Desenvolupament web en l'entorn client

CFGS.DAW.M06/0.16

Desenvolupament d'aplicacions web

Generalitat de Catalunya
Departament d'Educació

ioc
institut obert de catalunya

Aquesta col·lecció ha estat dissenyada i coordinada des de l'Institut Obert de Catalunya.

Coordinació de continguts
Joan Ramon Serret

Redacció de continguts
Joan Quintana
Àlex Salinas
Xavier García

Primera edició: setembre 2016

Dipòsit legal: DL B 29919-2017

Introducció

Actualment la gran majoria de les pàgines web són dinàmiques, és a dir, interactuen amb l'usuari. Per exemple, ho són les pàgines que us proporcionen accés a diferents serveis, com el correu electrònic, el xat, cercadors de continguts, visualitzadors de vídeos, enquestes *en línia*, etc.

Moltes d'aquestes interaccions no poden realitzar-se només amb la intervenció del servidor, ja que el temps de resposta seria massa lent per a l'usuari. Imagineu-vos per un moment que per veure un vídeo, el servidor, per a cada imatge, hagués d'enviar tota la pàgina. Per aquest motiu, cal que el client sigui capaç de realitzar algunes tasques en resposta a les accions de l'usuari o de la informació que li envia el servidor.

Això s'aconsegueix afegint petits programes a les pàgines web, que són enviats des del servidor. Aquests programes s'executen en l'ordinador client. Com a conseqüència de l'execució d'aquests programes, l'ordinador client pot fer canvis a allò que s'està presentant per pantalla, o enviar algun missatge al servidor.

Si us heu matriculat en aquest mòdul vol dir que ja teniu una bona base de programació i una base suficient d'HTML. Amb això en teniu prou per endinsarvos en la programació web des del costat client.

Al mòdul es treballa amb el llenguatge de programació del costat client més utilitzat avui en dia i que és suportat per tots els navegadors: JavaScript. Compte, però, perquè, malgrat el seu nom, és molt diferent respecte el llenguatge de programació Java, que ja coneixeu.

Aquest mòdul està dividit en 7 unitats:

1. A la unitat "Sintaxi del llenguatge. Objectes predefinits del llenguatge" es treballa la sintaxi bàsica de JavaScript i els objectes principals que ens proporciona.

2. A la unitat "Estructures definides pel programador" s'estudien les funcions i els vectors. Veureu que JavaScript té moltes peculiaritats respecte altres llenguatges.

3. A la unitat "Objectes definits pel programador" s'exposa la visió que té JavaScript dels objectes. Veureu que és una mica peculiar, però que, a la vegada, els objectes són bàsics per poder extreure el màxim del llenguatge.

4. A la unitat "Esdeveniments. Manejament de formularis" aprendreu a capturar i tractar amb JavaScript els esdeveniments generats pels usuaris.

5. A la unitat "Model d'objectes del document" s'aprofundeix en les possibilitats dels formularis.

6. A la unitat "Mecanismes de programació asíncrona client-servidor" s'estudia la comunicació client-servidor amb la tecnologia AJAX.

7. Per últim, a la unitat "Desenvolupament de casos pràctics" es treballa amb exemples d'una certa mida on s'aprofiten al màxim les possibilitats de comunicació client-servidor que proporciona JavaScript.

Com veieu, és un mòdul amb uns continguts i un enfocament molt pràctics. Per això és important que aneu comprovant tots els exemples que us hi apareixeran, que us esforceu per entendre'ls i que "jugueu" amb ells, fent-hi canvis en el codi i comprovant els resultats. Per ajudar-vos, veureu en molts dels exemples és possible accedir, a través d'un enllaç, a una implementació operativa.

Resultats d'aprenentatge

En finalitzar aquest mòdul l'alumne/a:

Sintaxi del llenguatge. Objectes predefinits del llenguatge

1. Selecciona les arquitectures i tecnologies de programació sobre clients web, identificant i analitzant les capacitats i característiques de cadascuna.
2. Escriu sentències simples, aplicant la sintaxi del llenguatge i verificant la seva execució sobre navegadors web.
3. Escriu codi, identificant i aplicant les funcionalitats aportades pels objectes predefinits del llenguatge.

Estructures definides pel programador. Objectes

1. Programa codi per a clients web analitzant i utilitzant estructures definides per l'usuari.

Esdeveniments. Manejament de formularis. Model d'objectes del document

1. Desenvolupa aplicacions web interactives integrant mecanismes de maneig d'esdeveniments.
2. Desenvolupa aplicacions web analitzant i aplicant les característiques del model d'objectes del document.

Programació asíncrona client-servidor

1. Desenvolupa aplicacions web dinàmiques, reconeixent i aplicant mecanismes de comunicació asíncrona entre client i servidor

Continguts

Sintaxi del llenguatge. Objectes predefinits del llenguatge

Unitat 1

Sintaxi del llenguatge. Objectes predefinits del llenguatge

1. Execució de programes al client. Introducció a JavaScript.
2. Programació amb els objectes predefinits de JavaScript. BOM (*Browse Object Model*).
3. Programació amb finestres, marcs i galetes.

Estructures definides pel programador. Objectes

Unitat 2

Estructures definides pel programador

1. Programació amb funcions.
2. Programació amb arrays.

Unitat 3

Objectes definits pel programador

1. Objectes definits pel programador.

Esdeveniments. Manejament de formularis. Model d'objectes del document

Unitat 4

Esdeveniments. Manejament de formularis

1. Programació d'esdeveniments.
2. Programació amb formularis.

Unitat 5

Model d'objectes del document

1. Programació amb el DOM (*Document Object Model*).
2. Programació amb el DOM i la biblioteca jQuery.

Programació asíncrona client-servidor

Unitat 6

Mecanismes de programació asíncrona client-servidor

1. Comunicació asíncrona amb JavaScript.
2. Comunicació asíncrona utilitzant la biblioteca jQuery.

Unitat 7

Desenvolupament de casos pràctics

1. Desenvolupament de casos pràctics.

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

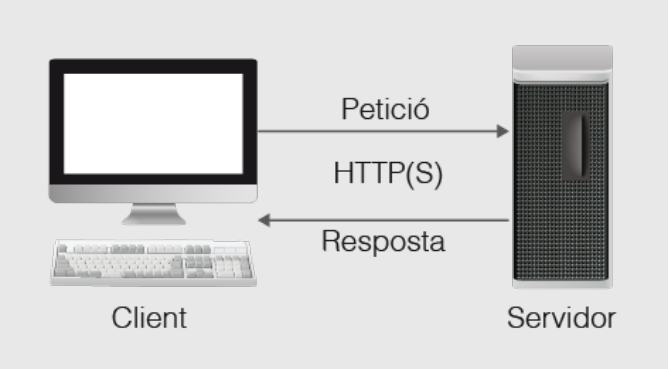
controlar una casa domòtica, on podem posar la calefacció a distància. O una aplicació per al mòbil de calendari, que se sincronitza contínuament i ens avisa de quan tenim una reunió. Per tant, hem de prendre consciència del fet que el que ens permet la tecnologia web, avui dia, és molt més que visitar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

```
$ 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:

```
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:

```
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>
```

```
8
9     <button onclick="calcular();">Calcular</button>
10
11    <h2>Cost extras: <span id="extra">0</span€></h2>
12
13    <script>
14      let opcio1 = document.getElementById('opcio1');
15      let opcio2 = document.getElementById('opcio2');
16      let opcio3 = document.getElementById('opcio3');
17      let extra = document.getElementById('extra');
18
19      function calcular() {
20        let total = 0;
21        if (opcio1.checked) {
22          total += Number(opcio1.value);
23        }
24        if (opcio2.checked) {
25          total += Number(opcio2.value);
26        }
27        if (opcio3.checked) {
28          total += Number(opcio3.value);
29        }
30        extra.innerHTML = total;
31      }
32    </script>
33  </body>
34
35  </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:

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

Així doncs, el contingut de fitxerExtern.js serà:

```
1  function funcio() {
2      document.getElementById("paragraf").innerHTML = "Escrivim en el paràgraf";
3  }
```

i el nostre codi quedarà de la següent manera:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta http-equiv="content-type" content="text/html; charset=utf-8">
5          <title>El meu primer exemple</title>
6          <script src="fitxerExtern.js"></script>
7      </head>
8      <body>
9          <h1>El meu primer exemple</h1>
10         <p id="paragraf">Text original del paràgraf</p>
11         <button type="button" onclick="funcio()">Clica'm</button>
12     </body>
13 </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)

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.

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

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.

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

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');
```

```
let gat = new Animal(2, 'Gat', 'Mamífers', 'Fèlid');

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

```
function hola(nom) {
  alert(`Hola ${nom}!`);
}

function suma(a, b) {
  return a + b;
}

hola('Joan');
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.

Assignació de valors i comparació lògica

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

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

```
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 />`.

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

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.

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.

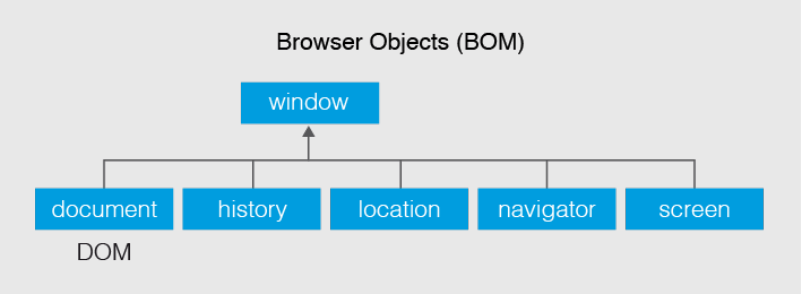
2.2 BOM (Browser Object Model)

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`:

```
1 console.log(‘appCodeName: ${navigator.appCodeName}‘);
2 console.log(‘appName: ${navigator.appName}‘);
3 console.log(‘appVersion: ${navigator.appVersion}‘);
4 console.log(‘cookieEnabled: ${navigator.cookieEnabled}‘);
5 console.log(‘language: ${navigator.language}‘);
6 console.log(‘onLine: ${navigator.onLine}‘);
7 console.log(‘platform: ${navigator.platform}‘);
8 console.log(‘product: ${navigator.product}‘);
9 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:

```
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:

```
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

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.

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.

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.

Estructures definides pel programador

Àlex Salinas

Índex

Introducció

El codi Javascript no es pot executar de manera independent: sempre s'ha d'executar utilitzant un navegador web. Aquest codi està associat a una pàgina web. Aquesta pàgina estarà allotjada en un servidor d'aplicacions, i quan un navegador la demani s'enviarà junt amb el codi Javascript.

Quan aquesta pàgina arriba al navegador web es comença a interpretar. Els navegadors no interpreten la pàgina després de rebre-la, sinó que la van interpretant i, per tant, executant el codi Javascript a mesura que la van rebent. Existeixen maneres per evitar executar el codi Javascript abans de rebre tota la pàgina, perquè normalment es imprescindible tindre-la sencera.

En un principi, l'utilització de Javascript es va limitar en la creació d'efectes dinàmics dintre de la pàgina com, per exemple, canviar una imatge per un altre quan el ratolí es posava damunt o bé ressaltar algun paràgraf. Però poc a poc, es va anant ampliant el seu ús i amb la aparició de biblioteques com JQuery o AngularJS ha esdevingut un llenguatge imprescindible per desenvolupar aplicacions web modernes. Amb aquestes eines, Javascript permet crear efectes dinàmics impressionants que milloren l'experiència que rep l'usuari al utilitzar l'aplicació.

En el primer apartat de la unitat s'explicaran els conceptes relacionats amb les funcions. Les funcions són fragments de codi que resolen una tasca clarament definida. El seu resultat és un valor que pot ser utilitzat per altres funcions o pel programa principal. Tanmateix, a JavaScript veurem que no és sempre així: les funcions tenen un paper més important.

En aquest apartat aprendreu per què les funcions són la clau per entendre els secrets del llenguatge JavaScript. Les funcions són objectes igual que qualsevol altre tipus de dades. Poden ser declarades, poden ser referenciades com si fossin variables i fins i tot es poden passar com a paràmetres d'altres funcions.

Aquesta peculiaritat fa de JavaScript un llenguatge diferent. Si intenteu programar com si programéssiu en un altre llenguatge, sobretot com si programéssiu en Java, aquest no es comportarà com penseu.

S'explicaran les funcions més importants, que hem d'aprendre per donar els primer passos amb aquest llenguatge. Aquestes funcions tenen a veure amb la manipulació de cadenes de text o de nombres. També s'explicaran funcions especials de la biblioteca JavaScript que permeten programar l'execució d'altres funcions.

En definitiva, començarem explorant la potència del llenguatge JavaScript per tractar amb els tipus de dades més comuns. A continuació, començarem a explorar el costat més diferent del llenguatge. Començarem a explicar les funcions que un

desenvolupador pot crear. Per exemple, un programador JavaScript pot decidir crear una funció i no donar-li un nom o bé que s'autoexecuti una vegada s'ha creat. Són característiques que donen una potència inesperada a aquest llenguatge de programació del costat client.

En el segon apartat de la unitat s'explicaran els *arrays*. Aquest tipus de dades és molt útil i molt pràctic a l'hora de crear programes. Es mereixen una especial atenció, i s'explicaran les funcionalitat que ens dóna la biblioteca JavaScript per tractar aquest tipus de dades en forma d'exemple. Es realitzarà la creació d'un programa per gestionar una botiga i, d'aquesta manera, explicar les funcionalitats dels *arrays* des d'una vessant merament pràctica. Es realitzaran uns exemples molt senzills i, a partir d'aquests, es proposaran diferents mètodes per resoldre els problemes plantejats.

S'explicaran les funcions més importants per tractar amb els *arrays* d'una manera pràctica i útil que us ajudarà a entendre molt millor les diferents opcions que ens proposa JavaScript per manipular-los.

Javascript està present en pràcticament qualsevol àmbit: bases de dades, desenvolupament mòbil, servidors d'Internet, sistemes operatius, plataformes de jocs, administració de sistemes, etc. La seva influencia és increïble. Javascript quasi bé ha complert la promesa que feia Java fa més de 20 anys de "*Write once, run everywhere*".

Resumint, en aquesta unitat s'explicarà:

- Les funcions i la seva importància
- Les funcions més importants del llenguatge JavaScript
- Com crear funcions
- Com assignar paràmetres
- Com el navegador invoca les funcions
- Reconèixer les funcions com objectes de primera classe
- Els tipus de dades
- Els *arrays*: característiques i funcionalitats

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Programa codi per a clients web analitzant i utilitzant estructures definides per l'usuari.

- Classifica i utilitza les funcions predefinides del llenguatge.
- Crea i utilitza funcions definides per l'usuari.
- Reconeix les característiques del llenguatge relatives a la creació i ús d'arrays.
- Crea i utilitza arrays.
- Depura i documenta el codi.

1. Programació amb funcions

La principal diferència entre escriure bon codi JavaScript i un codi mediocre és concebre-ho com un llenguatge funcional. Les funcions són la base d'aquest llenguatge i saber utilitzar-les marca la diferència.

El més important és que les funcions són objectes de primera classe, és a dir, coexisteixen amb qualsevol altre objecte i les podem tractar com un d'ells. Igual que els altres tipus de JavaScript (Integer, Boolean...) les funcions les podem crear com a literals i fins i tot les podem passar com a paràmetres d'altres funcions. És convenient començar l'estudi de les funcions veient les ja definides al llenguatge.

1.1 Funcions predefinides pel llenguatge

És important conèixer les diferents funcions que ens proporciona el llenguatge JavaScript per tractar cadenes de text, nombres, conversions de tipus i altres funcions interessants.

1.1.1 Funcions predefinides de cadenes de text

Abans de començar a parlar de les funcions predefinides de les cadenes de text cal destacar la propietat **length** que retorna la mida de la cadena de text, ja que aquesta és una propietat que s'utilitza força sovint. Veieu un exemple:

```
1 let frase = "Estudiant a l'IOC s'aprèn molt.";
2 document.write(frase.length);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/rNeNyGe?editors=0010.

A continuació podeu veure algunes de les funcions predefinides de les cadenes de text més utilitzades.

concat()

Per concatenar o unir dues paraules o frases emmagatzemades en variables diferents es pot utilitzar el símbol **+** o bé la funció **concat**. S'ha de tenir en compte que a l'hora d'utilitzar-les s'uniran les dues paraules sense cap separació, per tant, si es volen separar les paraules a la cadena resultant s'ha d'afegir el símbol corresponent.

```
let frase1 = "Estudiant a l'IOC";
let frase2 = "s'aprèn molt.";
//sense separació
let resultat = frase1 + frase2;
document.write(resultat + "<br>");

//amb separació
let resultat = frase1 + " " + frase2;
document.write(resultat + "<br>");

//sense separació
let resultat = frase1.concat(frase2);
document.write(resultat + "<br>");

//amb separació
let resultat = frase1.concat(" " +frase2);
document.write(resultat);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/LYNYWQz?editors=0010.

Fixeu-vos què passa si es concatena una variable de tipus String amb una variable de tipus numèric.

```
let frase = "L'IOC és un ";
let nombre = 10;
document.write(frase + nombre);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYqYZWo?editors=0010.

Al concatenar un String amb una variable numèrica, aquesta es converteix automàticament a String. La biblioteca JavaScript permet fer el canvi automàticament sense que nosaltres haguem de fer cap tipus de conversió.

> Una alternativa a la concatenació de cadenes de text i variables és la utilització de plantilles de literals, per exemple: `document.write(`L'IOC és un ${nombre}`)`.

toUpperCase() i toLowerCase()

La funció **toUpperCase()** converteix la cadena sencera a majúscules. I la funció **toLowerCase()** realitza l'operació inversa a `toUpperCase`, és a dir, canvia els caràcters en majúscula per minúscula.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
document.write(frase.toUpperCase() + "<br>");
document.write(frase.toLowerCase());
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/JjXjWmr?editors=0010.

charAt()

La funció **charAt(posició)** retorna el caràcter que es troba a la posició indicada pel paràmetre.

```
var frase = "Estudiant a l'IOC s'aprèn molt.";
document.write(frase.charAt(0));
document.write(frase.charAt(1));
document.write(frase.charAt(2));
document.write(frase.charAt(3));
document.write(frase.charAt(4));
document.write(frase.charAt(5));
document.write(frase.charAt(6));
document.write(frase.charAt(7));
document.write(frase.charAt(8));
document.write(frase.charAt(9));
document.write(frase.charAt(18));
document.write(frase.charAt(19));
document.write(frase.charAt(20));
document.write(frase.charAt(21));
document.write(frase.charAt(22));
document.write(frase.charAt(23));
document.write(frase.charAt(24));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/xxVxqQL?editors=0010.

El resultat de l'exemple és *Estudiant s'aprèn*. Aquesta frase és una porció de la frase original. Existeixen altres maneres de poder obtenir porcions de frases. Per exemple amb la funció `substring`.

substring

La funció **substring(inici, final)** retorna una porció d'un text des de la posició `inici` fins a la posició `final`. Si només s'indica el paràmetre `inici`, la funció retorna la part de la cadena original corresponent des d'aquesta posició fins al final.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
document.write( frase.substring(0,10) + "<br>");
document.write(frase.substring(18,25) + "<br>");
```

Si volem obtenir la mateixa cadena que a l'exemple `charAt` llavors s'ha de concatenar les dues porcions de la cadena original.

```
document.write(frase.substring(0,10) + frase.substring(18,25) + "<br>");
```

Podeu veure aquest exemple en l'enllaç següent:https://codepen.io/ioc-daw-m06/pen/ExKxWGX?editors=0010.

Fixeu-vos el comportament de la funció quan se li indica un valor negatiu com a posició inicial o final.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
document.write("Primer exemple: " + frase.substring(0,-12) + "<br>");
document.write("Segon exemple: " + frase.substring(-1,2)+ "<br>");
```

```
4 document.write("Tercer exemple:" + frase.substring(-1,-12)+ "<br>");
5 document.write("Quart exemple:" + frase.substring(-22));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawpEe?editors=0010.

Com heu vist, el primer exemple no retorna cap porció de la cadena original ja que la posició final negativa no la pot traduir a una posició final coherent.

En canvi, en el segon exemple al utilitzar una posició positiva, en aquest cas com a final de subcadena, llavors ha retornat des de l'inici (posició inicial igual a 0) fins a la posició final indicada (en aquest cas posició final igual a 2). El resultat és, doncs, la subcadena *Es*.

En el tercer exemple no ha retornat cap subcadena ja que, tal i com passava en l'exemple primer, no pot traduir les posicions negatives i per tant, no pot retornar cap subcadena.

En canvi, a l'últim exemple només se li indica d'una posició negativa. En aquest cas, el comportament de la funció `substring` és retornar la frase original sencera.

I si en comptes d'utilitzar nombres negatius utilitzem una posició final més petita que una posició inicial?

```
1 let frase = "Estudiant a l'IOC s'aprèn molt.";
2 document.write(frase.substring(9, 0));
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ExKxWrX?editors=0010.

Bé, en aquest cas, la funció `substring` canvia l'ordre de les posicions. El resultat del codi anterior és *Estudiant*.

indexOf()

La funció **indexOf(caracter)** retorna la posició de la primera ocurrència d'un caràcter en una cadena. Si la cadena no conté el caràcter, la funció retorna el valor -1. En canvi, la funció **lastIndexOf(caracter)** retorna l'última ocurrència del caràcter cercat.

```
1 let frase = "Estudiant a l'IOC s'aprèn molt.";
2 let posicio = frase.indexOf("IOC");
3 let posicio2 = frase.indexOf("ñ");
4 document.write('Posició de la paraula IOC: ${posicio}<br>');
5 document.write('Posició de la lletra ñ: ${posicio2}<br>');
6 posicio = frase.lastIndexOf("t");
7 posicio2 = frase.lastIndexOf("ñ");
8 document.write('Posició de la lletra t: ${posicio}<br>');
9 document.write('Posició de la lletra ñ: ${posicio2}');
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawpOq?editors=0010.

split()

La funció **split(separador)** separa una cadena de text en trossos. Es crea un pedaç de la cadena cada vegada que es troba el caràcter separador. Tots els pedaços de la cadena es retornen en un *array*.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
let trossos = frase.split(" ");
document.write(`Array: ${trossos}<br>`);
document.write(trossos[0] + "<br>");
document.write(trossos[1] + "<br>");
document.write(trossos[2] + "<br>");
document.write(trossos[3] + "<br>");
document.write(trossos[4] + "<br>");
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/KKzKWOV?editors=0010.

A cadascuna de les posicions de l'*array* `trossos` es troba un pedaç de la frase original. Els pedaços s'han creat cada vegada que s'ha trobat el caràcter *espai* (" "). La funció recorre caràcter a caràcter la frase original i cada vegada que troba el caràcter separador talla la cadena i guarda el pedaç tallat en una posició de l'*array*. Continua recorrent la cadena fins que s'acaba.

I si no li indiquem cap caràcter de separació? Si no li indiqueu cap caràcter separador, llavors separa caràcter a caràcter.

```
let frase = "Estudiant a l'IOC s'aprèn molt.";
let trossos = frase.split("");
document.write(trossos[0] + "<br>");
document.write(trossos[1] + "<br>");
document.write(trossos[2] + "<br>");
document.write(trossos[3] + "<br>");
document.write(trossos[4] + "<br>");
document.write(trossos[5] + "<br>");
document.write(trossos[6] + "<br>");
document.write(trossos[7] + "<br>");
document.write(trossos[8] + "<br>");
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyoPzb?editors=0010.

startsWith(), endsWith() i includes()

Aquestes tres funcions són molt útils a l'hora de comprovar cadenes de text, ja que ens permeten esbrinar si una cadena comença, acaba o inclou un fragment de text concret respectivament. Només cal passar com argument el fragment de text a cercar:

```
let cadena = "Estudiant a l'IOC s'aprèn molt";
console.log(`La cadena comença per "Estu"? ${cadena.startsWith("Estu")}`);
console.log(`La cadena acaba amb "molt"? ${cadena.endsWith("molt")}`);
console.log(`La cadena inclou "IOC"? ${cadena.includes("IOC")}`);

console.log(`La cadena comença per "1234"? ${cadena.startsWith("1234")}`);
console.log(`La cadena acaba amb "1234"? ${cadena.endsWith("1234")}`);
console.log(`La cadena inclou "1234"? ${cadena.includes("1234")}`);
```

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

repeat()

La funció **repeat(repeticions)** retorna la mateixa cadena repetida el nombre de repeticions indicada:

```
let cadena = "*";
for (let i=1; i<5; i++) {
  console.log(cadena.repeat(i));
}
```

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

padStart() i padEnd()

Aquestes funcions permeten omplir una cadena per la dreta o per l'esquerra per assegurar-nos que la longitud d'una cadena es fixa. Això és interessant quan es treballa amb estils o dispositius amb fonts monoespaiades (per exemple la consola del navegador), és a dir, que totes les lletres són de la mateixa amplada.

padStart(mida, cadenaPerAfegir) omple la cadena fins a arribar a la mida que s'especifica i amb la cadena indicada començant per l'esquerra, mentre que **padEnd(mida, cadenaPerAfegir)** fa el mateix però començant per la dreta. Podeu veure les diferències en el següent exemple:

```
let cadena = "Joan";
console.log(cadena);
console.log(cadena.padStart(2));
console.log(cadena.padStart(15));
console.log(cadena.padStart(15, "#"));
console.log(cadena.padStart(15, "#*"));
console.log(cadena.padEnd(2));
console.log(cadena.padEnd(15));
console.log(cadena.padEnd(15, "#"));
console.log(cadena.padEnd(15, "#*"));
let midaCadena = cadena.length;
let midaAmbPadStart = cadena.padStart(15).length;
let midaAmbPadEnd = cadena.padEnd(15).length;
console.log(`Mida original: ${midaCadena}`);
console.log(`Mida amb padStart: ${midaAmbPadStart}`);
console.log(`Mida amb padEnd: ${midaAmbPadEnd}`);
```

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

Com es pot apreciar, si no s'indica un segon argument, la cadena s'omple amb espais fins a arribar a la mida indicada. En el cas que la cadena passada com argument contingui més d'un caràcter es descarten els sobrants, de manera que la mida sempre es respecta. Per altra banda, si s'indica una mida inferior a la de la cadena original, es retorna la mateixa cadena i no s'ajusta la mida.

1.1.2 Funcions predefinides de nombres

La biblioteca JavaScript permet totes les operacions de nombres que es poden realitzar en qualsevol altre llenguatge. Per sumar dos nombres utilitzarem l'operador suma (+), per multiplicar utilitzarem l'operador *asterisc* (*), per restar utilitzarem l'operador guió (-) i per dividir utilitzarem l'operador barra (/). Aquestes són les operacions bàsiques que es poden realitzar amb dos nombres. Però, una operació pot donar valors inesperats si no es comproven els dos operands. Per exemple, què donaria una divisió on els dos valors són zero? O què passaria si es divideix un nombre qualsevol per zero? Bé, la biblioteca JavaScript ens dona alguns valors i funcions per protegir el nostre programa davant d'aquestes situacions.

Una de les funcions que ens ajudarà a evitar aquestes situacions és la funció **isNaN()**. Aquesta funció comprova si el valor retornat per una operació amb nombres és un nombre o bé no ho és. De fet **NaN** son les inicials, en anglès, de **Not A Number** (No és un nombre).

```
let nombre = 0;

if(isNaN(nombre/nombre)) {
   document.write("La divisió no és correcta.");
}
else {
   document.write(‘La divisió és: ${nombre/nombre}‘);
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/bGpGWmB?editors=0010.

Fixeu-vos en l'exemple anterior. Abans d'utilitzar el resultat de la divisió entre els dos nombres es comprova si aquesta retornarà un nombre o bé un valor no numèric. Així podem evitar utilitzar aquest valor en futures operacions. Bé, en el cas que no sigui un nombre s'informa a l'usuari d'aquesta situació i en el cas que l'operació sigui correcta es realitza la divisió.

A l'exemple següent forcem la divisió entre dos zeros. Quin és el resultat?

```
let nombre = 0;
document.write(nombre/nombre);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/RwawVqw?editors=0010.

La biblioteca JavaScript ha creat un valor que indica que no és un nombre. S'ha anomenat **NaN**. Com veieu aquest és el resultat de la divisió anterior.

També s'ha creat el valor **Infinity** per representar el valor infinit positiu i **-Infinity** per representar el valor infinit negatiu.

```
let nombre = 10;
let zero = 0;
document.write(nombre/zero);
document.write("<br>");
document.write(–nombre/zero);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYqYwML?editors=0010.

Si us fixeu, a la primera divisió dona com a resultat el valor positiu *Infinity* però a la segona divisió dona el valor *-Infinity* ja que és la divisió d'un nombre negatiu.

Una altra funció interessant de nombres és la funció **toFixed(digits)**. Aquesta funció arrodoneix els decimals d'un número.

Veieu-ne un exemple:

```
let nombre = 1234.56789;
document.write(nombre.toFixed(1) + "<br>");
document.write(nombre.toFixed(8)+ "<br>");
document.write(nombre.toFixed()+ "<br>");
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/gOrOWZa?editors=0010.

El resultat d'executar la funció amb el paràmetre 1 indica que només es vol un decimal. Al fer arrodoniment en comptes de donar el nombre *1234,5* dona *1234,6* ja que el valor 1234,56789 està més a prop d'aquesta xifra.

En canvi, si li enviem un nombre superior als decimals que té el nombre original el resultat de l'execució afegeix tants zeros com decimals falten. En aquest cas mostra el valor *1234,56789000*.

Finalment, si no li enviem cap dígit a la funció llavors li estem indicant que no volem decimals i per això el resultat de l'execució de la funció és *1235*. Fixeu-vos que en aquest cas també s'arrodoneix la xifra.

1.1.3 Altres funcions interessants

JavaScript té unes quantes funcions bastant diferents entre sí, però que en moltes ocasions són força útils. Algunes funcions demanen la intervenció de l'usuari per realitzar una acció i d'altres serveixen per executar una acció un nombre determinat de vegades.

Quadres de diàleg entre l'usuari i el programa

N'hi ha de tres tipus: funció `alert`, funció `confirm` i funció `prompt`.

Aquestes funcions mostren un diàleg amb un missatge i un o més botons. Cal destacar que aquestes funciones interrompen l'execució del programa fins que es clica algun dels botons.

