Sintaxi del llenguatge. Objectes predefinits del llenguatge

Joan Quintana

Índex

Introducció

Es pot pensar el món web com uns clients que consumeixen contingut web i uns servidors que subministren contingut. En el cantó del client se situa l'usuari i les eines que utilitza, per exemple, el navegador web. En el cantó del servidor hi ha els servidors web (per exemple, Apache Web Sever), que estan hostatjats en les empreses de *hosting*.

Tradicionalment, el navegador web (Mozilla Firefox, Chrome, Internet Explorer) s'ha encarregat d'interpretar el contingut HTML i maquetar-lo en forma de pàgina web. Avui dia, però, les pàgines web contenen un fort contingut d'interacció amb l'usuari i efectes visuals, cosa que fa la web molt atractiva i dinàmica. Aquests efectes i dinamisme, però, s'han de programar. Es codifiquen amb JavaScript, que és el codi que està inclòs dins els documents web i que és executat pel navegador web.

En aquesta unitat, "Sintaxi del llenguatge. Objectes predefinits del llenguatge", s'introdueix el funcionament general de la programació al costat client i el llenguatge de programació JavaScript. És el primer pas per poder aprofundir en la programació des del costat servidor. A la resta d'unitats s'anirà aprofundint en aspectes concrets d'aquest tipus de programació.

En l'apartat "Execució de programes al costat client. Introducció a JavaScript" farem una introducció al llenguatge JavaScript, comparant-lo amb d'altres llenguatges de programació i recalcant el paper que juga dins el marc de les tecnologies web. Introduirem la sintaxi bàsica, igual que s'ha de fer quan s'estudia qualsevol llenguatge de programació. Familiaritzar-se amb un nou llenguatge de programació pot resultar més o menys laboriós, en funció de l'experiència prèvia amb d'altres llenguatges de programació, però es considera que cada vegada que s'aprèn un nou llenguatge, la corba d'aprenentatge és més ràpida.

En l'apartat "Programació amb els objectes predefinits de JavaScript. BOM (*Browser Object Model*)" entrarem de ple en els objectes del llenguatge. D'una banda, els objectes predefinits (`String`, `Number`, `Math`, `Date` i `RegExp`), dels quals caldrà familiaritzar-se amb les principals propietats i mètodes. D'altra banda, els objectes relacionats amb el navegador web, que conformen el model d'objectes del navegador (BOM, *Browser Object Model*).

Finalment, en l'apartat "Programació amb finestres, marcs i galetes" aprofundirem més en el BOM i programareu amb galetes (*cookies*), finestres i marcs (*frames*). Aprendreu i fareu exercicis sobre la programació amb finestres i marcs, i l'intercanvi d'informació entre finestres i marcs.

A mesura que aneu aprenent JavaScript us anireu familiaritzant amb la sintaxi i el seu model orientat a objectes. Contínuament haureu d'accedir a propietats, mètodes i *events* (esdeveniments) associats a aquests objectes, que al cap i a la fi us permetran modificar el contingut web i interactuar-hi.

Al llarg de la unitat adquirireu, a més de conceptes i coneixements, procediments i metodologies que us permetran programar amb fluïdesa i claredat, així com depurar ràpidament els errors en el codi que segur que sorgiran.

Per superar la unitat amb èxit, haureu de provar tot el codi proposat, i tenir la iniciativa de fer modificacions sobre el codi, així com experimentar amb altres mètodes i propietats dels objectes que no necessàriament s'hauran treballat al material.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Selecciona les arquitectures i tecnologies de programació sobre clients web, identificant i analitzant les capacitats i característiques de cadascuna.

- Caracteritza i diferencia els models d'execució de codi al servidor i al client web.
- Identifica les capacitats i els mecanismes d'execució de codi dels navegadors web.
- Identifica i caracteritza els principals llenguatges relacionats amb la programació de clients web.
- Reconeix les particularitats de la programació de guions i els seus avantatges i desavantatges sobre la programació tradicional.
- Verifica els mecanismes d'integració dels llenguatges de marques amb els llenguatges de programació de clients web.
- Reconeix i avalua les eines de programació sobre clients web.

2. Escriu sentències simples, aplicant la sintaxi del llenguatge i verificant la seva execució sobre navegadors web.

- Selecciona un llenguatge de programació de clients web en funció de les seves possibilitats.
- Utilitza els diferents tipus de variables i operadors disponibles en el llenguatge.
- Identifica els àmbits d'utilització de les variables.
- Reconeix i comprova les peculiaritats del llenguatge respecte a les conversions entre diferents tipus de dades.
- Utilitza mecanismes de decisió en la creació de blocs de sentències.
- Utilitza bucles i verifica el seu funcionament.

3. Escriu codi, identificant i aplicant les funcionalitats aportades pels objectes predefinits del llenguatge.

- Identifica els objectes predefinits del llenguatge.
- Analitza els objectes referents a les finestres del navegador i els documents web que hi contenen.

- Escriu sentències que utilitzin els objectes predefinits del llenguatge per canviar l'aspecte del navegador i el document que hi conté.
- Genera textos i etiquetes com a resultat de l'execució de codi al navegador.
- Escriu sentències que utilitzin els objectes predefinits del llenguatge per interactuar amb l'usuari.
- Utilitza les característiques pròpies del llenguatge en documents compostos per diverses finestres i marcs.
- Utilitza galetes (*cookies*) per emmagatzemar informació i recuperar-ne el contingut.
- Depura i documenta el codi.

1. Execució de programes al costat client. Introducció a JavaScript

En l'inici de l'era d'Internet, les pàgines web eren bàsicament estàtiques, mostraven un contingut fix sense possibilitat d'interacció amb l'usuari. Però des de l'aparició de la Web 2.0, si una cosa defineix el món web és l'increment d'interacció i usabilitat, millorant l'experiència de l'usuari en la navegació. És aquí on el llenguatge JavaScript té un rol important. La programació en el cantó del client codificada dins les pàgines web és la responsable de fer pàgines web atractives tal com les coneixem avui dia.

Els autors de JavaScript van proposar el 1997 a la European Computer Manufacturers Association (ECMA) que el seu llenguatge s'adoptés com a estàndard. Així és com va néixer el ECMAScript, que avui dia és estàndard ISO. En els inicis d'Internet hi havia bastants problemes de compatibilitat entre navegadors (Internet Explorer, Netscape Navigator, Opera) a diferents nivells, i això va fer prendre consciència de la importància en l'adopció d'estàndards. El World Wide Web Consortium (WWWC) és una organització que es cuida de vetllar per la implementació d'estàndards en l'àmbit web, fent possible que diferents fabricants de programari (*software*) es posin d'acord en benefici de l'usuari final.

Per tal d'avançar en la compatibilitat entre navegadors web, no només és important que implementin el motor de JavaScript segons els estàndards ECMAScript, sinó que els diferents elements que hi ha en una web (botons, caixes de text, enllaços...) es comportin de la mateixa manera i responguin als mateixos *events* (esdeveniments). És per això que el WWWC va definir el Document Object Model (DOM, o Model d'Objectes del Document), que defineix un estàndard d'objectes per representar documents HTML i/o XML.

1.1 Introducció al llenguatge JavaScript

JavaScript és un llenguatge de guions (script, en anglès), implementat originàriament per Netscape Communications Corporation, i que va derivar en l'estàndard ECMAScript. És conegut sobretot pel seu ús en pàgines web, però també es pot utilitzar per realitzar tasques de programació i administració que no tenen res a veure amb la web.

Malgrat el seu nom, JavaScript no deriva del llenguatge de programació Java, tot i que tots dos comparteixen una sintaxi similar. Semànticament, JavaScript és més pròxim al llenguatge Self (basat també en l'ECMAScript).

El nom JavaScript és una marca registrada per Oracle Corporation.

Avui dia JavaScript és un llenguatge de programació madur, de manera que sobre d'ell s'han implementat diferents tecnologies, capes d'abstracció, biblioteques i

frameworks (entorns de treball) que amaguen els rudiments bàsics del llenguatge. Com a tecnologia podem parlar d'AJAX, un estàndard per intercanviar informació entre el client i el servidor web, que agilitza la comunicació i l'ample de banda, i millora l'experiència d'usuari. Quant a biblioteques hem de parlar de JQuery, que porta la programació amb JavaScript a un esglaó superior, i d'aquesta manera el programador web pot obtenir resultats espectaculars amb poques línies de codi. Quant a *frameworks*, el més utilitzat avui dia és Angular JS.

Cal tenir en compte que aquests materials pedagògics s'han redactat d'acord amb l'edició ES2020, encara que la major part dels exemples són compatibles amb les edicions ES2015 i posteriors.

El mes de juny del 2015 es va publicar l'edició ES2015 del llenguatge (inicialment conegut com a ES6). A partir d'aquesta edició, es van publicant noves actualitzacions del llenguatge cada any, actualitzades amb el nom de l'any corresponent: ES2015, ES2016, ES2017, etc. Tot i que algunes de les característiques publicades en l'especificació no són implementades per tots els navegadors, és d'esperar que totes siguin funcionals pròximament.

Aquesta edició conté canvis molt importants i afegeix nova sintaxi per desenvolupar aplicacions complexes, incloent-hi la declaració de classes i mòduls.

ECMA-262 5a edició

També es coneix com a JavaScript 5. És l'edició anterior a ES2015, i la seva especificació va ser publicada el desembre del 2009.

Tot i que els navegadors moderns admeten sense problemes ES2015, encara hi ha molt programari desenvolupat que utilitza la sintaxi d'ECMA-262 5a edició, ja que les noves edicions són compatibles amb les anteriors. És a dir, és possible actualitzar una aplicació implementada en una versió anterior afegint nou codi que faci servir les funcionalitats afegides en una versió posterior com ara ES2020.

Podeu trobar un resum de programació amb ECMA-262 5a edició les "Referències" d'aquesta unitat.

1.2 Llenguatges compilats i interpretats. Llenguatges de guions

Els llenguatges de programació es divideixen, quant a la manera d'executar-se, entre els **interpretats** i els **compilats**. Alguns llenguatges interpretats s'anomenen també *llenguatges de guions* (*scripts* en anglès).

Els llenguatges interpretats s'executen mitjançant un intèrpret, que processa les ordres que inclou el programa una a una. Això fa que els llenguatges interpretats siguin menys eficients en temps que els compilats (típicament 10 vegades més lents), ja que la interpretació de l'ordre s'ha de fer en temps d'execució. Com a avantatges podem dir que són més fàcils de depurar i són més flexibles per aconseguir una independència entre plataformes (portabilitat).

Exemples de llenguatges interpretats: JavaScript, Python, Perl, PHP, Bash...

D'altra banda, els llenguatges **compilats** necessiten del compilador com a pas previ a l'execució. Com a resultat de compilar el codi font (les instruccions que ha codificat el programador) s'obté el *codi màquina* (codi binari que és proper al microprocessador), i això fa que la seva execució sigui eficient en temps.

Exemples de llenguatges compilats: C, C++, Pascal, Fortran...

Per veure la diferència, mostrarem un exemple amb JavaScript (llenguatge interpretat) i un exemple amb llenguatge C (llenguatge compilat).

Amb JavaScript, calculem el factorial de tres números, els sumem, i mostrem la suma en la pantalla del navegador, dins d'una etiqueta HTML que ens serveix de contenidor:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Funció factorial recursiva</title>
    </head>
    <body>
        <h1>Funció factorial recursiva</h1>
        <p id="p1"></p>
        <script>

            let num1 = factorialRecursiu (5);
            let num2 = factorialRecursiu (6);
            let num3 = factorialRecursiu (7);

            document.getElementById("p1").innerHTML =  num1+num2+num3;

            function factorialRecursiu (n) {
                if (n == 0){
                    return 1;
                }
                return n * factorialRecursiu (n-1);
            }
        </script>
    </body>
</html>
```

Podeu provar el darrer exemple a: codepen.io/ioc-daw-m06/pen/dyGwGBd

Amb llenguatge C podem fer una petita aplicació equivalent, compilant el codi *factorialRecursiu.c*:

```
#include<stdio.h>

int factorialRecursiu (int n)
{
  int r;
  if (n==1) return 1;
  r=n*factorialRecursiu(n-1 ) ;
  return (r ) ;
}

int main()
{
  int res = 0;
  printf("Factorial Recursiu\n" ) ;;
  res += factorialRecursiu(5);
  res += factorialRecursiu(6);
  res += factorialRecursiu(7);
  printf("La suma és %d\n",res) ;
  return 0;
}
```

Per compilar-lo, executem en la línia d'ordres:

```
gcc -o factorialRecursiu factorialRecursiu.c
```

Es generarà l'executable *factorialRecursiu*. Aquest és un fitxer binari, per tant no el podem obrir com un document de text. El seu contingut és el codi objecte que sap executar el processador. És un codi optimitzat i ràpid. Segur que el càlcul per trobar els tres factorials i sumar-los es fa de manera molt més ràpida que en JavaScript. Per executar-lo:

```
$ ./factorialRecursiu

Factorial Recursiu
La suma és 5880
```

El procés de compilació amb *gcc* aquí descrit és el cas més simple possible. En aplicacions més completes, el procés de compilació implica generar els fitxers "objecte" i enllaçar-los (*link*) amb les llibreries per tal de generar el fitxer executable.

Java, un cas especial

No tots els llenguatges de programació entren en la divisió entre compilats i interpretats. En el cas del llenguatge Java, el codi font és independent de la plataforma en què es programa. El compilador de Java genera un codi màquina especial (el *bytecode*), i la Màquina Virtual de Java (JVM), existent per a cada plataforma (Linux, Windows, Solaris), sap interpretar aquest *bytecode*, executant-lo. D'aquesta manera, escrivint una sola vegada el codi, s'obtenen aplicacions multiplataforma.

Hi ha un tipus especial de compilació anomenada *transpiling*, que consisteix en la conversió d'un codi en un llenguatge en un altre. Hi ha molts llenguatges que fan servir aquest tipus de compilació per compilar a JavaScript, és més, aquest és el sistema idoni per assegurar la compatibilitat del programari entre versions. Per exemple, abans que les noves funcionalitats d'ES2015 fossin implementades àmpliament als navegadors, era possible escriure la implementació d'una aplicació amb ES2015 i "transpilar-la" a JavaScript 5, assegurant-ne així la compatibilitat.

Compatibilitat amb navegadors

Actualment tots els navegadors s'actualitzen automàticament. Per això es pot confiar en el fet que és segur utilitzar les noves funcionalitats del llenguatge. En cas de dubte es pot consultar la compatibilitat de les noves funcionalitats en el següent enllaç: bit.ly/2OFHQDp

Entre els principals llenguatges que es compilen a JavaScript mitjançant *transpiling* hi ha els següents:

- **JavaScript**: és molt freqüent fer conversions d'una versió més actual a una altra de més antiga per assegurar la compatibilitat.
- **Dart**: desenvolupat per Google, s'utilitza principalment en el desenvolupament d'aplicacions mòbils (iOS i Android natives) mitjançant el marc de treball Flutter.
- **TypeScript**: és un superconjunt de JavaScript desenvolupat per Microsoft. Per consegüent, qualsevol programa desenvolupat amb JavaScript és vàlid a TypeScript. Aquest llenguatge afegeix noves característiques al llenguatge i cal compilar-lo a JavaScript per poder executar les aplicacions al navegador. És el llenguatge recomanat per treballar amb el marc de treball Angular.

Els "transpiladors" (com Babel), a més de fer la conversió del codi, acostumen a afegir els *polyfills*. Un *polyfill* és una implementació d'una funcionalitat del llenguatge que pot ser que encara no es trobi implementada als navegadors. Primer es comprova si la funcionalitat existeix i, en cas contrari, s'afegeix la implementació alternativa.

1.3 El motor de JavaScript. Execució de JavaScript en línia d'ordres

El llenguatge de JavaScript existeix independentment de la web. És necessari recalcar-ho, ja que la manera habitual de treballar és integrant el llenguatge JavaScript dins les pàgines web (HTML).

Quan executem JavaScript dins una pàgina web, per exemple amb el navegador Mozilla Firefox, de l'execució del codi JavaScript se n'encarrega el motor de JavaScript. SpiderMonkey és el motor de JavaScript de Mozilla, que es pot instal·lar de forma independent sense estar associat a un navegador web. Trobareu més informació sobre Spidermonkey i com instal·lar-lo en els següents enllaços:

- developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
- developer.mozilla.org/En/SpiderMonkey/Build_Documentation

Si s'instal·la, de forma opcional, l'SpiderMonkey (tant per a sistemes Linux com Windows), es pot practicar el llenguatge JavaScript des de la línia d'ordres, en un entorn que res a veure té amb la web. Per tant, JavaScript és un llenguatge que existeix independentment de la web.

Així com SpiderMonkey és un motor de JavaScript implementat amb C/C++, Rhino també és un motor de JavaScript, en aquest cas implementat amb Java. També és un projecte desenvolupat per la fundació Mozilla:

- developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino.

Un altre ús molt habitual de JavaScript des de la línia d'ordres és la creació d'eines o aplicacions de servidor mitjançant Node.js. Es tracta d'un entorn d'execució per executar programes escrits en JavaScript, cosa que permet crear aplicacions de xat, servidors web i eines per automatitzar tasques. Es pot descarregar i consultar la documentació a:

- nodejs.org.

1.4 Aplicatius web: arquitectura client-servidor

Avui en dia tenim una gran quantitat de dispositius connectats al núvol. El portàtil, el telèfon mòbil, la tauleta, la televisió... i amb l'*Internet of Things* (IoT) cada vegada aquests dispositius aniran en augment. Una de les coses habituals que fem amb algun d'aquests dispositius és navegar per la web tot visitant pàgines web.

D'altres dispositius i aplicacions, tot i que no serveixen per navegar per la web, sí que fan consultes a serveis web. Imaginem per exemple un aplicatiu web per

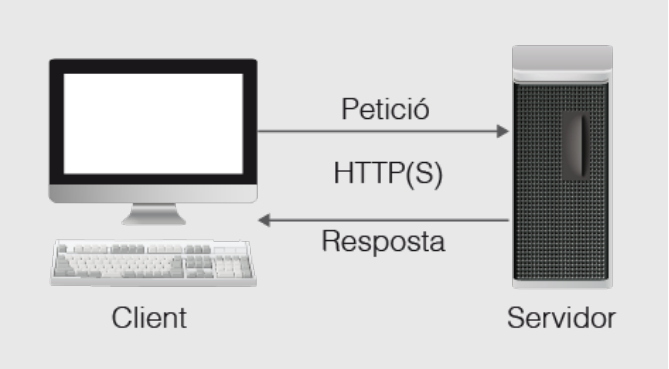
controlar una casa domòtica, on podem posar la calefacció a distància. O una aplicació per al mòbil de calendari, que se sincronitza contínuament i ens avisa de quan tenim una reunió. Per tant, hem de prendre consciència del fet que el que ens permet la tecnologia web, avui dia, és molt més que visitar pàgines web.

Si una cosa tenen en comú tots aquests aplicatius web és que es basen en l'arquitectura client-servidor. Quan parlem de client web ens ve el cap el navegador web (Firefox, Chrome, Internet Explorer). Ara bé, una aplicació que ens permet consultar l'horari de la parada del bus també és un client web, tot i que la presentació no té format de pàgina web.

Quan ens connectem a una pàgina web per Internet, el més habitual dels casos és que ens connectem a un servidor web Apache.

A l'altra banda tenim els servidors web, que serveixen el contingut web que tenen allotjat. Exemples de servidor web són l'Apache Web Server o l'Internet Information Server, però també el servidor d'aplicacions Tomcat (vegeu la figura 1.1).

FIGURA 1.1. Arquitectura client-servidor en la comunicació HTTP



Suposem que estem amb el nostre portàtil, i consultem una web de viatges des del navegador Chrome. Fixem-nos amb el camí que fan els paquets IP. Des del Chrome, en la barra de navegació, posem una URL que identifica unívocament el contingut que volem baixar (per exemple, www.vacances.com/pallars_jussa.php). Aquesta informació es trosseja en forma de paquets IP que viatgen per Internet. Els servidors DNS localitzen la direcció IP del servidor on està allotjat el contingut. A més, aquests paquets han de passar per *routers*, *firewalls* i diferents capes de seguretat.

Els paquets IP finalment s'ajunten en el destí, on tenim un servidor web Apache que allotja l'script *pallars_jussa.php*. Aquest script s'executa (es processa la part de PHP i s'ajunta amb la part d'HTML), i finalment el resultat es serveix al destinatari fent el camí invers.

Quan la pàgina web arriba al destí, en el navegador web podem visualitzar la informació demanada. El navegador web, el nostre client, és el responsable de maquetar correctament la pàgina web a partir del contingut HTML, els fulls d'estil CSS, i també el codi JavaScript que conté. Podem veure el codi font que hem rebut del servidor fent *Ctrl/U* en el navegador (fixa't que mai veuràs en el codi font el codi PHP; per contra, sí que pots veure el codi JavaScript).

L'entorn de client que estem estudiant és precisament la part del navegador web (Firefox, Chrome), en contraposició amb la part de servidor (el servidor Apache).

Així com el codi PHP s'executa en el servidor, el codi JavaScript que contenen les webs s'executa en el client (Firefox, Chrome).

Exemple de navegador web en mode text

Què passa si a una pàgina web li traiem tota la part gràfica i de disseny? No us hauria d'estranyar aquesta possibilitat si us poseu en la pell de les persones cegues. Els desenvolupadors web han de pensar en el concepte d'accessibilitat: intentar que el contingut i la funcionalitat bàsiques d'una web estigui disponible en un navegador web en mode text.

Instal·lar un navegador web que funcioni en la consola, sense interfície gràfica, és simple. Hi ha diferents possibilitats, entre elles, Lynx. Us podeu instal·lar Lynx tant en Linux com en Windows. Per a sistemes Linux basats en paquets Debian/Ubuntu és tan fàcil com fer:

```
1 $ sudo apt-get install lynx.
```

Us serà molt útil consultar aquests enllaços:

- ja.cat/JZslp
- ja.cat/RXRMd

Un cop el teniu instal·lat, des de la consola podeu visitar una pàgina web:

```
1 lynx www.google.com
```

I també podeu seguir els enllaços presents. Ara bé, té més sentit si visiteu una pàgina que tingui un fort component textual:

```
1 lynx https://ca.wikipedia.org/wiki/Tirant_lo_Blanc
```

Els navegadors textuals no executen codi JavaScript, a causa de motius tècnics, i al fet que la interacció amb l'usuari és molt limitada.

1.5 Beneficis d'utilitzar JavaScript en les pàgines web

En l'inici de l'era d'Internet les pàgines HTML eren estàtiques: només hi havia text fix i imatges. Avui dia, tot al contrari, les pàgines web són dinàmiques, amb efectes visuals i interacció amb l'usuari. Això ha estat possibles gràcies a la incorporació de JavaScript: el codi que s'executa dins el navegador web.

Vegeu-ne algun dels avantatges:

- JavaScript és l'únic llenguatge admès pels navegadors; per consegüent, no és possible fer servir altres llenguatges sense utilitzar un "transpilador" a JavaScript.
- JavaScript permet una gran quantitat d'efectes dins les pàgines. Entre d'altres: *popups*, text que es col·lapsa, capes que es fonen, *scrolls*, transicions d'imatges...
- JavaScript afegeix interactivitat amb l'usuari.

- JavaScript permet validació dels formularis en el cantó del client. Una validació inicial dels formularis és possible per eliminar simples errors, com assegurar-se de què el format de data, DNI, correu electrònic o telèfon són correctes... Com a resultat, l'usuari té una resposta més ràpida que no pas si el control dels errors el fes el servidor.
- JavaScript permet accedir a informació del navegador, cosa que inclou el sistema operatiu i el llenguatge.
- JavaScript permet el desenvolupament d'extensions per a navegadors com Chrome i Firefox.
- JavaScript permet el desenvolupament d'aplicacions complexes com els editors de text o fulls de càlcul de Google Docs i videojocs (per exemple: bit.ly/3hoeO7T).

Podeu trobar un exemple de videojoc a la unitat "Desenvolupament de casos pràctics".

1.6 Desavantatges de JavaScript

La seguretat és el principal problema a JavaScript. Els fragments de codi JavaScript que s'afegeixen a les pàgines web es descarreguen en els navegadors i s'executen en el cantó del client, permetent així la possibilitat que un codi maliciós es pugui executar en la màquina client i així explotar alguna vulnerabilitat de seguretat coneguda en les aplicacions, navegadors o fins i tot del sistema operatiu. Existeixen estàndards de seguretat que restringeixen l'execució de codi per part dels navegadors. Es tracta sobretot de deshabilitar l'accés a l'escriptura o lectura de fitxers (exceptuant les galetes). Tanmateix, la seguretat a JavaScript continua sent un tema polèmic i de discussió.

Un altre desavantatge de JavaScript és que tendeix a introduir una quantitat enorme de fragments de codi en els nostres llocs web. Això es resol fàcilment emmagatzemant el codi JavaScript en fitxers externs amb extensió JS. Així la pàgina web queda molt més neta i llegible. La tendència actual és de fet separar totalment la part del contingut HTML, la part de funcionalitat JavaScript, i la part de disseny i maquetació. De manera que tres perfils d'usuaris diferents (el que genera continguts HTML, el programador web, i el dissenyador) puguin estar treballant en el mateix projecte, i tocant arxius diferents.

Un avantatge de deixar ben net el contingut HTML és que estem facilitant la feina als motors de cerca (com Google), de manera que poden desxifrar fàcilment el contingut de la pàgina web, i aquest contingut es pot indexar correctament en els resultats de les cerques.

1.7 Integració del codi JavaScript en el codi HTML. Primer exemple

JavaScript és un llenguatge d'script que existeix i té sentit fora de la web, però que ha pres protagonisme i reconeixement gràcies a la web. Així doncs, és usual associar JavaScript amb una de les tecnologies clau en el desenvolupament d'aplicacions web. Per a nosaltres JavaScript serà un puntal per tal que les nostres aplicacions web es comportin de forma dinàmica i interactiva. El primer que caldrà veure és com conviuen la sintaxi HTML i la sintaxi JavaScript, com es pot integrar JavaScript dins les pàgines web.

La inclusió de la sintaxi de JavaScript es fa mitjançant l'etiqueta <*script*>.

Etiqueta <script>

És usual veure escrita la sintaxi <*script type="text/javascript"*>. Aquest atribut ja no és necessari ja que en HTML5 el llenguatge JavaScript és el llenguatge d'script per defecte. Tanmateix, això reforça la idea que podrien haver-hi d'altres llenguatges d'script que el navegador sabés interpretar.

```
<script>
  let nom = "Rita";
  alert("Hola " + nom);
</script>
```

Podem escriure moltes etiquetes <*script*> en un document, tot i que posar-ne una de sola és un bon hàbit. Aquestes etiquetes les podem posar tant en el <*head*> com en el <*body*>.

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>El meu primer exemple</title>
        <script>
            function funcio() {
                document.getElementById("paragraf").innerHTML = "Escrivim en el
                     paràgraf";
            }
        </script>
    </head>
    <body>
        <h1>El meu primer exemple</h1>
        <p id="paragraf">Text original del paràgraf</p>
        <button type="button" onclick="funcio()">Clica'm</button>
    </body>
</html>
```

Per provar aquest codi, obriu el vostre editor de text preferit. Però ens referim a un editor de text pla, no utilitzeu Openoffice, Microsoft Office o similars. A Linux podeu utilitzar el Gedit, o fins i tot editors de consola: Vim, Nano, Joe. A Windows podeu utilitzar el Notepad. Tot i que els desenvolupadors web utilitzen editors més complets, en aquesta ocasió utilitzareu un editor senzill. Un cop obert l'editor, copieu el codi i reanomeneu-lo a un fitxer HTML, per exemple, boto.html. Tingueu clar en quina ubicació l'heu gravat. Aquest fitxer l'heu d'obrir des del vostre navegador web preferit (preferentment Mozilla Firefox o Google Chrome). Ho podeu fer clicant sobre el fitxer amb el botó dret, o a la barra de navegació del navegador ficar la ruta. Per exemple:

```
file://ruta_al_fitxer/boto.html
```

Us ha d'aparèixer una web senzilla. Quan cliqueu sobre el botó, el text del paràgraf ha de canviar. Ja heu fet la vostra primera aplicació JavaScript. Ja heu

afegit interacció a una pàgina web. En aquest codi es donen per suposats molts conceptes. Primer de tot s'ha afegit un *event* a un botó. Un *event* és la capacitat de respondre a una acció de l'usuari, en aquest cas fer clic sobre un botó. Quan es fa clic, s'executa la funció `funcio()` definida entre les etiquetes <*script*>. Aquesta funció el que fa és localitzar l'element definit per l'identificador *id="paragraf"*, que és un paràgraf (<*p*>), i canvia el seu contingut. Aquest codi també funciona si col·loqueu l'etiqueta <*script*> en una altra banda:

És interessant col·locar l'etiqueta <*script*> a sota de tot del <*body*>. D'aquesta manera no bloquem la càrrega dels elements HTML. Tanmateix, s'ha de donar preferència a col·locar les etiquetes <*script*> a la capçalera <*head*> del document web.

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>El meu primer exemple</title>
    </head>
    <body>
        <h1>El meu primer exemple</h1>
        <p id="paragraf">Text original del paràgraf</p>
        <button type="button" onclick="funcio()">Clica'm</button>
        <script>
            function funcio() {
                document.getElementById("paragraf").innerHTML = "Escrivim en el
                    paràgraf";
            }
        </script>
    </body>
</html>
```

1.7.1 Introducció a la manipulació del DOM

Podeu trobar informació molt més completa a la unitat "Model d'objectes del document".

El DOM (*Document Object Model*) és una interfície per a documents HTML i XML que representa un document com nodes i objectes, de manera que els llenguatges de programació poden connectar amb la pàgina i modificar-la.

En el context dels navegadors web, les pàgines web són documents i el navegador és el responsable d'interpretar el codi HTML per generar el DOM, que pot ser modificat mitjançant JavaScript.

El DOM és accessible des de JavaScript mitjançant l'objecte global `document`, que exposa múltiples mètodes per consultar-lo i modificar-lo.

Quan s'assigna un identificador a un element HTML (atribut `id`) cal que aquest sigui únic.

Per seleccionar un node del document, el més simple és utilitzar el mètode `getElementById()`, que permet seleccionar un element pel seu identificador (l'atribut `id`). Per exemple:

```
<html>
  <body>
    <h1 id="titol">Aquést és el títol</h1>
    <p id="text">Això és un text</p>

  <script>
    let nodeTitol = document.getElementById('titol');
    let nodeText = document.getElementById('text');
  </script>
  </body>
</html>
```

Vegeu que s'han assignat a les variables `nodeTitol` i `nodeText` els nodes de tipus element corresponents als identificadors `titol` i `text`, respectivament.

Cal destacar que són nodes de tipus `element`. Així doncs, a partir d'aquestes variables es pot accedir a les propietats i els mètodes exposats per la interfície `Element`. Per exemple, la propietat `innerHTML` permet modificar el contingut intern d'un element:

Element

Podeu trobar informació detallada de la interfície `Element` a mzl.la/3a9vQ8N.

```
nodeTitol.innerHTML = "Títol reemplaçat";
nodeText.innerHTML = "Nou contingut de <b>text</b>";
```

Podeu veure aquest exemple a l'enllaç següent: codepen.io/ioc-daw-m06/pen/vYyyQQp?editors=1000.

Fixeu-vos que es reemplaça el contingut intern de l'*element*, així que qualsevol contingut anterior es perd i, al mateix temps, si s'ha afegit codi HTML, aquest és interpretat i afegit com a nous elements al DOM (per exemple, `<b>text</b>` s'afegeix un nou element dintre del paràgraf).

Atès que `innterHTML` és una propietat, és possible modificar-la, consultar el seu valor i assignar-la a altres variables.

Per afegir un element nou al document primer cal crear-lo mitjançant el mètode `createElement()` i indicar el tipus d'element. Per exemple:

```
document.createElement('li');
```

Però això no és suficient, ja que aquest element no s'ha afegit al document. Per afegir-lo hem de fer servir el mètode `append()` que es troba definit a la interfície `ParentNode` i que és implementada per `Document` i `Element`; això vol dir que es poden afegir nous elements tant al document com en d'altres elements.

Així, doncs, és necessari assignar l'element creat amb `createElement` a una variable i obtenir una referència a l'element pare on es vol afegir el node. Per exemple, podem afegir nous elements a una llista de la següent manera:

```
<html>
<body>
  <ul id="llista"></ul>

  <script>
    let llista = document.getElementById('llista');

    let nouElementLlista = document.createElement('li');
    llista.append(nouElementLlista);
    nouElementLlista.innerHTML = 'Primer element';

  </script>
</body>
</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/WNooPKX?editors=1000.

De la mateixa manera que podem afegir elements, també els podem modificar. Per fer-ho, només ens cal una referència al node i invocar al seu mètode `remove()`:

```
<html>

<body>
  <h1 id="primer">Primer títol</h1>
  <h1>Segon títol</h1>

  <script>
    let títol = document.getElementById('primer');
    títol.remove();
  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/OJbbqNz?editors=1000.

Per accedir al valor dels elements de formulari, com són els quadres de text i les llistes desplegables, s'ha de fer servir la propietat `value`:

```
<html>

<body>
  <h1 id="titol">Aquest és el títol</h1>
  <h2 id="subtitol">Aquest és el subtítol</h2>
  <label>Nou text:</label>
  <input id="entradaTitol" />
  <label>Selecciona:</label>
  <select id="selectTitol">
    <option value="Aquest és el primer subtítol">Primer</option>
    <option value="Aquest és el segon subtítol">Segon</option>
    <option value="Aquest és l'últim subtítol">Últim</option>
  </select>

  <button onclick="canviar();">Canviar Títols</button>

  <script>
    let nodeTitol = document.getElementById('titol');
    let nodeSubtitol = document.getElementById('subtitol');
    let nodeEntrada = document.getElementById('entradaTitol');
    let nodeSelect = document.getElementById('selectTitol');
    nodeEntrada.value = "nou títol";

    function canviar() {
      nodeTitol.innerHTML = nodeEntrada.value;
      nodeSubtitol.innerHTML = nodeSelect.value;
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/MWbbLYE?editors=1000.

Fixeu-vos que hem assignat el valor de `nodeEntrada` a "`nou títol`" fàcilment, però en el cas de la llista seleccionable s'ha de tenir en compte que es treballa amb dos valors diferents, corresponents als parells assignats mitjançant l'element d'HTML `option`:

- **value**: valor que s'assignarà a la llista desplegable.
- **text**: text que es mostra a la llista desplegable.

Per poder accedir a les opcions de l'element `select` es pot utilitzar la propietat `options` i `selectedIndex`:

- **options**: és un *array* d'elements d'opció a partir dels quals es pot obtenir el seu `value` i `innerHTML`.
- **selectedIndex**: és l'índex corresponent a l'opció seleccionada.

Donat que el contingut de la llista és determinat pels elements de tipus `option`, si volem afegir més opcions a la llista és necessari crear nous elements i afegir-los, com es pot apreciar en el següent exemple:

```
<html>

<body>
  <select id="seleccio">
    <option value="Barcelona">Barcelona</option>
  </select>

  <script>
    let seleccio = document.getElementById('seleccio');

    let novaOpcio1 = document.createElement('option');
    seleccio.append(novaOpcio1);
    novaOpcio1.innerHTML = "Girona";

    let novaOpcio2 = document.createElement('option');
    seleccio.append(novaOpcio2);
    novaOpcio2.innerHTML = "Lleida";

    let novaOpcio3 = document.createElement('option');
    seleccio.append(novaOpcio3);
    novaOpcio3.innerHTML = "Tarragona";

  </script>
</body>

</html>
```

Com que les opcions són elements, si volem eliminar un element de la llista cal eliminar el node corresponent cridant al seu mètode `remove()`. Per exemple:

```
novaOpcio1.remove();
```

Atès que a partir d'un element de tipus `select` podem accedir a la llista d'opcions contingudes, s'hi pot accedir sense necessitat d'un identificador únic. Per exemple:

```
let seleccio = document.getElementById('seleccio');
seleccio.options[0].value; // Mostra el valor de la primera opció de la llista
seleccio.options[0].innerHTML; // Mostra el text de la primera opció de la
    llista
seleccio.options[seleccio.selectedIndex].remove(); // Elimina ´lopció
    seleccionada de la llista
```

Com es pot apreciar, es pot accedir a qualsevol opció mitjançant l'índex (posició a la llista començant per 0), ja que es tracta d'un *array*.

Cal tenir en compte que el tractament de les caselles de selecció i els botons d'opció és una mica diferent dels anteriors, ja que l'estat (seleccionat o no) no

està assignat al valor, sinó que es tracta de la propietat `checked`. Això és així perquè quan es treballa amb formularis s'envien els elements marcats, associant al nom del camp amb el valor assignat al botó.

```
<html>

<body>
  <input type="radio" name="porta" id="boto_obert" value="oberta" checked/>
  <input type="radio" name="porta" id="boto_tancat" value="tancada" />

  <script>
      let botoTancat = document.getElementById('boto_tancat');
      botoTancat.checked = true;
  </script>
</body>

</html>
</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/oNYYVeY?editors=1000.

Els botons d'opció s'agrupen pel valor de l'atribut `name`.

Fixeu-vos que al codi HTML s'ha indicat que l'opció marcada és oberta (l'atribut `checked` es troba present), però amb `botoTancat.checked = true` canviem l'opció a `botoTancat` i, com que pertanyen al mateix grup, es desmarca l'anterior.

Les caselles de selecció s'utilitzen de la mateixa manera, amb la diferència que no estan agrupades i, per consegüent, poden marcar-se múltiples caselles alhora:

```
<html>

<body>
  <input type="checkbox" id="opcio1" value="groc" checked/><label>Groc</label>
  <input type="checkbox" id="opcio2" value="vermell" /><label>Vermell</label>
  <input type="checkbox" id="opcio3" value="blau" /><label>Blau</label>

  <script>
      let opcio3 = document.getElementById('opcio3');
      opcio3.checked = true;
  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/GRNmjXB?editors=1000.

Heu de tenir en compte que quan s'envia un formulari, en el cas de les caselles de selecció i els botons de ràdio només s'envien al servidor si estan marcats. Però si treballeu amb el client mitjançant JavaScript, haureu de controlar quines caselles estan marcades i quines no. Per exemple:

```
<html>

<body>
  <h1>Opcions adicionals</h1>
  <input type="checkbox" id="opcio1" value="325" /><label>64MB Memoria Ram</
      label>
  <input type="checkbox" id="opcio2" value="265" /><label>2TB SSD</label>
  <input type="checkbox" id="opcio3" value="1445" /><label>GeForce GTX 3090</
      label>
```

```
  <button onclick="calcular();">Calcular</button>

  <h2>Cost extras: <span id="extra">0</span€></h2>

  <script>
    let opcio1 = document.getElementById('opcio1');
    let opcio2 = document.getElementById('opcio2');
    let opcio3 = document.getElementById('opcio3');
    let extra = document.getElementById('extra');

    function calcular() {
      let total = 0;
      if (opcio1.checked) {
        total += Number(opcio1.value);
      }
      if (opcio2.checked) {
        total += Number(opcio2.value);
      }
      if (opcio3.checked) {
        total += Number(opcio3.value);
      }
      extra.innerHTML = total;
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/gOLLEKz?editors=1000.

1.7.2 Fitxers JavaScript externs

Un fitxer JavaScript, amb extensió JS, és un fitxer de text que conté instruccions JavaScript. Des de l'HTML podem incloure fàcilment un fitxer JavaScript:

```
<script src="fitxerExtern.js"></script>
```

Així doncs, el contingut de fitxerExtern.js serà:

```
function funcio() {
    document.getElementById("paragraf").innerHTML = "Escrivim en el paràgraf";
}
```

i el nostre codi quedarà de la següent manera:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>El meu primer exemple</title>
        <script src="fitxerExtern.js"></script>
    </head>
    <body>
        <h1>El meu primer exemple</h1>
        <p id="paragraf">Text original del paràgraf</p>
        <button type="button" onclick="funcio()">Clica'm</button>
    </body>
</html>
```

S'entén que el codi HTML ha de localitzar el fitxer JS. En aquest cas els dos fitxers estan en el mateix directori. És habitual que els fitxers JavaScript estiguin en una carpeta *js/* que penja del directori principal de l'aplicació. En aquest cas quedaria:

```
<script src="js/fitxerExtern.js"></script>
```

Quan es fan aplicacions web s'ha de tendir a la separació del contingut, el disseny i la funcionalitat. Això s'aconsegueix fent que el codi HTML sigui el més net possible. Tota la part de JavaScript ha d'anar a fitxers externs, de la mateixa manera que tot el disseny i estil van a fitxers externs CSS. Fixeu-vos en la sentència:

```
<button type="button" onclick="funcio()">Clica'm</button>
```

La definició d'un botó HTML és la que defineix el comportament d'un *event* amb una funció JavaScript. Aquí també estem barrejant HTML amb JavaScript, i també cal evitar-ho.

> És molt recomanable ser ordenats a l'hora de posar un nom als fitxers, versionant els fitxers. Fixeu-vos que en aquest primer exemple hem provat el codi de tres maneres diferents, i podeu identificar amb un nom diferent cadascuna de les proves.

1.7.3 Separació del codi en mòduls

Donat que les aplicacions en JavaScript van anar augmentant en complexitat, van aparèixer biblioteques especialitzades per carregar mòduls com AMD i CommonJS. Aquests mòduls contenien guions que podien carregar-se fent servir les paraules clau `import` i `export` per accedir a diferents funcionalitats. D'aquesta manera, es podien carregar només els mòduls necessaris i reutilitzar-los en diferents aplicacions.

Aquesta és una de les funcionalitats que s'han afegit a JavaScript en les darreres versions, de manera que és possible dividir les aplicacions en mòduls utilitzant `import` i `export`.

> **Càrrega de mòduls de forma local**
>
> No és possible carregar mòduls quan s'obre un fitxer HTML des del sistema de fitxers, és imprescindible treballar amb un servidor web local o un editor que tingui la capacitat de crear servidors web. Per exemple, Aptana Studio 3 permet llençar l'aplicació fent servir un servidor intern seleccionant "Firefox / Internal Server" quan es llença l'aplicació.

Per poder treballar amb mòduls cal indicar a la etiqueta `<script>` el tipus `module` de la següent manera:

```
<script type="module">
  // Codi de la aplicació
</script>
```

El següent exemple mostra el codi HTML per carregar el mòdul *salutacions.js*, que exporta les funcions `hola` i `adeu`:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>El meu primer exemple amb mòduls</title>

</head>
<body>
<h1>El meu primer exemple amb mòduls</h1>
</body>
<script type="module">
    import {hola} from './salutacions.js';
    import {adeu} from './salutacions.js';

    hola();
    adeu();
</script>
</html>
```

I aquest és el codi corresponent al mòdul `salutacions.js`:

```
export function hola() {
    mostrarAlerta("Hola món!");
}

export function adeu() {
    mostrarAlerta("Adeu món!");
}

function mostrarAlerta(text) {
    alert(text);
}
```

S'ha de tenir en compte que l'àmbit de cada mòdul és el mateix mòdul, al contrari del que passa quan es carrega un fitxer JavaScript (en aquest cas, l'àmbit és global). És a dir, des d'un mòdul només es pot accedir al codi declarat al mateix mòdul, a l'espai global i a les funcions, classes i variables importades d'altres mòduls.

Fixeu-vos que la funció `mostrarAlerta()` no és exportada des del mòdul i, per consegüent, es produirà un error si s'intenta cridar des del fitxer HTML.

Per aquest mateix motiu tampoc és possible accedir a les funcions importades des de les crides en línia com `<button type=''button'' onclick=''hola()''>Hola</button>`. Aquest tipus de crides es fan des de l'espai global i, per tant, cal afegir els mòduls exportats a l'espai global si volem utilitzar-les d'aquesta manera. Una possible implementació seria la següent:

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>El meu segon exemple amb mòduls</title>

</head>
<body>
<h1>El meu segon exemple amb mòduls</h1>
<button type="button" onclick="hola()">Hola</button>
<button type="button" onclick="adeu()">Adeu</button>
```

```
12 </body>
13
14 <script type="module">
15     import {hola} from './salutacions.js';
16     import {adeu} from './salutacions.js';
17
18     window.hola = hola;
19     window.adeu = adeu;
20 </script>
21 </html>
```

Com es pot apreciar, no ha estat necessari modificar el mòdul *salutacions.js* sinó que s'ha reaprofitat.

Una altra característica important dels mòduls és que el codi només és avaluat la primera vegada que s'importa, encara que sigui importat en múltiples llocs.

En resum, les característiques dels mòduls són les següents:

- Faciliten l'estructuració i reutilització del codi.
- Només es poden importar mòduls des d'altres mòduls.
- L'àmbit d'un mòdul és el mateix mòdul.
- No es poden carregar mòduls fent servir el sistema de fitxers, cal utilitzar un servidor web o un editor amb aquesta capacitat.
- El codi d'un mòdul només s'avalua la primera vegada que s'importa.
- Cal indicar amb `export` les funcions, classes i variables a exportar d'un mòdul; només aquestes estaran disponibles per importar.
- Per importar un element d'un mòdul es fa servir la paraula clau `import`.

1.8 JavaScript: programació dirigida per esdeveniments

La programació dirigida per esdeveniments (*event-driven programming*) és un paradigma de programació en què el flux del programa està determinat per esdeveniments (*events*) com ara accions de l'usuari (típicament, moviments del ratolí i ús del teclat). Aquest és precisament el model que es troba en la programació web i en el seu resultat: la navegació per pàgines web.

