Obxectos

Introdución

O tipo de datos fundamental de JavaScript é o obxecto. Un obxecto é un dato complexo, é dicir, unha **colección non ordenada de propiedades.** Unha propiedade é un par "chave: valor", onde a chave é unha cadea chamada "nome da propiedade" e o valor pode ser calquera cousa (un tipo de dato primitivo ou un obxecto).

En JavaScript todos os valores que non son dun tipo primitivo (String, Boolean, Number, BigInt, undefined, null, Symbol) son considerados obxectos. Os tipos primitivos tamén se comportan como obxectos inmutables, é dicir, non se poden modificar as súas propiedades. Por outro lado, os tipos primitivos son tratados por valor e os obxectos son tratados por referencia.

Poden distinguirse tres categorías ou clases de obxectos:

- **Obxectos nativos**. Son obxectos definidos na propia especificación de JavaScript.
- **Obxectos de plataforma**. Son aqueles que pon a disposición a contorna de execución JavaScript, como o navegador ou Node.js.
- **Obxectos de usuario**. Son aqueles que se crean polo programa durante a execución do mesmo.

Observarase que JavaScript é unha linguaxe de programación baseada en obxectos, que permite traballar como se todo fosen obxectos, tendo en conta que os obxectos herdan propiedades e métodos doutros.

Propiedades e funcións do obxecto global

Un obxecto global é un obxecto que sempre existe no ámbito global. En JavaScript sempre hai un obxecto global definido, que dependerá do contexto de execución. Así, por exemplo, no contexto dun navegador web existirá o obxecto global Window.

Todas as propiedades do obxecto global poden ser accedidas directamente:

```
alert("Hello");
// o mesmo que
window.alert("Hello");
```

As variables declaradas con **var** e as **declaracións de funcións** no ámbito global son creadas como propiedades do obxecto global. Ao mesmo tempo, as propiedades do obxecto global son automaticamente engadidas ao ámbito global. Sen embargo, as declaracións de variables que usan **let** ou **const** nunca crean propiedades no obxecto global.

```
var gVar = 5;
console.log(window.gVar);
```

Existen **propiedades globais**, ou propiedades predefinidas do obxecto global, que son accesibles no ámbito global:

- Infinity: valor numérico que representa infinito.
- NaN: valor que representa Not-A-Number
- undefined: valor que representa o tipo primitivo **undefined**.
- globalThis: propiedade global que devolve o obxecto global de nivel superior. Historicamente, o acceso ao obxecto global requiría diferente sintaxe en función da contorna de execución (window, self, global, this). A propiedade **globalThis,** recentemente engadida á linguaxe, proporciona unha forma estándar de acceder ao obxecto global independentemente da contorna de execución.

Tamén existen **funcións globais** (funcións que poden ser executadas globalmente, sen necesidade de invocalas en ningún obxecto) e devolven o seu resultado directamente:

- eval(): recibe unha cadea, avalíaa como código JavaScript e devolve o resultado.

  console.log(eval('2+2'));

  NOTA: eval() é unha función perigosa. Se se executa eval() con unha cadea que puidese estar afectada por código malicioso, este sería executado na máquina.

- isFinite(): determina se o valor pasado como parámetro é un número finito.
- isNaN(): determina se o valor pasado é NaN ou non.
- parseFloat(): converte a cadea pasada como argumento a un número en punto flotante.
- parseInt(): converte a cadea pasada como argumento a un número enteiro na base especificada.
- encodeURI(): función que codifica unha URI substituído certos caracteres por secuencias de escape.
- encodeURIComponent(): función que codifica unha URI substituíndo certos caracteres por secuencias de escape.
- decodeURI(): descodifica a URI previamente codificada con encodeURI().
- decodeURIComponent(): descodifica a URI previamente codificada con encodeURIComponent().

Date

O obxecto Date permite representar un momento no tempo nun formato independente da plataforma. Date contén un número que representa os milisegundos dende o 1 de xaneiro de 1970.

Hai 5 formas básicas do construtor Date();

- new Date(): sen parámetros: o obxecto creado representa a data e hora no instante actual.

- new Date(value): o valor representa o número de milisegundos dende o 1 de xaneiro de 1970.
- new Date(dateString): unha cadea representando unha data no formato ISO 8601.
- new Date(dateObject): pasando como parámetro un obxecto Date existente, crea unha copia do mesmo.
- new Date(valores individuais de compoñentes): pasando como parámetro como mínimo o ano e o mes, o resto de parámetros que falte inicialízanse ao valor máis pequeno posible (1 para día e 0 para o resto de parámetros). Calquera valor que sexa maior do seu rango, fará incrementar a seguinte unidade. Por exemplo new Date(1990, 12, 1) devolverá o 1 de xaneiro de 1991

Exemplos:

```
const today = new Date()
const birthday = new Date(628021800000)           // passing epoch timestamp
const birthday = new Date('1995-12-17T03:24:00')   // This is ISO-8601-compliant
const birthday = new Date(1995, 11, 17)           // the month is 0-indexed
const birthday = new Date(1995, 11, 17, 3, 24, 0)
```

NOTA: cando se invoca o construtor **new Date()**, este devolve un obxecto Date. Se a data que se pasa como parámetro é incorrecta, devolverá un obxecto Date tal que ao transformalo a string dará “Invalid date”. Cando se invoca a función **Date()** (sen a palabra chave new) devolve unha cadea representando a data.

```
let data = new Date();
console.log(`new Date() = ${data}`);
console.log(`Tipo de dato de new Date() = ${typeof data}`);

let data2 = Date();
console.log(`Date() = ${data2}`);
console.log(`Tipo de dato de Date() = ${typeof data2}`);
```

Principais propiedades e métodos do obxecto Date:

- getDate() e setDate(): devolve/establece o día do mes da data especificada.
- getDay(): devolve o día da semana (0-6) dunha data. (0 é o domingo).
- getFullYear() e setFullYear(): devolve/establece o ano.
- getHours() e setHours(): devolve/establece a hora (0-23).
- getMinutes() e setMinutes(): devolve/establece os minutos (0-59).
- getMonth() e setMonth(): devolve/establece o mes (0-11).
- getSeconds() e setSeconds(): devolve/establece os segundos (0-59).
- toDateString(): devolve a porción de data máis lexible para as persoas. "Wed Jul 28 1993"
- toString(): devolve unha cadea representando a data.
- valueOf(): devolve o número de milisegundos dende o 1 de xaneiro de 1970.

Exercicios:

1. Mostra o día da semana (en letra) do 25 de xullo de 2000.
2. Mostra o día da semana (en letra) do 25 de xullo deste ano.
3. Calcula o número de días que pasaron dende o 25 de xullo de 2000 ata hoxe.
4. Crea unha función á que se lle pase un mes (1-12) e un ano e devolva o número de días dese mes.
5. Crea unha función á que se lle pase unha data e diga se é fin de semana.
6. Crea unha función que reciba unha data como parámetro e devolva o número de días que pasaron dende que comezou o ano.

Math

O obxecto Math é un obxecto predefinido de JavaScript que incorpora propiedades e métodos para constantes e funcións matemáticas.

O obxecto Math traballa co tipo **Number**, non con BigInt.

A diferencia doutros obxectos globais, Math non é un construtor. **Todas as súas propiedades e métodos son estáticas**.

As constantes están definidas coa precisión dos números reais en JavaScript.

Principais propiedades e métodos do obxecto Math:

- Math.E: constante de Euler e base dos logaritmos naturais.
- Math.PI: almacena o número pi.
- Math.abs(): devolve o valor absoluto de x.
- Math.ceil(): redondeo cara arriba.
- Math.floor(): redondeo cada abaixo.
- Math.max(): devolve o valor máis alto.
- Math.min(): devolve o valor máis baixo.
- Math.pow(): realiza a operación de exponenciación (x^y).
- Math.random(): devolve un número aleatorio entre 0 e 1.
- Math.round(): redondea un número ao enteiro máis próximo.
- Math.sqrt(): devolve a raíz cadrada.
- Math.trunc(): devolve a parte enteira do número, eliminando a decimal.

Exercicios:

1. Crea unha función á que se lle pase como parámetro o número de minutos e devolva un string indicando a súa equivalencia en horas e minutos.
2. Crea unha función que dado o radio dun círculo, devolva a súa área. E fai outra función que reciba o radio e devolva o perímetro do círculo. Mostra por consola o resultado das funcións usando dúas cifras decimais.

Strings

Ademais do tipo de dato String, JavaScript define o obxecto String, que ofrece un conxunto de propiedades e métodos útiles á hora de traballar con cadeas de texto.

Diferentes formas de crear Strings:

```
const string1 = "A string primitive";
const string2 = `Yet another string primitive`;
const string3 = String("Another string");
const string4 = new String("A string object");
console.log("typeof string1 = " + typeof string1);
console.log("typeof string2 = " + typeof string2);
console.log("typeof string3 = " + typeof string3);
console.log("typeof string4 = " + typeof string4);
```

Fixarse que JavaScript distingue entre obxectos String e o tipo primitivo string (o mesmo pasa con Boolean e Number). Para crear un obxecto String é necesario usar o construtor **new String()**.

Cando sexa necesario invocar un método ou unha propiedade nunha cadea de tipo primitivo, JavaScript automaticamente fará referencia ao método ou propiedade do obxecto String.

NOTA: rara vez se crean cadeas usando new String().

Existen situacións na que se producen resultados diferentes ao usar tipos primitivos ou obxectos. Por exemplo, cando se utiliza a función eval(), o tipo primitivo string e o obxecto String dan resultados diferentes:

```
const s1 = '2 + 2';                 // creates a string primitive
const s2 = new String('2 + 2');  // creates a String object
console.log(eval(s1));              // returns the number 4
console.log(eval(s2));              // returns the string "2 + 2"
```

Concatenar cadeas

Para unir varias cadeas pode usarse o operador **+** ou usar os **patróns de cadeas** (**template literals** ou **string templates**).