> Aquestes funcions proporcionen una manera ràpida de visualitzar i entrar dades però no s'utilitzen en el desenvolupament de pàgines web, ja que no es pot canviar l'estil i aquest sempre depèn del navegador. En el seu lloc es fan servir diàlegs modals, on la visualització

s'implementa amb HTML i CSS i el comportament dels botons s'afegeix amb JavaScript (però no s'interromp l'execució).

- **Funció alert**: s'utilitza quan es vol donar una informació a l'usuari sense esperar cap resposta d'ell. Es crearà una finestra amb un únic botó.

```
alert("Hola Món");
```

- **Funció confirm**: s'utilitza quan es vol donar l'opció a l'usuari de confirmar o no l'execució d'una tasca del programa. Es crearà una finestra amb dos botons: *True* o *False*.

```
let resposta = confirm("Vols saber la hora?");
if (resposta){
   document.write(new Date());
}
else{
   document.write("oooh.");
}
```

- **Funció prompt**: s'utilitza per demanar a l'usuari alguna informació. Es crearà una finestra amb un quadre de text on l'usuari pot introduir la informació demanada.

```
let resposta = prompt("Escriu el teu nom:");
document.write('El teu nom és: ${resposta}');
```

Podeu veure un exemple complet en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWvXM?editors=1010.

Funcions de temporització

Hi ha dues funcions de temporització: setInterval(codi_a_executar, temps_en_ms) i setTimeout(codi_a_executar, temps_en_ms):

- **setTimeout(codi_a_executar, temps_en_ms)**: executa una funció després d'un instant de temps.
  - codi_a_executar: nom de la funció que s'executarà després que hagi passat el temps *temps_en_ms* en mil·lisegons.
  - temps_en_ms: temps que trigarà a executar-se la funció *codi_a_executar*.
  - Aquesta funció retorna un *ID* (identificador) per poder cancel·lar la seva execució.

Per veure clarament que hi ha un retard en executar la funció `setTimeout` a l'exemple ens ajudem de dos botons. El primer per configurar la funció `setTimeout` i el segon per cancel·lar la seva execució. El codi és el següent:

```
<p>Exemple de la funció setTimeOut:</p>
<button onclick="funcioRetardada();">Mostra una alerta amb retard</button>
<p></p>
<button onclick="cancelaFuncio();">Cancel·la l'alerta abans que es produeixi</
    button>
```

Codi JavaScript associat a l'HTML anterior:

```
let timeoutID;

function funcioRetardada() {
  timeoutID = setTimeout(alerta, 2000);
}

function alerta() {
  alert("Hola Món!");
}

function cancelaFuncio() {
  clearTimeout(timeoutID);
}
```

Podeu veure aquest exemple, completat amb codi HTML per fer-lo més entenedor, a l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOLXNj?editors=1010.

- **setInterval (codi_a_executar, temps_en_ms)**: executa una funció indefinidament. L'interval de temps entre cada execució de la funció és definit pel paràmetre *temps_en_ms*.
  - codi_a_executar: nom de la funció que s'executarà cada *temps_en_ms* mil·lisegons.
  - temps_en_ms: temps en mil·lisegons que s'esperarà entre cada execució de la funció *codi_a_executar*.
  - Aquesta funció retorna un *ID* (identificador) per poder cancel·lar la seva execució.

Per veure clarament el funcionament de la funció `setInterval`, a l'exemple, ens ajudem de dos botons. El primer per configurar la funció `setInterval` i el segon per cancel·lar la seva execució. El codi és el següent:

Codi HTML de l'exemple:

```
<p>Exemple de la funció setInterval:</p>
<button onclick="funcioReiterada();">Mostra una alerta cada 2 segons</button>
<p></p>
<button onclick="cancelaExecucio();">Cancel·la l'alerta</button>
```

Codi JavaScript associat a l'HTML anterior:

```
let intervalID;
let numExec = 1;

function funcioReiterada() {
  intervalID = setInterval(alerta, 2000);
}

function alerta() {
```

```
  alert(‘Hola, aquesta és ´lexecució número: ${numExec}‘);
  numExec++;
}

function cancelaExecucio() {
  clearInterval(intervalID);
}
```

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

1.2 Funcions definides pel programador

Les funcions, en JavaScript, són objectes de primera classe, és a dir, coexisteixen amb qualsevol altre objecte i poden tractar-se com un d'ells. Igual que els tipus més mundans de JavaScript, les variables poden fer referència a elles, es poden declarar amb literals i fins i tot passar-se com a paràmetres d'altres funcions. La funció és la principal unitat modular d'execució. Vol dir que excepte les instruccions incrustades en el codi que s'executen mentre s'avalua les etiquetes, tota la resta està dins d'una funció.

Els objectes tenen les següents capacitats:

- Poden crear-se a través de literals.
- S'assignen a variables, entrades de matriu i propietats d'altres objectes.
- Es poden passar com a arguments per a funcions.
- Es retornen com a valors a partir de funcions.
- Tenen propietats que poden crear-se i assignar-se de forma dinàmica.

Les funcions tenen totes aquestes capacitats i, a més, tenen la capacitat que poden invocar-se.

Exemple d'una funció auto-executable:

```
let parellOsenar = (function() {
    let avui = new Date()
    if (new Date().getDate() % 2 == 0) {
        return function() { document.write("Avui és dia parell") }
    } else {
        return function() { document.write("Avui és dia senar") }
    }
})();

parellOsenar();
```

La funció `parellOsenar` retorna una altra funció segons sigui el dia parell o senar. La funció retornada s'executa i s'escriu la frase apropiada.

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyowyV?editors=0010.

1.2.1 Declaració

Les funcions es declaren utilitzant la següent especificació:

```
function nom([parametre,[, parametre,[..., parametre]]]) {
   [instruccions]
}
```

Les funcions literals es componen de quatre parts:

- La paraula clau `function`
- Un nom opcional que, si s'especifica, ha de ser un identificador vàlid.
- Una llista separada per comes de noms de paràmetres entre parèntesis. La llista pot estar buida, però els parèntesis han d'estar presents.
- El cos de la funció. Una sèrie d'instruccions entre claus.

El nom de la funció és opcional, perquè si no hi ha una necessitat de posar un nom a una funció concreta, no cal fer-ho.

Quan se li posa un nom a una funció, és vàlid en tot l'àmbit en el qual aquesta es declara. Si una funció es declara amb nom en el nivell superior, es crea una propietat utilitzant el nom de la funció en l'objecte `window`. Finalment, totes les funcions tenen dins una propietat oculta anomenada `name` que emmagatzema el nom de la funció com una cadena.

En el cas de les funcions assignades a variables aquesta propietat tindrà el nom de la variable i serà buida en el cas de les funcions anònimes.

Exemple de sintaxi d'una funció JavaScript:

```
function estudiar()
{
  let frase  = "A l'IOC s'estudia molt.<br>";
  return frase;
}
```

Aquest tipus de funcions es declaren per a fer-ne ús en un altre moment i tantes vegades com sigui necessari.

Proveu tot el que s'ha comentat anteriorment amb un exemple. Utilitzareu un joc de proves per veure si realment les funcions són objectes de primera classe. Les funcions que provareu seran les següents:

```
function lleugera() {
  return true;
}
```

```
var sensenom = function(){return true;};

window.esVeritat = function(){return true;};

function externa(){
  function interna(){}
}

var una_funcio = function una_altre_funcio(){return true;};
```

S'utilitzarà el joc de proves amb la biblioteca *QUnit.js*. Aquesta biblioteca té una finalitat similar a la biblioteca *JUnit* de Java. S'utilitza per definir jocs de proves.

QUnit té implementada la funció `test` i la funció `equal`. Amb la funció `test` definim un nou joc de proves i li donem un nom. La funció `equal` comprova si la igualtat que li passem com a primer paràmetre correspon amb el que nosaltres esperem (segon paràmetre), si és així, el resultat de la execució de la funció `equal` serà *okay*:

```
test("Conjunt de Tests per a funcions", function() {
  equal(typeof window.lleugera === "function", true);
  equal(lleugera.name === "lleugera", true);
  equal(typeof window.sensenom === "function", true);
  equal(sensenom.name === "sensenom", true);
  equal(typeof window.esVeritat === "function", true);
  equal(typeof window.externa === "function", true);
  equal(typeof window.interna === "function", false);
  equal(window.una_funcio.name === "una_altre_funcio", true);
});
```

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

Aquest joc de proves demostra que les funcions s'afegeixen com a propietats de l'objecte `window`. A més, es pot veure també la propietat `name`. El fet que `window.sensenom` es defineixi com una funció demostra que les variables globals, fins i tot les que contenen funcions, acaben amb `window`. I, finalment, cal puntualitzar que `window.esVeritat` es defineix com una funció.

Actualment JavaScript inclou un altre tipus de funció, anomenada "funció de fletxa", que substitueix a les funcions anònimes quan el cos de la funció és molt simple. Aquest tipus de funcions és molt més concís i, quan s'utilitza adequadament, fa que el codi sigui més fàcil d'entendre.

Les funcions de fletxa s'utilitzen de forma semblant a les funcions Lambda en altres llenguatges com Java.

La sintaxi de les funcions de fletxa és la següent:

```
([parametre,[, parametre,[..., parametre]]]) => { [instruccions] }
```

Per exemple:

```
const sumar = (x, y) => { return x + y };
console.log(sumar(2, 3));
```

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

Quan el cos de la funció consisteix en el retorn d'una expressió pot eliminar-se tant el "return" com les claus. La funció anterior es podria reduir a:

```
const sumar = (x, y) => x + y;
```

1.2.2 Àmbit de les variables i de les funcions

Definició d'àmbit

Un **àmbit** (anomenat `scope` en anglès) és una zona del codi en la qual es pot accedir a una variable o funció.

Actualment a JavaScript trobem quatre tipus d'àmbits ben diferenciats:

- Espai global: en aquest espai es troben les funcions i propietats afegides pels navegadors (com `setInterval`, `alert` o `console`). Les variables i funcions declarades a l'espai global són accessibles des de qualsevol punt de l'aplicació.
- Àmbit de bloc: delimitat entre claus com en altres llenguatges com C i Java. Les variables i funcions declarades dintre d'un bloc no són accessibles fora d'aquest. S'aplica a les variables declarades amb `let` i `const`
- Àmbit de funció: aquest és el comportament tradicional de les variables a JavaScript i no es recomana utilitzar-lo. Les variables i funcions declarades dintre d'una altra funció només són accessibles dintre d'aquesta funció. S'aplica a les variables declarades amb `var`.

> Per respectar l'àmbit de bloc cal declarar sempre les variables amb **let** o **const** segons escaigui.
> Per aquest motiu **es desaconsella fer servir var**, ja que no respecta l'àmbit de bloc.

Cal tenir en compte que els àmbits es tracten com una pila, sempre es pot accedir als elements accessibles de l'àmbit superior. Veieu el següent exemple:

```
var cadena1 = "ambit global";

function prova() {
  var cadena2 = "ambit de funció";

  if (true) {
    let cadena3 = "ambit de bloc";

    console.log(cadena1);
    console.log(cadena2);
    console.log(cadena3);
  }
}
prova();
```

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

Com es pot apreciar, dintre del bloc es continua tenint accés a l'àmbit de funció i a l'àmbit global. Si ho observem com una jerarquia en aquest exemple tindríem

`àmbit global > àmbit funció prova > bloc if (true)`. És a dir, des de la funció tenim accés a l'àmbit global i desde el bloc `if` tenim accés a l'àmbit de la funció i a l'àmbit global.

Espai o àmbit global

Hi ha altres contexts on també s'utilitza JavaScript (com Node.js) però no existeix l'objecte `window`.

Als navegadors l'espai global es troba representat per l'objecte `window`, aquest objecte es troba a tots els navegadors i conté totes les propietats i funcions globals.

Qualsevol variable declarada amb `var` en un codi que no es trobi dintre de cap funció o objecte s'afegirà a l'espai global, per exemple:

```
var a = "hola món";

function adeu() {
  var b = "adéu món"
    console.log(‘valor de a (funció): ${a}‘);
    console.log(‘valor de b (funció): ${b}‘);
}

console.log(‘valor de a: ${a}‘);
console.log(‘valor de a (window): ${window.a}‘);

try {
  console.log(‘valor de b: ${b}‘);
} catch (e) {
  console.log(e.message);
}

console.log(‘valor de b: ${window.b}‘);

adeu();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/zYoYPLx?editors=1111.

Com es pot apreciar, el valor de `a` es troba definit a l'objecte `window` i és accessible des de qualsevol punt sense necessitat d'indicar que és una propietat d'aquest objecte. Per altra banda, el valor de `b` només es troba definit dintre de la funció `adeu` i, per consegüent, no es troba a l'espai global, no apareix com a propietat de l'objecte `window` i no és accessible fora de la funció.

Fixeu-vos que hem fet servir `var` per declarar les variables, si fem servir `let` la variable no s'afegeix a l'objecte `window`, encara que continua sent accessible globalment dintre d'aquest codi (així és com funcionen els mòduls), per exemple:

```
var a = "hola món";
let b = "adéu món";

console.log(‘valor de a: ${a}‘);
console.log(‘valor de a (window): ${window.a}‘);

console.log(‘valor de b: ${b}‘);
console.log(‘valor de b (window): ${window.b}‘);
```

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

Hi ha un altre cas en el qual les variables s'afegeixen a l'espai global: si oblidem declarar una variable, aquesta es declara automàticament a l'espai global, inclús si es troba dintre d'una funció, per exemple:

```
function adeu() {
  b = "adéu món"
  console.log(‘valor de b (funció): ${b}‘);
}

try {
  console.log(‘valor de b: ${b}‘);
} catch (e) {
  console.log(e.message);
}

adeu();
console.log(‘valor de b: ${b}‘);
console.log(‘valor de b (window): ${window.b}‘);
```

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

Fixeu-vos que abans de cridar la funció el valor de `b` no està definit, però un cop s'invoca la funció `b` es declara a l'espai global. Cal tenir molt de compte amb aquests errors perquè són difícils de detectar, ja que no es tracta d'un error sintàctic.

De la mateixa manera, quan es declara una funció que no es troba dintre de cap objecte ni altres funcions directament o amb `var`, aquestes s'afegeixen a l'espai global, però com passa amb les variables, si es declaren amb `let` o `const` no s'afegeixen a `window`:

```
function hola1() {}
var hola2 = function () {};
let hola3 = function () {};
const hola4 = () => {};

console.log(‘valor de b (codi): ${hola1}‘);
console.log(‘valor de b (window): ${window.hola1}‘);

console.log(‘valor de b (codi): ${hola2}‘);
console.log(‘valor de b (window): ${window.hola2}‘);

console.log(‘valor de b (codi): ${hola3}‘);
console.log(‘valor de b (window): ${window.hola3}‘);

console.log(‘valor de b (codi): ${hola4}‘);
console.log(‘valor de b (window): ${window.hola4}‘);
```

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

Com es pot apreciar, les funcions declarades amb `let` i `const` no s'han afegit a l'objecte window. Això permet encapsular correctament els mòduls, ja que mitjançant `let` i `const` les nostres variables i funcions es trobaran encapsulades dintre del mòdul encara que carreguem múltiples mòduls que facin servir els mateixos noms de variables; en canvi, si féssim servir `var`, en carregar un nou fitxer que declari variables o funcions amb el mateix nom se sobreescriurien i el programa no funcionaria o ho faria de formes inesperades.

Contaminació del espai global

Crear variables o funcions a l'espai global es considera una contaminació d'aquest espai i fer-ho es considera una mala pràctica. Es recomana utilitzar algun mecanisme com els mòduls o encapsular l'aplicació dintre d'un altre objecte per tal d'evitar-ho.

Àmbit de bloc

Quan es declaren funcions o variables amb `let` i `const`, el seu àmbit queda delimitat per les claus que les envolten o el mòdul on es declarin. És a dir, si es declara el comptador d'un bucle for amb `let` aquest només serà definit dins del bucle:

```
for (let i = 0; i < 3; i++) {
  console.log(‘valor de i (dins del bucle) ${i}‘);
}

try {
  console.log(‘valor de i (fora del bucle): ${i}‘);
} catch (e) {
  console.log(e.message);
}
```

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

Com es pot apreciar, fora del bucle el valor de `i` no es troba definit. El mateix passa amb qualsevol altre tipus de bloc, per exemple amb un bloc `if...else`:

```
if (true) {
  let i = 3;
  const j = 4;
  console.log(‘valor de i (dins del if) ${i}‘);
  console.log(‘valor de j (dins del if) ${j}‘);
}

try {
  console.log(‘valor de i (fora del if): ${i}‘);
} catch (e) {
  console.log(e.message);
}

try {
  console.log(‘valor de j (fora del if): ${j}‘);
} catch (e) {
  console.log(e.message);
}
```

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

Àmbit de mòdul

Es considera un mòdul el codi que es trobi dintre d'una etiqueta `script` amb el tipus `module` assignat com atribut:

Ampliar informació mòduls

Podeu trobar més informació sobre els mòduls al següent enllaç: mzl.la/3ibdoOl.

```
<script type="module">
  // Codi del mòdul
```

```
</script>
```

O quan es carrega un fitxer amb codi JavaScript indicant el tipus `module`:

```
<script type="module" src="fitxer_modul.js"></script>
```

En tots dos casos, el codi queda encapsulat dintre del mòdul, de manera que només són accessibles externament les funcions i variables quan:

- Dins del mòdul d'origen s'han exportat mitjançant la instrucció `export`.
- Al mòdul on es volen utilitzar s'han importat mitjançant la instrucció `import`.

Cal destacar que només es pot utilitzar `import` i `export` dins de mòduls, en cas contrari es produeix un error.

A continuació podeu veure un exemple d'exportació i d'importació de mòduls. Per provar-lo heu de crear un fitxer diferent per cada mòdul i un altre pel codi HTML, tots tres dins del mateix directori.

Codi del mòdul que exporta la funció (*modul.js*):

```
//modul.js
export function suma(a, b) { return a + b}

let cadena = "hola";
```

Codi del mòdul que importa la funció `suma` del mòdul *principal.js*:

```
//principal.js
import { suma } from './modul.js';

console.log(suma(2, 3));

try {
    console.log(`Valor de la cadena? ${cadena}`);
} catch (e) {
    console.log(e.message);
}
```

Codi HTML que carrega el fitxer *principal.js* que s'executa automàticament en carregar-se.

```
<html>
    <script src="principal.js" type="module"></script>
</html>
```

Com podeu comprovar la funció exportada `suma` es pot cridar des del mòdul que la importa, però la variable `cadena` no és accessible.

Àmbit de funció

Cal recordar que es desaconsella declarar les variables fent servir `var`.

Quan es declara una variable utilitzant `var` dintre d'una funció aquesta és accessible dintre de la mateixa funció independentment de si s'ha declarat dintre

d'un bloc com per exemple `for` o `if`:

```
function demostracio() {

  if (true) {
    var i = 3;
    console.log(‘valor de i (dins del if): ${i}‘);
  }

  console.log(‘valor de i (fora del if): ${i}‘);

  for (var j = 0; j<3; j++) {
    console.log(‘valor de j (dins del for): ${j}‘);
  }
  console.log(‘valor de j (fora del for): ${j}‘);

}

demostracio();
```

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

1.2.3 Invocacions

Existeixen maneres diferents d'invocar una funció i cadascuna d'elles té les seves peculiaritats. Per exemple, es pot invocar una funció de la manera habitual: *nomfunció()*. També es pot executar una funció com a mètode d'un objecte o bé es pot invocar una funció indirectament, per exemple, al crear un objecte s'invoca el mètode constructor. Aquestes maneres d'invocació de funcions es troben a tots els llenguatges orientats a objectes, però a JavaScript, a més a més, es poden invocar utilitzant dos mètode addicionals: ***apply()*** i ***call()***.

En totes aquestes maneres d'invocar una funció, a la majoria de llenguatges, el nombre d'arguments i paràmetres ha de ser el mateix. A JavaScript, però, això és diferent. Si hi ha més arguments que paràmetres, els arguments de més no s'assignen als noms de paràmetres i la invocació de la funció és correcta i no dona error.

```
function qualsevol(a,b,c){}
qualsevol(1,2,3,4,5,6); //4,5,6 no s'assignen a cap paràmetre.
```

Si, en canvi, hi ha més paràmetres que arguments, els paràmetres que no tinguin el seu argument corresponent s'estableixen com `undefined`.

```
function qualsevol(a,b,c){}
qualsevol(1); //b,c tenen el valor undefined
```

A JavaScript, a totes les invocacions de les funcions es passen 2 paràmetres extres (es passen en silenci i estan en el seu àmbit): `arguments` i `this`. El paràmetre **arguments** és una col·lecció de tots els paràmetres que s'han passat a la funció i té una propietat anomenada `length` que conté el numero de paràmetres que s'han passat. Els valors dels paràmetres es poden obtenir com si fos un *array*. En canvi,

el paràmetre **this** és el context de la funció. A Java `this` és la instància de la classe en la qual es defineix el mètode. A JavaScript és una mica diferent perquè es defineix segons com s'invoca la funció, és a dir, es defineix quan s'executa no quan es declara la funció.

Invocació com una funció

La invocació *com una funció* és la manera "normal" d'invocar una funció en qualsevol llenguatge.

Per exemple:

En el cas de la declaració de funcions s'acostuma a utilitzar `const` perquè el valor no ha de canviar.

```
function cridam(){};
cridam();
const unaltre = function(){};
unaltre();
```

Quan es declara com una funció sense indicar `let` ni `const`, la funció s'afegeix com a mètode de l'objecte `window`, per consegüent és possible invocar-lo a partir d'aquest objecte:

```
function cridam(){};
window.cridam();
```

En canvi, si provem d'executar la funció declarada com a `const` es produirà un error, ja que no s'afegeix a l'objecte `window`, només és accessible dintre del mòdul:

```
const unaltre = function(){};
window.unaltre();
```

De la mateixa manera es poden executar funcions de fletxa:

```
const saludar = () => "hola";
let text = saludar();
document.write(text);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWQeM?editors=0010.

Invocació com un mètode

La invocació *com un mètode* es produeix quan s'assigna una funció a la propietat d'un objecte. Exemple:

```
//es crea l'objecte anomenat 'o'
let o = {};

//es defineix una propietat anomenada 'nom_metode' i se li assigna una funció.
//aquesta propietat s'ha convertit en un mètode.
o.nom_metode = function(){return "hola";};

//crida al mètode.
let text = o.nom_metode();

document.write(text);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ExKxQBO?editors=0010.

Quan invoquem d'aquesta manera a la funció, l'objecte es converteix en el context de la funció, és a dir, el paràmetre implícit **this** apunta a l'objecte.

L'execució de funcions de fletxa es fa de la mateixa manera, l'única diferència és que la seva declaració és més concisa:

```
let o = {};
o.nom_metode = () => "hola";
let text = o.nom_metode();
document.write(text);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/oNxNqgK?editors=0010.

La paraula clau`this` té en JavaScript un comportament diferent al d'altres llenguatges però en general, el seu valor fa referència al **propietari** de la funció que l'està invocant.
Quan no estem dins d'una estructura definida el propietari d'una funció és sempre el context global. En el cas dels navegadors web, s'ha de recordar que aquest objecte és `window`:

```
console.log(this === window);  // true

function test(){
  console.log(this === window);
}

test(); // true
```

La variable `this`, normalment, s'utilitza per accedir als valors d'un objecte des del propi objecte. Per exemple, penseu en un objecte amb una sèrie de propietats:

```
let obj = {
  nom: 'Ramon',
  cognom: 'Llull',
  naixement: '1232',
  isMallorqui: true
};

document.write(obj.nom); // Ramón
document.write(obj.isMallorqui); // true
```

Suposem ara que necessitem una altra propietat més 'dinàmica' que participi dels valors assignats a qualsevol altra propietat. Per exemple, volem un `nomSencer` que uneixi `nom` i `cognom`. Sembla, a priori, que podríem utilitzar la variable `this`:

```
let obj = {
  nom: 'Ramon',
  cognom: 'Llull',
  naixement: '1232',
  isMallorqui: true,
  nomSencer: this.nom + " " + this.cognom
};
document.write(obj.nom + "<br>"); // Ramon
document.write(obj.isMallorqui + "<br>"); // true
```

```
document.write(obj.nom+ "<br>"); // Ramon
document.write(obj.nomSencer + "<br>"); // Ramon Llull?
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWmNO?editors=0010.

Encara que sembla coherent, quan passem a comprovar-ho veiem que el resultat no és l'esperat:

```
Ramon
true
Ramon
undefined undefined
```

El problema és que `this` no apunta a l'objecte anomenat `obj` sinó que `this` apunta al context global, és a dir, a l'objecte `window`.

Per obtenir el resultat esperat hem d'aplicar un patró d'invocació que modifiqui al propietari des del qual s'invoca el `this`.

En el desenvolupament d'aplicacions modernes, el patró més recurrent és el d'invocació per mètode. Quan una funció és emmagatzemada com a propietat d'un objecte es converteix en el que anomenem mètode.

Quan invoquem a un mètode, `this` fa referència al mateix objecte:

```
let obj = {
  nom: 'Ramon',
  cognom: 'Llull',
  naixement: '1232',
  isMallorqui: true,
  nomSencer: function() { return `${this.nom}  ${this.cognom}`;}
};
document.write(obj.nom + "<br>"); // Ramón
document.write(obj.isMallorqui + "<br>" ); // true
document.write(obj.nom+ "<br>" ); // Ramón
document.write(obj.nomSencer() + "<br>" ); // Ramon Llull?
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOLKaE?editors=0010.

En aquesta ocasió, podem comprovar com `this` apunta al mateix objecte i busca les propietats `nom` i `cognom` en comptes d'anar fins el context global.

Invocació amb els mètodes apply() i call()

JavaScript ens proporciona un mitjà per invocar una funció i especificar de forma explícita qualsevol objecte que vulguem com a context. És a dir, podem decidir quin és el valor del paràmetre `this` quan cridem a una funció.

Els mètodes apply() i call()

Totes les funciones tenen aquest dos mètodes disponibles ja que són objectes de primera classe i poden tenir propietats i mètodes igual que qualsevol altre objecte.

S'aconsegueix amb qualsevol dels dos mètodes que posseeixen les funcions: *apply()* i *call()*.

Paràmetres del mètode *apply()*:

- L'objecte que s'utilitzarà com a context de la funció
- *Array* de valors que s'utilitzaran com a arguments de la funció.

Paràmetres del mètode *call()*:

- L'objecte que s'utilitzarà com a context de la funció
- Llista de valors que s'utilitzaran com a arguments de la funció.

Exemple:

```
function exemple() {
  let total = 0;
  for (let i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }

  this.result = total;
};

let objecte1 = {};
let objecte2 = {};

exemple.apply(objecte1, [1, 2, 3, 4]);
exemple.call(objecte2, 5, 6, 7, 8);

// en aquest punt objecte1.result tindrà el valor 10
// i objecte2.result tindrà el valor 26
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/QWNWmmr?editors=0010.

La funció `exemple` utilitza la paraula reservada `this` per crear una propietat anomenada `total` que conté el resultat de l'execució de la pròpia funció.

Si executem la funció la variable `this` apunta a l'objecte `window`, llavors, la propietat `result` seria una propietat global. Però en utilitzar els mètodes `apply` o `call` canviem l'objecte on apunta la variable `this`.

En el primer cas, amb el mètode `apply`, la variable `this` apunta l'objecte1 (ja que aquest objecte és el primer paràmetre del mètode `apply`).

En el segon cas, la variable `this` apunta l'objecte2. Llavors quan la funció `exemple` executa l'última instrucció: *this.result = result;* està creant una propietat anomenada `result` a l'objecte1 o a l'objecte2, segons sigui el cas, i està guardant el resultat de l'execució del codi anterior.

Canvi de context d'una funció amb el mètode "bind"()

Una altra manera de canviar el context d'execució d'una funció és utilitzar el mètode `bind(context)`. Aquest mètode **crea una nova funció amb el context passat com a paràmetre**, això ens permet utilitzar-lo de dues formes:

- Assignant la funció retornada com una nova funció, això permet executar-la sempre amb el nou context.

- Invocar-la directament, de manera que s'executa només una vegada amb el nou context.

Veieu aquests dos casos en el següent exemple:

```
<html>
<script>
    let persona1 = {
        nom: 'Joan',
        saludar: function () {
            console.log(`Hola!, em dic ${this.nom}`);
        }
    };

    let persona2 = {
        nom: 'Maria',
    };

    // la funció s'executa en el seu context, persona1
    persona1.saludar();

    // la funció s'executa en el context de persona2
    persona1.saludar.bind(persona2)();

    // afegim una nova funció saludar amb el nou context
    persona1.saludarNou = persona1.saludar.bind(persona2);

    // el context per aquesta funció és persona 2
    persona1.saludarNou();

    // comprovem que si canviem les propietats del context, la crida a la funci
        ó continua sent correcte
    persona2.nom = "Rosa";
    persona1.saludarNou();

    // comprovem que saludar continua utilitzant el seu context original
    persona1.saludar();
</script>
</html>
```

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

Com es pot apreciar, quan s'invoca la funció `saludar` amb `bind(persona2)` la funció s'executa en el context de `persona2` però subsegüents invocacions a la funció `saludar` continuant utilitzant `persona1` com a context. Per altra banda quan assignem la funció retornada per `bind`, totes les invocacions a aquesta nova funció es fan utilitzant el context de `persona2`.

Fixeu-vos que encara que canviïn els valors de `persona2` quan es crida a la funció `saludarNou` el resultat continua sent correcte perquè el context és una referència a `persona2` i no una còpia.

1.2.4 Exemple: memoritzar valors calculats a la pròpia funció (Memoize)

La memorització (*memoization*) és el procés de crear una funció que pot recordar els seus valors calculats. Així es pot incrementar considerablement el rendiment evitant càlculs complexos innecessaris que ja s'han calculat.

Per exemple, escrivim una funció que ens digui si un enter positiu és o no primer:

```
function esPrimer(valor) {
  if (!esPrimer.cache) {
    esPrimer.cache = {}; // si no hi ha cache de resultats, la creem
  }

  if (esPrimer.cache[valor] != null) {
    // resultat ja calculat
    return `${valor} és primer (cache)? ${esPrimer.cache[valor]}`;
  }

  // cal calcular-lo perquè no tenim el resultat

  let primer = valor != 1; // 1 no és primer
  // busquem un divisor superior a 2 i inferior a valor

  for (let i = 2; i < valor; i++) {
    if (valor % i == 0) {
      primer = false;
      break;
    }
  }

  esPrimer.cache[valor] = primer;
  return `${valor} és primer (calculat)? ${primer}`;
}
console.log(esPrimer(5));
console.log(esPrimer(5));
console.log(esPrimer.cache);
```

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

La segona vegada que es crida a la funció entra dins del segon `if` perquè el troba en memòria i retorna el resultat directament.

Fixeu-vos que, donat que les funcions també són objectes, hem pogut crear una nova propietat anomenada `cache` directament a la funció. Es pot accedir a aquesta propietat com si es tractés de qualsevol altre objecte, en aquest cas amb `esPrimer.cache`. Per aquest mateix motiu és accessible des de fora de la funció, ja que les propietats a JavaScript són públiques.

1.2.5 Sobrecàrrega de funcions

En altres llenguatges de programació orientats a objectes, per sobrecarregar una funció se sol declarar diferents implementacions de mètodes amb el mateix nom

però amb un conjunt diferent de paràmetres. En JavaScript no es fa d'aquesta manera. En JavaScript se sobrecarreguen les funcions amb una única implementació que modifica el seu comportament mitjançant l'examen del nombre d'arguments que l'han proporcionat.

És fàcil d'imaginar que es podria implementar la sobrecàrrega de funcions utilitzant una estructura del tipus *if-then i else-if*. Però no sempre ho podrem fer.

Exemple d'una funció sobrecarregada de manera monolítica:

```
let persona = {
  calculMatricula = function(){
    switch(arguments.length){
      case 0:
        // fer algo
        break;
      case 1:
        // fer una altre cosa
        break;
      case 2:
        // fer una altre cosa més
        break;
      ... etc ...
    }
  }
}
```

Exemple: tècnica per sobrecarregar funcions

Veurem una tècnica per ens permet crear diverses funcions (aparentment amb el mateix nom, però es diferencia pel número de paràmetres) que poden escriure's com diferents, anònimes i independents i no com un bloc monolític *if-then-else-if*.

Tot això depèn d'una propietat poc coneguda de les funcions: la propietat `length`.

Algunes consideracions inicials a tenir en compte:

- El paràmetre `length` de funció no ha de confondre's amb la propietat `length` del paràmetre arguments.
- El paràmetre `length` de funció ens indica el número total de paràmetres formals amb els quals s'ha declarat la funció.
- La propietat `length` del paràmetre `arguments` ens indica el número total de paràmetres que s'han passat a la funció en el moment de cridar-la.

Exemple:

```
function max(a,b){
   ...
}

max(1,4,5,7,23,234);
```

En aquest cas el paràmetre `length` de la funció `max` és 2 i la propietat `length` del paràmetre `arguments` és 6.

Utilitzarem aquest paràmetre per crear funcions sobrecarregades.

