

UD 4. JAVASCRIPT

Índex

1. INTRODUCCIÓ	5
1.1. Què és JavaScript?	5
1.2. Per què es diu JavaScript?	5
1.3. Com treballen els motors?	5
1.4. Què pot fer JavaScript en el navegador?	6
1.5. Què NO POT fer JavaScript en el navegador?	6
1.6. Què fa a JavaScript únic?	7
2. HOLA MÓN	8
2.1. L'etiqueta "script"	8
2.2. Marcat Moderno	8
L'atribut type: <script type=...>	8
L'atribut language: <script language=...>	8
Scripts externs	9
3. CONTROL DE FLUX	10
3.1. Sentències	10
3.2. Punt i coma	10
3.3. Comentaris	11
4. EL MODE MODERN "USE STRICT"	12
4.1. "use strict"	12
Assegura't que "use strict" està a l'inici	12
4.2. Consola del navegador	13
4.3. Hauríem d'utilitzar "use strict"?	13
5. VARIABLES I TIPUS DE DADES	14
5.1. Una Variable	14
5.2. var en comptes de let	15
5.3. Nomenament de variables	15
5.4. Noms reservats	15
5.5. Una assignació sense utilitzar use strict	16
5.6. Constants	16
5.7. Constants majúscules	17
5.8. Tipus de dades	17
Number	18
BigInt	18

String	19
Boolean (tipus lògic)	20
El valor "null".	20
El valor "undefined" (indefinit)	20
Object i Symbol.....	21
L'operador typeof.....	21
6. INTERACCIÓ	22
6.1. alert	22
6.2. prompt.....	22
6.3. confirm	22
7. CONVERSIONS DE TIPUS	23
7.1. ToString.....	23
7.2. ToNumber	23
Addició '+' concatena strings.....	24
7.3. ToBoolean	24
8. OPERADORS BÀSICS I MATEMÀTICS	25
8.1. Termes: "unari", "binari", "operant"	25
8.2. Matemàtiques	25
Resta %	26
Exponenciación **	26
Concatenació de cadenes amb el binari +.....	26
Precedència de l'operador	27
Assignació.....	28
Modificar en el lloc.....	28
Incremente/decrement	29
Operadors a nivell de bit	29
9. COMPARACIONS.....	30
9.1. Booleà és el resultat	30
9.2. Comparació de cadenes	30
9.3. Comparació de diferents tipus.....	31
9.4. Igualtat estricta	31
9.5. Comparació amb nuls i indefinits.....	32
Para un control d'igualtat estricta ===	32
Per a una comparació no estricta ==	32
10. CONDICIONALS.....	33
10.1. La sentència "if"	33
Conversió Booleana	33
10.2. La clàusula "else"	34
10.3. Moltes condicions: "else if"	34
10.4. Operador ternari '?'	35
11. OPERADORS LÒGICS	36

11.1.	(OR)	36
11.2.	&& (AND)	37
11.3.	! (NOT)	38
12.	BUCLES: WHILE I FOR	39
12.1.	El bucle “while”	39
12.2.	El bucle “do...while”	40
12.3.	El bucle “for”	40
	Declaració de variable en línia.....	41
12.4.	Trencant el bucle	41
12.5.	Continuar a la següent iteració	42
13.	LA SENTÈNCIA “SWITCH”	43
13.1.	La sintaxi	43
13.2.	Agrupament de “case”	46
14.	FUNCIONS	47
14.1.	Declaració de funcions	47
14.2.	Variables Locals	48
14.3.	Variables Externes	48
14.4.	Paràmetres	49
14.5.	Valors predeterminats	50
14.6.	Paràmetres predeterminats alternatius	51
14.7.	Retornant un valor.....	52

1. INTRODUCCIÓ

1.1. Què és JavaScript?

JavaScript va ser creat per a “donar vida a les pàgines web”. Els programes en este llenguatge es diuen scripts. Es poden escriure directament en l'HTML d'una pàgina web i executar-se automàticament a mesura que es carrega la pàgina.

Els scripts es proporcionen i executen com a text pla. No necessiten preparació especial o compilació per a córrer. En este aspecte, JavaScript és molt diferent d'un altre llenguatge anomenat Java.

1.2. Per què es diu JavaScript?

Quan JavaScript va ser creat, inicialment tenia un altre nom: “LiveScript”. Però Java era molt popular en eixe moment, així que es va decidir que el posicionament d'un nou llenguatge com un “Germà menor” de Java ajudaria.

Però a mesura que evolucionava, JavaScript es va convertir en un llenguatge completament independent amb la seua pròpia especificació anomenada ECMAScript, i ara no té cap relació amb Java.

Hui, JavaScript pot executar-se no sols en els navegadors, sinó també en servidors o fins i tot en qualsevol dispositiu que compte amb un programa especial anomenat El motor o intèrpret de JavaScript.

El navegador té un motor embegut a vegades dit una “Màquina virtual de JavaScript”. Diferents motors tenen diferents “noms en clau”. Per exemple:

- V8 – en Chrome, Opera i Edge.
- SpiderMonkey – en Firefox.
- Existeixen altres noms en clau com “Chakra” per a IE, “JavaScriptCore”, “Nitro” i “SquirrelFish” per a Safari, etc.

És bo recordar estos termes perquè són usats en articles en internet.

1.3. Com treballen els motors?

Els motors són complicats, però els fonaments són fàcils.

- El motor (embegut si és un navegador) llig (“analitza”) el script.
- Després converteix (“compila”) el script a llenguatge de màquina.
- Finalment, el codi màquina s'executa, molt ràpid.

El motor aplica optimitzacions en cada pas del procés. Fins i tot observa com el script compilat s'executa, analitza les dades que flueixen a través d'ell i aplica optimitzacions al codi màquina basades en eixe coneixement.

1.4. Què pot fer JavaScript en el navegador?

El JavaScript modern és un llenguatge de programació “segur”. No proporciona accés de baix nivell a la memòria ni a la CPU (UCP); ja que es va crear inicialment per als navegadors, els quals no ho requereixen.

Les capacitats de JavaScript depenen en gran manera en l'entorn en què s'executa. Per exemple, Node.JS suporta funcions que permeten a JavaScript llegir i escriure arxius arbitràriament, realitzar sol·licituds de xarxa, etc.

En el navegador JavaScript pot realitzar qualsevol cosa relacionada amb la manipulació d'una pàgina web, interacció amb l'usuari i el servidor web.

Per exemple, en el navegador JavaScript és capaç de:

- Agregar nou HTML a la pàgina, canviar el contingut existent i modificar estils.
- Reaccionar a les accions de l'usuari, executar-se amb els clics del ratolí, moviments del punter i en oprimir tecles.
- Enviar sol·licituds de xarxa a servidors remots, descarregar i carregar arxius (Tecnologies anomenades AJAX i COMET).
- Obtindre i configurar cookies, fer preguntes al visitant i mostrar missatges.
- Recordar dades en el costat del client amb l'emmagatzematge local (“local storage”).

1.5. Què NO POT fer JavaScript en el navegador?

Les capacitats de JavaScript en el navegador estan limitades per a protegir la seguretat d'usuari. L'objectiu és evitar que una pàgina maliciosa accedisca a informació privada o danye les dades d'usuari.

Exemples de tals restriccions inclouen:

- JavaScript en el navegador no pot llegir i escriure arbitràriament arxius en el disc dur, copiar-los o executar programes. No té accés directe a funcions del Sistema operatiu (US).
- Els navegadors més moderns li permeten treballar amb arxius, però l'accés és limitat i solo permés si l'usuari realitza unes certes accions, com “arrossegar” un arxiu a la finestra del navegador o seleccionar-lo per mitjà d'una etiqueta `<input>`.
- Existeixen maneres d'interactuar amb la càmera, micròfon i altres dispositius, però això requereix el permís explícit de l'usuari.

- Diferents pestanyes i finestres generalment no es coneixen entre si. A vegades sí que ho fan: per exemple, quan una finestra usa JavaScript per a obrir una altra. Però fins i tot en este cas, JavaScript no pot accedir a l'altra si provenen de diferents llocs (de diferent domini, protocol o port).