Os **patróns de cadeas** son cadeas literais delimitados polo símbolo ` (acento grave). Funcionan como cadeas normais, excepto que permiten incluír variables usando o envoltorio ${ }.

```
const nome = 'Ana';
const greeting = `Hello, ${nome}`;
console.log(greeting); // 'Hello, Ana'

const hello = 'Hello';
const question = 'how are you?';
const joined = `${hello}, ${question}`;      // const joined = hello + ', ' + question
console.log(joined); // 'Hello, how are you?'
```

Estes patróns de cadeas tamén permiten crear cadeas de **múltiples liñas** e usar as aspas dentro da cadea.

```
const output = `I like the ‘song’.
I gave it a score of 90%.`;
console.log(output);  // I like the ‘song’.
                      // I gave it a score of 90%.
```

Para facer algo equivalente usando unha cadea normal, hai que usar o carácter especial de salto de liña (**\n**):

```
const output = 'I like the song.\nI gave it a score of 90%.';
console.log(output);  // I like the song.
                      // I gave it a score of 90%.
```

Tamén se poden substituír expresións:

```
const a = 5;
const b = 10;
// Usando cadeas normais
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
// 'Fifteen is 15 and
// not 20.'

// Sintaxe Usando literais
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

Traballando con strings

Os métodos de manipulación de cadeas non modifican o obxecto actual, senón que devolven o obxecto resultante de aplicar a modificación.

Principais propiedades e métodos do obxecto String:

- A propiedade length devolve a lonxitude da cadea.

  NOTA: length non inclúe parénteses () porque é unha propiedade (un valor que xa foi calculado), sen embargo .toLowerCase() é un método que require parénteses () porque é unha acción que se realiza sobre a cadea.

- toLowerCase() e toUpperCase() devolven a cadea convertida en minúsculas ou maiúsculas.

  NOTA: cando se recolle información tecleada polas persoas usuarias é boa idea pasar o texto a minúsculas e traballar todo en minúsculas.

- charAt(posicion): devolve o carácter na posición indicada. O primeiro carácter ocupa a posición 0. Tamén é posible tratar as cadeas como un array, polo que se pode acceder aos caracteres usando corchetes ('cat'[1] // gives value "a").
- concat(str1, str2, …): concatena os argumentos á cadea dende a que se invoca.
- endsWith(): comproba se a cadea remata coa cadea especificada.
- includes(cadea): comprobar se unha cadea está contida noutra.
- indexOf(cadea): busca a cadea pasada como parámetro e devolve o índice da primeira ocorrencia.
- lastIndexOf(cadea): busca a cadea pasada como parámetro e devolve o índice da última ocorrencia.
- match(patrón): úsase para buscar unha expresión regular.
- matchAll(patrón): devolve un iterador con todas as coincidencias encontradas.
- padStart(): método que completa a cadea actual coa cadea pasada como parámetro ata chegar ao número de caracteres indicados.
- repeat(): constrúe e devolve unha nova cadea que contén o número especificado de copias da cadea que invoca a función, concatenadas entre sí.
- replace(patron, substituto): devolve unha nova cadea resultado de substituír a primeira ocorrencia do patrón polo argumento indicado como segundo parámetro.
- replaceAll(patron, substituto): devolve unha nova cadea resultado de substituír a todas as ocorrencias do patrón pola cadea especificada de substitución.
- search(patrón): busca unha expresión regular na cadea.
- slice(): extraer unha subcadea doutra.
- split(separador): devolve un array de cadeas resultado de dividir a cadea orixinal no punto onde se encontran os separadores.
- startsWith(cadea): comproba se a cadea empeza coa cadea especificada.
- substring(inicio, fin): devolve a parte da cadea entre o inicio e fin. Exemplo: **'Mozilla'.substring(1,3)** devolve “oz”.
- trim(): elimina os espazos en branco do principio e fin da cadea.

NOTA: en JavaScript é posible encadear métodos: **"abc".toUpperCase().charAt(0)**

Exercicios:

1. Crea unha función á que se lle pase unha cadea e devolva a cadea en sentido inverso.

   console.log(reverseString("I am a string")) // gnirts a ma I

2. Crea unha función á que se lle pase unha cadea e un array de caracteres e devolva a cadea orixinal eliminando os caracteres do array.

   console.log(removeCharacters("I am an example string", ["a", "x"]));
   // // I m n emple string

3. Crea unha función á que se lle pase unha cadea e devolva o carácter máis repetido.

   console.log(caracterMaisRepetido("abcddefg")) // d

4. Crea unha función á que se lle pase unha cadea der números e devolva unha cadea da mesma lonxitude formada por * e as últimas 4 cifras do parámetro de entrada.

   console.log(enmascarar(“1234123412347777”)); // **********7777**

5. Escribe o código necesario para procesar unha cadea con información de voos como a do exemplo e mostrar a información por consola formateada como aparece na imaxe:

   const flightsInfo =
   "_Delayed_Departure;scq93766109;bio2133758440;11:25+_Arrival;bio0943384722;scq93766109;11:45+_Delayed_Arrival;svq7439299980;scq93766109;12:05+_Departure;scq93766109;svq2323639855;12:30";

   Fixarse que a información mostrada por consola está aliñada pola dereita.

```
  Delayed Departure SCQ BIO (11h25)
            Arrival BIO SCQ (11h45)
Delayed Arrival SVQ SCQ (12h05)
          Departure SCQ SVQ (12h30)
```

Arrays

O obxecto Array permite almacenar un conxunto de múltiples valores baixo unha mesma variable.

Os Arrays non teñen un tamaño fixo, polo que poden engadirse elementos en calquera momento, resultando no redimensionado do array. Ademais poden conter elementos de diferentes tipos.

Para acceder aos elementos do array utilízanse como índices números enteiros positivos. O primeiro elemento do array ten o índice 0.

Poden crearse arrays de diferentes formas:

```
const fruits = ['Apple', 'Banana'];    //notación literal - Recomendada

const fruits2 = new Array(2);  // Usar un número no construtor, indica a lonxitude do array
console.log(fruits2.length); // 2
console.log(fruits2[0]);     // undefined

const fruits3 = new Array('Apple', 'Banana');

// poden conter elementos de diferente tipo
const varios = ['Apple', 3];
varios[3] = 'Banana',
console.log(varios);
```

Traballando con Arrays

JavaScript inclúe propiedades e métodos predefinidos para traballar con arrays.

Os principais métodos para traballar con arrays:

- length: establece ou devolve o número de elementos dun array.
- pop(): elimina o último elemento do array. Ademais, devolve o elemento eliminado.
- push(): engade elementos ao final do array. A función push() devolve como valor a nova lonxitude do array, unha vez engadido o elemento.
- shift(): elimina o primeiro elemento do array e devolve o elemento eliminado.
- unshift(): engade un elemento ao comezo do array. Ao igual que a función push(), a función unshift() devolve a lonxitude do novo array.
- includes(): comproba se un elemento está contido no array. Esta función utiliza a igualdade estrita para comprobar se o elemento está contido no array e devolve **true** en caso afirmativo ou **false** en caso negativo.

Outros métodos para traballar con arrays:

- at(index): devolve o elemento co índice indicado. Índices negativos contan dende o final do array. Tamén se pode acceder a un elemento usando corchetes (array[0]).
- concat(): úsase para fusionar dous ou máis arrays. Este método non modifica os arrays orixinais, crea un novo array e devólveo.
- entries(): devolve un novo obxecto *Array Iterator* que contén pares chave/valor en cada elemento do array.

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

for (const item of letras.entries())
   console.log(`${item[0]}: ${item[1]}`);
```

```
// desestruturando o array
for (const [index, element] of letras.entries())
  console.log(`${index}: ${element}`);
```

- every(): comproba se todos os elementos do array pasan o test implementado na función pasada como parámetro. Devolve un valor booleano.
- indexOf(): devolve o primeiro índice no que se atopa o elemento a buscar. Se o elemento a buscar non está no array, devolve **-1.**
- join(): devolve unha **cadea** resultado de concatenar todos os elementos no array separados por comas ou usando o separador especificado.
- reverse(): inverte a orde dos elementos do array, o primeiro elemento pasa a ser o último. Esta función devolve unha referencia ao mesmo array (**modifica o array orixinal**).
- slice(): devolve unha copia dunha porción do array. Non modifica o array orixinal.
- some(): comproba se algún elemento do array pasa o test implementado na función proporcionada. Devolve un valor booleano.
- sort(): ordena os elementos do array e devolve unha referencia ao mesmo array, agora ordenado. Por defecto, os elementos do array son convertidos a string e ordenados de forma alfabética (tamén se o array contén números).
- splice(): **modifica o contido dun array** eliminando ou substituíndo elementos existentes e/ou agregando novos elementos. Devolve un array que contén os elementos eliminados.

Exercicios:

1. Garda nun array a lista de froitas: peras, mazás, kiwis, plátanos e mandarinas. Fai os seguintes apartados con **splice**:
    a. Elimina as mazás.
    b. Engade detrás dos plátanos, laranxas e sandía.
    c. Quita os kiwis e pon no seu lugar cereixas e nésperas.

    Despois de realizar cada operación, mostra por pantalla o array coa lista de froitas, onde os elementos estean separados por unha coma e espazo. Por exemplo, inicialmente o array debe mostrarse como “peras, mazás, kiwis, plátanos, mandarinas”.