```
function afegirMetode(objecte, nom, funcio){
  let old = objecte[nom];
  objecte[nom] = function(){
      if(funcio.length == arguments.length){
          return funcio.apply(this, arguments);
            }
      else  if (typeof old == 'function'){
          return old.apply(this, arguments);
            }
    };
}


let persona = {};
afegirMetode(persona, "calculMatricula", function(){ //fer algo  });
afegirMetode(persona, "calculMatricula", function(a){ //fer una altre cosa  });
afegirMetode(persona, "calculMatricula", function(a,b){  //fer una altre cosa m
    és  });
```

La funció `afegirMetode` s'utilitza per afegir un mètode a un objecte. Aquesta funció rep tres paràmetres:

- L'objecte al qual volem afegir un mètode
- El nom del mètode que volem afegir
- La funció associada al nom del mètode anterior

La funció `afegirMetode` utilitza el mètode `apply` per afegir a un objecte la pròpia funció `afegirMetode`. Aquesta funció és recursiva ja que tots els mètodes afegits d'aquesta manera tenen disponible la variable `old`, que guarda els mètodes amb els mateixos noms però amb un nombre de paràmetres diferent. Noteu que en aquest cas la variable `this` apunta a la funció `afegirMetode`.

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/MWyWRPa?editors=0010.

1.2.6 Clausures

A JavaScript és possible crear funcions dintre de funcions i això genera un *àmbit lèxic*, és a dir, les funcions internes tenen accés a l'àmbit de la funció externa (per exemple, variables i paràmetres). Aquesta característica de JavaScript ens permet crear *clausures*:

Una clausura és la combinació d'una funció i l'entorn lèxic en el qual aquesta funció és declarada amb les següents característiques:

- Té accés a les variables i funcions definides a la funció interna (àmbit de funció).
- Té accés a totes les variables i funcions declarades en la funció externa, incloent-hi els paràmetres amb què es va invocar la funció (àmbit lèxic).
- Té accés a l'àmbit global.

Veieu un exemple:

```
// funció externa
function inicialitzacio() {
  let nom = 'Joan';
  // funció interna
  function mostrarNom() {
    console.log(nom);
  }
  mostrarNom();
}
inicialitzacio();
```

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

Fixeu-vos que la variable `nom` es declara a la `inicialitzacio` però és accessible des de la funció `mostrarNom` i que aquesta funció és invocada per inicialització.

Com que a JavaScript és possible retornar una funció, podríem retornar la funció `mostrarNom` i aquesta funció continuaria tenint accés a l'àmbit de la funció externa com podeu veure en el següent exemple:

```
// funció externa
function inicialitzacio() {
  let nom = 'Joan';
  // funció interna
  function mostrarNom() {
    console.log(nom);
  }
  return mostrarNom;
}
let mostrar = inicialitzacio();
mostrar();
```

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

Quan s'assigna el valor a la variable `mostrar` s'executa la funció `inicialitzacio`, s'assigna el valor a la variable `nom` i es retorna la funció `mostrarNom`; així, doncs, el valor de la variable `mostrar` és la funció `mostrarNom` amb accés a l'àmbit de `inicialitzacio`.

Fixeu-vos que el valor de `nom` no és accessible des de fora de la funció, per tant es pot considerar que els valors declarats a la funció externa són `privats`.

> L'únic mecanisme que proporciona JavaScript per encapsular valors de manera el seu accés sigui **privat** és mitjançant **clausures**.

Una altra característica remarcable és que l'àmbit es crea en el moment en què s'invoca la funció, això vol dir que si cridem a una mateixa funció amb diferents paràmetres, cada funció retornada continuarà tenint accés a l'àmbit existent quan es va invocar la funció externa:

```
// funció externa
function crearTaula(multiplicador) {
  // funció interna
  function multiplicar() {
    for (let i = 1; i<=10; i++) {
      console.log(`${i} x ${multiplicador} = ${i * multiplicador}`);
    }
  }
  return multiplicar;
}

const taulaDel5 = crearTaula(5);
taulaDel5();

const taulaDel9 = crearTaula(9);
taulaDel9();
taulaDel5();
```

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

Com es pot apreciar, s'ha assignat a les variables `taulaDel5` i `taulaDel9` el retorn de la funció `crearTaula` invocada amb diferents arguments i quan s'invoca a qualsevol d'aquestes funcions el retorn és l'esperat, a l'àmbit de `taulaDel5` el valor de `multiplicador` és 5, mentre que al de `taulaDel9` és 9.

1.2.7 Funcions sense nom. Definició i usos

Les funcions **sense nom**, o anònimes, les fem servir quan volem poder disposar d'elles posteriorment. Per exemple, quan les emmagatzemem en una variable, quan les posem com a mètodes d'objectes o bé quan les utilizem com a *callback* (callback de timeout, callback de controladors d'esdeveniments, etc).

> Un **callback** és un pedaç de codi executable (una funció) que es passa com a paràmetre a un altre codi, que s'espera per executar-lo. La invocació pot ser immediata (síncrona), o pot ocórrer en un moment posterior (asíncrona).

En tots aquests casos, les funcions no necessiten tenir un nom per poder invocar-les.

```
window.onload = function() {alert("1");};

let obj = {print : function() {alert("2");}}
```

```
obj.print();

setTimeout(function(){ alert("3"); }, 500);
```

També podíem haver fet amb el controlador de l'esdeveniment de càrrega de pàgina:

```
function salutacio(){
  document.write("Hola estudiants de l'IOC!");
};

window.onload = salutacio;
```

Però no cal donar-li nom si mai més l'utilitzarem. Realment no necessitem que `salutacio` sigui un mètode de l'objecte `window`. A més a més, podem pensar que el mètode `print` és el nom de la funció anònima que li assignem, però no ho és.

Una funció anònima es pot definir sense assignar-li un nom ni assignar-la a cap variable, però en fer-ho d'aquesta manera no es pot invocar. Per consegüent el codi és vàlid però no és correcte:

```
function () {
  document.write("Hola estudiants de l'IOC!");
}
```

De vegades ens interessa poder posar nom a les funcions anònimes. Normalment, les funcions sense nom o anònimes les utilitzem quan creem els mètodes dels objectes. El mètode ha de tenir un nom, la seva funció no.

```
let persona = {
    cantar : function (){return 1;}
}
console.log(persona.cantar.name);
```

En aquest exemple s'ha declarat un objecte anomenat `persona`, que conté un mètode anomenat `cantar`.

Malgrat que aquest mètode s'ha declarat com una funció anònima, automàticament s'ha assignat el nom de la propietat com a nom de la funció.

Tot i que no és gaire utilitzat, JavaScript permet anomenar una funció assignada a una propietat, en aquest cas el nom de la funció serà l'indicat i no pas el de la propietat:

```
let persona = {
  cantar : function(n){ return n > 1 ? this.cantar(n - 1) + "-fiu" : "fiu"; },
  cantar2 : function xiular(n){ return n > 1 ? xiular(n - 1) + "-fiu" : "fiu";
      }
}
console.log("Cridem al mètode cantar que a la seva vegada crida a la funció
    xiular: persona.cantar(3):");
console.log(persona.cantar(3));
console.log(persona.cantar2(3));
console.log(persona.cantar.name);
console.log(persona.cantar2.name);
```

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

1.2.8 Funcions de fletxa

Les funcions de fletxa proporcionen una manera molt més concisa d'utilitzar les funcions anònimes; d'aquesta manera es poden assignar a variables, mètodes o, també, utilitzar com a *callbacks*:

```
window.onload = () => console.log("1");

let obj = {print: () => console.log("2")};
obj.print();

setTimeout(() => console.log("3"), 500);
```

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

Si el cos de la funció tingués més d'una instrucció, caldria posar-les entre claus.

Utilització de funcions de fletxa

No es recomana utilitzar aquestes funcions quan el cos de la funció sigui llarg, ja que l'avantatge principal és fer el codi més llegible. Si utilitzar una funció de fletxa el codi no ho fa més entenedor, és preferible fer servir la implementació habitual.

Com podeu veure a continuació hi ha casos en què la utilització d'aquest tipus de funció fa el codi més entenedor, ja que mostren només la informació rellevant en funcions molt simples (com `sumar`) o permeten escriure una funció de callback en una sola línia:

```
// Assignat com a funció
const sumar = (a, b) => a + b;

// Utilitzat com a callback
let tick = 0;
setInterval(() => {console.log( tick % 2 == 0 ? "Tic" : "Tac");tick++;}, 1000);
```

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

En els casos en què només hi ha un paràmetre encara es poden simplificar una mica més les funcions de fletxes, ja que no són necessaris els parèntesis:

```
const duplicar = a => a * 2;
console.log(duplicar(2));
```

1.2.9 Funcions immediates

Una funció immediata es basa en el concepte de les clausures.

Exemple de funció immediata:

```
(function(){})()
```

Analitzarem la construcció de la funció ignorant el primer grup dels parèntesis.

```
(instruccions)()
```

Sabem que podem fer la crida d'una funció utilitzant la sintaxi *functionName()*, però en lloc del nom podem utilitzar qualsevol expressió que es refereixi a una de les seves instàncies.

```
let estudiar = function(){instrucions};
result = estudiar();
// o també podem fer:
result = (estudiar)();
```

Això significa que a **(funció)()**, el primer joc de parèntesis és un limitador que tanca una expressió. El segon joc de parèntesis és un operador.

Ara en lloc d'una variable, hi posem la funció anònima directament.

```
(function(){instruccions})();
```

Aquesta funció s'instancia, tot seguit s'executa i, finalment, es descarta. Per exemple, es poden utilitzar aquest tipus de funcions per crear un àmbit temporal que emmagatzemi l'estat:

```
(function(){
  let numclicks = 0;
  document.addEventListener("click", function(){alert(++numclicks);}, false);
})();
```

L'important és observar que es crea una clausura pel controlador que inclou `numclicks`, llavors només ell pot fer referència a aquesta variable. Ningú més podrà modificar el seu valor.

Aquesta és una de les formes d'ús comú de les funcions immediates: com embolcalls simples e independents.

També es poden passar paràmetres a les funcions immediates incloent-los al final. Per exemple:

```
(function(salutacio){alert(salutacio);})("Hola");
```

1.2.10 Les funcions niades

A l'estructura que esdevé al declarar funcions una dintre d'altres s'anomena estructura de funcions niades. Exemple:

```
function estudiar(vegades) {

    function molt(frase) {
        document.write(frase)
    }
    molt('Estudieu més de ${vegades} hores.');

}
estudiar(10);
```

Podem combinar aquesta estructura amb el retorn de les funcions. La funció pare pot escollir una de les funcions internes i retornar-la. Les clausures permeten mantenir la informació que hi ha dintre dels paràmetres o de les variables a les que té accés la funció interna.

```
function estudiar(frase) {

    function molt() {
        document.write(frase)
    }
    return molt;

}

const ioc = estudiar('Estudiant a ´lIOC ´saprèn molt.')

ioc()
```

Podem combinar tot el que hem après sobre les funcions. Podem crear-les anònimes i auto-executables.

```
const estudiar = (function(frase) {
    function molt() {
        alert(frase)
    }

    return molt
})("Estudiant a l'IOC s'aprèn molt.")

estudiar();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/abNzbLr?editors=0010.

1.2.11 Desestructuració i assignació de valors per defecte als paràmetres

Actualment JavaScript permet la desestructuració d'objectes i arrays, això significa que es poden assignar directament els valors d'un array o objecte a un grup de

variables:

```
// Desestructuració d'un array
let colors = ['verd', 'ambar', 'vermell'];
let [pasar, perill, prohibit] = colors;
console.log(pasar, perill, prohibit);

// Desestructuració d'un objecte
let persona = {nom: "Joan", poblacio: "Barcelona"};
// variables amb el mateix nom que les propietats
let {nom, poblacio} = persona;
console.log(nom, poblacio);

// variables amb diferent nom que les propietats
let {nom: a, poblacio: b} = persona;
console.log(a, b);
```

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

Fixeu-vos que a la banda esquerra s'indiquen les variables a les quals s'assignarà el valor i a la dreta l'objecte o array que es desestructurarà.

En el cas dels arrays, cal indicar els noms de les variables entre claudàtors i els valors s'assignaran en ordre (l'índex 0 a la primera variable, l'índex 1 a la segona, l'índex 2 a la tercera, etc.).

Per altra banda, en el cas dels objectes, s'indicaran els noms entre claus i, en cas que els noms de les variables siguin diferents dels noms de les propietats, caldrà indicar-los separats per dos punts, separant els parells de propietat i variable per una coma.

Aquesta mateixa desestructuració es pot aprofitar per passar arguments amb nom a les funcions:

```
function saludar({ nom, poblacio }) {
  console.log(`Hola ${nom} de ${poblacio}`);
}

let joan = {nom: "Joan", poblacio: "Barcelona"};
let maria = {nom: "Maria", poblacio: "Lleida"};

saludar(joan);
saludar(maria);
saludar({nom: "Pere", poblacio: "Tarragona"});
```

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

Fixeu-vos que, en lloc d'indicar un nom de paràmetre, el que hem fet és indicar una desestructuració d'un objecte i, per tant, els paràmetres `nom` i `població` rebran el valor de les propietats corresponent a l'objecte que es passi com argument.

Una altra característica de les desestructuracions és que permeten assignar valors per defecte pels casos en què l'objecte reestructurat no contingut cap valor per assignar:

```
// Desestructuració d'un objecte
let persona = {nom: "Joan"};
```

```
// variables amb el mateix nom que les propietats
let {nom, poblacio = "desconegut"} = persona;
console.log(nom, poblacio);
```

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

Com es pot apreciar, quan l'objecte desestructurat no conté la propietat `poblacio` (com en aquest exemple), el valor assignat és `desconegut`.

Això ens permet crear funcions amb paràmetres que acceptin valors per defecte:

```
function saludar({nom, poblacio = "desconegut"}) {
  console.log(`Hola ${nom} de ${poblacio}`);
}
let maria = {nom: "Maria", poblacio: "Lleida"};
saludar(maria);
saludar({nom: "Pere"});
```

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

Així doncs, quan invoquem a la funció `saludar` amb un objecte que no contingui la propietat `poblacio` s'assigna el valor `desconegut` i, en cas contrari, s'assigna el valor de la propietat.

1.2.12 Propagació i retorn de múltiples valors

Utilització avançada de paràmetres i retorns

Podeu trobar més exemples avançats d'utilització de la desestructuració i la propagació al següent enllaç: https://bit.ly/3ifbZq1.

En alguns casos pot ser necessari que una funció retorni més d'un valor. En aquest cas es pot fer retornat un objecte o un array.

Per exemple, retornant un array:

```
function getCoordenades() {
  let coordenades = [2, 4];
  return coordenades;
}

let novesCoordenades = getCoordenades();
console.log(`Primera coordenada: ${novesCoordenades[0]}`);
console.log(`Segona coordenada: ${novesCoordenades[1]}`);
```

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

O retornant un objecte:

```
function getCoordenades() {
  let coordenades = {x: 2, y: 4};
  return coordenades;
}

let novesCoordenades = getCoordenades();
console.log(`Primera coordenada: ${novesCoordenades.x}`);
console.log(`Segona coordenada: ${novesCoordenades.y}`);
```

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

Com es pot apreciar, en tots dos casos cal accedir a l'element de l'array o la propietat de l'objecte respectivament, però, gràcies a l'operador de propagació (que es representa per tres punts suspensius, ...), és possible desestructurar el retorn:

```
function getCoordenades() {
  let coordenades = {a: 1, x: 2, y: 4, z: 3};
  return {...coordenades};
}

let {x, y} = getCoordenades();
console.log('Primera coordenada: ${x}');
console.log('Segona coordenada: ${y}');
```

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

Fixeu-vos que al retorn es troba entre claus i s'ha fet servir l'operador de propagació. D'aquesta manera en invocar la funció `getCoordenades` s'assigna automàticament el valor de les propietats a les variables `x` i `y` com si es tractés d'una desestructuració.

2. Programació amb col·leccions

Les col·leccions són tipus de dades compostos que ens permeten treballar amb grups de dades en lloc de treballar amb dades individuals. Per exemple:

```
// dades individuals
let persona1 = {nom: "Joan", poblacio: "Barcelona"};
let persona2 = {nom: "Maria", poblacio: "Lleida"};
let persona3 = {nom: "Pere", poblacio: "Tarragona"};
let persona4 = {nom: "Lluisa", poblacio: "Girona"};

// col·lació
let persones = [persona1, persona2, persona3, persona4];
```

Quan s'ha de treballar amb grups de dades no és viable treballar amb variables, ja que això requereix crear una nova variable per a cada valor i es dificulta el processament, ja que s'ha de modificar cada variable individualment.

En canvi, una col·lecció consisteix en un grup de dades, la qual cosa permet afegir dinàmicament a una mateixa col·lecció qualsevol quantitat de dades i processar tota la col·lecció completa a partir d'una única variable (a la qual està assignada la col·lecció).

Ampliant l'exemple anterior, a partir de la variable `persones`, a la quals s'ha assignat un array d'objectes, podem accedir a qualsevol dels elements. Per exemple, es poden recórrer tots els elements:

```
for (let persona of persones) {
  console.log(`${persona.nom} viu a ${persona.poblacio}`);
}
```

En aquest cas es recorren tots els elements de la col·lecció `persones` i s'assigna l'element actual a `persona` (aquest valor s'actualitza en cada iteració del bucle). Seguidament es mostra per la consola el missatge amb el nom i la població de l'element.

Una alternativa fent servir variables individuals podria ser la següent:

```
console.log(`${persona1.nom} viu a ${persona1.poblacio}`);
console.log(`${persona2.nom} viu a ${persona2.poblacio}`);
console.log(`${persona3.nom} viu a ${persona3.poblacio}`);
console.log(`${persona4.nom} viu a ${persona4.poblacio}`);
```

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

Com es pot apreciar, d'aquesta manera es duplica el codi per cada element. És a dir, si afegim 10 persones més, també hauríem d'afegir 10 línies més.

Vegem un altre exemple:

```
// Utilitzant variables individuals
let nota1 = 6;
let nota2 = 4;
let nota3 = 5;
let nota4 = 8;
let nota5 = 7;
let nota6 = 4;
let nota7 = 9;
let nota8 = 8;
let nota9 = 5;
let nota10 = 6;

let mitjana = (nota1 + nota2 + nota3 + nota4 + nota5 + nota6 + nota7 + nota8 +
    nota9 + nota10) / 10;
console.log(‘Nota mitjana: ${mitjana} (variables)‘)

// Utilitzant una col·lecció
let notes = [6, 4, 5, 8, 7, 4, 9, 8, 5, 6];
let acumulat = 0;
for (let nota of notes) {
  acumulat += nota;
}
console.log(‘Nota mitjana: ${acumulat/notes.length} (col·leció)‘);
```

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

Com es pot apreciar, quan treballem amb col·leccions el codi és molt més entenedor i s'eviten repeticions. A més a més, aquest codi és escalable, ja que, si afegim nous valors a la col·lecció `notes` (per exemple, utilitzant un formulari HTML), el codi continuaria processant les dades correctament; en canvi utilitzant variables individuals això no seria possible.

2.1 Col·leccions

Diccionaris i Map/Object

Un diccionari és una col·lecció desordenada que fa servir parelles de clau i valor per accedir als seus elements. Per aquest motiu, es pot considerar que tant Map com Object (quan s'utilitza en aquest context) són diccionaris.

Per treballar amb col·leccions, JavaScript ens proporciona els objectes predefinits Array, Map i Sets:

- **Array**: un array és una llista de valors ordenats on cada valor és **accessible mitjançant un índex**. Aquest índex sempre és un nombre enter igual o superior a 0.
- **Map**: és una col·lecció desordenada que permet associar parells de clau i valor. La clau pot ser qualsevol cosa (incloent-hi funcions, objectes i tipus primitius).
- **Set**: és una col·lecció de valors únics on es pot iterar sobre els elements en ordre d'inserció, s'hi pot afegir elements, se n'hi pot eliminar i es pot comprovar si hi existeixen; no es pot, però, modificar la seva posició ni accedir-hi directament, ja que no existeix ni un índex ni una clau: és referèncien directament els valors.

Altres col·leccions

Existeixen també els objectes WeakSet i WeakMap que són variants de Set i Map amb uns usos molt més concrets, podeu trobar més informació al següent enllaç: mzl.la/3gDAnRR.

Addicionalment, el comportament dels objectes de JavaScript ens permet utilitzar-

los d'una forma similar a un Map, ja que es pot utilitzar el nom de la propietat com a clau i el valor com a valor de l'element.

Veiem un exemple de cada tipus de col·lecció:

```
let exempleArray = ["Verd", "Ambar", "Vermell", "Vermell"];

let exempleSet = new Set();
exempleSet.add("verd");
exempleSet.add("ambar");
exempleSet.add("vermell");
exempleSet.add("vermell");

let exempleMap = new Map();
exempleMap.set("verd", "Passar");
exempleMap.set("ambar", "Compte!");
exempleMap.set("vermell", "Prohibit");
exempleMap.set("vermell", "No passar!");

let exempleObject = {
  verd: "Passar",
  ambar: "Compte!",
  vermell: "Prohibit",
  vermell: "No passar!"
};

console.log(exempleArray, exempleArray.length);
console.log(exempleSet, exempleSet.size);
console.log(exempleMap, exempleMap.size);
console.log(exempleObject, Object.keys(exempleObject).length);
```

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

Com es pot apreciar, l'única col·lecció que permet elements repetits és l'Array, tots els altres tipus es descarten els valors duplicats conservant l'última assignació.

Fixeu-vos que en el cas de l'objecte no existeix una propietat o mètode per obtenir la mida directament, hem hagut d'utilitzar la funció estàtica `Object.keys` per obtenir un array amb totes les propietats i llavors consultar el valor de la propietat `length` d'aquest array.

A continuació podeu veure les diferents maneres d'iterar sobre aquestes col·leccions:

```
console.log("Iteració array: utilitzant l'índex");
for (let i = 0; i < exempleArray.length; i++) {
  console.log(exempleArray[i]);
}

console.log("Iteració array: accedint directament al valor");
for (let element of exempleArray) {
  console.log(element);
}

console.log("Iteració set");
for (let element of exempleSet) {
  console.log(element);
}

console.log("Iteració object");
for (let nom in exempleObject) {
  console.log(exempleObject[nom]);
}
```

```
20
21 console.log("Iteració map");
22 for (let element of exempleMap) {
23   console.log(element);
24 }
25
26 console.log("Iteració map: desesctructurat");
27 for (let [clau, valor] of exempleMap) {
28   console.log(‘Clau: ${clau}, valor: ${valor}‘);
29 }
```

Per iterar sobre els valors d'un array es pot utilitzar un bucle `for` que recorri els índexs des de 0 fins a l'últim valor de l'array (mida de l'array -1) o es pot utilitzar un bucle `for...of` per recórrer directament els valors de l'array.

Utilitzar un tipus de bucle o un altre dependrà de si necessitem utilitzar l'índex o només el valor, o si l'ordre és important. Per exemple, amb un bucle `for` es poden recórrer els elements en ordre invers.

Per iterar sobre els valors d'un map o un set cal utilitzar `for...of`, aquest tipus de bucle funciona només amb col·leccions iterables (com Array, Map i Set) i per aquest motiu no es pot utilitzar amb objectes, ja que els objectes no són iterables.

Com es pot apreciar, quan s'iteren els elements d'un mapa el valor de l'element és el parell clau i valor, per consegüent es poden desestructurar i assignar a variables independents (en aquest exemple anomenades `clau` i `valor`).

Finalment, per iterar sobre les propietats d'un objecte cal utilitzar `for...in`, el nom de la propietat s'assigna a la variable i per accedir al valor hem de fer la consulta utilitzant claudàtors, com si és tractés d'un array.

2.1.1 Arrays

Als arrays, tot i ser un tipus de dades que s'utilitza amb molta freqüència en tots els llenguatges de programació, existeixen diferencies en la seva utilització. Aquestes diferències es mostren amb claredat quan es compara JavaScript amb Java.

A Java, els arrays s'han d'inicialitzar amb la seva mida, en canvi, a JavaScript la mida d'un array pot variar en qualsevol moment. Una altre diferència és que els arrays de JavaScript poden emmagatzemar elements de qualsevol tipus (Integet, Boolean, String, etc.) i poden estar emmagatzemats en posicions no consecutives; de fet, poden tenir com a índex d'accés un valor no numèric.

Podeu trobar una llista exhaustiva amb tots els mètodes dels arrays en el següent enllaç: mzl.la/33Dyr8k.

Un programador en JavaScript ha de conèixer totes les singularitats que aporta aquest llenguatge quan fa ús d'aquest tipus de dades. Conèixer aquestes singularitats és bàsic per desenvolupar programes simples i eficients, per això, un bon programador ha de conèixer les següents propietats i mètodes dels arrays (i un que retornen un array: `split`):

- `length`: nombre d'elements que conté l'array.

- `concat(nouarray1, nouarray2, ...)`: crea un nou array amb els elements de dos o més arrays diferents.
- `push(element1, element2, ...)`: agrega un o més elements al final de l'array.
- `unshift(element1, element2, ...`: afegeix un o més elements a l'inici de l'array.
- `pop()`: elimina l'últim element de l'array i el retorna.
- `shift()`: elimina el primer element de l'array i el retorna.
- `sort([funció])`: ordena alfabèticament els elements d'un array. Es pot passar una funció com argument per realitzar ordenacions avançades.
- `split(separador[, limit])`: converteix una cadena de text en un array de cadenes de text.
- `join([separador])`: uneix tots els elements d'un array en una cadena de text utilitzant un caràcter d'unió.
- `reverse()`: modifica un array col·locant els seus elements en l'ordre invers a la seva posició original.
- `indexOf(elementAcercar[, posicioInical])`: retorna la posició de l'array on es troba la primera ocurrència del valor cercat.
- `lastIndexOf(elementAcercar[, posici´ó inicial]`: retorna la posició de l'array on es troba l'última ocurrència del valor cercat.
- `every(funció)`: executa la funció passada com argument i comprova que el resultat de la funció aplicada a tots els valors de l'array sigui cert.
- `some(funció)`: executa la funció passada com argument i comprova que almenys un dels resultats de la funció aplicada als elements de l'array sigui cert.
- `filter(funció)`: executa la funció per a tots els elements de l'array i retorna un nou array amb tots els elements als quals la funció hagi retornat cert.
- `slice(inici, final)`: retorna un nou array amb els valors que es troben des de l'índex inici fins a l'índex final.
- `splice(inici, comptadorEliminar[, element1, element2, ...]`: afegeix i elimina elements a partir d'una posició de l'array.
- `fill(valor [, posInicial, posFinal])`: omple un array amb el valor indicat; opcionalment es pot indicar una posició d'inici i una de final.
- `includes(elementCercat[, desdePosicio])`: retorna cert si l'array inclou el valor passat com argument.
- `forEach(funció)`: itera sobre tots els elements de l'array i executa la funció passada com argument per a tots els elements de l'array.

- `map(funció)`: executa la funció sobre tots els elements de l'array i retorna un nou array amb els resultats.

A banda d'aquests mètodes, l'objecte Array inclou també les següents funcions estàtiques:

- `Array.from(arrayOIterable[, funció])` : genera un nou array a partir d'un altre array o objecte iterable (per exemple, un *set*). Opcionalment es pot afegir una funció per processar cada element.
- `Array.isArray()`: retorna cert si l'element passat com a argument és un array.

