Profesor: Borja Rey Seoane

TEMA 2

INTRODUCIÓN A JAVASCRIPT

Módulo: **Desenvolvemento Web en Contorna Cliente**

Ciclo: **DAW**

Introdución

Como xa vimos no tema anterior, nos primeiros tempos da Web e debido á escasa velocidade das conexións a Internet, xurdiron métodos para incluír programación no cliente de xeito que se puidera modificar o contido dos documentos HTML fornecidos polos servidores Web (e interpretados polos navegadores), sen necesidade de facer máis peticións ós servidores.

Dende eses inicios até a actualidade foise facendo cada vez máis habitual o traballo no lado do cliente, para o cal xurdiron diferentes tecnoloxías de *scripting* entre as que salientamos:

- **Visual Basic Script (VBS):** linguaxe desenvolvida por Microsoft para os seus navegadores Internet Explorer, cunha sintaxe semellante á de JavaScript, pero non soportado por outros navegadores.
- **Applets Java:** pequenas aplicacións integradas nos documentos HTML e escritas en Java que permitían aproveitar a potencia da devandita linguaxe, pero con serios problemas de seguridade que remataron por significar o fin da tecnoloxía.
- **ActionScript:** desenvolvida por Adobe, esta linguaxe tiña coma obxecto fornecer capacidades interactivas aos ficheiros Flash.
- **JavaScript:** linguaxe cuxas características introducimos xa no tema anterior e da cal trataremos neste tema.

Fundamentos da Sintaxe

Para comezar manexar JS, coma calquera outra linguaxe de programación, primeiro é preciso coñecer a súa sintaxe básica (o que inclúe o seu léxico e gramática) a cal baséase parcialmente na das primeiras versións da linguaxe Java, que pertence á familia de linguaxes C. A pesares do anterior, existen notables diferenzas entre a operativa de Java e a de JavaScript, as cales iremos vendo ao longo do presente tema.

Citamos a continuación algunhas das características máis fundamentais de JS que debemos considerar dende o primeiro intre en que nos aproximamos a linguaxe, como por exemplo:

- JS distingue maiúsculas e minúsculas, polo tanto:

```
ola ≠ OLA ≠ OLa
```

- Non é preciso rematar as sentenzas con punto e coma (`;`), aínda que si é habitual facelo por motivos de claridade ao partillar código nunha mesma equipa de traballo.
- Non se consideran os espazos repetidos nin as liñas en branco (de xeito análogo ao que ocorre en HTML).

Comentarios

Os comentarios empréganse en JS, coma no resto de linguaxes de programación, para explicar o código, facéndoo máis lexible. Así mesmo, úsanse para anular provisionalmente a execución de código durante as fases de desenvolvemento e probas.

Coma xa sabemos, aquilo que incluamos dentro de comentarios será automaticamente eliminado polo intérprete do código, do mesmo xeito que se non estivese escrito no *script*.

JS posúe dúas grafías para os comentarios:

- Comentarios dunha liña, iniciarán sempre co dobre *forward slash* (`//`).

```
// Almacena a latitude GPS
var lat;
var lon; // Almacena a lonxitude GPS
```

- Comentarios multiliña, irán acoutados entre as combinacións de carácteres `/*` e `*/`.

```
/* Ás veces é preciso facer comentarios máis extensos para documentar unha parte
do código de maior complexidade sintáctica ou funcional */
```

Palabras reservadas

Coma en calquera linguaxe de programación, JS dispón dun glosario de termos reservados que non poderán ser empregados no nomeado de variables, funcións, obxectos ou outros elementos, xa que son empregados para a construción do fluxo de programa e outras instrucións propias da linguaxe posuíndo, polo tanto, un significado especial que non pode ser alienado.

Dependendo de se están xa presentes no léxico actual de JS ou non, atopamos:

- **Palabras en uso na actualidade:** `break`, `case`, `class`, `catch`, `const`, `continue`, `debugger`, `default`, `delete`, `do`, `else`, `export`, `extends`, `false`, `finally`, `for`, `function`, `if`, `import`, `in`, `instanceof`, `let`, `new`, `null`, `return`, `super`, `switch`, `this`, `throw`, `true`, `try`, `typeof`, `undefined`, `var`, `void`, `while`, `with`, `yield`.
- **Palabras reservadas para usos futuros:** `abstract`, `boolean`, `byte`, `char`, `double`, `enum`, `final`, `float`, `goto`, `implements`, `int`, `interface`, `long`, `native`, `package`, `private`, `protected`, `short`, `static`, `synchronized`, `transient`, `volatile`.

Variables

As variables son contedores de información que teñen un nome asociado e que corresponden con posicións concretas en memoria. Outro xeito de definilas sería considerar ás variables coma posicións de memoria con nome propio.

Declaración

As variables decláranse en JS mediante o uso da palabra reservada `var`, aínda que tamén podemos acompañar a súa creación das palabras `let` e `const` con outras finalidades que trataremos máis adiante.

En canto á súa nomenclatura, os nomes de variable poden conter unicamente os carácteres alfanuméricos, o dólar (`$`) e o guión baixo (_); e sumando ao anterior a restrición de que non poden comezar por un número.

Velaquí algúns exemplos de declaracións válidas de variables:

```
var nome;
var apelido1;
var $divisa;
var _telefono;
```

E tamén de declaracións inválidas de variables:

```
var codigo-postal; // Ten un carácter non permitido
var 1curso; // Comeza por número
var case; // Palabra reservada
```

Ao crear unha variable non se indica o seu tipo, senón que este é establecido no intre de asignarlle valor á mesma. Dese xeito, unha mesma variable pode almacenar diferentes tipos de datos durante a súa vida útil. No caso de non indicar un valor no intre da declaración, a variable tomará o valor `undefined`.

```
var x; // Valor de tipo especial undefined
var y = "Ola!"; // Valor de tipo texto
y = 3; // Valor de tipo numérico
var a=1, b=2;
var telefono = "982112233";
var contacto = telefono;
```

Tipos de datos

JavaScript é unha linguaxe «pouco tipada» en tanto que fai un tratamento de tipos de datos laxo, como puidemos ver no punto anterior en referencia á declaración de variables. A pesares diso, distínguense os seguintes tipos:

- **Elementais:** aqueles que non están compostos doutros tipos de datos.
  - Texto: calquera cadea de carácteres acoutados entre comiñas dobres ou simples. Ademais, se nas mesmas queremos incluír algún carácter reservado (isto é, un carácter especial que non pode ser escrito directamente na cadea en cuestión sen violar a sintaxe da linguaxe) deberemos escapalo precedéndoo do carácter do *backward slash* (`\`).

```
var nome = "Manny \'Pac-man\' Pacquiao";
var pais = 'Filipinas';
```

Na táboa seguinte vemos exemplos dos carácteres máis habitualmente escapados:

Sintaxe	Descrición
`\\`	\
`\'`	'
`\"`	"
`\n`	Salto de liña
`\t`	Tabulación horizontal
`\v`	Tabulación vertical
`\f`	Salto de páxina
`\r`	Retorno de carro
`\b`	Retroceso

  - Números: poden ser enteiros ou reais, separándose nestes últimos os decimais mediante o carácter do punto (`.`).

```
var puntos = 24;
var media = 74.25;
```

  - Booleanos: permiten os valores binarios verdadeiro (`true`) e falso (`false`).

```
var x = false;
```

  - `undefined`: valor que teñen as variables non declaradas ou inicializadas.

- **Compostos:** aqueles que están compostos por outros datos (elementais ou compostos).
    - Arrays: coleccióńs de elementos que poden ser de distintos tipos. Créanse empregando os corchetes ([]) ou mediante o método new Array().

```
var modulos = ["DWCC", "DIW", "CODE"];
var modulos = new Array("DWCC", "DIW", "CODE");
```

    - Obxectos: son estruturas compostas por un par clave-valor e un conxunto de métodos (funcións internas) asociadas aos mesmos. Créanse mediante chaves (`{}`).

```
var estudante = {nome:"Allen", apelido:"Carr", idade:58};
```

    - Funcións: construción que permite executar varias instrucións a través dunha chamada única.

```
function show(text) {console.log(text)};
```

    - `null`: emprégase para indicar un valor baleiro e compórtase coma un obxecto.

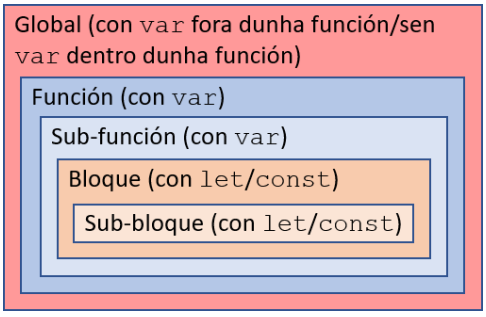
JS pon á nosa disposición un método que nos permite coñecer o tipo de datos dunha variable, e que leva por nome `typeof`:

```
typeof variable;
```

Ámbito

O ámbito das variables vai determinar a súa visibilidade dende outros puntos do programa. Isto débese a que as variables van permanecer en memoria namentres o programa estea a executar unha instrución do mesmo ámbito que a variable. Unha vez que o programa saia dese ámbito, as súas variables serán eliminadas.

En xeral, podemos considerar a visibilidade entre ámbitos en base ao seguinte condicionante: as variables que pertenzan a un ámbito concreto poderán ser accedidas desde ámbitos máis internos.



Existen tres ámbitos (*scopes*) recoñecidos na versión estable actual de JavaScript (ECMAScript 2020):

- **Global**, existen dúas posibilidades:
    - Aquelas variables que se declaran fora de funcións e precedéndoas da palabra reservada `var`.

```
var x = "global";
console.log(x); // global
function ambitoLocal() {}
```

    - Aquelas declaradas sen `var` directamente dentro dunha función.

```
localScope();
console.log(x); // global
function ambitoLocal() { x = "global"; }
```

- **Local:** aquelas variables declaradas dentro dunha función precedendo da palabra reservada `var`. Este tipo de ámbito limita o acceso ás devanditas variables ao interior da función, xa que deixan de existir co retorno da mesma.

```
localScope();
console.log(x); // Erro de referencia a variable global
function localScope() {
    var x = "local";
    console.log(x); // local
}
```

- **De bloco:** aquelas variables que se declaran dentro dunha estrutura acoutada entre chaves (por exemplo: `if`, `for`, `switch`, `while`...) e precedidas das palabras reservadas `let` ou `const`. A diferenza estriba en que, se declaramos unha variable con `const`, esta será de valor constante e, polo tanto, non poderemos modificar o seu valor despois da primeira asignación.

```
localScope();
function localScope() {
    if (true) {
        let x = "bloco";
        console.log(x); // bloco
    }
    console.log(x); // Erro de referenza a variable local
}
```

Podedes atopar máis información ao respecto dos ámbitos en JavaScript no **titorial oficial da W3C** e no **titorial oficial de MDN**.

Entrada e saída de texto

Todas as linguaxes de programación precisan dalgún tipo de instrución que permita amosar e rexistrar datos en formato texto, algo especialmente útil cando estamos a facer trazas para depuración ou cando queremos construír un *log*.