Esta restricció és coneguda com a “política del mateix origen” (“Same Origin Policy”). És possible la comunicació, però totes dues pàgines han d'acordar l'intercanvi de dades i també han de contindre el codi especial de JavaScript que permet controlar-lo.

- JavaScript pot fàcilment comunicar-se a través de la xarxa amb el servidor d'on la pàgina actual prové. Però la seua capacitat per a rebre informació d'altres llocs i dominis està bloquejada. Encara que siga possible, això requereix un acord explícit (expressat en els encapçalats HTTP) des del lloc remot. Una vegada més: això és una limitació de seguretat.

Tals limitacions no existeixen si JavaScript és usat fora del navegador; per exemple, en un servidor. Els navegadors moderns també permeten complements i extensions que poden sol·licitar permisos estesos.

1.6. Què fa a JavaScript únic?

Existeixen almenys tres coses genials sobre JavaScript:

- Completa integració amb HTML i CSS.
- Les coses simples es fan de manera simple.
- Suportat per la majoria dels navegadors i habilitat de manera predeterminada.
- JavaScript és l'única tecnologia dels navegadors que combina estes tres coses.

Això és el que fa a JavaScript únic. Per això és l'eina més estesa per a crear interfícies de navegador.

Dit això, JavaScript també permet crear servidors, aplicacions mòbils, etc.

2. HOLA MÓN

2.1.L'etiqueta "script".

Els programes de JavaScript es poden inserir en quasi qualsevol part d'un document HTML amb l'ús de l'etiqueta `<script>`.

Per exemple:

```
<!DOCTYPE HTML>
<html>
  <body>
    <p>Antes del script...</p>
    <script>
      alert( '¡Hola, mundo!' );
    </script>
    <p>...Después del script.</p>
  </body>
</html>
```

Pots executar l'exemple fent clic en el botó "Play" a la cantonada superior dreta del quadre de dalt.

L'etiqueta `<script>` conté codi JavaScript que s'executa automàticament quan el navegador processa l'etiqueta.

2.2.Marcat Moderno

L'etiqueta `<script>` té alguns atributs que rares vegades s'usen en l'actualitat, però encara es poden trobar en codi antic:

L'atribut type: `<script type=...>`

L'antic estàndard HTML, HTML4, requeria que un script tinguera un type. En general, era `type="text/javascript"`. Ja no és necessari. A més, l'estàndard HTML modern va canviar totalment el significat d'este atribut. Ara, es pot utilitzar per a mòduls de JavaScript. Però això és un tema avançat, parlarem sobre mòduls en una altra part del tutorial.

L'atribut language: `<script language=...>`

Este atribut estava destinat a mostrar el llenguatge del script. Este atribut ja no té sentit perquè JavaScript és el llenguatge predeterminat. No hi ha necessitat d'usar-ho.

Scripts externs

Si tenim un munt de codi JavaScript, podem posar-lo en un arxiu separat.

Els arxius de script s'adjunten a HTML amb l'atribut src:

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

Ací, /path/to/script.js és una ruta absoluta a l'arxiu de script des de l'arrel del lloc. També es pot proporcionar una ruta relativa des de la pàgina actual. Per exemple, src="script.js" significaria un arxiu "script.js" en la carpeta actual.

També podem donar una URL completa. Per exemple:

```
<script src="https://linkarecurso.com/script.js"></script>
```

Per a adjuntar diversos scripts, usa diverses etiquetes:

```
<script src="/js/script1.js"></script>  
<script src="/js/script2.js"></script>  
...
```

Com a regla general, només els scripts més simples es col·loquen en l'HTML. Els més complexos resideixen en arxius separats.

L'avantatge d'un arxiu separat és que el navegador el descarregarà i l'emmagatzemarà en caixet.

Altres pàgines que fan referència al mateix script ho prendran del caixet en lloc de descarregar-lo, per la qual cosa l'arxiu només es descarrega una vegada.

Això redueix el trànsit i fa que les pàgines siguin més ràpides.

Si s'estableix src, el contingut del script s'ignora.

Una sola etiqueta <script> no pot tindre l'atribut src i codi dins.

3. CONTROL DE FLUX

3.1. Sentències

Les sentències són construccions sintàctiques i comandos que realitzen accions.

Ja hem vist una sentència, `alert('Hola món!')`, que mostra el missatge “Hola món!”.

Podem tindre tantes sentències en el nostre codi com vulguem, les quals es poden separar amb un punt i coma.

Per exemple, ací separem “Hello World” en dos alerts:

```
alert('Hola'); alert('Món');
```

Generalment, les sentències s'escriuen en línies separades per a fer que el codi siga més llegible:

```
alert('Hola');  
alert('Món');
```

3.2. Punt i coma

Es pot ometre un punt i coma en la majoria dels casos quan existeix un salt de línia.

Això també funcionaria:

```
alert('Hola')  
alert('Món')
```

Ací, JavaScript interpreta el salt de línia com un punt i coma “implícit”. Això es denomina inserció automàtica de punt i coma.

En la majoria dels casos, una nova línia implica un punt i coma. Però “en la majoria dels casos” no significa “sempre”!

Hi ha casos en què una nova línia no significa un punt i coma. Per exemple:

```
alert(3 +  
1  
+ 2);
```

El codi dona com a resultat 6 perquè JavaScript no inserida punt i coma ací. És intuïtivament obvi que si la línia acaba amb un signe més “+”, és una “expressió incompleta”, un punt i coma ací seria incorrecte. I en este cas això funciona segons el que es preveu.

Però hi ha situacions en les quals JavaScript “falla” en assumir un punt i coma on realment es necessita.

Els errors que ocorren en tals casos són bastant difícils de trobar i corregir.

Recomanem col·locar punts i coma entre les sentències, fins i tot si estan separades per salts de línia. Esta regla està àmpliament adoptada per la comunitat. Notem una vegada més que és possible ometre els punts i coma la majoria del temps. Però és més segur, especialment per a un principiant, usar-los.

3.3.Comentaris

A mesura que passa el temps, els programes es tornen cada vegada més complexos. Es fa necessari agregar comentaris que descriuen el que fa el codi i per què.

Els comentaris es poden posar en qualsevol lloc d'un script. No afecten la seua execució perquè el motor simplement els ignora.

Els comentaris d'una línia comencen amb dos caràcters de barra diagonal //.

La resta de la línia és un comentari. Pot ocupar una línia completa pròpia o seguir una sentència.

Com ací:

```
// Este comentari ocupa una línia pròpia.  
alert('Hello');  
  
alert('World'); // Este comentari segueix a la sentència.
```

Els comentaris de diverses línies comencen amb una barra inclinada i un asterisc /* i acaben amb un asterisc i una barra inclinada */.

Com ací:

```
/* Un exemple amb dos missatges.  
Este és un comentari multilínia.  
*/  
alert('Hola');  
alert('Món');  
  
El contingut dels comentaris s'ignora, per la qual cosa si col·loquem el codi dins de  
/* ... */ , no s'executarà.
```

Els comentaris niats no són admesos! No pot haver-hi /*...*/ dins d'un altre /*...*/.

Els comentaris augmenten la grandària general del codi, però això no és un problema en absolut. Hi ha moltes eines que minimitzen el codi abans de publicar-lo en un servidor de producció. Eliminen els comentaris, per la qual cosa no apareixen en els scripts de treball. Per tant, els comentaris no tenen cap efecte negatiu en la producció.

4. EL MODE MODERN “USE STRICT”

Durant molt de temps, JavaScript va evolucionar sense problemes de compatibilitat. S'afegien noves característiques al llenguatge sense que la funcionalitat existent canviara.

Això tenia el benefici de mai trencar codi existent, però el dolent era que qualsevol error o decisió incorrecta presa pels creadors de JavaScript es quedava per sempre en el llenguatge.

Això va ser així fins a 2009, quan ECMAScript 5 (ES5) va aparèixer. Esta versió va afegir noves característiques al llenguatge i va modificar algunes de les ja existents. Per a mantindre el codi antic funcionant, la major part de les modificacions estan desactivades per defecte. Has d'activar-les explícitament usant una directiva especial: "use strict".

4.1. “use strict”

La directiva s'assembla a un string: "use strict". Quan se situa al principi d'un script, el script sencer funciona de la manera “moderna”.