2.1.2 Maps

Els mapes ens permeten crear col·leccions desordenades d'elements de manera que podem accedir als valors directament utilitzant una clau.

Al contrari del que passa amb els arrays o els objectes no es pot accedir directament als elements utilitzant claudàtors, ja que en utilitzar la sintaxi de claudàtors el que es modifica o consulta són les propietats de l'objecte i no pas el contingut del mapa. Cal utilitzar els mètodes `set` i `get` per assignar i recuperar els valors respectivament.

```
let exempleMap = new Map();
exempleMap.set("verd", "Passar");
exempleMap.set("ambar", "Compte!");
exempleMap.set("vermell", "Prohibit");
exempleMap["groc"] = "desconegut";

// verd no és una propietat de l'objecte, és un valor del mapa
console.log(exempleMap["verd"]);
// groc és una propietat de l'objecte
console.log(exempleMap["groc"]);

// verd és un element del mapa
console.log(exempleMap.get("verd"));
// groc no és un element del mapa
console.log(exempleMap.get("groc"));
```

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

Els mapes es poden manipular utilitzant les següents propietats i mètodes:

- `size`: mida del mapa.
- `clear()`: buida el contingut del mapa.
- `delete(clau)`: elimina l'element del mapa.
- `forEach(funció)`: recorre el mapa i executa la funció per a cada element.

- `get(clau)`: retorna el valor associat a la clau passada com argument.
- `has(clau)`: retorna cert si el mapa conté la clau passada com argument.
- `set(clau, valor)`: assigna un valor a una clau.

Una diferència important amb la utilització d'objectes com a diccionaris és que les claus d'un mapa poden ser qualsevol mena d'objecte incloent funcions i tipus primitius, mentre que el nom de les propietats d'un objecte només poden ser cadenes de text (encara que aquests noms no es trobin entre cometes es continuen considerant cadenes de text.)

2.1.3 Sets

Quan no és necessari que la col·lecció sigui ordenada i aquesta no ha d'admetre elements duplicats, cal considerar utilitzar un *set* en lloc d'un array.

Hi ha dos avantatges importants quan s'utilitza un set en lloc d'un array:

- Eliminar arrays per valor és molt lent: array.splice(array.indexOf(valor)), en canvi eliminar-los d'un set és immediat: set.delete(valor).
- No s'ha de controlar que hi hagin duplicats, ja que els valors d'un set sempre són únics.

Els sets es poden manipular utilitzant les següents propietats i mètodes:

- `size`: mida del set.
- `clear()`: buida el contingut del set.
- `delete(valor)`: elimina l'element del set.
- `forEach(funció)`: recorre el set i executa la funció per a cada element.
- `has(valor)`: retorna cert si el mapa conté la clau passada com argument.

Fixeu-vos que al contrari del que passa amb els mapes, no existeix un mètode `get`, ja que un set només conté una llista de valors i només podem comprovar si el set conté un valor o no mitjançant el mètode `has`.

2.2 Gestió de l'estoc d'una botiga

Una bona manera de veure com s'utilitzen les funcionalitats de les col·leccions és a través de l'ús que se'n fa en un exemple concret, com pot ser la gestió de l'estoc d'una botiga.

Es vol tenir enregistrat l'estoc dels productes d'una botiga. De moment, és una botiga petita i només tenen deu productes, però es venen molt. Per això hem de controlar l'estoc de cadascun d'ells per tal que sempre els tinguem disponibles.

El botiguer ens ha demanat un llistat amb tot l'estoc dels seus productes.

I aquesta ha estat la nostra implementació:

Codi HTML i CSS dels exemples

El codi HTML i CSS utilitzat per aquests exemples es pot trobar als enllaços a CodePen.

```
let productes = new Array(10);
productes.fill(50)

function veureEstoc(){
    let llistat = "";
    for(let i = 0; i < productes.length; i++){
        llistat += '<li>El producte número ${i} té ${productes[i]} unitats';
    }
    document.getElementById("llistat").innerHTML=llistat;
}

veureEstoc();
```

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

Si observeu el codi anterior s'ha utilitzat un array per guardar l'estoc de cada producte. Cada posició de l'array `productes` correspon a un dels productes de la botiga. L'array conté el número d'unitats que hi ha de cadascun d'ells. Inicialment, l'array s'omple

amb el número 50.

Per crear un array s'ha utilitzat la instrucció:

```
let productes = new Array(10);
```

Amb aquesta instrucció es crea un array inicialitzat amb 10 posicions, però també ho podíem haver assignat literalment:

```
let productes = [];
```

Quina diferència existeix entre les dues maneres de crear un array?

A priori, sembla que són iguals però no, mireu el següent exemple:

```
let a = [],           // aquests són iguals
    b = new Array(),  // a i b son arrays amb longitud 0

    c = ['foo', 'bar'],          // aquests també són iguals
    d = new Array('foo', 'bar'), // c i d són arrays amb dos strings

    // Aquí hi ha la diferència:
    e = new Array(3),   // e.length == 3, e[0] == undefined
    f = [3]             // f.length == 1, f[0] == 3
;
```

> La diferència entre crear un array amb el seu constructor o crear-lo de manera literal és que si s'utilitza el constructor es pot definir una mida inicial a l'array.

Podem definir, amb el constructor, un array d'una mida concreta. En canvi, si utilitzem l'assignació literal `[]` per crear arrays, no podem definir la mida inicial. Si afegim en un array literal un número, aquest serà el primer element de l'array i no la seva mida.

La sintaxi és la següent:

```
[element0, element1, ..., elementN]
 new Array(element0, element1[, ...[, elementN]])
 new Array(mida)
```

L'array s'inicialitza amb els elements donats excepte si només hi ha un element. En aquests cas, aquest element simbolitza la mida de l'array només si s'utilitza la creació de l'array amb la paraula reservada `new`.

Un cop creat l'array l'omplim amb el valor 50:

```
productes.fill(50)
```

Amb el mètode `fill` s'assigna el valor 50 a tots els elements de l'array. Cal destacar que si l'array és buit, invocar aquest mètode no tindrà cap efecte.

Si en lloc de fer servir un valor estàtic volguéssim generar els valors aleatòriament, ho podríem fer utilitzant la instrucció `map` per crear un nou array a partir d'una funció i assignant-lo a l'array `productes`:

```
productes = productes.map(() => Math.floor(Math.random() * 100));
```

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

Fixeu-vos que al mètode `map` se li ha passat com a argument una funció de fletxa que retorna un nombre aleatori entre 0 i 99:

```
() => Math.floor(Math.random() * 100)
```

És important recordar que la funció `map` només s'aplica als valors assignats, ja sigui mitjançant el mètode `fill` o assignant els valors manualment.

La funció `veureEstoc()`, recorre l'array `productes` i mostra per pantalla la informació de cadascun d'ells dins de l'etiqueta amb identificador `llistat` (al codi HTML).

Només queda veure el funcionament de la funció `veureEstoc`:

```
function veureEstoc(){
    let llistat = "";
    for(let i = 0; i < productes.length; i++){
        llistat += '<li>El producte número ${i} té ${productes[i]} unitats';
    }
    document.getElementById("llistat").innerHTML=llistat;
}
```

Aquesta funció la invoquem directament des de JavaScript i és l'encarregada de generar el llistat que es visualitza a la pantalla.

Tot i que sabem que només hi ha 10 productes, no s'ha utilitzat aquest número com a condició de control. És molt més útil accedir a la propietat `length` de l'array.

> La propietat `length` d'un array ens dóna el nombre d'elements que conté.

Veiem un altre exemple de creació d'un array i accés a la seva mida:

```
let vocals = ["a", "i", "i", "o", "o"];
console.log(‘Nombre de vocals: ${vocals.length}‘); // Nombre de vocals: 5
```

Tornem a l'exemple. La funció `veureEstoc()` utilitza una variable local anomenada `llistat` que es va omplint poc a poc a mesura que el bucle va donant voltes. A cada volta del bucle la variable recull les dades del producte en curs. Finalment, quan s'arriba al final de l'array, la variable conté la informació desitjada per l'usuari.

```
llistat += ‘<li>El producte número ${i} té ${productes[i]} unitats‘;
```

Només falta mostrar la informació a l'usuari. Utilitzem l'objecte `document` de JavaScript per accedir al codi HTML de la pàgina web.

Cerquem l'etiqueta identificada amb `id=llistat` i l'assignem el nou codi HTML que volem que tingui. Així, podem canviar el codi HTML de la pàgina i donar la informació que desitgem.

```
document.getElementById("llistat").innerHTML=llistat;
```

2.3 Ampliació del número de productes de la botiga

Volem ampliar el número de productes de la botiga. Cada vegada que pitgem un botó, s'ha d'ampliar el número de productes. Tant el número de productes com l'estoc han de ser aleatoris.

```
function ampliarEstoc(){
  let nouEstoc = demanarProductes();
  productes.push(...nouEstoc);
  veureEstoc();
}

function demanarProductes(num){
  if(arguments.length === 0){
    num = Math.floor((Math.random() * 5) + 1);
    }

  let nousProductes = new Array(num);
  nousProductes.fill();

  return nousProductes.map(() => Math.floor((Math.random() * 100)))
}
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/abNzmvO?editors=1010.

En aquest exemple s'han afegit dos botons al codi HTML que criden a les funcions `veureEstoc` i `ampliarEstoc`. Si observeu el codi HTML dels botons, veureu que l'event `onclick` dels botons ten assignada les crides a les seves respectives funcions.

```
<button onclick="veureEstoc()">Veure estoc</button>
<button onclick="ampliarEstoc()">Ampliar estoc</button>
```

Observeu que s'han creat dues noves funcions: `ampliarEstoc()` i `demanarProductes(num)`. Aquesta última funció s'ha creat per fer refacció del codi.

> **Fer refacció** (*Code Refactoring*) vol dir, en programació, reestructurar el codi per tal de millorar-lo i reduir la seva complexitat sense canviar les funcionalitats que hi havien fins al moment.

No s'han canviat les funcionalitats, sinó que s'han aprofitat millor. De fet, la funció `demanarProductes(num)` s'utilitza per dues coses: inicialitzar l'array productes i, a més a més, per demanar productes nous. Fem un cop d'ull a la funció.

```
function demanarProductes(num){
  if(arguments.length === 0){
    num = Math.floor((Math.random() * 5) + 1);
  }

  let nousProductes = new Array(num);
  nousProductes.fill();

  return nousProductes.map(() => Math.floor((Math.random() * 100)))
}
```

La funció admet un paràmetre anomenat `num`. Aquest paràmetre és opcional. Si s'envia el paràmetre crearà un array amb el nombre de productes demanats. En canvi, si no s'envia el número de productes que es crearan serà aleatori.

Aquesta funcionalitat s'aconsegueix preguntant el número de paràmetres enviats a la funció. En concret, amb `argument.length` podem saber exactament el número de paràmetres enviats. Si la mida és `1` significa que la variable `num` està plena i ens demanen un `array` amb aquesta mida. En canvi, si la mida és `0` vol dir que ens demanen un array amb una mida indeterminada. Aquesta mida l'haurem de calcular aleatòriament.

La resta de la funció ja és coneguda. Només s'ha canviat el punt del codi on es realitza la inicialització i la mida inicial, en lloc de crear l'array directament es fa la crida a `demanarProductes` i s'assigna com a array inicial el retorn d'aquesta funció.

Llavors, per inicialitzar l'*array productes* es fa una crida a aquesta funció.

```
productes = demanarProductes(10);
```

Com que inicialment es volen 10 productes, s'envia la mida de l'array desitjat a la funció. La funció crearà un array temporal que s'assignarà a la variable `productes`.

La funció `demanarProductes(num)` ha estat creada per utilitzar-la també en el cas que es vulgui ampliar l'oferta de productes de la botiga.

```
function ampliarEstoc(){
  let nouEstoc = demanarProductes();
  productes.push(...nouEstoc);
  veureEstoc();
}}
```

Per ampliar aleatòriament tant el nombre de productes com l'estoc utilitzarem la funció `demanarProductes()` sense enviar cap paràmetre. Si executem la funció sense paràmetres, ens retornarà un array amb les característiques desitjades.

Una vegada tenim l'array amb els nous productes s'han d'afegir a l'array `productes`.

Per afegir els elements retornats hem fet servir l'operador de propagació `...nouEstoc` i els hem afegit mitjançant el mètode `push`.

Cal recordar que aquest mètode afegeix un o més elements al final d'un array i que aquests elements han d'estar separats per comes. És per aquest motiu que es fa servir l'operador de propagació, ja que fa la conversió d'un array a una llista de valors separats per coma.

> El mètode **push** afegeix un o més elements al final de l'array. L'array original es modifica i augmenta la seva longitud. També és possible afegir els elements a l'inici de l'array amb el mètode **unshift**.

La sintaxi del mètode `push` és la següent:

```
array.push(element1, ..., elementN)
```

Tots els elements que es passen com a paràmetre del mètode `push` s'afegeixen al final de l'array.

En cas d'haver invocat el mètode passant com argument `nouEstoc`, el resultat hauria estat diferent, ja que, en lloc d'afegir tots els valors un a un, s'hauria afegit tot l'array com un únic element.

Fixeu-vos en els dos casos representats en els següents exemples: Primer exemple:

```
let a = [1, 2, 3];
let b = [4, 5, 6];

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

Amb aquest cas el contingut de l'array seria el següent:

```
1
2
3
[4, 5, 6]
```

Com es pot apreciar el quart element de l'array és un array amb 3 valors. Segon exemple:

```
let a = [1, 2, 3];
let b = [4, 5, 6];

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

La sortida en aquest cas seria la següent:

```
1
2
3
4
5
6
```

Com es pot apreciar, en aquest cas el resultat és el desitjat, s'han afegit els continguts de l'array `b` a l'array `a` que ara té una mida de 6.

Alternativament es podria haver fet servir la funció `concat` per crear un array nou afegint el nou estoc al final de l'array `productes`. Aquest nou array creat s'ha assignat a la variable `productes`, ja que la funció `concat` no altera l'array sobre el qual es crida:

```
productes = productes.concat(nouEstoc);
```

Podeu provar l'exemple amb `concat` a: https://codepen.io/ioc-daw-m06/pen/dyMPpNv?editors=10.

> La funció **concat** es fa servir per crear un nou array amb els elements de dos arrays diferents.

Cal tenir en compte que en aquest cas la utilització del mètode `concat` és menys eficient que la utilització de `push`, ja que el primer crea un array addicional mentre que el segon només afegeix els valors d'un array a un altre.

La sintaxi de la funció és la següent:

```
let arrayNou = arrayAntic.concat(valor1[, valor2[, ...[, valorN]]])
```

La funció `concat` retorna una còpia dels valors de l'array `arrayAntic`, afegint-hi els valors passats com argument, que poden ser valors individuals o altres arrays.

Un altre exemple de l'ús de la funció `concat` és:

```
let array1 = [1, 2, 3];
array2 = array1.concat(4, 5, 6); // array2 = [1, 2, 3, 4, 5, 6]
array3 = array1.concat([4, 5, 6]); // array3 = [1, 2, 3, 4, 5, 6]
```

Veiem, que amb la funció `concat` podem concatenar arrays o valors individuals. Utilitzarem la que millor ens convingui segons el tipus de dades que tinguem.

2.4 Rànquing amb els productes més venuts

En aquest apartat volem afegir unes quantes funcionalitats addicionals per fer la botiga més interactiva. La idea és que es puguin comprar i vendre productes i poder tenir un rànquing ordenat dels productes més venuts.

En concret, les funcionalitats que volem tenir són:

- **Funcionalitats del botiguer**
  - Afegir 1 producte: afegir només un producte al catàleg de productes disponibles a la venda. Aquest producte tindrà un nombre d'estoc aleatori.
  - Eliminar 1 producte: eliminar l'últim producte afegit al catàleg de productes. Elimina tot l'estoc disponible del producte.
  - Ordenar els productes segons l'estoc que queda: ordena els productes segons el seu estoc. El número de producte associat a l'estoc pot canviar.
- **Funcionalitats del comprador**
  - Comprar un producte: el comprador pot comprar un producte dels disponibles en el catàleg de productes. Cada vegada que es compra un producte es reordena i es visualitza el rànquing de productes més venuts.

```
let productes = [];
let vendes = new Map();

function veureEstoc() {
  let llistat = "";
  for (let i in productes) {
    llistat += '<li>El producte ${i} té ${productes[i]} unitats';
  }
  document.getElementById("llistat").innerHTML = llistat;
}

function ampliarEstoc() {
  let nouEstoc = demanarProductes();
  productes.push(...nouEstoc);
  veureEstoc();
}

function demanarProductes(num) {
  if (arguments.length === 0) {
    num = Math.floor(Math.random() * 5 + 1);
  }

  let nousProductes = new Array(num);
  nousProductes.fill();

```

```
  return nousProductes.map(() => Math.floor(Math.random() * 100));
}

function comprar(numproducte) {
  if (productes[numproducte] !== undefined && productes[numproducte] > 0) {
    let quantitat = 0
    if (vendes.has(numproducte)) {
      quantitat = vendes.get(numproducte);
    }
    quantitat++;
    vendes.set(numproducte, quantitat);
    productes[numproducte]--;

    ranquing();
    veureEstoc();
  }
}

function nouProducte() {
  productes.push(Math.floor(Math.random() * 100));
  veureEstoc();
}

function eliminarProducte() {
  productes.pop();
  veureEstoc();
}

function ordenarProductes() {
  productes.sort((a, b) => a - b);
  veureEstoc();
}

// funció per ordenar el producte segons les unitats venudes
function sortProducte(a, b) {
  const aa = a.split("-")[0];
  const bb = b.split("-")[0];
  return aa - bb;
}

function ranquing() {
  let aux = [];
  // array auxiliar per no perdre el numproducte després de la ordenació
  // s'afegeix el numproducte juntament amb les unitats venudes del producte
  // com a valor del nou array
  // numproducte: Numero del producte venut
  // vendes.get(numproducte): unitats venudes del producte numproducte
  for (let [numproducte, quantitat] of vendes) {
    aux.push(quantitat + "-" + numproducte);
  }
  // ordenar els productes segons les unitats venudes
  // s'ha creat una funció d'ordenació per separar les unitats venudes
  // del numproducte.
  aux.sort(sortProducte).reverse();
  // llistar els productes ordenats
  let llistat = "";
  for (let valor of aux) {
    // Tallem la cadena pel simbol d'unió '-'.
    // a la posició [0] el numero d'unitats venudes del producte
    // a la posició [1] tenim el numero de producte
    let [stockProducte, codiProducte] = valor.split("-");

    llistat += '<li>´sha venut ${stockProducte} unitats del producte ${
        codiProducte}</li>';
  }

  document.getElementById("venuts").innerHTML = llistat;
}

productes = demanarProductes(10);
```

```
95 veureEstoc();
```

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

Totes aquestes funcionalitats s'han aconseguit utilitzant funcions del llenguatge JavaScript que treballen amb arrays i modifiquen la informació que contenen.

2.4.1 Afegir un producte

Per afegir un producte al catàleg de productes s'ha utilitzat el codi següent:

```
1 function nouProducte() {
2     productes.push(Math.floor((Math.random() * 100)));
3     veureEstoc();
4 }
```

I el codi HTML associat a aquesta funció és el següent:

```
1 <button onclick="nouProducte()" >Afegir 1 producte</button>
```

Com podeu veure, en afegir un nou element amb aquest mètode només s'ha de calcular l'estoc d'aquest producte ja que aquest s'afegeix al final de l'array i, per tant, el número de producte es calcularà automàticament. Es fa la crida a la funció `veureEstoc` per fer més interactiu el programa, així es veu al moment que s'ha actualitzat els productes del catàleg.

La sintaxi de la funció `unshift` és la següent:

```
1 array.unshift([element1[, ...[, elementN]]])
```

Tots els elements que es passen com a paràmetre de la funció `unshift` s'afegeixen al principi de l'array.

Veiem un exemple d'utilització de `unshift`:

```
1 let productes = ["Llegums", "Tomàquet", "Ceba", "patata"];
2 let nousProductes = ["all", "pebrot"];
3 let numProductes = productes.unshift("Pastanaga", ...nousProductes);
4 console.log(productes);
5 console.log(numProductes);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/yLOyrpW?editors=1111.

Com es pot apreciar, es pot barrejar la inserció d'elements individuals (`Pastanaga`) i els elements propagats (`...nousProductes`).

Els mètodes `push` o `unshift` retornen la nova longitud de l'array i, si ens interessa, la podem emmagatzemar en una variable. En aquest cas, la variable utilitzada es diu `numproductes`. </newcontent>

2.4.2 Eliminar un producte

Per eliminar un producte del catàleg de productes s'ha utilitzat el següent codi:

```
function eliminarProducte() {
  productes.pop();
  veureEstoc();
}
```

I el codi HTML associat a aquesta funció és el següent:

```
<button onclick="delProducte()" >Eliminar 1 producte</button>
```

La sintaxi del mètode `pop` és la següent:

```
let element = array.pop()
```

El mètode `pop` esborra l'últim element de l'array però el retorna per poder utilitzar-lo.

> Per eliminar un nou producte utilitzem el mètode **pop**. Aquest mètode elimina l'últim element de l'array. L'array original es modifica i decreix la seva longitud en 1 element. També és possible eliminar el primer element de l'array si utilitzem el mètode **shift** , en comptes d'utilitzar el mètode `pop`.

Com es pot apreciar, només cal invocar el mètode `pop` de l'array per eliminar l'últim element i seguidament invoquem la funció `veureEstoc` per refrescar les dades al document HTML.

<newcontent> Veiem ara el mateix exemple però utilitzant `shift`:

```
function delProducte() {
  productes.shift();
  veureEstoc();
}
```

La sintaxi del mètode `shift` és la següent:

```
let element = array.shift()
```

El mètode `shift` elimina el primer element de l'array i mou tots els altres per tal que l'índex de l'array continuï començant per 0. L'element eliminat de l'array es retorna per poder utilitza-lo en el programa si fos necessari.

Tant si s'utilitza el mètode `pop` com si s'utilitza el mètode `shift` es pot recuperar l'element extret i guardar-lo en una variable. Per exemple:

```
function delProducte() {
  let produteEliminat = productes.pop();
  veureEstoc();
}
```

Tot i que en aquest cas no fem res amb el valor retornat, us trobareu casos en els quals serà interessant portar a terme alguna acció sobre l'element eliminat o caldrà retornar-lo. Per exemple, per notificar a l'usuari quin element ha estat eliminat.

2.4.3 Ordenar productes segons l'estoc

Aquesta funcionalitat permet ordenar els productes segons l'estoc que tenen. Els productes amb poc estoc seran els primers. El codi associat a aquesta funcionalitat és el següent:

```
function ordenarProductes() {
  productes.sort((a, b) => a - b);
  veureEstoc();
}
```

El codi HTML associat a aquesta funció és el següent:

```
<button onclick="ordenarProductes()" >Ordenar Productes segons l'estoc que
    queda</button>
```

En utilitzar la funció `ordenarProductes` ha d'aparèixer un llistat semblant a aquest:

```
Llistat de productes:

El producte 0 té 6 unitats
El producte 1 té 11 unitats
El producte 2 té 35 unitats
El producte 3 té 35 unitats
El producte 4 té 51 unitats
El producte 5 té 58 unitats
El producte 6 té 59 unitats
El producte 7 té 70 unitats
El producte 8 té 73 unitats
El producte 9 té 98 unitats
```

Si ens hi fixem, per ordenar només necessitem el mètode `sort` que ens proporciona la biblioteca JavaScript.

> El mètode `sort`,per defecte, ordena alfabèticament, és a dir, com si tot fos una *string*.

La sintaxi del mètode `sort` és la següent:

```
array.sort([funcioComparacio])
```

El mètode `sort` permet ordenar els elements de l'array. El comportament per defecte del mètode és ordenar els elements de l'array utilitzant el codi *Unicode* dels caràcters de l'array a ordenar. Si es vol canviar aquest comportament existeix la possibilitat d'enviar una funció d'ordenació com a paràmetre.

Per exemple, per ordenar números no ens interessa aquest comportament: si els números s'ordenen com si fossin *strings*, "25" seria major que "100", ja que "2" és més gran que "1".

Per solucionar aquest inconvenient, el mètode `sort` permet canviar el seu comportament si li proporcionem la funció que ha d'utilitzar per fer l'ordenació.

Com es pot apreciar, en aquest cas s'ha passat com argument del mètode `sort` una funció de fletxa (aquest és un cas d'ús molt habitual per les funcions de fletxa):

```
(a, b) => a - b
```

Quan la funció `sort` ha de comparar dos valors per determinar quin és més gran, envia aquests dos valors a la funció de comparació i aquesta ha de retornar un valor negatiu, un zero o un valor positiu, depenent dels paràmetres.

Per exemple, si es compara el número 40 i el 100, la funció `sort` crida a la funció de comparació amb els paràmetres (40, 100). La funció de comparació calcula `40 - 100` i retorna `-60` (un número negatiu).

El mètode `sort` determina que 40 és més petit que 100 i ordenarà els valors segons aquesta informació.

2.4.4 Comprar un producte

La funcionalitat *comprar producte* és indispensable si volem fer un rànquing dels productes més venuts. La funcionalitat que es vol implementar és que el comprador informi sobre el producte desitjat i que decreixi el número d'estoc del producte. En aquest cas, el comprador ha de dir el número de producte que vol comprar.

El codi JavaScript associat a aquesta funcionalitat el podeu veure en el codi següent:

```
function comprar(numproducte) {
  if (productes[numproducte] !== undefined && productes[numproducte] > 0) {
    let quantitat = 0
    if (vendes.has(numproducte)) {
      quantitat = vendes.get(numproducte);
    }
    quantitat++;
    vendes.set(numproducte, quantitat);
    productes[numproducte]--;

    ranquing();
    veureEstoc();
  }
}
```

A continuació es mostra la crida HTML des del botó comprar producte. Des de la pàgina HTML es fa el pas de paràmetre amb el valor que l'usuari introdueixi en el *textbox*. Aquest valor ha de coincidir amb un número de producte vàlid.

```
<input type="text" id="numProducte" size="10"/>
<button onclick="comprar(document.getElementById('numProducte').value);" >
    Comprar Producte</button>
```

L'*event* `onclick` del botó fa la crida a la funció JavaScript `comprar` però aquesta funció necessita del número de producte que es vol comprar. Així, des de la pàgina web s'accedeix a l'element `numProducte`, que és el *textbox* que utilitza l'usuari per informar del producte que vol comprar, i s'agafa el valor. Aquest valor s'envia per paràmetre a la funció `comprar`.

Una vegada es realitza la crida a la funció `comprar` aquesta ha de realitzar les accions següents:

- Comprovar que l'estoc del producte s'hagi definit i aquest sigui superior a 0: `if (productes[numproducte] !== undefined && productes[numproducte] > 0)`.
- Comprovar si el producte ja es troba al mapa `vendes`; si és així, recuperem el valor associat a la clau `numproducte` i, en cas contrari, assignem 0 a la quantitat: `if (vendes.has(numproducte))`.
- Incrementar en un la quantitat venuda.
- Assignar el nou valor a `vendes`: `vendes.set(numproducte, quantitat);`
- Reduïm el nombre de productes en estoc en un: `productes[numproducte]--;`.
- En qualsevol cas, es mostra el rànquing i s'actualitza el llistat de productes per pantalla.

Fixeu-vos que `vendes` és un *map* i no pas un array, així que es tracta d'una col·leccio desordenada i s'afegeixen els elements indicant la clau i el valor amb el mètode `set`, es recuperan amb `get` i es comprova si la clau existeix amb `has`.

En aquest cas hem optat per fer servir un diccionari de dades perquè l'ordre en què s'afegeixen els elements no és important i, en cas que s'eliminés algun valor del mapa, no volem que les claus canviïn (cosa que passa quan s'elimina un element d'un array, ja que els índexs de tots els elements posteriors s'actualitzen per continuar sent consecutius).

La clau de `vendes` ha de coincidir amb l'índex de `productes` i, per consegüent, és més segur fer servir un mapa.

2.4.5 Mostrar el rànquing dels productes més venuts

La funció que calcula el rànquing dels productes més venuts es basa en un array on hi ha emmagatzemats els productes que ha comprat l'usuari. Aquest array es va omplint a mesura que l'usuari utilitza la funció *compra(numProducte)*.

El codi per obtenir un rànquing ordenat de major a menor és el següent:

```
//funció per ordenar el producte segons les unitats venudes
function sortProducte(a, b) {
  const aa = a.split("–")[0];
  const bb = b.split("–")[0];
  return aa – bb;
}

function ranquing() {
  let aux = [];
  //array auxiliar per no perdre el numproducte després de la ordenació
  //s'afegeix el numproducte juntament amb les unitats venudes del producte
  //com a valor del nou array
  //numproducte: Numero del producte venut
  //vendes.get(numproducte): unitats venudes del producte numproducte
  for (let [numproducte, quantitat] of vendes) {
    aux.push(quantitat + "–" + numproducte);
  }
  //ordenar els productes segons les unitats venudes
  //s'ha creat una funció d'ordenació per separar les unitats venudes
  // del numproducte.
  aux.sort(sortProducte).reverse();
  //llistar els productes ordenats
  let llistat = "";
  for (let valor of aux) {
    //Tallem la cadena pel simbol d'unió '–'.
    //a la posició [0] el numero d'unitats venudes del producte
    //a la posició [1] tenim el numero de producte
    let [stockProducte, codiProducte] = valor.split("–");

    llistat += '<li>´sha venut ${stockProducte} unitats del producte ${
        codiProducte}</li>';
  }

  document.getElementById("venuts").innerHTML = llistat;
}
```

El mapa `vendes` utilitza la seva clau com a número de producte.

És important remarcar aquest fet ja que a l'hora de fer l'ordenació, la funció *.sort()* posa a la primera posició de l'array el major número d'unitats venudes i perdrem a quin producte correspon.

Per generar el rànquing de vendes ens trobem amb dos problemes:

- `vendes` és un *map* i els mapes no es poden ordenar.
- si el convertim en un array i l'ordenem mitjançant el mètode `sort`, es perd la correspondència "índex-clau" amb l'array de productes, ja que fem servir l'índex de `productes` com identificador per a cada producte.

