiedade length. A lonxitude dun array sempre será unha unidade maior que o índice do último elemento: list[list.length - 1].

  NOTA: dado que .length é unha propiedade (valor precalculado) e non é unha función, non hai que usar ().

Modo estrito

O modo estrito, que apareceu na versión 5 de ECMAScript, é un modo especial que se pode activar en JavaScript que 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';

Para activar o modo estrito, a liña anterior 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.

“use strict” é ignorado por versións anteriores 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, fai varios cambios na semántica normal de JavaScript:

- 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
```

- Non permite eliminar variables/obxectos/funcións.
- Non permite escribir en propiedades de obxectos definidas como de só lectura.
- Prohibe 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
```

Operadores

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

Hai moitas categorías de operadores: matemáticos, de comparación, lóxicos, de asignación, etc

Operadores de asignación:

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:

Operador	Nome	Exemplo
+	Suma	6 + 9
-	Resta	20 - 15
*	Multiplicación	3 * 7
/	División	10 / 5
%	Resto	8 % 3
**	Expoñente	5 ** 2 (equivalente a 5 * 5)
++	Incremento	y = ++x; // (x = x + 1; y = x) y = x++; // y = x; x = x + 1;

- -	Decremento	y = - -x; // x = x - 1; y = x;
		y = x- -; // y = x; x = x - 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'; let text2 = 'Doe'; let text3 = text1 + ' ' + text2;
+=	Concatenar cadeas	let text1 = 'What a very '; text1 += 'nice day';

NOTA: o operador + usado con cadeas e números, devolverá unha cadea. Exemplo.

Operadores de comparación:

Comparan valores entre si e devolven un valor booleano: true ou false.

Operador	Nome	Exemplo
===	Igualdade **estrita**. Compara o valor e o tipo de dato. Non fai conversión automática de tipos	5 === 2 + 4
!==	Desigualdade estrita Compara os elementos e o tipo.	5 !== '5' 5 !== 2 + 3
==	Igualdade. Fai conversión automática de tipos.	'1' == 1 0 == false
!=	Desigualdade. Fai conversión automática de tipos	'1' != 2 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' Exemplo.

NOTA: recoméndase usar a igualdade estrita (===/!==) xa que se comproba o valor e o tipo de datos, resultando en menos erros no código. No caso de que haxa que facer conversión de tipos, mellor facelo manualmente.

Cando se comparan diferentes tipos de datos poden producirse resultados inesperados. Cando se compara unha cadea con un número, JavaScript intenta converter a cadea a número para facer a comparación. A cadea baleira convértese a 0 e unha cadea non

numérica convértese en NaN. NaN non é igual a si mesmo. Se os dous operandos a comparar son cadeas, faise a comparación alfabeticamente.

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

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

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

Exercicios: 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 para 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');
}
```

Operadores lóxicos:

Operador	Nome	Exemplo
&&	and lóxico	(x < 10 && y > 1)
||	or lóxico	(x == 5 || y == 5)

!	negación lóxica	!(x == y)

Os operadores && e || usan lóxica de curtocircuíto, o que significa que algunhas veces non será necesario executar o segundo operando, dependendo do resultado do primeiro. Isto é útil para verificar obxectos nulos antes de acceder aos seus atributos:

```
const name = o && o.getName();
```

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);
```

Operadores a nivel de bit:

Operador	Nome	Exemplo
&	AND	5 & 1
|	OR	5 | 1
~	NOT	~5
^	XOR	5 ^ 1
<<	desprazamento esquerda	5 << 1
>>	desprazamento dereita	5 >> 1
>>>	desprazamento dereita sen signo	5 >>> 1

Operadores avanzados

Operador de coalescencia nula (??)

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

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

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

Este operador é útil para asignar un valor por defecto a unha variable. Ata agora, para facelo, usábase o operador || coa súa lóxica de curtocircuito:

```
let foo;

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

```
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 and not 0
console.log(message); // "hi!" and not ""
```

```
const myText = ''; // An empty string (which is also a falsy value)

const notFalsyText = myText || 'Hello world';
console.log(notFalsyText); // Hello world

// o operador de coalescencia nula mantén o valor de inicialización da varible
const preservingFalsy = myText ?? 'Hi neighborhood';
console.log(preservingFalsy); // '' (as myText is neither undefined nor null)
```

Asignación lóxica nula (??=)

O operación de asignación lóxica nula (x ??= y) só fai a asignación se x é **null** ou **undefined**.