Per exemple:

```
"use strict";  
// este codi funciona de la manera moderna  
...
```

Aprendrem funcions (una manera d'agrupar comandos) en breu, però avancem que "use strict" es pot posar a l'inici d'una funció. D'esta manera, s'activa el mode estricte únicament en eixa funció. Però normalment s'utilitza per al script sencer.

Assegura't que “use strict” està a l'inici

Per favor, assegura't que "use strict" està al principi dels teus scripts. Si no, el mode estricte podria no estar activat.

El mode estricte no està activat ací:

```
alert("una mica de codi");  
// la directiva "use strict" de baix és ignorada, ha d'estar al principi  
"use strict";  
// el mode estricte no està activat
```

Únicament poden aparèixer comentaris per damunt de "use strict".

4.2. Consola del navegador

Quan utilitzes la consola del navegador per a executar codi, tingues en compte que no utilitza `use strict` per defecte.

A vegades, on `use strict` cause diferència, obtindràs resultats incorrectes. Llavors, com utilitzar `use strict` en la consola?

Primer pots intentar prement `Shift+Enter` per a ingressar múltiples línies i posar `use strict` al principi, com ací:

```
'use strict'; <Shift+Enter per a una nova línia>
// ...el teu codi
<Intro per a executar>
```

Això funciona per a la majoria dels navegadors, específicament Firefox i Chrome.

Si això no funciona, com en els vells navegadors, hi ha una lletja però de confiança manera d'assegurar `use strict`. Posa-ho dins d'esta espècie d'embolcall:

```
(function() {
  'use strict';

  // ...el teu codi...
})();
```

4.3. Hauríem d'utilitzar “`use strict`”?

La pregunta podria semblar òbvia, però no ho és.

Un podria recomanar que es comencen els script amb `"use strict"`... Però saps el que és interessant?

El JavaScript modern admet “classes” i “mòduls”, estructures de llenguatge avançades, que automàticament habiliten `use strict`. Llavors no necessitem agregar la directiva `"use strict"` si les usem.

Llavors, ara com ara `"use strict"`; és un convidat benvingut al topall dels teus scripts. Després, quan el teu codi siga tot classes i mòduls, pots ometre'l.

5. VARIABLES I TIPUS DE DADES

La majoria del temps, una aplicació de JavaScript necessita treballar amb informació. Ací hi ha 2 exemples:

- Una botiga en línia – La informació pot incloure els béns a la venda i un “carret de compres”.
- Una aplicació de xat – La informació pot incloure els usuaris, missatges, i molt més.

Utilitzem les variables per a emmagatzemar esta informació.

5.1. Una Variable

Una variable és un “magatzem amb un nom” per a guardar dades. Podem usar variables per a emmagatzemar llepolies, visitants, i altres dades.

Per a generar una variable en JavaScript, s'usa la paraula clau `let`.

La següent declaració genera (en altres paraules: declara o defineix) una variable amb el nom “message”:

```
let message;
```

Ara podem introduir dades en ella en utilitzar l'operador d'assignació `=`:

```
let message;  
message = 'Hola'; // emmagatzemar la cadena 'Hola' en la variable anomenada  
message
```

La cadena ara està emmagatzemada en l'àrea de la memòria associada amb la variable. La podem accedir utilitzant el nom de la variable:

```
let message;  
message = 'Hola!';  
  
alert(message); // mostra el contingut de la variable
```

Per a ser concisos, podem combinar la declaració de la variable i la seua assignació en una sola línia:

```
let message = 'Hola!'; // defineix la variable i assigna un valor  
  
alert(message); // Hola!
```

5.2. var en comptes de let

En scripts més vells, a vegades es troba una altra paraula clau: var en lloc de let:

```
var missatge = 'Hola';
```

La paraula clau var és quasi el mateix que let. També fa la declaració d'una variable, encara que d'un mode lleugerament diferent, i més antic.

Existeixen subtils diferències entre let i var, però no ens interessen en este moment.

5.3. Nomenament de variables

Existeixen dues limitacions de nom de variables en JavaScript:

- El nom únicament pot incloure lletres, dígit, o els símbols \$ i _.
- El primer caràcter no pot ser un dígit.

Exemples de noms vàlids:

```
let userName;  
let test123;
```

Quan el nom conté diverses paraules, se sol usar l'estil camelCase (capitalització en camell), on les paraules van pegades una darrere d'una altra, amb cada inicial en majúscula: miNombreMuyLargo.

És interessant notar que el símbol del dòlar '\$' i el guió sota '_' també s'utilitzen en noms. Són símbols comuns, tal com les lletres, sense cap significat especial.

Els següents noms són vàlids:

```
let $ = 1; // Declara una variable amb el nom "$"  
let _ = 2; // i ara una variable amb el nom "_"  
alert($ + _); // 3
```

5.4. Noms reservats

Hi ha una llista de paraules reservades, les quals no poden ser utilitzades com a nom de variable perquè el llenguatge en si les utilitza.

Per exemple: let, class, return, i function estan reservades.

5.5. Una assignació sense utilitzar use strict

Normalment, hem de definir una variable abans d'utilitzar-la. Però, en els vells temps, era tècnicament possible crear una variable simplement assignant un valor sense utilitzar 'let'. Això encara funciona si no posem 'use strict' en els nostres scripts per a mantindre la compatibilitat amb scripts antics.

```
// nota: no s'utilitza "use strict" en este exemple

num = 5; // es crea la variable "num" si no existeix abans

alert(num); // 5

Això és una mala pràctica que causaria errors en 'strict mode':

"use strict";

num = 5; // error: num no està definida
```

5.6. Constants

Per a declarar una variable constant (immutable) use const en comptes de let:

```
const myBirthday = '18.04.1982';
```

Les variables declarades utilitzant const es diuen "constants". No poden ser alterades. En intentar-ho causaria un error:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, no es pot reassignar la constant!
```

Quan un programador està segur que una variable mai canviarà, pot declarar la variable amb const per a garantir i comunicar clarament este fet a tots.

5.7.Constants majúscules

Existeix una pràctica utilitzada àmpliament d'utilitzar constants com aliars de valors difícils-de-recordar i que es coneixen previ a l'execució.

Tals constants es nomenen utilitzant lletres majúscules i guions baixos.

Per exemple, creiem constants per als colors en el format “web” (hexadecimal):

```
const COLOR_XARXA = "#F00";  
const COLOR_GREEN = "#0F0";  
const COLOR_BLUE = "#00F";  
const COLOR_ORANGE = "#FF7F00";  
// ...quan hem de triar un color  
let color = COLOR_ORANGE;  
alert(color); // #FF7F00
```

Avantatges:

- COLOR_ORANGE és molt més fàcil de recordar que "#FF7F00".
- És molt més fàcil escriure mal "#FF7F00" que COLOR_ORANGE.
- En llegir el codi, COLOR_ORANGE té molt més significat que #FF7F00.

5.8.Tipus de dades

Un valor en JavaScript sempre pertany a una mena de dada determinada. Per exemple, un string o un número.

Hi ha huit tipus de dades bàsiques en JavaScript. Podem emmagatzemar un valor de qualsevol tipus dins d'una variable. Per exemple, una variable pot contindre en un moment un string i després emmagatzemar un número:

```
// no hi ha error  
let message = "hola";  
message = 123456;
```

Els llenguatges de programació que permeten estes coses, com JavaScript, es denominen “dinàmicament tipados”, cosa que significa que allí hi ha tipus de dades, però les variables no estan vinculades rígidament a cap d'ells.

Number

```
let n = 123;  
n = 12.345;
```

El tipus number representa tant nombres enters com de punt flotant.

Hi ha moltes operacions per a números. Per exemple, multiplicació *, divisió /, suma +, resta -, i altra.

A més dels números comuns, existeixen els anomenats “valors numèrics especials” que també pertanyen a esta mena de dades: Infinity, -Infinity i NaN.

Infinity representa l'Infinit matemàtic ∞ . És un valor especial que és major que qualsevol número.

Podem obtindre-ho com a resultat de la divisió per zero:

```
alert( 1 / 0 ); // Infinity
```

O simplement fer referència a ell directament:

```
alert( Infinity ); // Infinity
```

NaN representa un error de càlcul. És el resultat d'una operació matemàtica incorrecta o indefinida, per exemple:

```
alert( "no és un número" / 2 ); // NaN, tal divisió és errònia
```

NaN és “enganxós”. Qualsevol altra operació sobre NaN retorna NaN:

```
alert( NaN + 1 ); // NaN  
alert( 3 * NaN ); // NaN  
alert( "not a number" / 2 - 1 ); // NaN
```

Per tant, si hi ha un NaN en alguna part d'una expressió matemàtica, es propaga a tot el resultat (amb una única excepció: NaN ** 0 és 1).

Els valors numèrics especials pertanyen formalment al tipus “número”. Per descomptat que no són números en el sentit estricte de la paraula.

BigInt

En JavaScript, el tipus “number” no pot representar de manera segura valors sencers majors que (253-1) (això és 9007199254740991), o menor que -(253-1) per a negatius.

BigInt es va agregar recentment al llenguatge per a representar enters de longitud arbitrària.

Un valor BigInt es crea agregant n al final d'un enter:

```
// la "n" al final significa que és un BigInt  
const bigint = 1234567890123456789012345678901234567890n;
```

String

Un string en JavaScript és una cadena de caràcters i ha de col·locar-se entre cometes.