En una pàgina web tenim un contingut estàtic, informatiu, però aquest contingut canvia a mesura que interactuem amb la pàgina web. Quan naveguem per la web, contínuament estem llençant *events* al motor de JavaScript, que s'encarrega de processar-los. Cliquem sobre un enllaç, fem *scroll* per la pàgina, passem el ratolí per damunt d'un menú desplegable... Tota la interacció que fem amb la pàgina web és recollida pel processador d'*events* que executa a temps real trossos de codi JavaScript que modifiquen el contingut de la pàgina. Per tant, les pàgines web esdevenen interactives, i el seu contingut és interactiu. Són els conceptes

d'hipermèdia i hipertext, en què el consum d'informació no és lineal, sinó que és l'usuari qui decideix com interactua amb l'aplicació.

En el següent exemple estem associant al botó l'*event* `onclick`, de manera que quan cliquem sobre el botó s'executa una sentència JavaScript:

Llistat complet d"events'

Podeu trobar una llista completa dels *events* disponibles al següent enllaç: mzl.la/3eLjWRB.

```
<button onclick="this.innerHTML=Date()">L'hora és ...</button>
```

`this` fa referència al propi botó, i té la propietat `innerHTML` que representa l'etiqueta que mostra el botó. El contingut d'aquesta etiqueta canvia pel valor que retorna la funció `Date()`, que és l'hora del sistema.

Podeu provar el darrer exemple a: codepen.io/ioc-daw-m06/pen/YWZZPE

En aquest exemple fem servir l'*event* `onmouseover` associat a la capa (`div`) que conté dos paràgrafs:

```
<div onmouseover="document.getElementById('p1').innerHTML='Curs de JavaScript';
    document.getElementById('p2').innerHTML='IOC (Institut Obert de Catalunya)
    '">
<p id="p1">Primer paràgraf</p>
<p id="p2">Segon paràgraf</p>
</div>
```

Quan passem el ratolí per sobre la capa, canvia el contingut dels dos paràgrafs.

Podeu provar el darrer exemple a: codepen.io/ioc-daw-m06/pen/xOqqwR

Si us hi fixeu bé, en els dos exemples mostrats estem barrejant el llenguatge HTML amb sentències JavaScript. Tot i que és usual, la tendència serà separar totalment la part d'HTML de la part de JavaScript.

1.9 Eines i entorns per al desenvolupament web

Per programar amb JavaScript es necessita un editor de text. Molts llenguatges de programació es troben suportats amb **IDE** (Integrated Developement Environment) per tal d'escriure codi més ràpid, accedir a la documentació i evitar errors. En el cas de JavaScript creiem que l'aproximació és diferent. No importa tant l'editor de text que es fa servir, però sí que és necessari un entorn on el programador pugui provar de forma ràpida i còmoda el codi, i pugui depurar els errors. Sortosament, els navegadors web (Mozilla Firefox, Google Chrome) ja porten incorporats de forma predeterminada entorns de desenvolupament que ajuden a la programació i depuradors.

Així doncs, per desenvolupar aplicacions amb JavaScript fareu servir un editor de text a escollir, i les eines de desenvolupament integrades en el navegador. En els dos casos haureu d'esmerçar temps a trobar les dreceres de teclat, opcions i possibilitats per treure'n el màxim profit i rendiment, amb la idea de desenvolupar ràpid, i també corregir i depurar els errors amb rapidesa. Així mateix, és recomanable en la mesura del possible treballar amb dues pantalles: una per veure

el codi (codi HTML, JavaScript, CSS, eines de depuració...) i una altra per veure el resultat.

Quant als editors, de text tenim diferents opcions. D'un editor de text adaptat al desenvolupament esperem que tingui les següents característiques:

- Elecció de llenguatge de programació. En el vostre cas, JavaScript.
- Coloració sintàctica.
- Autocompleció.
- Eines avançades de cerca i reemplaçament.
- Marcadors (*bookmarks*). Navegació pels marcadors.
- Autotabulació.
- Agrupament (i desagrupament) de codi.
- Noves funcionalitats amb instal·lació d'extensions (*plugins*).

Entre els editors que podeu fer servir destaquem:

- Sublime (multiplataforma)
- Notepadqq (Linux)
- Notepad ++ (Windows)
- Aptana Studio (multiplataforma)
- Brackets (multiplataforma)

Us podeu registrar en la plataforma codepen.io, encara que sense fer-ho també podeu editar directament els exemples, però sense desar-los.

Una altra possibilitat és utilitzar un IDE, com ara NetBeans o Eclipse. En el cas de desenvolupar per JavaScript, això només està justificat si coneixeu aquestes eines amb profunditat i fluïdesa perquè les utilitza amb d'altres llenguatges de programació.

S'utilitza codepen.io al llarg de tot el mòdul per tal que pugueu comprovar directament el funcionament de molts dels exemples.

De la mateixa manera necessitem entorns per fer proves del nostre codi. El codi JavaScript s'emmarca dins una pàgina web, tanmateix, en comptes de fer una pàgina web cada vegada que vulguem provar codi, hi ha entorns en línia que ens ajudaran a provar el codi de forma ràpida, com ara codepen.io.

1.10 Sintaxi del llenguatge JavaScript

A partir de l'edició de l'estàndard ECMA-262 del mes de juny del 2015, les actualitzacions a l'estàndard es publiquen anualment. Sobre aquest document trobareu una activitat a la part web del mòdul.

JavaScript està basat en el estàndard ECMA-262, que defineix el llenguatge de propòsit general ECMAScript. Ecma International és una associació de la indústria fundada l'any 1961, i dedicada a la estandardització dels sistemes de la informació. Podeu consultar la pàgina web oficial a www.ecma-international.org, i des de ja.cat/jD3X5 descarregar-vos tot el document en format PDF o HTML.

1.10.1 Sentències, comentaris, variables

JavaScript és un llenguatge de programació, és a dir, amb el seu codi d'instruccions podem fer programes informàtics. Aquests programes estaran orientats a fer pàgines web funcionals, atractives i interactives. Un programa es composa d'una llista d'instruccions que seran executades, en aquest cas interpretades, per l'ordinador. Aquestes instruccions són les sentències.

> **La importància d'utilitzar el punt i coma**
>
> A JavaScript, el punt i coma (;) indica el final d'una sentència. A JavaScript, posar-lo és opcional, però és molt convenient afegir-lo al final de les sentències perquè, en cas contrari, és el navegador qui els "posa" on creu convenient i pot no coincidir amb els nostres desitjos o suposicions.

Una cosa important, i que l'usuari aprèn de seguida, és que JavaScript distingeix entre majúscules i minúscules.

Així doncs, ja podeu fer el vostre primer programa i integrar-lo dins el codi HTML d'una pàgina web:

```
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>El meu primer programa amb variables</title>
</head>
<body>
  <h1 id="capcalera">El meu primer programa amb variables</h1>
  <script>
  //declarem una variable
  let x = 3;
  /* Els comentaris també
     poden ser multilínia.
     Declarem una altra variable */
  let y = 5;
  let z = x * y;
  </script>
</body>
</html>
<html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/bGEOmKN.

En aquest programa hem declarat les variables *x, y, z*. Declarem les variables amb la paraula reservada `let`. A les dues primeres els hem donat un valor, i la tercera l'hem declarada com a producte de les dues primeres.

Actualment a JavaScript es poden declarar les variables fent servir tres paraules clau diferents:

- `let`: és la forma més habitual de declarar les variables en aplicacions de JavaScript modern. L'àmbit de la funció és de bloc, és a dir, si es declara entre claus `{}` el seu valor és accessible només dintre d'aquest bloc.
- `const`: l'àmbit és el mateix que l'anterior, però prohibeix sobreescriure el valor i, per consegüent no s'utilitza amb variables sinó amb constants.

- `var`: l'àmbit de la variable és la funció on es declara o l'espai global si no es declara dins de la funció, sense importar si es troba dins d'un bloc o no. Aquesta era l'única forma de declarar les variables en ES5, i actualment no es recomana el seu ús.

Encara que no es produeix cap error si no es declaren les variables, és important fer-ho, perquè en cas contrari el navegador considera que es tracta d'una variable global, sense importància del context on es troba.

Normes sobre les variables

Les variables han de ser identificadors únics, i han de complir les normes següents:

- Només poden contenir lletres, dígits, signe de subratllat (_) i signe de dòlar ($).
- Han de començar amb una lletra, o bé amb el signe de subratllat (_) o el signe de dòlar (%%$%).
- Es distingeix entre majúscules i minúscules (la variable `x` és diferent de la variable `X`).
- No es poden utilitzar les paraules reservades de JavaScript (`while`, `for`, `next`...).

A més, hem comentat el codi, la qual cosa sempre està bé, mai és sobrer. Aquest programa ja fa alguna cosa, però l'usuari espera veure el resultat del producte. De fet, en JavaScript sempre tenim diferent alternatives. Per veure el resultat, podem fer-ho de quatre maneres diferents:

- Escrivint una caixa de text amb *window.alert()*.
- Utilitzant un element HTML com ara un paràgraf.
- Escrivint directament en el document amb *document.write()*.
- Escrivint a la consola del navegador amb *console.log()*.

I sense proposar-nos-ho, aquí introduïm el concepte d'objecte: `window`, `document` i `console` són objectes que tenen els mètodes `alert()`, `write()` i `log()`. I és que JavaScript també és un llenguatge orientat a objectes, i contínuament haurem de parlar de mètodes i propietats dels objectes.

Aleshores el nostre codi pot quedar de la següent manera:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>El meu primer programa</title>
    </head>
    <body>
        <h1 id="capcalera">El meu primer programa</h1>
        <script>
            //declarem una variable
            let x = 3;
            /* Els comentaris també
            poden ser multilínia.
            Declarem una altra variable */
            let y = 5;
```

```
            let z = x*y;
            document.getElementById("capcalera").innerHTML = z;
            console.log(z);
            window.alert(z);
            document.write(z);
        </script>
    </body>
</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/mdVaQyb?editors=1000.

Com que estem en les primeres proves, us recomanem crear un fitxer HTML (per exemple, *primerPrograma.html*) amb l'anterior codi, i executar-lo. Per tal de veure la consola del navegador, en el navegador s'ha d'anar a les eines de desenvolupament i veure quina és la combinació de tecles adient. Depèn de si es fa servir Google Chrome o Mozilla Firefox, i fins i tot depèn de la versió d'aquests programes.

Fixem-nos que l'`script` és a la part de sota del `body`. Si el pugem a dalt de tot del `body` o al `head`, a la consola obtindrem un missatge d'error:

```
TypeError: document.getElementById(...) is null
```

El programa no sap què és *id="capcalera"* fins que no s'ha carregat la línia HTML *<h1 id="capcalera">*. Sempre s'ha de tenir ben present que el navegador web carrega la pàgina HTML línia a línia des del principi, i que no podem fer referència a un objecte o element HTML que encara no hem carregat en memòria. Una manera de resoldre aquest conflicte és utilitzar el següent codi, que fa ús de l'*event* `onload()`:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>El meu primer programa</title>
    </head>
    <script>
        function carregar() {
          //declarem una variable
          let x = 3;
          /* Els comentaris també
          poden ser multilínia.
          Declarem una altra variable */
          let y = 5;
          let z = x*y;
          document.getElementById("capcalera").innerHTML = z;
          console.log(z);
          window.alert(z);
          document.write(z);
        }
    </script>
    <body onload="carregar()">
        <h1 id="capcalera">El meu primer programa</h1>
    </body>
</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ZEBKBGp?editors=1001.

Només quan s'ha carregat tota la pàgina HTML s'executa la funció JavaScript *carregar()*, que ara sí que ja podem definir en el `head`, la qual cosa fa que el codi sigui més ordenat.

En aquest segon cas haureu vist que la sortida per pantalla és diferent (només hi ha un valor del 15), i que la consola està buida. Això és degut a què utilitzar *document.write()* un cop s'ha carregat totalment el document HTML fa que s'esborri la sortida HTML (*document.write* s'acostuma a utilitzar només per fer proves).

En aquest exemple hem utilitzat el mètode `getElementById()` de l'objecte `document`. Aquest mètode, molt útil, ens permet fer una cerca de l'element HTML que té per *id="capcalera"*.

Fixem-nos que, pretenent fer una petita introducció a les sentències i variables, hem hagut d'introduir diversos conceptes com ara objectes, mètodes, fins i tot l'*event* `onload()`.

Us podeu registrar i practicar en aquesta plataforma, tot i que també podreu fer proves amb els exemples tot editant-los i provant-los directament en el navegador.

1.10.2 Tipus de dades

A JavaScript les variables poden emmagatzemar molts tipus de dades, incloent-hi números, cadenes de text, objectes i funcions:

```
let dies = 365;
let cadena1 = "curs de JavaScript";
let cadena2 = 'IOC';
let objecte = {nom: 'Joan', edat: 23};
let hola = function() {alert('Hola món!');};
```

Observeu que els números van sense cometes, i les cadenes de text van entre cometes simples o dobles. Si posem un número entre cometes, el tracta com una cadena de text.

```
let dies = '365';
let cad = "L'any té " + dies + " dies";
```

En l'anterior exemple podeu veure per què és útil que les cadenes es puguin emmarcar entre cometes dobles o simples, tot i que també es pot utilitzar el caràcter d'escapament (contrabarra, '\'):

```
let dies = '365'; // es tracta d'una cadena de text
let cadena = 'L´\any té ' + dies + ' dies';
```

Utilització de cometes dobles o simples

En general no hi ha diferència entre utilitzar cometes dobles o simples quan es treballa amb cadenes de text. Si la cadena de text conté cometes dobles, es recomana fer servir les cometes simples per estalviar-nos d'escapar les cometes dobles, i viceversa.

Cal tenir en compte que la concatenació de cadenes fa la conversió automàtica de nombres a cadenes de text sí que és possible. Per exemple:

```
let dies = 365; // es tracta d'un nombre
let cadena = 'L´\any té ' + dies + ' dies';
```

Una altra manera de construir cadenes de text és utilitzar plantilles de literals o *template literals*, en anglès (anteriorment conegudes com a plantilles de cadenes de text o *template strings*), en lloc de concatenar cadenes de text amb variables:

```
let dies = 365; // es tracta d'un nombre
let cadena = 'L´\any té ${dies} dies'; // plantilla de literal
```

Les plantilles de literals es troben entre accents greus (`) i permeten la interpolació de variables escrites entre claus precedides per un signe de dòlar: `${variable}`.

La utilització d'aquestes plantilles facilita la llegibilitat i la interpolació de variables i respecta els salts de línia dintre de la plantilla, com es pot apreciar en el següent exemple:

```
let dies = 365;
let mesos = 12;
let cad = 'L´\any té:
${dies} dies
${mesos} mesos';
```

Després de declarar una variable, aquesta no té valor (o diem que té valor `undefined`). És el que passa en el següent exemple:

```
let color;
console.log(color);
```

Per donar valor a una variable, utilitzem l'operador d'assignació ('='):

```
color = 'blau';
```

Podem declarar moltes variables en una sola sentència:

```
let color1 = 'vermell', color2 = 'taronja', color3 = 'groc';
```

A continuació podeu veure un exemple complet d'utilització de variables:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Tipus de dades</title>
    </head>
    <script>
        console.clear();

        let dies = 365;
        let curs = "curs de JavaScript";
        let ioc = 'IOC';
        console.log (curs);
        console.log (ioc);

        let cadCometesDobles = "L'any té " + dies + " dies";
        console.log (cadCometesDobles);

        let cadCometesSimples = 'L´\any té ' + dies + ' dies';
        console.log (cadCometesSimples);

        let cadPlantillaLiteral = '´Lany té
        ${dies} dies';
        console.log (cadPlantillaLiteral);
```

```
25
26          let color;
27          console.log(color); // undefined
28          color = 'blau';
29          console.log(color); // blau
30
31
32          let color1 = 'vermell', color2 = 'taronja', color3 = 'groc';
33          console.log (color1);
34          console.log (color2);
35          console.log (color3);
36      </script>
37      <body>
38          <h1>Tipus de dades</h1>
39      </body>
40  </html>
```

</code>

Podeu provar el darrer exemple a: codepen.io/ioc-daw-m06/pen/qBbLQjw?editors=1001

Quan proveu aquest exemple podeu utilitzar tant la consola integrada en el *codepen* com la consola del vostre navegador web.

L'onzena edició d'ECMAScript defineix nou tipus:

Tipus de dades

Es recomana consultar el lloc web de Mozilla en anglès per veure els tipus de dades definits (mzl.la/3hiYYLy), ja que en altres llocs web pot ser que la llista no estigui actualitzada a l'última edició del llenguatge.

Enumeracions

Les enumeracions s'utilitzen per associar valors únics i constants d'una manera entenedora. Exemples: els colors d'un semàfor, les opcions d'una enquesta o els punts cardinals.

- Sis **tipus de dades** que són primitius (diferenciats per l'operador `typeof`):
  - **undefined**: variable a la qual no s'ha assignat cap valor o no declarada.
  - **boolean**: cert o fals.
  - **number**: nombres enters i reals, incloent-hi alguns valors especials com `infinity`.
  - **string**: cadenes de text.
  - **bigInt**: permet operar amb nombres enters de mida arbitrària, no està limitada la mida com passa al tipus `number`. Cal afegir `n` al final d'un nombre per convertir-lo en `BigInt`. Fixeu-vos que un nombre no ha de ser obligatòriament "gran" per declarar-lo com a `BigInt`. Pot aplicar-se a qualsevol valor enter, per exemple: `let x = 42n`.
  - **Symbol**: és un tipus especial que permet crear funcionalitats similars a les enumeracions d'altres llenguatges, ja que cada `symbol` que es crea es garanteix que es tracta d'un valor únic.
- **null**: es tracta d'un valor especial. L'operador `typeof` el considera un `object` i és el valor retornat per algunes funcions quan el resultat no és vàlid, a diferència d'`undefined`, que és el valor d'una variable a la qual no s'ha assignat cap valor.
- **Object**: tipus assignat a gairebé qualsevol element instanciat mitjançant la paraula clau `new` (*array*, *map*, *set*, *date*, etc.), així com a objectes declarats com a literals (per exemple, `{nom: 'Joan', edat:'23'}`).
- **Function**: tipus assignat a les funcions.

Si executeu el següent exemple, podeu veure a la consola el valor retornat per l'operador `typeof` per a cadascun d'aquests tipus:

```
let activat = true;
console.log(‘Activat: ${activat}. Tipus: ${typeof activat}‘);
let tipusNul = null;
console.log(‘Tipus nul: ${tipusNul}. Tipus: ${typeof tipusNul}‘);
let tipusSenseDefinir;
console.log(‘Tipus sense definir: ${tipusSenseDefinir}. Tipus: ${typeof
    tipusSenseDefinir}‘);
let nombre = 42;
console.log(‘Nombre: ${nombre}. Tipus: ${typeof nombre}‘);
let nombreEnterGran = 42n;
console.log(‘Nombre enter gran: ${nombreEnterGran}. Tipus: ${typeof
    nombreEnterGran}‘);
let cadena = "Hola món!";
console.log(‘Cadena: ${cadena}. Tipus: ${typeof cadena}‘);
let simbol = Symbol("Simbol");
console.log(‘Simbol: ${simbol.description}. Tipus: ${typeof simbol}‘);
let objecte = {nom: ’Joan’, edat: 23};
console.log(‘objecte: ${objecte.nom}, ${objecte.edat}. Tipus: ${typeof objecte
    }‘);
let funcio = function() {return ’funció cridada!’};
console.log(‘Retorn de la funció: ${funcio()}. Tipus: ${typeof funcio}‘);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/JjGwxNO?editors=0012.

Utilitzant plantilles de literals és possible interpolar no només valors de variables sinó també propietats d'objectes, i inclús els valors retornats per funcions.

Embolcall automàtic de valors primitius en objectes

Cal destacar que JavaScript embolcalla els valors dels tipus primitius `boolean`, `string` i `number` automàticament en objectes quan accedim a mètodes i propietats.

Per exemple, l'objecte `Boolean` implementa el mètode `toString()`, que converteix el valor en la cadena `true` o `false`. Així, doncs, quan escrivim `true.toString()`, el que ocorre internament és que el valor `true` s'embolcalla en un objecte de tipus Boolean i es crida a aquest mètode.

Conversió de tipus

Per operar amb diferents variables, JavaScript ha de conèixer el tipus de dades de les variables, i aleshores pot aplicar les seves regles internes.

Per exemple, en JavaScript és vàlid escriure:

```
let animal = "Àliga"; // String
let numPotes = 2; // Number

console.log (animal + numPotes);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/dyGwEGd?editors=0012.

Això passa perquè internament `numPotes` es converteix a cadena, que es concatena amb la variable `animal`.

A JavaScript les expressions s'avaluen d'esquerra a dreta. Aquestes dues sentències donen resultats diferents:

```
let animal = "Àliga"; // String
```

```
let numPotes = 2; // Number

console.log (numPotes + numPotes + animal); // 4Àliga
console.log (animal + numPotes + numPotes); // Àliga22
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/mdVaZXV?editors=0012.

En el primer cas s'han sumat els valors de `numPotes` i, a continuació, s'ha concatenat "`Àliga`" (Number + Number = tipus Number i després Number + String = tipus String), mentre que en el segon cas primer es produeix la concatenació de "`Àliga`" i 2, i a la cadena resultant es concatena el valor 2.

Per la mateixa raó, si el valor d'una de les variables fos un nombre entre cometes (per exemple: "2"), el resultat seria la concatenació dels dos nombres en lloc de la suma:

```
let numPotes = 2; // Number
let numCues = "1"; // String

console.log (numPotes + numCues);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/zYryQWd?editors=0012.

Per asegurar que el tipus és l'esperat, JavaScript ofereix la posibilitat de fer les següents conversions de tipus:

- `Boolean(valor)`: converteix el valor booleà.
- `String(valor)`: converteix el valor en una cadena de text. Una altra opció és fer una concatenació del valor amb una cadena buida.
- `Number(valor)`: converteix el valor en un nombre. Si el valor no és vàlid, el resultat serà `NaN` (no és un nombre).
- `parseInt(valor)`: converteix el valor en un nombre enter encara que es trobi un separador decimal.
- `parseFloat(valor)`: converteix el valor en un nombre real.

```
let cadena = "3.1415";
let nombre = "42";
let nom = "Joan";
let aprovat = true;

// Conversions a booleà
console.log(Boolean(cadena)); // true
console.log(Boolean(0)); // false
console.log(Boolean("")); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false

// Conversió a cadena
console.log(String(nombre)); // "42"
console.log(String(nombre) + nombre); // "4242"
console.log(String(aprovat)); // "true"

// Conversions a nombres
```

```
19 console.log(Number(nom)); // NaN, no es un nombre
20 console.log(Number(cadena) * 2); // 6.283
21 console.log(Number(aprovat)); // 1
22 console.log(parseInt(cadena)); // 3
23 console.log(parseFloat(cadena)); // 3.1415
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/bGEzBmG?editors=0012.

Cal tenir en compte que JavaScript no dona cap altra opció per fer conversions de tipus, però això no suposa cap inconvenient perquè és innecessari, ja que no hi ha restriccions als valors que es poden passar a les funcions i els tipus de les variables són dinàmics. És a dir, que la mateixa variable pot canviar de tipus de dades sense problema:

```
1 let z = 34;
2 z = "ara soc una cadena";
3 z = true;
4 z = undefined;
```

Podeu comprovar que aquest exemple és vàlid en l'enllaç següent: codepen.io/ioc-daw-m06/pen/bGEzBmG?editors=0012.

Tipus de dades compostes

Un **tipus de dades compost** és un tipus que permet emmagatzemar en una sola variable més d'un valor. A diferència d'altres llenguatges, a JavaScript les dades emmagatzemades **poden ser de diferents tipus**, ja que les variables són dinàmiques.

Originalment, a JavaScript només existien dos tipus de dades compostes: els *arrays* i els objectes.

Un *array* permet emmagatzemar múltiples valors en una variable i manipular-los mitjançant un índex.