```
const a = { duration: 50 };

a.duration ??= 10;
console.log(a.duration); // expected output: 50

a.speed ??= 25;
console.log(a.speed); // expected output: 25
```

Ao igual que OR e AND, este operador funciona por **curtocirtuito**, é dicir, o operador 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. Exemplo.

```
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. Isto permitirá usar o nome dunha variable para comprobar se é **true** ou se está inicializada. Exemplo: **if (isOldEnough)** …

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
  statement
while (condition)
```

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 ([initialization]; [condition]; [final-expression])
  sentencia
```

for...of: permite iterar sobre obxectos iterables, como arrays ou cadeas. Exemplo de iteración sobre un Array e iteración sobre unha cadea.

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

```
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). Exemplo con obxectos.

Os arrays tamén son obxectos con un índice que é considerado como unha propiedade enumerable. O bucle **for…in** recorrerá todas os índices do array en orde crecente, polo que funciona como unha forma de iterar sobre un array. Exemplo con arrays.

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

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

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

NOTA: No caso de arrays con elementos baleiros, o bucle **for…in** non iterará sobre os elementos baleiros, polo que é mellor usar o bucle **for…of**

```
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
```

Diferencias for…of e for…in. A sentencia for…in itera sobre as propiedades de tipo cadea enumerables, mentres que o for…of itera sobre os valores que o obxecto iterable define como iterables.

```
const iterable = [3, 5, 7];
iterable.foo = "hello";

for (const i in iterable) {
  console.log(i); // logs "0", "1", "2", "foo"
}

for (const i of iterable) {
  console.log(i); // logs 3, 5, 7. Non mostra “hello”
}
```

switch: a sentencia switch avalía unha expresión, comparando o seu valor con unha serie de cláusulas “case”. Executaranse as sentencias despois da primeira cláusula “case” que concorde co valor da expresión, ata encontrar unha sentencia **break**. Saltarase á cláusula **default** se ningunha das cláusulas “case” concorda coa expresión.

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

A cláusula **default** é opcional.

```
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;]]
}
```

try…catch: esta sentencia está composta dun bloque **try** e un bloque **catch** ou **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
}
```

Exercicios:

1. Pide por teclado o día da semana (luns - domingo) e mostra unha mensaxe indicando se é laborable ou non.
2. Pide 3 números por teclado e indica cal é o maior.
3. Escribe os números pares do 0 ao 30.
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. Crea unha variable e asígnalle un número aleatorio entre 0 - 100. Elabora un programa para que o usuario adiviñe o número aleatorio. É dicir, debe pedirse un número por teclado ao usuario e informar se o número introducido é maior ou menor que o número a adiviñar. Volverase a pedir outro número ata que sexa adiviñado o secreto.
6. Pide un número por teclado ao usuario. Calcula o factorial dese número e mostra a resultado por consola. (5! = 5*4*3*2*1).
7. Pide un número por teclado ao usuario. Despois crea unha variable de tipo array e asígnalle a lista de números enteiros dende o 0 ao número tecleado polo usuario. Ademais, fai a suma de todos os valores contidos no array e mostra por consola o resultado da suma.

Utiliza un bucle for para recorrer o array. Proba a facelo das 3 formas posibles: for con contador, for…in e for…of.

8. Cálculo do IMC (índice de masa corporal):
    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.

Unha función consta das seguintes partes básicas:

- O nome da función.
- 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, os cambios nas propiedades vense fóra da función. Se se pasa como parámetro un array, os cambios dos seus valores veranse fóra da función.

  Dentro da función, os parámetros compórtanse como variables locais.

- 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.
- 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.

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 ao obxecto función, mentres que **toCelsius()** refírese ao resultado da función:

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

Ademais das funcións que se poden implementar, hai moitas funcións proporcionadas pola linguaxe que permiten realizar tarefas comúns sen ter que programalas manualmente. Exemplo: console.log(), parseInt(), …

Existen dúas formas de definir funcións: usando unha declaración ou unha expresión.

- **Declaración de funcións:**

  Unha función pode definirse usando unha declaración.

  Sintaxe dunha declaración de función:

  ```
  function nomeFuncion(parámetros) {
    // código a executar
    // return …;
  }
  ```

Unha declaración de función utiliza a palabra chave **function** seguida do nome da función e parénteses.

Por convenio, os **nomes** das funcións seguen as mesmas normas que os nomes das variables.

