Obxectos

Introdución

O tipo de datos fundamental de JavaScript é o obxecto. Un obxecto é un dato complexo composto por unha **colección non ordenada de propiedades.** Unha propiedade é un par "chave: valor", onde a chave é unha cadea identificada como o "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 compórtanse como obxectos inmutables, é dicir, non se poden modificar as súas propiedades. Ademais, cando se pasan a unha función, 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 credos 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 estes herdan propiedades e métodos doutros obxectos.

Strings

Ademais do tipo de dato string, JavaScript define o obxecto String que é un wrapper sobre o tipo primitivo string. O obxecto String ofrece un conxunto de propiedades e métodos útiles para 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"); // rara vez se usa
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. Para crear un obxecto String é necesario usar o construtor **new String()**. Os tipos primitivos teñen acceso aos métodos e propiedades do obxecto String porque son envoltos no correspondente obxecto (wrapper).

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

Concatenar cadeas

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

Os **patróns de cadeas** son cadeas literais delimitados polo símbolo ` (acento grave). Funcionan como cadeas normais, excepto que permiten incluír variables e avaliar expresións usando o envoltorio ${ }. Foron introducidos en JavaScript no 2015 (ES6).

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

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

O obxecto String proporciona métodos para manipular cadeas, que **non modifican** o obxecto orixinal, senón que devolven un novo 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.

- at(): devolve o carácter na posición indicada. O primeiro carácter ocupa a posición 0. Permite usar números negativos que contan dende o último carácter.

  Exercicio: como se accede ao último carácter dunha cadea?

  Tamén é posible tratar as cadeas como un array, polo que se pode acceder aos caracteres usando corchetes ('cat'[1] // gives value "a"). Sen embargo, non é posible modificar os caracteres utilizando a notación de arrays:

```
let x = 'DWCC';
x[0] = 'd'; // TypeError:  Cannot assign to read only property '0' of string 'DWCC'
```

- endsWith(): comproba se a cadea remata coa cadea especificada.
- includes(): comprobar se unha cadea está contida noutra.
- indexOf(): busca a cadea pasada como parámetro e devolve o índice da primeira ocorrencia.
- lastIndexOf(): busca a cadea pasada como parámetro e devolve o índice da última ocorrencia.
- padStart() e padEnd(): métodos que completan a cadea actual coa cadea pasada como parámetro ata chegar ao número de caracteres indicados. Os caracteres de recheo colócanse ao inicio (padStart) da cadea ou ao final (padEnd) da cadea.
- replace(): devolve unha nova cadea resultado de substituír a primeira ocorrencia do patrón polo argumento indicado como segundo parámetro. Se o patrón é unha cadea só se substituirá a primeira ocorrencia da mesma.
- slice(): extraer unha subcadea doutra. Devolve unha nova cadea sen modificar a orixinal.
- split(): devolve un array de cadeas resultado de dividir a cadea orixinal utilizando o separador pasado como parámetro.
- startsWith(): comproba se a cadea empeza coa cadea especificada.

- substring(): devolve a parte da cadea entre o inicio e fin. Exemplo: **'Mozilla'.substring(1,3)** devolve “oz”.

  NOTA: slice() e substring() teñen diferencias moi pequenas: unha delas é que slice permite utilizar números negativos como parámetros, empezando a contar polo final da cadea en lugar de polo principio.

- trim(): elimina os espazos en branco do principio e fin da cadea.

NOTA: é posible encadear métodos: **'Mozilla'.substring(1,3).toUpperCase().at(1)**

Exercicios:

1. Dada a seguinte constante, fai as operacións necesarias para que o primeiro carácter estea en maiúscula:

```
const cadea = 'desenvolvemento web';

// engade o teu código aquí

console.log(novaCadea); // 'Desenvolvemento web'
```

2. 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

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

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

Number

Ademais do tipo de dato primitivo Number, JavaScript ofrece un obxecto Number con propiedades e métodos interesantes para traballar con números.