Ens hem d'inventar una manera de no perdre la referència al producte. Per sort, sabem que la funció d'ordenació la podem canviar. La idea és la següent: per no perdre la referència al producte, després de fer la ordenació, aquest ha d'estar dins del valor juntament amb les unitats venudes. Així, podem crear una cadena de text on, una part de la cadena siguin les unitats venudes i una altra part correspongui al producte associat. Per exemple, si tenim 50 unitats venudes del producte 3, podem construir una cadena de text semblant a "50-3". I així, amb tots els productes venuts.

Veieu el codi utilitzat per crear un nou array on guardem les cadenes de text amb el número d'unitats venudes i el seu producte.

```
for (let [numproducte, quantitat] of vendes) {
  aux.push(quantitat + "-" + numproducte);
}
```

Fixeu-vos que s'ha utilitzat `for...of` per recorrer el mapa de `vendes`, desestructurant l'element, de manera que la clau queda assignada a `numproducte` i el valor a `quantitat`.

Una vegada obtingudes les varibles, les concatenem per generar la cadena de text i les afegim a l'array auxiliar i, seguidament, l'ordenem mitjançant el mètode `sort`, passant com a argument la funció `sortProducte`. Aquesta es la funció que es passa com argument:

```
function sortProducte(a, b) {
  const aa = a.split("-")[0];
  const bb = b.split("-")[0];
  console.log(aa, bb);
  return aa - bb;
}
```

En aquest cas, en lloc de passar la funció `sortProducte` podríem haver utilitzat una funció de fletxa, però, com que aquesta funció d'ordenació és una mica llarga, s'ha optat per implementar-la com una funció estàndard; d'aquesta manera el codi resulta menys enrevessat.

Cal recordar que, per defecte, el mètode `sort` fa l'ordenació alfabèticament, però això no ens interessa perquè s'interpretaria que 5 és major que 100, ja que en ordenar cadenes de text es fa la comparació del codi cada caràcter i no s'interpreta com a número.

Tot i que en aquest exemple hem fet servir `const` per indicar que els valors de `aa` i `bb` no canviaran, això no és necessari. Es podria haver fet servir `let`.

Totes les funcions d'ordenació han de retornar un nombre positiu si el segon valor és més petit, un nombre negatiu si el primer valor és més petit i 0 si són iguals. Per fer el rànquing dels productes més venuts s'ha de comparar les unitats venudes de dos productes. En aquest cas, `a` té la forma *unitatsVenudesDe-producteA* i `b` té la forma *unitatsVenudesDe-producteB*. Hem de separar les unitats venudes del número de producte.

Per separar una cadena de text podem utilitzar el mètode `split`. A aquest mètode li podem passar un caràcter o una *string* i trencarà la cadena de text cada vegada que trobi aquest caràcter o *string*.

La sintaxi de la funció `split()` és la següent:

```
let nou_array = cadena.split([separador[, limit]])
```

El primer paràmetre és opcional i especifica el caràcter a utilitzar per separar la cadena de text. Cada cadena de text separada de l'original s'emmagatzemarà en

una posició de l'array retornat. El segon paràmetre indica el nombre de vegades que es separarà com a màxim la cadena de text utilitzant el separador.

En definitiva, si invoquem el mètode `split`, als paràmetres de la funció tenim que, a la primera posició, *[0]*, estem accedint al nombre d'unitats venudes i, a la segona posició, *[1]*, al producte associat.

<newcontent>Només s'ha d'agafar la primera posició, després d'invocar el mètode `split`, per obtenir les unitats venudes:

```
let aa = a.split("—")[0];
```

També existeix la funció inversa anomenada *join()*. La sintaxi de la funció és la següent:

```
let cadena = array.join([separador = ','])
```

El mètode `join` uneix tots els elements d'un array i els converteix en una cadena de text. Si s'especifica un separador, aquest s'utilitza per fer la unió entre els diferents elements de l'array.

Vegem un exemple:

```
let array = ["hola", "món"];
let missatge = array.join(""); // missatge = "holamón"
console.log(missatge);
missatge = array.join(" "); // missatge = "hola món"
console.log(missatge);
```

Una vegada tinguem l'array ordenat tindrem ordenats els valors (sencers, sense fer l´`split`) on a la posició `aux[0]` hi haurà el producte amb menys vendes. Per fer un rànquing, ens interessa que estigui a l'inrevés, és a dir, a la posició `aux[0]` hi hauria d'haver el producte amb més unitats venudes. Es podia haver tingut en compte a l'hora de realitzar la funció de comparació de la funció `.sort()`, però és interessant que conegueu la funció `reverse()`, que inverteix l'ordre dels elements de l'array.

```
aux.sort(sortProducte).reverse();
```

La sintaxi del mètode és la següent:

```
array.reverse()
```

El mètode `split` converteix una cadena de text en un array de cadenes de text. La funció divideix la cadena de text determinant els seus trossos a partir del caràcter separador indicat.
El mètode `join` uneix tots els elements d'un array en una cadena de text utilitzant el caràcter d'unió.
El mètode `reverse` modifica un array col·locant els seus elements en l'ordre invers a la seva posició original.

Exemple:

```
let array = [1, 2, 3];
array.reverse();
console.log(array); // ara array = [3, 2, 1]
```

En aquest punt ja tenim l'array amb el rànquing dels productes més venuts. Només cal llistar-lo per pantalla. Veieu el codi que permet fer el llistat:

```
//llistar els productes ordenats
  let llistat = "";
  for (let valor of aux) {
    //Tallem la cadena pel simbol d'unió '-'.
    //a la posició [0] el numero d'unitats venudes del producte
    //a la posició [1] tenim el numero de producte
    let [stockProducte, codiProducte] = valor.split("-");

    llistat += `<li>´sha venut ${stockProducte} unitats del producte ${
        codiProducte}</li>`;
  }

  document.getElementById("venuts").innerHTML = llistat;
```

S'utilitza la variable `llistat` per emmagatzemar tot l'HTML que es mostrarà a l'usuari. Una vegada contingui totes les dades amb els productes més venuts es mostrarà per pantalla utilitzant la línia de codi següent:

```
document.getElementById("venuts").innerHTML = llistat;
```

Com ja hem vist en altres ocasions, s'utilitza l'objecte `document` per accedir a l'element de la pàgina identificat amb la paraula `venuts` i introduït l'HTML que ha de mostrar. En aquest cas, l'HTML serà el rànquing dels productes més venuts que conté la variable `llistat`.

2.5 Funcionalitats addicionals

En aquest codi s'han ampliat les funcionalitats del programa. Concretament es vol:

- Saber quin és el primer producte que té *X* unitats d'estoc.
- Saber l'últim producte que té *X* unitats d'estoc.
- Comprovar si tots els productes tenen més de 10 unitats d'estocs.
- Comprovar si algun producte té 0 unitats d'estoc.
- Saber quins productes tenen 0 unitats d'estoc.
- Un llistat de l'estoc des del producte X al producte Y.
- Afegir 1 producte a partir d'una posició determinada de l'array `productes`.

```
//Volem saber quin és el primer producte té //X// unitats d'estoc (indexOf)
function primerProducte(unitats) {
  let index = productes.indexOf(parseInt(unitats));

  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
        unitats} unitats.';
  }
}

// Volem saber l'últim producte que té //X// unitats d'estoc (lastIndexOf()
function ultimProducte(unitats) {
  let index = productes.lastIndexOf(parseInt(unitats));
  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
        unitats} unitats.';
  }
}

// Volem comprovar si tots els productes tenen més de 10 unitats d'estoc  (
    every)
function minDeuUnitats() {
  document.getElementById("info").innerHTML =
    'Tots els productes tenen més de 10 unitats? ${productes.every(unitats =>
        unitats >= 10)}';
}

// Volem comprovar si algun producte té 0 unitats d'estoc (some)

function senseEstoc() {
  document.getElementById("info").innerHTML =
    'Hi ha productes sense estoc? ${productes.some(unitats => unitats === 0)}';
}

// Volem saber quins productes tenen 0 unitats d'estoc (filter)
function checkBuitSplit(unitats) {
  let aa = unitats.split("-")[0];
  return parseInt(aa) === 0;
}
function llistatProductesSenseEstoc() {
  let llistat = "No hi ha productes sense estoc.";

  if (productes.some(unitats => unitats === 0)) {
    let aux = [];
    let arr = [];
    llistat = "";
    for (let numproducte in productes) {
      arr.push(productes[numproducte] + "-" + numproducte);
    }
    aux = arr.filter(checkBuitSplit);
    for (let index in aux) {
      llistat +=
        '<li>El producte ${aux[index].split("-")[1]} no té estoc.</li>';
    }
    document.getElementById("info").innerHTML = llistat;
  } else {
    document.getElementById("info").innerHTML = llistat;
  }
}


//  Volem un llistat de l'estoc des del producte X al producte Y (slice)
function llistatParcialdEstoc() {
  let ini = parseInt(document.getElementById("valor1").value);
  let fin = parseInt(document.getElementById("valor2").value);
```

```
  let estoc = productes.slice(ini, fin);
  document.getElementById("info").innerHTML = estoc.toString();
  veureEstoc();
}

//  Volem afegir 1 producte a partir d'una posició X de l'array productes(splice).

function afegirEstocPosicioValor1() {
  let ini = parseInt(document.getElementById("valor1").value);
  productes.splice(ini, 0, Math.floor(Math.random() * 100));
  veureEstoc();
}
```

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

2.5.1 Primer i últim producte amb X unitats d'estoc

Es vol saber quin és el primer producte i quin és l'últim producte que té X unitats d'estoc, on X és un valor que ha introduït l'usuari.

El codi per saber quin és el primer producte amb X unitats d'estoc és el següent:

```
function primerProducte(unitats) {
  let index = productes.indexOf(parseInt(unitats));

  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${unitats} unitats.';
  }
}
```

I l'HTML associat a aquesta funció és el següent:

```
<input type="text" id="unitats" size="10"/>
<button onclick="primerProducte(document.getElementById('unitats').value);" >
   Primer producte amb X unitats d'estoc
</button>
```

El mètode que ens retorna el primer índex de l'array que coincideix el seu valor amb el passat per paràmetre es diu `indexOf`.

> La funció `indexOf` retorna la posició de l'array on es troba la primera ocurrència del valor cercat. Si no es troba el valor cercat, la funció retorna el valor -1.
> En canvi, la funció `lastIndexOf` retorna la posició de l'array on es troba l'última ocurrència del valor cercat.
> En tots dos casos, si no és troba el valor cercat retornen -1.

El mètode `indexOf` té la següent sintaxi:

```
let index = array.indexOf(elementCercat[, inici = 0])
```

Aquest mètode retorna el primer índex de l'array que coincideix el valor d'aquesta posició amb el paràmetre de la funció.

En canvi, el mètode `lastIndexOf` retorna l'últim índex i té la següent sintaxi:

```
let index = arr.lastIndexOf(elementCercat[, inici = arr.length - 1])
```

El codi per saber quin és l'últim producte amb *X* unitats d'estoc és el següent:

```
function ultimProducte(unitats) {
  let index = productes.lastIndexOf(parseInt(unitats));
  if (index !== -1) {
    document.getElementById("info").innerHTML = index;
  } else {
    document.getElementById("info").innerHTML = 'No hi ha cap producte amb ${
        unitats} unitats.';
  }
}
```

I el codi HTML associat és el següent:

```
<input type="text" id="unitats" size="10"/>
<button onclick="ultimProducte(document.getElementById('unitats').value);" >
    Últim producte amb X unitats d'estoc
</button>
```

2.5.2 Comprovar si tots els productes tenen 10 unitats d'estoc

Es vol comprovar si tots els productes tenen com a mínim 10 unitats d'estoc. Es pot fer recorrent l'array amb un bucle tipus `for`, però existeix una funcionalitat que ho fa per nosaltres: el mètode `every`.

Es pot crear una funció condicional o, com hem fet nosaltres, utilitzar una funció de fletxa per passar com argument. Aquesta funció és la que s'utilitzarà per comprovar si tots els elements acompleixen amb el requeriment (en el nostre cas, que hi hagi almenys 10 unitats).

El codi corresponent a aquesta funcionalitat és el següent:

```
// Volem comprovar si tots els productes tenen més de 10 unitats d'estoc  (
    every)
function minDeuUnitats() {
  document.getElementById("info").innerHTML =
    'Tots els productes tenen més de 10 unitats? ${productes.every(unitats =>
        unitats >= 10)}';
}
```

La funció de fletxa comprova si una unitat és més gran que 10 i es passa com a argument al mètode `every` per comprovar si l'estoc de cada producte és més gran que 10.

Fixeu-vos que, com només hi ha un argument, es pot prescindir dels parèntesis a la funció de fletxa:

```
unitats => unitats >= 10
```

És a dir, per cada element de l'array es cridarà a la funció de fletxa:

- S'assignarà a `unitats` el valor de l'element
- La funció de fletxa retornarà cert si `unitats >= 10` o fals en cas contrari.
- Si la funció de fletxa retorna fals en algun cas, el mètode `every` retornarà fals.

El mètode `every(funció)` comprova, element a element, si tots els valors de l'array compleixen la funció passada com a argument. Si tots els valors la compleixen retorna *true*, però si hi ha algun valor que no ho fa retorna *false*.

La sintaxi del mètode `every` és la següent:

```
resultat = arr.every(funcióComparacio[, parametres])
```

2.5.3 Comprovar si hi ha algun producte sense estoc

Es vol comprovar si hi ha algun producte sense estoc. Es pot fer amb un bucle `for` o bé utilitzar un mètode d'arrays que comprova si existeix algun valor que compleix un requisit. Aquesta funció s'anomena `some`. La funció rep per paràmetre una funció que realitza la comprovació. En aquest cas, es vol que la quantitat d'un producte sigui 0 per saber si hi ha o no estoc.<newcontent> Veieu el codi associat a aquesta funcionalitat:

```
function senseEstoc() {
  document.getElementById("info").innerHTML =
    `Hi ha productes sense estoc? ${productes.some(unitats => unitats === 0)}`;
}
```

<newcontent>

El mètode `some(funció)` comprova si hi ha algun element que compleix la funció que es passa com a argument. Si és així, la funció retorna *true*, si no, *false*.

El mètode `some` té la següent sintaxi:

```
let resultat = array.some(funcioComparacio[, parametres])
```

2.5.4 A quins productes se'ls ha esgotat l'estoc?

Aquesta funcionalitat és una modificació de l'anterior. No només volem saber si hi ha algun producte, sinó que volem saber quins són aquests productes.

El mètode que ens permet extreure una part d'un array segons una condició es diu `filter()`. Com que volem, a part dels valors, el número de producte, utilitzarem, igual que es va fer amb el rànquing dels productes més venuts, la cadena de text "unitats-producte" com a valor de l'array sobre el que utilitzarem el mètode `filter`.

Fixeu-vos en el codi:

```
// Volem saber quins productes tenen 0 unitats d'estoc (filter)
function checkBuitSplit(unitats) {
  let aa = unitats.split("-")[0];
  return parseInt(aa) === 0;
}
function llistatProductesSenseEstoc() {
  let llistat = "No hi ha productes sense estoc.";

  if (productes.some(unitats => unitats === 0)) {
    let aux = [];
    let arr = [];
    llistat = "";
    for (let numproducte in productes) {
      arr.push(productes[numproducte] + "-" + numproducte);
    }
    aux = arr.filter(checkBuitSplit);
    for (let index in aux) {
      llistat +=
        `<li>El producte ${aux[index].split("-")[1]} no té estoc.</li>`;
    }
    document.getElementById("info").innerHTML = llistat;
  } else {
    document.getElementById("info").innerHTML = llistat;
  }
}
```

::important: El mètode `filtre(funció)` retorna un nou array amb tots els elements pels quals la funció ha retornat `true`. :: El mètode `filter` té la següent sintaxi:

```
resultat = array.filter(funcioComprovacio[, parametres])
```

La funció `filter` crea un nou array amb els elements que passen la funció de comprovació proporcionada a la funció.

2.5.5 Llistat dels estocs del producte X al producte Y

Es vol obtenir el llistat dels estocs, des d'una posició de l'array fins a una altra. Les dues posicions les escull l'usuari.

Fixeu-vos en el codi següent:

```
// Volem un llistat de l'estoc des del producte X al producte Y (slice)
function llistatParcialdEstoc() {
  let ini = parseInt(document.getElementById("valor1").value);
  let fin = parseInt(document.getElementById("valor2").value);
  let estoc = productes.slice(ini, fin);
  document.getElementById("info").innerHTML = estoc.toString();
}
```

> El mètode `slice(inici, final)` retorna un nou array amb els valors que es troben des de l'índex `inici` fins a l'última posició de l'array o fins a l'índex `final` si s'ha passat el segon argument.

El mètode `slice` té la següent sintaxi:

```
nou array = array.slice([començament[, final]])
```

Com es pot veure, la funció `llistatParcialdEstoc` assigna a les variables `ini` i `fin` els valors introduïts per l'usuari a les caixes de text. A continuació es fan servir aquestes variables per invocar al mètode `slice`, que retorna un nou array amb els valors de `productes` compresos entre aquests dos indexs i, finalment, és mostra aquest nou array a la pàgina convertint-los en una cadena de text mitjançant el mètode `string`.

2.5.6 Afegir un producte a partir d'una posició

Es vol afegir un producte a partir d'una posició donada de l'array `productes`. Tots els productes que hi hagi després de l'índex d'inserció es desplaçaran tantes unitats com productes afegits.

Veieu l'exemple:

```
//  Volem afegir 1 producte a partir d'una posició X de l'array productes(
    splice).

function afegirEstocPosicioValor1() {
  let ini = parseInt(document.getElementById("valor1").value);
  productes.splice(ini, 0, Math.floor(Math.random() * 100));
  veureEstoc();
}
```

Com es pot apreciar, primerament obtenim la informació introduïda per l'usuari, la convertim en nombre enter i l'assignem a la variable `ini`. Aquest serà el punt d'inserció.

No volem eliminar cap element, així que com a segon argument de la funció posem `0` i, finalment, generem un valor pseudoaleatori entre 0 i 99, que passem com a tercer paràmetre:

```
productes.splice(ini, 0, Math.floor(Math.random() * 100));
```

El mètode `splice(inici, eliminats, valor1, valor2,... valorN)` afegeix i elimina elements a partir d'una posició de l'array.

Com que `splice` accepta qualsevol quantitat de valors separats per comes, es pot utilitzar l'operador de propagació per inserir un array en aquesta posició, per exemple:

```
let a = ['a', 'b', 'c', 'd'];
let b = [1, 2, 3, 4];
a.splice(1,0, ...b); // insereix a la segona posició tots els elements de b
```

En concret, els paràmetres de la funció son els següents:

- *inici*: és la posició on començarà a afegir o eliminar.
- *eliminats*: és el nombre d'elements que es volen esborrar a partir del paràmetre *inici*. Si posem un 0, només afegirà valors.
- *valor1, valor2,... valorN*: són els valor que volem afegir.

A continuació podeu veure un exemple amb els diferents resultats d'utilitzar el mètode `splice`:

```
let array = [1, 2, 3, 4, 5];
array.splice(1, 3);
// Ara 'array elimina 3 elements a partir de la posició 1,  i queda així: [1, 5]
console.log(`Contingut del array (1): ${array}`);

array = [1, 2, 3, 4, 5];
array.splice(2, 0, 2.5);
// Ara 'array' afegeix l'element 2.5 a partir de la posició 2 i queda així: [1, 2, 2.5, 3, 4, 5]
console.log(`Contingut del array (2): ${array}`);

array = [1, 2, 3, 4, 5, 6];
array.splice(2, 3);
// Ara 'array' elimina 3 elements a partir del segon element (no inclòs) i queda així: [1, 2, 6]
console.log(`Contingut del array (3): ${array}`);

array = [1, 2, 3, 4, 5];
array.splice(1, 3, "two", "three", "four");
// Ara 'array' elimina 3 elements a partir del primer element (no inclòs) i s' afegeixen
// 'two', 'three' i 'four' i queda així: [1, "two", "three", "four", 5]
console.log(`Contingut del array (4): ${array}`);
```

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

Objectes definits pel programador

Xavier Garcia Rodríguez

Índex

Introducció

Tot i que JavaScript és un llenguatge orientat a objectes, en el qual pràcticament tots els seus elements són objectes, es tracta d'un sistema basat en prototips i no en classes. Aquesta diferència, juntament amb algunes decisions poc afortunades en la implementació del llenguatge, fan que pocs desenvolupadors utilitzin els objectes de JavaScript correctament.

Per aquesta raó, la correcta assimilació d'aquesta unitat és fonamental pel desenvolupament d'aplicacions en l'entorn del client, ja que en tot moment es treballa amb objectes.

En aquesta unitat es tractarà la creació, actualització, augment i modificació dels objectes a més a més dels usos alternatius que se'ls pot donar, com pot ser crear un diccionari de dades o un espai de noms.

Seguidament, es descriurà el sistema de prototipus, les seves característiques, com utilitzar-lo i com augmentar els objectes predefinits del sistema per afegir noves característiques globalment.

A continuació, veureu com implementar diferents patrons de creació d'objectes, juntament amb els seus avantatges i inconvenients. Concretament es tractaran els patrons prototípic, funcional i constructor.

Atesa la importància de l'herència en els llenguatges orientats a objectes, s'explicarà quines alternatives es troben a la vostra disposició per implementar-la a JavaScript, tant en forma de jerarquia d'objectes i constructors, com a través de la composició i la delegació per construir objectes complexos.

Finalment, trobareu un resum de les característiques relacionades amb la creació d'objectes afegides a ES6. En concret el sistema de classes, molt similar al que es pot trobar en altres llenguatges clàssics com Java o C++.

Cal recordar que, com en tots els llenguatges de programació, la millor manera d'assimilar els conceptes d'aquesta unitat és practicant, modificant els exemples que trobareu al llarg de la unitat, escrivint els vostres propis programes, consultant l'extensa documentació en línia que podeu trobar a Internet i preguntant al fòrum de l'assignatura.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Programa codi per a clients web analitzant i utilitzant estructures definides per l'usuari.

- Reconeix les característiques d'orientació a objectes del llenguatge.
- Crea codi per definir l'estructura d'objectes.
- Crea mètodes i propietats.
- Crea codi que faci ús d'objectes definits per l'usuari.
- Depura i documenta el codi.

1. Objectes definits pel programador

El terme *herència clàssica* es refereix a un tipus d'herència basada en classes.

A excepció dels nombres, les cadenes de caràcters, els booleans (`true` i `false`), `null` i `undefined`, a JavaScript tot són objectes. Fins i tot les funcions són tractades com objectes i, per tant, comparteixen les seves característiques. Encara que es pot pensar que els tipus primitius també són objectes (inclouen mètodes i propietats) aquests són immutables i per tant no en són, per exemple: el número 2 sempre serà 2, no pot ser un altre número i, en conseqüència, és immutable.

Ara bé, si tenim una variable a la qual s'ha assignat un valor numèric és possible cridar mètodes a partir d'aquesta variable com per exemple:

```
const PI = 3.141592653589793;
console.log(num.toFixed(2); // mostra "3.14"
```

Això és possible perquè internament es fa la conversió del valor assignat a un objecte de tipus `Number` i llavors es crida al mètode `toFixed`.

Actualment JavaScript admet la creació d'objectes fent servir l'*herència clàssica*, mitjançant la definició de classes i la seva instanciació mitjançant l'operador `new`, de forma similar a com es fa en altres llenguatges com Java o C++:

```
class Persona {
  constructor(nom, edat) {
    this.nom = nom;
    this.edat = edat;
  }
}

let ricard = new Persona('ricard', 40);
let patrici = new Persona('patrici', 35);

console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat);
```

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

Fixeu-vos que a la definició de la classe s'ha posat el primer caràcter en majúscules: `Persona`, en canvi el nom de les variables comença amb minúscules(`ricard`). Això es fa per convenció: els noms de les classes sempre comencen per majúscules, d'aquesta manera és fàcil diferenciar quan es fa referència a una classe o a un altre element.

> Una **classe** és una plantilla a partir de la qual es creen **objectes** i es diu que aquests objectes són **instàncies** de la classe.

Un altre sistema per crear objectes a JavaScript és la declaració literal d'objectes:

```
let ricard = {nom: 'ricard', edat: 40};
let patrici = {nom: 'patrici', edat: 35};

console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat);
```

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

Com es pot apreciar, és molt més concís, ja que no requereix la definició de les classes, però requereix definir totes les propietats de cada objecte.

En versions anteriors a ES6 no existien les classes i per simular l'herència clàssica calia utilitzar funcions constructores. Aquest sistema continua funcionant, però no es recomana utilitzar-lo perquè complica la creació d'objectes complexos i és més limitada, ja que per implementar l'herència cal modificar el prototipus de l'objecte i la manera de simular els mètodes estàtics no és clara. A continuació podeu veure un exemple on s'utilitzen aquestes funcions constructores, típiques del codi anterior a ES6:

```
const Persona = function (nom, edat) {
  this.nom = nom;
  this.edat = edat;
};

let ricard = new Persona('Ricard', 40);
let patrici = new Persona('Patrici', 35);
console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat);
```

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

La instanciació dels objectes `Persona` es fa de la mateixa manera que quan es defineix una classe, però s'utilitza una funció juntament amb l'operador `new`. Cal tenir compte amb això, perquè si ens oblidem d'afegir l'operador, l'aplicació continuarà funcionant però no s'hauria creat l'objecte, ja que s'hauria invocat la funció i assignat el retorn d'aquesta que serà `undefined`:

```
let ricard = new Persona('Ricard', 40);
let patrici = Persona('Patrici', 35);
console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat); // error, patrici és undefined
```

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

Fixeu-vos que `patrici` és `undefined`, ja que en lloc d'assignar-se l'objecte creat mitjançant la funció constructora `persona` s'ha assignat el retorn d'aquesta, que en aquest cas és `undefined`.

Un dels problemes d'utilitzar funcions constructores és que per crear objectes complexos cal recórrer a la modificació del prototipus de l'objecte o afegir tot el codi dintre del cos de la funció constructora (per exemple, múltiples funcions niuades).

Originalment JavaScript només permetia l'*herència prototípica*. Tots els objectes es creen a partir d'un prototipus del qual hereten les seves propietats i mètodes.

En aquests materials ens centrarem en la creació d'objectes mitjançant la definició de classes i la declaració literal d'objectes.

1.1 Introducció a la programació orientada a objectes

Les característiques que ha d'acomplir un llenguatge per considerar-se orientat a objectes són les següents:

- **Abstracció**: s'han de poder crear objectes que ens serviran per modelar la realitat i el problema a resoldre (per exemple en un joc sobre un apocalipsi zombi s'han de poder modelar els personatges, les armes, els zombis, els vehicles que es troben, etc.)
- **Encapsulament**: s'ha de poder encapsular informació dins d'un objecte que no sigui accessible externament. En llenguatges com Java seria equivalent a mètodes i propietats amb accés privat, en canvi en JavaScript es fa a través de les **clausures**.
- **Herència**: una classe ha de poder heretar d'un altre de manera que la nova classe tingui totes les característiques (accessibles) del que hereta a més de les pròpies.
- **Polimorfisme**: una classe que hereti d'un altre ha de poder modificar el comportament de les accions, per exemple sobreescrivint els mètodes.

Una classe conté propietats i mètodes. Les **propietats** són similars a les variables, ja que permeten emmagatzemar valors (nombres, cadenes de text, objectes, *arrays*, etc.), però lligades a l'objecte. Per altra banda, els **mètodes** són funcions lligades a l'objecte i el seu funcionament és idèntic, amb excepció del context `this` que passa a ser el mateix objecte.

Cal destacar que a JavaScript el concepte de modificador d'accés (públic o privat) no forma part del llenguatge. **Totes les propietats i mètodes són considerats públics**, encara que és possible encapsular-los mitjançant *clausures*.

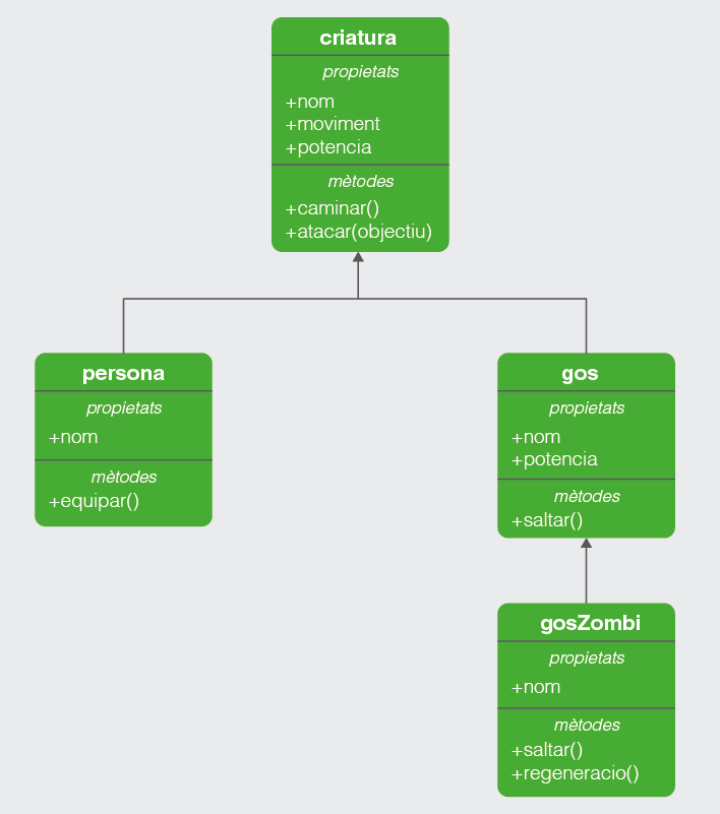
Actualment JavaScript permet utilitzar explicitament totes aquestes característiques a excepció de l'encapsulament, que requereix la utilització de clausures.

Quant a l'herència cal tenir clars dos conceptes basats en l'herència clàssica:

- **Generalització**: consisteix en la creació d'una classe que implementi les propietats i mètodes comuns d'altres classes que heretaran d'aquesta.
- **Especialització**: consisteix en la creació d'una classe amb un comportament especialitzat que afegeix o modifica el comportament de la classe de la qual hereta.