```
1 let dies = ["dilluns", "dimarts", "dimecres", "dijous", "divendres", "dissabte"
      , "diumenge"];
2 console.log(‘Dia 3: ${dies[2]}‘);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/Bajveqa?editors=0012.

A la primera línia es declara un *array* assignant com a valor una llista de cadenes de text separades per comes entre claudàtors, i a la segona línia s'accedeix al valor emmagatzemat a la posició 2, que correspon al tercer valor perquè les posicions dels *arrays* comencen a comptar des de 0.

Històricament, a JavaScript s'han fet servir objectes com diccionaris de dades o mapes, que són uns tipus de dades compostes que admeten parells de valors, que són tractats com a parells propietat-valor:

```
1 var temperatures = {dilluns: 24, dimarts: 23, dimecres: 25, dijous: 20,
      divendres: 21, dissabte: 20, diumenge: 25};
2 console.log(‘La temperatura de dijous va ser ${temperatures.dijousº}C‘);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/QWyYGzy?editors=0012.

Format JSON

El format JSON es tracta d'un format basat en la notació d'objectes de JavaScript molt utilitzat per enviar informació des de servidors a aplicacions o per crear fitxers que són llegits des del disc (per exemple, fitxers de configuració).

A diferència dels *arrays*, l'assignació literal d'aquest tipus es realitza envoltant els valors entre claus, separant els parells de valor i clau amb dos punts, i cada parell amb una coma. Per altra banda, per accedir als valors es fa servir el punt, indicant a l'esquerra el nom de la variable i a la dreta la clau. Aquest tipus de dades és l'origen del format d'intercanvi de dades JSON, que comparteix exactament la mateixa estructura.

Cal destacar que encara que les claus no es trobin entre cometes simples ni dobles, internament es tracta de cadenes de text, ja que el nom de les propietats són de tipus `string`. Això permet accedir a aquests valors indicant la clau com si es tractés d'una cadena de text fent servir claudàtors:

```
1 var temperatures = {dilluns: 24, dimarts: 23, dimecres: 25, dijous: 20,
      divendres: 21, dissabte: 20, diumenge: 25};
2 console.log(‘La temperatura de dijous va ser ${temperatures["dijous"º]}C‘);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/xxZMRMW?editors=0012.

> Els ***arrays* són una col·lecció de dades ordenades** mentre que els **objectes utilitzen claus** per accedir i manipular els valors.

Actualment a JavaScript existeixen altres tipus de dades compostes:

- **Map**: és un diccionari de dades. És com utilitzar un objecte, però la clau pot ser qualsevol tipus i no només `string` (per exemple: nombres, booleans o inclús objectes).
- **Set**: és un tipus especial per crear llistes sense elements duplicats que no dona accés a valors concrets. Permet afegir elements, comprovar si un element es troba al conjunt, eliminar elements i recórrer tot conjunt, però no accedir a valors concrets, perquè no s'assigna als valors ni un índex ni una clau.

Tipus: undefined i null

Recol·lector de brossa

El recol·lector de brossa (*Garbage Collector* en anglès) és un procés que periòdicament comprova les referències als objectes que es troben a memòria i quan no troba cap referència els elimina.

Els tipus `undefined` i `null` són molt similars, tots dos permeten alliberar un objecte (és decrementa el comptador de referències de l'objecte per indicar al recol·lector de brossa quan s'ha d'eliminar de la memòria) i s'avaluen com a fals quan fem operacions booleanes.

Una diferència important entre `undefined` i `null` és que tot i que els dos són tipus, l'operador `typeof` retorna `undefined` i `object` respectivament, ja que `null` es considera un objecte.

Habitualment no assignarem el valor `undefined`, ja que aquest és el valor assignat automàticament pel llenguatge a les variables a les quals no s'ha assignat cap valor,

als paràmetres de les funcions quan es criden sense haver passat tots els paràmetres o quan s'intenta consultar una propietat que no existeix a un objecte.

En canvi, el valor `null` l'assignarem si és necessari alliberar una variable o quan sigui necessari indicar que no es vol assignar cap valor a la variable. Per exemple, perquè s'ha produït una entrada de valors errònia, perquè no s'ha pogut crear un objecte o perquè no s'ha trobat un objecte.

Un avantatge d'utilitzar el valor `null` és que ens indica que s'ha assignat expressament, mentre que el valor `undefined` no podem saber si ha estat assignat pel llenguatge (per exemple, perquè una variable o propietat no ha estat definida o perquè hem escrit malament el nom de la variable) o ha estat assignat expressament pel desenvolupador.

Tipus: Boolean

El tipus booleà només accepta dos valors: `true` i `false` (cert i fals). Aquest és el tipus que retornen les operacions comparatives de valors com: igual, major que, menor que, diferent, etc.

Cal distingir entre un **valor de tipus booleà** i un **objecte de tipus booleà**, ja que les instruccions condicionals tracten qualsevol objecte (a excepció de `null`) com a cert. Podeu comprovar-ho amb el següent exemple:

```
// Assignem com a valor un objecte Boolean amb el valor false;
let x = new Boolean(false);
let y = false;
console.log(‘Valor de x: ${x}, tipus: ${typeof x}‘);
console.log(‘Valor de y: ${y}, tipus ${typeof y}‘);

if (x) {
  console.log("Codi executat malgrat que x es fals");
}

if (y) {
  // això no s’executa mai
  console.log("Codi executat malgrat que y es fals");
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/abdXWOB?editors=0012.

En canvi, si en lloc de fer servir l'operador `new` es crida `Boolean()` com a funció, el retorn serà un valor primitiu i el resultat serà l'esperat:

```
let x = Boolean(false);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/YzwBVpa?editors=0012.

Tipus: Number i BigInt

A JavaScript només existeixen dos tipus de dades pels nombres:

- Number: utilitzat per a treballar amb nombres enters i reals:
- BigInt: afegit recentment al llenguatge per a treballar amb enters de mida arbitrària.

Els nombres normals estan limitats a una mida màxima, un cop superada aquesta mida el seu valor passa a ser `Infinity` o `-Infinity`. Per altra banda en treballar amb enters hi ha un límit a partir del qual les operacions no són correctes. Es poden consultar aquests valors amb `Number.MAX_SAFE_INTEGER`, `Number.MIN_SAFE_INTEGER`, `Number.MAX_VALUE` i `Number.MIN_VALUE`, com es pot comprovar en el següent exemple:

```
let limitRealA = Number.MAX_VALUE; // Valor máxim segur pels nombres reals
let limitRealB = Number.MIN_VALUE; // Valor mínim segur pels nombres reals
let limitRealExces1 = limitRealA + 1;
let limitRealExces2 = limitRealA + 2;

console.log(‘Rang segur de nombres reals: [${limitRealB},  ${limitRealA}]‘);
console.log(‘Limit superior de nombres reals + 1: ${limitRealExces1}‘);
console.log(‘Limit superior de nombres reals + 2: ${limitRealExces2}‘);
console.log(‘Són iguals els limits superior +1 i +2?: ${limitRealExces1 ===
    limitRealExces2}‘);
console.log(‘Diferència dels limits superiors +1 i +2: ${limitRealExces2 –
    limitRealExces1}‘);
console.log(‘Quadrat del limit real superior: ${limitRealA ** 2}‘);

let limitEnterA = Number.MAX_SAFE_INTEGER; // Valor máxim segur pels nombres
    enters
let limitEnterB = Number.MIN_SAFE_INTEGER; // Valor mínim segur pels nombres
    enters
let limitEnterExces1 = limitEnterA + 1;
let limitEnterExces2 = limitEnterA + 2;

console.log(’Rang segur de nombres enters: [${limitEnterB},  ${limitEnterA}]’);
console.log(’Limit superior de nombres enters + 1: ${limitEnterExces1}’);
console.log(’Limit superior de nombres enters + 2: ${limitEnterExces2}’);
console.log(’Són iguals els limits superiors d´\enters +1 i +2?: ${
    limitEnterExces1 === limitEnterExces2}’);
console.log(’Diferència dels limits superiors d´\enters +1 i +2: ${
    limitEnterExces2 – limitEnterExces1}’);
console.log(’Quadrat del limit enter superior: ${limitEnterA ** 2}’);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/LYGqLbM?editors=0012.

Com es pot apreciar, els resultats ja no són correctes un cop se superen els màxims: en incrementar en 1 i 2 els valors màxims no existeix cap diferència entre els resultats, sigui comparant-los o operant la diferència. Per altra banda, un cop el nombre és suficientment gran (com es pot veure en el cas del quadrat del límit real) el valor es torna infinit.

En alguns casos no es recomana treballar amb nombres reals (especialment quan es treballa amb diners), ja que els nombres reals afegeixen errors de precisions. Per exemple `0.1 * 0.2 = 0.020000000000000004`. Una alternativa és convertir els valors en enters multiplicant-los per una potència de 10, d'aquesta manera es conserven tots els dígits significatius.

Per a resoldre aquest problema, s'ha afegit al llenguatge el tipus BigInt, que permet treballar amb enters de qualsevol mida com es pot comprovar en el següent exemple:

```
let limitEnterA = Number.MAX_SAFE_INTEGER; // Valor máxim segur pels nombres enters
let limitEnterB = Number.MIN_SAFE_INTEGER; // Valor mínim segur pels nombres enters

let enterGranA = BigInt(Number.MAX_SAFE_INTEGER); // Valor máxim segur pels nombres enters
let enterGranIncrement1 = enterGranA + 1n;
let enterGranIncrement2 = enterGranA + 2n;

console.log('Rang segur de nombres enters: [${limitEnterB},  ${limitEnterA}]');
console.log('Límit superior de nombres enters + 1n: ${enterGranIncrement1}');
console.log('Límit superior de nombres enters + 2n: ${enterGranIncrement2}');
console.log('Són iguals els limits superiors ´denters grans +1n i +2n?: ${enterGranIncrement1 === enterGranIncrement2}');
console.log('Diferència dels limits superiors ´denters grans +1n i +2n: ${enterGranIncrement2 - enterGranIncrement1}');
console.log('Quadrat de ´lenter gran: ${enterGranA ** 2n}');
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/mdVvMPR?editors=0012.

Com es pot apreciar en executar-se aquest exemple, l'avaluació d'igualtat entre `enterGranIncrement1` i `enterGranIncrement2` dona fals i la diferència entre els dos valors és `1`, per consegüent el problema es resol correctament.

Fixeu-vos que un cop convertit el valor enter en BigInt només pot operar amb altres nombres de tipus BigInt, és a dir, `1n + 1` no és vàlid i es produeix un error.

Tipus: String

Les *strings* es poden posar entre cometes simples o dobles.

```
let animal = "àliga";
console.log (animal);
```

Les *strings* permeten fer el següent:

```
console.log (animal.length);
console.log (animal.toUpperCase());
```

És a dir, l'*string* s'embolcalla automàticament en un objecte de tipus `string` que té propietats i mètodes. Per tant es pot tractar com un objecte i, depenent de com es declari, el seu tipus serà `String` o `Object`:

```
let animal = "àliga"
console.log (animal);
console.log (typeof(animal));

let vegetal = new String("bleda");
console.log (vegetal);
console.log (typeof(vegetal));
```

Aquesta segona forma, però, és més ineficient i, per tant, no la fem servir.

Tipus: Object

Els objectes es troben explicats amb més profunditat a la unitat "Objectes definits pel programador".

A JavaScript existeixen múltiples maneres de crear objectes. Actualment les dues més utilitzades són la instanciació a partir de classes mitjançant l'operador `new` i la declaració literal. En el cas de la declaració literal d'objectes s'han de definir entre claus.

Dels objectes en podem definir les propietats i els mètodes. Les propietats les escrivim amb parells `nom:valor`, separat per comes. Per exemple:

```
let animal = { id : 345, nom : "Gos", classe : "Mamífers", familia : "Cànids"
    };
```

Hem definit quatre propietats de l'objecte animal, i podem accedir fàcilment als seus valors:

```
console.log (animal.id)
```

L'objecte "animal" té quatre propietats: `id`, `nom`, `classe`, `familia`. I com que és un objecte, també podem definir-ne mètodes. L'exemple més senzill seria:

```
let gos = {
  id : 345,
  nom : "Gos",
  classe : "Mamífers",
  familia : "Cànids",
  getTaxonomia : function() {
    return `${this.classe} - ${this.familia}`;
  }
};

console.log (gos.getTaxonomia());
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/VwegMQd?editors=0012.

Com es pot apreciar, la creació d'objectes d'aquesta manera és molt concisa, però en cas de requerir crear més objectes amb la mateixa estructura (per exemple, un objecte que representi un gat) aquest sistema deixa de ser pràctic.

Classes en edicions anteriors

Les edicions anteriors de JavaScript permetien la creació d'objectes utilitzant un sistema basat en funcions constructores i prototips per simular la creació de classes. Tot i que aquest sistema continua sent vàlid, no es recomana utilitzar-lo, ja que és més enrevessat i en general no aporta cap avantatge.

La instanciació a partir de classes, afegida a JavaScript en l'edició ES2015, soluciona aquest problema, ja que permet crear una classe que serveixi com a plantilla per generar totes les instàncies necessàries:

```
class Animal {
  constructor(id, nom, classe, familia) {
    this.id = id;
    this.nom = nom;
    this.classe = classe;
    this.familia = familia;
  }

  getTaxonomia() {
    return `${this.classe} - ${this.familia}`;
  }
}

let gos = new Animal(1, 'Gos', 'Mamífers', 'Cànids');
```

```
15 let gat = new Animal(2, 'Gat', 'Mamífers', 'Fèlid');
16
17 console.log(gos.getTaxonomia());
18 console.log(gat.getTaxonomia());
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/RwrvZLV?editors=0012.

Fixeu-vos que un cop creada la classe és molt fàcil instanciar qualsevol quantitat d'elements mitjançant l'operador `new`, indicant el nom de la classe i passant com a arguments l'identificador, el nom, la classe i la família. A més a més, no cal tornar a definir la funció `getTaxonomia()`, ja que totes les instàncies la inclouen.

Tipus: Function

Una funció és un bloc de codi que rep uns valors d'entrada com a paràmetres, els processa i, opcionalment, en retorna un resultat. Per exemple:

Les funcions es troben explicades amb més profunditat a la unitat "Objectes definits pel programador".

```
1  function hola(nom) {
2    alert(`Hola ${nom}!`);
3  }
4
5  function suma(a, b) {
6    return a + b;
7  }
8
9  hola('Joan');
10 console.log(suma(2, 2));
```

Per cridar a una funció només cal indicar el nom de la funció i, entre parèntesis, els arguments necessaris separats per comes.

A JavaScript les funcions són objectes de primera classe, és a dir, són tractades com qualsevol altre variable:

- Una funció pot ser assignada a una variable, i per aquest motiu també pot ser emmagatzemada en un *array* o un *map*.
- Es pot passar una funció com a argument a una altra funció.
- Una funció pot retornar una altra funció.
- Com que es tracta d'un objecte, té mètodes propis que poden ser cridats sobre aquest. Per exemple, `apply`, `bind`, o `call`.

Cal tenir en compte que quan parlem d'objectes s'acostuma a parlar de mètodes, però també són funcions.

Per altra banda, a l'edició ES2015 es va incloure una sintaxi més compacta d'expressar les funcions anomenada "expressió de funció fletxa", que és molt pràctica per passar com a argument en alguns casos. Aquest tipus de sintaxi no es pot utilitzar com a constructor ni es recomana utilitzar-la per definir mètodes.

1.10.3 Operadors

Els operadors serveixen per fer operacions entre dues o més variables. El primer operador que s'esmenta és el d'assignació. Després vénen els operadors aritmètics, que serveixen per fer operacions aritmètiques:

- Suma (+)
- Resta (-)
- Multiplicació (*)
- Divisió (/)
- Mòdul (%)
- Increment (++)
- Decrement (--)

```
let a = 100;
a = ((a + 10 – 15 ) * 10 / 2 ) % 2;
```

Com a operador de cadena tenim la concatenació. Es representa amb el símbol + en el sentit que estem sumant cadenes.

```
let cad1 = "curs de javaScript";
let cad2 = " de l'IOC";
let cad = cad1 + cad2; // curs de javaScript de l'IOC
```

Com que a JavaScript els tipus són dinàmics, si sumem números i cadenes, el més lògic és que el número passi a ser una cadena, per tal de poder fer una concatenació (al revés no té cap sentit):

```
console.log (a + cad);
console.log (cad + a);
```

Operadors lògics i de comparació

Els operadors lògics i de comparació disponibles són:

- igual en valor que (==)
- igual en valor i en tipus que (===)
- no igual en valor que (!=)
- no igual en valor o en tipus que (!==)
- més gran que (>)

- més petit que (<)
- més gran o igual que (>=)
- més petit o igual que (< =)
- conjunció lògica (&&)
- disjunció lògica (||)
- negació lògica (!)
- operador ternari (?)

Per evitar confusions **sempre** farem servir els operadors === i !== en lloc de == i !=, ja que la utilització d'aquests últims pot generar resultats inesperats com: `10 == "10"` o `0 == false`.

L'operador ternari (?) mereix un tractament especial. La seva funció és clara amb els condicionals.

Assignació de valors i comparació lògica

No hem de confondre l'assignació (=) amb la comparació lògica (==). És un error freqüent en els principiants.

```
let a = 10;
let b = 10;
let c = 20;
let d = "10";

console.log (a == c); // false
console.log (a === b); // true
console.log (a == d); // true
console.log (a === d); // false
console.log (a != c); // true
console.log (a !== d); // true
console.log (a > c); // false
console.log (a < c); // true
console.log (a >= b); // true
console.log (a <= b); // true
```

La comparació entre cadenes es fa alfabèticament, com era d'esperar:

```
let cad1 = "avió";
let cad2 = "vaixell";
console.log (cad1 > cad2); // false
```

Si fem una comparació entre diferents tipus, com entre cadena i número, podem obtenir resultats inesperats. JavaScript converteix la cadena a número. Si la cadena és no-numèrica es converteix a NaN (*Not a Number*), que sempre serà fals. Si la cadena és buida, es converteix a 0.

Els operadors de bit

Els operadors de bit (*bitwise operators*) no els veurem ja que difícilment es faran servir en el marc de la programació web. Si voleu aprofundir-hi, trobareu més informació a mzl.la/3s0JAsv

```
let a = 10;
let cad1 = "10";
let cad2 = "15";
let str = "cotxe";

console.log (a < cad2); // true. El '15' es converteix a 15
console.log (a == str); // false. La cadena 'cotxe' no es pot convertir a nú
    mero
console.log (cad1 < cad2); // true. Els dos són cadenes. S'ha de fer una
    comparació alfabètica. Com que el primer caràcter és el mateix, s'ha de
    mirar el segon caràcter.
```

Els exemples anteriors els podeu provar a: codepen.io/ioc-daw-m06/pen/KKVJQXw?editors=0012.

Operador de propagació

L'operador de propagació es representa com `...` i s'utilitza per "propagar" els valors d'un *array*, de manera que els elements de l'*array* se "separen" com elements individuals. D'aquesta manera és possible passar un *array* propagat com argument a una funció que accepti múltiples arguments. Per exemple:

```
let dades = ['Joan', 27];
mostrarDades(...dades);

function mostrarDades(nom, edat) {
  console.log(`${nom} té ${edat} anys`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/gOPqvBy?editors=0012.

En aquest exemple el primer valor de l'*array* s'assigna al paràmetre `nom` i el segon element, al paràmetre `edat`.

Una altra aplicació de l'operador de propagació és treballar amb *arrays*, ja que propagant un *array* podem interpolar-los fàcilment i afegir tots els elements d'un *array* a un altre d'una tacada.

```
let colors = ['blau', 'vermell', 'groc'];
let fruites = ['taronja', ...colors, 'llimona'];
console.log(fruites);

let nousColors = ['taronja', 'rosa', 'blanc', 'negre'];
colors.push(...nousColors);
console.log(colors);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/wvMNyZv?editors=0012.

Fixeu-vos que a l'*array* `fruites` al punt on s'ha propagat `colors` s'han afegit tots els elements individualment.

En el segon cas es crida al mètode `push()` de l'*array* colors, que serveix per afegir un element al final de l'*array*, però com que s'ha propagat `nousColors`, en lloc d'afegir l'*array* com un únic element s'afegeix cada element de `nousColors`, un a un.

Es pot veure un exemple d'utilitació de l'operador *Rest* a la secció "Assignacions compostes" d'aquest mòdul.

Finalment, cal tenir en compte que hi ha un cas en què `...` no és l'operador de propagació sinó l'operador *Rest*, que té el funcionament contrari, i el que fa és assignar a la variable a la qual s'aplica la "resta" de valors no assignats d'un *array*.

1.10.4 Assignacions compostes

Els operadors d'assignació assignen valor a les variables. A part de l'operador igual ('='), a continuació podeu trobar una llista dels operadors compostos més utilitzats:

Es pot consultar la llista completa d'operadors d'assignació compostos al següent enllaç: mzl.la/2ZPhwgp

- `a += b` equival a **a = a + b**
- `a -= b` equival a **a = a - b**
- `a *= b` equival a **a = a * b**
- `a /= b` equival a **a = a / b**
- `a %= b` equival a **a = a % b**

Vegeu-ne un exemple de cada:

```
let a = 100;
let b = 2;

a += 10; // 110
console.log(a);
a -= 15; // 95
console.log(a);
a *= 10; // 950
console.log(a);
a /= 2; // 475
console.log(a); // 475
a %= 2; // 1 (el mòdul és 1, doncs 475 és senar)
console.log(a); // 1 (el mòdul és 1, doncs 475 és senar)
```

Proveu l'exemple anterior a: codepen.io/ioc-daw-m06/pen/vYLbWwL?editors=0012.

Una de les noves incorporacions a JavaScript és la sintaxi d'"assignació desestructurant", que facilita l'extracció de dades d'*arrays* i les propietats d'objectes.