```
let str = "Hola";  
let str2 = 'Les cometes simples també estan bé';  
let phrase = `es pot incrustar un altre ${str}`;
```

En JavaScript, hi ha 3 tipus de cometes.

- Cometes dobles: "Hola".
- Cometes simples: 'Hola'.
- Backticks (cometes invertides): `Hola`.

Les cometes dobles i simples són cometes “senzilles” (és a dir, funcionen igual). No hi ha diferència entre elles en JavaScript.

Els backticks són cometes de “funcionalitat estesa”. Ens permeten incrustar variables i expressions en una cadena de caràcters tancant-les en \${...}, per exemple:

```
let name = "John";  
// incrustar una variable  
alert( `Hola, ${name}!` ); // Hola, John!  
// incrustar una expressió  
alert( `el resultat és ${1 + 2}` ); //el resultat és 3
```

L'expressió dins de \${...} s'avalua i el resultat passa a formar part de la cadena. Podem posar qualsevol cosa ací dins: una variable com name, una expressió aritmètica com 1 + 2, o una cosa més complexa.

Té en compte que això només es pot fer amb els backticks. Les altres cometes no tenen esta capacitat d'incrustació!

Boolean (tipus lògic)

El tipus boolean té només dos valors possibles: true i false.

Este tipus s'utilitza comunament per a emmagatzemar valors de sí/no: true significa "sí, correcte, vertader", i false significa "no, incorrecte, fals".

Per exemple:

```
let nameFieldChecked = true; // sí, el camp name està marcat
let ageFieldChecked = false; // no, el camp age no està marcat
```

Els valors booleans també són el resultat de comparacions:

```
let isGreater = 4 > 1;
alert( isGreater ); // vertader (el resultat de la comparació és "sí")
```

El valor "null".

El valor especial null no pertany a cap dels tipus descrits anteriorment.

Forma un tipus propi separat que conté només el valor null:

```
let age = null;
```

En JavaScript, null no és una "referència a un objecte inexistent" o un "punter nul" com en altres llenguatges. És només un valor especial que representa "res", "buit" o "valor desconegut".

El codi anterior indica que el valor d'age és desconegut o està buit per alguna raó.

El valor "undefined" (indefinit)

El valor especial undefined també es distingeix. Fa un tipus propi, igual que null. El significat d'undefined és "valor no assignat". Si una variable és declarada, però no assignada, llavors el seu valor és undefined:

```
let age;
alert(age); // mostra "undefined"
```

Tècnicament, és possible assignar undefined a qualsevol variable:

```
let age = 100;
// canviant el valor a undefined
age = undefined;
alert(age); // "undefined"
```

...Però no recomanem fer això. Normalment, usem null per a assignar un valor "buit" o "desconegut" a una variable, mentre undefined és un valor inicial reservat per a coses que no han sigut assignades.

Object i Symbol

El tipus object (objecte) és especial.

Tots els altres tipus es diuen “primitius” perquè els seus valors poden contindre una sola cosa (ja siga una cadena, un número o el que siga). Per contra, els objectes s'utilitzen per a emmagatzemar col·leccions de dades i entitats més complexes.

El tipus symbol (símbol) s'utilitza per a crear identificadors únics per als objectes. Hem d'esmentar-ho ací per a una major integritat, però és millor estudiar este tipus després dels objectes.

L'operador typeof

L'operador typeof retorna el tipus de dada de l'operand. És útil quan volem processar valors de diferents tipus de manera diferent o simplement volem fer una comprovació ràpida.

L'anomenada a typeof x retorna una cadena amb el nom del tipus:

```
typeof undefined // "undefined"  
typeof 0 // "number"  
typeof 10n // "bigint"  
typeof true // "boolean"  
typeof "foo" // "string"  
typeof Symbol("aneu") // "symbol"  
typeof Math // "object" (1)  
typeof null // "object" (2)  
typeof alert // "function" (3)
```

Les últimes tres línies poden necessitar una explicació addicional:

- Math és un objecte incorporat que proporciona operacions matemàtiques.
- El resultat de typeof null és "object". Això està oficialment reconegut com un error de comportament de typeof que prové dels primers dies de JavaScript i es manté per compatibilitat. Definitivament null no és un objecte. És un valor especial amb un tipus propi separat.
- El resultat de typeof alert és "function" perquè alert és una funció. Estudiarem les funcions en els pròxims capítols on veurem que no hi ha cap tipus especial “function” en JavaScript. Les funcions pertanyen al tipus objecte. Tècnicament aquest comportament és incorrecte, però pot ser convenient en la pràctica.

6. INTERACCIÓ

Cobrim 3 funcions específiques del navegador per a interactuar amb els usuaris:

6.1.alert

Muestra un missatge.

```
alert("Hello");
```

La mini finestra amb el missatge es diu * finestra modal *. La paraula “modal” significa que el visitant no pot interactuar amb la resta de la pàgina, pressionar altres botons, etc., fins que s'haja ocupat de la finestra. En este cas, fins que pressionen “OK”.

6.2.prompt

Muestra un missatge demanant a l'usuari que introduísca un text. Retorna el text o, si es fa clic a CANCEL·LAR o es pressiona Esc, retorna null.

```
result = prompt(title, [default]);
```

6.3.confirm

mostra un missatge i espera al fet que l'usuari preme “OK” o “CANCEL·LAR”. Retorna true si es pressiona OK i false si es pressiona CANCEL·LAR/Esc.

Tots estos mètodes són modals: detenen l'execució del script i no permeten que l'usuari interactue amb la resta de la pàgina fins que la finestra s'haja tancat.

Hi ha dues limitacions comunes a tots els mètodes anteriors:

- La ubicació exacta de la finestra modal està determinada pel navegador. Normalment, està en el centre.
- L'aspecte exacte de la finestra també depén del navegador. No podem modificar-ho.

Eixe és el preu de la simplicitat. Existeixen altres maneres de mostrar finestres més atractives i interactives per a l'usuari, però si l'aparença no importa molt, estos mètodes funcionen bé.

7. CONVERSIONS DE TIPUS

La majoria de les vegades, els operadors i funcions converteixen automàticament els valors que se'ls passen al tipus correcte. Això és anomenat “conversió de tipus”.

Per exemple, `alert` converteix automàticament qualsevol valor a string per a mostrar-lo. Les operacions matemàtiques converteixen els valors a números.

També hi ha casos on necessitem convertir de manera explícita un valor al tipus esperat.

7.1.ToString

La conversió a string ocorre quan necessitem la representació en forma de text d'un valor.

Per exemple, `alert(value)` ho fa per a mostrar el valor com a text.

També podem cridar a la funció `String(value)` per a convertir un valor a string:

```
let value = true;
alert(typeof value); // boolean
value = String(value); // ara value és l'string "true"
alert(typeof value); // string
```

La conversió a string és bastant òbvia. El boolean `false` es converteix en `"false"`, `null` en `"null"`, etc.

7.2.ToNumber

La conversió numèrica ocorre automàticament en funcions matemàtiques i expressions.

Per exemple, quan es divideixen valors no numèrics usant `/`:

```
alert( "6" / "2" ); // 3, els strings són convertits a números
```

Podem usar la funció `Number(value)` per a convertir de manera explícita un valor a un número:

```
let str = "123";
alert(typeof str); // string
let num = Number(str); // es converteix en 123
alert(typeof num); // number
```

La conversió explícita és requerida usualment quan llegim un valor des d'una font basada en text, com ho són els camps de text en els formularis, però que esperem que continguin un valor numèric.

Si l'string no és un número vàlid, el resultat de la conversió serà NaN. Per exemple:

```
let age = Number("un text arbitrari en comptes d'un número");  
  
alert(age); // NaN, conversió fallida
```

Addició '+' concatena strings

Quasi totes les operacions matemàtiques converteixen valors a números. Una excepció notable és la suma +. Si un dels valors sumats és un string, l'altre valor és convertit a string.

Després, els concatena (uneix):

```
alert( 1 + '2' ); // '12' (string a la dreta)  
alert( '1' + 2 ); // '12' (string a l'esquerra)
```

Això ocorre només si almenys un dels arguments és un string, en cas contrari els valors són convertits a número.

7.3.ToBoolean

La conversió a boolean és la més simple.

Ocorre en operacions lògiques (més endavant veurem test condicionals i altres coses similars), però també pot realitzar-se de manera explícita cridant a la funció Boolean(value).

Les regles de conversió:

- Els valors que són intuïtivament “buits”, com 0, "", null, undefined, i NaN, es converteixen en false.
- Altres valors es converteixen en true.

Per exemple :

```
alert( Boolean(1) ); // true  
alert( Boolean(0) ); // false  
alert( Boolean("hola") ); // true  
alert( Boolean("") ); // false  
  
Tingues en compte: l'string amb un zero "0" és true  
Alguns llenguatges (com PHP) tracten "0" com false. Però en JavaScript, un string no buit és sempre true.  
  
alert( Boolean("0") ); // true  
alert( Boolean(" ") ); // només espais, també true
```

8. OPERADORS BÀSICS I MATEMÀTICS

8.1.Termes: “unari”, “binari”, “operant”

Abans de continuar, compreguem la terminologia comuna.

Un operand – és al que s'apliquen els operadors. Per exemple, en la multiplicació de 5 * 2 hi ha dos operands: l'operand esquerre és 5 i l'operand dret és 2. A vegades, la gent els diu “arguments” en lloc de “operands”.

Un operador és unari si té un sol operant. Per exemple, la negació unària - inverteix el signe d'un número:

```
let x = 1;  
x = -x;  
alert( x ); // -1, es va aplicar negació unària
```

Un operador és binari si té dos operands. El mateix negatiu també existeix en forma binària:

```
let x = 1, i = 3;  
alert( i - x ); // 2, binari negatiu resta valors
```

Formalment, estem parlant de dos operadors diferents: la negació unària (un operand: reverteix el símbol) i la resta binària (dos operands: resta).

8.2.Matemàtiques

Estan suportades les següents operacions:

- Suma +,
- Resta -,
- Multiplicació *,
- Divisió /,
- Resta %,
- Exponenciación **.

Els primers quatre són coneguts mentre que % i ** han de ser explicats més àmpliament.

Resta %

L'operadora resta %, malgrat la seua aparença, no està relacionat amb percentatges.

El resultat de $a \% b$ és la resta de la divisió sencera de a per b .

Per exemple:

```
alert( 5 % 2 ); // 1, és la resta de 5 dividit per 2
alert( 8 % 3 ); // 2, és la resta de 8 dividit per 3
alert( 8 % 4 ); // 0, és la resta de 8 dividit per 4
```

Exponenciación **

L'operador exponenciación $a ** b$ eleva a a la potència de b .

En matemàtiques de l'escola, ho escrivim com a^b .

Per exemple:

```
alert( 2 ** 2 ); // 22 = 4
alert( 2 ** 3 ); // 23 = 8
alert( 2 ** 4 ); // 24 = 16
```

Matemàticament, l'exponenciación està definida per a operadors no sencers també.

Per exemple, l'arrel quadrada és l'exponent $\frac{1}{2}$:

```
alert( 4 ** (1/2) ); // 2 (potència de 1/2 és el mateix que arrel quadrada)
alert( 8 ** (1/3) ); // 2 (potència de 1/3 és el mateix que arrel cúbica)
```

Concatenació de cadenes amb el binari +

Ara vegem les característiques dels operadors de JavaScript que van més enllà de l'aritmètica escolar.

Normalment l'operador $+$ suma números.

Però si s'aplica el $+$ binari a una cadena, els uneix (concatena):

```
let s = "my" + "string";
alert(s); // mystring
```

Tinga present que si un dels operands és una cadena, l'altre és convertit a una cadena també.

Per exemple:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Van veure, no importa si el primer operant és una cadena o el segon.

Precedència de l'operador

Si una expressió té més d'un operador, l'ordre d'execució es defineix per la seua precedència o, en altres paraules, l'ordre de prioritat predeterminat dels operadors.

Des de l'escola, tots sabem que la multiplicació en l'expressió $1 + 2 * 2$ ha de calcular-se abans de la suma. Això és exactament la precedència. Es diu que la multiplicació té una major precedència que la suma.

Els parèntesis anul·len qualsevol precedència, per la qual cosa si no estem satisfets amb l'ordre predeterminat, podem usar-los per a canviar-lo. Per exemple, escriu $(1 + 2) * 2$.

Hi ha molts operadors en JavaScript. Cada operador té un número de precedència corresponent. El que té el número més gran s'executa primer. Si la precedència és la mateixa, l'ordre d'execució és d'esquerra a dreta.

Ací hi ha un extracte de la taula de precedència (no necessita recordar això, però tinga en compte que els operadors unaris són més alts que l'operador binari corresponent):

Precedencia	Nombre	Signo
...
14	suma unaria	+
14	negación unaria	-
13	exponenciación	**
12	multiplicación	*
12	división	/
11	suma	+
11	resta	-
...
2	asignación	=
...

Assignació

Tinguem en compte que una assignació = també és un operador. Està llistat en la taula de precedència amb la prioritat molt baixa de 2.

És per això que, quan assignem una variable, com a $x = 2 * 2 + 1$, els càlculs es realitzen primer i després s'avalua el `=`, emmagatzemant el resultat en `x`.

```
let x = 2 * 2 + 1;  
alert( x ); // 5
```

Modificar en el lloc

Sovint necessitem aplicar un operador a una variable i guardar el nou resultat en eixa mateixa variable.

Per exemple:

```
let n = 2;  
n = n + 5;  
n = n * 2;
```

Esta notació pot ser acurtada utilitzant els operadors `+=` i `*=`:

```
let n = 2;  
n += 5; // ara n = 7 (és el mateix que n = n + 5)  
n *= 2; // ara n = 14 (és el mateix que n = n * 2)  
alert( n ); // 14
```

Els operadors curts “modifica i assigna” existeixen per a tots els operadors aritmètics i de nivell bit: `/=`, `-=`, etcètera.

Tals operadors tenen la mateixa precedència que l'assignació normal, per tant s'executen després d'altres càlculs:

```
let n = 2;  
n *= 3 + 5; // el costat dret és avaluat primer, és el mateix que n *= 8  
alert( n ); // 16
```

Incremente/decrement

Augmentar o disminuir un número en un és una de les operacions numèriques més comunes.

Llavors, hi ha operadors especials per a això:

- Increment ++ incrementa una variable per 1:

```
let counter = 2;  
counter++; // funciona igual que counter = counter + 1, però és més curt  
alert( counter ); // 3
```

- Decrement -- decrementa una variable per 1:

```
let counter = 2;  
counter--; // funciona igual que counter = counter - 1, però és més curt  
alert( counter ); // 1
```

Operadors a nivell de bit

Els operadors a nivell bit tracten els arguments com a nombres enters de 32 bits i treballen en el nivell de la seua representació binària.

Estos operadors no són específics de JavaScript. Són compatibles amb la majoria dels llenguatges de programació.

La llista d'operadors:

- AND (&)
- OR (|)
- XOR (^)
- NOT (~)
- LEFT SHIFT (<<)
- RIGHT SHIFT (>>)
- ZERO-FILL RIGHT SHIFT (>>>)

Estos operadors s'usen molt rarament, quan necessitem manejar la representació de números en el seu més baix nivell. No tenim en vista usar-los prompte perquè en el desenvolupament web té poc ús; però en unes certes àrees especials, com la criptografia, són útils. Pots llegir l'article Operadors a nivell de bit en MDN quan sorgisca la necessitat.

9. COMPARACIONS

Coneixem molts operadors de comparació de les matemàtiques:

En JavaScript s'escriuen així:

- Major/menor que: $a > b$, $a < b$.
- Major/menor o igual que: $a \geq b$, $a \leq b$.
- Igual: $a == b$ (tingues en compte que el doble signe $==$ significa comparació, mentre que un sol símbol $a = b$ significaria una assignació).
- Diferent. En matemàtiques la notació és \neq , però en JavaScript s'escriu com una assignació amb un signe d'exclamació davant: $a != b$.

9.1. Booleà és el resultat

Com tots els altres operadors, una comparació retorna un valor. En este cas, el valor és un booleà.

- `true` – significa “sí”, “correcte” o “veritat”.
- `false` – significa “no”, “equivocat” o “no veritat”.

Per exemple:

```
alert( 2 > 1 ); // true (correcte)
alert( 2 == 1 ); // false (incorrecte)
alert( 2 != 1 ); // true (correcte)
```

El resultat d'una comparació pot assignar-se a una variable, igual que qualsevol valor:

```
let result = 5 > 4; // assignar el resultat de la comparació
alert( result ); // true
```

9.2. Comparació de cadenes

Per a veure si una cadena és “major” que una altra, JavaScript utilitza l'anomenat ordre “de diccionari” o “lexicogràfic”.

En altres paraules, les cadenes es comparen lletra per lletra.

Per exemple:

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

L'algorisme per a comparar dues cadenes és simple:

Compare el primer caràcter de totes dues cadenes.

Si el primer caràcter de la primera cadena és major (o menor) que el de l'altra cadena, llavors la primera cadena és major (o menor) que la segona. Hem acabat.

En cas contrari, si els primers caràcters de totes dues cadenes són els mateixos, compare els segons caràcters de la mateixa manera.

Repetisca fins al final de cada cadena.

Si totes dues cadenes tenen la mateixa longitud, llavors són iguals. En cas contrari, la cadena més llarga és major.

En els exemples anteriors, la comparació 'Z' > 'A' arriba a un resultat en el primer pas.

9.3. Comparació de diferents tipus

En comparar valors de diferents tipus, JavaScript converteix els valors a números.

Per exemple:

```
alert( '2' > 1 ); // true, la cadena '2' es converteix en el número 2  
alert( '01' == 1 ); // true, la cadena '01' es converteix en el número 1
```

Per a valors booleans, true es converteix en 1 i false en 0.

Per exemple:

```
alert( true == 1 ); // true  
alert( false == 0 ); // true
```

9.4. Igualtat estricta

Una comparació regular d'igualtat == té un problema. No pot diferenciar 0 de 'fals':

```
alert( 0 == false ); // true
```

El mateix succeeix amb una cadena buida:

```
alert( "" == false ); // true
```

Això succeeix perquè els operands de diferents tipus són convertits a números per l'operador d'igualtat ==. Una cadena buida, igual que false, es converteix en un zero.

Què fer si volem diferenciar 0 de false?

Un operador d'igualtat estricte === comprova la igualtat sense conversió de tipus.

En altres paraules, si a i b són de diferents tipus, llavors a === b retorna immediatament false sense intentar convertir-los.

9.5.Comparació amb nuls i indefinits

Hi ha un comportament no intuïtiu quan es compara null o undefined amb altres valors.

Para un control d'igualtat estricte ===

Estos valors són diferents, perquè cadascun d'ells és d'un tipus diferent.

```
alert( null === undefined ); // false
```

Per a una comparació no estricta ==

Hi ha una regla especial. Estos dos són una " parella dolça ": són iguals entre si (en el sentit de ==), però no a cap altre valor.

```
alert( null == undefined ); // true
```

10. CONDICIONALS

10.1. La sentència "if"

La sentència `if(...)` avalua la condició en els parèntesis, i si el resultat és vertader (`true`), executa un bloc de codi.

Per exemple:

```
let year = prompt('En què any va anar publicada l'especificació ECMAScript?', '');  
if (year == 2015) alert( 'Tens raó!' );
```

Ací la condició és una simple igualtat (`year == 2015`), però podria ser molt més complexa.

Si volem executar més d'una sentència, hem de tancar el nostre bloc de codi entre claus:

```
if (year == 2015) {  
    alert( "És Correcte!" );  
    alert( "Eres molt intel·ligent!" );  
}
```

Recomanem tancar el nostre bloc de codi entre claus `{}` sempre que s'utilitzi la sentència `if`, fins i tot si només s'executarà una sola sentència. En fer-ho millorem la llegibilitat.

Conversió Booleana

La sentència `if (...)` avalua l'expressió dins dels seus parèntesis i converteix el resultat en booleà.

Llavors, el codi sota esta condició mai s'executaria:

```
if (0) { // 0 és fals  
    ...  
}
```

...i dins d'esta condició sempre s'executarà:

```
if (1) { // 1 és vertader  
    ...  
}
```

10.2. La clàusula “else”

La sentència if pot contindre un bloc else (“si no”, “en cas contrari”) opcional. Este bloc s'executarà quan la condició siga falsa.

Per exemple:

```
let year = prompt('En quin any va ser publicada l'especificació ECMAScript?', '');
if (year == 2015) {
  alert( 'Ho vas endevinar, correcte!' );
} else {
  alert( 'Com pots estar tan equivocat?' ); // qualsevol valor excepte 2015
}
```

10.3. Moltes condicions: “else if”

A vegades volem provar més d'una condició. La clausula else if ens permet fer això.

Per exemple:

```
let year = prompt('En quin any va ser publicada l'especificació ECMAScript?', '');

if (year < 2015) {
  alert( 'Molt poc...' );
} else if (year > 2015) {
  alert( 'Molt Tard' );
} else {
  alert( 'Exactament!' );
}
```

En el codi de dalt, JavaScript primer revisa si `year < 2015`. Si això és fals, continua a la següent condició `year > 2015`. Si esta també és falsa, mostrarà l'última alert.

Podria haver-hi més blocs else if. I l'últim else és opcional.

10.4. Operador ternari '?'

A vegades necessitem que el valor que assignem a una variable depenga d'alguna condició.

Per exemple :

```
let accessAllowed;  
let age = prompt('Quina edat tens?', '');  
  
if (age > 18) {  
    accessAllowed = true;  
} else {  
    accessAllowed = false;  
}  
  
alert(accessAllowed);
```

El “operador condicional” ens permet executar això en una forma més curta i simple.

L'operador està representat pel signe de tancament d'interrogació ?. A vegades és anomenat “ternari” perquè l'operador té tres operands, és l'únic operador de JavaScript que té eixa quantitat.

La Sintaxi és:

```
let result = condition ? value1 : value2;
```

S'avalua condition: si és vertadera llavors retorna value1 , en cas contrari value2.

Per exemple :

```
let accessAllowed = (age > 18) ? true : false;
```

11. OPERADORS LÒGICS

Hi ha quatre operadors lògics en JavaScript: || (O), && (I), ! (NO).

Encara que siguin anomenats lògics, poden ser aplicats a valors de qualsevol tipus, no sols booleans. El resultat també pot ser de qualsevol tipus.

11.1. || (OR)

L'operador OR es representa amb dos símbols de línia vertical:

```
result = a || b;
```

En la programació clàssica, l'OR lògic està pensat per a manipular sol valors booleans. Si qualsevol dels seus arguments és true, retorna true, en cas contrari retorna false.

En JavaScript, l'operador és un poc més complicat i poderós. Però primer, vegem què passa amb els valors booleans.

Hi ha quatre combinacions lògiques possibles:

```
alert(true || true); // true (vertader)
alert(false || true); // true
alert(true || false); // true
alert(false || false); // false (fals)
```

Com podem veure, el resultat és sempre true excepte quan tots dos operands són false.

Si un operand no és un booleà, li ho converteix a booleà per a l'avaluació.

Per exemple, el número 1 és tractat com true, el número 0 com false:

```
if (1 || 0) { // Funciona com if( true || false )
  alert("valor vertader!");
}
```

La majoria de les vegades, OR || és usat en una declaració if per a provar si alguna de les condicions donades és true.

Per exemple :

```
let hour = 9;
if (hour < 10 || hour > 18) {
  alert( 'L'oficina esta tancada.' );
}
```

11.2. && (AND)

L'operador AND és representat amb dos ampersands &&:

```
result = a && b;
```

En la programació clàssica, AND retorna true si tots dos operands són valors vertaders i false en qualsevol altre cas.