Diferentes formas de crear números:

```
let numero1 = 1;
let numero2 = Number(2);
let numero3 = new Number(3);    // rara vez se usa
console.log(numero1, typeof numero1);
console.log(numero2, typeof numero2);
console.log(numero3, typeof numero3);
```

Non é habitual usar o construtor Number(), mais hai que ter en conta o seu funcionamento:

```
console.log(`new Number(42) === 42 -> ${new Number(42) === 42}`);
console.log(`new Number(42) == 42 -> ${new Number(42) == 42}`);
```

Métodos estáticos:

- Number.isInteger(): determina se o número pasado é enteiro.
- Number.isNaN(): determina se o número pasado é NaN.
- Number.parseInt()
- Number.parseFloat()

Métodos de instancia:

- Number.prototype.toFixed(): devolve unha cadea representando o número cos decimais indicados.

Exercicio:

1. ¿Como calcularías o número de cifras dun número enteiro positivo utilizando propiedades e métodos dos obxectos vistos ata agora?

Math

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

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

A diferencia doutros obxectos globais, Math non ten 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 maior ou igual a 0 e menor que 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.

NOTA: ollo co uso de Math.ceil con Math.round.

> Math.random() -> devolve un número entre [0, 1)
> Math.ceil(Math.random() * 10) -> devolve un número **enteiro** entre [0, 10]
> Math.floor(Math.random() * 10) -> devolve un número **enteiro** entre [0, 10)
> Aínda que é pouco probable, o 0 está incluído no rango de números devoltos.

Exercicios:

1. Números aleatorios:
    a. Xera un número **enteiro** aleatorio entre 0 e 3 (incluídos).
    b. Xera un número enteiro aleatorio entre 1 e 3 (incluídos).
    c. Crea unha función que devolva un número enteiro aleatorio entre os dous valores pasados como parámetros. Así, por exemplo, a seguinte instrución debe mostrar un número aleatorio entre 5 e 10 (incluídos):
    console.log(numeroAleatorio(5, 10));
2. 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.
3. 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.

Date

O obxecto Date permite representar un momento no tempo nun formato independente da plataforma. Internamente é representada como un número (*timestamp*), que representa os milisegundos dende o 1 de xaneiro de 1970.

Sempre hai dúas formas de interpretar o timestamp: como hora local ou UTC (Coordinated Universal Time) hora global definida polo World Time Standard. A zona horaria non se almacena no obxecto Date, senón que é determinada pola contorna de execución.

Hai 5 formas básicas do construtor Date():

- new Date(): sen parámetros, o obxecto creado representa a data e hora actual.
- new Date(value): pásase como parámetro un enteiro que representa o número de milisegundos dende o 1 de xaneiro de 1970.
- new Date(dateString): unha cadea representando unha data no formato ISO 8601.
  YYYY-MM-DDTHH:mm:ss.sssZ
  Máis información do formato dunha data.
- new Date(dateObject): pasando como parámetro un obxecto Date existente, crea unha copia do mesmo.
- new Date(valores individuais de compoñentes): pásanse como parámetros o ano, mes (0-11), día, hora, minuto, segundo e milisegundo. Como mínimo deben pasarse o ano e o mes. O resto de parámetros que falten 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 data á seguinte unidade. Por exemplo new Date(1990, 12, 1) devolverá o 1 de xaneiro de 1991

Exemplos:

```
const today = new Date();
console.log(today);

const date1 = new Date(628021800000); // passing epoch timestamp
console.log(date1);

// DISCOURAGED: may not work in all runtimes
const date2 = new Date('December 17, 1995 03:24:00');
console.log(date2);

// This is standardized and will work reliably
const date3 = new Date('1995-12-17T03:24:00'); // This is ISO-8601-compliant
console.log(date3);

const date4 = new Date(1995, 11, 17); // the month is 0-indexed
console.log(date4);

const date5 = new Date(1995, 11, 17, 3, 24, 0);
console.log(date5);
```

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 (toString()) devolverá “Invalid date” e valueOf() devolverá NaN.