Saída de texto

JavaScript pon á nosa disposición varias alternativas a este respecto:

- Escribir na consola do navegador, a través do propio método de *log* de que dispón a mesma:

```
console.log("Ola mundo!");
```

- A través dun diálogo emerxente:

```
window.alert("Ola mundo!");
```

- Directamente inserido na parte do documento afectada polo *script*:

```
document.write("Ola mundo!");
```

- Inserindo nunha etiqueta HTML *ad hoc*:

```
document.getElementById("dous").innerHTML = "Ola mundo!";
```

Entrada de texto

Un xeito común de ingresar información vía JavaScript (empregado xeralmente durante a depuración de erros) é mediante unha xanela emerxente cunha etiqueta personalizada e un cadro de texto

```
var nome = prompt("Introduce o teu nome: ", "SEU NOME AQUÍ");
```

Operadores

Os operadores básicos soportados por JavaScript pódense ver na seguinte táboa:

Tipo	Sintaxe	Descrición
Aritméticos	`+ - * / % ++ --`	Calculan un novo valor facendo unha operación aritmética sobre 1 ou 2 operandos.
Asignadores	`= += -= *= /= %=`	Asignan o valor da dereita do operador á variable da esquerda.
Comparadores	`== === != !== > < >= <=`	Comparan o valor de 2 operandos.
Booleanos	`! && ||`	Realizan operacións lóxicas sobre 1 ou 2 operandos.
Binarios	`~ & | ^ << >>`	Realizan operacións lóxicas e de desprazamento de bits sobre representacións binarias dos operandos.
De obxectos	`delete in instanceof new this . [] ()`	Permiten traballar co tipo de datos complexo obxecto.
Misceláneos	`, ?: typeof`	Operadores non encadrados nas demais categorías.

Operadores aritméticos

Aqueles que toman un ou dous operandos e devolven o resultado de aplicarlle a operación arimética correspondente. Velaquí algúns exemplos:

Operador	Descrición	Exemplo	Resultado
`+`	Suma	`x + y`	`7`
`-`	Resta	`x - y`	`-1`
`*`	Multiplicación	`x * y`	`12`
`/`	División	`x / 2`	`1.5`
`**`	Potencia	`x ** 2`	`9`
`%`	Módulo (resto da división enteira)	`y % x`	`1`
`++`	Incremento	`x++ ou ++x`	`4`
`--`	Decremento	`x-- ou --x`	`2`
`+ (unario)`	Positivo	`+x`	`3`
`- (unario)`	Negativo	`-x`	`-3`

Operadores de asignación

Aqueles que asignan ao valor da variable á súa esquerda ben o valor situado á súa dereita ou o resultado de aplicarlle aos mesmos certas operacións aritméticas, lóxicas ou binarias. Na seguinte táboa podemos ver algúns exemplos de aplicación destes operadores:

Operador	Descrición	Exemplo	Equivalencia
=	Asignación	x = 3	
+=	Suma e asignación	x += 3	x = x+3
-=	Resta e asignación	x -=3	x = x-3
*=	Multiplicación e asignación	x *= 3	x = x*3
/=	División e asignación	x /= 3	x = x/3
%=	Módulo e asignación	x %= 3	x = x%3
**=	Potencia e asignación	x **= 3	x = x**3
<<=	Desprazamento de bits á esquerda e asignación	x <<= 3	x = x<<3
>>=	Desprazamento de bits á dereita con signo e asignación	x >>= 3	x = x>>3
>>>=	Desprazamento de bits á dereita e asignación	x >>>= 3	x = x>>>3
&=	AND lóxico e asignación	x &= 3	x = x&3
|=	OR lóxico e asignación	x |= 3	x = x|3
^=	XOR lóxico e asignación	x ^= 3	x = x^3
~	NOT lóxico e asignación	~x	

Operadores de comparación

Estes operadores realizan comparacións lóxicas entre os valores situados aos seus lados, devolvendo un valor booleano. Velaquí algúns exemplos de aplicación:

Operador	Descrición	Exemplo	Resultado
==	Igual	3 == 3 3 == 3.0 "3" == 3 "Texto" == "texto"	true true true false
===	Igual en valor e tipo	3 === 3 3 === 3.0 "3" === 3 "Texto" === "texto"	true true false false
!=	Distinto	3 != 3 3 != 3.0 "3" != 3	false false false

Operador	Descrición	Exemplo	Resultado
		`"Texto" != "texto"`	`true`

`!==`	Distinto en valor ou tipo	`3 !== 3`
		`3 !== 3.0`
		`"3" !== 3`
		`"Texto" !== "texto"`
			`false`
			`false`
			`true`
			`true`

`>`	Maior	`3 > 4`
		`"3.0" > 22`
		`"Gomez" > "Garcia"`
		`"Gomez" > "gomez"`
			`false`
			`false`
			`true`
			`false`

`<`	Menor	`3 < 4`
		`"3.0" < 22`
		`"Gómez" < "García"`
		`"Gómez" < "gómez"`
			`true`
			`true`
			`false`
			`true`

`>=`	Maior ou igual	`3 >= 4`
		`"3.0" >= 22`
		`"Gomez" >= "Garcia"`
		`"Gomez" >= "gomez"`
			`false`
			`false`
			`true`
			`false`

`<=`	Menor ou igual	`3 <= 4`
		`"3.0" <= 22`
		`"Gomez" <= "Garcia"`
		`"Gomez" <= "gomez"`
			`true`
			`true`
			`false`
			`true`

`?`	Operador ternario	`(1>9)? x=3:x=4`	`x=4`

Operadores booleanos

Aqueles que permiten realizar operacións lóxicas sobre un ou dous operandos, producindo coma resultado un valor booleano. Velaquí algúns exemplos da súa aplicación práctica:

Operador	Descrición	Exemplo	Resultado
`&&`	AND lóxico	`(3<4) && (4>5)`	`false`
`||`	OR lóxico	`(3<4) || (4>5)`	`true`
`!`	NOT lóxico	`!(3<4)`	`false`

Podemos ver tamén as táboas de verdade das operacións anteriores a continuación:

x	y	x && y	x || y	!x
`false`	`false`	`false`	`false`	`true`
`false`	`true`	`false`	`true`	`true`
`true`	`false`	`false`	`true`	`false`
`true`	`true`	`true`	`true`	`false`

Operadores binarios

Nesta categoría atopamos aqueles operadores que realizan operacións a nivel de bit empregando operandos binarios con signo e unha lonxitude de 32 bits. Na seguinte táboa vemos algúns exemplos:

Operador	Descrición	Exemplo	Resultado
`&`	AND binario	`3 & 5 (0011 & 0101)`	`0001 (1)`
`|`	OR binario	`3 | 5 (0011 | 0101)`	`0111 (7)`
`^`	XOR binario	`3 ^ 5 (0011 & 0101)`	`0110 (6)`
`~`	NOT binario	`~3 (~0011)`	`1100 (12)`
`<<`	Desprazamento á esquerda enchendo con 0	`3 << 1 (0011 << 1)`	`0110 (6)`
`>>`	Desprazamento á dereita con signo	`5 >> 1 (0101 >> 1)`	`0010 (2)`
`>>>`	Desprazamento á dereita enchendo con 0	`5 >>> 1 (0101 >>> 1)`	`0010 (2)`

Na seguinte táboa podemos ver o comportamento completo destes operadores:

x	y	x & y	x | y	x ^ y	!x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Operadores de obxectos

Aqueles que permiten traballar sobre o tipo de datos composto «object». A continuación algúns exemplos dos mesmos:

Operador	Descrición	Exemplo
`.`	Permite acceder ás propiedades/métodos dun obxecto	`dni.numero;`
`[]`	Permite enumerar membros dun obxecto	`dni["letra"] = "K"`
`delete`	Elimina un elemento dun obxecto	`delete dni.letra`
`in`	Inspecciona propiedades dun obxecto	`"letra" in dni`
`instance of`	Comproba se o tipo dun operando se corresponde co dun obxecto nativo	`dni instanceof Object`
`new`	Permite crear obxectos nativos da linguaxe	`new Date()`

Operador	Descrición	Exemplo
`this`	Accede á referencia do propio obxecto	`this.toString()`

O operador de propagación

Existe en JS un operador especial, coñecido coma **operador de propagación** ou ***spread***, que permite realizar operacións «especiais» sobre *arrays* e obxectos.

A continuación vemos os distintos usos que podemos darlle:

- **Combinación de *arrays*:** permite concatenar dous ou máis *arrays* nun novo *array* que disporá dunha copia dos datos dos orixinais.

```
let arr1 = [1, 2, 3];
let arr2 = [4, 5, 6];
let combinado = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]
```

- **Copia de *arrays*:** permite realizar unha copia «por valor» dun *array*.

```
let arr = [1, 2, 3];
let copia = [...arr]; // [1, 2, 3]
```

- **Combinar ou estender obxectos:** permite unir dous ou máis obxectos, combinando as propiedades de nome distinto e sobreescribindo as compartidas en orde de lectura.

```
let obx1 = { a: 1, b: 2 };
let obx2 = { a: 5, c: 3, d: 4 };
let combinado = { ...obx1, ...obx2 }; // { a: 5, b: 2, c: 3, d: 4 }
```

- **Copia de obxectos:** permite realizar unha copia «por valor» dun obxecto.

```
let obx = { a: 1, b: 2 };
let copia = { ...obx }; // { a: 1, b: 2 }
```

- **Como operador de resto (*rest operator*):** permite convertir nun *array* o conxunto de parámetros pasados a unha función.

```
function suma(...numeros) {
  return numeros.reduce((total, actual) => total + actual, 0);
}
suma(1, 2, 3, 4); // 10
```

- **Desestruturar *arrays*:** permite asignar nome ás distintas partes dun *array* por eliminación, coma se de variables se tratase.

```
let [a, b, ...resto] = [1, 2, 3, 4, 5];
console.log(a); // 1
console.log(b); // 2
console.log(resto); // [3, 4, 5]
```

- **Desestruturar obxectos:** permite asignar nome ás distintas partes dun *array* por eliminación, coma se de variables se tratase.

```
let { x, y, ...resto } = { x: 1, y: 2, z: 3, a: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(resto); // { z: 3, a: 4 }
```

Precedencia de operadores

É importante, ao tempo de traballar cos operadores vistos ao longo dos puntos anteriores, coñecer a precedencia que teñen, isto é, a orde na que serán executados cando aparecen no mesmo nivel dunha mesma operación.

No tempo de decidir a orde de execución dos operadores observaremos as seguintes normas:

- No caso de existir parénteses, o que vaia dentro dos mesmos primará sobre o que vaia fora.
- No caso de que os operadores teñan a mesma precedencia, a orde de aplicación será a de lectura (de esquerda a dereita).
- Finalmente, observaremos o establecido na seguinte táboa:

Operadores	Descrición	Exemplo
`()`	Agrupación de expresións	`3 + (4 + "5")`
`. []`	Acceso a membros dun obxecto	`dni.letra` `dni["letra"]`
`++ --`	Incremento/decremento unario	`3++`
`!`	NOT lóxico	`!(3>4)`
`**`	Potencia	`3 ** 4`
`* / %`	Multiplicación, división e módulo	`3 / 2`
`+ -`	Suma e resta	`3 + 4`
`<< >> >>>`	Desprazamento de bits	`3 >> 4`
`< <= > >=`	Comparación con maior/menor que	`4 >= 3`
`== === != !==`	Comparación só de igualdade	`"3" == 3`

Operadores	Descrición	Exemplo
`&`	AND binario	`3 & 5`
`^`	XOR binario	`3 ^ 5`
`|`	OR binario	`3 | 5`
`&&`	AND lóxico	`true && false`
`||`	OR lóxico	`true || false`
`+= -= *= /= %= <<= >>= >>>= &= ^= |=`	Asignación con operador	`x += 3`

Conversión de tipos de datos

Para determinar o dato dunha variable ou dun valor literal, JavaScript dispón do operador `typeof`, que xa tratamos con anterioridade. No seguinte exemplo podemos ver o seu funcionamento:

```
typeof "Diego" // string
typeof 3.14 // number
typeof NaN // number
typeof true // boolean
typeof [1,2,3,4] // object (un array é un tipo obxecto)
typeof {nome:'Diego', idade:28} // object
typeof new Date() // object (unha data é un tipo obxecto)
typeof function () {} // function
typeof myCar // undefined
typeof null // object (o valor Null é un tipo obxecto)
```

En JavaScript, as variables ou valores literais poden cambiar dinamicamente o seu tipo de 2 xeitos:

- Automaticamente polo motor de interpretación da linguaxe.
- Polo uso expreso dunha función para tal efecto.