```
alert(true && true); // true  
alert(false && true); // false  
alert(true && false); // false  
alert(false && false); // false
```

Un exemple amb if:

```
let hour = 12;  
let minute = 30;  
  
if (hour == 12 && minute == 30) {  
  alert("L'hora és 12.30");  
}
```

Igual que amb OR, qualsevol valor és permès com operant d'AND:

```
if (1 && 0) { // avaluat com true && false  
  alert( "no funcionarà perquè el resultat és un valor fals" );  
}
```

11.3.! (NOT)

L'operador booleà NOT es representa amb un signe d'exclamació !.

La sintaxi és bastant simple:

```
result = !value;
```

L'operador accepta un sol argument i realitza el següent:

- Converteix l'operand al tipus booleà: true/false.
- Retorna el valor contrari.

Per exemple:

```
alert(!true); // false  
alert(!0); // true
```

Un doble NOT !! és a vegades usat per a convertir un valor al tipus booleà:

```
alert(!!"cadena de text no buida"); // true  
alert(!!null); // false
```

Això és, el primer NOT converteix el valor a booleà i retorna l'invers, i el segon NOT ho inverteix de nou. Al final, tenim una simple conversió a booleà.

Hi ha una manera un poc mes prolixa de realitzar el mateix – una funció integrada Boolean:

```
alert(Boolean("cadena de text no buida")); // true  
alert(Boolean(null)); // false
```

La precedència de NOT ! és la major de tots els operadors lògics, així que sempre s'executa primer, abans que && o | |.

12. BUCLES: WHILE I FOR

Usualment necessitem repetir accions.

Per exemple, mostrar els elements d'una llista un darrere l'altre o simplement executar el mateix codi per a cada número de l'1 al 10.

Els Bucles són una manera de repetir el mateix codi diverses vegades.

Els bucles for...of i for...in

12.1.El bucle “while”

El bucle while (mentre) té la següent sintaxi:

```
while (condition) {  
  // codi  
  // anomenat "cos del bucle"  
}
```

Mentre la condició condition siga vertadera, el codi del cos del bucle serà executat.

Per exemple, el bucle davall imprimeix i mentre es complisca $i < 3$:

```
let i = 0;  
while (i < 3) { // mostra 0, després 1, després 2  
  alert( i );  
  i++;  
}
```

Cada execució del cos del bucle es diu iteració. El bucle en l'exemple de dalt realitza 3 iteracions.

Si faltara `i++` en l'exemple de dalt, el bucle seria repetit (en teoria) eternament. En la pràctica, el navegador té maneres de detindre tals bucles desmesurats; i en el JavaScript del costat del servidor, podem eliminar el procés.

Qualsevol expressió o variable pot usar-se com a condició del bucle, no sols les comparacions: El while avaluarà i transformarà la condició a un booleà.

12.2.El bucle “do...while”

La comprovació de la condició pot ser moguda davall del cos del bucle usant la sintaxi do..while:

```
do {  
    // cos del bucle  
} while (condition);
```

El bucle primer executa el cos, després comprova la condició, i, mentre siga un valor vertader, l'executa una vegada i una altra.

Per exemple :

```
let i = 0;  
do {  
    alert( i );  
    i++;  
} while (i < 3);
```

Esta sintaxi només ha de ser usada quan vols que el cos del bucle siga executat almenys una vegada sense importar que la condició siga vertadera. Usualment, es prefereix l'altra forma: while(...) {...}.

12.3.El bucle “for”

El bucle for és més complex, però també el més usat.

Es veu així:

```
for (begin; condition; step) { // (començament, condició, pas)  
    // ... cos del bucle ...  
}
```

Aprenuem el significat de cada part amb un exemple. El bucle davall corre alert(i) per a i des de 0 fins a (però no incloent) 3:

```
for (let i = 0; i < 3; i++) { // mostra 0, després 1, després 2  
    alert(i);  
}
```

Declaració de variable en línia

Ací, la variable “counter” i és declarada en el bucle. Això és anomenat una declaració de variable “en línia”. Aquestes variables són visibles solo dins del bucle.

```
for (let i = 0; i < 3; i++) {  
  alert(i); // 0, 1, 2  
}  
alert(i); // error, no existeix aquesta variable
```

En comptes de definir una variable, podem usar una que ja existisca:

```
let i = 0;  
for (i = 0; i < 3; i++) { // usa una variable existent  
  alert(i); // 0, 1, 2  
}  
alert(i); // 3, visible, perquè va ser declarada fora del bucle
```

12.4. Trencant el bucle

Normalment, s'ix d'un bucle quan la condició es torna falsa.

Però podem forçar una eixida en qualsevol moment usant la directiva especial break.

Per exemple, el bucle davall li demana a l'usuari per una sèrie de números, “trencant-lo” quan un número no és ingressat:

```
let sum = 0;  
while (true) {  
  let value = +prompt("Ingressa un número", "");  
  if (!value) break; // (*)  
  sum += value;  
}  
alert( 'Suma: ' + sum );
```

La directiva break és activada en la línia (*) si l'usuari ingressa una línia buida o cancel·la l'entrada. Deté immediatament el bucle, passant el control a la primera línia després del bucle. En este cas, alert.

La combinació “bucle infinit + break segons siga necessari” és ideal en situacions on la condició del bucle ha de ser comprovada no a l'inici o al final del bucle, sinó a la meitat o fins i tot en diverses parts del cos.

12.5. Continuar a la següent iteració

La directiva `continue` és una “versió més lleugera” de `break`. No deté el bucle complet. En el seu lloc, deté la iteració actual i força al bucle a començar una nova (si la condició ho permet).

Podem usar-ho si hem acabat amb la iteració actual i ens agradaria moure'ns a la següent.

El bucle davall usa `continue` per a mostrar sol valors imparells:

```
for (let i = 0; i < 10; i++) {  
  // si és vertader, saltar la resta del cos  
  if (i % 2 == 0) continue;  
  alert(i); // 1, després 3, 5, 7, 9  
}
```

Per als valors parells d'`i`, la directiva `continue` deixa d'executar el cos i passa el control a la següent iteració de `for` (amb el següent número). Així que l'`alert` només és anomenat per a valors imparells.

13. LA SENTÈNCIA "SWITCH"

Una sentència switch pot reemplaçar múltiples condicions if.

Proveeix una millor manera de comparar un valor amb múltiples variants.

13.1. La sintaxi

switch té un o mes blocs casey un opcional default.

Es veu d'esta manera:

```
switch(x) {  
  case 'valor1': // if (x === 'valor1')  
    ...  
    [break]  
  case 'valor2': // if (x === 'valor2')  
    ...  
    [break]  
  default:  
    ...  
    [break]  
}
```

El valor de x és comparat contra el valor del primer case (en este cas, valor1), després contra el segon (valor2) i així successivament, tot això sota una igualtat estricta.

Si la igualtat és trobada, switch comença a executar el codi iniciant pel primer case corresponent, fins al break més pròxim (o fins al final del switch).

Si no es compleix cap cas llavors el codi default és executat (si existeix).

Exemple

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Molt xicotet' );
    break;
  case 4:
    alert( 'Exacte!' );
    break;
  case 5:
    alert( 'Molt gran' );
    break;
  default:
    alert( "Desconec estos valores" );
}
```

Ací el switch inicia comparant a amb la primera variant case que és 3. La comparació falla.

Després 4. La comparació és reeixida, per tant l'execució comença des de case 4 fins al break més pròxim.