Cando se invoca a **función Date()** (sen a palabra chave new) devolve unha cadea representando a data e hora actual.

```
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:

- Date.now(): devolve o número de milisegundos que pasaron dende o 1 de xaneiro de 1970.
- getDate() e setDate(): devolve/establece o día do mes (1-31) da data especificada.
- getDay(): devolve o día da semana (0-6) dunha data. (0 é o domingo).
- getFullYear() e setFullYear(): devolve/establece o ano (4 díxitos).
- 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).
- toLocaleString(): devolve unha cadea con unha representación da data nun formato sensible ao idioma. Utiliza a API Intl.DateTimeFormat
- 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 do ano actual.
2. Crea unha función á que se lle pase un mes (1-12) e un ano e devolva o número de días dese mes.
3. Crea unha función á que se lle pase unha data e que devolva true se é fin de semana.
4. Crea unha función que reciba unha data como parámetro e devolva o número de días que pasaron dende que comezou o ano.

Arrays

Un array é un obxecto especial en JavaScript que permite almacenar múltiples valores.

Características de arrays en JavaScript:

- Son redimensionables.
- Poden conter valores de diferentes tipos de datos.
- Para acceder aos elementos do array utilízanse como índices números enteiros positivos. O primeiro elemento do array ten o índice 0.
- As operacións de copia de arrays crean copias pouco profundas (shallow copy).

Exemplos de creación de arrays:

```
const list = ['bread', 'milk', 'cheese', 'hummus'];   //notación literal - Recomendada
console.log(list);

const randomNumbers = ['tree', 795, [0, 1, 2]];      // array multidimensional
console.log(randomNumbers);

const people = [];
people[0]= "Ada";
people[1]= "Erea";
people[2]= "Navia";
console.log(people);

// Outra forma, aínda que menos habitual, de crear Arrays é usar o construtor
const anos = new Array(1991, 1984, 2008, 2020);
console.log(anos);

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

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

Operacións básicas con arrays:

- Acceder a un elemento do array:

```
const list = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
console.log(list[0]);

// arrays multidimensionais
const randomNumbers = ['tree', 795, [0, 1, 5]];
console.log(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.

Propiedades e métodos básicos de arrays

JavaScript inclúe propiedades e métodos predefinidos para traballar con arrays.

As principais propiedades e métodos para traballar con arrays:

- A propiedade length representa o número de elementos no array. A lonxitude dun array sempre será unha unidade maior que o índice do último elemento: list[list.length - 1].

  A propiedade **length pode modificarse**. Establecer o valor de length a un valor menor da lonxitude actual do array, eliminará os elementos do final do array. Polo contrario, establecer o valor de length a un valor maior que a lonxitude actual, estenderá o array:

```
const numbers = [1, 2, 3, 4, 5];
numbers.length = 3;
console.log(numbers); // [1, 2, 3]

const arr = [1, 2];
arr.length = 5; // set array length to 5 while currently 2.
console.log(arr); // [ 1, 2, <3 empty items> ]
```

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

- pop(): elimina o último elemento do array e devolve o elemento eliminado.
- push(): engade elementos ao final do array e devolve 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.
- 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]).
- 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**).
- includes(): comproba se un elemento está contido no array e devolve **true** en caso afirmativo ou **false** en caso negativo.
- indexOf(): devolve o primeiro índice no que se atopa o elemento a buscar. Se o elemento a buscar non está no array, devolve **-1.**
- slice(): devolve unha copia pouco profunda (shallow copy) dunha porción do array. **Non modifica o array orixinal**.
- splice(): **modifica o contido dun array** eliminando ou substituíndo elementos existentes e/ou agregando novos elementos. Devolve un array que contén os elementos eliminados.
- concat(): úsase para fusionar dous ou máis arrays. Este método non modifica os arrays orixinais, crea un novo array e devólveo.
- join(): devolve unha **cadea** resultado de concatenar todos os elementos no array separados por comas ou usando o separador especificado.
- flat(): crea un novo array con todos os elementos do sub-array concatenados ata a profundidade especificada.

JavaScript permite encadear varios métodos na mesma instrución. Para que funcione hai que ter en conta o valor devolto polo método anterior:

```
arr.splice(1, 4).reverse();
```

NOTA: Recordar que se pode iterar sobre un array utilizando o bucle for...of.

Exercicios:

1. Crea unha función que reciba un elemento e un array como parámetros. A función debe devolver un novo array que conteña os índices onde aparece ese elemento no array.

   Exemplo:

```
const numeros = [1, 3, 5, 1, 4, 1, 6, 8, 10, 1];

function indices(elemento, arrayElementos) {
  …
}
console.log(indices(1, numeros)); // (4) [0, 3, 5, 9]
```

2. Dado o array froitas (const froitas = ['peras', 'mazás', 'kiwis', 'plátanos', 'mandarinas'];) , fai os seguintes apartados co método **splice**:

    a. Elimina as mazás.
    b. Engade laranxas e sandía detrás dos plátanos,.
    c. Quita os kiwis e pon no seu lugar cereixas e nésperas.

    Despois de realizar cada operación mostra por pantalla a lista de froitas do array separadas por unha coma e un espazo. Por exemplo, inicialmente o array debe mostrarse como “peras, mazás, kiwis, plátanos, mandarinas”.

3. Crea unha función á que se lle pase unha frase con varias palabras e devolva a mesma frase coa primeira letra de cada palabra en maiúsculas e o resto de letras en minúsculas.

Copiar arrays

Cando se fai unha asignación dunha variable de tipo primitivo ou se pasa un tipo primitivo como parámetro a unha función, faise unha copia do seu valor. Calquera modificación posterior da copia non afecta á variable orixinal:

```
let a = 54;
let b = a; // a = 54 b = 54
b = 86;
console.log(`a = ${a}, b = ${b}`); // a = 54 b = 86
```

Sen embargo, ao facer unha asignación de 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'
console.log(
  `data1 = ${data1.toLocaleString()}, data2 = ${data2.toLocaleString()}`
);
```

Para facer 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];
console.log(fruitsCopy); // ["Strawberry", "Mango"]

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

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

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.

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 de aniñamento dos elementos do iterable. Se o array a copiar ten outros arrays aniñados, tanto no orixinal como na copia apuntarán á mesma dirección de memoria, polo que a modificación dun afectará ao outro.

```
const numeros = [1, 2, [3, 4]];
const copia = [...numeros];

copia[2][0] = 5;
console.log(copia);
console.log(numeros);
```

Para crear unha copia profunda dun array, pode usarse JSON.stringify() para converter o array a unha cadea JSON, e despois JSON.parse() para converter a cadea a un array completamente independente do orixinal:

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

Desestruturación de arrays

A desestruturación é unha característica que permite desempaquetar os valores dun array, ou propiedades de obxectos, en distintas variables.

A desestruturación de arrays permite recuperar os elementos individuais 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

const numeros = [10, 20, 30, 40, 50];
[a, b] = numeros;
console.log(`a = ${a}`); // expected output: 10
console.log(`b = ${b}`); // expected output: 20

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

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

// 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);  // ['a', 'c', 'b']
```

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

```
let a, b, rest;
[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.

Observar que 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 elemento de tipo array.

Exercicios para practicar a desestruturación de arrays:

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, proporciona unha <u>única sentencia</u> JavaScript para cada unha das seguintes instrucións:
   a. Crea as variables **players1, players2** que conteñan un array cos xogadores de cada equipo. Así, players1 terá os xogadores do primeiro equipo e players2 os do segundo 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 substituíu os xogadores iniciais por 'Thiago', 'Coutinho', 'Periscic'. Crea unha nova variable de tipo array chamada **players1Final** que conteña todos os xogadores: os iniciais e tamén os 3 novos.

2. Dado un array con nomes de variables formados por dúas palabras separadas por “_”, fai as operacións necesarias para mostrar 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”.