2. Fai unha función que ordene as notas dun array pasado como parámetro. Por exemplo, se se pasa o array [4,8,3,10,5] debe devolver [3,4,5,8,10]. Debes utilizar a función sort e pasarlle como parámetro unha función que ti definas que serva para realizar a comparación de elementos.
3. Dado un array cos días da semana, indica se algún comeza por ‘s’. Comproba tamén se todos acaban en ‘s’.
4. Crea unha función á que se lle pase un texto e devolva o mesmo texto coa primeira letra de cada palabra en maiúsculas e o resto en minúsculas.

Algunhas das funcións predefinidas para traballar con arrays, permiten pasar como parámetro outra función que personaliza a operación a realizar. Para pasar unha función como parámetro, hai que definila como unha expresión función, unha función anónima, ou unha función frecha.

```
const letters = ["a", "b", "c", "d"];

// Con funcións por expresión
const f = function () {
  console.log("Un elemento.");
};
letters.forEach(f);

// Con funcións anónimas
letters.forEach(function () {
  console.log("Un elemento.");
});

// Con funcións frecha
letters.forEach(() => console.log("Un elemento."));
```

A función que se pasa como parámetro, á súa vez, tamén pode ter 1, 2 ou 3 parámetros:

- o primeiro parámetro será o elemento do array.
- o segundo parámetro será a posición (índice) no array.
- o terceiro parámetro será o array en cuestión.

```
const letters = ["a", "b", "c", "d"];

letters.forEach((element) => console.log(element));                // Devolve 'a' / 'b' / 'c' / 'd'
letters.forEach((element, index) => console.log(element, index));  // Devolve 'a' 0 / 'b' 1 / 'c' 2 / 'd' 3
letters.forEach((element, index, array) => console.log(array[index])); // Devuelve 'a' / 'b' / 'c' / 'd'
```

Esta forma de programar segue o **paradigma da programación funcional**, xa que se centra máis no que debe facer unha función que en como debe facelo. Para comparalo co paradigma imperativo, observar o seguinte exemplo, onde se devolve un array coas notas maiores ou iguais a 5 do array de entrada. A primeira versión usa un paradigma de programación imperativo, mentres que a segunda versión utiliza a programación funcional.

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let aprobados = [];
for (let i = 0; i < arrayNotas.length; i++) {
  let nota = arrayNotas[i];
  if (nota >= 5) {
    aprobados.push(nota);
  }
}
console.log(aprobados);
```

```
let arrayNotas = [5.2, 3.9, 6, 9.75, 7.5, 3];
let aprobados = arrayNotas.filter((nota) => nota >= 5);

console.log(aprobados);
```

Algunhas das funcións para traballar con arrays reciben outra función como parámetro:

- filter(): filtra os elementos do array e devolve un array con aqueles que pasan un test definido pola función proporcionada.
- find(): devolve o primeiro elemento do array que cumpre a condición indicada. Se non hai ningún valor que cumpra a condición, devolve undefined.
- findIndex(): devolve o índice do primeiro elemento do array que cumpre a condición indicada. Se ningún elemento cumpre a condición, devolve -1.
- forEach(): executa unha función proporcionada en cada elemento do array. O método forEach recorrerá sempre todo o array, non sendo posible usar sentencias como break.
- map(): crea un novo array cos resultados de aplicar unha función a cada elemento do array.
- reduce(): devolve un valor calculado a partir dos elementos do array. Os parámetros de reduce son:
    - Unha función que se pode ter ata 4 parámetros:
        - previousValue: o valor resultado da anterior chamada á función. Na primeira chamada, terá o valor *initialValue*, se se especifica, en caso contrario terá o valor array[0].
        - currentValue: o valor do elemento actual. Na primeira chamada terá o valor array[0] se se especifica o *initialValue*, en caso contrario terá o valor array[1].
        - currentIndex: índice do elemento actual do array. Na primeira chamada valerá 0 se se especifica o *initialValue*, en caso contrario valerá 1.
        - array: o array de elementos.
    - initialValue (opcional): é o valor ao que se inicializa *previousValue* a primeira vez que se chama á función. Se *initialValue* está inicializado, *currentValue* inicializarase a array[0]. Se *initialValue* non está especificado, *previousValue* inicialízase a array[0] e *currentValue* inicialízase a array[1].

```
const array = [15, 16, 17, 18, 19];

function reducer(previousValue, currentValue, index) {
  const returns = previousValue + currentValue;
  console.log(
    `previousValue: ${previousValue}, currentValue: ${currentValue}, index: ${index},
returns: ${returns}`
  );
  return returns;
}
```

```
let resultado = array.reduce(reducer);
console.log(`Resultado = ${resultado}`);

console.log("Usando un valor inicial");
resultado = array.reduce(reducer, 0);
console.log(`Resultado = ${resultado}`);

console.log("Usando unha función frecha");
resultado = array.reduce(
  (previousValue, currentValue, index) => previousValue + currentValue,
  0);
console.log(`Resultado = ${resultado}`);
```

Exercicios:

1. Dado un array cos días da semana, obtén un array cos días que empecen por “m”.
2. Dado un array cos días da semana, obtén o primeiro día que empeza por “m”.
3. Dado un array cos días da semana, obtén a posición no array do primeiro día que empeza por “m”.
4. Dado un array cos días da semana, devolve outro array cos días en maiúsculas.
5. Dado un array de números, obtén o valor máis alto. (Usa algunha das funcións para traballar con arrays).

Rest e Spread

A sintaxe dos parámetros rest permite que unha función acepte un número indeterminado de argumentos nunha variable array. É dicir, os argumentos pasados á función englóbanse dentro dunha variable de tipo array. Só o último parámetro da definición da función pode ser un parámetro rest.

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);         // a, one
  console.log("b", b);         // b, two
  // manyMoreArgs, ["three", "four", "five", "six"]
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");
```

Recordar que os parámetros **rest** son arrays reais, mentres que o obxecto arguments non o é. Isto implica que se poden aplicar métodos de arrays ao parámetro rest e non a arguments.

A sintaxe **spread** (...) permite que un **iterable** (array, string, map ou set) se expanda onde se precisan varios argumentos (chamadas a funcións) ou elementos (notación literal de arrays). É dicir, saca os elementos do obxecto iterable e trátaos individualmente.

A sintaxe spread é idéntica á sintaxe rest, sen embargo fan cousas opostas. Mentres que spread expande o array nos seus elementos, rest recolle os múltiples argumentos e condénsaos nun único de tipo array. A sintaxe spread só pode usarse nos argumentos dunha función ou na construción dun array.

```
const arr = [7, 8, 9];
//  const newArray = [1, 2, arr[0], arr[1], arr[2]];
// equivalente ao anterior
const newArray = [1, 2, ...arr];
console.log(newArray);
```

Tamén é útil usar spread cando é necesario **pasar múltiples elementos a unha función**.

```
function sum(x, y, z) {
  return x + y + z;
}

const numbers = [1, 2, 3];
console.log(sum(...numbers));

function myFunction(v, w, x, y, z) {}
const args = [0, 1];
myFunction(-1, ...args, 2, ...[3]);
```

A sintaxe spread tamén pode usarse para **crear novos arrays**.

```
const parts = ['shoulders', 'knees'];
const lyrics = ['head', ...parts, 'and', 'toes'];    //  ["head", "shoulders", "knees", "and", "toes"]
```

A sintaxe spread tamén é útil para **copiar arrays**:

```
const fruits = ['Strawberry', 'Mango'];
// Create a copy using spread syntax.
const fruitsCopy = [...fruits];      // ["Strawberry", "Mango"]
```

Desestruturación de arrays

A desestruturación é unha característica que permite desempaquetar os valores dun array ou obxecto en variables separadas. Noutras palabras, a desestruturación permite romper unha estrutura de datos complexa noutras estruturas máis pequenas, como unha variable.

A desestruturación de arrays permite recuperar os elementos do array e almacenalos de forma fácil en variables. Exemplo:

```
let a, b, c, rest;
[a, b] = [10, 20];
console.log(`a = ${a}`); // expected output: 10
console.log(`b = ${b}`); // expected output: 20

[a, , b, , c] = [10, 20, 30, 40, 50];
console.log(`a = ${a}, b = ${b}, c = ${c}`);

// valor por defecto
const [d = 1] = []; // d is 1
console.log(`d = ${d}`);

// intercambio de variables
let x = 1;
let y = 3;
[x, y] = [y, x];
console.log(`x = ${x}, y = ${y}`);

const arr = ["a", "b", "c"];
[arr[2], arr[1]] = [arr[1], arr[2]];
console.log(arr);
```

Pode rematarse a desestruturación do array con unha propiedade …rest. A rest asignaránselle o resto de propiedades do array nun novo obxecto array.

```
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(rest); // expected output: Array [30,40,50]
```

NOTA: A propiedade rest debe ser a última.