- **Expresión función**

Unha función tamén pode ser definida usando unha expresión. Desta forma, a función pode ser almacenada nunha variable, que será usada como función. Exemplo.

```
const square = function (number) {
  return number * number;
};
const x = square(4); // x gets the value 16
console.log(x);
```

A función anterior en realidade é unha **función anónima**, é dicir, funcións que non levan nome. As funcións anónimas son útiles para pasar como argumento a outra función.

```
function map(f, a) {
  const result = [];
  for (let i = 0; i < a.length; i++) {
    result[i] = f(a[i]);
  }
  return result;
}

const f = function (x) {
  return x * x * x;
}

const numbers = [0, 1, 2, 5, 10];
const cube = map(f, numbers);
console.log(cube);
```

Tamén é posible proporcionar un nome para unha expresión función. Desta forma permítese á función referirse a ela mesma e fai máis fácil identificar a función durante a depuración de código:

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
}

console.log(factorial(3))
```

As funcións definidas desta forma son expresións e, como toda expresión, esta produce un valor que se almacena nunha variable, que actuará como función.

Parámetros dunha función

Ademais de indicar os parámetros dunha función explicitamente, é posible acceder aos argumentos a través do array **arguments**.

Os argumentos pasados a unha función almacénanse nun obxecto similar a array. Pódese acceder aos argumentos coa expresión **arguments[i]**, onde i é o índice da posición que ocupa o argumento. O total de argumentos indícase con **arguments.length**.

Usando o obxecto **arguments** é posible chamar a unha función con máis argumentos que os declarados.

```
function myConcat(separator) {
  let result = ""; // initialize list

  // Empeza en 1 porque o índice 0 corresponde co parámetro “separator”
  for (let i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}

console.log(myConcat(", ", "red", "orange", "blue"));
console.log(myConcat("; ", "elephant", "giraffe", "lion", "cheetah"));
```

Os **parámetros Rest** permiten representar un número indefinido de argumentos como un array.

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));    // expected output: 6

console.log(sum(1, 2, 3, 4));   // expected output: 10
```

Só o último parámetro dunha función pode ser un parámetro Rest e vai precedido por “...”.

Diferencias entre os parámetros rest e o obxecto arguments:

- Os parámetros rest son só aqueles aos que non se lles asignou un nome, mentres que o obxecto arguments contén todos os argumentos que se lle pasaron á función.
- O obxecto arguments non é un array real, mentres que os parámetros rest son instancias de Array, o que significa que os métodos map ou forEach se poden aplicar directamente.

En versións actuais do código recoméndase usar os parámetros rest.

Funcións frecha (Arrow functions)

As funcións frecha permiten escribir unha expresión función de forma máis resumida.

Para entender mellor as funcións frecha, vaise facer a descomposición dunha función tradicional anónima:

```
// Función tradicional anónima
function (a) {
  return a + 100;
};

// Descomposición dunha función frecha
// 1. Eliminar a palabra "function". Colocar unha frecha entre os argumentos e a chave
(a) => {
  return a + 100;
};

// 2. Borrar as chaves e a palabra "return", pois sobreenténdese.
(a) => a + 100;

// 3. Borrar os parénteses dos argumentos
a => a + 100;
```

Sintaxe dunha función frecha:

```
// un parámetro
param => expression
(param) => expression

// múltiples parámetros requiren parénteses.
(param1, paramN) => expression

// Varias sentencias requiren chaves {}
param => {
  const a = 1;
  return a + param;
}
```

Cando a función frecha só ten unha sentencia non é necesario escribir **return** nen **as chaves {}**. Se a función frecha ten máis dunha liña será necesario incluír as **chaves** e a instrución **return**. Exemplo:

```
// ES5
const x = function(x, y) {
  return x * y;
}

// ES6 - función frecha equivalente
// pode omitirse a sentencia return e as chaves se a función só ten unha sentencia
const x = (x, y) => { return x * y };
const x = (x, y) => x * y;
console.log(x(5, 2));

// poden omitirse os parénteses se a función só ten un parámetro.
const calcAge = birthYear => 2037 - birthYear;
```

As funcións frecha non teñen o seu propio **this.** Cando se usa this nunha función frecha, faise referencia aao obxecto Window.

Resumo tipos funcións

O seguinte exemplo mostra como crear a mesma función usando unha declaración, unha expresión e unha función frecha:

```
// Declaración de función
function calcAge1(birthYear) {
  return 2037 - birthYear;
}

// Expresión función
const calcAge2 = function (birthYear) {
  return 2037 - birthYear;
};

// Función frecha
const calcAge3 = birthYear => 2037 - birthYear;

console.log(calcAge1(2000));
console.log(calcAge2(2000));
console.log(calcAge3(2000));
```

Hoisting de funcións

Hoisting é o comportamento de JavaScript de mover as declaracións á cima do ámbito actual. Aplícase á declaración de variables e á declaracións de funcións. Isto permite que as funcións poidan ser chamadas antes de ser declaradas.

```
myFunction(5);

function myFunction(y) {
  return y * y;
}
```

NOTA: nin ás funcións definidas usando unha expresión nin ás función frecha se lles aplica o hoisting.