Conversión automática

Cando empregamos operadores con operandos de tipos distintos, o motor de interpretación converte ao tipo máix axeitado para poder realizar a operación en cuestión:

- **Operador de *strings* (+):** este operador pode ser aritmético ou de concatenación de *strings*, pero cando un dos operados é unha cadea de carácteres prevalece o seu tipo.

```
3 + "4" // 34
3 + 4 + "5" // 75 (o primeiro + fai unha suma aritmética)
"" + 3 // "3"
"3" + true // "3true"
```

- **Operadores aritméticos (+ - * / %):** convértense (cando sexa posible) os operandos nas súas cifras equivalentes.

```
"3" * 2 // 6
3 - true // 2 (true equivale a 1)
3 + true // 4
+"4" // 4 (co operador unario + tamén se converten as cadeas de carácteres en
números)
```

- **Operadores relacionais (> >= < <=):** convértense, coma no caso anterior e sempre e cando sexa posible, os operandos nas súas cifras equivalentes.

```
"3" > 4 // false
3 > true // true
```

Conversión a texto

Ademais da conversión automática mediante o operador de concatenación, como vimos no punto anterior, existen outras dúas formas de converter outros tipos de datos a texto. Vexamos:

- **toString():** este método é aplicable a calquera tipo de datos e, de feito, é o que é chamado por defecto cando visualizamos calquera tipo.

```
(3 + 4).toString(); // 7
new Date().toString(); // Mon Oct 14 2019 12:13:27 GMT+0200
console.log(true); // true
```

- **Co construtor do obxecto string():** directamente construíndo un novo obxecto de tipo cadea de carácteres.

```
new String(3 + 4); // 7
new String(new Date()); // Mon Oct 14 2019 12:13:27 GMT+0200
new String(true); // true
```

Conversión a número

Tamén para converter outros tipos de datos a número dispoñemos de varias opcións:

- **parseInt(valor)** e **parseInt(valor, base):** este método converte o valor recibido (que pode ser unha cadea de carácteres ou un número) nun número enteiro. Temos a posibilidade de incluír un segundo parámetro que indica a base na que está expresado o noso número

```
parseInt(3.14); // 3
parseInt('-F', 16); // -15
parseInt(4.7, 10); // 4
```

- **parseFloat(valor)** e **parseFloat(valor, base):** de xeito paralelo ao caso anterior, este método converte o valor recibido (que pode ser unha cadea de carácteres ou un número) nun número real. Máis unha vez, temos a posibilidade de incluír un segundo parámetro que indica a base na que está expresado o noso número

```
parseFloat(3.14); // 3.14
parseFloat(4.7, 10); // 4.7
```

- **Co construtor do obxecto Number():** directamente creando un novo obxecto de tipo número.

```
Number(3.14); // 3.14
```

Na seguinte táboa pode verse o resultado da súa aplicación sobre distintos valores e tipos de datos:

Valor	Number()	parseInt()	parseFloat()
"3.14"	3.14	3	3.14
"5.8"	5.8	5	5.8
" "	0	NaN	NaN
""	0	NaN	NaN
"3.14 16"	NaN	3	3.14
new Date()	1571049671687	NaN	NaN
true	1	NaN	NaN
null	0	NaN	NaN

Conversión a booleano

En JavaScript pódense converter outros tipos de datos a booleano mediante o construtor do obxecto **Boolean**, do xeito que podemos ver nos exemplos seguintes:

```
Boolean(3.14); // true
Boolean(""); // false
```

Na seguinte táboa pódense ver os valores cos que se obterá true ou false:

Boolean()	Number	String	Outros
false	0	""	null undefined
true	valores ≠ 0	valores ≠ ""	-

En resumo, na seguinte táboa pódese ver o resultado de realizar conversións entre os tipos numéricos, de cadea de carácteres e booleano:

Valor	Number	String	Boolean
false	0	"false"	
true	1	"true"	
0		"0"	false
1		"1"	true
"0"	0		true
"000"	0		true
"1"	1		true
NaN		"NaN"	false
Infinity		"Infinity"	true
-Infinity		"-Infinity"	true
""	0		false
"3"	3		true
"tres"	NaN		true
[]	0	""	true
[3]	3	"3"	true
[3,4]	NaN	"3,4"	true
["tres","catro"]	NaN	"tres,catro"	true
function(){}	NaN	"function(){}"	true
null	0	"null"	false
undefined	NaN	"undefined"	false
{}	NaN	"[object Object]"	true

Estruturas de control

Neste capítulo aprenderemos traballar coas estruturas de JavaScript que permiten controlar o fluxo de execución dun programa, nomeadamente, as estruturas condicionais e os bucles.

Estruturas condicionais

Este tipo de estruturas permiten executar uns ou outros blocos de programa en base ao cumprimento de condicións lóxicas preestablecidas.

A estrutura `if`

Na súa versión máis simple, esta estrutura permite decidir se executamos un bloco de código en base ao cumprimento dunha condición, é dicir, se a condición é certa execútase o código, en caso contrario non.

```
if (condición) {
  // Bloco de programa
}
```

Exemplo:

```
if (idade >= 18) {
  console.log("Pode votar");
}
```

A condición avaliada na cabeceira da estrutura pode ser simple ou composta (para o cal empregamos os operadores lóxicos vistos con anterioridade), pero sempre deberá producir coma resultado un valor booleano:

```
if (idade >= 18 && capacitado == true) {
  console.log("Pode votar");
}
```

A estrutura `if-else`

É unha variante da anterior que engade un segundo bloco de programa que será executado no caso de que a condición avaliada resultara ser falsa. A súa sintaxe é coma segue:

```
if (condición) {
  // Bloco de programa (se a condición é verdadeira)
} else {
  // Bloco de programa (se a condición é falsa)
}
```

Exemplo:

```
if (idade >= 18) {
  console.log("Pode votar");
} else {
  console.log("Non pode votar");
}
```

Deste xeito asegurarase sempre que o fluxo do programa entrará no `if` e executará algún dos blocos de instrucións.

A estrutura if-else conta, en JavaScript, cunha construción abreviada que posúe a sintaxe seguinte:

```
(condición) ? // Bloco (verdadeiro): // Bloco (falso)
```

Vexamos a forma do exemplo anterior expresada mediante esta construción:

```
(idade >= 18) ? console.log("Pode votar") : console.log("Non pode votar");
```

A estrutura `if-else-if`

Esta estrutura permite especificar varias condicións a avaliar cos seus correspondentes blocos de programa no caso de seren verdadeiras. Así, no caso de avaliarse unha condición e esta resultar ser falsa, pasa a avaliarse a seguinte. A súa sintaxe xeral é coma segue:

```
if (condición1) {
  // Bloco de programa (se condición1 é verdadeira)
} else if (condición2) {
  // Bloco de programa (se condición2 é verdadeira)
} else if (condición3) {
  // Bloco de programa (se condición3 é verdadeira)
} else if …
} else {
  // Bloco de programa (se todas as condicións foron falsas)
}
```

Exemplo:

```
if (nota >= 9) {
  console.log("Sobresaínte");
} else if (nota >= 7) {
  console.log("Notable");
} else if (nota >= 6) {
  console.log("Ben");
} else if (nota >= 5) {
  console.log("Suficiente");
} else {
  console.log("Suspenso");
}
```

Dado que o uso da estrutura anterior pode dificultar a lectura do código cando se inclúen demasiadas condicións, recoméndase empregar en tales casos a estrutura `switch`, que veremos a continuación.

A estrutura `switch`

A estrutura `switch` permite executar un bloco de código entre varios dispoñibles en base ao resultado dunha expresión. A súa sintaxe xeral é:

```
switch (expresión) {
  case valor1:
    // Bloco de programa (se expresión vale valor1)
    break;
  case valor2:
    // Bloco de programa (se expresión vale valor2)
    break;
    …
  default: // Esta parte é opcional
    // Bloco de programa (en calquera outro caso)
}
```

O seu funcionamento toma os seguintes pasos:

- Avalíase a expresión unha única vez.
- Faise unha comparación de igualdade estrita (=== deben coincidir tanto valor coma tipo) do valor da expresión cos valores de cada `case`.
- Éntrase no primeiro `case` para o que haxa coincidencia e execútase a secuencia de instrucións até atopar un `break`.

Exemplo:

```
switch (new Date().getDay()) {
  case 6:
    dia = "Sábado";
    break;
  case 0:
    dia = "Domingo";
    break;
}
```

No remate de cada bloco de cada `case` o habitual (que non obrigado) é colocar un `break`, que vai provocar a saída do `switch`. En caso de non poñer o `break`, continuarase avaliando os case seguintes, existindo polo tanto a posibilidade de executar múltiples blocos de programa cun mesmo `switch`.

```
switch (new Date().getDay()) {
  case 6:
  case 0:
    dia = "Fin de semana";
    break;
}
```

Por último, o bloco `default` especifica o código a executar se a expresión non coincide con ningún dos valores especificados nos distintos `case`. Non ten por que ir obrigadamente no final do `switch` (aínda que é habitual facelo así) e só pode haber un por estrutura.

```
switch (new Date().getDay()) {
  case 6:
  case 0:
    dia = "Fin de semana";
    break;
  default:
    dia = "Laborable";
}
```

Bucles

As estruturas de repetición fornecen un mecanismo para a execución múltiple dun mesmo bloco de programa. O número de iteración pode estar prefixado ou determinado polo cumprimento (ou non) dunha condición o cal, ao seu tempo, pode ser modificado durante a execución do bucle.

O bucle `while`

Este tipo de estrutura permite executar instrucións de xeito repetido namentres se cumpra unha condición lóxica (é dicir, namentres sexa verdadeira). A devandita condición será avaliada no comezo de cada iteración. Se operamos con variables dentro do bucle, estas deben ser inicializadas antes do inicio do mesmo.

É imprescindible facer as modificacións oportunas para que nalgún intre a condición acade o valor `true`, pois en caso contrario produciremos un bucle infinito que pode chegar saturar o motor do navegador Web.

A continuación vemos a sintaxe xeral deste tipo de bucle:

```
while (condición) {
  // Bloco de programa (repite se a condición é verdadeira)
}
```

Eis un exemplo de impresión no rexistro do navegador dos primeiros dez números pares positivos:

```
var i=1;
while (i<=10) {
  console.log(i*2);
  i++; // Incrementamos contador para asegurar que o bucle remata
}
```

O bucle do-while

Este bucle, por contraposición ao anterior, avalía a súa condición ao remate do bloco de programa a repetir, polo que vai executarse sempre coma mínimo unha volta. Rematará en canto a condición avaliada sexa falsa.

Aquí temos a sintaxe xeral deste bucle en JS:

```
do {
  // Bloco de programa (repite se a condición é falsa)
} while (condición);
```

Velaquí temos un exemplo no que, coma no caso anterior, amosamos os primeiros dez pares positivos:

```
var i=1;
do {
  console.log(i*2);
  i++; // Incrementamos contador para asegurar que o bucle remata
} while (i<10);
```

O bucle `for`

Este bucle diferénzase dos anteriores principalmente na súa estrutura, algo máis ríxida e na que se determinan tanto os valores iniciais das variables que compoñen a condición, como a comprobación da condición e a modificación dos iteradores, todo na cabeceira do bucle.

Vemos no seguinte fragmento de código a sintaxe xeral dun bucle `for` en JS:

```
for (inicialización; condición; modificación) {
  // Bloco de programa (a repetir namentes a condición é certa)
}
```

Onde interpretamos cada parte como segue:

- **Inicialización:** sentenzas executada unha sola vez, denantes comezar iterar o bucle. Xeralmente emprégase para establecer o valor da variable que vai actuar a xeito de contador, determinando o número de iteracións do bucle.
- **Condición:** expresión lóxica cuxo valor determina a continuidade do bucle (isto non quita da posibilidade de que o mesmo poida ser interrompido mediante `break`).
- **Modificación:** sentenza a executar cada vez que finaliza o bloco de sentenzas a repetir (é dicir, no remate de cada iteración do bucle). Polo xeral é empregada para modificar o valor da variable contador de xeito que aseguremos que a condición chegue ser falsa nalgún punto.