Copiar arrays

Cando se copia unha variable de tipo primitivo ou se pasa como parámetro a unha función, faise unha copia da mesma e calquera modificación da variable copia, non afectará á variable orixinal:

```
let a = 54;
let b = a; // a = 54 b = 54
b = 86; // a = 54 b = 86
```

Sen embargo, ao copiar obxectos (e os arrays son un tipo de obxectos) a nova variable apunta ao mesmo enderezo de memoria que a antiga, polo que os datos de ambas son compartidos:

```
let a = [54, 23, 12];
let b = a; // a = [54, 23, 12] b = [54, 23, 12]
b[0] = 3; // a = [3, 23, 12] b = [3, 23, 12]

let data1 = new Date("2018-09-23");
let data2 = data1; // data1 = '2018-09-23'   data2 = '2018-09-23'
data2.setFullYear(1999); // data1 = '1999-09-23'   data2 = '1999-09-23'
```

Para obter unha copia dun array que sexa independente do array orixinal, poden usarse varios métodos:

```
const fruits = ['Strawberry', 'Mango'];

// Create a copy using spread syntax.
const fruitsCopy = [...fruits];     // ["Strawberry", "Mango"]

// Create a copy using the from() method.
const fruitsCopy2 = Array.from(fruits);     // ["Strawberry", "Mango"]

// Create a copy using the slice() method.
const fruitsCopy3 = fruits.slice();     // ["Strawberry", "Mango"]
```

Os exemplos anteriores para copiar arrays funcionan, aínda que teñen un problema. Son **copias superficiais** (*shallow copy*), que quere dicir que só se fai unha copia do primeiro nivel dos elementos do iterable.

```
let a, copia;
a = [[1, 2], 3, 4];
copia = [...a];
console.log(`a = ${a}, copia = ${copia}`);

// ao modificar un elemento
copia[0][0] = 5;

// tamén se modifica na copia
console.log(`a = ${a}, copia = ${copia}`);
```

JSON (JavaScript Object Notation) é un formato para representar obxectos en forma de cadea de caracteres de forma que sexa fácil de ler para as persoas. Inicialmente foi creado para JavaScript, aínda que hoxe en día hai librerías para moitas linguaxes, polo que é fácil usar JSON para intercambiar información cando o cliente usa JavaScript e o servidor está escrito en Ruby/PHP/Java/…

JavaScript proporciona os métodos:

- JSON.stringify(): converte obxectos nunha cadea JSON.
- JSON.parse(): converte a cadea JSON a un obxecto.

Estes métodos poden usarse para facer unha copia profunda dun array. É dicir, pode usarse JSON.stringify() para converter o array a unha cadea JSON e despois usar JSON.parse() para converter a cadea de novo a array.

```
const fruits = ['Strawberry', 'Mango'];

const fruitsDeepCopy = JSON.parse(JSON.stringify(fruits));
```

NOTA: undefined, Function e Symbol non son valores válidos para un JSON cando se usa stringify. Se se encontra un destes valores, ou é omitido ou transformado a null.

Exercicios:

1. Imaxinar que se recolle a seguinte información relativa a un xogo dun servidor web:

```
const players = [
  [
    "Neuer",
    "Pavard",
    "Martinez",
    "Alaba",
    "Davies",
    "Kimmich",
    "Goretzka",
    "Coman",
    "Muller",
    "Gnarby",
    "Lewandowski",
  ],
  [
    "Burki",
    "Schulz",
    "Hummels",
    "Akanji",
    "Hakimi",
    "Weigl",
    "Witsel",
    "Hazard",
    "Brandt",
    "Sancho",
    "Gotze",
  ],
];
```

Utilizando o contido aprendido sobre arrays, crea unha única sentencia para cada unha das seguintes instrucións:

a. Crea as variables **players1, players2** que conteña un array cos xogadores de cada equipo.
b. O primeiro xogador do array é o porteiro e o resto son xogadores de campo. Crea unha variable chamada **gk** que conteña o porteiro do primeiro equipo e unha variable de tipo array chamada **fieldPlayers** que conteña o resto de xogadores do equipo.
c. Crea un array **allPlayers** que conteña os xogadores dos dous equipos.
d. O primeiro equipo substituiu os xogadores iniciais por 'Thiago', 'Coutinho', 'Periscic'. Crea unha nova varible de tipo array chamada **players1Final** que conteña os xogadores iniciais e tamén os 4 novos.

2. Dado un array con nomes de variables formados por dúas palabras separadas por "_", mostra por consola os nomes das variables en formato camelCase. Por exemplo, se o array de entrada é ["first_name", " last_NAME"], deberase mostrar por consola "firtsName" e "lastName".

Obxectos

Un obxecto é unha estrutura de datos que permite almacenar información. Ata agora, a estrutura de datos que permitía almacenar valores relacionados na mesma variable era o array:

```
// array onde os valores están escritos en filas
const ereaArray = [
  'Erea',
  'Pereiro',
  2037 - 1991,
  'profesora',
  ['Navia', 'Comba', 'Paz']
];
```

Observando o array anterior, vese que almacena datos relativos a unha persoa e pódese deducir que o primeiro valor se corresponde co nome, o segundo co apelido, o terceiro coa idade, o cuarto coa profesión e o quinto coas amigas. Sen embargo, utilizando arrays, non hai ningunha forma de acceder aos valores que almacena mediante un nome, senón que sempre haberá que acceder a través do índice da posición que ocupa o elemento.

Os **obxectos** proporcionan unha estrutura de datos que permite almacenar pares propiedade-valor, facendo posible acceder aos valores almacenados usando o nome dunha propiedade. Unha **propiedade** dun obxecto pode definirse como unha variable asociada ao obxecto.

Os obxectos de JavaScript poden ser considerados como unha colección **propiedades**. As propiedades son equivalentes aos pares chave-valor. As chaves poden ser cadeas ou

símbolos, mentres que os valores poden ser de calquera tipo, incluso outros obxectos, permitindo crear estruturas de datos complexas.

A continuación móstrase un obxecto que almacena información de forma equivalente ao array mostrado anteriormente. Fixarse que neste caso úsanse **chaves { }**:

```
const ereaObxecto = {
  nome: 'Erea',
  apelido: 'Pereiro',
  idade: 2037 - 1991,
  profesion: 'profesora',
  amigas: ['Navia', 'Comba', 'Paz']
};
```

Arrays e obxectos son estruturas de datos diferentes, cada unha coas súas características. Dado que o acceso aos elementos do array se fai a través da posición que ocupan os elementos, os **arrays** son apropiados para almacenar **datos ordenados**, mentres que os obxectos son máis apropiados para almacenar valores que non teñen unha orde específica.

Hai dúas formas básicas de **crear un obxecto baleiro**:

```
const obx = new Object();
const obx = { };   // sintaxe literal
```

As instrucións anteriores son equivalentes; o segundo exemplo chámase sintaxe literal de obxecto e é máis adecuado usala. Esta sintaxe tamén é o núcleo do formato JSON e é preferible en todo momento.

A **sintaxe literal** de obxecto pode usarse para inicializar un obxecto na súa totalidade:

```
const obx = {
  firstName: 'Carol',
  details: {
    mobile: '611223344',
    email: 'email@gmail.com'
  }
};
```

Se o valor dunha propiedade é o valor dunha variable que se chama como ela, dende ES2015 pode usarse a seguinte sintaxe:

```
let firstName =  'Erea';
const obx = {
  firstName,
  …
};
```

Unha vez creado o obxecto, pode accederse ás súas propiedades de dúas formas:

```
// usando un punto
obx.firstName = 'Ana';
// usando corchetes
obx['firstName'] = 'Ana';
```

En xeral prefírese usar a notación con punto, aínda que hai algunhas situacións nas que hai que usar corchetes. Por exemplo, se o nome dunha propiedade está almacenado nunha variable, non se pode usar a notación con punto e si os corchetes. Tamén se pode usar uha expresión para calcular o nome da propiedade e usar a notación de corchetes. Exemplo:

```
const propiedade = 'firstNaxme';
console.log(obx[propiedade]);

const nameKey = 'Name';
console.log(obx['first' + nameKey]);
```

O acceso ás propiedades pode **encadearse**:

```
obx.details.mobile; // orange
obx['details']['email']; // 12
```

Cando se inicializa un obxecto, tamén se inicializan un conxunto de propiedades. Posteriormente poden engadirse ou eliminarse máis propiedades:

```
obx.location = 'Santiago';
obx['school'] = 'IES San Clemente';
// comprobación
console.log(obx);
delete obx.location;
console.log(obx);
```

Se se intenta acceder a unha **propiedade que non existe**, non se produce un erro, senón que se devolve ***undefined***. Sen embargo, cando se intenta acceder a propiedades de algo que non é un obxecto si que xera un erro.

```
console.log(obx.surname);  // undefined
console.log(obx.surname.description);  // erro

// Para evitar este erro, debe usarse o operador de encadeamento opcional (?.)
console.log(obx.surname?.description);
```

Cando se teclea na consola de JavaScript o nome dun obxecto seguido de “.” aparecen a lista de propiedades dispoñibles para o obxecto. Sen embargo, aparecen máis propiedades que as definidas explicitamente. ¿Por que?. Porque todo obxecto en JavaScript herda dun

obxecto, que é denominado **prototipo**. A referencia ao prototipo almacénase nunha propiedade especial, que aínda que non ten un nome estándar, soe usarse _proto_. Á súa vez, o obxecto prototipo tamén ten unha referencia a outro obxecto que é o seu prototipo, creando unha cadea de prototipos que remata cando o prototipo dun obxecto é null.

NOTA: en realidade **_proto_** é un getter/setter da propiedade [[Prototype]].

Cando se accede a unha propiedade dun obxecto, se non se encontra no propio obxecto, búscase a propiedade na cadea de prototipos. Isto é o que se chama herdanza de prototipos. Se finalmente non se encontra, devólvese "undefined".

Poden percorrerse as propiedades dun obxecto cun bucle **for…in**. O bucle for…in permite iterar sobre as propiedades **enumerables** dun obxecto, incluídas as herdadas.