Per assignar els valors d'un *array* a la banda esquerra s'ha de posar entre claudàtors els noms de les variables a les quals s'assignaran els valors i a la banda dreta, l'*array*:

```
let a, b, altres;
let valors = [10, 20, 30, 40, 50, 60];

[a, b, ...altres] = valors;
console.log(a); // 10
console.log(b); // 20
console.log(altres); // [30, 40, 50, 60]
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/bGEzaOx?editors=0012.

Fixeu-vos que és possible utilitzar l'operador ... (en aquest context és l'operador *Rest*) a l'última variable, de manera que els valors no assignats de l'*array* s'assignen a aquesta variable.

Per altra banda, per assignar els valors de propietats a variables, cal posar tota l'assignació entre parèntesis. A la banda esquerra s'afegeixen les variables entre claus i a la banda dreta l'objecte del qual s'extrauran les propietats.

```
let nom, edat;
let objecte = {nom: 'Joan', edat: 27};

({nom, edat} = objecte);
console.log(nom); // Joan
console.log(edat); // 27
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/qBbgxdm?editors=0012.

Cal destacar que el valor de les variables només s'assignarà si es troba una propietat a l'objecte amb aquest mateix nom.

1.10.5 Decisions

Quan es programa, sovint cal prendre decisions i bifurcar el codi. Les instruccions condicionals ajuden a fer-ho. Les més habituals són `if`, `else` i `else if`, d'una banda, i la clàusula `switch`, d'altra banda.

Per altra banda, a JavaScript existeix una sintaxi de gestió d'errors que permet executar un bloc de codi o altre segons si es produeix un error o no, es tracta de la estructura `try...catch`.

If, else, else if

S'usa `if` per especificar un bloc de codi que s'executarà quan es compleixi una determinada condició:

Cal utilitzar el sagnat (*identation*) dels blocs per tal que el codi sigui més llegible.

```
let a = 99, b = 98;
if ( a % 3 == 0) console.log(`${a} és divisible per 3`);

if ( b % 2 == 0) {
  console.log(`${b} és divisible per 2`);
  console.log(`${b} és parell`);
}
```

S'usa `else` per especificar el bloc de codi que s'executarà quan la condició és falsa:

```
let a = 99;

if ( a % 2 == 0) {
  console.log (`${a} és divisible per 2`);
  console.log (`${a} és parell`);
} else {
  console.log (`${a} és senar`);
}
```

S'usa `else if` per especificar noves condicions a avaluar després que s'hagin avaluat les prèvies:

```
let b = 98;

if ( b % 4 == 0) {
  console.log (‘${b} és divisible per 4‘);
  console.log (‘${b} és parell‘);
} else if ( b % 3 ) {
  console.log (‘${b} és divisible per 3‘);
} else if ( b % 2 ) {
  console.log (‘${b} és divisible per 2‘);
  console.log (‘${b} és parell‘);
} else {
  console.log (’un altre cas’);
}
```

Evidentment, l'avaluació de la condició pot ser més complicada:

```
let year = 2016;
if ((year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0))) {
  console.log (‘´Lany ${year} és un any de traspàs.\nPerò això no vol dir que
      tots els anys de traspàs siguin divisibles per 4.\´nLany 2000 no va ser
      un any de traspàs!‘);
  document.write (‘´Lany ${year} és un any de traspàs.<br />Però això no vol
      dir que tots els anys de traspàs siguin divisibles per 4.<br />´Lany
      2000 no va ser un any de traspàs!‘);
} else {
  console.log (‘´Lany ${year} NO és un any de traspàs.‘);
  document.write (‘´Lany ${year} NO és un any de traspàs.‘);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/qBbgYaE?editors=0011

En el context de la consola, per fer una línia nova fem servir el metacaràcter `\n`. Però en el context de la pàgina web cal fer servir `<br />`.

Per tal d'ampliar l'exemple anterior, mirarem si l'any actual és de traspàs, i després mirarem si un any aleatori entre 0 i 2500 també és de traspàs. Hem d'utilitzar els objectes `Date` i `Math`, que tots dos tenen la seva col·lecció de propietats i mètodes.

Quan s'utilitzen plantilles de literals pot utilitzar-se un salt de línia dins del codi en lloc del metacaràcter `\n`.

Recordeu que no es pot fer servir `any` com a nom de variable perquè és una paraula reservada de JavaScript.

```
let d = new Date();
year = d.getFullYear();
if ((year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0))) {
  console.log (‘´Lany ${year} és un any de traspàs.
Però això no vol dir que tots els anys de traspàs siguin divisibles per 4.´
Lany 2000 no va ser un any de traspàs!‘);
  document.write (‘´Lany ${year} és un any de traspàs.<br />Però això no vol
      dir que tots els anys de traspàs siguin divisibles per 4.<br />´Lany
      2000 no va ser un any de traspàs!‘);
 } else {
  console.log (‘´Lany ${year} NO és un any de traspàs.‘);
  document.write (‘´Lany ${year} NO és un any de traspàs.‘);
 }

document.write(’<br>’);

var num = 2500 * Math.random()
year = parseInt(num);
if ((year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0))) {
  console.log (‘´Lany ${year} és un any de traspàs.
Però això no vol dir que tots els anys de traspàs siguin divisibles per 4.
L´\any 2000 no va ser un any de traspàs!‘);
  document.write (‘´Lany ${year} és un any de traspàs.<br />Però això no vol
      dir que tots els anys de traspàs siguin divisibles per 4.<br />L´\any
      2000 no va ser un any de traspàs!‘);
```

```
} else {
  console.log (‘´Lany ${year} NO és un any de traspàs.‘);
  document.write (‘´Lany ${year} NO és un any de traspàs.‘);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/wvMNjKr?editors=0011

Ara és el moment de fer programació estructurada i definir una funció per tal de reutilitzar el codi. A l'hora d'aprendre JavaScript, cal aplicar tots els conceptes de programació coneguts:

```
let year = 2016;
mostrarInfo(year);

let d = new Date();
year = d.getFullYear();
mostrarInfo(year);

let num = 2500 * Math.random()
year = parseInt(num);
mostrarInfo(year);

function mostrarInfo(year) {
  if ((year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0))) {
    console.log (‘´Lany ${year} és un any de traspàs.
Però això no vol dir que tots els anys de traspàs siguin divisibles per 4.´
Lany 2000 no va ser un any de traspàs!‘);
    document.write (‘´Lany ${year} és un any de traspàs.<br />Però això no vol
        dir que tots els anys de traspàs siguin divisibles per 4.<br />´Lany
        2000 no va ser un any de traspàs!<br />‘);
  } else {
    console.log (‘L´\any ${year} NO és un any de traspàs.‘);
    document.write (‘L´\any ${year} NO és un any de traspàs.‘);
  }
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/rNxPvmg?editors=0011.

Switch

S'utilitza `switch` quan tenim una sola expressió que s'ha de comparar moltes vegades, i només una és la correcta. En aquest exemple mostrem l'estació de l'any (primavera, estiu, tardor, hivern) en funció del mes, sense tenir en compte que els canvis d'estació es fan el dia 21 de març, 21 de juny, 21 de setembre i 21 de desembre:

```
let mes;
let estacio;

switch (new Date().getMonth()) {
  case 0:
    mes = "Gener";
    estacio = "Hivern";
    break;
  case 1:
    mes = "Febrer";
    estacio = "Hivern";
    break;
  case 2:
```

```
    mes = "Març";
    estacio = "Hivern a primavera";
    break;
  case 3:
    mes = "Abril";
    estacio = "Primavera";
    break;
  case 4:
    mes = "Maig";
    estacio = "Primavera";
    break;
  case 5:
    mes = "Juny";
    estacio = "Primavera a estiu";
    break;
  case 6:
    mes = "Juliol";
    estacio = "Estiu";
    break;
  case 7:
    mes = "Agost";
    estacio = "Estiu";
    break;
  case 8:
    mes = "Setembre";
    estacio = "Estiu a tardor";
    break;
  case 9:
    mes = "Octubre";
    estacio = "Tardor";
    break;
  case 10:
    mes = "Novembre";
    estacio = "Tardor";
    break;
  case 11:
    mes = "Desembre";
    estacio = "Tardor a hivern";
    break;
  default:
    console.log ("El codi mai passarà per aquí, però en altres casos podem
        utiltizar la clàusula default");
}

console.log (`Mes: ${mes}`);
console.log (`Estació: ${estacio}`);
```

Proveu l'exemple anterior a: codepen.io/ioc-daw-m06/pen/LYGqmjG?editors=0012.

Com que els *symbols* representen valors únics, també es poden utilitzar en un bloc *switch*:

```
// Definim els simbols pels possibles colors del semàfor
let vermell = Symbol('vermell');
let groc = Symbol('groc');
let verd = Symbol('verd');

// Assignem el color actual del semàfor
let colorSemafor = groc;

let missatge;

switch (colorSemafor) {
  case vermell:
    missatge = "no es pot passar!";
    break;
  case groc:
```

```
    missatge = "compte!";
    break;
  case verd:
    missatge = "es pot passar";
    break;
  default:
    missatge = "Error, no s'ha assignat cap color al semàfor!"
}

console.log(`Color del semàfor: ${colorSemafor.description} - ${missatge}`);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ZEQworN?editors=0012.

Cal destacar que si no es posa `break` en finalitzar un cas, es continuen executant casos un darrere l'altre fins que es troba un `break` o es tanca el bloc *switch*.

Això pot ser útil en alguns casos, però és considerat una mala pràctica perquè pot generar confusions, ja que si un altre desenvolupador analitza el codi no pot saber si ha estat intencionat o es tracta d'un error.

En cas de voler aprofitar aquesta característica es recomana afegir un comentari de manera que quedi clara la intenció:

```
let estacio;

switch (new Date().getMonth()) {
  case 0: // Caiguda intencionada
  case 1:
    estacio = "Hivern";
    break;
  case 2:
    estacio = "Hivern a primavera";
    break;
  case 3: // Caiguda intencionada
  case 4:
    estacio = "Primavera";
    break;
  case 5:
    estacio = "Primavera a estiu";
    break;
  case  6: // Caiguda intencionada
  case  7:
    estacio = "Estiu";
    break;
  case  8:
    estacio = "Estiu a tardor";
    break;
  case  9: // Caiguda intencionada
  case  10:
    estacio = "Tardor";
    break;
  case  11:
    estacio = "Tardor a hivern";
    break;
  default:
    console.log ("El codi no passarà mai per aquí, però en altres casos podem
        utiltizar la clàusula default");
}
console.log (`Estació: ${estacio}`);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/wvMNjNx?editors=0012.

Try...Catch

La sintaxi dels blocs `try...catch` també es troba en altres llenguatges com Java.

Quan JavaScript detecta un error, l'execució del bloc de codi queda interrompuda. Per evitar-ho podem gestionar aquests errors mitjançant els blocs `try...catch`. Fixeu-vos en el següent exemple, on es crida una funció que no existeix:

A la consola de Codepen no es mostra el contingut de l'objecte `Error`.

```
try {
  funcioInexistent();
  console.log('Continua execució');
} catch (error) {
  // El contingut d'aquest objecte pot variar segons el navegador
  console.log(error);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ZEQPzzz?editors=0012.

Quan es crida la funció, com que no existeix s'interromp l'execució del codi. Per aquest motiu no es mostra mai el missatge de "Continua execució", sinó que l'execució continua al bloc `catch`, que rep com a paràmetre `error`: un objecte de tipus `Error` amb la informació del missatge.

Dins del bloc `try` s'ha de posar el codi on pot produir-se l'error i al bloc `catch`, el codi que es vol executar en cas que es produeixi algun error.

Addicionalment es pot afegir un bloc `finally`, que serà executat tant si es produeix un error com si l'execució es realitza amb èxit. Vegeu-ho en el següent exemple:

```
try {
  funcioInexistent();
  console.log('Continua execució');
} catch (error) {
  // El contingut d'aquest objecte pot variar segons el navegador
  console.log(error);
} finally {
  console.log('Bloc executat');
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/LYGaPVW?editors=0012.

1.10.6 Bucles

Els bucles són necessaris quan es vol executar un codi un nombre determinat o indeterminat de vegades, potser fins que s'acompleixi una condició de sortida. Tenim diferents tipus de bucles:

- `for`: bucles que repeteixen el bloc de codi un nombre fix de vegades.
- `for...in`: bucles que recorren les propietats d'un objecte o els elements d'un *array*.

- `for...of`: bucle per recórrer elements iterables: *array*, *map*, *set*, l'objecte `arguments`, etc.
- `Array.forEach`: bucle per recórrer i processar tots els elements d'un *array*.
- `while`: bucles que repeteixen el bloc de codi mentre la condició de sortida sigui certa.
- `do...while`: igual que l'anterior, però la condició de sortida s'avalua al final.

Bucles for, for...in i for...of

La sintaxi bàsica del bucle `for` és:

```
for ( expressió1; expressió2; expressió3) {
  bloc de codi a executar;
}
```

L´`expressió1` s'avalua a l'inici; l´`expressió2` és la condició de sortida del bucle; l´`expressió3` s'executa a cada volta del bucle. Un exemple típic:

```
// taula de multiplicar del 6
let cad = "";

for (let i = 1; i <= 10; i++) {
  cad += "6 * " + i  + " = " + 6*i + "\n";
}

console.log(cad);
```

I amb dos bucles `for` niats podem fer totes les taules de multiplicar de l'1 al 10:

```
// Taules de multiplicar de l'1 al 10
let cad;

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

Les tres expressions dins el `for` són, de fet, opcionals. L'única cosa que hem de tenir en compte és de no provocar un bucle sense fi, la qual cosa penjaria el navegador web des del qual estem provant. Per tant, sempre hi haurà d'haver una condició de sortida, i podrem sortir del bucle amb la clàusula `break`:

```
let i = 0;
for (;;) {
  console.log(i);
  i++;
  if (i>15) break;
}
```

Els exemples anteriors els podeu provar a: codepen.io/ioc-daw-m06/pen/xxZMzqd?editors=0012.

Per recórrer els elements d'un *array* també es pot fer servir el bucle *for*, ja que els índexs dels *arrays* són consecutius i, per consegüent, només cal recórrer els valors des de 0 fins a l'últim element de l'*array*, que correspondrà a la mida de l'*array* menys 1:

```
let colors = ['vermell', 'blau', 'groc']

for (let i = 0; i < colors.length; i++) {
  console.log(`Color: ${colors[i]}`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/jOWdKaO?editors=0012.

Malauradament, aquest sistema no permet recórrer les propietats d'un objecte (per exemple, perquè s'ha utilitzat com una estructura de dades). Per iterar sobre les propietats d'un objecte cal utilitzar la sintaxi `for...in`:

```
let persona = {nom: 'Joan', edat: 23, poblacio: 'Martorell'};

for (let clau in persona) {
  console.log(`${clau}: ${persona[clau]}`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/LYGqrQe?editors=0012.

També es pot fer servir `for...in` amb *arrays*, però en aquest cas el valor de la variable correspondrà a l'índex.

Fixeu-vos que el valor que s'assigna a `clau` és el nom de la propietat; així doncs, per accedir al valor de la propietat a l'objecte, s'ha de posar la clau entre claudàtors: `persona[clau]`.

Però cap d'aquests mètodes serveix per recórrer els elements iterables *map* i *set*. Per iterar sobre aquest tipus d'estructures cal fer servir *for...of*:

```
let llista = new Set();
llista.add('Joan');
llista.add('Maria');
llista.add('Pere');
llista.add('Rosa');

for (let noms of llista) {
  console.log(`Noms: ${noms}`);
}

let persones = new Map();
persones.set('Joan', 23);
persones.set('Maria', 21);
persones.set('Pere', 43);
persones.set('Rosa', 52);

for (let [nom, edat] of persones) {
  console.log(`Nom: ${nom}, edat: ${edat}`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/vYLbrbQ?editors=0012.

Fixeu-vos que en el cas del mapa es requereixen dues variables: una per a la clau i una altra per al valor.

Com podeu observar, a diferència de `for...in`, que s'assignava a la variable el nom de la propietat, en el cas de `for...of` a la variable se li assigna el valor. Així doncs, és possible iterar sobre un *array* i obtenir directament els valors sense necesitat d'utilitzar un índex:

```
let colors = ['vermell', 'blau', 'groc']

for (let color of colors) {
  console.log(`Color: ${color}`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/OJMdEGK?editors=0012.

Bucles Array.forEach

En versions anteriors de JavaScript la utilització d'`Array.forEach` era més rellevant perquè el bucle s'executava en un àmbit de funció que era l'únic respectat per les variables.

Aquest sistema d'iteració permet recórrer tots els elements d'un *array* i cridar una funció per a cadascun dels elements.

```
let colors = ['vermell', 'blau', 'groc']

colors.forEach(mostrarMissatge);

function mostrarMissatge(color) {
  console.log(`Color: ${color}`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/yLeZqNM?editors=0012.

Cal destacar que l'única manera de sortir d'un bucle d'aquest tipus és llençant una excepció des de la funció.

Bucles while, do...while

El cas més senzill és el bucle `while`. Mentre es compleixi la condició s'executarà el bloc de codi:

```
while (condició) {
  bloc de codi
}
```

Un exemple que utilitza la funció `charAt()` de les *strings*. Mostrem la part de la cadena `str` fins que no trobem el caràcter *i*:

```
let str = "curs de javaScript de l'IOC";
let cad = "";
let i = 0;
let res = "";

while (res != 'i') {
  res = str.charAt(i);
  cad += res;
  i++;
}

```

```
12 console.log(cad);
```

Una variant és el bucle `do...while`. El bloc s'executa com a mínim una vegada. La condició de sortida s'avalua al final, i es va executant el codi fins que no es compleixi la condició de sortida.

```
1 do {
2   bloc de codi
3 }
4 while (condició);
```

Per exemple,

```
1  let str = "curs de javaScript de l'IOC";
2  let cad = "";
3  let i = 0;
4  let res = "";
5
6  do {
7    res = str.charAt(i);
8    cad += res;
9    i++;
10 } while (res != 'i')
11
12 console.log(cad);
```

En aquest cas, tant se val avaluar la condició al principi o al final, el resultat és el mateix.

Podem sortir del bucle en qualsevol moment fent un *break*:

```
1  let str = "curs de javaScript de l'IOC";
2  let cad = "";
3  let i = 0;
4  let res = "";
5
6  while (res != 'i') {
7    res = str.charAt(i);
8  if (res == 'a') break;
9    cad += res;
10   i++;
11 }
12
13 console.log(cad);
```

Proveu els exemples anteriors a: codepen.io/ioc-daw-m06/pen/oNbmMLJ?editors=0012.

2. Introducció. Objectes predefinits i objectes del client web

Entre els objectes predefinits per JavaScript hem de distingir entre aquells que són propis de JavaScript com a llenguatge de programació, independentment del context; i aquells que es creen quan el client web (Firefox, Chrome, IE) carrega un document HTML, és a dir, en un context de navegador web.

Els primers representen, bàsicament, cadenes de caràcters, nombres i elements matemàtics i dates, com passa a molts llenguatges de programació. A més, també n'hi ha que representen les expressions regulars, que permeten realitzar amb relativa facilitat operacions de tractament de text amb un grau de complexitat important.

Els segons estan compresos en el BOM (*Browser Object Model*, que pot traduir-se per Model d'Objectes del Navegador). En aquest model s'inclouen objectes que representen els diferents elements del navegador com la finestra, l'historial, l'adreça web i, sobre tot, els documents HTML que s'estan mostrant. Aquests documents presenten una complexitat important i requereixen un sistema d'objectes propi, el DOM (*Document Object Model*, que pot traduir-se per Model d'Objectes del Document).

Tot això suposa llargues llistes de propietats i mètodes. Caldrà estar-hi familiaritzats amb els d'ús més habitual i, també, amb la documentació que tots aquests objectes tenen associada. És bàsic ser capaç d'interpretar-la correctament per poder utilitzar fins i tot elements que no hàgiu utilitzat mai abans.

2.1 Objectes predifinits de JavaScript

Alguns dels objectes predefinits de JavaScript més utilitzats són:

- Objecte `String`: per representar i operar amb cadenes de text.
- Objecte `Number` i `BigInt`: per representar números.
- Objecte `Math`: per representar constants i funcions matemàtiques.
- Objecte `Date`: per representar dates.
- Objecte `RegExp`: per representar expressions regulars.
- Objectes `Array`, `Map` i `Set`: per treballar amb col·leccions d'elements.

Com a objectes que són, tots tenen unes propietats i uns mètodes. És difícil practicar amb totes les propietats i mètodes dels objectes. El que és important

és saber cercar dins de la documentació i saber aplicar les propietats i mètodes d'aquests objectes.

Un enllaç directe on podeu trobar la referència de tots els objectes predefinits, les seves propietats i els seus mètodes és: mzl.la/3dkksJe.

2.1.1 L'objecte String

En l'apartat d'"Activitats" del web del mòdul trobareu un exemple de com afegir un nou mètode a un objecte `String`.

L'objecte `String` és un constructor de cadenes. De fet, no és exactament el mateix el tipus primitiu `string` que un objecte `String`.