Velaquí un exemplo de uso deste bucle:

```
for (let i=1; i<=10; i++) {
  console.log(i*2);
}
```

Do mesmo xeito, esta sintaxe do bucle admite formas alternativas nas que a inicialización das variables realízase denantes coemzar o bucle, e tamén onde a modificación da variable contadora é realizada dentro do corpo do bucle:

```
let i=1; // Inicialización
for (;i<=10;) {
  console.log(i*2);
  i++; // Modificación
}
```

Un exemplo de uso habitual dos bucles for é o percorrido de coleccións como, por exemplo, arrays:

```
var ciclos = ["SMR","DAW", "DAM", "ASIR"];
for (let i=0; i<ciclos.length; i++) {
  console.log(ciclos[i]);
}
```

O bucle `for-in`

Trátase dunha construción abreviada pensada para percorrer os elementos dunha colección. O código dentro do bucle executarase automaticamente unha vez por cada elemento da colección, existindo unha variable que actuará coma contador, recollendo os valores precisos para indexar os elementos da colección.

Velaquí o percorrido dun array mediante este tipo de bucle:

```
var ciclos = ["SMR","DAW", "DAM", "ASIR"];
for (let i in ciclos) {
  console.log(ciclos[i]);
}
```

O bucle `for-of`

Esta estrutura, tamén pensada para o percorrido de coleccións, actúa de xeito semellante ao anterior pero extraendo automaticamente cada elemento da colección directamente á variable definida da declaración do bucle, isto é, non hai variable contadora xa que a colección vaise percorrendo paso a paso en cada iteración até o seu remate.

Máis unha vez, temos o percorrido dos valores dun array:

```
var ciclos = ["SMR","DAW", "DAM", "ASIR"];
for (let i of ciclos) {
  console.log(i);
}
```

Mecanismos auxiliares

Existen unha serie de mecanismos auxiliares que podemos combinar co uso de bucles de xeito que gañemos versatilidade e potencia no código.

`break`

O uso desta palabra reservada (xa introducida lenemente con anterioridade) implica a saída total do bucle actual no intre que é executada.

```
var i=1;
while (i<10) {
  if(i==4)
    break;
  console.log(i*2); // Non chega amosar o dobre de 4
  i++;
}
```

`continue`

Esta palabra reservada permite que o fluxo de programa salte ao principio da seguinte iteración do bucle sen executar as instrucións que hai a continuación nesa iteración.

```
var i=0;
while (i<=10) {
  i++; // Imprescindible incrementar i e facelo antes do continue
  if(i==4)
    continue;
  console.log(i*2);
}
```

Bucles aniñados

Trátase dun recurso moi empregado para a resolución de problemas. Por exemplo, para acceder a estruturas de datos bidimensionales tales coma táboas.

```
let i=1;
while (i<=5) {
  let j=1;
  let fila = "";
  while (j<10) {
    fila += i*j + "\t";
    j++;
  }
  console.log(fila);
  i++;
}
```

Vémolo cun exemplo que permite amosar as táboa de multiplicar dos primeiros cinco números naturais:

```
for (let i=1; i<=5; i++) {
  let fila="";
  for (let j=1; j<10; j++) {
    fila += i*j+"\t";
  }
  console.log(fila +"\n");
}
```

Nestes casos hai que ter un coidado especial co uso das variables contador de cada bucle para que non haxa accesos indesexados dende outros contextos que provoquen resultados non esperados e difíciles de detectar.

Obxectos nativos

Os obxectos nativos de JavaScript permiten almacenar os datos máis empregados en estruturas non elementares, de xeito que incorporan así mesmo métodos para traballar con eles que outorgan maior potencia ao código incrementando o abano de posibilidades.

`String`

Este obxecto permite almacenar datos de tipo texto e fornece numerosos métodos para traballar cos mesmos. Internamente, consiste nunha secuencia de cero ou máis caracteres pechados entre comiñas, simples (`'`) ou dobles (`"`), mais sempre de xeito consistente.

```
var name = "Allen"; // Variable de tipo string
var surname = 'Carr'; // Variable de tipo string
var phone = new String('600102030'); // Variable de tipo obxecto
```

NOTA: en JS non é recomendable crear as *strings* coma obxectos, xa que isto ralentiza a execución do código ao tempo que facilita a aparición de resultados inesperados, por exemplo, na comparación.

Pódese acceder a cada caracter individualmente empregando corchetes (`[]`) para indicar a posición concreta, tendo en conta que o primeiro elemento posuirá sempre o índice 0, namentres que o último elemento terá coma índice a lonxitude da cadea menos 1.

```
var name = "Allen Carr";
name[0] + ". " + name[6] + "."; // A. C.
```

As *strings* atópanse entre as estruturas máis importantes nas linguaxes de programación modernas (xunto cos tipos numéricos) dada a súa ubicuidade, e por iso é preciso coñecer e saber empregar as propiedades e métodos máis habituais das mesmas.

Propiedades

Este obxecto conta só cunha única propiedade, sendo:

- **`length`:** devolve o número de caracteres da *string*. A cadea de caracteres baleira ("") terá sempre lonxitude 0.

```
"Allen".length; // 5
```

Métodos

O obxecto string conta con máis de trinta métodos propios, entre os que salientamos os seguintes:

- **`concat()`:** permite concatenar dúas ou máis *strings*, sendo funcionalmente equivalente ao operador +.

```
var name = "Allen Carr";
name[0].concat(". ") + name[6].concat("."); // A. C.
```

- **`trim()`:** elimina os espazos en branco (" ") ou tabuladores ("\t") que poida haber ao comezo e/ou remate dunha cadea de carácteres.

```
var name = "\tAllen Carr  ";
"[" + name + "]"; // [   Allen Carr  ]
"[" + name.trim() + "]"; // [Allen Carr]
```

- **`toUpperCase()`, `toLowerCase()`, `toLocaleUpperCase()`, `toLocaleLowerCase()`:** permiten convertir textos entre maiúsculas e minúsculas, podendo tamén considerar a configuración de idioma do navegador web (cos métodos Locale).

```
var name= "Allen Carr";
name.toLowerCase(); // allen carr
name.toLocaleUpperCase(); // ALLEN CARR
```

- **`charAt()`, `charCodeAt()`:** devolven o carácter ou o valor Unicode do carácter que está nunha posición concreta da *string*.

```
"Allen Carr".charAt(0); // A
"Allen Carr".charCodeAt(0); // 65
```

- **`fromCharCode()`:** converte un valor Unicode nun caracter.

```
String.fromCharCode(65); // A
```

- **`includes()`, `indexOf()`, `lastIndexOf()`:** permiten procurar a ocorrencia dun texto dentro doutro, tendo en conta maiúsculas e minúsculas na comparación. Devolven respectivamente un valor booleano, a posición da primeira ocorrencia e a posición da última ocorrencia.

```
if ("Allen Carr".includes("Car")) {
  "Allen Carr".indexOf("Car"); // 6
}
```

- **`substr()`, `substring()`, `slice()`:** extraen unha parte da *string* especificando a posición de inicio e o número de carácteres ou a posición de remate.

```
"Allen Carr".substr(3,5); // en Ca
"Allen Carr".substring(3,7); // en C
"Allen Carr".slice(3,7); // en C
```

- **`localeCompare()`:** permite comparar dúas *strings* tendo en conta a configuración de idioma do navegador web. Devolve os valores -1, 0 ou 1 en función de se, nunha orde alfabética, a cadea de referencia é anterior, igual ou posterior á cadea pasada como parámetro.

```
"Carr, Allen".localeCompare("Carr, Jimmy"); // -1
"Carr, Allen".localeCompare("Carr"); // 1
"Carr, Allen".localeCompare("CARR, ALLEN"); // -1
"Carr, Allen".toUpperCase().localeCompare("CARR, ALLEN"); // 0
```

As *strings*, coma xa sabemos, tamén poden compararse mediante os operadores de comparacion (== e ===), tendo en conta que, se se declaran con `new`, serán obxectos (polo tanto non do tipo simple *string*) e que dous obxectos nunca serán iguais aínda que conteñan os mesmos valores.

```
"Allen" == new String("Allen"); // true
"Allen" === new String("Allen"); // false
new String("Allen") == new String("Allen"); // false
```

- **`split()`:** separa unha *string* en cachos máis pequenos en función dun ou varios carácteres separadores, devolvendo un *array* coas *substrings* resultantes. Se o caracter separador é a cadea baleira (""), devolverá un *array* cuxos elementos serán cada carácter da *string* orixinal.

```
"Lorem ipsum dolor sit".split(" "); // Array(4) ["Lorem", "ipsum", "dolor", "sit"]
"Lorem".split(""); // Array(5) ["L", "o", "r", "e", "m"]
```

- **`search()`, `match()`, `replace()`:** estes métodos permiten procurar unha secuencia de carácteres ou expresión regular. O último deles posibilita ademais substituír as ocorrencias por outro *substring*. Devolven respectivamente a posición da primeira ocorrencia, un array coa primeira ocorrencia (ou con todas) e unha *string* coa primeira ocorrencia (ou con todas) substituída.

```
"LoRem LOreM ipsum rem".search("re"); // 8
"Lorem LOreM ipsum rem".match(/re/g); // Array(3) ["re", "re", "re"]
"LoRem LOreM ipsum rem".replace("re","*"); // LoRem LO*M ipsum rem
"LoRem LOreM ipsum rem".replace(/re/g,"*"); // LoRem LO*M ipsum *m
```

- **`toString()`, `valueOf()`:** ámbolos dous métodos devolven o valor do obxecto *string* coa diferencia de que o segundo deles devolve o valor primitivo do mesmo.

```
new String("Lorem"); // String {[[PrimitiveValue]]: "Lorem"}
new String("Lorem").toString(); // Lorem
new String("Lorem").valueOf(); // Lorem
typeof(new String("Lorem").valueOf()); // string
```

`Date`

Os obxectos de tipo `Date` representan un intre concreto no tempo nun formato independente da plataforma, de xeito que cada data corresponde ao número (mediante un obxecto `Number`) de milisegundos transcorridos dende o 1 de xaneiro de 1970 (o ano cero UTC).

No tempo de visualizar unha data contida nun obxecto deste tipo, vaise empregar a zona horaria establecida no navegador:

```
Tue Oct 26 2021 11:52:03 GMT+0100 (hora estándar de Europa central)
```

Creación

Existen catro xeitos distintos de crear estes obxectos, todos eles mediante a palabra reservada `new`, pero cun construtor no que muda o número de argumentos:

- **`new Date()`:** se non se lle pasan parámetros ao construtor, gardarase a data/hora actual.

```
new Date(); // Tue Oct 26 2021 11:52:03 GMT+0100
```

- **`new Date(milliseconds)`:** máis unha forma de crear unha data é pasando o número de milisegundos transcorridos dende a data de referencia (Thu Jan 01 1970 01:00:00). O devandito número pode ser positivo ou negativo, para representar unha data posterior ou anterior á data de referencia.

```
new Date(1000); // Thu Jan 01 1970 01:00:01 GMT+0100
```

- **`new Date(dateString):`** permite especificar a data e a hora (opcional) en forma de texto. Existen 3 formatos posibles, dos cales o recomendable é o primeiro deles, en tanto corresponde ao estándar internacional:
    - Data ISO: emprega o formato «`YYYY-MM-DDTHH:hh:ssZ`»[1].
    - Data curta: emprega o formato «`MM/DD/YYYY HH:mm:ss`».
    - Data longa: emprega o formato «`MMM DD, YYYY HH:mm:ss`».

```
new Date("2021-10-26"); // Tue Oct 26 2021 01:00:00 GMT+0100
```

- **`new Date(year, month, day, hours, minutes, seconds, milliseconds):`** permite especificar unha data/hora concreta por medio de argumentos numéricos, dos cales os cinco últimos son opcionais[2]:

```
x = new Date(2021, 10, 26, 15, 30); // Tue Oct 26 2021 15:30:00 GMT+0100
```

Métodos

Coma ocorría no caso de `String`, o obxecto `Date` dispón dunha grande variedade de métodos dos cales só trataremos aquí os de uso máis habitual, agrupándoos ademais de tres categorías.