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

for (const propiedade in obxecto) {
  console.log(`obxecto.${propiedade} = ${obxecto[propiedade]}`);
}
```

As propiedades dos obxectos en JavaScript poden clasificarse por tres factores:

- Enumerable ou non enumerable. As propiedades enumerables teñen un atributo interno activado a true. Por defecto, todas as propiedades que se crean manualmente nun obxecto, son enumerables. As propiedades definidas con Object.defineProperty son **non** enumerables por defecto.
- String ou símbolo.
- Propiedade propia ou herdada da cadea de prototipos.

Pode iterarse polas propiedades e valores dun obxecto de diferentes formas:

- for…in: recorre todas as propiedades enumerables de tipo string (non símbolos) dun obxecto e tamén as herdadas.
- Object.keys(myObj): devolve un array cos nomes das propiedades enumerables do obxecto.
- Object.values(myObj): devolve un array cos valores das propiedades enumerables do obxecto.
- Object.entries(myObj): devolve un array de pares [chave, valor] coas propiedades enumerables do obxecto.

É posible copiar obxectos ou fusionalos utilizando a sintaxe spread:

```
const obj1 = { foo: 'bar', x: 42 };
const obj2 = { foo: 'baz', y: 13 };

const clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }
```

```
const mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

Métodos

Unha propiedade dun obxecto pode ser unha función, ou método. Cando se define unha función usando unha expresión é posible almacenar a función nunha variable. Isto, aplicado a obxectos, permite crear un par chave - valor no cal o valor sexa unha función. Exemplo:

```
const ereaObxecto = {
  nome: 'Erea',
  apelido: 'Pereiro',
  profesion: 'profesora',
  calcAge: function (year) {
    return year - 1991;
  }
};

console.log(ereaObxecto.calcAge(2037));

// Tamén é posible acceder ao método usando corchetes
console.log(ereaObxecto['calcAge'](2037));
```

Dende ES2015 é posible usar unha sintaxe máis sinxela para os métodos:

```
const ereaObxecto = {
    …,
    calcAge(year) {
      return year - 1991;
    }
  };
```

Dentro dun método, a palabra chave **this** fai referencia ao obxecto actual onde se está escribindo o código. No seguinte exemplo **this** é equivalente a ereaObxecto e móstrase como acceder a unha propiedade do obxecto (birthYear) dende o método, usando a palabra chave **this.**

```
const ereaObxecto = {
  nome: 'Erea',
  apelido: 'Pereiro',
  profesion: 'profesora',
  birthYear: 1991,
  calcAge: function (year) {
    return year - this.birthYear;
  },
};
```

```
console.log(ereaObxecto.calcAge(2037));
```

Desestruturación de obxectos

Ao igual que os arrays, os obxectos tamén poden desestruturarse, para romper a estrutura do obxecto noutras máis pequenas.

Os obxectos desestrutúranse usando { } e as variables deben levar o mesmo nome que as propiedades do obxecto a desestruturar:

```
const obj = { a: 1, b: 2 };
const { a, b } = obj;
// equivalente a
// const a = obj.a;
// const b = obj.b;

// Se as variables xa están declaradas, é necesario poñer a sentencia entre ( )
let a, b;
{ a, b } = obj;   // Erro de sintaxe
({ a, b } = obj);
```

Tamén é posible **cambiar o nome das variables** cando se quere que sexan diferentes aos nomes das propiedades. O seguinte exemplo colle a propiedade p do obxecto o e asígnaa á variable local chamada foo.

```
const o = { p: 42, q: true };
const { p: foo, q: bar } = o;

console.log(foo); // 42
console.log(bar); // true
```

Cada propiedade pode ter un **valor por defecto** que se aplicará cando a propiedade non está presente ou ten un valor *undefined*. Este valor por defecto non se usa no caso de que a propiedade sexa *null*.

```
const { b = 2 } = { b: undefined }; // b is 2
const { c = 2 } = { c: null }; // c is null
```

Pódese facer a desestruturación de obxectos e cambiar o nome das variables ao mesmo tempo que se proporcionan **valores por defecto**:

```
const { a: aa = 10, b: bb = 5 } = { a: 3 };
console.log(aa); // 3
console.log(bb); // 5
```

Desestruturar obxectos aniñados:

```
const user = {
  id: 42,
  displayName: 'jdoe',
  fullName: {
    firstName: 'John',
    lastName: 'Doe',
  },
};

const { displayName, fullName: { firstName: name } } = user;
console.log(displayName, name);
```

Desempaquetar propiedades de obxectos pasados como parámetros a unha función:

```
const user = {
  id: 42,
  displayName: 'jdoe',
  fullName: {
    firstName: 'John',
    lastName: 'Doe',
  },
};

function userId({ id }) {
  return id;
}

console.log(userId(user)); // 42
```

Exercicios:

1. Crea un obxecto chamado ***television*** coas propiedades marca, categoría (televisores), unidades (4), prezo (354.99) e un método chamado importe que devolva o prezo total das unidades (unidades x prezo).

2. Imaxinar que se recolle a seguinte información relativa a un xogo dun servidor:

   ```
   const game = {
      odds: {
        team1: 1.33,
        x: 3.25,
        team2: 6.5,
    }
   };
   ```

Utilizando a desestruturación de obxectos crea unha variable para cada unha das propiedades do obxecto **odds**. Estas variables deben levar os nomes “team1”, “draw” e “team2”.

3. Dado o seguinte obxecto:

```
const game = {
  scored: ["Lewandowski", "Gnarby", "Lewandowski", "Hummels"]
};
```

   a. Recorre o array **game.scored** e mostra por pantalla información do xogador que marcou e o número de gol marcado. Exemplo: “Gol 1: Lewandowski”.

   b. Crea un novo obxecto chamado **scorers** que conteña o nome dos xogadores que marcaron e o número de goles que marcaron como valor. Neste exemplo sería algo así: **{Lewandowski: 2, Gnarby: 1, Hummels: 1}**

4. Dada a seguinte información:

```
const inventors = [
  { first: "Albert", last: "Einstein", year: 1879, passed: 1955 },
  { first: "Isaac", last: "Newton", year: 1643, passed: 1727 },
  { first: "Galileo", last: "Galilei", year: 1564, passed: 1642 },
  { first: "Marie", last: "Curie", year: 1867, passed: 1934 },
  { first: "Johannes", last: "Kepler", year: 1571, passed: 1630 },
  { first: "Nicolaus", last: "Copernicus", year: 1473, passed: 1543 },
  { first: "Max", last: "Planck", year: 1858, passed: 1947 },
  { first: "Katherine", last: "Blodgett", year: 1898, passed: 1979 },
  { first: "Ada", last: "Lovelace", year: 1815, passed: 1852 },
  { first: "Sarah E.", last: "Goode", year: 1855, passed: 1905 },
  { first: "Lise", last: "Meitner", year: 1878, passed: 1968 },
  { first: "Hanna", last: "Hammarström", year: 1829, passed: 1909 },
];
```

   a. Filtra o array de inventores e crea un array só cos inventores que naceron no século XVI.
   b. Crea un array co nome completo dos inventores. Por exemplo: ["Albert Einstein", "Isaac Newton", ...]
   c. Unha vez obtido o array co nome completo dos inventores do exercicio anterior, ordénao alfabeticamente polo apelido
   d. Ordena o array de inventores alfabeticamente polo apelido
   e. Ordena o array de inventores pola data de nacemento
   f. Calcula a suma dos anos que viviron todos os inventores.
   g. Ordena os inventores polos anos que viviron, primeiro o máis lonxevo

5. Dada a seguinte información, obtén un obxecto con unha propiedade para cada medio de transporte, indicando o número de veces que se repite no array. É dicir, o resultado debería ser {car: 5, truck: 3, bike: 2, walk: 2, van: 2, pogostick: 1}

```
const data = [
  "car",
  "car",
  "truck",
  "truck",
  "bike",
  "walk",
  "car",
  "van",
  "bike",
  "walk",
  "car",
  "van",
  "car",
  "truck",
  "pogostick",
];
```

Set

O obxecto Set é unha colección de valores **únicos** de calquera tipo, valores primitivos ou obxectos. Os valores non poden estar repetidos no set.

```
const set1 = new Set([1, 2, 1, 2, 3, 3]);
console.log(set1);

const mySet1 = new Set();

mySet1.add(1); // Set [ 1 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add(5); // Set [ 1, 5 ]
mySet1.add("some text"); // Set [ 1, 5, 'some text' ]

console.log(`mySet1.has(1) = ${mySet1.has(1)}`); // true

console.log(`mySet1.size = ${mySet1.size}`); // 3

mySet1.delete(5); // removes 5 from the set
console.log(`mySet1.size = ${mySet1.size}`); // 2, porque se eliminou un valor

console.log(mySet1); // logs Set(2) { 1, "some text" }
```

NOTA: fixarse que a propiedade **size** devolve o número de valores no Set. Non confundir con length en arrays.

Non se pode acceder a elementos individuais do Set. A única forma de compobar se un elemento está no conxunto é usar o método has().

Un set pode percorrerse usando un bucle **for…of.**

```
for (const item of mySet1) {
  console.log(item);
}

for (const item of mySet1.keys()) {
  console.log(item);
}

for (const item of mySet1.values()) {
  console.log(item);
}
```

Unha utilidade dos Set é **eliminar os duplicados dun array**:

```
const numeros = new Set([1, 2, 1, 2, 3, 3]);
// converter o conxunto nun array
const numerosNonRepetidos = [...new Set(numeros)];
console.log(numerosNonRepetidos);
```

Map

O obxecto Map almacena pares chave-valor e recorda a orde de inserción das chaves. Calquera valor (tanto obxectos como tipos primitivos) pode ser usado como chave ou valor. As chaves deben ser únicas no mapa.