Si no existeix break llavors l'execució continua amb el pròxim case sense cap revisió.

Un exemple sense break:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Molt xicotet' );
  case 4:
    alert( 'Exacte!' );
  case 5:
    alert( 'Molt gran' );
  default:
    alert( "Desconec estos valores" );
}
```

En l'exemple anterior veurem execucions de tres alert seqüencials:

```
alert( 'Exacte!' );
alert( 'Molt gran' );
alert( "Desconec estos valores" );
```

13.2. Agrupament de “case”

Diverses variants de case els quals comparteixen el mateix codi poden ser agrupades.

Per exemple, si volem que s'execute el mateix codi per a case 3 i case 5:

```
let a = 2 + 2;
switch (a) {
  case 4:
    alert('Correcte!');
    break;
  case 3: // (*) agrupant dos cases
  case 5:
    alert('Incorrecte!');
    alert("Per què no prens una classe de matemàtiques?");
    break;
  default:
    alert('El resultat és estrany. Realment.');
```

Ara tots dos, 3 i 5, mostren el mateix missatge.

La capacitat de “agrupar” els case és un efecte secundari de com treballa switch/case sense break. Ací l'execució de case 3 inicia des de la línia (*) i continua a través de case 5, perquè no existeix break.

14. FUNCIONS

Molt sovint necessitem realitzar accions similars en molts llocs del script.

Per exemple, hem de mostrar un missatge atractiu quan un visitant inicia sessió, tanca sessió i tal vegada en altres moments.

Les funcions són els principals “blocs de construcció” del programa. Permeten que el codi es digui moltes vegades sense repetició.

Ja hem vist exemples de funcions integrades, com `alert(message)`, `prompt(message, default)` i `confirm(question)`. Però també podem crear funcions pròpies.

14.1. Declaració de funcions

Per a crear una funció podem usar una declaració de funció.

Es veu com ací:

```
function showMessage() {  
    alert( 'Hola a tots!' );  
}
```

La paraula clau `function` va primer, després va el nom de funció, després una llista de paràmetres entre parèntesis (separats per comes, buida en l'exemple anterior) i finalment el codi de la funció entre claus, també dit “el cos de la funció”.

```
function name(parameter1, parameter2, ... parameterN) {  
    // bodi  
}
```

La nostra nova funció pot ser cridada pel seu nom: `showMessage()`.

Per exemple:

```
function showMessage() {  
    alert( 'Hola a tots!' );  
}  
  
showMessage();  
showMessage();
```

L'anomenada `showMessage()` executa el codi de la funció. Ací veurem el missatge dues vegades.

Este exemple demostra clarament un dels propòsits principals de les funcions: evitar la duplicació de codi...

14.2. Variables Locals

Una variable declarada dins d'una funció només és visible dins d'eixa funció.

Per exemple :

```
function showMessage() {  
  let message = "Hola, Soc JavaScript!"; // variable local  
  alert( message );  
}  
  
showMessage(); // Hola, Soc JavaScript!  
alert( message ); // <-- Error! La variable és local per a esta funció
```

14.3. Variables Externes

Una funció també pot accedir a una variable externa, per exemple:

```
let userName = 'Juan';  
  
function showMessage() {  
  let message = 'Hola, ' + userName;  
  alert(message);  
}  
  
showMessage(); // Hola, Juan
```

La funció té accés complet a la variable externa. Pot modificar-ho també.

Per exemple:

```
let userName = 'Juan';  
  
function showMessage() {  
  userName = "Bob"; // (1) Va canviar la variable externa  
  let message = 'Hola, ' + userName;  
  alert(message);  
}  
  
alert( userName ); // Juan abans de dir la funció  
showMessage();  
alert( userName ); // Bob, el valor va anar modificat per la funció
```

14.4. Paràmetres

Podem passar dades arbitràries a funcions usant paràmetres.

En el següent exemple, la funció té dos paràmetres: from i text.

```
function showMessage(from, text) { // paràmetres: from, text
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hola!'); // Ann: Hola! (*)
showMessage('Ann', "Com estàs?"); // Ann: Com estàs? (**)
```

Quan la funció es diu (*) i (**), els valors donats es copien en variables locals from i text. I la funció les utilitza.

Ací hi ha un exemple més: tenim una variable from i la passem a la funció. Tinga en compte: la funció canvia from, però el canvi no es veu fora, perquè una funció sempre obté una còpia del valor:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // fa que "from" es veja millor
  alert( from + ': ' + text );
}

let from = "Ann";
showMessage(from, "Hola"); // *Ann*: Hola

// el valor de "from" és el mateix, la funció va modificar una còpia local
alert( from ); // Ann
```

Quan un valor és passat com un paràmetre de funció, també es denomina argument.

14.5. Valors predeterminats

Si una funció és anomenada, però no se li proporciona un argument, el seu valor corresponent es converteix en `undefined`.

Per exemple, la funció esmentada anteriorment `showMessage(from, text)` es pot cridar amb un sol argument:

```
showMessage("Ann");
```

Això no és un error. La crida mostraria "Ann: undefined". Com no es passa un valor de text, este es torna `undefined`.

Podem especificar un valor anomenat "predeterminat" o "per defecte" (és el valor que s'usa si l'argument va ser omès) en la declaració de funció usant =:

```
function showMessage(from, text = "sense text") {  
    alert( from + ": " + text );  
}  
  
showMessage("Ann"); // Ann: sense text
```

Ara, si no es passa el paràmetre text, obtindrà el valor "sense text"

El valor predeterminat també s'assigna si el paràmetre existeix però és estrictament igual a `undefined`:

```
showMessage("Ann", undefined); // Ann: sense text
```

Ací "sense text" és un string, però pot ser una expressió més complexa, la qual només és avaluada i assignada si el paràmetre falta. Llavors, això també és possible:

```
function showMessage(from, text = anotherFunction()) {  
    // anotherFunction() només s'executa si text no va ser assignat  
    // el seu resultat es converteix en el valor de text  
}
```

14.6. Paràmetres predeterminats alternatius

A vegades té sentit assignar valors predeterminats als paràmetres més tard, després de la declaració de funció.

Podem verificar si un paràmetre és passat durant l'execució de la funció comparant-lo amb undefined:

```
function showMessage(text) {  
  // ...  
  if (text === undefined) { // si falta el paràmetre  
    text = 'missatge buit';  
  }  
  alert(text);  
}  
showMessage(); // missatge buit
```

...O podem usar l'operador ||:

```
function showMessage(text) {  
  // si text és indefinida o falsa, l'estableix a 'buit'  
  text = text || 'buit';  
  ...  
}
```

14.7. Retornant un valor

Una funció pot retornar un valor al codi de crida com a resultat.

L'exemple més simple seria una funció que suma dos valors:

```
function sum(a, b) {  
  return a + b;  
}  
  
let result = sum(1, 2);  
alert( result ); // 3
```

La directiva return pot estar en qualsevol lloc de la funció. Quan l'execució ho aconsegueix, la funció es deté i el valor es retorna al codi de crida (assignat al resultat anterior).

Pot haver-hi molts return en una sola funció. Per exemple :

```
function checkAge(age) {  
  if (age > 18) {  
    return true;  
  } else {  
    return confirm('Tens permís dels teus pares?');  
  }  
}  
  
let age = prompt('Quina edat tens?', 18);  
if ( checkAge(age) ) {  
  alert( 'Accés atorgat' );  
} else {  
  alert( 'Accés denegat' );  
}
```

És possible utilitzar return sense cap valor. Això fa que la funció isca o acabe immediatament.

Per exemple :

```
function showMovie(age) {  
  if ( !checkAge(age) ) {  
    return;  
  }  
  alert( "Mostrant-te la pel·lícula" ); // (*)  
  // ...  
}
```

En el codi de dalt, si checkAge(age) retorna false, llavors showMovie no mostrarà l'alert.

Una funció amb un return buit, o sense return, retorna undefined

Si una funció no retorna un valor, és el mateix que si retornara undefined:

```
function doNothing() { /* empty */ }  
alert( doNothing() === undefined ); // true
```

Un return buit també és el mateix que return undefined:

```
function doNothing() {  
  return;  
}  
alert( doNothing() === undefined ); // true
```