Getters/Setters

Nesta primeira categoría atopamos aqueles métodos de `Date` que permite obter (se comezan por get) ou establecer (se comezan por set) o valor dunha data ou dalgunha das súas partes. Entre eles salientamos os seguintes:

- **`getFullYear():`** devolve o valor do ano.
- **`setFullYear():`** establece o valor do ano.
- **`getMonth():`** devolve o valor do mes.
- **`setMonth():`** establece o valor do mes.
- **`getDate():`** devolve o día do mes.
- **`setDate():`** establece o día do mes.
- **`getDay():`** devolve o día da semana[3].
- **`setDay():`** establece o día da semana.
- **`getHours():`** devolve a hora[4].

[1] O Z implica que a data será representada en formato UTC, no caso de non estar presente sería en formato *Local Time*.

[2] Hai que ter en conta que os meses comezan numerar en 0 para evitar confusións.

[3] Onde o domingo é o 0 e o sábado o 6.

[4] En formato 24 horas.

- **`setHours()`:** establece a hora.
- **`getMinutes()`:** devolve o minuto.
- **`setMinutes()`:** establece o minuto.
- **`getSeconds()`:** devolve o segundo.
- **`setSeconds()`:** establece o segundo.
- **`getMilliseconds()`:** devolve os milisegundos.
- **`setMilliseconds()`:** establece os milisegundos.
- **`getTime()`:** especifica o número de milisegundos transcorridos dende a data de referencia.
- **`setTime()`:** establece a data a partir do número de milisegundos transcorridos dende a data de referencia.

```
var d = new Date();
var msg = "Estamos en " + d.getFullYear() + ". Son as " + d.getHours() + ":" +
d.getMinutes() + " horas."; // Estamos en 2021. Son as 11:21 horas.
```

Para todos os métodos anteriores existe unha versión que permite obter e establecer os valores en formato universal, engadindo UTC xusto despois de `get/set` no nome do método. Por exemplo: `getUTCTime()`.

Saída como texto

Dado que posiblemente o uso máis habitual do obxectos `Date` sexa o de simplemente amosar unha data, JS pon á nosa disposición unha serie de métodos que permiten formatar o valor destes obxectos en *strings*, podende escoller a cantidade de información que queremos presentar. Temos así:

- **`toDateString()`:** devolve unha cadea de texto coa data (sen hora).
- **`toLocaleDateString()`:** devolve unha cadea de texto coa data (sen hora) tendo en conta a configuración local.
- **`toTimeString()`:** devolve unha cadea de texto coa hora.
- **`toLocaleTimeString()`:** devolve unha cadea de texto coa hora tendo en conta a configuración local.

```
var d = new Date();
var msg = "Hoxe é " + d.toLocaleDateString() + ". Son as " +
d.toLocaleTimeString() + " horas."; //Hoxe é 2021-10-26. Son as 10:16:53 AM horas.
```

- **`toString()`:** devolve unha cadea de texto coa data completa.
- **`toLocaleString()`:** devolve unha cadea de texto coa data completa tendo en conta a configuración local.
- **`toUTCString()`:** devolve unha cadea de texto coa data completa en formato UTC.

- **`toISOString():`** devolve unha cadea de texto coa data completa en formato estendido ISO 8601.
- **`toJSON():`** devolve unha cadea de texto coa data completa no mesmo formato ca anterior, pero nunha *string* JSON.

```
var d = new Date();
var msg = d.toString(); // Tue Oct 26 2021 10:19:35 GMT+0100
msg = d.toUTCString(); // Tue, 26 Oct 2021 09:21:05 GMT
msg = d.toLocaleString(); // 2021-10-26 10:20:36 AM
```

Saída como número

Máis un xeito de obter o valor dunha data, e polo xeral moito máis útil no tempo de realizar cálculos e comparacións sobre a mesma, é obter o número de milisegundos transcorridos dende a data de referencia e operar sobre o mesmo. Para tal fin dispoñemos, entroutros, dos seguintes métodos:

- **`valueOf():`** amosa o valor do obxecto desde o cal é chamado o método, que xa estará construído cunha data concreta.
- **`now():`** presente o valor correspondente ao intre actual.
- **`parse():`** indica o valor dunha data pasada coma texto.
- **`UTC():`** devolve o valor dunha data indicando como número o ano, mes, día, hora, minuto, segundo, milisegundo.

```
// Dous xeitos de obter o mesmo resultado (nótese o cambio no número do mes)
Date.parse("2019-11-20"); // 1574208000000
Date.UTC(2019,10,20); // 1574208000000
```

Number

Este obxecto permite almacenar datos de tipo numérico, xa sexa en notación enteira, de punto decimal, científica ou hexadecimal; sen diferenzar entre os distintos tipos.

```
var x = 3.14;
var x = 3;
var x = 3.14e3; // 3140
var x = 0xFF; // 255
var x = new Number(3.14);
```

NOTA: non deben crearse números coma obxectos se non é completamente imprescindible, xa que o seu procesamento introducide problemas nas comparacións e é notablemente máis lento có daqueles tipos simples equivalentes.

Independentemente de se tratamos de números enteiros ou decimais, JavaScript almacena os mesmos internamente nun formato de coma aboiante de dobre precisión de 64 bits, nos que á base correspóndelle os primeiros 52 bits, ao expoñente os 11 seguintes ficando o derradeiro bit reservado

para representar o signo. Por mor desa representación interna, os números enteiros en JS posúen unha precisión de até 15 díxitos:

```
var x = 999999999999999;  // 999999999999999
var x = 9999999999999999; // 10000000000000000
```

De xeito semellante, o número de decimais é de 17, pero as operacións ariméticas poden producir erros de precisión (que poden ser resoltos, por exemplo, multiplicando os operandos por algunha potencia 10 e dividíndoos posteriormente polo mesmo número).

```
var x = 0.1 + 0.2; // 0.30000000000000004
var x = (0.1*10 + 0.2*10) / 10; // 0.3
```

Propiedades

Velaquí algunhas das propiedades que podemos empregar no contexto deste tipo de obxecto:

- **`MIN_VALUE`:** devolve o mínimo número representable en JavaScript.
- **`MAX_VALUE`:** devolve o máximo número representable en JavaScript.
- **`POSITIVE_INFINITY`:** representa os números positivos que saen do rango posible (é dicir, que producen *overflow*). Por exemplo: os resultantes de dividir un número non nulo entre cero.
- **`NEGATIVE_INFINITY`:** representa os números negativos que saen do rango posible (é dicir, que producen *overflow*). Por exemplo: os resultantes de dividir un número non nulo entre cero.
- **`NaN`[5]:** representa aqueles resultados que non son números válidos. Por exemplo: facer unha operación aritmética cun *string* non convertible en número.

```
var x = 3.14 / "Pi"; // NaN
```

Métodos

A continuación podemos ver aqueles métodos do obxecto `Number` que cómpre salientar pola habitualidade do seu uso:

- **`valueOf()`:** este método devolve o valor primitivo dun obxecto de tipo numérico.

[5] *Not a Number.*

De comprobación

Nesta categoría atopamos aqueles métodos que permiten comprobar a natureza dun número. Todos eles devolven un valor booleano.

- **`Number.isInteger()`:** devolve true se o número é enteiro e false en caso contrario.

```
Number.isInteger(123) // true
Number.isInteger(-123) // true
Number.isInteger(5-2) // true
Number.isInteger(0) // true
Number.isInteger(0.5) // false
Number.isInteger('123') // false
Number.isInteger(false) // false
Number.isInteger(Infinity) // false
Number.isInteger(-Infinity) // false
Number.isInteger(0 / 0) // false
```

- **`Number.isFinite()`:** comproba se o número está dentro do rango admitido para os números de JavaScript.

```
Number.isFinite(123) // true
Number.isFinite(-1.23) // true
Number.isFinite(5-2) // true
Number.isFinite(0) // true
Number.isFinite('123') // false
Number.isFinite('Hello') // false
Number.isFinite('2005/12/12') // false
Number.isFinite(Infinity) // false
Number.isFinite(-Infinity) // false
Number.isFinite(0 / 0) // false
```

- **`Number.isNaN()`:** comproba se o valor non é un número válido.

```
Number.isNaN(123) // false
Number.isNaN(-1.23) // false
Number.isNaN(5-2) // false
Number.isNaN(0) // false
Number.isNaN('123') // false
Number.isNaN('Hello') // false
Number.isNaN('2005/12/12') // false
Number.isNaN('') // false
Number.isNaN(true) // false
Number.isNaN(undefined) // false
Number.isNaN('NaN') // false
Number.isNaN(NaN) // true
Number.isNaN(0 / 0) // true
```

De transformación

Aquí atopamos aqueles métodos que permiten facer modificacións sobre o valor, precisión, formato ou tipo de datos do número representado:

- **`toString()`:** converte o número en *string*.

- **`toLocaleString()`:** converte o número en *string* considerando a configuración local do navegador.
- **`toFixed()`:** devolve unha *string* co número representado nunha notación de punto-fixo (ou de coma-fixa).
- **`toPrecision()`**: devolve unha *string* co número representado nunha precisión especificada (en notación de punto-fixo ou científica).
- **`toExponential()`:** devolve unha *string* co número representado en notación científica.

```
var n = new Number(0.0003141592);
n = n.toExponential(); // 3.141592e-4
n = (n*10000).toFixed(4); // 3.1416
n = (1000 + n).toLocaleString("es-ES"); // 1.003,142
```

Math

Trátase dun obxecto orientado ao desenvolvemento de operacións matemáticas de maior complexidade que aquelas fornecidas polos operadores fundamentais.

Dado que non conta con construtor, non se poden crear variables deste tipo. Polo tanto, todas as propiedades/métodos deberanse empregar facendo chamadas directas ao obxecto. Por exemplo: `Math.random()`.

Propiedades

A continuación podemos ver todas as propiedades estándar deste obxecto, empregadas todas elas para a representación de constantes de uso habitual nas matemáticas:

- **`E`:** constante de Euler, e (≈2,718).
- **`PI`:** π (≈3,1416).
- **`LN2`:** logaritmo neperiano de 2 (≈0,693).
- **`LN10`:** logaritmo neperiano de 10 (≈2,302).
- **`LOG2E`:** logaritmo en base 2 de e (≈1,442).
- **`LOG10E`:** logaritmo en base 10 de e (≈0,434).
- **`SQRT2`:** raíz cadrada de 2 (≈1,414).
- **`SQRT1_2`:** raíz cadrada de ½ (≈0,707).

```
Math.PI; // 3.141592653589793
```

Métodos

Veremos a continuación os principais métodos de que dispón o obxecto `Math`, agrupadas segundo o contexto funcional das mesmas.