Un mapa parécese a un obxecto, xa que estes tamén almacenan un conxunto de propiedades-valor. As propiedades dos obxectos son basicamente cadeas de caracteres. A diferencia dos obxectos, as chaves dos mapas poden ser de calquera tipo, incluso obxectos, conxuntos ou outros mapas.

```
const map1 = new Map();

map1.set('a', 1);
map1.set('b', 2);
map1.set('c', 3);
map1.delete('c');
```

```
console.log(map1);
console.log(map1.get('a'));   // expected output: 1
console.log(map1.has('a'));  // true
console.log(map1.size);    // expected output: 2
```

Formas de crear un mapa:

```
const myMap = new Map([
  [0, 'one'],
  [1, 'two'],
  [2, 'three'],
]);
console.log(myMap);

// equivalente a:
const arr = ['one', 'two', 'three'];
const mapa = new Map(arr.entries());
console.log(mapa);
```

Un mapa pode percorrerse usando un bucle **for…of** na orde en que se inseriron as chaves. En cada iteración do bucle devólvese un array con dous valores [chave, valor].

```
const myMap = new Map();
myMap.set(0, 'zero');
myMap.set(1, 'one');

for (const [key, value] of myMap) {
  console.log(`${key} = ${value}`);
}

for (const key of myMap.keys()) {
  console.log(key);
}
// 0 1

for (const value of myMap.values()) {
  console.log(value);
}
// zero one
```

Un obxecto é similar a un mapa, xa que ambos teñen pares chave-valor, permiten recuperar eses valores e borrar chaves. Por estes motivos, historicamente usáronse os obxectos como mapas. Sen embargo teñen algunhas diferencias:

Mapa	**Obxecto**
Non contén chaves por defecto	Un obxecto ten un prototipo, polo que contén chaves por defecto
Calquera valor pode ser unha chave	As chaves deben ser un String ou Symbol
As chaves están ordenadas pola orde de inserción	Aínda que as chaves están ordenadas hoxe en día, non foi sempre así. Mellor non fiarse da orde proporcionada.
O número de elementos obtense fácil coa propiedade size.	O número de propiedades debe ser determinado manualmente.

Exercicios:

1. O seguinte mapa almacena información dos eventos ocorridos durante un partido indicando o minuto no que se produciron:

```
const gameEvents = new Map([
  [17, "GOAL"],
  [36, "Substitution"],
  [47, "GOAL"],
  [61, "Substitution"],
  [64, "Yellow card"],
  [69, "Red card"],
  [70, "Substitution"],
  [72, "Substitution"],
  [76, "GOAL"],
  [80, "GOAL"],
  [92, "Yellow card"],
]);
```

   a. Crea un novo array chamado **eventos** que almacene os diferentes eventos (non o minuto) ocorridos durante o partido (sen que haxa duplicados).
   b. Recorre con un bucle o mapa gameEvents e mostra información de cada evento, indicando se ocorreu na primeira metade ou na segunda metade do partido, por exemplo: [PRIMEIRA PARTE] 17: GOAL.

Programación orientada a obxectos en JavaScript

JavaScript é unha linguaxe baseada en prototipos, xa que o comportamento dun obxecto está especificado polas súas propiedades e as propiedades do seu prototipo.

Dende ES2015, a programación orientada a obxectos en JavaScript funciona de forma similar a outras linguaxes de programación. A funcionalidade das clases, a creación de xerarquías de obxectos e a herdanza de propiedades fan de JavaScript unha linguaxe máis parecida a outras linguaxes orientadas a obxectos, como Java.

Entre os conceptos vistos ata agora de JavaScript, xa apareceron as clases:

```
const bigDay = new Date(2019, 6, 19);
console.log(bigDay.toLocaleDateString());
```

A primeira liña do exemplo anterior crea unha instancia da clase Date, chamada bigDay. Na segunda liña chámase ao método toLocaleDateString() da instancia bigDay.

En JavaScript pode declararse unha clase usando a palabra chave **class**.

```
class MyClass {
  // Constructor
  constructor() {
    // Constructor body
  }

  // Instance field
  myField = "foo";

  // Instance method
  myMethod() {
    // myMethod body
  }

  // Static field
  static myStaticField = "bar";

  // Static method
  static myStaticMethod() {
    // myStaticMethod body
  }

  // Static block
  static {
    // Static initialization code
  }

  // Fields, methods, static fields, and static methods all have "private" forms
  #myPrivateField = "bar";
}
```

```
class Color {
  constructor(r, g, b) {
    // Assign the RGB values as a property of `this`.
    this.values = [r, g, b];
  }
}

// Fixarse que se chama ao construtor usando o nome da clase.
const vermello = new Color(255, 0, 0);
console.log(vermello);
```

Para crear unha instancia da clase, úsase o construtor (new), que realiza as seguintes tarefas:

- crea un novo obxecto.
- enlaza **this** co novo obxecto creado.
- executa o código do construtor.
- devolve o novo obxecto. **NOTA**: no construtor non se pode usar a sentencia return para devolver un valor. Por defecto debe devolver o novo obxecto.

Cada vez que se chama ao construtor, créase unha instancia diferente.

Se non se necesita ningunha inicialización especial, pode omitirse o construtor, xa que se xerará automaticamente un por defecto:

```
class Animal {
  sleep() {
    console.log('zzzzzzz');
  }
}

const spot = new Animal();
spot.sleep(); // 'zzzzzzz'
```

A palabra chave **static** define métodos ou propiedades estáticas de clase, é dicir, para usalos non é necesario instanciar a clase.

```
class Point {
  static displayName = "Point";
  static getInfo() {
    return '...';
  }
}
console.log(Point.displayName);      // "Point"
console.log(Point.getInfo());
```

Poden engadirse **métodos** a unha clase:

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
  }
  getRed() {
    return this.values[0];
  }
  setRed(value) {
    this.values[0] = value;
  }
}

const vermello = new Color(255, 0, 0);
console.log(vermello.getRed());
vermello.setRed(0);
console.log(vermello.getRed());
```

No exemplo anterior, os métodos color.getRed() e color.setRed() permiten ler e modificar a compoñente vermella da cor. Linguaxes como Java utilizan este patrón para acceder ás variables, sen embargo, en JavaScript poden usarse outros mecanismos de acceso (*accessor fields*) ás propiedades dun obxecto.

```
class Color {
  constructor(r, g, b) {
    this.values = [r, g, b];
  }
  get red() {
    return this.values[0];
  }
  set red(value) {
    this.values[0] = value;
  }
}

const vermello = new Color(255, 0, 0);
vermello.red = 0;
console.log(vermello.red); // 0
```

No exemplo anterior os métodos de acceso get/set permiten **simular** que o obxecto ten unha propiedade **red.** En realidade son dous métodos precedidos por **get** e **set**, que permiten ser manipulados como se fosen variables.

Unha utilidade deste setter é que, antes de asignar un valor a unha propiedade, permite realizar a comprobación de se o valor é apropiado. No exemplo anterior podería comprobarse que o valor pasado como parámetro entra dentro do rango permitido na coordenada de cores.

NOTA: se unha propiedade só ten o getter e non o setter, é unha propiedade de só lectura.

En JavaScript é posible implementar a **herdanza** creando subclases que herden doutra:

```
class ColorWithAlpha extends Color {
  #alpha;

  constructor(r, g, b, a) {
    // hai que chamar a super antes de usar this
    super(r, g, b);
    this.#alpha = a;
  }

  get alpha() {
    return this.#alpha;
  }

  set alpha(value) {
    if (value < 0 || value > 1) {
      throw new RangeError("Alpha value must be between 0 and 1");
    }
    this.#alpha = value;
  }
}
```

A clase ColorWithAlpha herda todos os métodos da clase ancestra:

```
const color = new ColorWithAlpha(255, 0, 0, 0.5);
console.log(color.red); // 255
```

As subclases tamén poden sobrescribir métodos das clases ancestras. Por exemplo, todas as clases herdan da clase Object, que define métodos básicos como por exemplo toString(). Sen embargo, o método por defecto toString non é útil porque imprime “[object Object]” na maioría dos casos. Pode redefinirse este método para darlle unha implementación máis útil.

```
class Color {
  // …
  toString() {
    return this.values.join(", ");
  }
}
```

O método **toString()** tamén se utiliza para ordenar un array de obxectos, pois sort() ordena alfabeticamente para o que chama ao método toString() do obxecto a ordenar.

```
class Student {
    ...
    toString() {
        return this.apelidos + ', ' + this.nome;
    }
}

// alumnado é un array de estudantes, sort ordena usando o método toString()
alumnado.sort();
```

Exercicio:

1. Crea unha clase chamada Produto coas propiedades marca, categoría, unidades, prezo e un método chamado importe que devolva o prezo total das unidades (unidades x prezo). Ademais terá un método getInfo que devolverá “Categoría (marca): unidades x prezo = importe”. Crea tres obxectos diferentes da clase Produto.
2. Crea unha clase Televisor que herde de Produto e teña unha nova propiedade chamada tamaño. O método getInfo mostrará o tamaño, ademais do resultado do método getInfo da calse ancestra.
3. Crea 5 produtos e gárdaos nun array. Crea as seguintes funcións (todas reciben o array como parámetro):
    a. prodsSortByCategory: devolve un array cos produtos ordenados alfabeticamente por categoría.
    b. prodsSortByPrice: devolve un array cos produtos ordenados por importe de maior a menor.
    c. prodsTotalPrice: devolve o importe total dos produtos do array, con dúas cifras decimais.
    d. prodsWithLowUnits: ademais do array, recibe como segundo parámetro un número de unidades e devolve un array con todos os produtos dos que quedan menos das unidades indicadas
    e. prodsList: devolve unha cadena que di ‘Listado de produtos:’ e en cada liña un guión e a información dun produto do array.

Expresións regulares

As expresións regulares permiten crear patróns para buscar combinacións de caracteres nunha cadea, é dicir, permiten realizar a busca de texto dunha forma relativamente sinxela. Son moi útiles á hora de validar formularios e substituír texto. En JavaScript, as expresións regulares son consideradas obxectos.

Poden construírse expresións regulares de dúas maneiras:

- Usando unha **expresión regular literal**, que consiste nun patrón entre barras /:

```
// Sintaxe:  /regexp/flags
const re = /ab/;
```

  As expresións regulares literais **compílanse cando se carga o script**. Se a expresión permanece constante, este método mellora o rendemento.

- Chamando ao construtor do **obxecto RegExp**:

```
// Sintaxe: new RegExp(regexp, flags)
const re = new RegExp(/ab/);    // notación literal
const re = new RegExp('ab');    // cadea
```

  Este tipo de expresións regulares **compílase en tempo de execución**. Debe usarse este método cando se sabe que o patrón vai cambiar durante a execución do script, ou cando non se sabe o patrón inicialmente, porque se vai recoller doutra fonte, por exemplo como entrada de usuario.

As dúas formas anteriores de crear expresións regulares permiten engadir **modificadores adicionais (flags)**, ben despois da segunda barra (/) na primeira opción, ou como segundo parámetro na segunda opción. Poden consultarse todos os modificadores en internet. Os máis usados son:

- **/i** (.ignoreCase): non distingue entre maiúsculas e minúsculas. Por exemplo. /html/i buscará html, Html, HTML, …
- **/g** (.global): busca global. Permite seguir buscando coincidencias en lugar de pararse ao encontrar a primeira.
- **/m** (multiline): só afecta ao comportamento de ^ e $. Permite buscar en cadeas con saltos de liña (\n). Cando está activado busca, non só ao principio e ao final da cadea, senón que tamén busca ao inicio/fin de liña.