Fixeu-vos en la figura 1.1 per veure un exemple de jerarquía d'herència:

FIGURA 1.1. Generalització i especialització



Quan es tracta d'objectes el seu nom comença amb minúscula.

Com es pot apreciar, `Criatura` és la generalització de `Persona` i `Gos`, ja que conté els mètodes comuns. Per altra banda, `Persona` i `Gos` són especialitzacions de `Criatura`, ja que actualitzen algunes propietats i afegeixen nous mètodes. Al mateix temps, `GosZombi` és una especialització de `Gos` perquè hereta d'aquest i hi afegeix les seves propietats i mètodes.

En els **diagrames de classes**, com el de la figura 1.1, es posa a la capçalera del requadre el nom de la classe, a continuació les propietats i en darrer lloc els mètodes (fàcilment distingibles perquè inclouen parèntesis). Per indicar l'herència es fan servir fletxes amb la punta buida, que assenyalen a la super-classe (o objecte/constructor pare).

Es pot trobar més informació sobre els *diagrames de classes* a l'enllaç següent: ca.wikipedia.org/wiki/Diagrama_de_classes.

1.2 Creació d'objectes amb el constructor

En els llenguatges clàssics es denomina constructor a la funció que es crida automàticament quan es crea un objecte d'una classe concreta. Aquesta funció pot acceptar paràmetres que habitualment s'utilitzen per inicialitzar la classe.

A JavaScript existeix un concepte de constructor que és una mica diferent, ja que no fa referència al constructor que es pot trobar a una classe, sinó que es tracta

d'una funció per inicialitzar un objecte i que s'utilitza juntament amb l'operador `new`:

Tots els objectes hereten d'`Object`, que es pot fer servir com a funció constructora per crear un objecte buit.

```
let persona = new Object();
```

El resultat d'executar aquest codi és equivalent a: `let persona = {}`. En cas de voler crear un objecte nou amb propietats fent servir aquest format es pot fer de dues maneres:

- Afegint un objecte declarat literalment com a paràmetre de `Object`, per exemple: `new Object({nom: "Ricard"})`.
- Augmentar-lo afegint les propietats que siguin necessaries.

De la mateixa manera, si volem crear una instància d'una classe, només cal indicar el nom de la classe, i, si escau, els paràmetres que passaran al constructor:

```
let ricard = new Persona('ricard', 40);
```

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

1.3 Declaració literal d'objectes

La manera més simple de crear un objecte en JavaScript és a través de la seva declaració literal. Aquesta declaració consisteix a fer servir un parell de claus `{}` dins de les quals es defineixen els valors desitjats en forma de parells clau-valor, que corresponen al nom i al valor de la propietat respectivament, separats per comes.

Per exemple, es pot crear un objecte referenciat per la variable `persona` que guardarà les dades d'aquesta persona:

```
let persona = {
  nom: 'Ricard',
  ocupacio: 'Policia',
  edat: 40
};
```

És important tenir en compte que el format és diferent de l'habitual que trobem a JavaScript, ja que en lloc de fer servir l'operador d'assignació `=` es fan servir els dos punts `:`.

Aquesta forma de crear objectes és la base del **format d'intercanvi de dades JSON** (*JavaScript Object Notation* en anglès), un format molt utilitzat tant en JavaScript com en altres llenguatges.

Podeu trobar més informació sobre el format JSON en la secció "Annexos" del web del mòdul.

1.3.1 Assignar i accedir a propietats

Les **propietats** d'un objecte són similars a les variables, però els seus valors són accessibles només a través de l'objecte. En aquest cas per mostrar el valor del nom de `persona` es faria de la manera següent:

```
1 console.log(persona.nom);
```

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

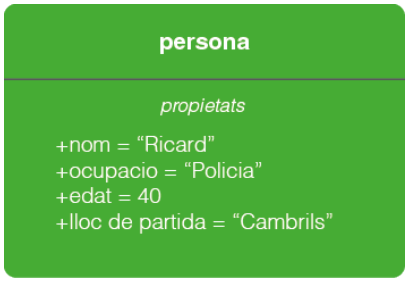
El nom de les propietats es pot afegir amb cometes o sense, excepte si ha d'incloure espais (cosa poc habitual), llavors és obligatori:

```
1 let persona = {
2   'nom': 'Ricard',
3   'ocupacio': 'Policia',
4   'edat': 40,
5   'lloc de partida': 'Cambrils'
6 };
7
8 console.log(persona.nom);
```

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

i el diagrama corresponent a la figura 1.2.

FIGURA 1.2. Diagrama d'objecte amb quatre propietats



El símbol + indica que es tracta de propietats públiques

En aquest cas s'han fet servir les cometes per totes les propietats encara que només són necessàries en el cas de `lloc de partida`, ja que pel fet de contenir espais es produiria un error.

Cal tenir en compte que es poden fer servir indistintament les cometes simples o les cometes dobles per definir el nom de les propietats, de la mateixa manera que si es tractessin de cadenes de text, com es pot apreciar en el següent exemple:

```
1 let persona = {
2   "nom": "Ricard",
3   "ocupacio": "Policia",
4   "edat": 40,
5   "lloc de partida": "Cambrils"
6 };
```

Com podeu apreciar, s'hi han substituït les cometes simples per cometes dobles, tant al nom de les propietats com al de les cadenes de text. Utilitzar les primeres és un punt més òptim que fer servir les dobles, però allò realment important és mantenir un mateix criteri al llarg del codi.

A més de la notació de punt, és possible accedir a les propietats fent servir la notació de claudàtors, com si es tractés d'un *array* en lloc d'un objecte. Per exemple, podem afegir a l'exemple anterior:

```
console.log(persona.['edat']);
console.log(persona.['ocupacio']);
```

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

Això permet tractar aquests objectes de la mateixa manera que en altres llenguatges es tracten els *arrays* associatius o els diccionaris. A més a més, al contrari que en altres llenguatges, els *arrays* de JavaScript no són tan eficients, raó per la qual es poden utilitzar objectes o *arrays* indistintament segons les vostres necessitats.

Encara que s'ha de tenir en compte que els objectes no ofereixen les mateixes funcionalitats que els *arrays*, ja que l'ordre és irrellevant.

A més a més, actualment JavaScript inclou l'objecte predefinit `Map` que permet crear col·leccions de dades en parells clau-valor i funcionalitats més apropiades per treballar amb dades com la iteració, la comprovació dels valors, l'addició i l'eliminació de les dades.

En el següent exemple, en el cas de les ocupacions només interessa conèixer el nom de l'ocupació i per tant és més pràctic fer servir un *array*, en canvi, en el cas dels supervivents és més útil poder accedir a la informació directament fent servir el continent com a clau:

```
let ocupacions = ['Sense ocupació', 'Policia', 'Militar', 'Metge', 'Tècnic', '
    Mecànic', 'Delinqüent'];

let supervivents = {
    'Barcelona': 23140,
    'Girona': 6789,
    'Lleida': 11298,
    'Tarragona': 19830
};

console.log('Array: Primera ocupació?', ocupacions[0]);
console.log('Objecte: Supervivents a Girona?', supervivents['Girona']);
```

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

Però no només es poden emmagatzemar valors primitius, les propietats dels objectes ens permeten emmagatzemar tot tipus de dades com poden ser *arrays* i altres objectes. Al mateix temps, aquests objectes poden ser emmagatzemats en *arrays*, el que permet crear estructures de dades complexes:

```
let MOTOR_DEL_JOC = {
```

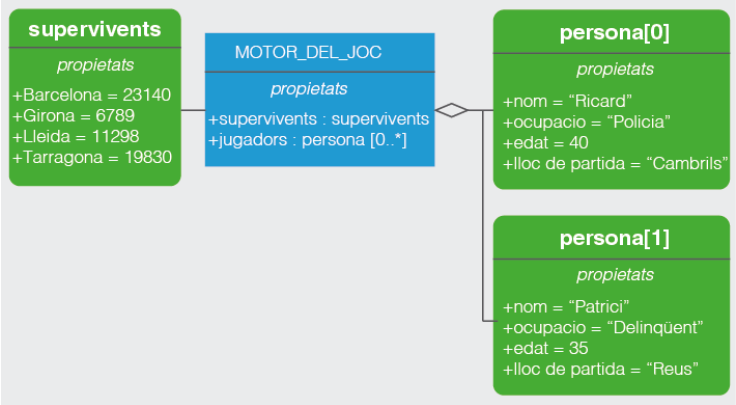
Ús de caràcters no estàndard com a nom de propietats

Quan es tracten els objectes com a diccionaris és normal utilitzar caràcters que habitualment no són recomanables pels noms de les propietats i variables, per exemple: caràcters amb titlla o dièresi.

```
  ocupacions: ['Sense ocupació', 'Policia', 'Militar', 'Metge', 'Tècnic', 'Mecà
      nic', 'Delinqüent'],

  supervivents: {
    'Barcelona': 23140,
    'Girona': 6789,
    'Lleida': 11298,
    'Tarragona': 19830,
  },

  jugadors: [{
    'nom': 'Ricard',
    'ocupacio': 'Policia',
    'edat': 40,
    'lloc de partida': 'Cambril'
  }, {
    'nom': 'Patrici',
    'ocupacio': 'Delinqüent',
    'edat': 35,
    'lloc de partida': 'Reus'
  }]
}

console.log('Tercera ocupació:', MOTOR_DEL_JOC.ocupacions[3]);
console.log('Supervivents a Girona:', MOTOR_DEL_JOC.supervivents['Girona']);
console.log('Nom del primer jugador:', MOTOR_DEL_JOC.jugadors[0]['nom']);
console.log('Ocupació del primer jugador:', MOTOR_DEL_JOC.jugadors[0].ocupacio)
    ;
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ExKNvro?editors=0012 i el diagrama corresponent a la figura 1.3.

Figura 1.3. Estructura de dades que conté arrays i altres objectes



Als diagrames de classe es fa servir un rombe a l'extrem de la classe que conté la **composició** (els múltiples elements que formen l'altre objecte, per exemple els elements d'un *array*), en canvi quan es tracta d'un sol element es defineix com una associació i es fa servir una línia simple que uneix les dues classes.

Com es pot apreciar, per accedir als elements d'*arrays* i d'altres objectes niuats només cal fer servir la notació de punt o de claudàtors segons correspongui:

```
console.log('Nom del primer jugador:', motorDelJoc.jugadors[0]['nom']);
```

Per enumerar les propietats d'un objecte o els elements d'un diccionari de dades s'utilitza una variant de la sentència `for`, coneguda com `for...in`. Aquesta sentència recorre tots els elements d'un objecte o *array* sense haver d'especificar el valor inicial ni final, ja que fa un recorregut complet, com es pot apreciar en el

següent exemple:

```
let ocupacions = ['Sense ocupació', 'Policia', 'Militar', 'Metge', 'Tècnic', '
    Mecànic', 'Delinqüent'];

let supervivents = {
  'Barcelona': 23140,
  'Girona': 6789,
  'Lleida': 11298,
  'Tarragona': 19830
};

console.log('Llistat de professions');
for (let clau in ocupacions) {
  console.log(ocupacions[clau]);
}

console.log('Llistat de supervivents per províncies')
for (let provincia in supervivents) {
  console.log(provincia, ':', supervivents[provincia]);
}
```

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

> La sentència `for...in` itera sobre totes les propietats de l'objecte, diccionari o *array* **sense oferir cap garantia de respectar l'ordre**.

Aquesta variant de la sentència `for` emmagatzema a la variable la clau de la propietat actual de l'objecte especificat a continuació del `in`, iterant sobre totes les propietats de l'objecte o els elements d'un *array*.

Un altre ús molt habitual d'un objecte declarat literalment és utilitzar-lo com a paràmetre de funcions i mètodes. D'aquesta manera només cal passar l'objecte com a argument i no cal recordar l'ordre dels paràmetres, ja que s'accedeix a aquell que sigui necessari fent servir el nom de la propietat (o clau):

```
function mostrarDades(dadesPersona) {
  console.log(`Nom: ${dadesPersona.nom}`);
  console.log(`Ocupació: ${dadesPersona.ocupacio}`);
  console.log(`Edat: ${dadesPersona.edat}`)
};

let dadesPersona = {
  nom: 'Ricard',
  ocupacio: 'Policia',
  edat: 40
};

mostrarDades(dadesPersona);
```

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

A més a més, és possible desestructurar l'objecte per convertir les propietats en paràmetres:

```
function mostrarDades({nom, ocupacio, edat}) {
  console.log(`Nom: ${nom}`);
```

Desestructuració d'objectes

Per desestructurar un objecte cal escriure els noms de les propietats que volem extreure entre claus. Els valors d'aquestes propietats s'assignen automàticament a variables amb el mateix nom.

```
  console.log(‘Ocupació: ${ocupacio}‘);
  console.log(‘Edat: ${edat}‘)
};

let dadesPersona = {
  nom: ’Ricard’,
  ocupacio: ’Policia’,
  edat: 40
};

mostrarDades(dadesPersona);
```

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

Utilitzant la desestructuració es poden definir també valors per defectes per alguns o per a tots els paràmetres:

```
function mostrarDades({nom = "Lluis", ocupacio = "Comerciant", edat}) {
  console.log(‘Nom: ${nom}‘);
  console.log(‘Ocupació: ${ocupacio}‘);
  console.log(‘Edat: ${edat}‘)
};

let dadesPersona = {
  nom: ’Ricard’,
  edat: 40
};

mostrarDades(dadesPersona);
```

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

Com es pot apreciar, la funció `mostrarDades` utilitza el `nom` i l´`edat` que hem passat com argument i l´`ocupacio` per defecte.

És recomanable aplicar aquesta tècnica quan hi ha molts paràmetres optatius, ja que s'evita haver de passar valors nuls com a paràmetres a la funció. L'inconvenient és que, si no es documenta la funció o mètode que rep l'objecte, s'ha d'analitzar el seu codi per saber quines són les propietats que ha de contenir l'objecte.

Quan es treballa ma classes es possible de definir valors per defecte assignant-los directamente a la classe:

```
class Persona {

  constructor (nom) {
    this.nom = nom;
  }

  edat = 0
  ocupacio = ’desconeguda’
}

let persona = new Persona(’Ricard’);

console.log(persona.nom);
console.log(persona[’edat’]);
console.log(persona[’ocupacio’]);

console.log("=".repeat(15));
```

```
18 persona.edat = 40;
19 persona.ocupacio = 'Policia';
20
21 console.log(persona.nom);
22 console.log(persona['edat']);
23 console.log(persona['ocupacio']);
```

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

Fixeu-vos que el cas del `nom` hem optat per assignar-lo al constructor. Tot i que no ho hem definit com a propietat, un cop es crea una instància de l'objecte es crida automàticament al constructor i s'assigna el valor passat com a paràmetre a la propietat `nom`.

Per altra banda, hem assignat a `edat` i `ocupacio` uns valors per defectes, per tant es poden consultar a partir de la instància de `Persona` i el seu valor serà el definit a la classe.

> **Definició de propietats dins de classes**
>
> Tot i que és possible definir propietats fora del constructor, no s'acostuma a fer així. Es recomana definir totes les propietats amb els seus valors per defecte dins del constructor de la classe.

Com es pot apreciar a l'hora de consultar-los i modificar-los ho podem fer de la mateixa manera que es fa amb la declaració literal.

1.3.2 Assignar mètodes

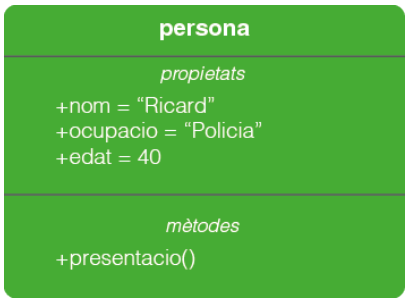
Mètode és el nom que rep una funció quan forma part d'un objecte o classe.

Atès que, en JavaScript, les funcions poden ser emmagatzemades com a variables, el fet d'assignar una funció a una propietat converteix aquesta en un mètode. Aquests funcionen exactament igual que una funció, però **el seu context d'execució (`this`) és l'objecte**.

```
1  let persona = {
2    nom: 'Ricard',
3    ocupacio: 'Policia',
4    edat: 40,
5    presentacio: function () {
6      console.log(`Hola, em dic ${this.nom}, tinc ${this.edat} anys i sóc ${this.
           ocupacio}`);
7    }
8  };
9
10 persona.presentacio();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/NWNbaxr?editors=0012 i el diagrama que el representa a la figura 1.4.

Figura 1.4. Objecte amb mètodes



Els mètodes s'afegeixen a la capsa inferior del requadre de la classe o objecte.

Com es pot apreciar, per invocar un mètode cal escriure primer el nom de la variable que referencia a l'objecte i a continuació fent servir la notació de punt (o de claudàtors) el nom del mètode a invocar seguit dels parèntesis.

Totes les classes inclouen un mètode per defecte anomenat `constructor`, que és cridat automàticament quan es crea una nova instància de la classe:

```
1  class Persona {
2
3    constructor(nom, edat) {
4      this.nom = nom;
5      this.edat = edat;
6    }
7  }
```

Fixeu-vos que per definir mètodes dins d'una classe la sintaxi és diferent: `nomFuncio(arguments) {/* Cos de la funció */}`. Com es pot apreciar, no cal utilitzar `function` ni assignar la funció a cap variable ni propietat.

Per definir altres mètodes ho fem de la mateixa manera:

```
1  class Persona {
2    constructor(nom, edat, ocupacio) {
3      this.nom = nom;
4      this.edat = edat;
5      this.ocupacio = ocupacio;
6    }
7
8    saludar() {
9       console.log('Hola, em dic ${this.nom}, tinc ${this.edat} anys i sóc ${this
            .ocupacio}');
10   }
11 }
12
13 let ricard = new Persona('Ricard', 40, 'Policia');
14 ricard.saludar();
```

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

Cal recordar que les funcions a JavaScript es poden declarar amb nom o com a funcions anònimes. En qualsevol dels dos casos es poden afegir com a mètodes d'un objecte, només cal assignar com a valor de la propietat el nom de la funció o el nom de la variable que fa referència a la funció sense incloure els parèntesis.

Fixeu-vos que, tot i que no és idèntic, el format en què es declaren els mètodes

és molt similar a com es declaren funcions anònimes que es guarden en variables: `let nomVariable = function(arguments) {/* Cos de la funció */}`.

> Si a una variable o propietat s'assigna una funció afegint parèntesis aquesta **és invocada i el resultat es guarda a la funció**. En canvi, si **no** es posen els parèntesis el que es guarda és la referència a la funció, el que permet invocar-la.

Veieu en el següent exemple les dues formes en què es pot declarar una funció i com s'assignen a un objecte:

```
function funcioPresentacio() {
  console.log(‘Hola, em dic ${this.nom}, tinc ${this.edat} anys i sóc ${this.
      ocupacio}‘);
}

const funcioComiat = function() {
  console.log(‘${this.nom} abandona la sala‘);
};

let persona = {
  nom: ’Ricard’,
  ocupacio: ’Policia’,
  edat: 40,
  presentacio: funcioPresentacio,
  comiat: funcioComiat
};

// El context d’aquestes invocacions és l’objecte persona
persona.presentacio();
persona.comiat();

// El context d’aquestes invocacions es l’espai global (window)
funcioPresentacio();
funcioComiat();
```

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

Com es pot apreciar, l'assignació com a mètode és igual en tots dos casos i, encara que les funcions s'han declarat fora de l'objecte, el context del mètode és correcte. En canvi, si s'invoquen les funcions de forma independent, el context d'aquestes és l'espai global i per tant no es troben definides les propietats `nom`, `ocupacio`, ni `edat` i mostrarà `undefined` en el seu lloc.

De la mateixa manera que les funcions, els mètodes també poden acceptar paràmetres que són aplicats exactament igual. És a dir, es crea un objecte `arguments` que pot ser manipulat, poden passar-se múltiples arguments i en cas de no passar-ne suficients, el valor d'aquests serà `undefined` però no es produirà cap error:

```
let persona = {
  nom: ’Ricard’,
  parlar: function (missatge, buit) {
    console.log(‘${this.nom} diu: ${missatge}‘);
    console.log(‘Contingut de buit: ${buit}‘);
    console.log(’Contingut d\’arguments: ’, arguments);
  }
};
```

```
9
10 persona.parlar('Bon dia!');
```

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

Conflicte de contextos

Possiblement el punt més conflictiu en treballar amb objectes és el canvi de context, ja que pot resultar molt confós saber a quin context fa referència `this`. Especialment en el moment que es comença a treballar de forma asíncrona, per exemple amb temporitzadors:

```
1  let persona = {
2    nom: 'Ricard',
3    sortir: function(temps) {
4      setTimeout(function() {
5        console.log(`${this.nom} surt de la sala ${temps}s després`);
6      }, temps * 1000);
7    }
8  };
9
10 persona.sortir(3);
```

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

Com podeu veure el resultat és confós, per una banda `this.nom` ha estat avaluat com `undefined`, en canvi el valor del paràmetre `temps` és correcte.

En el primer cas el valor és `undefined` perquè quan la funció és invocada per `setTimeout` es fa amb el seu context (l'espai global, `window`) i, per tant, `this.nom` no està definit.

Les **clausures** es tracten a la unitat "Estructures definides pel programador".

El segon cas (`temps`) és un paràmetre i, per tant, es tracta com una variable. Dins del mètode es crea una **clausura**, de manera que la funció que es passa com a argument a `setTimeout` té accés a les variables declarades dins del mètode `sortir` i per aquesta raó el valor és correcte.

Hi ha tres maneres de solucionar aquest conflicte, una consisteix a aprofitar el funcionament de les clausures:

- Creant una variable que guardi el context.
- Utilitzar el mètode `bind` (propi de totes les funcions) que estableix el context en què s'executarà la funció
- Utilitzar una funció de fletxa en lloc d'una funció anònima.

Vegeu un exemple en el qual s'apliquen les tècniques de la clausura i la utilització del mètode `bind`:

Quan es guarda el valor del context `this` en una variable s'acostuma a anomenar-la `that` o `self`.

```
1 let persona = {
```

```
  nom: 'Ricard',
  entrar: function(temps) {
    setTimeout(function() {
      console.log(`${this.nom} entra a la sala al cap de ${temps}s`);
    }.bind(this), temps * 1000);
  },
  sortir: function(temps) {
    let that = this;

    setTimeout(function() {
      console.log(`${that.nom} surt de la sala ${temps}s després`);
    }, temps * 1000);
  }
};

persona.entrar(1);
persona.sortir(3);
```

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

Com es pot apreciar, en aquest cas concret se soluciona el conflicte aprofitant la clausura i fent servir el mètode `bind`. Fer servir una tècnica o altra dependrà de les circumstàncies. Per exemple, si la funció que s'ha de cridar no es declara dins de la clausura, no serà possible passar-li el context fent servir una variable ni tampoc si es tracta d'un mètode d'un altre objecte.

bind, call i apply

Els mètodes `call` i `apply` es poden utilitzar quan la invocació es realitza immediatament (síncrona), en canvi `bind` s'utilitza quan l'execució es produirà en el futur (asíncrona).

Cal destacar que actualment és possible evitar el conflicte de canvis de context fent servir funcions de fletxa. Si en lloc de fer servir una funció anònima fem servir una funció de fletxa, el context d'execució de la funció continua sent l'objecte:

Els mètodes `call` i `apply` es tracten a la unitat "Estructures definides pel programador".

```
let persona = {
  nom: 'Ricard',
  sortir: function(temps) {
    setTimeout(() => {console.log(`${this.nom} surt de la sala ${temps}s despré
        s`)}, temps * 1000);
  }
};

persona.sortir(3);
```

Les funcions de fletxa es tracten a la unitat "Estructures definides pel programador".

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

Atés que les funciones de fletxa conserven el context original, el codi és més clar. A més a més, un dels casos d'ús més importants per les funcions de fletxes és fer-les servir com a paràmetres de funcions.

1.3.3 Actualitzar propietats

A JavaScript no existeix el concepte d'àmbit privat o públic d'altres llenguatges, així doncs, totes les propietats dels objectes són accessibles tant per llegir-les com per canviar-les (o actualitzar-les).

Canviar el valor d'una propietat és tan simple com assignar-li un nou valor tal com

es pot comprovar en l'exemple següent:

```
let persona = {
  nom: 'Ricard',
  edat: 40,
  saludar: function () {
    console.log(`Hola, em dic ${this.nom} i tinc ${this.edat} anys`);
  }
};

console.log('Salutació original');
persona.saludar();

persona.nom = 'Pere';
persona.edat++;

console.log('Salutació després de modificar l\'objecte');
persona.saludar();
```

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

Primer se li ha modificat el nom, assignant-n'hi un de nou, i seguidament s'ha incrementat l'edat fent servir l'operador ++. Com es pot apreciar, modificar el valor d'una propietat és igual que canviar el valor d'una variable.

Modificar una funció és igual de fàcil, fixeu-vos en l'exemple següent com s'actualitza el mètode `atacar` per modificar el comportament de l'objecte:

```
let persona = {
  nom: 'Ricard',
  atacar: function(objectiu) {
    console.log(`— ${this.nom} dona un cop de puny a ${objectiu}`);
  }
};

console.log('Personatge ataca desarmat:');
persona.atacar('Zombi');

console.log('Es troba una pistola i l\'equipa:');
persona.atacar = function(objectiu) {
  console.log(`— ${this.nom} dispara dos trets a ${objectiu}`);
};

console.log('Utilitza la nova arma:');
persona.atacar('Zombi');
```

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

Inicialment l'objecte `persona` quan invoca `atacar` fa servir la funció predefinida (per donar cops de puny). Però, una vegada actualitzem el mètode amb la nova funció, dispara trets a l<nocode>'</nocode>`objectiu`.

Una manera més neta de fer aquests canvis interns és afegir un mètode que s'encarregui de fer la substitució com a l'exemple següent, on s'ha afegit el mètode `equiparArma`, de manera que fer servir l'objecte és més clar:

```
let persona = {
  nom: 'Ricard',
  atacar: function(objectiu) {
    console.log(`— ${this.nom} dóna un cop de puny a ${objectiu}`);
```

```
  },
  equiparArma: function(arma) {
    this.atacar = arma;
  }
};

let pistola = function(objectiu) {
  console.log(`— ${this.nom} dispara dos trets a ${objectiu}`);
}

console.log('Personatge ataca desarmat:');
persona.atacar('Zombi');

console.log('Es troba una pistola i l\'equipa:');
persona.equiparArma(pistola);

console.log('Utilitza la nova arma:');
persona.atacar('Zombi');
```

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

Aquesta solució també és aplicable quan es treballa amb classes, ja que un cop instanciat un objecte les seves propietats i mètodes poden ser reassignats:

```
class Persona {

  constructor(nom) {
    this.nom = nom;
  }

  atacar(objectiu) {
    console.log(`— ${this.nom} dona un cop de puny a ${objectiu}`);
  }

  equiparArma(arma) {
    this.atacar = arma;
  }
}

let persona = new Persona('Ricard');
let pistola = function(objectiu) {
  console.log(`— ${this.nom} dispara dos trets a ${objectiu}`);
}

console.log('Personatge ataca desarmat:');
persona.atacar('Zombi');

console.log('Es troba una pistola i l\'equipa:');
persona.equiparArma(pistola);

console.log('Utilitza la nova arma:');
persona.atacar('Zombi');
```

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

En altres llenguatges, aplicar un comportament similar no és trivial, ja que els mètodes no es poden actualitzar directament sobre els objectes i s'ha de recórrer a l'herència o la *delegació*.

La delegació és una tècnica que es basa a encarregar les operacions d'un objecte a un altre en lloc de realitzar-les ell mateix.

L'actualització de mètodes és una clara demostració de com s'aplica el **polimorfisme** a JavaScript.

Per altra banda, com que JavaScript és un llenguatge dèbilment *tipat*, s'ha de tenir en compte que en actualitzar una propietat és possible assignar un tipus de dada incorrecte. Per exemple, si proveu el següent codi, no es produirà cap error, però el resultat no és el desitjat:

Un llenguatge és dèbilment tipat (*weak typing*) si les variables poden tenir valors de tipus diferents al llarg de l'execució del programa . En un llenguatge fortament tipat (*strong typing*), les variables sempre tenen valors del mateix tipus. En aquests llenguatges sol haver també mecanismes de conversió de tipus. A vegades als llenguatges fortament tipats se'ls anomena, simplement, tipats.

```
let persona = {
  nom: 'Ricard',
  edat: 40,
  saludar: function () {
    console.log(`Hola, em dic ${this.nom} i tinc ${this.edat} anys`);
  }
};

console.log('Salutació original');
persona.saludar();

persona.edat = 'quaranta';
persona.edat++;

console.log('Salutació després de modificar l\'objecte');
persona.saludar();
```

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

Com que s'ha canviat un nombre per una cadena de text, l'operació d'increment ja no és possible i el resultat és `NaN` (*Not a number*).

Però no només afecta les propietats. Atès que els mètodes no són més que propietats que referencien una funció, és possible actualitzar una propietat assignant com a valor una funció i a la inversa, canviar un mètode assignant-li qualsevol altra cosa:

```
let persona = {
  nom: 'Ricard',
  edat: 40,
  saludar: function () {
    console.log(`Hola, em dic ${this.nom} i tinc ${this.edat} anys`);
  }
}

console.log('Salutació original');
persona.saludar();

persona.edat = function() {
  console.log('Aquesta funció sobreescriu el valor de l\'edat');
};

persona.saludar = 42;

console.log('Contingut de la propietat edat: ', persona.edat)

console.log('Salutació després de modificar l\'objecte');
persona.saludar();
```

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

En el cas de la propietat `edat`, en lloc de mostrar el seu valor, ens mostra el codi de la funció i en el cas del mètode `saludar` es produeix un error (es pot veure a la consola de les eines de desenvolupador), ja que s'intenta invocar a `persona.saludar` i com que es tracta d'un nombre això no és possible.

A ES6 s'han afegit mètodes d'accés i actualització juntament amb la definició de classe.

Una possible solució és fer servir **mètodes d'accés** (*getters* o *accessors* en anglès) i **mètodes d'actualització** (*setters* o *mutators* en anglès). Consisteix a agregar dos mètodes per cada propietat: un per accedir i un altre per actualitzar-la, de manera que es pot fer el control del tipus dins del mètode.