Misceláneas

- **`abs()`:** devolve o valor absoluto do número pasado coma parámetro.
- **`max()`:** devolve o valor máis alto de entre dous ou máis parámetros.
- **`min()`:** devolve o valor máis baixo de entre dous ou máis parámetros.
- **`random()`:** devolve un número aleatorio no rango [0-1).

```
Math.random(); // 0.7661343673727452
Math.max(Math.abs(-7),1,3,5); // 7
```

Trigonometría

- **`sin()`:** calcula o seno do ángulo (expresado en radiáns) pasado coma parámetro.
- **`cos()`:** calcula o coseno do ángulo (expresado en radiáns) pasado coma parámetro.
- **`tan()`:** calcula a tanxente do ángulo (expresado en radiáns) pasado coma parámetro.
- **`asin()`:** calcula o arcoseno (en radiáns) do seno pasado coma parámetro.
- **`acos()`:** calcula o arcocoseno (en radiáns) do seno pasado coma parámetro.
- **`atan()`:** calcula o arcotanxente (en radiáns) do seno pasado coma parámetro.

```
Math.sin(Math.PI/2); // 1
Math.cos(Math.PI); // -1
```

Operacións con decimais

- **`round()`:** devolve o enteiro máis próximo (por arriba ou por abaixo) do número pasado coma parámetro.

```
Math.round(3.5); // 4
```

- **`trunc()`:** devolve a parte enteira do número pasado coma parámetro.

```
Math.trunc(3.5); // 3
```

- **`floor()`:** Devolve o enteiro máis próximo (por abaixo) do número pasado coma parámetro.

```
Math.floor(-3.5); // -4
```

- **`ceil()`:** Devolve o enteiro máis próximo (por arriba) do número pasado coma parámetro.

```
Math.ceil(Math.PI); // 4
```

Potencias

- **`exp()`:** eleva o número **e** ao número pasado coma parámetro.
- **`pow()`:** eleva o primeiro número recibido coma parámetro ao segundo argumento.

```
Math.pow(2,10); // 1024
```

Raíces

- **`sqrt()`:** calcula a raíz cadrado do número pasado coma parámetro.
- **`cbrt()`:** calcula a raíz cúbica do número pasado coma parámetro.

```
Math.sqrt(100); // 10
```

Logaritmos

- **`log()`:** calcula o logaritmo neperiano do número pasado coma parámetro.
- **`log2()`:** calcula o logaritmo en base 2 do número pasado coma parámetro.
- **`log10()`:** calcula o logaritmo en base 10 do número pasado coma parámetro.

```
Math.log(Math.exp(3)); // 3
```

`Error`

Cando se produce un erro na execución de código JS, o comportamento xeral será o de deter a execución e amosar unha mensaxe na consola, aínda que a depuración de código de JS, en tanto linguaxe interpretada, segue a ser un chisco máis difícil que a de linguaxes compiladas coma Java ou C++.

Desque se incluíu o obxecto `Error` en JS, xunta coa estrutura `try-catch-finally`, a linguaxe permite manexar tipos de erros para dar resposta a distintas eventualidades. Deste xeito poderase continuar coa execución do programa co gaio de desenvolver aplicacións máis robustas.

Creación e lanzamento

Os erros en tempo de execución en JS implican a creación automática dun obxecto de tipo `Error` e o lanzamento do mesmo para que poida ser tratado polo motor do do navegador.

Poden, así mesmo, crearse erros personalizados (coma en calquera obxecto instanciable), mediante a emprega da palabra reservada `new`.

```
var e = new Error();
var e = new Error("O meu erro personalizado");
```

Unha vez creado, o obxectivo será lanzar o erro para que o motor JS poida captura e proceda ao seu tratamento (isto realízase mediante a emprega da palabra reservada `throw`).

```
throw e;
throw new Error("O meu erro personalizado");
throw "O meu erro personalizado"; // Forma abreviada
```

Ademais do obxecto Error xenérico, existen 6 subtipos a maiores que permiten personalizar o tratamento de erros:

- **`EvalError`:** lánzase cando se produce un erro na execución de código nunha función `eval()`.
- **`RangeError`:** prodúcese cando a unha variable élle asignado un valor fora do seu rango válido.
- **`ReferenceError`:** obtense este tipo de erro no caso de que unha referencia non sexa válida no momento de facela.
- **`SyntaxError`:** xérase, coma no primeiro caso, cando se produce un erro no código introducido en `eval()`, aínda que nesta ocasión dase ao «parsear» a sintaxe do devandito código.
- **`TypeError`:** prodúcese cando unha variable ou parámetro non é do tipo axeitado.
- **`URIError`:** obtense cando non se lle pasan parámetros válidos ás funcións `encodeURI()` e `decodeURI()`.

Propiedades

A continuación as propiedades deste obxecto destacamos:

- **`name`:** almacena o nome do erro.
- **`message`:** almacena un texto curto coa descrición do erro.

```
var e = new RangeError("Number too big");
console.log(e.name); // RangeError
console.log(e.message); // Number too big
```

try-catch-finally

Xunto coas estruturas de control condicionais e repetitivas, JavaScript (do mesmo xeito que ocorre con outras linguaxes de programación) fornece unha estrutura específica para o tratamento de erros que permite a xeración de código máis robusto.

A sintaxe desta estrutura é a seguinte:

```
try {
  // Código que pode xerar erros
} catch(err) {
  // Código para manexar erros
} finally {
  // Código que se executa no remate, independentemente da xeración de erros ou
non
}
```

- **`try`:** indica o inicio dun bloco de programa que vai ser vixiado por se ocorren erros. No caso de que se produzan, o fluxo de control salta inmediatamente ó bloco seguinte (`catch`). Os erros poden xerarse automáticamente o ser lanzados expresamente mediante a instrución `throw`.
- **`catch`:** nesta sección inclúese o bloco de programa que será executarado inmediatamente ao se producir un erro. Pódese acceder á súa información a través do parámetro `err`.
- **`finally`:** esta sección acolle un bloco de programa opcional que sería executado no remate do bloco `try` ((se non se produce ningún erro) ou no remate do bloco `catch` (caso de producirse algún erro).
- **`throw`:** esta instrución permite lanzar expresamente erros para ter un mellor control do fluxo de programa. O tipo de erro lanzado pode ser un obxecto de `String`, `Number`, `Boolean` ou `Object`.

```
try {
  console.log("Non error code"); // Bloque de programa a tentar
  throw new Error("O meu erro personalizado"); // Prodúcese un erro e salta ao
catch
  console.log("Código non acadable"); // Código que non chega executar
} catch(err) {
  // Código para o manexo de erros
  console.log(err.name + ": " + err.message);
} finally {
  // Código que é executado independentemente de se ocorren erros ou non
  console.log("Código executado con/sen erros");
}
```

Arrays

As estruturas de datos existen nas linguaxes de programación de alto nivel (JS incluída) para permitir, entre outras cousas, gardar información máis complexa e heteroxénea daquela que poden conter os tipos de datos elementais. Podemos, polo tanto, considerar as estruturas de datos coma coleccións de datos (elementais ou compostos) que se caracterizan pola súa organización interna e as operacións que se poden facer cos datos que almacenan.

Actividade

Cando desenvolvemos aplicacións é habitual que teñamos que almacenar listas de elementos relacionados. Un exemplo poderían ser as estacións do ano:

```
var estacion1 = "Primavera";
var estacion2 = "Verán";
var estacion3 = "Outono";
var estacion4 = "Inverno";
```

No caso anterior, atopariámonos cun problema se tiveramos, por exemplo, que facer comparacións con todas as variables anteriores para comprobar a coincidencia cun valor seleccionado. Este feito agravaríase se, no canto de estacións, fosen os meses do ano ou os días do mes.

Para facilitar o tratamento destes datos nos que pode haber varios elementos relacionados, as linguaxes de programación dispoñen dun tipo de dato especial, os *arrays*. A característica principal dos *arrays* é que poden almacenar varios valores baixo un mesmo nome, podendo estes ser do mesmo ou diferente tipo, e simples ou compostos (como outros *arrays*, obxectos, funcións, etc.).

```
var estacions; // Variable que será de tipo array
```

En resumo, un *array* ou matriz é unha colección de elementos ordenados nunha ou máis dimensións.

Creación

Hai 2 xeitos principais para crear *arrays*:

- **Empregando a palabra clave new:**
    - Xa sexa para crear un *array* baleiro:

```
var estacions = new Array();
estacions instanceof Array; // true
```

- Ou un *array* a partir dos elementos que van compoñelo:

```
var estacions = new Array("Primavera", "Verán", "Outono", "Inverno");
```

- **De forma literal empregando corchetes (`[]`).**
  - Unha vez máis, xa para crear un *array* baleiro:

```
var estacions = [];
estacions instanceof Array; // true
```

  - Ou un *array* co seu contido xa definido:

```
var estacions = ["Primavera", "Verán", "Outono", "Inverno"];
```

Empregando calquera das dúas formas anteriores conseguimos o mesmo.

Acceso

Os elementos dun *array* en JS son accedidos mediante o seu índice, isto é, o seu número de orde na colección que é o propio *array*. O devandito índice indícase entre corchetes, sendo sempre o primeiro índice o cero.

```
estacions[0]; // Primavera
estacions[3]; // Inverno
estacions[estacions.length-1]; // Inverno
```

Así, unha forma equivalente de crear o *array* de estacións do ano sería crear primeiro o *array* baleiro e, posteriormente, ir engadindo os elementos:

```
var estacions = new Array();
estacions[0] = "Primavera";
estacions[1] = "Verán";
estacions[2] = "Outono";
estacions[3] = "Inverno";
```

Desta forma tamén se poderán percorrer todos os elementos dun *array* empregando un bucle, sen máis que modificar o índice de acceso:

```
// Mediante un bucle WHILE
var i=0;
while (i<=estacions.length-1) {
  console.log(estacions[i]);
  i++;
}

// Mediante un bucle FOR
for (var i=0; i< estacions.length; i++) {
  console.log(estacions[i]);
}
```

```
// Mediante un bucle FOR-IN
for (i in estacions) {
  console.log(estacions[i]);
}

// Mediante un bucle FOR-OF
for (var estacion of estacions) {
  console.log(estacion);
}
```

Eliminación

A primeira forma empregada en JS para eliminar elementos de *arrays* era a asignación ás posicións correspondentes do valor `null`, o que non constitúe unha verdadeira eliminación, xa que a posición continúa a existir na colección (é o que se coñece en programación coma unha «eliminación preguiceira»). Este procedemento, que se ben ten a vantaxe da súa simplicidade, implica que a partires dese intre o dato da lonxitude do *array* non coincida co número de elementos realmente contidos na colección. Velaquí un exemplo:

```
estacions.length; // 4
estacions[2] = null;
estacions.length; // 4
estacions[2]; // null
```

Versións posteriores de JS incorporaron o operador `delete`, que permite ao intérprete da linguaxe liberar memoria cando a precise. Este método eliminará o índice do *array*, pero non reducirá a súa lonxitude:

```
delete estacions[2];
estacions.length; // 4
estacions [2]; // undefined
```

O máis axeitado é, pois, empregar os métodos destinados a tal efecto que trataremos no punto seguinte.

Tipos

Segundo a cantidade de dimensións e o xeito en que os empregamos podemos clasificar os arrays en:

- **Arrays unidimensionais:** aqueles que almacenan elementos simples, como os vistos ata o de agora:

```
var estacions = ["Primavera", "Verán", "Outono", "Inverno"];
```

- **Arrays paralelos:** esta técnica baséase en almacenar en varios *arrays* de unha única dimensión información relacionada entre si:

```
var estacions = ["Primavera", "Verán", "Outono", "Inverno"];
var temperatura = [15, 25, 20, 10]; // Temperatura máxima
var chuvia = [25, 10, 30, 20]; // Precipitación media en l/m2
```

Deste xeito, empregando o mesmo índice conseguimos acceder aos datos da mesma estación do ano (p.ex: o índice 2 vainos dar en todos os *arrays* os datos correspondentes Outono):

```
estacions[2] +": "+ temperatura[2] +"°C, "+ chuvia[2] +"mm"; // Outono: 20°C, 30mm
```

Podemos, así mesmo, extrapolar unhas características comúns aos *array* membros dun mesmo grupo de *arrays* paralelos:

  - Deben ter todos a mesma lonxitude.
  - Débese manter unha orde estrita para que os datos non muden o seu significado.
  - Permiten simular *arrays* multidimensionais.
  - Son doados de interpretar visualmente.
  - Permiten facer un percorrido dos datos máis rápido.