```
const re = /ab/i; // notación literal
const re = new RegExp('ab', 'i'); // construtor con cadea como 1º argumento
const re = new RegExp(/ab/, 'i'); // construtor con notación literal como 1º argumento
```

```
const text = `Ola mundo`;

// Probar o modificador i:
const regexp = /ola/i;
console.log(regexp.test(text));
regexp.exec(text);
```

As expresións regulares son usadas con métodos propios do obxecto (RegExp) como exec() e test() e tamén cos métodos match(), matchAll(), replace(), replaceAll(), search(), e split() de String.

Creación de expresións regulares

Unha expresión regular está composta de caracteres simples, como /abc/, ou unha combinación de caracteres simples e especiais, como /ab*c/.

Un patrón formado por caracteres simples constitúe unha expresión regular. Por exemplo /abc/ buscará exactamente a combinación de caracteres /abc/ (todos os caracteres xuntos e nesa orde).

Cando se pretenden buscar algo como unha ou máis letras b, deben incluírse caracteres especiais no patrón. Por exemplo, /ab*c/ buscará a letra 'a' seguida de cero ou máis letras 'b', seguidas de 'c'. Polo tanto * significa "0 ou máis ocorrencias do elemento precedente". Así, esta busca no patrón "cbbabbbbcdebc", encontrará a cadea "abbbbc".

A continuación proporciónanse unha lista de caracteres especiais, coa descrición e exemplos. Pode encontrarse unha táboa resumo en Regular expression syntax cheatsheet - JavaScript | MDN :

- **Asercións**: as asercións inclúen bordes e límites que indican o inicio e fin de liñas e palabras. Exemplos.

^	principio de texto.
	/^A/.test("alias A"); // false
	/^A/.test("Alias A"); // true

$	fin de texto. /t$/.exec("eat");

\b	delimitador de palabra.
	/\bm/.test("moon"); // true
	/oon\b/.test("moon"); // true

\B	que non sexa borde de palabra. /\Bon/.test("at noon"); // true
x(?=y)	aserción anticipada. Coincide con "x" só se "x" vai seguida de "y". /Jack(?=Sprat)/.test("JackSprat"); /Jack(?=Sprat)/.exec("JackSprat");
x(?!y)	aserción anticipada negativa. Coincide con "x" só se "x" non está seguida de "y". Por exemplo, /\d+(?!\.)/ coincide con un número só se non vai seguido dun punto decimal. /\d+(?!\.)/.exec('3.141'); // coincide con 141 e non con 3.

(?<=y)x	aserción de busca inversa. Coincide con "x" só se "x" está precedida por "y". Por exemplo, /(?<=Jack)Sprat/ coincide con "Sprat" só se vai precedida de "Jack". /(?<=Jack)Sprat/.test("JackSprat"); /(?<=Jack)Sprat/.exec("JackSprat");
(?<!y)x	aserción de busca inversa negativa. Coincide con "x" só se "x" non está precedida por "y". Por exemplo, /(?<!-)\d+/ coincide con un número só se non está precedido por un signo de menos. /(?<!-)\d+/.test('3') /(?<!-)\d+/.test('-3')

- **Clases de caracteres**: permite distinguir entre caracteres. Exemplos.

[abcd] [a-d]	calquera dos caracteres entre corchete. Pode especificarse un rango co guión (-). /[abcd]/.test("tree");
[^abcd] [^a-d]	calquera excepto os caracteres indicados. Pode especificarse un rango co guión (-). /[^abcd]/.exec("tree");

.	Ten dous significados
	• un único carácter (calquera excepto principio ou final de liña).
	/.y/.test("my"); // true
	/.y/.test("yes"); //false
	• Dentro dunha clase de caracteres, o punto perde o seu significado especial e concorda co punto literal.
	/[.]/.exec("abc.");

\d	un díxito. É equivalente a [0-9]. /\d/.exec("B2"); // coincide con "2"
\D	calquera carácter que non sexa un díxito. Equivalente a [^0-9]. /\D/.exec("B2"); // coincide con "B"
\w	calquera carácter do alfabeto latino básico, incluído o subliñado (_). Equivalente a [A-Za-z0-9_]. Por exemplo, /\w/ coincide con "m" en "mazá", "5" en "$5.28" e "m" en "Émanuel". /\w/.exec("mazá"); /\w/.exec("$5.28") /\w/.exec("Émanuel")
\W	calquera carácter que non sexa do alfabeto latino básico. Equivalente a [^A-Za-z0-9_]. Por exemplo, /\W/ ou /[^A-Za-z0-9_]/ coincide con "%" en "50%" e "É" en "Émanuel". /\W/.exec("50%"); /\W/.exec("Émanuel");

\s	espazo en branco. Inclúe espazo, tabulación, avance de páxina, avance de liña e outros espazos Unicode. Por exemplo, /\s\w*/ encontra " bar" en "foo bar". /\s\w*/.exec("foo bar");
\S	un só carácter que non sexa espazo en branco (pode ser unha letra). /\S\w*/.exec("foo bar");
\t	tabulación.
\r	retorno de carro
\n	salto de liña
\f	avance de páxina
\	indica que o seguinte carácter debe tratarse de forma especial ou "escaparse". Por exemplo, na expresión regular /\b/, b ten o significado do carácter especial co significado de límite de palabra. E na expresión regular /a*/, indica que o * debe tratarse como carácter e non co significado especial.
x|y	coincide con x ou con y. Proporciona dúas alternativas de busca. Por exemplo, /green|red/ coincide con "green" en "green apple" e "red" en "red apple". /green|red/.exec("green apple"); /green|red/.exec("red apple");

- **Cuantificadores**: indican o número de caracteres ou expresións que deben coincidir. Exemplos.

x*	**0 ou máis veces** o elemento "x" anterior. /bo*/.exec("A ghost booooed"); /bo*/.exec("A bird warbled");
x+	**1 ou máis veces** o elemento "x" anterior. Equivalente a {1,}. Por exemplo, /a+/ coincide coa "a" en "candy" e todas as "a"es en "caaaaaaandy". /a+/.exec("candy"); /a+/.exec("caaaaaaandy");
x?	**0 ou 1 vez** o elemento "x" anterior". Por exemplo, /e?le?/ coincide con "el" en "angel" e "le" en "angle". /e?le?/.exec("angel"); /e?le?/.exec("angle");

x{n}	“n” aparicións do elemento “x” (n é un número enteiro positivo). Por exemplo, /a{2}/ non coincide coa "a" de "candy", pero coincide coas "a"es de "caandy" e as dúas primeiras "a"es en "caaandy". /a{2}/.exec("candy"); /a{2}/.exec("caandy"); /a{2}/.exec("caaandy");
x{n,}	“n” ou máis aparicións do elemento “x” (“n” é un número enteiro positivo). Por exemplo, /a{2,}/ non coincide coas "a"es en "candy", pero coincide con todas as "a"es en "caandy" e en "caaaaaaandy". /a{2,}/.exec("candy"); /a{2,}/.exec("caandy"); /a{2,}/.exec("caaandy");
x{n,m}	entre “n” e como máximo “m” aparicións do elemento “x” (“n” é 0 ou un número positivo, “m” é un número positivo e m>n). Por exemplo, /a{1,3}/ non coincide con nada en "cndy", a "a" en "candy", as dúas "a"es en "caandy" e as tres primeiras "a"es en "caaaaandy". Observa que ao comparar "caaaaandy", encontra as "aaa", aínda que a cadea orixinal tiña maís "a"es. /a{1,3}/.exec("candy"); /a{1,3}/.exec("caandy"); /a{1,3}/.exec("caaandy"); /a{1,3}/.exec("caaaandy");

De maneira predeterminada, os cuantificadores * e + son “ambiciosos”, o que significa que intentan facer coincidir a maior cantidade posible da cadea. Colocar o carácter “?” despois de *, +, ? ou {} (x*?, x+?, x??, x{n}?, x{n,}?, x{n,m}?) fai que non sexan “ambiciosos”, o que significa que se deterá tan pronto como encontre unha coincidencia. Por exemplo, dada a cadea "some <foo> <bar> new </bar> </foo> thing":

- **/<.*>/.exec("some <foo> <bar> new </bar> </foo> thing");** //coincidirá con "<foo> <bar> new </bar> </foo>"
- **/<.*?>/.exec("some <foo> <bar> new </bar> </foo> thing");** // coincidirá con "<foo>"

Métodos de expresións regulares e cadeas

Calquera obxecto RegExp ten os seguintes métodos para realizar unha busca:

- exec(str): busca a expresión regular na cadea pasada como parámetro.

  Se non está activado o modificador **g**, devolve a primeira coincidencia.

  Se o modificador **g** está activado, devolve a primeira coincidencia e almacena a posición seguinte á coincidencia na propiedade **regexp.lastIndex.** A seguinte vez

que se realiza a busca, empeza a buscar dende lastIndex, devolve a coincidencia e actualiza lastIndex, e así sucesivamente.

Se non a encontra devolve null e establece a propiedade lastIndex da expresión regular a 0.

Se encontra o patrón, devolve un array con información da coincidencia. O array devolto ten a coincidencia como primeiro elemento do array ademais dun elemento por cada grupo de captura. O array tamén ten outras propiedades adicionais, entre as que están:

- **index**: o índice da coincidencia
- **input**: a cadea orixinal.