```
let strPrimitiva = 'IOC';
let strObjecte = new String(strPrimitiva);

console.log(typeof strPrimitiva); // string
console.log(typeof strObjecte);  // object

console.log(strPrimitiva == strObjecte); //true
console.log(strPrimitiva === strObjecte); //false
```

Tot i que normalment no ens n'haurem de preocupar, podem obtenir resultats sorprenents. Per exemple, si utilitzem la funció de JavaScript `eval()`, que avalua o executa una expressió:

```
let str1 = '2 + 2';    //tipus primitiu de cadena
let str2 = new String('2 + 2'); // objecte String
console.log(eval(str1));        // retorna 4
console.log(eval(str2));        // retorna la cadena "2 + 2"
```

Amb el mètode `valueOf()` podem convertir un objecte al seu tipus primitiu:

```
console.log(eval(str2.valueOf())); //retorna 4
```

Una altra propietat interessant dels objectes `String` és `length` que ens retorna la mida d'una cadena:

```
let str1="IOC";
console.log(str1.length); // retorna 3
```

Mètodes de l'objecte String

Els principals mètodes que farem servir són els següents.

- `charAt(x)`: retorna el caràcter a la posició `x` (començant pel 0).
- `concat(str)`: concatena la cadena `str` a la cadena original.
- `startsWith(str)`: comprova si la cadena comença amb els caràcters o cadena especificats.
- `endsWith(str)`: comprova si la cadena acaba amb els caràcters o cadena especificats.

- `includes(str)`: comprova si la cadena conté els caràcters o cadena especificats.
- `indexOf(str)`: retorna la posició de la primera ocurrència de la cadena que passem com a paràmetre.
- `lastIndexOf(str)`: retorna la posició de l'última ocurrència de la cadena que passem com a paràmetre.
- `match(regexp)`: cerca dins una cadena comparant-la amb l'expressió regular `regexp`, i retorna les coincidències en un *array*.
- `repeat(x)`: retorna una nova cadena que és la repetició de la cadena tantes vegades com el valor del paràmetre `x`.
- `replace(str1, str2)`: cerca dins la cadena els caràcters (o expressió regular) `str1`, i retorna una nova cadena amb aquests valors reemplaçats pel segon paràmetre `str2`.
- `search(str)`: cerca dins de la cadena els caràcters (o expressió regular) `str`, i retorna la posició de la primera ocurrència.
- `slice(x,y)`: extreu una part d'una cadena entre els caràcters `x` i `y`, i retorna una nova cadena.
- `split(car)`: separa una cadena en un *array* de subcadenes, agafant com a llavor del separador el caràcter `car`.
- `substr(x, y)`: extreu caràcters d'una cadena, començant en la posició `x`, i el número de caràcters especificat per `y`.
- `substring(x, y)`: extreu caràcters d'una cadena entre els dos index especificats, `x` i `y`.
- `toLowerCase(str)`: converteix una cadena a minúscules.
- `toUpperCase(str)`: converteix una cadena a majúscules.
- `trim()`: elimina els espais en blanc d'ambdós extrems de la cadena.
- `toString()`: retorna el valor d'un objecte `String`.
- `valueOf()`: retorna el tipus primitiu d'un objecte `String`.

Hi ha altres mètodes que només tenen sentit en el context de la web. Són els *wrapper HTML methods*. Per exemple, el mètode `bold()` retorna la mateixa cadena embolcallat amb els tags `<b>` i `</b>`, de manera que dins d'una pàgina web la cadena es veu com a negreta. Aquests mètodes no són un estàndard, poden tenir un comportament diferent en diversos navegadors i, a més, la majoria d'ells són obsolets. Els principals mètodes d'embolcall HTML són: `big()`, `bold()`, `fontcolor()`, `fontsize()`, `italics()`, `small()`, que embolcallen la cadena amb *tags* perquè es mostri, respectivament: amb font gran, en negreta, d'un color determinat, amb una mida determinada, en cursiva i amb font petita.

Exemple: funció per validar el DNI

En el DNI tenim 8 números i una lletra. Per a cada número només una lletra és vàlida, i la manera de calcular-la la tenim en la funció següent.

S'utilitzen diversos mètodes de `String`: `substr()`, `charAt()` i `toUpperCase()`.

```
function validarDNI(dni) {
  let lletres = "TRWAGMYFPDXBNJZSQVHLCKE";
  let numDni = dni.substr(0, dni.length - 1);

  //la línia següent pot substituir-se per: var lletra_dni=dni.
      substr(dni.length-1,1).toUpperCase();
  let lletraDni = dni.charAt(dni.length - 1).toUpperCase();

  if (lletres.charAt(numDni % 23) === lletraDni) {
    return true;
  }

  return false;
}

console.log(validarDNI("38540343T")); //false
console.log(validarDNI("38540343w")); //true
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/jOWdjmy?editors=0012.

2.1.2 L'objecte Number

A JavaScript comptem amb dos tipus primitius per treballar amb nombres:

- `Number`: per a nombres enters i reals.
- `BigInit`: per treballar amb nombres enters grans.

En aquesta secció ens centrem en el tipus `Number`, que guarda els números en memòria amb una resolució de 64 bits (doble precisió) i coma flotant.

```
let a = 20;
let b = -2.718;
let c = 1e4;
let d = 2.23e-3
```

A part de la notació decimal, podem representar els números en binari, octal o hexadecimal.

```
console.log(0b1010);
console.log(0xFF);
```

I amb el mètode `toString()` podem representar els números en diferents notacions:

```
let num = 76;
console.log(`${num.toString(16)} en hexadecimal`);
console.log(`${num.toString(8)} en octal`);
console.log(`${num.toString(2)} en binari`);
```

`Infinity` és el valor per representar que hem sobrepassat la precisió dels números (que és fins a 15 dígits en la part entera):

```
console.log (5/0); //retorna Infinity
```

Not a Number (`NaN`) és una paraula reservada per indicar que el valor no representa un número. Podem utilitzar la funció global `isNaN()` per saber si l'argument és un número:

```
console.log(3 * "a"); //NaN. Recordeu que 3+"a" retorna "3a"
console.log(isNaN(3 * 4)); //false
console.log(isNaN(3 * "a")); //true
```

De la mateixa manera que passa amb les cadenes, a JavaScript normalment els números són tipus primitius, però també els podem crear com a objectes:

```
let x = 25; //tipus primitiu
let y = new Number(25); //objecte
console.log(typeof x); //number
console.log(typeof y); //object

console.log (x == y); //true
console.log (x === y); //false
```

Podeu veure els exemples anteriors a: codepen.io/ioc-daw-m06/pen/ZEQwdrv?editors=0012.

Mètodes de l'objecte Number

Els principals mètodes de l'objecte `Number` són:

- `isFinite()`: comprova si un valor és un número finit.
- `isInteger()`: comprova si un valor és un enter.
- `isNaN()`: comprova si un valor és un `NaN`. Alternativament es pot fer la comprovació `a === Number.NaN`.
- `toExponential(x)`: representa un número amb la seva notació exponencial. Per exemple, 20,31 a 2.031e+1, que significa 2.031 * 10^1.
- `toFixed(n)`: formata un número amb una precisió decimal de n dígits. Retorna una cadena que és la representació del número.
- `toPrecision(n)`: formata un número a una precisió de n dígits (número total de dígits, incloent la part entera i la decimal).
- `toString()`: converteix un número a una cadena.
- `parseInt()`: converteix un número a enter.
- `parseFloat()`: converteix un número a decimal.
- `valueOf()`: retorna el valor del tipus primitiu del número.

Posem alguns exemples:

```
console.log((18).toString() + (20).toString());

let x = 234.456;
console.log(typeof x); //number
console.log (Number.isInteger(x)); //false
console.log (x.toExponential());
x = x.toFixed(2); // 234.46, arrodoneix de la forma esperada
console.log(x);
console.log(typeof x); //compte! x ara és un string
x = parseFloat(x);
console.log(typeof x); //number. Torna a ser number

x = x.toPrecision(6); //234.460
console.log(x);
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/bGEzPKB?editors=0012.

Paràmetres i valors retornats per funcions

Aquest exemple que acabeu de veure no era trivial. Fixeu-vos que després d'utilitzar toFixed() el resultat és una cadena, i per tant no podem aplicar toPrecision(), que només s'aplica a *Numbers*. Per convertir a Number tampoc puc utilitzar Number.parseFloat(), que només s'aplica a Number, sinó la funció global `parseFloat(x)`. Un cop tenim Number, ja podem aplicar el mètode toPrecision(). Aquesta manera de raonar és habitual en JavaScript i en d'altres llenguatges de programació. Així doncs, quan us trobeu amb un resultat inesperat no heu de treure conclusions precipitades, i heu d'aprendre a trobar i raonar sobre la causa del problema.

2.1.3 L'objecte Math

L'objecte `Math` permet realitzar tasques matemàtiques, com per exemple crear números aleatoris, realitzar funcions trigonomètriques, etc.

Propietats de Math

L'objecte `Math` és especial en el sentit que no té constructor. No hi ha cap mètode per crear un objecte `Math`. Podem fer-lo servir sense haver de crear-lo.

JavaScript té emmagatzemades les principals constants matemàtiques (número pi, número e, etc.), i per accedir-hi ho fem a través de les propietats corresponents de l'objecte `Math`. Així doncs, trobem: `Math.E` (número d'Euler, 2.718...), `Math.LN2` (logaritme neperià de 2), `Math.LN10` (logaritme natural de 10), `Math.PI` (número Pi), `Math.SQRT2` (arrel quadrada de 2), i algunes altres més.

Per exemple,

```
console.log (Math.PI);
```

Mètodes de l'objecte Math

Amb els següents mètodes de `Math` podem realitzar operacions matemàtiques usuals:

- `abs(x)`: valor absolut.
- `sin(x)`, `cos(x)`, `tan(x)`: funcions trigonomètriques de sinus, cosinus i tangent.
- `asin(x)`, `acos(x)`, `atan(x)`: retorna en radians l'arcsinus, arccosinus, arctangent.
- `round(x)`: arrodoneix x al valor enter més proper.
- `ceil(x)`, `floor(x)`: retorna el mateix número però arrodonit a l'enter més proper cap a dalt (`ceil`) o cap a baix (`floor`).
- `truncx(x)`: elimina la part fraccionària d'un número (i queda només la part entera).
- `exp(x)`: retorna el valor e^x.
- `max(a,b,...)`, `min(a,b,...)`: retorna el valor més gran (o més petit) de la llista de números.
- `pow(x,y)`: retorna x^y.
- `log(x)`: retorna el logaritme natural (base e) de x.
- `random()`: retorna un número aleatori entre 0 (inclòs) i 1 (exclòs).
- `sqrt(x)`: retorna l'arrel quadrada de x.

Quan es programa, sovint és necessari obtenir números aleatoris. Per generar-los, JavaScript utilitza una llavor (*seed*) interna que no és accessible a l'usuari. En el següent exemple es veu un exemple típic de com es poden generar números aleatoris en diferents rangs:

```
1  // Funció que retorna un número aleatori entre 0 (inclusiu) i 1 (exclusiu)
2  function getRandom() {
3    return Math.random();
4  }
5
6  // Funció que retorna un número aleatori entre min (inclusiu) i max (exclusiu)
7  function getRandomArbitrary(min, max) {
8    return Math.random() * (max - min) + min;
9  }
10
11 // Funció que retorna un número enter aleatori entre min (inclusiu) i max (
       exclusiu)
12 function getRandomInt(min, max) {
13   return Math.floor(Math.random() * (max - min)) + min;
14 }
15
16 // Funció que retorna un número enter aleatori entre min (inclusiu) i max (
       inclusiu)
17 function getRandomIntInclusive(min, max) {
18   return Math.floor(Math.random() * (max - min + 1)) + min;
19 }
20
21 console.log('\nNúmero aleatori entre 0 (inclusiu) i 1 (exclusiu)');
22 for (let i=0; i<10; i++) {
23   console.log(getRandom());
24 }
25
26 console.log('\nNúmero aleatori entre 1 (inclusiu) i 5 (exclusiu)');
```

```
for (let i=0; i<10; i++) {
    console.log(getRandomArbitrary(1,5));
}

console.log('\nNúmero enter aleatori entre 1 (inclusiu) i 5 (exclusiu)');
for (let i=0; i<10; i++) {
    console.log(getRandomInt(1,5));
}

console.log('\nNúmero enter aleatori entre 1 (inclusiu) i 5 (inclusiu)');
for (let i=0; i<10; i++) {
    console.log(getRandomIntInclusive(1,5));
}
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/jrBeNW?editors=0012.

2.1.4 L'objecte Date

L'objecte `Date` s'utilitza per treballar amb dates i temps. Per instanciar un objecte tipus `Date` s'utilitza `new`, i admet diferents arguments:

```
let data = new Date(); //data del sistema
console.log(data);
data = new Date(34885453664); //genera un data que representa els mil·lisegons
    que han passat des de l'1 de gener de 1970
console.log(data);
data = new Date('2016/05/23');
console.log(data);
data = new Date(2016,5,23,12,15,24,220);  //any,mes,dia,hora,minuts,segons,mil·
    lisegons
console.log(data);
```

Treballar amb dates no sempre és fàcil i immediat: haurem de tenir en compte el format de data i el fus horari.

Sigui quin sigui el format en què expressem les dates, aquestes es guarden internament com un número que representa els mil·lisegons que han passat des de l'1 de gener de 1970 (00:00:00). Per exemple, des de la data actual (amb precisió de mil·lisegons) fins aquesta data original han passat:

```
let data = new Date(); //data del sistema
console.log(data.getTime()); //milisegons
console.log(`Des de 1970 han passat ${data.getTime()/1000/3600/24/365} anys
    aprox`);
```

Podeu veure els exemples a: codepen.io/ioc-daw-m06/pen/jOWdjXG?editors=0012.

Mètodes de l'objecte Date

Hi ha molts mètodes de l'objecte `Date`. En veurem alguns, en especial aquells que permeten representar les dates en el format usual (dd/mm/aaaa):

- `getDay()`: retorna el dia de la setmana (0-6, començant per diumenge).

- `getFullYear()`: retorna l'any (4 dígits).
- `getMonth()`: retorna el mes (0-11).
- `getDate()`: retorna el dia del mes (1-31).
- `getHours()`: retorna l'hora (0-23).
- `getMinutes()`: retorna els minuts (0-59).
- `getSeconds()`: retorna els segons (0-59).
- `getMilliseconds()`: retorna els mil·lisegons (0-999).
- `getUTCDate()`: retorna el dia del mes (1-31), d'acord amb l'horari UTC universal.
- `getUTCDay()`: retorna el dia de la setmana (0-6, començant per diumenge), d'acord amb l'horari UTC universal.
- `getUTCFullYear()`: retorna l'any (4 dígits), d'acord amb l'horari UTC universal.
- `getUTCMonth()`: retorna el mes (0-11), d'acord amb l'horari UTC universal.
- `getUTCHours()`: retorna l'hora (0-23), d'acord amb l'horari UTC universal.
- `getUTCMinutes()`: retorna els minuts (0-59), d'acord amb l'horari UTC universal.
- `getUTCSeconds()`: retorna els segons (0-59), d'acord amb l'horari UTC universal.
- `getUTCMilliseconds()`: retorna els mil·lisegons (0-999), d'acord amb l'horari UTC universal.
- `getTime()`: retorna el nombre de mil·lisegons que han transcorregut des de la data fins l'1 de gener de 1970.
- `now()`: retorna el nombre de mil·lisegons que han transcorregut des de la data del sistema fins l'1 de gener de 1970.
- `parse()`: transforma una cadena de text (representant una data), i retorna el nombre de mil·lisegons transcorreguts des de la data fins l'1 de gener de 1970.
- `setFullYear()`: especifica el dia de l'any.
- `setMonth()`: especifica el mes.
- `setDate()`: especifica el dia del mes.
- `setHours()`: especifica l'hora.
- `setMinutes()`: especifica els minuts.

- `setSeconds()`: especifica els segons.
- `setMilliseconds()`: especifica els mil·lisegons.
- `setTime()`: especifica una data a partir del nombre de mil·lisegons abans o després de l'1 de gener de 1970.
- `setUTCFullYear()`: especifica l'any, d'acord amb l'horari UTC universal.
- `setUTCMonth()`: especifica el mes, d'acord amb l'horari UTC universal.
- `setUTCDate()`: especifica el dia del mes, d'acord amb l'horari UTC universal.
- `setUTCHours()`: especifica l'hora, d'acord amb l'horari UTC universal.
- `setUTCMinutes()`: especifica els minuts, d'acord amb l'horari UTC universal.
- `setUTCSeconds()`: especifica els segons, d'acord amb l'horari UTC universal.
- `setUTCMilliseconds()`: especifica els mil·lisegons, d'acord amb l'horari UTC universal.
- `toDateString()`: converteix la part de la data en una cadena llegible.
- `toLocaleDateString()`: converteix la part de la data en una cadena llegible, utilitzant les convencions locals.
- `toISOString()`: converteix la data a cadena, utilitzant la convenció ISO.
- `toJSON()`: converteix la data a cadena amb format JSON.
- `toTimeString()`: converteix la part del *timestamp* de l'objecte `Date` a cadena.
- `toLocaleTimeString()`: converteix la part del *timestamp* de l'objecte `Date`, utilitzant les convencions locals.
- `toString()`: converteix l'objecte `Date` a cadena.
- `toLocaleString()`: converteix l'objecte `Date` a cadena, utilitzant les convencions locals.

Diferents exemples per mostrar com s'utilitzen aquests mètodes són:

```
1 let data = new Date();
2 let arrayMesos = ['gener', 'febrer', 'març', 'abril', 'maig', 'juny', 'juliol',
      'agost', 'setembre', 'octubre', 'novembre', 'desembre'];
3 var arrayDiesSetmana = ['diumenge', 'dilluns', 'dimarts', 'dimecres', 'dijous',
      'divendres', 'dissabte'];
4 console.log(`Avui és ${arrayDiesSetmana[data.getDay()]}, ${data.getDate()} de $
     {arrayMesos[data.getMonth()]} de ${data.getFullYear()}`);
```

```
//Per veure el nombre de mil·lisegons des de l'origen dels temps (1 de gener de 1970 00:00:00):
let data = new Date();
console.log(data.getTime());
// i també:
console.log(Date.now());
```

```
let d1 = Date.parse("23 March 2016"); //parse sap interpretar diferents formats de data, però en anglès.
let d2 = Date.parse("28 May 2016");
console.log(d1);
console.log(d2);
console.log(d1 > d2); //Podem comparar les dates per saber quina és la més gran
```

Per veure la diferència entre utilitzar UTC o no, hem de pensar que aquí a Catalunya estem en el fus horari UTC/GMT +1 hora, però hem de tenir en compte si estem en horari d'estiu o d'hivern. Quan estem a l'estiu s'ha de compensar el fus horari i aleshores és UTC/GMT +2 hora. Per tant, podem veure la diferència entre l'hora local i l'hora absoluta internacional:

```
let data = new Date();
console.log(data.getHours()); //per ex, 14, hora real d'un rellotge local.
console.log(data.getUTCHours()); //per ex, 12, hora UTC
```

Vegeu la diferència entre utilitzar les convencions locals o no:

```
let d = new Date();
//Dia del Treball
d.setDate(1);
d.setMonth(4); //Representa el mes de maig
d.setFullYear(2016);
console.log(d);
console.log(d.toDateString()); //Sun May 01 2016
console.log(d.toLocaleDateString()); //1/5/2016

//Com que toLocaleDateString() retorna una cadena, ja puc utilitzar les funcions de cadena:
let arrayData = d.toLocaleDateString().split('/');
console.log(arrayData[0]); //dia
console.log(arrayData[1]); //mes
console.log(arrayData[2]); //any
```

Utilitzant les convencions locals:

```
let d = new Date();
console.log(d.toLocaleString()); // 2/5/2016, 14:44:37
console.log(d.toLocaleDateString()); // 2/5/2016
console.log(d.toLocaleTimeString()); // 14:44:37
```

Podeu veure els exemples a: codepen.io/ioc-daw-m06/pen/LYGqKoK?editors=0012.

2.1.5 L'objecte RegExp

Les expressions regulars estan explicades amb deteniment en la unitat "Esdeveniments. Manejament de formularis".

Les expressions regulars són objectes de JavaScript que s'utilitzen per descriure un patró de caràcters. Aplicades a les cadenes de text, amb les expressions regulars

podem esbrinar si una cadena de text compleix un patró, i realitzar funcions de cerca i reemplaçament.

En una expressió regular distingim entre el **patró** i els **modificadors**: */pattern/-modificadors;*.

```
var patro = /IOC/i
```

En aquest cas, la *i* és un modificador que significa que farem una cerca sense tenir en compte majúscules/minúscules. Hi ha altres modificadors que pots consultar, entre d'altres llocs, al tutorial de la Fundació Mozilla mzl.la/3atdqA2.

Per veure el funcionament bàsic de les expressions regulars veurem els mètodes `test()` i `match()`:

- `test()`: mètode que retorna `true` o `false` si la cadena de text compleix el patró.
- `match()`: mètode de l'objecte `String` que fa una cerca de la cadena contra un patró, i retorna un *array* amb els resultats trobats, o el valor primitiu `null` en cas que no en trobi cap.

```
let patro1 = /ioc/i
let patro2 = /ioc/
let cad="Curs de JavaScript de l'IOC";

console.log(patro1.test(cad));
console.log(patro2.test(cad));

let res = cad.match(patro1);
console.log(res.length); //1, una sola ocurrència
console.log(res[0]); //IOC
```

Dins una expressió regular podem utilitzar *brackets* per expressar un rang de caràcters, i metacaràcters, que tenen un significat especial. Per exemple, anem a veure una expressió regular per comprovar si una cadena compleix amb el format de data dd/mm/aaaa:

```
regexp = /^(3[01]|[12][0-9]|0?[1-9])\/(1[0-2]|0?[1-9])\/([0-9]{2})?[0-9]{2}$/;
console.log(regexp.test("01/01/2016")); //true
console.log(regexp.test("2016/01/01")); //false
console.log(regexp.test("01/01/16")); //true
console.log(regexp.test("1/1/16")); //true
console.log(regexp.test("IOC")); //false
```

- Amb el circumflex (^) estem indicant com ha de començar l'expressió.
- Amb el dòlar ($) estem indicant com ha d'acabar l'expressió.
- Utilitzem la contrabarra (\) per indicar que la barra (/) no és un metacaràcter, sinó un caràcter a tenir en compte.
- El dia del mes pot ser: 30 o 31; del 10 al 29; o també del 1 al 9, ficant o no un 0 al davant.

- El mes pot ser 1-12, ficant o no un zero al davant.
- L'any pot ser de 2 o 4 dígits.

Un altre exemple, en aquest cas per validar un número de targeta de crèdit:

```
regexp = /^[0-9]{4}(-|\s)[0-9]{4}(-|\s)[0-9]{4}(-|\s)[0-9]{4}$/;
console.log(regexp.test("3782-8224-6310-0067")); //true
console.log(regexp.test("3782 8224 6310 0067")); //true
console.log(regexp.test("3782*8224*6310*0067")); //false
console.log(regexp.test("37828224630006")); //false
```

Podeu veure els exemples a: codepen.io/ioc-daw-m06/pen/XWXOLvy?editors=0012.

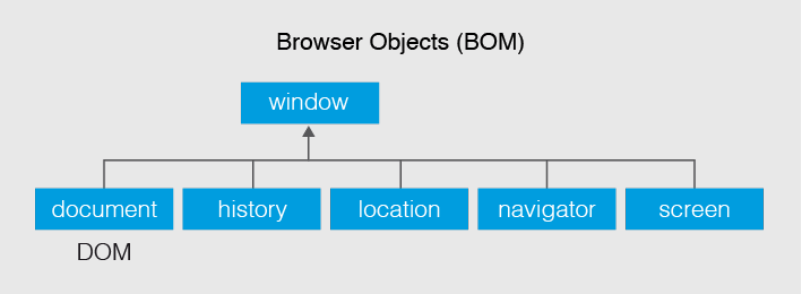
2.2 BOM (Browser Object Model)

El DOM (Document Object Model) està explicat amb més detall a la unitat "Model d'objectes del document".

L'objectiu del BOM no és només canviar el contingut del document HTML, sinó també realitzar altres manipulacions relacionades amb el navegador i les seves finestres. Per exemple, és possible redimensionar i moure una finestra del navegador, modificar la informació que es mostra a la barra d'estat, moure's per l'historial d'URLs en les quals navegador ha accedit, gestionar les *cookies* (galetes), etc.

Quan un client web (el navegador web: Firefox, Chrome, IE) carrega un document HTML, crea uns objectes associats al document, al seu contingut, i altra informació útil. Aquests objectes guarden una jerarquia en què l'objecte `window` és l'arrel, i d'aquí pengen els altres objectes que podem referenciar (vegeu la figura 2.1).