- **Arrays multidimensionais:** en JS todos os *arrays* son teoricamente unidimensionais, pero como cada elemento dun *array* pode ser de calquera tipo (mesmo *arrays*), entón faise posible «simular» a multidimensionalidade.

```
var clima = new Array();
clima[0] = ["Primavera", "Verán", "Outono", "Inverno"];
clima[1] = [15, 25, 20, 10]; // Temperatura máxima
clima[2] = [25, 10, 30, 20]; // Precipitación media en l/m2
```

Neste caso, para simular unha estrutura bidimensional, tamén deben ser todos os *arrays* da mesma lonxitude e, para poder acceder aos datos da mesma estación do ano débese empregar o mesmo índice secundario (p.ex: o índice secundario 2 vainos dar, no *array* bidimensional, os datos correspondentes ao Outono):

```
clima[0][2] +": "+ clima[1][2] +"°C, "+ clima[2][2] +"mm"; // Outono: 20°C, 30mm
```

Este tipo de *arrays* son moito máis recomendables para programar o acceso aos seus elementos (p.ex: para recorrer secuencialmente todos os elementos mediante un bucle aniñado):

```
for (var i=0; i<clima.length; i++) { // Percorre a 1ª dimensión
  for (var j=0; j< clima[i].length; j++) { // Percorre a 2ª dimensión
    console.log(clima[i][j]);
  }
}
```

Métodos

Unha vez vistas as operacións básicas con *arrays*, o máis importante para traballar con eles é coñecer o funcionamento dos seus métodos, xa que van ser os que nos permitan aproveitar ao máximo as posibilidades dos mesmos:

- **Edición:** atopamos nesta categoría aqueles métodos que permiten inserir/extraer elementos do *array*, xa sexa modificando ou non o orixinal:
    - `pop()`: elimina o último elemento dun *array* modificando este, ao tempo que devolve o elemento eliminado.

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.pop(); // 24
console.log(idades); // [40, 17, 40, 85]
```

    - `push()`: engade un novo elemento no remate do *array*, devolvendo a nova lonxitude.

```
var idades = [40, 17, 40];
var resultado = idades.push(85, 24); // 5
console.log(idades); // [40, 17, 40, 85, 24]
```

    - `shift()`: elimina o primeiro elemento do *array* modificando este. O seu valor de retorno é o propio elemento eliminado.

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.shift(); // 40
console.log(idades); // [17, 40, 85, 24]
```

    - `unshift()`: engade un elemento ao principio do *array* ao tempo que devolve a nova lonxitude.

```
var idades = [40, 17, 40];
var resultado = idades.unshift(85, 24); // 5
console.log(idades); // [85, 24, 40, 17, 40]
```

- `slice()`: extrae un sub-*array* formado polos elementos indicados entre polas posicións dos argumentos inicio e fin (non incluído). Pódense indicar posicións negativas para comezar polo final do array.

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.slice(1,3); // [17, 40]
var resultado = idades.slice(-2); // [85, 24]
```

- `splice()`: permite eliminar elementos do *array* en posicións concretas e inserir secuencias de elementos no seu lugar. O primeiro parámetro recibido indica a posición e o segundo ao número de elementos a eliminar. Os parámetros subseguintes son os elementos que queremos engadir. O seu valor de retorno será un *array* cos elementos eliminados.

```
var idades = [40, 17, 24];
var resultado = idades.splice(2, 1, 40, 85); // [24]
console.log(idades); // [40, 17, 40, 85]
```

- `concat()`: une dous o máis *arrays* devolvendo o resultado tamén en forma de *array*.

```
var android = ["samsung", "huawei", "xiaomi"];
var outros = ["iphone"];
var smartphones = android.concat(outros);
```

- **Busca e conversión:** agrupamos aquí aqueles métodos que permiten procurar dentro do *array*, ordenalo, representalo en forma de *String* e obter información do mesmo. Son os seguintes:
  - `includes()`: determina se o array inclúe un valor concreto.

```
var idades = [40, 17, 85, 24];
var resultado = idades.includes(85); // true
```

  - `indexOf()`: procura no *array* o elemento indicado e devolve a súa posición (ou `-1` se non o atopa). Pódese indicar, opcionalmente, unha posición de inicio da procura.

```
var idades = [40, 17, 85, 40, 24];
var resultado = idades.indexOf(40, 2); // 3
```

  - `lastIndexOf()`: procura no *array* o elemento indicado comezando polo final ou, opcionalmente, nunha posición indicada *ad hoc*. Devolve a súa posición do elemento (ou `-1` se non o atopa).

```
var idades = [40, 17, 40, 85, 24];
```

```
var resultado = idades.lastIndexOf(40, 3); // 2
```

- `toString()`: devolve unha *string* con todos os valores do *array* separados por comas (`,`).

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.toString(); // 40, 17, 40, 85, 24
```

- `join()`: devolve os elementos dun *array* concatenados nunha *string*. Os elementos irán unidos por un separador (opcional), empregándose por defecto a coma (`,`).

```
var idades = [40, 17, 85, 40, 24];
var resultado = idades.join("|"); // "40|17|85|40|24"
```

- `sort()`: ordena os elementos do *array*, empregando ordenación alfabética ascendente por defecto. No caso de querermos organizar segundo un criterio diferente, deberemos fornecer unha función *ad hoc*, de xeito que a mesma reciba dous parámetros argumentos e devolva un número (positivo, cero ou negativo) que a relación entrambos (maior, igual ou menor).

```
var letras = ["beta", "gamma", "alpha", "delta"];
letras.sort(); // ["alpha", "beta", "delta", "gamma"]

var idades = [40, 17, 24, 5, 110];
idades.sort(); // [110, 17, 24, 40, 5]
idades.sort( function (a,b) {return a-b;} ); // [5, 17, 24, 40, 110]
```

- `reverse()`: inverte a orde dos elementos dun *array*, modificando o mesmo.

```
var idades = [40, 17, 40, 85, 24];
idades.reverse(); // [24, 85, 40, 17, 40]
```

- `keys()`: devolve un obxecto `Array Iterator` cos índices do *array*.

```
var idades = [40, 17, 85, 24];
var resultado = idades.keys(); // Array Iterator object
for (var i of resultado) { console.log(i); }
```

- `isArray()`: determina se un obxecto é un *array* (`true`) ou non (`false`).

```
var idades = [40, 17, 85, 40, 24];
var resultado = Array.isArray(idades); // true
```

- **Iteración:** neste grupo están aqueles métodos que aplican a cada elemento unha función, de forma automática e en secuencia**:**
    - `some()`: comproba se algún dos elementos do *array* pasa unha proba (fornecida por unha función pasada como parámetro). Este método executa a función para cada elemento e:
        - Se algún deles devolve `true`, o resultado será `true` e non seguirá comprobando.
        - Se todos devolven `false`, o resultado será `false`.

```
var idades = [40, 17, 85, 24];
var resultado = idades.some(comprobarVoto); // true

function comprobarVoto(idade) { return idade>=18; }
```

    - `every()`: comproba se todos os elementos do *array* pasan unha proba (fornecida por unha función pasada como parámetro). Este método executa a función para cada elemento e:
        - Se algún deles devolve `false`, o resultado será `false` e non seguirá comprobando.
        - Se todos devolven `true`, o resultado será `true`.

```
var idades = [40, 17, 85, 24];
var resultado = idades.every(podeVotar); // false

function podeVotar(idade) { return idade>=18; }
```

    - `find()`: devolve o primeiro valor do *array* que pasa unha proba (fornecida por unha función pasada como parámetro). Se non atopa ningún devolve `undefined`.

```
var idades = [40, 17, 85, 24];
var resultado = idades.find(eMenor); // 17

function eMenor(idade) { return idade<18; }
```

    - `findIndex()`: devolve o índice do primeiro valor do *array* que pasa unha proba (fornecida por unha función pasada como parámetro). Se non atopa ningún devolve `-1`.

```
var idades = [40, 17, 85, 24];
var resultado = idades.findIndex(eMenor); // 1

function eMenor(idade) { return idade<18; }
```

- `filter()`: devolve un *array* novo cos elementos do *array* fonte que pasan unha proba (fornecida por unha función pasada como parámetro).

```
var idades = [40, 17, 85, 24];
var resultado = idades.filter(eMaior); // [40, 85, 24]

function eMaior(idade) { return idade>=18; }
```

- `map()`: crea un novo *array* cos resultados de aplicar a cada elemento unha función (pasada como parámetro).

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.map(dobrar); // [80, 34, 80, 170, 48]

function dobrar(idade) { return idade*2; }
```

- `forEach()`: executa unha función (pasada como parámetro) unha vez por cada elemento do *array*.

```
var idades = [40, 17, 85, 24];
var resultado = 0;
idades.forEach(sumaIdades); // resultado = 166

function sumaIdades(idade) { resultado += idade; }
```

- `reduce()`: reduce os valores do *array* a un só, aplicando unha función (pasada como parámetro) a cada elemento de esquerda a dereita. O valor devolto pola función gárdase nun acumulador.

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.reduce(sumaIdades); // 206

function sumaIdades(total, idade) { return total + idade; }
```

- `reduceRight()`: reduce os valores do *array* a un só, aplicando unha función (pasada como parámetro) a cada elemento de dereita a esquerda. O valor devolto pola función gárdase nun acumulador.

```
var idades = [40, 17, 40, 85, 24];
var resultado = idades.reduce(sumaIdades); // 206

function sumaIdades(total, idade) { return total + idade; }
```

Funcións

As funcións son bloques de código deseñados para levar a feito unha tarefa particular, de xeito que poden ser reempregados varias veces e executados fornecendo valores diferentes para obter distintos resultados.

En JS as funcións considéranse obxectos, polo que terán propiedades e métodos aos que podemos acceder para obter información:

```
function objectInfo() {
  return arguments.length;
}
typeof objectInfo; // function
objectInfo(3,4,5); // 3
objectInfo.toString(); // function objectInfo() { return arguments.length; }
```

Unha función pode considerarse, segundo o seu ámbito de uso:

- **Método:** se é definida como unha propiedade dun obxecto.
- **Construtor:** se é definida para crear obxectos novos.