3. 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)
```

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 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 se usan **chaves { }** neste caso e as propiedades:valor están separadas por comas “,”:

```
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 obxecto1 = new Object();
const obxecto2 = { };   // 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 obxecto = {
  firstName: 'Carol',
  details: {
    mobile: '611223344',
    email: 'email@gmail.com'
  }
};
```

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

```
// usando un punto
obxecto.firstName = 'Ana';
// usando corchetes
obxecto['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, polo que haberá que usar corchetes. Tamén se usan corchetes cando o nome da propiedade se calcula en base a unha expresión. Exemplo:

```
const propiedade = 'firstName';
console.log(obxecto[propiedade]);

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

O acceso ás propiedades pode **encadearse**:

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

Se unha propiedade coincide con unha variable que se chama como ela, dende ES2015 pode usarse a seguinte sintaxe:

```
let firstName =  'Erea';
const obxecto = {
  firstName    // equivalente a firstName: firstName
};
console.log(obxecto);
console.log(obxecto.firstName);
```

Despois de creado un obxecto poden engadirse ou eliminarse propiedades:

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

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 unha propiedade de algo que non é un obxecto si que xera un erro.

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

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

Unha propiedade dun obxecto pode ser unha función, ou método. Exemplo:

```
const ereaObxecto = {
  nome: 'Erea',
  apelido: 'Pereiro',
  profesion: 'profesora',
  birthYear: 1991,
  greet: function () {
    console.log(`Ola, o meu nome é ${this.nome}`);
  },
  calcAge: function (year) {
    return year - this.birthYear;
  }
};

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

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

NOTA: nunha chamada a un método dun obxecto a palabra chave **this** fai referencia ao obxecto de dito método.

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

```
const ereaObxecto = {
    …,
  greet() {
    console.log(`Ola, o meu nome és ${this.nome}`);
  },
};
```

É posible **copiar** obxectos ou **fusionalos** utilizando a sintaxe **spread**, o que creará unha copia superficial do obxecto orixinal (similar á copia de arrays):

```
const obxecto1 = { foo: 'bar', x: 42 };
const obxecto2 = { foo: 'baz', y: 13 };

const clonedObj = { ...obxecto1 };
// Object { foo: "bar", x: 42 }

const mergedObj = { ...obxecto1, ...obxecto2 };   // Object { foo: "baz", x: 42, y: 13 }
```

JSON

JSON (JavaScript Object Notation) é un formato lixeiro para o intercambio de datos. É unha forma de representar obxectos nun formato simple e facilmente lexible para as persoas.

En desenvolvemento web, especialmente en JavaScript, trabállase con APIs para enviar e recibir información do servidor. Hai uns anos, XML (Extensible Markup Language) era o estándar para intercambiar información co servidor, mais actualmente é JSON o formato que se usa.

Por exemplo, se se accede á URL https://api.github.com/users (documentación) verase que devolve un array que contén os primeiros 30 usuarios de GitHub cunha sintaxe moi similar aos obxectos JavaScript. A diferencia aquí é que os nomes das propiedades dun obxecto, deben ir entre comiñas “”. Este é só un exemplo dunha API que devolve información en formato JSON, que é o formato máis usado hoxe en día en APIs.

É posible crear ficheiros con formato JSON, que ten unha sintaxe moi parecida á de obxectos de JavaScript. Exemplo de ficheiro JSON:

```
[
  { "id": 1, "nome": "DWCS" },
  { "id": 2, "nome": "DWCC" }
]
```

JavaScript proporciona os seguintes métodos para traballar con JSON e que permiten converter obxectos JavaScript a cadeas JSON e viceversa:

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

Exemplo:

```
const modulo = { id: 1, nome: 'DWCC' };

const str = JSON.stringify(modulo);
console.log(str);

const obxecto = JSON.parse(str);
console.log(obxecto);

const modulos = [
  { id: 1, nome: 'DWCC' },
  { id: 2, nome: 'DWCS' },
];

const str2 = JSON.stringify(modulos);
console.log(str2);

const obxecto2 = JSON.parse(str2);
console.log(obxecto2);
```

Estes métodos poden usarse para facer unha copia profunda (deep copy) dun array ou obxecto. 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 numeros = [1, 2, [3, 4]];

const copiaString = JSON.stringify(numeros);
const copia = JSON.parse(copiaString);

copia[2][0] = 5;
console.log(copia);
console.log(numeros);
```

NOTAS:

- 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.
- Algúns obxectos non se poden serializar (converter a string con JSON.stringify) como por exemplo funcións ou elementos do DOM.

Desestruturación de obxectos

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

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

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

const { a } = obxecto;
// equivalente a const a = obxecto.a;

console.log(a);
```