Figura 2.1. Jerarquia d'objectes del BOM



Com es veu a la figura 2.1, cada document HTML crea els següents objectes:

- `window`: representa l'arrel de l'arbre del BOM. Una finestra pot tenir subfinestres (per exemple, *pop-ups*), que seran fills en aquest arbre que representa l'estructura.
- `location`: conté les propietats de la direcció URL del document.
- `history`:conté la informació de les direccions URL que s'han visitat en la sessió actual.

- `document`: conté informació sobre el document actual, com ara el títol, imatges, *links*, taules, formularis que conté, etc. Per accedir a tota aquesta informació tenim el DOM. Les propietats de l'objecte `document` poden ser alhora altres objectes (com ara l'objecte `cookies`).

A més de propietats i mètodes, també hi tenim *events* associats. És a dir, aquests objectes poden respondre a *events* disparats per les accions de l'usuari.

2.2.1 L'objecte Window. Jerarquia d'objectes associats al navegador

En l'arrel de la jerarquia trobem l'objecte `window`, que representa una finestra oberta en el navegador. En la figura figura 2.1 es pot veure la jerarquia d'objectes que pengen de `window`.

Com que una imatge forma part del contingut del document HTML més que no pas de les propietats del navegador, s'estudia amb més profunditat en la unitat "Model d'objectes del document".

En aquesta jerarquia, els descendents d'un objecte es representen mitjançant propietats de l'objecte. Per exemple, una imatge `img1` és un objecte però també és una propietat de l'objecte `document`, i serà referenciat com a `document.img1`. Per referenciar una propietat s'han de precisar els seus ascendents. De fet, seria `window.document.img1`, però `window`, que és l'arrel, es pot ometre. Alhora aquesta imatge, com a objecte que és, té les seves propietats (típicament `width` i `heigh`): `document.img1.width`.

Un altre exemple: utilitzant la jerarquia podem saber la URL actual de la pàgina que estem visitant: `window.location.href` (o `location.href`).

El codi HTML per poder provar els anteriors exemples és:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Objecte window</title>
    </head>
    <body>
        <img src="./muntanya.jpg" name="img1" />
        <script>
            console.log(window.document.img1.width);
            console.log(document.img1.height);
            console.log(window.location.href);
            console.log(location.protocol);
        </script>
    </body>
</html>
```

Evidentment, l'usuari haurà de tenim una imatge *muntanya.jpg* en la mateixa carpeta que el document HTML.

Propietats de window

Les propietats de `window` són:

- `defaultStatus`: és el missatge que es visualitza en la barra d'estat del navegador.
- `frames`: és la matriu de la col·lecció de *frames* que conté una finestra. Per exemple, si volem fer referència al primer *frame* de la finestra: `window.frames[0]`. `frames.length` és el número de *frames* que conté una finestra.
- `parent`: ascendent d'una finestra.
- `self`: fa referència a la finestra actual.
- `status`: missatge que mostra l'estat del client web.
- `top`: fa referència a l'arrel de la jerarquia d'objectes.
- `window`: fa referència a la finestra actual.
- `innerWidth`, `innerHeight`: amplada i altura interiors de la finestra.
- `closed`: retorna un *boolean* indicant si la finestra s'ha tancat o no.
- `screenX`, `screenY`: coordenades de la finestra en relació a la pantalla. Vàlid per als navegadors Google Chrome i IE (per a Firefox utilitzar `screenLeft` i `screenTop`).
- `opener`: retorna una referència a la finestra que ha creat la finestra.
- `localStorage`: retorna una referència a l'objecte `localStorage` que s'utilitza per emmagatzemar dades. Al contrari que les galetes, no té data d'expiració.
- `document`, `history`, `location`, `navigator`, `screen`: són propietats de `window` que fan referència als objectes `document`, `history`, `location`, `navigator`, `screen`. Recordeu que formen part d'una estructura jeràrquica.
- `sessionStorage`: retorna un objecte `storage` per emmagatzemar dades en una sessió web.

Les propietats més clares les podem veure en aquest exemple:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Objecte window</title>
    </head>
    <body>
        <script>
            console.log(window.innerWidth + '-' + window.innerHeight); //Pots
                redimensionar la finestra per veure com canvia aquest valor
            console.log(window.screenX + '-' + window.screenY); //Si mous la
                finestra de posició per veure com canvia aquest valor
            window.defaultStatus = "informació a la barra d'estat";
            console.log(window.defaultStatus);
            console.log(window.closed);
        </script>
    </body>
</html>
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/JKWmdv.

Mètodes de window

Els principals mètodes de l'objecte `window` són:

- `alert(missatge)`: crea una finestra de diàleg on es mostra el missatge.
- `confirm(missatge)`: crea una finestra de confirmació (OK/Cancel) on es mostra el missatge. Retorna `true` o `false`.
- `prompt(missatge, text)`: crea una finestra de diàleg on es mostra el missatge i permet l'edició. El paràmetre `text` és el valor predeterminat. Igual que `confirm()`, conté Acceptar i Cancelar. Si acceptem, el mètode retorna el valor inserit (o el valor per defecte). Si cancelem, retorna `null`.
- `close()`: només es poden tancar les finestres que també s'han creat amb un script.
- `stop()`: atura la càrrega de la finestra.
- `setTimeout(expressió, msec)`: executa l'expressió que es passa com a argument, un cop han passat `msec` mil·lisegons. S'utilitza per prorrogar l'execució de l'expressió.
- `clearTimeout(timeoutID)`: restaura el compte enrere iniciat per `setTimeout()`.
- `setInterval(expressió, msec)`: executa l'expressió passada com a argument cada `msec` mil·lisegons. S'utilitza per executar l'expressió a intervals regulars de temps.
- `open(url, windowName, params)`: crea una finestra nova, li associa el nom `windowName`, i accedeix a la URL que li hem passat. Li passem un conjunt de paràmetres que descriuen les propietats de la *finestra*. Entre d'altres: *toolbar*, *width*, *height*, *left*, *top*, *fullscreen*, *resizable*, *location*, *status*, *scrollbars*, *menubar*.

Cal tenir en compte que el nom de la finestra no és el títol i, per consegüent, no es mostra a la nova finestra.

En el següent exemple utilitzem alguns d'aquests mètodes:

```
1  <html>
2
3  <head>
4    <meta http-equiv="content-type" content="text/html; charset=utf-8">
5    <title>Creació ´duna finestra</title>
6  </head>
7
8  <body>
9    <script>
10     let finestra = window.open("http://ioc.xtec.cat", "", "width=300,height
           =300, left=300, top=300, resizable=1");
11     setTimeout(function() {
12       finestra.close();
13     }, 2000);
14
15     setTimeout(function() {
```

```
      let nom = prompt("Posa el nom de la finestra:", "Nom");
      crearFinestra(nom);
    }, 2000);

    function crearFinestra(nom) {
      let opcions = "toolbar=0, location=0, directories=0, status=0,";
      opcions += "menubar=0, scrollbars=0, resizable=0, copyhistory=0,";
      opcions += "width=100,height=100";
      window.open("", nom, opcions);
    }
  </script>
</body>

</html>
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/ezvPgG?editors=1002. Us recomanem, però, que proveu l'exercici amb un fitxer propi, ja que és posible que el *prompt* i les noves finestres siguin bloquejades.

2.2.2 L'objecte document

Quan un document HTML es carrega en un navegador web, obtenim un objecte `document`. L'objecte `document` és el node arrel d'un document HTML, i d'ell pengen tots els altres nodes: nodes d'elements, de text, d'atributs, de comentaris.

Recordeu que el `document` és part de l'objecte `window` i, per tant, s'hi pot accedir amb `window.document`.

Propietats de document

Les propietats de `document` són:

La programació amb *cookies* es detalla a l'apartat "Programació amb galetes, finestres i marcs" d'aquesta unitat.

- `location`: conté la URL del document.
- `title`: títol del document HTML.
- `lastModified`: data de l'última modificació.
- `cookie`: cadena de caràcters que conté la *cookie* associada al recurs representat per l'objecte.
- Les propietats `anchors`, `forms`, `images`, `links` retornen un objecte amb el qual podem accedir a cadascun dels elements (àncores, formularis, imatges, enllaços) presents en el document.

Les propietats `anchors`, `forms`, `images` i `links` es veuran amb més detall a la unitat "Model d'objectes del document".

El següent exemple utilitza algunes d'aquestes propietats:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Objecte document. Propietats</title>
    </head>
```

```
    <body>
        <a href="http://ioc.xtec.cat">IOC</a><br />
        <img src="http://ioc.xtec.cat/educacio/templates/ioc/images/ioc_logo.
            png" /img>
        <script>
            console.log(‘location: ${document.location}‘);
            console.log(‘títol: ${document.title}‘);
            console.log(‘lastModified: ${document.lastModified}‘);
            console.log(document.links[0].href);
            console.log(document.links.item(0).href);
            console.log(document.images[0].src);
        </script>
    </body>
</html>
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/yLeZmRw?editors=1011.

Mètodes de document

Els mètodes més importants de l'objecte `document` són:

- `write()`: escriu codi HTML en el document.
- `writeln()`: idèntic a `write()`, però inserta un retorn de carro al final.
- `open(tipus MIME)`: obre un *buffer* per recollir el que retorna els mètodes `write()` i `writeln()`. Els tipus MIME que es poden utilitzar són: `text/html`, `text/plain`, `text/jpeg`, etc.
- `close()`: tanca el *buffer*.
- `clear()`:esborra el contingut del document.

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Objecte document. Mètodes</title>
        <script type="text/javascript">
          function ReemplacarContingut () {
            document.open ("text/html");
            document.write ("<a href=\"http://web.gencat.cat/ca/inici/\">gencat
                .cat</a><br />");
            document.write ("<img src=\"http://web.gencat.cat/web/.content/
                Imatge/marca_home/NG_logo_generalitat_home.png_602392567.png\"
                 /img>");
            document.close ();
          }
        </script>
    </head>
    <body>
        <a href="http://ioc.xtec.cat">IOC</a><br />
        <img src="http://ioc.xtec.cat/educacio/templates/ioc/images/ioc_logo.
            png" /img><br />
        <button onclick="ReemplacarContingut();">Reemplaçar el contingut del
            document</button>
    </body>
</html>
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/pbeYEK?editors=1000.

2.2.3 L'objecte location

L'objecte `location` conté informació sobre l'actual URL, conté les propietats de la direcció URL del document.

L'objecte `location` és part de l'objecte `window` i, per tant, s'hi pot accedir a través de la propietat `window.location`.

Propietats de Location

Les propietats de l'objecte `location` són:

- `hash`: retorna la part de la URL que representa l'àncora (#). Per ex, *url#IOC*.
- `host`: retorna el *hostname* i el número de port de la URL.
- `hostname`: retorna el *hostname* de la URL.
- `href`: retorna la URL sencera.
- `origin`: retorna el protocol, *hostname* i port de la URL.
- `pathname`: retorna el *path* de la URL.
- `port`: retorna el número de port de la URL.
- `protocol`: retorna el protocol de la URL.
- `search`: retorna la part de *querystring* de la URL. Per ex, *url?search=IOC*.

Mètodes de location

Els mètodes més importants de l'objecte `location` són:

- `assign()`: carrega un nou document.
- `reload()`: recarrega el document actual.
- `replace()`: reemplaça el document actual per un de nou.

Un exemple per veure la major part d'aquestes propietats i mètodes:

Si proveu l'anterior exemple com a fitxer HTML obtindreu `location.protocol: file`. Si el proveu com a pàgina web que ens serveix un servidor web obtindreu `location.protocol: http`.

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Objecte location. Propietats i Mètodes</title>
        <script type="text/javascript">
          function assignUrl () {
            document.location.assign('http://www.gencat.cat');
          }
```

```
10          function reloadUrl () {
11            document.location.reload();
12          }
13          function replaceUrl () {
14            document.location.replace('http://www.gencat.cat'); // fixeu-vos
                  que no tenim l'històric de tirar enrere
15          }
16          console.log(`location.hash: ${location.hash}`);
17          console.log(`location.hostname: ${location.hostname}`);
18          console.log(`location.href: ${location.href}`);
19          console.log(`location.origin: ${location.origin}`);
20
21          console.log(`location.pathname: ${location.pathname}`);
22          console.log(`location.port: ${location.port}`);
23          console.log(`location.protocol: ${location.protocol}`);
24          console.log(`location.search: ${location.search}`);
25        </script>
26    </head>
27    <body>
28        <a href="http://ioc.xtec.cat">IOC</a><br />
29        <img src="http://ioc.xtec.cat/educacio/templates/ioc/images/ioc_logo.
              png" /img><br />
30        <button onclick="assignUrl();">Mètode location.assign</button>
31        <button onclick="reloadUrl();">Mètode location.reload</button>
32        <button onclick="replaceUrl();">Mètode location.replace</button>
33    </body>
34  </html>
```

Podeu veure aquest exemple en el següent enllaç, però tingueu en compte que a la web de Codepen els botons no funcionaran: codepen.io/ioc-daw-m06/pen/ZEQwgaP?editors=1111.

2.2.4 L'objecte history

L'objecte `history` conté les URLs visitades per l'usuari (en el marc d'una finestra del navegador). L'objecte `history` és part de l'objecte `window`, i s'hi pot accedir a través de la propietat `window.history`.

Propietats d'history

La propietat més important de l'objecte `history` és:

- `length`: retorna el nombre d'URL a la llista de l'històric de navegació.

Mètodes de history

Els mètodes més importants de l'objecte `history` són:

- `back()`: carrega la URL prèvia en la llista de l'històric de navegació.
- `forward()`: carrega la següent URL en la llista de l'històric de navegació.
- `go()`: carrega una URL específica en la llista de l'històric de navegació.

Com a exemple, i sempre dins de la mateixa finestra, podeu navegar per les següents URLs, i finalment introduir la URL del codi que es proposa.

- ca.wikipedia.org/wiki/Manresa
- ca.wikipedia.org/wiki/Balsareny
- ca.wikipedia.org/wiki/Navàs
- ca.wikipedia.org/wiki/Puig-reig
- ca.wikipedia.org/wiki/Gironella
- ca.wikipedia.org/wiki/Berga

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Objecte history. Propietats i Mètodes</title>
    </head>
    <body>
        <button onclick="console.log('Nombre elements: ${history.length}');">
            Nombre d'elements a la llista</button>
        <button onclick="history.back()">URL anterior</button>
    </body>
</html>
```

2.2.5 L'objecte navigator

L'objecte `navigator` conté informació sobre el navegador web que s'està utilitzant.

Propietats de navigator

Les propietats de `navigator` són:

- `appCodeName`: retorna el codi del navegador, per exemple, Mozilla.
- `appName`: retorna el nom oficial del navegador. Per exemple, en el cas de Mozilla Firefox, retorna *netscape*.
- `appVersion`: la versió del navegador (potser no coincideix amb la versió real del navegador).
- `cookieEnabled`: retorna `true` si les galetes estan habilitades.
- `language`: retorna l'idioma del navegador.
- `onLine`: retorna `true` si el navegador està en línia.
- `platform`: retorna en quina plataforma es va compilar el navegador.

- `product`: retorna el nom del motor del navegador.
- `userAgent`: retorna tota la capçalera *user-agent* que el navegador envia al servidor en el protocol HTTP. Aquest valor l'utilitza el servidor per identificar el client. En aquesta capçalera sí que podem veure la versió real del nostre navegador (per exemple, Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:22.0) Gecko/20100101 Firefox/22.0).

En el següent exemple s'utilitzen les principals propietats de l'objecte `navigator`:

```
console.log(‘appCodeName: ${navigator.appCodeName}‘);
console.log(‘appName: ${navigator.appName}‘);
console.log(‘appVersion: ${navigator.appVersion}‘);
console.log(‘cookieEnabled: ${navigator.cookieEnabled}‘);
console.log(‘language: ${navigator.language}‘);
console.log(‘onLine: ${navigator.onLine}‘);
console.log(‘platform: ${navigator.platform}‘);
console.log(‘product: ${navigator.product}‘);
console.log(‘userAgent: ${navigator.userAgent}‘);
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/wWJOJr?editors=1012.

2.2.6 L'objecte screen

L'objecte `screen` conté informació sobre la pantalla on està obert el navegador del client. Aquesta informació s'extreu a partir de les propietats descrites més avall. En canvi, no té mètodes. No s'ha de confondre l'objecte `screen` amb l'objecte `window`, que representa la finestra del navegador.

Propietats d'Screen

Als dissenyadors web sempre els ha preocupat saber quina és la resolució del navegador del client per tal que les pàgines web es vegin bé en els diferents formats de pantalla. Avui en dia, gràcies a *frameworks* com *bootstrap*, l'adaptació de la web a diferents formats de pantalla és molt més fàcil.

Les propietats d'`screen` són:

- `availHeight`: retorna l'altura de la pantalla.
- `height`: retorna l'altura total de la pantalla. La diferència amb `availHeight` és l'altura de la barra de tasques (depèn del sistema operatiu).
- `availWidth`: retorna l'amplada de la pantalla.
- `width`: retorna l'amplada total de la pantalla.
- `colorDepth`: retorna la resolució de la paleta de colors (en número de bits). Per exemple: 8 significa 256 colors; 24 significa 16 milions de colors.
- `pixelDepth`:retorna la resolució de color (en bits per píxel) de la pantalla.

En el següent exemple s'utilitzen les principals propietats de l'objecte `screen` que acabem de veure:

```
console.log(‘availHeight: ${screen.availHeight}‘);
console.log(‘height: ${screen.height}‘);
console.log(‘availWidth: ${screen.availWidth}‘);
console.log(‘width: ${screen.width}‘);
console.log(‘colorDepth: ${screen.colorDepth}‘);
console.log(‘pixelDepth: ${screen.pixelDepth}‘);
```

Podeu veure l'exemple a: codepen.io/ioc-daw-m06/pen/NWxoQzz?editors=0012.

3. Programació amb galetes, finestres i marcs

Les galetes (*cookies*) són dades, emmagatzemades en la màquina client dins de fitxers de text (o base de dades SQLite, depèn de la versió del navegador).

Quan un servidor web serveix una pàgina a un client, la connexió finalitza, i el servidor no recorda res sobre l'usuari. Les galetes es van implementar per tal que el servidor recordi informació del client. Així podem guardar informació de forma persistent entre clients.

Des del cantó del servidor, per exemple programant amb PHP, també podeu accedir a les galetes del client. En PHP existeix per exemple la funció `setcookie()`, o les variables `$_COOKIE`. Això és així perquè les galetes formen part de la capçalera HTTP, és a dir, en el procés de comunicació s'envia la informació de les galetes.

Les galetes es guarden en parelles nom-valor com ara:

```
1 lang = cat
```

Quan el navegador demana una pàgina web al servidor, les galetes que pertanyen a aquesta pàgina s'adjunten a la capçalera de la petició IP. I d'aquesta manera en el cantó del servidor es pot recordar la sessió a què pertany la petició.

Les finestres obertes al navegador són accessibles a través de l'objecte `window`, que la representa.

Un marc (*frame*) és una àrea de la pantalla que té un contingut independent de la resta de la pàgina. Normalment s'especifiquen amb el tag `<iframe>`, que especifica un *inline frame* i s'utilitza per inserir un document HTML dins un altre document HTML.

3.1 Programació amb galetes ('cookies')

Quan un usuari visita una pàgina web, hi ha determinada informació que es pot guardar entre sessions. Això es fa amb les galetes (*cookies*). Per exemple, podeu recordar si preferiu que l'aplicatiu web es mostri en català, castellà o anglès. La informació s'emmagatzema en les galetes, en l'ordinador del client. L'usuari pot activar o desactivar les galetes en les preferències del seu navegador web.

Política de galetes

Recordeu que si un lloc web fa ús de galetes, cal demanar confirmació a l'usuari i afegir un enllaç a la política de galetes. Podeu trobar la *Guia sobre l'ús de galetes* al següent enllaç: bit.ly/2EcoTq3.

L'ús de les galetes s'ha considerat tradicionalment un forat de seguretat potencial. Tanmateix, avui dia és habitual tenir activades les galetes, de manera que els programadors web dóna per fet que en el cantó del client les galetes estaran habilitades.

Recordeu que tant a Mozilla Firefox com a Chrome teniu les eines de desenvolupador. No només teniu una consola per llegir, sinó que també teniu en ella un espai per escriure. Aquest és el lloc correcte per fer proves. Per exemple: `document.cookie = "idioma = castella";`, per gravar una galeta, o bé `document.cookie` per veure les galetes.

Les galetes són parelles nom-valor, com ara:

```
1 idioma=catala
```

Amb JavaScript podem crear, modificar i esborrar les galetes mitjançant la propietat `cookie`: `document.cookie`:

```
document.cookie = "idioma = catala";
document.cookie = "idioma = catala; expires=31 Dec 2030";
document.cookie = "colorFons = red";
console.log(document.cookie);
```

Llegint `document.cookie` obtenim una cadena similar a:

```
colorFons=red; idioma=catala;
```

Fixem-vos que el format sempre és punt i coma seguit d'un espai en blanc. A més a més, si voleu que la galeta es mantingui un cop tancada la sessió, cal indicar la data d'expiració. En el nostre cas: `expires=31 DEC 2030`.

Les galetes es poden llegir amb `document.cookie`, que retorna una cadena amb totes les galetes que estan definides. Per canviar el valor d'una galeta, només cal cridar-la igual que s'ha creat:

```
document.cookie = "idioma = castella";
document.cookie ="colorFons = blue";
console.log(document.cookie);
document.cookie ="idioma = angles; colorFons = green";
console.log(document.cookie);
```

Per esborrar una galeta n'hi ha prou amb posar la data d'expiració a una data passada:

```
document.cookie = "idioma=; expires=01 Jan 2000 00:00:00 UTC";
console.log(document.cookie);
```

Podem veure com la galeta `idioma` ha desaparegut.

Ja que la propietat `document.cookie` ens retorna una cadena amb les diferents galetes separades per punt i coma - espai, podem llistar-les una per una si fem un *split* de la cadena amb el *;* com a caràcter separador:

```
let arrayCookies = document.cookie.split('; ');
for(let cookie of arrayCookies) {
  console.log(cookie);
}
```

Per distingir entre el nom de la galeta i el seu valor, hem de veure que el signe = és el separador. Podem tornar a fer servir `split()`:

```
let arrayCookies = document.cookie.split('; ');
for(let cookie of arrayCookies) {
  console.log(cookie);
  let temp = cookie.split('=');
  let nomCookie = temp[0];
  let valorCookie = temp[1];
  console.log(`Nom de la galeta: ${nomCookie}; valor de la cookie: ${
      valorCookie}`);
}
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/jOWdRLp?editors=0012.