```
let str = "I am amazed in America";
let reg = /am/gi;
console.log(reg.exec(str)); // Imprime ["am", index: 2, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["am", index: 5, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime ["Am", index: 15, input: "I am amazed in America"]
console.log(reg.exec(str)); // Imprime null
```

NOTA: lastIndex é unha propiedade de RegExp que especifica o índice no que empezar a buscar a seguinte vez.

- test(str): busca a expresión regular na cadea pasada como parámetro. Devolve **true** se a encontra e **false** en caso contrario.

  Se o modificador **g** está activado, busca dende a posición indicada en **regexp.lastIndex** e actualiza esta propiedade igual a como o fai que o método exec.

```
let str = "I am amazed in America";
let reg = /am/g;
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime true
console.log(reg.test(str)); // Imprime false, hai só dúas coincidencias
```

NOTA: test recibe unha cadea como parámetro. Se o parámetro non é string poden producirse resultados inesperados.

As cadeas tamén teñen métodos para traballar con expresións regulares:

- match(regexp): devolve un array coas coincidencias do patrón na cadea se está activado o flag *g*. En caso contrario devolve só a primeira coincidencia. Se o patrón non coincide, devolve null.

```
const paragraph = 'The quick brown fox jumps over the lazy dog. It barked.';
const regex = /[A-Z]/g;
const found = paragraph.match(regex);

console.log(found);   // expected output: Array ["T", "I"]
```

- search(regexp): devolve o índice da primeira coincidencia ou -1 se non coincide.

```
const str = "hey JudE";
const regex = /[A-Z]/;
console.log(str.search(regex)); // devolve 4, o índice da primeira maiúscula
```

- replace(pattern, replacement): devolve unha nova cadea con un, algún ou todas as coincidencias do patrón substituídas por *replacement*.

```
const str = 'Twas the night before Xmas...';
const newstr = str.replace(/xmas/i, 'Christmas');
console.log(newstr);  // Twas the night before Christmas...
```

  O segundo parámetro pode ser:

  - unha cadea.
  - unha función, que se invocará para cada coincidencia e o valor devolto pola función será usado como texto de substitución. Os **parámetros** desta función son: a coincidencia, cadeas encontradas polos grupos de captura (p1, p2, …, pN), offset (índice da coincidencia na cadea), a cadea completa e un obxecto grupos (obxecto onde as chaves son os nomes dos grupos e os valores as coincidencias atopadas para cada grupo, ou undefined se non houbo coincidencia. Só estará presente se o patrón contén polo menos un grupo de captura con nome).

```
function styleHyphenFormat(propertyName) {
  function upperToHyphenLower(match, offset, string) {
    return (offset > 0 ? '-' : '') + match.toLowerCase();
  }
  return propertyName.replace(/[A-Z]/g, upperToHyphenLower);
}

console.log(styleHyphenFormat("borderTop"));   // border-top
```

No segundo argumento do método **replace**, pódense usar combinacións de caracteres especiais para inserir fragmentos da coincidencia.

- $&: inserta a coincidencia completa
- $`: inserta a parte da cadea antes da coincidencia
- $': inserta a parte da cadea despois da coincidencia.
- $n: inserta a coincidencia do paréntese número n.
- $<name>: inserta a coincidencia do paréntese co nome **name**.
- $$: inserta o carácter $.

```
console.log("I love HTML".replace(/HTML/, "$& and JavaScript"));
```

- split(separador): usa unha cadea ou expresión regular como separador para romper a cadea nun array de subcadeas.

```
const str = 'one,two.three four';
const result = str.split(/[,.\s]/);
// ['one', 'two', 'three', 'four']
console.log(result);
```

Grupos e rangos

Un "**grupo de captura**" é unha parte dunha expresión regular que se escribe entre paréntese (...), creando unha subexpresión.

Unha das utilidades dos grupos de captura é que permiten aplicar cuantificadores ao grupo. Por exemplo, sen paréntese **go+** significa o carácter **g** seguido de **o** unha ou máis veces, é dicir, coincidiría con go, gooo ou gooooo, por exemplo. O uso de parénteses permite agrupar caracteres, polo que (go)+ coincidirá con go, gogo, gogogo, etc.

Exemplo:

```
"Gogogo".match(/(go)+/ig)
// ['Gogogo']
```

Outra das utilidades dos grupos de captura é que, cando se usan para buscar unha expresión regular, o resultado da busca tamén contén o texto que coincide cos grupos de captura definidos. Por exemplo, o método **str.match(regExp)**, cando non se usa o modificador **g**, devolve na posición 0 do array resultado a coincidencia completa, na posición 1 estará o texto que coincide co contido do primeiro paréntese, na posición 2 o texto que coincide co contido do segundo paréntese, etc.

```
'<h1>Hello, world!</h1>'.match(/<(.*?)>/);
// ['<h1>', 'h1', index: 0, input: '<h1>Hello, world!</h1>', groups: undefined]
```

O método **str.replace(regExp, replacement)** permite usar o contido dos parénteses na cadea de substitución usando $n, sendo n o número do grupo.

```
let regexp = /(\w+) (\w+)/;
"John Bull".replace(regexp, '$2, $1');    // 'Bull, John'
```

Existen diferentes ferramentas en liña para depurar expresións regulares, editalas e probalas: RegExr, Debuggex.

Exercicios:

1. Escribe unha expresión regular para comprobar que cada un dos seguintes elementos aparece nunha cadea.
    a. *car* e *cat*
    b. *pop* e *prop*
    c. *ferret*, *ferry* e *ferrari*
    d. Calquera palabra rematada en *ious*
    e. Un espazo seguido de punto, coma, dous puntos ou punto e coma.
    f. Unha palabra con máis de 6 letras.
    g. Unha palabra sen a letra e (ou E).

    Unha vez teñas a expresión regular creada, comproba se se pode facer máis pequena.

    Utiliza o seguinte código para comprobar o resultado:

```
verify(/.../, ["my car", "bad cats"], ["camper", "high art"]);
verify(/.../, ["pop culture", "mad props"], ["plop", "prrrop"]);
verify(/.../, ["ferret", "ferry", "ferrari"], ["ferrum", "transfer A"]);
verify(/.../, ["how delicious", "spacious room"], ["ruinous", "consciousness"]);
verify(/.../, ["bad punctuation ."], ["escape the period"]);
verify(
  /.../,
  ["Siebentausenddreihundertzweiundzwanzig"],
  ["no", "three small words"]
);
verify(/.../, ["red platypus", "wobbling nest"], ["earth bed", "learning ape", "BEET"]);

function verify(regexp, yes, no) {
  // Ignore unfinished exercises
  if (regexp.source == "...") return;
  for (let str of yes) if (!regexp.test(str)) {
    console.log(`Failure to match '${str}'`);
  }
  for (let str of no) if (regexp.test(str)) {
    console.log(`Unexpected match for '${str}'`);
  }
}
```

2. Unha dirección de rede MAC está composta por 6 grupos de dous números hexadecimais separados por “:”. Por exemplo "01:32:54:67:89:AB".

   Escribe unha expresión regular que comprobe se unha dirección MAC é correcta.

3. Crea unha expresión regular para buscar códigos de cores no formato #RGB ou #RRGGBB nun texto. Un código de color é unha cadea formada polo carácter “#” seguido de 3 ou 6 cifras hexadecimais.
4. Escribe unha expresión regular para números. Debe incluír números enteiros, decimais e números negativos. Así, na cadea "-1.2 4 0 -123.5." debe encontrar 4 números
5. Crea unha función que comprobe se un contrasinal é válido, é dicir, cumpre as seguintes condicións:
   a. Mínimo 8 caracteres.
   b. Sen espazos en branco.
   c. Que teña, polo menos, 3 das seguintes tipos de caracteres:
      i. maiúsculas
      ii. minúsculas
      iii. números
      iv. caracteres especiais: ¡!$?%&#@/\()=¿?*[];,:._<>+-
6. Ás veces é útil eliminar as etiquetas HTML dun texto para evitar que se inclúa código mal intencionado nunha páxina web.

   Crea unha función á que se lle pase un texto e devolva o mesmo texto coas etiquetas HTML eliminadas.

7. Dado o seguinte array de insultos, fai un script tal que cada vez que apareza un deles nun texto o substitúa pola primeira letra do insulto e un número de asteriscos igual á lonxitude do insulto - 1.

   ```
   let insultos = ["testán", "langrán", "fervellasverzas", "baldreo", "lacazán", "pillabán"];
   ```

   Así, por exemplo, cada vez que apareza testán nun texto, debe substituírse por "t*****".

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
- Desarrollo Web en Entorno Cliente | materials
- Standard built-in objects - JavaScript | MDN

- Objetos predefinidos, arrays y funciones · MTIG-17 JS
- Types of JavaScript Objects: Built-in vs User-defined | by Joseph Khan | tajawal | Medium
- RegExp - JavaScript | MDN
- Javascript en español - Lenguaje JS
- Regular Expressions :: Eloquent JavaScript
- Capturing groups
- The Modern JavaScript Tutorial
- Eloquent JavaScript