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

const { a, b } = obxecto;
// equivalente a
// const a = obxecto.a;
// const b = obxecto.b;

console.log(a, b);
```

Se as variables xa están declaradas, é necesario poñer a sentencia entre ( )

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

{ a, b } = obxecto;   // Erro de sintaxe
({ a, b } = obxecto);
```

Desestruturar obxectos aniñados:

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

const {
  displayName,
  fullName: { firstName },
} = user;

// equivalente a
// const displayName = user.displayName;
// const firstName = user.fullName.firstName;

console.log(displayName, firstName); //jdoe, John
```

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
```

Na desestruturación de obxectos pode asignarse un **valor por defecto** á propiedade, que se aplicará cando a propiedade non está presente ou ten un valor *undefined*. Este valor por defecto non se asignará no caso de que a propiedade sexa *null*.

```
const { a = 2 } = { a: '1' };
const { b = 2 } = { b: undefined }; // b is 2
const { c = 2 } = { c: null }; // c is null

console.log(a, b, c);
```

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
```

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
```

Operacións con obxectos

Cada propiedade dun obxecto en JavaScript pode ser clasificada por tres factores:

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

Poden percorrerse as propiedades dun obxecto co bucle **for…in**, que itera sobre as propiedades **enumerables** de tipo string (non símbolos) do obxecto, incluídas as herdadas.

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

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

Poden obterse as propiedades e valores dun obxecto de diferentes formas:

- Object.keys(myObj): devolve un array cos nomes das propiedades enumerables de tipo string do obxecto.
- Object.values(myObj): devolve un array cos valores das propiedades enumerables de tipo string do obxecto.
- Object.entries(myObj): devolve un array de pares [chave, valor] coas propiedades enumerables de tipo string do obxecto.
- Object.hasOwn(): devolve true se o obxecto ten a propiedade especificada como propia. Se a propiedade é herdada ou non existe, o método devolve false.

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 as seguintes variables:

   - team1: debe inicializarse co valor da propiedade team1 do obxecto inicial.
   - draw: debe inicializarse co valor da propiedade x do obxecto inicial.
   - team2: debe inicializarse co valor da propiedade team2 do obxecto inicial.

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 como propiedades o nome dos xogadores que marcaron e como valor o número de goles que marcaron respectivamente. Neste exemplo sería algo así: **{Lewandowski: 2, Gnarby: 1, Hummels: 1}**

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(['a', 'b', 'a', 'b', 'c', 'c']);
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);

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 comprobar se un elemento está no conxunto é usar o método has().

```
const set1 = new Set(['a', 'b', 'a', 'b', 'c', 'c']);
console.log(set1);

console.log(set1.has('a')); // true
```

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

```
for (const item of set1) {
  console.log(item);
}

for (const item of set1.keys()) {  // equivalente a set1.values()
  console.log(item);
}

for (const item of set1.values()) {
  console.log(item);
}
```

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

```
const numeros = [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 tamén almacena un conxunto de pares 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.

Formas de crear un mapa:

```
const map1 = new Map();

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

```
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);
```

Poden realizarse diferentes operacións nun mapa: eliminar elementos con Map.prototype.delete(), obter o valor asociado a unha chave con Map.prototype.get(), comprobar se un mapa ten unha chave con Map.prototype.has(), obter o número de elementos no mapa con Map.prototype.size, etc.

```
const map1 = new Map();

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

// eliminar elementos
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
```

Un mapa pode percorrerse usando un bucle **for…of** que mostrará os valores 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 por JavaScript.
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.

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.

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

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

Existen **propiedades globais**, ou propiedades predefinidas do obxecto global, que son accesibles directamente, xa que son engadidas automaticamente ao 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* no navegador e *global* en Node). 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:

- alert(): mostra unha ventá con unha mensaxe.
- 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.
- decodeURI(): descodifica a URI previamente codificada con encodeURI().

Todas as propiedades do obxecto global poden ser accedidas directamente, xa que son engadidas automaticamente ao ámbito global:

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

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