Una mostra d'una petita aplicació escrivint i llegint els valors de les galetes `idioma` i `colorFons` és la següent:

```
<!DOCTYPE html>
<html>

<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>Programació amb galetes</title>
  <script>
    function carregarCookies() {
      let arrayCookies = document.cookie.split("; ");
      let nomCookie;
      let valorCookie;
      for (let cookie of arrayCookies) {
        let temp = cookie.split("=");
        nomCookie = temp[0];
        valorCookie = temp[1];
        console.log('Nom de la cookie: ${nomCookie}; valor de la cookie: ${
            valorCookie}');
      };
      let h1 = document.getElementById("header");
      if (nomCookie === "idioma" && valorCookie === "catala") {
        h1.innerHTML = "Text en català";
      } else if (nomCookie === "idioma" && valorCookie === "castella") {
        h1.innerHTML = "Texto en castellano";
      } else if (nomCookie === "idioma" && valorCookie === "angles") {
        h1.innerHTML = "Text in English";
      }
      if (nomCookie === "colorFons") {
        document.body.style.backgroundColor = valorCookie;
      }
    }
  </script>
</head>

<body onload="carregarCookies()">
  <h1 id="header">Text en català</h1>
  Per veure els canvis, actualitzar la pàgina.<br />
  <button onclick="document.cookie = 'idioma = catala';">Català</button>
  <button onclick="document.cookie = 'idioma = castella';">Castellà</button>
  <button onclick="document.cookie = 'idioma = angles';">Anglès</button><br />
  <button onclick="document.cookie = 'colorFons = red';">Vermell</button>
  <button onclick="document.cookie = 'colorFons = blue';">Blau</button>
  <button onclick="document.cookie = 'colorFons = green';">Verd</button>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/bGEzJoL?editors=1011.

On es guarden les galetes?

Les galetes es guarden de forma permanent en el sistema. Tant si reiniciem el navegador com si reiniciem l'ordinador, recuperem el seu valor. Per tant, s'han de guardar en fitxers. La manera com es guarden depèn tant del navegador (Mozilla, Chrome, Internet Explorer) com de la versió del navegador.

Era usual que les galetes es guardessin en un fitxer cookies.txt. És també usual que es guardin en un fitxer de base de dades del sistema SQLite. En qualsevol cas, també es pot accedir a les galetes i esborrar-les des de les preferències del navegador.

3.2 Emmagatzemar informació amb Local Storage

Les galetes s'inclouen en cada *request* HTTP, encara que no les necessitem, i no estan encriptades (a no ser que utilitzem SSL en el servidor). Això representa una disminució de la velocitat de transmissió. A més, les galetes estan limitades a uns 4 KB d'informació.

Seria bo que en els clients web disposéssim d'un espai d'emmagatzematge que fos persistent, i que no es transmetés al servidor.

Abans de l'HTML5 això no era possible. Però ara, amb HTML5 i els navegadors que el tenen suportat, això ja és possible gràcies al *DOM Storage* en general i al *Local Storage* en particular.

L'emmagatzematge DOM (*DOM Storage*) (també emmagatzematge WEB, *WEB Storage*) és el nom donat a un conjunt de característiques relacionades amb l'emmagatzematge introduïdes a HTML5 i ara detallades en l'especificació *W3C Web Storage*. L'emmagatzematge DOM està dissenyat per facilitar una forma ampla, segura i simple per emmagatzemar informació alternativa a les galetes. A més, proporciona la possibilitat de guardar les dades durant un llarg període de temps sense necessitat d'estar connectat.

Dins del *DOM Storage* ens interessa principalment l'objecte `localStorage` per accedir a l'emmagatzematge persistent, i l'objecte `sessionStorage` per guardar un espai d'emmagatzematge disponible durant la sessió de navegació:

- `localStorage`: guarda informació que quedarà emmagatzemada un temps indefinit, sense importar que el navegador es tanqui.
- `sessionStorage`: emmagatzema les dades d'una sessió, que s'eliminen quan es tanca.

Les característiques de `localStorage` i `sessionStorage` són:

- Permeten emmagatzemar entre 5MB i 10MB d'informació, incloent text i multimèdia.
- La informació es guarda en local, i a diferència de les galetes, no s'envia al servidor en cada petició que es fa al servidor.
- Utilitzen un número mínim de peticions al servidor i el tràfic queda molt reduït.
- Quan treballem sense connexió o ens quedem sense connexió, es garanteix l'emmagatzematge.
- La informació es guarda a nivell de domini web (inclou totes les pàgines del domini).

Exemple de 'localStorage'

En el cas de `localStorage` utilitzem els mètodes `setItem()` i `getItem()` per escriure i llegir els parells nom-valor per emmagatzemar la informació.

```
localStorage.setItem("animal", "gos");
console.log(localStorage.getItem("animal"));
localStorage.removeItem("animal");
```

Fem l'exemple més complet amb un formulari:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html;
            charset=utf-8">
        <title>Exemple localStorage</title>
        <script>
            function guardar() {
              let animal = document.getElementById("nomanimal").
                  value;
              let vegetal = document.getElementById("nomvegetal")
                  .value;
              /*Guardem les dades en el localStorage*/
              localStorage.setItem("animal", animal);
              localStorage.setItem("vegetal", vegetal);
              /* netegem el camp */
              document.getElementById("nomanimal").value = "";
              document.getElementById("nomvegetal").value = "";
            };

            function carregar() {
              let animal = localStorage.getItem("animal");
              let vegetal = localStorage.getItem("vegetal");
              document.getElementById("nomanimal").value = animal
                  ;
              document.getElementById("nomvegetal").value =
                  vegetal;
            }
        </script>
    </head>
    <center>
        <h1>Exemple localStorage</h1>
        <input type="text"  placeholder="nom animal" id="
            nomanimal"><br />
        <input type="text"  placeholder="nom vegetal" id="
            nomvegetal"><br />

        <button onclick="guardar()">Guardar</button>
        <button onclick="carregar()">Carregar</button>
    </body>
</html>
```

Com a prova, podeu tancar el navegador i tornar-lo a obrir, i quan cliqueu *Carregar* veureu que es recuperen les dades emmagatzemades.

Podeu provar aquest exemple a: codepen.io/ioc-daw-m06/pen/rNxPbRN?editors=1010.

Exemple de 'sessionStorage'

Amb `sessionStorage` la informació es guarda en les diferents pàgines de la mateixa sessió. Utilitzem els mètodes `setItem()` i `getItem()` per escriure i llegir els parells nom-valor per emmagatzemar la informació.

```
sessionStorage.setItem("animal", "gos");
console.log(sessionStorage.getItem("animal"));
sessionStorage.removeItem("animal");
```

Les dades amb `sessionStorage` són accessibles mentre dura la sessió de navegació. Hem de tenir en compte, i això és important de cara la realització de l'exercici, que les dades no són recuperables si:

- Es tanca el navegador i es torna a obrir.
- S'obre una pestanya de navegació independent i se segueix navegant en aquesta pestanya.
- Es tanca la finestra de navegació i se n'obre una altra.

Fem l'exemple més complet. Necessitem els scripts *animal.html* i *vegetal.html*. Des dels dos scripts podem veure totes les variables de sessió, però per veure-les recomanem utilitzar els enllaços, i no obrir una nova pestanya.

animal.html:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Exemple sessionStorage. Animal</title>
        <script>
            function guardar() {
              let animal = document.getElementById("nomanimal").value;
              /*Guardem les dades en el sessionStorage*/
              sessionStorage.setItem("animal", animal);
              /* Netegem els camps */
              document.getElementById("nomanimal").value = "";
              document.getElementById("lblanimal").value = "";
              document.getElementById("lblvegetal").value = "";
            };

            function carregar() {
              let animal = sessionStorage.getItem("animal");
              let vegetal = sessionStorage.getItem("vegetal");
              document.getElementById("lblanimal").innerHTML = animal;
              document.getElementById("lblvegetal").innerHTML = vegetal;
            }
        </script>
    </head>
    <center>
        <h1>Exemple sessionStorage</h1>
        <input type="text"  placeholder="nom animal" id="nomanimal"><br />
        <button onclick="guardar()">Guardar</button>
        <button onclick="carregar()">Carregar</button><br />
        Animal: <label id="lblanimal"></label><br />
        Vegetal: <label id="lblvegetal"></label><br />
        <a href="vegetal.html">vegetal.html</a>
    </body>
</html>
```

vegetal.html:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Exemple sessionStorage. Vegetal</title>
        <script>
            function guardar() {
```

```
            let vegetal = document.getElementById("nomvegetal")
                .value;
            /*Guardem les dades en el sessionStorage*/
            sessionStorage.setItem("vegetal", vegetal);
            /* netegem els camps */
            document.getElementById("nomvegetal").value = "";
            document.getElementById("lblvegetal").value = "";
            document.getElementById("lblvegetal").value = "";
          };

          function carregar() {
            let animal = sessionStorage.getItem("animal");
            let vegetal = sessionStorage.getItem("vegetal");
            document.getElementById("lblanimal").innerHTML =
                animal;
            document.getElementById("lblvegetal").innerHTML =
                vegetal;
          }
        </script>
    </head>
    <center>
        <h1>Exemple sessionStorage</h1>
        <input type="text"  placeholder="nom vegetal" id="
            nomvegetal"><br />
        <button onclick="guardar()">Guardar</button>
        <button onclick="carregar()">Carregar</button><br />
        Animal: <label id="lblanimal"></label><br />
        Vegetal: <label id="lblvegetal"></label><br />
        <a href="animal.html">animal.html</a>
    </body>
</html>
```

Podeu provar el darrer exemple a: codepen.io/ioc-daw-m06/pen/xxZMeeV?editors=1010 i a codepen.io/ioc-daw-m06/pen/MWKLRRM?editors=1010.

3.3 Comunicació entre finestres

L'objecte `window` representa una finestra oberta en un navegador. Quan des d'una finestra s'obre una altra finestra, el navegador web "coneix" aquesta relació de dependència, i es pot parlar de finestra *pare* i finestra *filla*. En el següent exemple mostrem com es comunica una finestra amb la finestra filla que crea; i a l'hora la finestra filla també es comunica amb el seu pare. Aquestes tècniques només es poden utilitzar entre finestres que tenen relació de pare i fills. No ens podem comunicar amb finestres independents.

En el mètode `window.open()`, si no posem el tercer argument, on definim els paràmetres `top`, `left`, `width` i `height`, la finestra s'obre en una nova pestanya del navegador. Aleshores es podria parlar de comunicació entre pestanyes del navegador.

script *finestraPare.html*:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Comunicació entre finestres</title>
        <script>
            let i = 0;
            function crearFinestraFilla() {
                let finestraFilla = window.open("finestraFilla.html", "
                    Comptador", "top=200,left=200,width=200,height=200");
                setInterval(function(){ temporitzador(finestraFilla) }, 1000);
            }
```

```
12
13              function temporitzador(finestraFilla) {
14                  i++;
15                  let comptador_fill = finestraFilla.document.getElementById("
                        comptador");
16                  comptadorFill.innerHTML= i;
17              }
18          </script>
19      </head>
20      <body>
21          <button onclick="crearFinestraFilla();">Crear la finestra filla</button
                >
22      </body>
23  </html>
```

script *finestraFilla.html*:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta http-equiv="content-type" content="text/html; charset=utf-8">
5           <title>Comunicació entre finestres</title>
6           <script>
7               function randomColorInParent() {
8                 let color = 'rgb(' + (Math.floor(Math.random() * 256)) + ',' + (
                      Math.floor(Math.random() * 256)) + ',' + (Math.floor(Math.
                      random() * 256)) + ')';
9                 //finestra del pare:
10                opener.document.body.style.backgroundColor = color;
11              }
12          </script>
13      </head>
14      <body>
15          <button onclick="randomColorInParent();">Canviar el color de la
                finestra pare</button><br />
16          Comptador processat pel pare:
17          <p id="comptador"></p>
18      </body>
19  </html>
```

`window.open()` existeix des dels primers inicis de JavaScript i el web. Però el temps ha passat, i amb HTML5 i les versions actuals dels navegadors han aparegut noves tècniques que permeten establir comunicació entre finestres independents, com ara `localStorage`.

Trobareu un exemple concret amb `localStorage` a la secció "Activitats" del web del mòdul.

Gràcies a l'objecte `localStorage` es pot guardar informació en el cantó del client, i es pot establir un canal de comunicació entre finestres independents. Quan una finestra vol enviar un missatge a una altra finestra, n'hi ha prou a emmagatzemar el missatge i d'alguna manera s'ha d'aixecar un *event* per tal que la finestra receptora sàpiga que ha de llegir el missatge.

3.4 Marcs (frames/iframes)

El tag `<iframe>` especifica un *inline frame*, que s'utilitza per inserir un document HTML dins un altre document HTML. Per exemple:

```
1   <!DOCTYPE html>
```

```
<html>
    <body>
        <iframe src="http://ioc.xtec.cat/educacio/"></iframe>
    </body>
</html>
```

Segurament l'exemple anterior no té un disseny adequat als estàndards actuals de disseny web. Per tal que l'`<iframe>` tingui l'aspecte desitjat, tenim una col·lecció d'atributs, així com la possibilitat d'aplicar estils CSS. Hem de tenir present que hi ha diferències substancials entre els atributs disponibles a HTML5, i els que teníem a HTML4. Els més interessants suportats per HTML5 són:

- `width`, `height`: amplada i altura dels `<iframe>`. Aquests atributs s'han conservat a HTML5, però no hi ha cap motiu per no utilitzar CSS per especificar l'amplada i altura.
- `name`: el nom del `<iframe>`.
- `src`: URL del document que incrustarem a l'`<iframe>`.
- `srcdoc`: el contingut HTML que incrustarem a l'`<iframe>`.

Les possibilitats que tenim avui dia amb HTML5 i CSS3 d'incloure-hi estils nostres és molt ample. En aquest exemple pots provar: `-moz-transform:rotate(20deg);`

Molts atributs que teníem en versions anteriors de HTML com ara `align`, `marginheight`, `marginwidth` no estan suportats a HTML5 (la qual cosa no vol dir que el navegador web no les sàpiga interpretar, de moment). Utilitzarem estils CSS per acabar de polir l'aspecte que desitgem per al nostre `<iframe>`. Aquesta manera de fer està totalment d'acord amb la tendència actual de separar el contingut web del seu disseny. Vegem-ho amb un exemple:

```
<!DOCTYPE html>
<html>
    <head>
        <title>iframes</title>
        <style>
            iframe {
              width: 600px;
              height: 600px;
              border: #ff0000 1px dotted;
              margin-top: 30px;
              margin-bottom: 30px;
              padding: 20px;
            }
        </style>
    </head>
    <body>
        <h1>iframe de la IOC</h1>
        <iframe src="http://ioc.xtec.cat/educacio/"></iframe>
    </body>
</html>
```

Si no volem que apareguin els *scrolls*, i que l'`<iframe>` es fusioni totalment amb la nostra pàgina, podem provar el següent codi:

```
<!DOCTYPE html>
<html>
    <head>
        <title>iframes</title>
        <style>
```

```
            iframe {
              width: 1000px;
              height: 1000px;
              border:none;
              padding: 0;
              margin: 0;
              overflow: hidden;
              overflow-x: hidden;
              overflow-y: hidden;
            }
        </style>
    </head>
    <body>
        <h1>iframe de la IOC</h1>
        <iframe src="http://ioc.xtec.cat/educacio/" scrolling="no"></iframe>
    </body>
</html>
```

<frame> vs <iframe>

A HTML existeixen els tags `<frame>` i `<frameset>`, que ja no estan suportats per HTML5. Amb `<frameset>` podem dividir una pàgina en diverses parts, horitzontalment i verticalment, i incrustar diferents `<frame>`. Amb aquesta tècnica es van fer moltes pàgines web al principi d'Internet comercial, a finals dels anys 90. Avui dia ja no fem servir els `<frame>` sinó els `<iframe>`, que ens permeten incrustar una pàgina dins d'una altra.

3.4.1 Programació amb marcs (iframes)

L'objecte `window` té la propietat `frames`, que retorna tots els elements `<iframe>` de la finestra actual.

Vegeu aquest exemple, on es recorren tots els marcs i s'hi carregua un contingut de la Viquipèdia:

```
<html>

<head>
  <title>Llista de municipis</title>
  <style>
    iframe {
      width: 800px;
      height: 200px;
      border: #ff0000 1px dotted;
      margin-top: 10px;
      margin-bottom: 0px;
    }
  </style>
  <script>
    function carregarFrames() {
      let array = ["Manresa", "Sallent", "Balsareny", "Puig-reig"];
      let frames = window.frames;
      console.log(frames.length);
      console.log(frames[0].width);
      for (let i = 0; i < frames.length; i++) {
        frames[i].location = "https://ca.wikipedia.org/wiki/" + array[i];
        console.log(window.frames[i].parent === window) //true: el pare del
            iframe és la finestra on està incrustada
      }
    }
  </script>
</head>

```

```
<body onload="carregarFrames()">
  <h1>Llista de municipis</h1>
  <iframe></iframe>
  <iframe></iframe>
  <iframe></iframe>
  <iframe></iframe>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/xxZMNbE?editors=1010.

Un cas d'ús molt habitual per a la utilització d'`iframe` és incrustar editors de text o vídeos en una pàgina web. Per exemple, Youtube inclou un botó per compartir vídeos i l'opció *Inserir el vídeo* mostra el codi HTML per inserir-lo a qualsevol pàgina web, només cal enganxar-lo. Per exemple:

```
<iframe width="560" height="315" src="https://www.youtube.com/embed/F1UP7wRCPH8
    " frameborder="0" allow="accelerometer; autoplay; encrypted-media;
    gyroscope; picture-in-picture" allowfullscreen></iframe>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ExPrzPy?editors=1000.

Com que el contingut dels *frames* pot trigar a carregar-se (i més si el seu contingut està en un domini remot), té sentit llençar l'*event* de quan s'ha acabat de carregar l'*iframe*:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Llista de municipis</title>
    </head>
    <body>
        <h1>Llista de municipis</h1>
        <iframe src="http://ioc.xtec.cat/educacio/" name="IOC" style="height
            :100px"></iframe>
        <script>
          // set onload on element
          document.getElementsByTagName('iframe')[0].onload = function() {
            alert('1. Frame carregat del tot')
          }

          // set onload on window
          frames[0].onload = function() {
            alert('2. Frame carregat del tot')
          }
        </script>
    </body>
</html>
```

Arribats a aquest punt s'ha de tenir en compte que la crida a les funcions amb `onload` us funcionarà en el primer exemple i en el segon no. I és que hi ha un concepte important, que són les **polítiques d'accés a dominis creuats** (*cross-domain access policies*), o la política del mateix origen (*same origin policy*). Per motius de seguretat, accedir a contingut i propietats d'URLs que es troben en altres dominis està deshabilitat. I per tant, no podem pretendre que puguem accedir a totes les propietats d'un *iframe* quan no tenen el mateix nom de domini, protocol i port que la URL pròpia. Per exemple:

```
frames[0].height = '30px'; //Error: Permission denied to access property '
    height'
```

Però en canvi sí que podem fer:

```
document.getElementById("frame1").style.height = '100px';
```

És a dir, no podem accedir a la propietat com a element del DOM, de l'objecte `window`, però sí que podem accedir a l'estil HTML de l'element.

3.4.2 Comunicació entre marcs (iframes)

Es pot enviar missatges entre els diferents *frames* que composen una pàgina. Per fer-ho, s'ha de fer referència als *frames* com a objectes del DOM: a JavaScript, un cop es té referenciat un objecte, es pot accedir als seus mètodes i propietats. A més a més, un objecte pot tenir com a propietats els descendents que hi pengen en la seva jerarquia. Per exemple, veureu tot seguit com, en un document HTML, es pot accedir als formularis que conté; i en un formulari, es pot accedir als elements que conté.

Per implementar el següent exemple necessitareu tres fitxers: pare.html, frame1.html i frame2.html:

pare.html:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Comunicació entre iframes</title>
    </head>
    <body>
        <h1>Comunicació entre iframes</h1>
        En aquest exemple hem de posar el valor que volem que tingui l'altura
            de l'altre iframe. Per exemple: 100.<br />
        <iframe id="iframe1" src="frame1.html"></iframe>
        <iframe id="iframe2" src="frame2.html"></iframe>
    </body>
</html>
```

frame1.html:

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>iframe 1</title>
        <script>
            function funcio1() {
              let ifrm1 = parent.document.getElementById('iframe1');
              let ifrm2 = parent.document.getElementById('iframe2');
              let win1 = ifrm1.contentWindow;
              let win2 = ifrm2.contentWindow;
              console.log(win1);
              let doc1 = ifrm1.contentDocument? ifrm1.contentDocument: ifrm1.
                  contentWindow.document;
```

```
14              let doc2 = ifrm2.contentDocument? ifrm2.contentDocument: ifrm2.
                    contentWindow.document;
15              console.log (doc1);
16              let fld1 = doc1.forms[0].elements[0];
17              console.log(fld1);
18              let valor1 = fld1.value;
19              console.log(valor1);
20              doc2.forms[0].elements[0].value = valor1;
21              ifrm2.style.height = valor1 + 'px';
22            }
23        </script>
24    </head>
25    <body>
26        <h1>iframe 1</h1>
27        <form>
28            <input id="txt1" type="text" />
29            <button onclick="funcio1()">Passar al iframe2 i canviar altura</
                  button>
30        </form>
31    </body>
32 </html>
```

frame2.html:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta http-equiv="content-type" content="text/html; charset=utf-8">
5          <title>iframe 2</title>
6          <script>
7              function funcio2() {
8                let ifrm1 = parent.document.getElementById('iframe1');
9                let ifrm2 = parent.document.getElementById('iframe2');
10               let win1 = ifrm1.contentWindow;
11               let win2 = ifrm2.contentWindow;
12               console.log(win2);
13               let doc1 = ifrm1.contentDocument? ifrm1.contentDocument: ifrm1.
                     contentWindow.document;
14               let doc2 = ifrm2.contentDocument? ifrm2.contentDocument: ifrm2.
                     contentWindow.document;
15               console.log (doc2);
16               let fld2 = doc2.forms[0].elements[0];
17               console.log(fld2);
18               let valor2 = fld2.value;
19               console.log(valor2);
20               doc1.forms[0].elements[0].value = valor2;
21               ifrm1.style.height = valor2 + 'px';
22             }
23         </script>
24     </head>
25     <body>
26       <h1>iframe 2</h1>
27         <form>
28         <input id="txt2" type="text" />
29         <button onclick="funcio2()">Passar al iframe1 i canviar altura</button>
30       </form>
31     </body>
32 </html>
```

Des d'un dels *frames* es pot accedir al seu pare mitjançant `parent.document`. Podeu accedir als continguts dels *frames* amb `contentDocument` o `contentWindow.contentDocument` (depenent dels navegadors); i un cop es té referenciat el document, es pot accedir als elements del DOM, com ara els formularis i els elements que contenen (les caixes de text). Es pot canviar el contingut de les caixes de text amb la propietat `value`.

Per acabar, cal tenir en compte que cada *frame* es considera un document independent i, per consegüent, quan es faci una cerca de nodes al document no es trobaran els nodes dintre d'un *frame*. Pel mateix motiu, els estils CSS aplicats al document principal no afecten el contingut dels *frames*, ja que són diferents documents i habitualment carreguen els seus propis estils.