Funcións auto-invocadas (Self-Invoking functions)

Hai outra utilidade das funcións anónimas. JavaScript proporciona un mecanismo para definir e invocar simultaneamente unha función usando unha expresión. A isto chámase Immediately invoked function expression (IIFE) (Expresión de función executada inmediatamente). A función invócase automaticamente, sen que sexa necesario chamala ou invocala.

```
(function () {
    statements
})();
```

É un patrón de deseño coñecido como función autoexecutable e componse de dúas partes. A primeira é a función anónima con alcance léxico encerrado polo operador (). Isto impide o acceso a variables fóra do IIFE. A segunda parte é o operador () que executa a función inmediatamente.

Asignar o IIFE a unha variable almacena o valor de retorno, non a definición da función. Exemplo:

```
const result = (function () {
    const name = 'Barry';
    return name;
})();

result; // 'Barry'
```

NOTA: non se poden auto-invocar declaracións de funcións.

Exercicios:

1. Crea unha función frecha que devolva o cubo dun número pasado como parámetro.
2. Crea unha función á que se lle pase un array e devolva como resultado un array cos elementos impares do array de entrada.

```
Exemplo:
arrayEntrada = [10, 2, 3, 5, 7, 8, 23, 50]
arraySaida = [3, 5, 7, 23]
```

3. Crea unha función que sume todos os valores pasados como parámetros, sendo estes un número indeterminado.
4. Crea unha función á que se lle pasen varios números como parámetros (un número indeterminado de parámetros). Debe devolver a media dos números pasados. Proba a realizala con diferentes bucles for (con contador, for…in e for…of).
5. Fai unha función á que se lle pase un DNI (ex: 12345678w ou 87654321T) e devolva se é correcto ou non.
6. Crea unha función que reciba un array bidimensional de lonxitude variable que se corresponda cun escenario do xogo de Buscaminas. Este array almacenará un -1 nas posicións onde hai minas e un 0 en caso contrario. A función debe devolver un array bidimensional onde cada posición que non teña mina, debe ter a información do número de minas adxacentes (diagonal, horizontal e vertical).

```
Exemplo:
arrayEntrada = [[0, 0, -1, 0],
                [0, -1, -1, 0]];
arraySaida = [[1, 3, -1, 2],
              [1, -1, -1, 2]];
```

7. Crea unha función JavaScript que comprobe se é poxible axendar unha reunión dentro do horario laboral.

   A estrutura da función e do programa indícanse a continuación:

```
const inicioXornada = "07:30";
const finalXornada = "17:45";

function axendarReunion(horaInicioReunion, duracionEnMinutos) {
  // A función debe devolver true se a reunión ocorre dentro da xornada laboral
  // e false en caso contrario
}

// Comprobacións
console.assert(axendarReunion("7:00", 15) == false,
  'Fallo comprobando axendarReunión("7:00", 15) == false');
```

```
console.assert(axendarReunion("7:15", 30) == false,
  'Fallo comprobando axendarReunión("7:15", 30) == false');

console.assert(axendarReunion("7:30", 30) == true,
  'Fallo comprobando axendarReunión("7:30", 30) == true);

console.assert(axendarReunion("11:30", 60) == true,
  'Fallo comprobando axendarReunión("11:30", 60) == true);

console.assert(axendarReunion("17:00", 45) == true,
  'Fallo comprobando axendarReunión("17:00", 45) == true');

console.assert(axendarReunion("17:30", 30) == false,
  'Fallo comprobando axendarReunión("17:30", 30) == false');
```

8. Crea unha función chamada **buscarPatron(texto, patron)** que reciba como parámetros un texto e un patrón. A función debe devolver como resultado o número de veces que aparece o patrón no texto.

   Hai que implementar a función de forma manual e non utilizar as funcións proporcionadas pola linguaxe JavaScript.

   Non se deben distinguir maiúsculas de minúsculas.

   Un carácter pode formar parte de máis dun patrón encontrado.

   Exemplo: buscarPatron(“000111101000ABCHO”, “00”) debe devolver 4.

9. Crea unha función que reciba como parámetro unha cantidade enteira e faga o desglose do número de billetes e moedas necesarios para obtela. Debe usarse o número mínimo de billetes e moedas.

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