Funcións predefinidas

As funcións predefinidas son os métodos integrados no obxecto global da contorna de execución (p.ex.: nun navegador web o obxecto global será `window`). Estas son consideradas predefinidas posto que non é preciso facer referencia ao obxecto global para invocalas:

```
// Nun navegador web
alert("Hello!"); // equivalente a window.alert("Hello!");
```

Velaquí as funcións predefinidas de uso habitual nos navegadores web:

- **`decodeURI()`:** decodifica, coma o seu nome indica, un URI (Uniform Resource Identifier) fornecido coma *string* substituíndo as secuencias de escape UTF-8 (excepto as correspondentes con: `,` `/` `?` `:` `@` `&` `=` `+` `$` `#`) propias dos navegadores web polos caracteres correspondentes.

```
decodeURI("search.php?place=Ca%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao"); //
search.php?place=Cañón del río Mao
```

- **`decodeURIComponent()`:** decodifica todo o ou parte dun URI fornecido coma *string* substituíndo as secuencias de escape UTF-8 (excepto as correspondentes con: `,` `/` `?` `:` `@` `&` `=` `+` `$` `#`) propias dos navegadores web polos caracteres correspondentes..

```
decodeURIComponent("search.php%3Fplace%3DCa%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao");
// search.php?place=Cañón del río Mao
```

- **`encodeURI()`:** codifica un URI fornecido coma *string* substituíndo os caracteres especiais (excepto: `,` `/` `?` `:` `@` `&` `=` `+` `$` `#`) polas secuencias de escape UTF-8 correspondentes, para que sexa axeitado para os navegadores web.

```
encodeURI("search.php?place=Cañón del río Mao"); //
search.php?place=Ca%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao
```

- **`encodeURIComponent()`:** codifica todo ou parte dun URI fornecido coma *string* substituíndo os caracteres especiais (excepto: `,` `/` `?` `:` `@` `&` `=` `+` `$` `#`) polas secuencias de escape UTF-8 correspondentes, para que sexa axeitado para os navegadores web.

```
encodeURIComponent("search.php?place=Cañón del río Mao"); //
search.php%3Fplace%3DCa%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao
```

- **`eval()`:** avalía e executa o código proporcionado como *string* no caso de que sexa código JS válido.

```
x = eval("((3+4)/2).toFixed(2)"); // 3.50
```

- **`isFinite()`:** determina se un valor é un número finito. Polo tanto, devolve `false` se o valor é `+Infinity`, `-Infinity` ou `NaN`. En calquera outro caso devolve `true`.

```
isFinite(1); // true
isFinite(1/0); // false
isFinite(+Infinity); // false
isFinite(-3.14); // true
isFinite(new Date()); // true
isFinite("2019-11-22"); // false
```

- **`isNaN()`:** determina se un valor non é un número legal devolvendo `true`/`false`.

```
isNaN(1); // false
isNaN(1/0); // false
isNaN(NaN); // true
isNaN(-3.14); // false
isNaN(new Date()); // false
isNaN("2019-11-22"); // true
```

- **`Number()`:** converte o valor dun obxecto a un número. Devolve `NaN` se non é válido.

```
Number(true); // 1
Number(new Date()); // 1576224556541
Number("3.14"); // 3.14
Number("3 14"); // NaN
```

- **`parseFloat()`:** determina se o primeiro caracter dunha *string* é un número (omitindo espacios en branco ao principio). Nese caso colle caracteres até que atope un non numérico e convérteo nun número decimal. Se non atopa ningún caracter numérico devolve `NaN`.

```
parseFloat("3") // 3
parseFloat("3.0") // 3
parseFloat("3.14") // 3.14
parseFloat("3 4 5") // 3
parseFloat("   3   ") // 3
parseFloat("70 kg") // 70
parseFloat("I am 70") // NaN
```

- **`parseInt()`:** determina se o primeiro caracter dunha *string* é un número (omitindo espacios en branco ao principio). Nese caso colle caracteres até que atope un non numérico e convérteo nun número enteiro. Se non atopa ningún caracter numérico devolve `NaN`.

  A función recibe un segundo parámetro opcional que indica a base (entre 2 e 36) na que está especificado o número.

```
parseInt("3") // 3
parseInt("1001", 2) // 9
parseInt("3.14") // 3.14
parseInt("3 4 5") // 3
parseInt("   3   ") // 3
parseInt("70 kg") // 70
parseInt("I am 70") // NaN
```

- **`String()`:** converte o valor dun obxecto nunha *string*. Equivale a empregar o método `toString()` nos obxectos en cuestión.

```
String(3+4); // 7
String(Boolean(1)); // "true"
String(new Date()); // Fri Dec 13 2019 09:28:51 GMT+0100
```

Funcións definidas polo usuario

Unha boa práctica de programación é a división do código en partes máis pequenas de xeito que, na medida do posible, sexan dedicadas a tarefas atómica, isto é, o máis concretas posible e que poidan ser empregadas sen coñecer como están deseñadas internamente e, desta forma, faciliten a súa reutilización e mantemento.

A programación estruturada fornece un mecanismo para acadar o anterior: as funcións definidas polo usuario. A súa declaración en JS ten as seguintes partes, por orde:

- **Palabra clave `function`.**
- **Nome da función (opcional):** seguindo as mesmas regras que o nomeado das variables: pode conter letras ASCII, díxitos, o guión baixo (_) e o dólar ($).
- **Parénteses (`()`):** para conter os parámetros de entrada.
- **Parámetros (opcionais) dentro dos parénteses:** compórtanse como variables locais á función. Os valores que se fornezan a estes parámetros no intre de chamar á función son coñecidos como argumentos.
- **Chaves (`{}`):** para delimitar o bloque de código da propia función.
- **Bloque de código (opcional):** contido dentro das chaves. Ao seu tempo pode incluír:
  - Instrución `return` (opcional): como mecanismo para devolver un valor. Supón a saída inmediata da función sen executar o posible código que veña a continuación.

```
function [faiAlgo]([parámetro1, parámetro2, ...]) {
  // Código a executar
  [return true;]
}
```

NOTA: os corchetes (`[]`) indican partes opcionais na declaración.

Funcións estándar

Aquelas que seguen a sintaxe tradicional e, polo tanto, teñen que levar nome.

```
function greet(a,b) {
  return a+" "+b+"!";
} // Función estándar
greet("Hello", "World"); // Hello World!
```

Funcións anónimas

Aquelas que non posúen nome propio. Velaquí os escenarios típicos nos que vamos atopalas:

- Para asignar a variables e poder invocalas posteriormente.

```
var minhaFuncion = function (a,b) { return a+", "+b+"!"; } // Función anónima
minhaFuncion("Como vai", "amig@"); // Como vai, amig@!
```

- Pasadas como parámetro de métodos que reciban unha función, co gaio de realizar algún procesamento.

```
[21,3,5,13,8,44].filter(function (a) {return a>10;}); // [21,13,44]
```

Construtor de funcións

No canto de empregar a palabra reservada `function`, pódese facer o mesmo co construtor do obxecto `Function()`.

```
var novaFuncion = new Function ("a","b","return a+', '+b+'!';"); // Función
construtor
novaFuncion ("Chao", "xente"); // Chao, xente!
```

NOTA: este é un método moi útil para crear funcións alternativas dinamicamente en base ás respostas do usuario.

Funcións frecha

Consideradas simplemente como un xeito «reducido» de escribir funcións (o que coloquialmente no mundo da programación é coñecido coma «azucre sintáctico»), de xeito que posibilita empregar unha sintaxe máis curta para expresar as funcións. Mediante o mecanismo das funcións frecha elimínase a necesidade de empregar as palabras clave `function`, `return` e as chaves (`{}`):

```
var x = function (a,b) {return a+b}; // ECMAScript 5
x(3,4); // 7
const y = (a,b) => a+b; // ECMAScript 6
y(3,4); // 7
```

As súas características máis salientables son:

- Non teñen o seu propio obxecto `this`, polo que non son axeitadas para definir métodos de obxectos.
- Non admiten *hoisting*, polo que deben ser definidas sempre denantes ser empregadas.
- É recomendable empregar `const` no canto de `var` para declarar aquelas variables ás que se lles vaia asignar valor mediante funcións frecha, porque estas últimas sempre teñen valor constante.
- Só se pode omitir o uso de `return` e as chaves (`{}`) se a función é unha expresión simple, polo que é un bo hábito usalos sempre.

```
const z = (a,b) => {return a+b};
z(3,4); // 7
```

Invocación de funcións

Coñecese como **invocación** ao procedemento mediante o cal chamamos ás funcións para a súa execución. Para invocar unha función empregamos o operador parénteses (`()`). Faise así referencia

ao valor devolto pola función xa que, de non engadir os parénteses, o que estariamos a facer sería referencia ao obxecto función:

```
function faiAlgo() {return "Ola mundo!";}
console.log(faiAlgo()); // Ola mundo!
console.log(faiAlgo); // function faiAlgo() {return "Ola mundo!";}
```

A chamada dunha función pode ocorrer en distintas circunstancias:

- **Función global:** cando a función é chamada expresamente dende o propio código JS.

```
function faiAlgo() {return "Ola mundo!";}
faiAlgo(); // Ola mundo!
```

- **Evento web:** acontece cando un evento é desencadeado nunha páxina web (p.ex: o usuario preme un botón que ten asociada unha función).

```
<button id="action" onclick="faiAlgo();">Invocar!</button>
```

- **Autoinvocación:** prodúcese automaticamente no intre de definir a función. Para elo débese usar o operador parénteses (`()`).

```
(function faiAlgo() {return "Ola mundo!";})(); // Ola mundo!
(function () {return "Ola mundo!";})(); // Ola mundo! (función anónima)
(function (a,b) {return a+b;})(3,4); // 7 (chamada con parámetros)
```

- **Método:** cando a función pertence a un obxecto haberá que antepoñer o nome do mesmo para invocala.

```
window.alert("Ola mundo!");
```

- **Construtor:** trátase dun tipo especial de función empregado para crear obxectos. Invócase mediante a palabra reservada `new`.

```
function meuArray(a,b) { return [a,b]; } // Función construtor
var x = new meuArray(3,4); // Invocación da función construtor
console.log(x.length); // 2
console.log(x); // [3,4]
```

Parámetros

Os parámetros son os mecanismos empregados polas funcións para recibir valores cos que traballar. Estes inclúense en forma de listaxe de valores separados por comas na definición das funcións. Pola contra, os valores reais recibidos á hora de invocar a función coñécense como

argumentos. Desta forma, os parámetros funcionan como variables locais á función, e son inicializados automaticamente cos valores pasados (argumentos) no intre da invocación.

Na definición das funcións non se especifica o tipo de datos dos parámetros e, polo tanto, o intérprete de JS non fai ningunha comprobación de:

- **Tipo dos argumentos:** será tarefa da programadora ou programador realizar as comprobacións e conversións de tipos que correspondan.
- **A cantidade dos argumentos:**
  - Se unha función é chamada con menos argumentos dos declarados, asignaráselles un valores `undefined`. Para evitar isto pódense establecer valores por defecto para os parámetros na definición da función (posibilidade incorporada en ECMAScript 2015 ES6).
  - Se unha función é chamada con máis argumentos dos declarados, simplemente descártanse os sobrantes.

```
function probarParametros1(a, b) { return b; }
probarParametros1(3); // undefined
function probarParametros2(a, b=1) { return b; }
probarParametros2(3); // 1
function probarParametros3(a, b) {return b;}
probarParametros3(3,4,5); // 4
```

As funcións JS teñen incorporado un obxecto `arguments` (excepto as funcións frecha, dado que non poden acceder a `this`) que contén un *array* cos argumentos recibidos na invocación da función.

```
function amosarArgumentos() {
  for (var i=0; i<arguments.length; i++) {
    console.log(arguments[i]);
  }
}
amosarArgumentos(3,4,5); // 3 4 5
```

En JS o paso de parámetros de tipo simple faise por valor, o que quere decir que, se dentro dunha función múdase o valor dun argumento, non é mudado o valor do parámetro orixinal (os cambios nos argumentos non son visibles fora das funcións).

```
var a=3,b=4;
console.log(faiAlgo(a,b)); // 8
console.log(a); // 3
console.log(b); // 4

function faiAlgo(a, b) {
    a += 1; // Modifica o argumento 'a' que funciona como variable local
    return a+b;
}
```

Non obstante, se o que se pasa á función é un obxecto (como un *array*), o valor que se pasa é a referencia ao obxecto e, polo tanto, se dentro da función é mudada algunha propiedade do obxecto,

si muda o valor orixinal do mesmo (os cambios nas propiedades de obxectos si son visibles fora das funcións).

```
// PASO POR VALOR
var num = 3; // O dato simple pasarase por valor
function incrementar(a) { return ++a; } // Modifica o argumento 'a' pero non a
variable 'num' de fora
console.log(incrementar(num)); // 4
console.log(num); // 3

// PASO POR REFERENCIA
var numArray = [3,4,5]; // Obxecto array pasarase por referencia
function incrementarArray(a) {
    for (var i in a) { a[i]++; } // Modifica o argumento 'a' e a variable
'numArray' de fora
    return a;
}
console.log(incrementarArray(numArray)); // [4,5,6]
console.log(numArray); // [4,5,6]
```

ÍNDICE