Aquesta solució pot facilitar que els tipus de cada propietat i mètode siguin correctes, però a JavaScript presenta inconvenients que la fan inviable en molts casos:

- Si no es fan servir clausures tant els mètodes com les propietats continuen sent accessibles i actualitzables directament, per tant, **no es garanteix que es conservi la integritat de l'objecte**.
- **El codi es complica**, ja que per cada propietat s'han d'afegir 2 mètodes.

Així doncs, s'ha de valorar cas per cas si és raonable afegir aquesta complexitat extra. Per aquesta raó JavaScript requereix una major disciplina per part dels desenvolupadors perquè mentre en altres llenguatges aquests tipus d'error són detectats pel compilador, a JavaScript moltes vegades produeixen errors silenciosos difícilment detectables.

Un altre punt important a tenir en compte a l'hora de treballar amb objectes és que, igual que els *arrays*, aquests es passen a les funcions/mètodes per referència. És a dir, quan s'invoca una funció a la qual es passa un objecte com argument, qualsevol canvi que es produeixi afectarà l'objecte original, ja que es tracta del mateix:

```
let pistola = {
  municio: 19
};

let recarregar = function (arma) {
  arma.municio = 42;
};

console.log('Munició actual de l\'arma: ', pistola.municio);
recarregar (pistola);
console.log('Munició actual de la pistola: ', pistola.municio)
```

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

Com es pot apreciar, la propietat `municio` de l'objecte s'ha actualitzat tot i que l'operació s'ha realitzat en una funció aliena a l'objecte.

1.3.4 Augmentar objectes

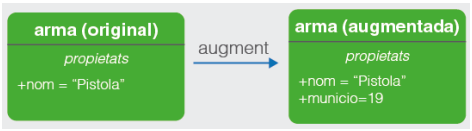
Una altra peculiaritat de JavaScript, que no es troba en gaires llenguatges, és que els objectes poden ser augmentats. És a dir, **es poden afegir noves propietats i mètodes a qualsevol objecte ja existent**.

La manera d'augmentar-los és molt simple, només cal establir el nom de la propietat i assignar-li el valor desitjat. A partir d'aquest moment la nova propietat formarà part de l'objecte com es pot veure a la figura 1.5 i a l'exemple següent:

```
let arma = {
  nom: 'Pistola'
};

arma.municio = 19;

console.log(`Munició de ${arma.nom}: ${arma.municio}`);
```

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

FIGURA 1.5. Objecte augmentat amb una propietat

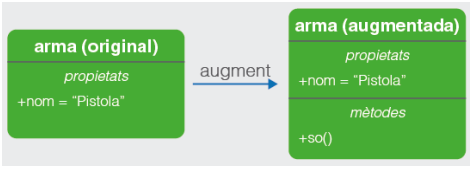


Com es pot apreciar, la declaració de l'objecte `arma` no inclou cap propietat `municio`, però una vegada es fa l'assignació aquesta s'afegeix i es pot tractar com qualsevol altra propietat. En cas de tractar-se d'una funció, l'augment es realitza exactament igual com es pot comprovar en la figura 1.6 i en l'exemple següent:

```
let arma = {
  nom: 'Pistola'
};

arma.so = function() {
  console.log('Bang!');
};

arma.so();
```

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

FIGURA 1.6. Objecte augmentat amb un mètode



Fixeu-vos que aquesta flexibilitat permet afegir noves propietats i mètodes directament o copiant-les d'altres objectes o *arrays* (iterant sobre ells fent servir la sentència `for...in`).

La possibilitat d'augmentar els objectes fa factible utilitzar-los com a diccionaris o estructures de dades.

Cal remarcar que fer ús d'aquesta característica pot portar fàcilment a errors de difícil detecció, ja que un error tipogràfic en una propietat pot causar que en lloc d'actualitzar un valor s'agregui una nova propietat amb el nom mal escrit i malauradament no es produirà cap error que us permeti detectar-lo:

```
let arma = {
  municio: 19,
  disparar: function() {
    if (this.municio > 0) {
      this.mumicio = this.municio – 1;
      console.log(‘Bang! resten ${this.municio} bales‘);
    } else {
      console.log(’Click! munició esgotada’);
    }

  }
};

arma.disparar();
arma.disparar();
arma.disparar();

console.log(arma);
```

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

Com es pot apreciar, no es produeix cap error, però el resultat és incorrecte. Dins del mètode `disparar` s'ha comès un error tipogràfic i en lloc de `municio` s'ha escrit `mumicio`, cosa que provoca un augment de l'objecte afegint una propietat amb aquest mateix nom i valor 18 (obtingut a partir de `this.municio - 1`) en lloc de reduir la propietat `this.municio`.

S'ha de tenir en compte que l'augmentació es produeix sobre els objectes, independentment de com s'hagin creat, i no té cap efecte sobre les classes.

1.3.5 Eliminar propietats i mètodes

De la mateixa manera que es pot augmentar un objecte, JavaScript ens ofereix l'operador `delete` per eliminar mètodes o propietats:

```
let arma = {
  nom: ’pistola’,
  municio: 19
};

console.log(‘Munició de ${arma.nom}: ${arma.municio}‘);
delete arma.municio;
console.log(‘Munició de ${arma.nom}: ${arma.municio}‘);
```

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

m06/pen/JjXbMJr?editors=0012.

En el mateix moment en què es crida l'operador `delete` seguit de la propietat a eliminar, aquesta deixa d'estar vinculada a l'objecte. En cas de voler eliminar propietats niuades només s'ha d'aplicar la notació de punt o de claudàtors per especificar-la, de la mateixa manera que per accedir, augmentar o actualitzar:

```
let motxilla = {
  armes: [{
    nom: 'pistola',
    atacs: 1,
    abast: 60
  }, {
    nom: 'ganivet',
    atacs: 2
  }]
};

function mostrarInventari(inventari) {
  for (let i = 0; i < inventari.armes.length; i++) {
    for (let propietat in inventari.armes[i]) {
      console.log(`${propietat}: ${inventari.armes[i][propietat]}`);
    }
    console.log('————————');
  }
};

console.log('Mostrant inventari:');
mostrarInventari(motxilla);

delete motxilla.armes[0]['abast'];

console.log('Mostrant inventari després d\'eliminar l\'abast:')
mostrarInventari(motxilla);
```

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

Fixeu-vos que per eliminar la propietat s'ha fet servir la notació de claudàtors per accedir al primer element de l'*array* i a continuació el nom de la propietat. En aquest cas s'ha optat per fer servir la notació de claudàtors en tots dos casos, però fent servir la notació de punt el resultat hauria estat el mateix: `delete motxilla.armes[0].abast`.

En el cas d'intentar accedir a una propietat esborrada el valor retornat serà `undefined`. En canvi, en invocar un mètode esborrat es produirà un error i s'aturarà l'execució, com es pot comprovar en l'exemple següent:

```
let arma = {
  nom: 'Pistola',
  so: function() {
    console.log('Bang!');
  }
};

delete arma.nom;
delete arma.so;

console.log('Nom de l\'arma: ', arma.nom);
arma.so(); // Produeix un error i s'atura l'execució
console.log('Aquesta línia mai s\'executa');
```

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

Com es pot apreciar l'última línia no s'executa mai perquè en invocar a `arma.so` es dispara una excepció de tipus `TypeError` i s'atura l'execució del codi.

Generalment l'operador `delete` s'utilitza amb diccionaris i estructures de dades per poder eliminar els elements, ja que la necessitat d'eliminar una propietat o mètode és molt menys habitual.

1.4 Definir un espai de noms

Un dels problemes de treballar amb JavaScript és la facilitat amb què es pot **contaminar l'espai global del navegador**, ja que si dins d'una funció no es declara una variable aquesta es defineix a l'espai global i els resultats poden ser imprevisibles.

L'espai global al navegador es correspon amb l'objecte `window` i amb `global` a **Node.js**.

Per altra banda, s'ha de tenir en compte que en una mateixa pàgina és habitual carregar múltiples fonts de codi JavaScript que no són controlades per nosaltres com poden ser llibreries (jQuery, Google Maps, Google Analytics, etc.) o codi afegit pels gestors de continguts.

En el cas de fer servir variables globals, és molt fàcil que alguna d'aquestes aplicacions sobreescrigui alguns dels vostres valors (o al contrari), fet que genera uns errors molt difícils de depurar, ja que tant el codi de tercers com el propi funcionarà correctament de forma individual.

També és més difícil depurar i gestionar els objectes de l'aplicació si aquests es troben barrejats amb els objectes del navegador i el codi de tercers, fixeu-vos en el contingut de l'objecte `window` amb les eines de desenvolupador del navegador:

```
1 console.log(window);
```

Com es pot apreciar, inclou tots els objectes predefinits, variables i funcions de JavaScript, a més de les que afegeixen els navegadors i això sense comptar amb el vostre codi ni carregar llibreries externes.

Hi ha dues solucions a aquest problema:

- Fer servir un **espai de noms** (*namespace* en anglès), ficant tots els components de l'aplicació es troben al seu interior. D'aquesta manera és més fàcil inspeccionar els objectes que formen part de l'aplicació al mateix temps que s'eviten possibles conflictes amb altres aplicacions.
- Fer servir **mòduls** per encapsular l'aplicació. D'aquesta manera tota l'aplicació es troba dintre de l'àmbit del mòdul i no a l'espai global.

Podeu trobar informació sobre els mòduls al següent enllaç: mzl.la/3ibdoOl i a la unitat "Objectes predefinits del llenguatge".

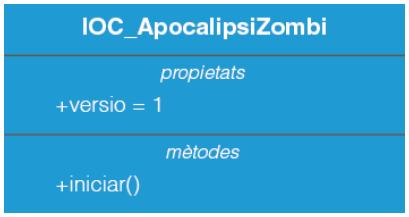
Actualment, el mètode preferit per aïllar l'aplicació és la utilització de **mòduls**.

Crear un espai de noms consisteix a crear un únic objecte que es trobarà a l'espai global del navegador, dins del qual s'afegeix la resta de l'aplicació:

```
1  // Aquest és l'espai de noms per l'aplicació
2  let IOC_ApocalipsiZombi = {
3    // Codi de l'aplicació
4    versio: 1,
5    iniciar: function() { /*Inicialització de l'aplicació */}
6  };
7
8  // Inici de l'aplicació
9  IOC_ApocalipsiZombi.iniciar();
10
11 // Contingut de l'aplicació
12 console.log(window.IOC_ApocalipsiZombi);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/mdPOpLa?editors=0012 i el diagrama corresponent a la figura 1.7.

Figura 1.7. Espai de noms que agrupa diferents components de l'aplicació



S'ha de tenir en compte que tant la propietat com el mètode s'han afegit amb finalitat demostrativa i no són obligatoris. El contingut serà el que necessiteu per la vostra aplicació.

Si us hi fixeu, tot el contingut de la vostra aplicació es troba ara dins de `window.IOC_ApocalipsiZombi`, de manera que s'eviten possibles conflictes i és més fàcil comprovar el seu contingut.

Al contrari del que passa en altres llenguatges, a JavaScript no existeix cap convenció per establir el nom de l'objecte que es fa servir com a espai de noms, per aquesta raó se'n pot fer servir qualsevol que considereu prou únic per evitar conflictes.

Podeu trobar més informació sobre el *patró mòdul* en la secció "Annexos" del web del mòdul.

El punt feble d'aquesta tècnica és que no s'encapsula la informació i, conseqüentment, es pot accedir a qualsevol de les propietats o mètodes de l'espai de noms directament. Per solucionar aquest problema es pot aplicar el **patró mòdul**, una tècnica basada en l'ús de clausures i funcions autoexecutables.

En conclusió, es recomana fer servir sempre un mòdul o un espai de noms per encapsular la vostra aplicació, encara que això no s'aplica al codi d'exemples o demostracions perquè causaria una complicació innecessària del codi.

1.5 Prototipus

JavaScript és un llenguatge basat en prototipus, és a dir, els objectes poden construir-se a partir d'uns altres objectes (el seu prototipus) i seguidament augmentar-lo amb nous mètodes i propietats.

> A causa de deficiències del propi llenguatge el funcionament d'aquest mecanisme porta fàcilment a confusió. Per aquest motiu a partir de la versio ECMAScript 2015 **es recomana treballar sempre amb classes**, ja que la seva finalitat és molt semblant.

Un prototipus és un objecte del qual altres objectes hereten propietats i mètodes. Aquest objecte només es troba definit a les funcions, com a propietat `prototype` i és possible actualitzar-lo o augmentar-lo.

Quan s'utilitza una funció com a **constructor**, el prototipus de l'objecte creat referencia al de la funció constructora, de manera que aquest hereta totes les propietats i mètodes establerts al prototipus. Conseqüentment, només els objectes generats a partir d'aquest patró tenen accés al prototipus automàticament.

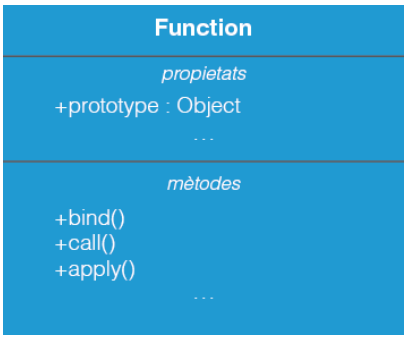
> Un **constructor** a JavaScript és una funció que crea un objecte. Per convenció, el nom d'aquesta funció sempre comença amb majúscules i la creació de l'objecte es realitza amb l'operador `new`.

```
let Persona = function (nom) {
  this.nom = nom;
};

Persona.prototype.saludar = function () {
  console.log('Hola, em dic ${this.nom}');
};

let jugador = new Persona('Ricard');
jugador.saludar();
```

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

Cal recordar que les funcions a JavaScript són objectes i, per consegüent, tenen propietats i mètodes. Podeu veure'n alguns a la figura 1.8.

Figura 1.8. Representació d'una funció



S'ha de tenir en compte que la propietat `prototype` dels objectes generats a partir d'un constructor és una referència al prototipus de la funció constructora i, conseqüentment, si s'augmenta o actualitza en un dels objectes aquest canvi afectarà el prototipus del constructor i de totes les instàncies generades.

Per aquesta raó **es desaconsella modificar el prototipus d'un objecte generat a partir d'un constructor** fent servir la seva propietat `prototype`. Si realment es vol modificar el seu prototipus, s'han d'aplicar els canvis a la propietat `prototype` de la funció constructora.

1.5.1 Augmentar objectes predefinits

Gràcies als prototipus és possible augmentar també els objectes predefinits del sistema, per exemple els objectes `Array`, `String` o `Number`; ja que una vegada augmentat un prototip s'aplica a tots els objectes dels quals forma part immediatament.

No és possible invocar un mètode a partir d'un número literal (per exemple `5`), però és posible invocar-lo a partir d'una variable amb el valor `5` assignat com a literal.

Per exemple, es pot augmentar el prototip de `Number` per afegir un mètode anomenat `duplicar` a tots els nombres:

```
let x = 9.5;

Number.prototype.duplicar = function () {
  return this * 2;
};

let y = 21;

console.log(x.duplicar());
console.log(y.duplicar());
```

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

Com es pot apreciar, una vegada modificat el prototip de `Number`, tots els nombres de JavaScript tenen accés a aquest mètode, tant els que existien abans de l'augment com els nous.

Fixeu-vos també que s'ha fet servir `this` dins del nou mètode i el context s'ha

aplicat correctament, ja que en invocar al mètode `duplicar` primer s'ha cercat entre els mètodes de l'objecte i a continuació entre els mètodes del seu prototip, no es tracta d'una funció externa.

Cal remarcar que d'aquesta manera es poden augmentar objectes per accedir a una nova funcionalitat globalment, fins i tot entre diferents espais de nom, sense contaminar l'espai global. Veieu a continuació un altre exemple, aquest cop amb `String`:

```
String.prototype.revertir = function() {
  let revertit = '';
  for (let i = this.length - 1; i >= 0; i--) {
    revertit += this[i];
  }
  return revertit;
};

console.log('Bang!'.revertir());
```

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

En aquest cas, al contrari que amb els nombres, és possible cridar directament al mètode a partir de la cadena de caràcters.

1.6 Classes en JavaScript

Actualment, tots els navegadors admeten l'ús de classes de JavaScript, això ens permet treballar amb un sistema d'herència clàssica en lloc de l'herència prototípica pròpia de JavaScript, afegir mètodes d'accés, mètodes actualitzadors i mètodes estàtics.

1.6.1 Mètodes d'accés i actualitzadors (getters i setters)

Les classes permeten afegir mètodes d'accés i actualitzadors directament a les classes (cosa que també es pot fer amb ES5, però resulta més farragós):

```
class Persona {
  constructor(nom, cognom) {
    this._nom = nom;
    this._cognom = cognom;
  }

  set nom(nom) {
    this._nom = nom;
  }

  get nom() {
    return this._nom;
  }

```

```
  set cognom(cognom) {
    this._cognom = cognom;
  }

  get cognom() {
    return this._cognom;
  }

  get nomComplet() {
    return `${this._nom} ${this._cognom}`;
  }
}

let ricard = new Persona('Ricard', 'Ensutjs');

console.log('Nom (amb accessor):', ricard.nom);
console.log('Cognom (amb accessor):', ricard.cognom);
console.log('Nom complet:', ricard.nomComplet);
console.log('Nom (propietat):', ricard._nom);
console.log('Cognom (propietat):', ricard._cognom);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/gOrLEZv?editors=0012. </newcontent>

Fixeu-vos en l'*accessor* a `nomComplet` i com s'utilitza. No s'invoca, sinó que s'hi accedeix com si es tractés d'una propietat, però realment no existeix, el que es fa és generar-la dinàmicament a partir de les propietats `nom` i `cognom`. De la mateixa manera s'accedeix a les propietats `nom` i `cognom` que són generades dinàmicament (però sense variació) a partir de `_nom` i `_cognom` respectivament.

Malauradament, com es pot apreciar, les propietats `_nom` i `_cognom` continuen sent accessibles (i actualitzables) i per tant no s'està aplicant l'encapsulació de la informació. Per aquesta raó, la utilització d'*accessors* i actualitzadors només és interessant en casos molt marginals (per exemple, per generar propietats dinàmiques com el cas de `nomComplet`).

L'encapsulament de la informació encara no forma part de cap especificació ni esborrany, per tant cal recórrer a mètodes alternatius que escapen a l'abast d'aquests materials. Així doncs, la recomanació és fer servir el prefix '_' per distingir-les i en cas de requerir mètodes o propietats privades utilitzar la implementació del patró constructor o funcional per aquests objectes en lloc de les classes d'ES6.

Podeu trobar mètodes d'implementar l'encapsulació a JavaScript en l'enllaç següent: goo.gl/2t6jfU.

1.6.2 Mètodes estàtics

Les classes a JavaScript també permeten crear mètodes estàtics, és a dir, mètodes que poden utilitzar-se sense instanciar cap classe, per exemple:

```
class FactoriaArmes {
  static crearArma({tipus}) {
    let arma;

    switch (tipus) {
      case 'simple':
        arma = new Arma(arguments[0]);
```

```
        break;
      case 'arma de foc':
        arma = new ArmaAmbMunicio(arguments[0]);
        break;
      default:
        console.error(`No existeix aquest tipus d´\arma: ${params.tipus}`);
    }

    return arma;
  }
}
```

Tal com es pot apreciar només consta d'un mètode, però va precedit per la paraula clau `static`, això indica que es tracta d'un mètode estàtic i s'hi pot accedir sense instanciar la classe `FactoriaArmes`.

En aquest cas, la factoria crea instàncies de les classes `Arma` i `ArmaAmbMunicio` que han d'implementar un constructor que accepti un objecte com a paràmetre, per exemple:

```
class Arma {
  constructor: ({nom, potencia}) {
    // cos del constructor
  }
}

class ArmaAmbMunicio extends Arma {
  constructor: ({nom, potencia, maxMunicio}) {
    // cos del constructor
  }
}
```

Per instanciar els nous objectes a partir de la factoria i utilitzar els objectes es faria de la següent manera:

```
let ganivet = FactoriaArmes.crearArma({
  tipus: 'simple',
  nom: 'Ganivet',
  potencia: 1
});

let pistola = FactoriaArmes.crearArma({
  tipus: 'arma de foc',
  nom: 'Pistola',
  maxMunicio: 2,
  potencia: 2
});

ganivet.atacar('Zombi');
pistola.atacar('Gos Zombi');
pistola.atacar('Gos Zombi');
pistola.atacar('Gos Zombi');
```

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

Fixeu-vos que no s'instancia cap objecte fent servir l'operador `new`, es generen cridant al mètode estàtic `crearArma` que rep un objecte amb l'estructura de dades que conté la informació necessària per instanciar cada objecte.

Els mètodes estàtics resulten útils per la implementació de diferents patrons de

disseny com ara les *factories* o la creació de biblioteques, però s'ha de tenir en compte la complexitat afegida per encapsular la informació utilitzant el sistema de classes si realment és necessari utilitzar propietats o mètodes privats.

1.7 Herència

El paradigma de la programació orientada a objectes, a banda de permetre crear abstraccions de la realitat, facilita la reutilització del codi gràcies a l'herència.

En els llenguatges clàssics, l'herència permet crear noves classes que hereten d'unes altres. Això permet modificar-ne el comportament (polimorfisme) o augmentar-les.

Cal tenir en compte que a JavaScript no existeix el concepte d'**interfície** (*interface* en anglès) d'altres llenguatges clàssics, ja que no es pot forçar el compliment del "contracte" que representa la interfície en tractar-se d'un llenguatge dèbilment tipat.

> **Interficies**
>
> Una interfície és el conjunt de mètodes que un objecte ha de tenir per permetre la comunicació amb uns altres. Així doncs, tots els objectes amb aquests mètodes podran utilitzar-se indistintament com a component. Atès que l'especificació de JavaScript no inclou les interfícies i, consegüentment, no es pot forçar el comportament, el desenvolupador ha de parar atenció per respectar aquest "contracte".

Per altra banda, encara que JavaScript no suporta l'herència múltiple (un objecte que hereti de múltiples objectes), és possible implementar-la gràcies a la gran flexibilitat del llenguatge. Per exemple, utilitzant una funció per crear els *mix-ins* (mescles d'objectes).

Es pot trobar més informació sobre els *mix-ins* al següent enllaç: es.wikipedia.org/wiki/Mixin.

En aquests materials ens centrarem en l'herència clàssica fent servir classes en lloc de prototips.

1.7.1 Implementar l'herència

JavaScript ofereix diverses tècniques per reaprofitar el codi, ja que es poden crear jerarquies d'objectes utilitzant el sistema de classes o aplicant diferents patrons de generació (fins i tot combinant-los) amb alguna petita modificació, és a dir, no està restringit només al sistema de classes.

Podeu trobar més informació i exemples sobre les tècniques alternatives a l'ús de constructors per implementar l'herència prototípica en la secció "Annexos" d'aquesta unitat.

Cal tenir en compte que independentment del sistema empleat per implementar l'herència, els objectes generats funcionen exactament de la mateixa manera.

Herència a partir d'altres objectes

La implementació de l'herència fent servir aquesta tècnica és la més simple de totes. Consisteix a crear un primer objecte que serà el pare dels altres objectes (contindrà les propietats i mètodes comuns) i a continuació invocar `Object.create` per crear noves instàncies que podran ser actualitzar-se i augmentar-se.

Herència a través de la generació funcional

Les modificacions que s'han de fer per afegir un objecte pare a aquest patró són mínimes. Només cal substituir la creació de l'objecte buit per la generació de l'objecte pare, que seguidament serà actualitzat i augmentat segons sigui necessari.

Herència a partir de constructors

La creació d'objectes a partir de constructors és la tècnica que més s'assembla a l'herència clàssica sense utilitzar classes, ja que s'afegeix la propietat `prototype` que permet distingir quins objectes tenen un mateix pare, aplicar la cadena de prototipus i cridar mètodes del constructor pare.

La creació de constructors especialitzats requereix aplicar els canvis següents:

- Dins del constructor s'ha de cridar al constructor pare passant com a context el nou objecte i els arguments: `ConstructorPare.apply(this, arguments);`.
- La propietat `prototype` del constructor s'ha de substituir per una còpia de la del constructor pare: `ConstructorFill.prototype = Object.create(ConstructorPare.prototype);`.
- Una vegada substituït el prototipus s'ha de corregir el constructor perquè apunti al nou constructor: `ConstructorFill.prototype.constructor = ConstructorFill`.

Com podeu imaginar, la implementació de l'herència fent servir constructors es força complicada, per aquest motiu es recomana fer servir el sistema d'herència mitjançant classes que es va afegir a ES2015.

Herència a mitjançant el sistema de classes

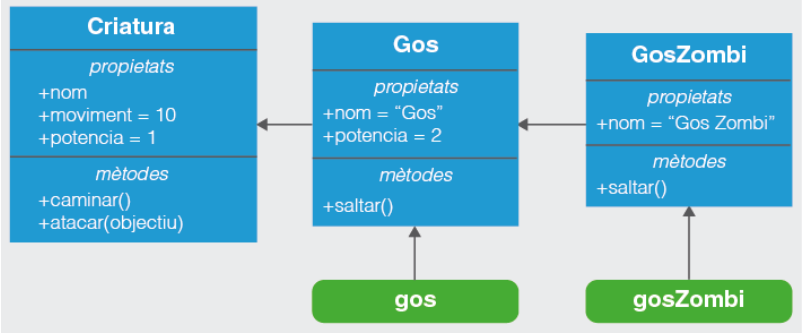
Actualment JavaScript permet utilitzar el sistema de classes molt similar al que trobem en altres llenguatges com Java o C++, això ens permet crear subclasses fàcilment, només cal utilitzar la paraula clau `extends`:

```
1 // Superclasse
2 class Criatura {
3   constructor() {
```

```
    this.nom = 'Criatura de tipus desconegut';
    this.moviment = 10;
    this.potencia = 1;
  }

  caminar() {
    console.log(`${this.nom} camina ${this.moviment} metres`);
  }

  atacar(objectiu) {
    console.log(`${this.nom} ataca a ${objectiu.nom} i li causa ${this.potencia
        } punts de dany`);
  }
}

class Gos extends Criatura {
  constructor() {
    super();
    this.nom = "Gos";
    this.potencia = 2;
  }

  saltar() {
    console.log(`${this.nom} salta ${this.moviment / 5} metres`);
  }
}

class GosZombi extends Gos {
  constructor() {
    super();
    this.nom = "Gos Zombi";
  }

  saltar() {
    console.log(`${this.nom} intenta saltar... però no pot`);
  }
}

// Es generen les instancies del objectes a partir de les classes
let gos = new Gos(),
    gosZombi = new GosZombi();

gos.caminar();
gos.atacar(gosZombi);
gos.saltar();
gosZombi.atacar(gos);
gosZombi.saltar();
gosZombi.caminar();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyNKPr?editors=0012 i el diagrama corresponent a la figura 1.9.

FIGURA 1.9. Herència fent servir classes



Els noms de les classes comencen amb majúscula mentre que els objectes instanciats ho fan amb minúscula.

Tal com es pot apreciar, encara que els mètodes `caminar` i `atacar` no s'han definit a les classes `Gos` ni `GosZombi`, es poden invocar correctament perquè totes dues tenen com a superclasse `Criatura`, que és on es troben definits aquests mètodes.

Per altra banda, podem veure que totes dues subclasses reassignen el valor de la propietat `nom` dins del constructor, per consegüent encara que no s'hagi indicat cap nom en crear les instàncies s'assigna automàticament.

La classe `Gos` afegeix el mètode `saltar`, augmentant la classe `Criatura` i la classe `GosZombi` la sobreescriu, modificant el comportament (polimorfisme). És a dir, quan s'invoca aquest mètode des d'una instància de `Gos` es crida al mètode de `Gos`, però quan s'invoca des d'una instància de `GosZombie` el codi executat és el corresponent al mètode definit en `GosZombie`.

Quant a jerarquia de classes, es pot comprovar que es creen les referències correctament i es pot determinar si un objecte és instància d'una classe determinada. Proveu d'afegir les següents línies a l'exemple anterior:

```
console.log('GosZombie és una instància d\'Object?', gosZombi instanceof Object);
console.log('GosZombie és una instància d\'Criatura?', gosZombi instanceof Criatura);
console.log('GosZombie és una instància d\'Gos?', gosZombi instanceof Gos);
console.log('GosZombie és una instància d\'GosZombie?', gosZombi instanceof GosZombi);
```

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

Fixeu-vos que dins dels constructors de les subclasses s'ha utilitzat la paraula clau `super`. Aquesta paraula clau es pot fer servir de dues maneres diferents:

```
super([arguments]); // crida al constructor de la superclasse amb els arguments
super.funcioEnSuperclasse([arguments]); // crida a la funció concreta de la superclasse
```

Cal tenir en compte que encara que els arguments són opcionals, s'han de passar els paràmetres adequats en invocar `super` (o a la funció de la superclasse), en cas contrari el resultat no serà l'esperat. Per exemple, si el constructor de la superclasse espera el paràmetre `nom` i des de la subclasse no es passa el seu valor serà `undefined`.

> Tingueu en compte que dins dels constructors cal invocar a `super` **abans de poder utilitzar** `this`, en cas contrari es produirà un error.

Veiem un exemple d'utilització de `super` en les dues situacions:

```
class Arma {
  constructor(nom) {
    this.nom = nom;
    this.so = 'Zas!'
    this.potencia = 1;
  }

  atacar(objectiu) {
```

```
9       console.log(`${this.so} S´\han causat ${this.potencia} punts de dany a ${
            objectiu} amb ${this.nom}`);
10    }
11  }
12
13  class ArmaAmbMunicio extends Arma {
14    constructor(nom, maxMunicio) {
15      super(nom);
16      this.municio = maxMunicio;
17      this.so = 'Bang!';
18      this.potencia = 2;
19    }
20
21    atacar(objectiu) {
22      if (this.municio > 0) {
23        this.municio--;
24        super.atacar(objectiu);
25      } else {
26        console.log("Click! no queda munició!");
27      }
28    }
29  }
30
31  let ganivet = new Arma('Ganivet');
32  let pistola = new ArmaAmbMunicio('Pistola', 2);
33
34  ganivet.atacar('Zombi');
35  pistola.atacar('Gos Zombi');
36  pistola.atacar('Gos Zombi');
37  pistola.atacar('Gos Zombi');
```

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

Com es pot apreciar, quan s'invoca `atacar` des de la pistola només es comprova que hi hagi munició, si hi ha munició es crida al mètode `atacar` de la superclasse i, en cas contrari, mostra un missatge.

Fixeu-vos que quan s'invoca a `super` en el constructor de `ArmaAmbMunicio` només es passa el paràmetre `nom` i no `maxMunicio`, ja que la superclasse només accepta un paràmetre.

1.7.2 Herència múltiple: mix-ins

L'herència múltiple consisteix en el fet que una classe hereti de múltiples classes. Per exemple es poden definir dues classes diferents que no estiguin relacionades però que tinguin subclasses que comparteixin algun comportament.

Utilitzant l'herència simple hauríem de duplicar el codi del comportament compartit a cada subclasse, en canvi utilitzant l'herència múltiple es podria definir el nou comportament en una classe externa i fer que les subclasses heretin de la seva superclasse i d'aquesta nova classe.

Mix-ins

És una associació de mètodes i propietats que, després, poden ser heretats per qualsevol classe.

Tot i que JavaScript no permet l'herència múltiple directament, es pot implementar utilitzant *mix-ins*. Per crear un mix-in, en lloc de definir una classe, el que fem és definir una funció que:

- rep com a paràmetre la classe a la qual s'ha d'afegir el mix-in.
- retorna una nova classe que té com a superclasse la classe passada com a paràmetre.

Per exemple, per crear un nou mix-in que afegeixi un nou comportament definirí-em la funció així:

```
let nouMixin = Base => class extends Base {
  nouComportament() { }
};
```

Fixeu-vos que la funció s'ha definit com una funció de fletxa. Això no és necessari però fa que el codi sigui més net.

Tingueu en compte que en aquest exemple només hem afegit un nou comportament, però un mix-in pot contenir qualsevol quantitat de mètodes i propietats, com una classe normal.

Per crear una subclasse que utilitzi aquest mix-in es faria de la següent manera:

```
class ClasseBase {
  // mètodes i propietats de ClasseBase
};

class NovaClasse extends nouMixin(ClasseBase) {
  // mètodes i propietats de NovaClasse
}
```

Com es pot apreciar, es crida a la funció `nouMixin` passant com a paràmetre la classe que volem fer servir com a classe base. Si volguéssim afegir més mix-ins només caldria niuar les invocacions a les funcions de manera que el resultat d'una passa com argument de la següent:

```
class NovaClasse extends nouMixin1(nouMixin2(nouMixin3(ClasseBase))) {
  // mètodes de NovaClasse
}
```

Vegeu a continuació un exemple complet:

```
class Arma {
  constructor({nom}) {
    this.nom = nom;
    this.so = "Zas!";
    this.potencia = 1;
  }

  atacar(objectiu) {
    console.log(
      `${this.so} S\'han causat ${this.potencia} punts de dany a ${objectiu}
          amb ${this.nom}`
    );
  }
}

let carguesMixin = (Base) =>
  class extends Base {
    constructor({cargues}) {
      super(arguments[0]);
      this.cargues = cargues;
```

```
      this.maxCargues = cargues;
    }

    recarregar() {
      this.cargues = this.maxCargues;
      console.log(`${this.nom} recarregada`);
    }
  };

class ArmaDeFoc extends carguesMixin(Arma) {
  constructor({nom}) {
    super(arguments[0]);
    this.so = "Bang!";
    this.potencia = 2;
  }

  atacar(objectiu) {
    if (this.cargues > 0) {
      this.cargues--;
      super.atacar(objectiu);
    } else {
      console.log("Click! no queda munició!");
    }
  }
}

let ganivet = new Arma({nom: "Ganivet"});
let pistola = new ArmaDeFoc({nom: "Pistola", cargues: 2});

ganivet.atacar("Zombi");
pistola.atacar("Gos Zombi");
pistola.atacar("Gos Zombi");
pistola.atacar("Gos Zombi");
pistola.recarregar();
pistola.atacar("Gos Zombi");


class Dispositiu {
  constructor ({nom}) {
    this.nom = nom;
  }

  utilitzar() {
    console.log(`Utilitzant ${this.nom}`);
  }
}

class DispositiuAmbPiles extends carguesMixin(Dispositiu) {

  utilitzar() {
    if (this.cargues > 0) {
      this.cargues--;
      super.utilitzar();
    } else {
      console.log("Piles esgotades!");
    }
  }
}

let llinterna = new DispositiuAmbPiles({nom: 'llinterna', cargues: 3});
llinterna.utilitzar();
llinterna.utilitzar();
llinterna.utilitzar();
llinterna.utilitzar();
```

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

Fixeu-vos que s'ha fet servir un objecte com a paràmetre dels constructors (on són desestructurats). D'aquesta manera es poden passar els paràmetres per nom sense importar l'ordre. Això permet que el mix-in només agafi el paràmetre que necessita `cargues`, mentre que les superclasses agafen el paràmetre `nom`.

Atès que el constructor rep un objecte desestructurat, no podem passar a `super` només el nom del paràmetre, ja que llavors es perdria la resta d'informació passada originalment com objecte. Afortunadament podem accedir a l'objecte original mitjançant l'objecte `arguments`, i com només es passa un paràmetre podem assegurar que l'índex 0 conté aquest objecte. Per consegüent, amb `super(arguments[0])` es passa l'objecte original als constructors de les classes pares.

objecte arguments

És un objecte especial similar a un array que rep cada funció i conté tots els paràmetres indexats per ordre. És a dir, l'índex 0 conté el primer paràmetre, l'índex 1 el segon, l'índex 2 el tercer, etc.

Com es pot apreciar, s'han fet servir dues classes base diferents:, `Arma` i `Dispositiu`, i s'ha afegit el mix-in `carguesMixin` a les subclasses `ArmaDeFoc` i `DispositiuAmbPiles`, de manera que totes dues compareixen les propietats `cargues` i `maxCargues` i el mètode `recargar`.

En conclusió, la utilització dels mix-ins permet afegir a subclasses, que pertanyen a diferents jerarquies de classes, un comportament compartit sense duplicar el codi.

1.7.3 Patró: factoria

A causa de la complexitat que pot presentar la generació d'objectes i les jerarquies d'herència a JavaScript, de vegades pot ser recomanable utilitzar alguna tècnica que simplifiqui aquesta tasca.

Una opció és utilitzar el **patró factoria**, un patró de disseny que amaga la implementació de la creació d'objectes i exposa un mètode a través del qual es poden obtenir instàncies dels objectes sense necessitat de conèixer la seva implementació.

Podeu trobar més informació sobre el *patró factoria* en l'apartat "Annexos" del web del mòdul.

És a dir, la implementació d'aquest patró ens ajuda a crear un objecte a través del qual invocant un mètode es generen completament altres objectes, per exemple una *factoria* d'armes ens permet invocar els mètodes: `factoria.crearArma('ganivet')`, `factoria.crearArma('subfusil')`, etc.

Patrons de disseny

Els patrons de disseny són solucions comunes a problemes de disseny de programari. Van ser introduïts pel *Gang of Four* el 1994. Aquestes solucions són aplicables a tots els llenguatges de programació. Podeu trobar-ne més informació en l'enllaç següent: bit.ly/3bznlEV.

1.7.4 Composició i delegació

Una tècnica molt potent que afegeix una gran flexibilitat a la utilització d'objectes és la **composició**. Consisteix a afegir objectes com a propietats de l'objecte compost, de manera que es pot dividir la complexitat entre diferents components.

> Un dels principis de la programació orientada a objectes recomana **afavorir la composició per sobre de l'herència** (*composition over inheritance* en anglès), ja que la composició és molt més flexible i l'herència pot portar fàcilment a la duplicació de codi i la generació de jerarquies d'objectes molt més complexes.

Per determinar si cal utilitzar una composició o una relació d'herència podeu comprovar si la relació entre els dos objectes és de *ser* o de *tenir*:

- Un gos zombi **és** un zombi, per tant, és una relació d'**herència**.
- Ricard **té** una arma, així doncs, es tracta d'una **composició**.

Simplificació dels diagrames

Per simplificar els diagrames s'ha optat per fer servir la representació d'associació (una línia contínua) quan la composició és amb un sol objecte i la d'agregació (rombe contigu a l'objecte compost) quan són múltiples (per exemple, un *array* d'objectes).

Per altra banda, per fer més entenedors els diagrames es mostra tant la propietat a l'objecte com l'associació, encara que allò correcte és que només s'utilitzi una representació o l'altra (associació/composició o propietat).

A diferència dels llenguatges clàssics, a JavaScript qualsevol objecte pot formar part d'una composició (no hi ha restriccions per classe o interfície). Això permet una gran flexibilitat però, per altra banda, és possible establir com a propietats objectes que no implementin els mètodes necessaris.

En el següent exemple podeu veure com les instàncies de `Persona` estan compostes per una propietat, `arma`, on s'espera que s'estableixi un objecte amb el mètode `atacar`. Una vegada es detecta que s'ha establert la propietat, en invocar al mètode `atacar` es delega la invocació a l'objecte emmagatzemat a `arma`.

D'aquesta manera només actualitzant la propietat `arma` es pot canviar el comportament de l'objecte, per exemple: quan es fa servir el ganivet es pot atacar un nombre il·limitat de vegades, però en utilitzar la `pistola` s'afegeix una complexitat extra ja que s'ha afegit el control de munició.

Fixeu-vos que sense haver de tocar la implementació de `Persona` es podrien crear nous tipus d'armes amb funcionaments molts diferents, sempre que complissin amb els següents requisits:

- El nou tipus d'arma té assignada a la propietat `atacar` una funció, és a dir, és tracta d'un mètode.
- Aquest mètode accepta un paràmetre, que serà l'objectiu de l'atac.
- Opcionalment, el constructor podrà rebre un o dos paràmetres, corresponents a les propietats `nom` i la `potencia`.

```
1 class Arma {
2   constructor(nom, potencia) {
```

```
    this.nom = nom;
    this.potencia = potencia;
  }

  atacar(objectiu) {
    console.log(`Zas! ${this.personatge.nom} ataca amb ${this.nom} i causa ${
        this.potencia} punts de dany a ${objectiu}`);
  }
}

class ArmaAmbMunicio extends Arma {
  constructor (nom, potencia, municio) {
    super(nom, potencia);
    this.municio = municio;
  }

  atacar (objectiu) {
    if (this.municio > 0) {
      console.log(`Bang! ${this.personatge.nom} ataca amb ${this.nom} i causa $
          {this.potencia} punts de dany a ${objectiu}`);
      this.municio--;
    } else {
      console.log('Click! no hi ha munició!');
    }
  }
}

class Persona {
  constructor (nom) {
    this.nom = nom;
  }

  atacar(objectiu) {
    if (this.arma) {
      this.arma.atacar(objectiu)
    } else {
      console.log('No es pot atacar perquè no hi ha cap arma equipada!');
    }
  }

  equipar(arma) {
    this.arma = arma;
    arma.personatge = this;
    console.log(this.nom + ' ha equipat ' + arma.nom);
  }
}


let ricard = new Persona('Ricard');
let ganivet = new Arma('Ganivet', 2);
let pistola = new ArmaAmbMunicio('Pistola', 3, 2);

ricard.atacar('Zombi');
ricard.equipar(ganivet);
ricard.atacar('Zombi');
ricard.equipar(pistola);
ricard.atacar('Zombi');
ricard.atacar('Zombi');
ricard.atacar('Zombi');
```

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

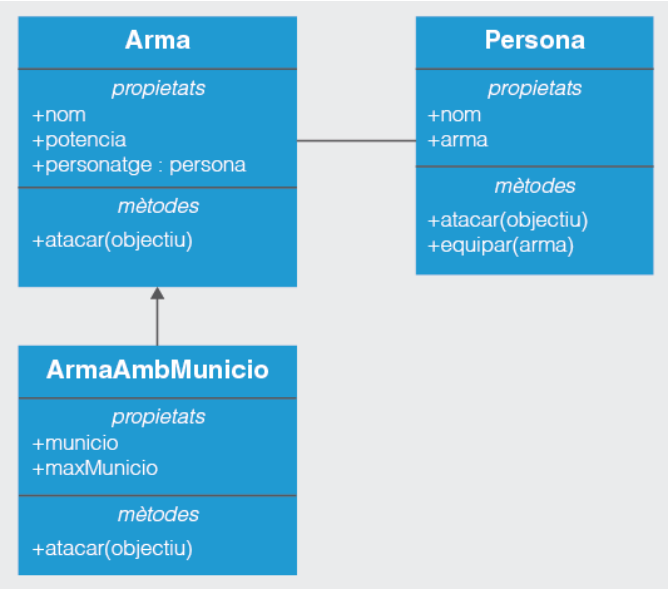
i el diagrama corresponent a la figura 1.10.

Tal com es pot apreciar, la clau de la composició és assignar un objecte a una propietat (en aquest cas a través del mètode `equipar`) i delegar de-

terminades accions a aquests objectes (com el mètode `atacar` de l'exemple: `this.arma.atacar(objectiu);`).

Com que `ArmaAmbMunicio` hereta d' `arma` i aquesta té el mètode `atacar` que és el requisit per poder equipar-lo (encara que no se'n força la comprovació), es pot assegurar que, encara que no s'actualitzés el valor d' `atacar`, tots els objectes instanciats per aquest constructor es podran equipar correctament.

FIGURA 1.10. Constructor d'objecte compost Persona amb Arma



S'ha afegit la propietat 'personatge' amb valor 'undefined' perquè és afegida per la Persona quan s'invoca el mètode equipar.

La composició es pot realitzar de diverses maneres. En aquest cas s'ha fet a través d'un mètode, però es podria haver generat l'objecte al constructor. Proveu de substituir la declaració del constructor de `Persona` per la següent:

```
constructor (nom) {
  this.nom = nom;
  let puny = new Arma('Cop de puny', 1);
  this.equipar(puny);
}
```

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

En aquest cas s'ha creat un nou objecte `puny` i s'ha equipat. Fixeu-vos que no s'ha assignat directament a la propietat `arma`, sinó que s'ha cridat al mètode `equipar`. Pareu atenció a la declaració del mètode:

```
equipar(arma) {
  this.arma = arma;
  arma.personatge = this;
  console.log(this.nom + ' ha equipat ' + arma.nom);
}
```

Primerament es mostra un missatge i seguidament s'actualitza la propietat `arma` amb l'objecte passat com a paràmetre. Però, a continuació, s'augmenta l'objecte `arma` amb la propietat `personatge` i el valor `this`. El valor de `this` en aquest

cas és la instància de `Persona`. Així doncs, si la instància de `Persona` és l'objecte referenciat `ricard`, en equipar un ganivet es pot entendre que `arma.personatge = ricard;`.

D'aquesta manera, `ganivet` té la informació de qui l'ha equipat i pot fer servir `this.personatge.nom` per accedir al `nom` de la `Persona`. És a dir, l' `Arma` té accés a tota la informació de la `Persona` que l'equipa i per tant en un cas real es podria fer servir per determinar la probabilitat d'encertar un atac, sumar la potència a la del personatge, etc.

Un altre detall a tenir en compte és que s'ha aplicat l'augment al paràmetre `arma` i no a la propietat `this.arma` (que faria referència a la instància de `Persona`). Es podria haver fet de les dues maneres:

```
1 arma.personatge = this; // Paràmetre
2 this.arma.personatge = this; // Propietat d'instància de Persona
```

La raó per la qual el resultat és el mateix d'una forma o l'altra és que `arma` és un objecte i, per tant, s'està treballant amb una referència a aquest i qualsevol canvi al paràmetre o a la propietat afecten el mateix objecte.

Esdeveniments. Manejament de formularis

Xavier Garcia Rodriguez

Índex

Introducció

Una de les principals necessitats que cobreixen les aplicacions en l'entorn client és la de gestionar l'entrada de dades a través de les interfícies web i, conseqüentment, per poder portar a terme aquestes tasques, és imprescindible ser capaços de gestionar els esdeveniments que es produeixen a l'aplicació, així com de gestionar l'entrada de dades.

En aquesta unitat veureu com mitjançant la detecció d'*events* es pot controlar quan l'usuari ha modificat un camp d'un formulari, ha clicat sobre un botó o ha finalitzat un element d'una pàgina; mentre que l'entrada de dades a través de formularis permet a les aplicacions comunicar-se amb altres components interns o servidors remots.

En l'apartat "Programació d'events" aprendreu què és un *event* en JavaScript i quina diferència hi ha amb un *esdeveniment*. Seguidament, descriurem alguns dels *events* reconeguts pels diferents components de JavaScript:

- *Events* disparats per accions del ratolí
- *Events* disparats per accions del teclat
- Altres *events* disparats pel DOM
- *Events* afegits a HTML5

També es farà una presentació dels web API més utilitzats, juntament amb els *events* que hi estiguin relacionats.

L'apartat "Programació amb formularis" comença fent una introducció als formularis en l'entorn client. Seguidament, aprendreu com s'estructura un formulari i quins elements d'HTML en formen part. També aprendreu a modificar-ne l'aparença i el comportament.

A continuació, veureu com s'han de validar els formularis tant a través de les noves característiques afegides a HTML5 com fent servir les vostres pròpies implementacions amb JavaScript o expressions regulars.

Per acabar, mostrarem com es treballa amb galetes i com podeu crear les vostres pròpies funcions per guardar i recuperar la informació que hi ha emmagatzemada, així com a utilitzar una de les Web API introduïda a HTML5: Web Storage.

Per assimilar els continguts d'aquesta unitat es molt important provar tots els exemples en el vostre equip o a CodePen i modificar-los per comprovar com afecten aquests canvis al codi. A més a més es recomana consultar la web Mozilla Developer Network (www.developer.mozilla.org) per aprofundir en els diferents aspectes tractats.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions web interactives integrant mecanismes de maneig d'esdeveniments.

- Reconeix les possibilitats del llenguatge de marques relatives a la captura dels esdeveniments produïts.
- Identifica les característiques del llenguatge de programació relatives a la gestió dels esdeveniments.
- Diferencia els tipus d'esdeveniments que es poden manejar.
- Crea un codi que capturi i utilitzi esdeveniments.
- Reconeix les capacitats del llenguatge relatives a la gestió de formularis web.
- Valida formularis web utilitzant esdeveniments.
- Utilitza expressions regulars per facilitar els procediments de validació.
- Prova i documenta el codi.

2. Desenvolupa aplicacions web analitzant i aplicant les característiques del model d'objectes del document.

- Reconeix el model d'objectes del document d'una pàgina web.
- Identifica els objectes del model, les seves propietats i els seus mètodes.
- Identifica les diferències que presenta el model en diferents navegadors.
- Crea i verifica un codi que accedeixi a l'estructura del document.
- Crea nous elements de l'estructura i en modifica elements ja existents.
- Associa accions als esdeveniments del model.
- Programa aplicacions web de manera que funcionin en navegadors amb diferents implementacions del model.
- Independitza les tres facetes (contingut, aspecte i comportament), en aplicacions web.

1. Programació d'events

Dominar la programació d'*events* és una destresa imprescindible per qualsevol desenvolupador web, ja que tots els elements d'HTML disposen de diversos *events* que poden ser detectats per la vostra aplicació i executar diferents funcions.

Gràcies a aquest sistema és possible crear aplicacions interactives que responguin a les accions de l'usuari, o l'estat de l'aplicació. Per exemple, és possible detectar quan s'ha fet clic sobre un botó o un enllaç, quan ha finalitzat la descàrrega d'una imatge, o quan s'ha mogut el cursor del ratolí sobre una àrea concreta de la pàgina.

'Events' o esdeveniments?

Encara que la traducció al català seria 'esdeveniment', ens hi referim amb el seu nom en anglès perquè no hi hagi confusions.

Altres tipus d'*events* menys habituals, però que és interessant conèixer, són els relacionats amb les Web API, ja que en molts casos són imprescindibles per poder implementar correctament les aplicacions que les utilitzen. Per exemple, la Web API encarregada de gestionar les peticions AJAX (peticions asíncrones enviades a un servidor) depèn dels esdeveniments per determinar quin codi executar una vegada es rep la resposta del servidor.

1.1 Què és un event?

Per entendre en què consisteix el sistema d'*events* caldria fer una petita introducció als **patrons de disseny** i, més específicament, al patró **observador** (en anglès, *observer*).

Els patrons de disseny són solucions reutilitzables a problemes que ocorren sovint. No es tracta d'algorismes ni implementacions concretes, més aviat és una explicació clara de com resoldre un problema concret. Per exemple, en el llibre original sobre patrons de disseny podem trobar una descripció del problema, una solució en forma de diagrama UML i una implementació en C++ o Smalltalk.

A les referències bibliogràfiques, trobareu uns llibres que tracten a fons els patrons de disseny.

No es tracta, com en un receptari, d'agafar el codi que posen i enganxar-lo al nostre programa, sinó d'entendre quin és el problema, com aplicar una solució provada que funciona i quins són els pros i contres d'aplicar-la-hi.

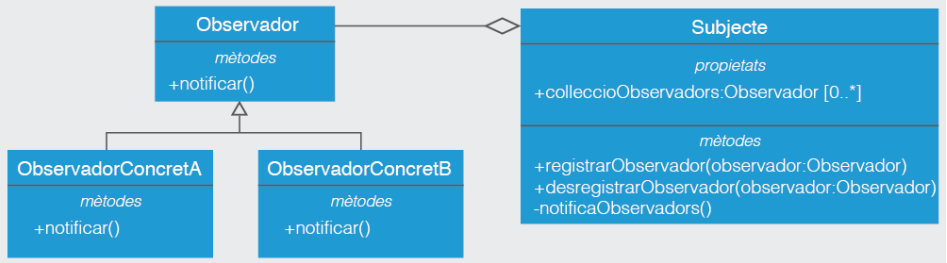
Un altre avantatge que té conèixer els patrons de disseny és que és molt útil per comunicar-se; per exemple: en lloc de descriure el problema i la solució que hi has aplicat, que generalment seria una explicació molt llarga, pots dir simplement: "He aplicat el patró de disseny X"; i si l'altra persona també els coneix, tindrà clar a què et refereixes sense haver d'afegir més detalls.

Patró Publicador-Subscriptor

El patró *Publicador-Subscriptor* és una variant del patró observador i en molts casos es poden fer servir de forma intercanviable.

El cas que ens ocupa, el sistema d'*events*, és una implementació del patró observador. En la figura 1.1, podeu veure amb més detall una versió simplificada de com funciona aquest patró.

FIGURA 1.1. Diagrama del patró observador



Aquest patró s'utilitza quan necessitem un sistema capaç d'indicar quan s'ha produït un esdeveniment i de comunicar-lo només als interessats, que s'han de poder afegir i eliminar dinàmicament.

Tot i que el diagrama no indica que es passi cap paràmetre a l'operació `notificar()`, normalment es passarà la informació del subjecte als observadors fent servir aquest paràmetre.

Exemple d'aplicació del patró observador

Si volem seguir l'IOC (@ioc) a Twitter, el que faríem és registrar-nos com a observadors al compte de l'IOC, que seria el subjecte. Llavors, quan l'@ioc publiqués un nou tuit (*tweet*), recorreria la seva llista d'observadors registrats i faria arribar aquesta informació a través de l'operació `notificar()` de cadascun dels observadors, passant el tuit com a paràmetre.

Dins d'aquesta operació, cada observador farà una cosa o altra amb aquesta piulada segons la seva implementació. Per exemple, en el cas dels usuaris, rebrien una notificació indicant que l'@ioc ha publicat una piulada; un administrador podria afegir-lo a una llista històrica, o en el cas d'un programari especialitzat d'automatització, podria retuitejar-lo (tornar-lo a publicar).

Si més endavant volem deixar de rebre notificacions, el que faríem és desregistrar-nos com a observador del subjecte, de manera que ja no rebríem més notificacions.

Aquesta seria una implementació d'aquest patró en JavaScript:

```
let subjecte = {
  observadorsRegistrats: [],

  registrarObservador: function(observador) {
    this.observadorsRegistrats.push(observador);
  },

  desregistrarObservador: function(observador) {
    let index = this.observadorsRegistrats.indexOf(observador);
    this.observadorsRegistrats.splice(index, 1);
  },

  notificaObservadors: function(missatge) {
    console.log("Iniciem la notificació del missatge als observadors");
    for (let i = 0; i < this.observadorsRegistrats.length; i++) {
      this.observadorsRegistrats[i].notificar(missatge);
    }
    console.log("Fi de les notificacions");
  },

  saludar: function() {
    this.notificaObservadors('Hola mon');
  }
}

let observador1 = {
```

```
  notificar: function(missatge) {
    console.log("Soc l'observador 1 i he rebut el missatge: " + missatge);
  }
}

let observador2 = {
  notificar: function(missatge) {
    console.log("Soc l'observador 2 i també mostro el missatge : " + missatge);
  }
}

// Enregistrem els dos observadors del subjecte
subjecte.registrarObservador(observador1);
subjecte.registrarObservador(observador2);

// Comprovem que funciona enviant un missatge
subjecte.saludar();

// Desenregistrem a un observador i tornem a saludar
subjecte.desregistrarObservador(observador1);
subjecte.saludar();

// Desenregistrem a un observador i tornem a saludar; com que ja no hi ha més
    observadors, no farà res
subjecte.desregistrarObservador(observador2);
subjecte.saludar();
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/vNrpXx?editors=1011. Tingueu en compte que si voleu provar l'exemple amb Internet Explorer, haureu de gravar la pàgina web al disc per després obrir-lo amb aquest navegador.

Una cosa semblant és el que fa el sistema d'*events*, però en aquest cas els Subjectes són els elements de la pàgina, per exemple els botons, les imatges, els paràgrafs... I els observadors, en lloc de ser objectes complets, només són funcions que compleixen el mateix paper que l'operació notificar; en lloc de cridar aquesta operació, el que fa el subjecte és invocar aquestes funcions.

Funcions: objectes de primera classe

En JavaScript les funcions són objectes de primera classe i, com a tals, es poden tractar igual que altres objectes i guardar-los en variables i *arrays*.

En JavaScript i en altres llenguatges, aquestes funcions que passen a altres objectes es diuen ***callbacks***. Sempre que vegeu aquest terme heu de tenir en compte que el que s'espera és una funció que serà executada en un altre punt del programa.

En el cas dels navegadors, un *event* és el que es produeix quan el navegador detecta determinats esdeveniments; per exemple, quan es clica a un enllaç, s'activa un botó o es carrega una pàgina.

Llista d'events

Podeu trobar una llista exhaustiva dels *events* disponibles a www.goo.gl/sSVjnG.

Actualment hi ha una gran quantitat d'*events* estàndards associats als elements HTML; els hem agrupat de la manera següent:

- **DOM**: *events* activats per accions sobre el DOM (Document Object Model), per exemple, si s'ha enviat un formulari, si s'ha canviat de camp o si s'ha produït cap canvi en un text que havíem introduït. Podem classificar-los en:
  - **MouseEvent**: *events* que es produeixen en detectar-se accions relacionades amb el ratolí.

  - **KeyboardEvent**: *events* produïts en detectar-se accions de teclat.
  - **UIEvent**: *events* relacionats amb els canvis a la interfície, com és el fet de carregar una pàgina o canviar la mida de la finestra.

- **HTML5**: *events* específics d'HTML5 que no són compatibles amb versions anteriors; ens permeten treballar amb *events* addicionals, que fan que puguem detectar quan es comença a arrossegar un element.
- **HTML5 media**: aquests *events* els produeixen els elements *media*, com l'àudio i el vídeo, i permeten conèixer el seu estat (si està a l'espera, si està cercant, si ha finalitzat...).
- **Web API**: a banda dels *events* proporcionats que formen part de l'especificació d'HTML5, s'han creat moltes API que són admeses pels navegadors i afegeixen funcionalitats molt importants. Aquestes API també fan servir el sistema d'*events* per comunicar-nos quan es produeixen canvis a l'emmagatzematge local, si s'ha rebut un missatge d'un *web worker* indicant que ha finalitzat la tasca, o si s'han detectat canvis a la càrrega de la bateria.

Web worker

Un *web worker* és un tipus de procés que permet realitzar tasques en paral·lel a l'execució principal.

Les Web API

Encara que existeixen moltes Web API, s'ha de tenir en compte que no totes són estàndard; per exemple vosaltres mateixos en podríeu crear una. Les estàndard són les que es troben a les especificacions web oficials. A l'enllaç següent, podeu veure quins són aquests *events* estàndard i, en cada cas, a quina especificació de Web API pertanyen: www.goo.gl/wXGpNN.

Encara que aquestes Web API es troben a la llista d'especificacions del W3C, es desenvolupen de manera paral·lela a l'estàndard HTML5.

Alguns exemples de Web API són:

- **Web Workers**: tot i que els navegadors no permeten la programació multifil amb suport per múltiples nuclis, amb aquesta API es poden programar tasques que s'executen de manera concurrent.
- **Web Storage**: ens permet emmagatzemar dades en el nostre navegador de manera que no s'han de descarregar cada vegada; és semblant al que es fa amb les galetes però amb un límit molt més gran (el volum depèn del navegador).
- **XMLHttpRequest**: aquesta és l'API que ens permet fer servir crides AJAX des del navegador.
- **WebSocket**: fent servir aquesta API és possible mantenir una connexió bidireccional amb servidors, cosa que permet crear aplicacions tipus xat o jocs "multijugador" al navegador.

AJAX

AJAX és un conjunt de tecnologies que permeten actualitzar els continguts d'una pàgina web sense haver de carregar una altra pàgina.

Podeu veure amb més detall el funcionament d'AJAX a la secció "API XMLHttpRequest" d'aquest mateix apartat.

1.2 Gestió d'events

Un dels problemes de JavaScript és que es poden fer les mateixes coses de moltes maneres i, com passa habitualment en aquests casos, algunes són millors que d'altres. En aquesta secció veurem les diverses maneres de treballar amb *events* amb JavaScript.

Començarem veient com podem registrar-nos com a observadors d'un *event*, en aquest cas el paper de l'operació `notificar()` el realitza la funció *alert*; el subjecte és el botó que ens notificarà el moment que es dispari l'*event* `click`:

```
<button onclick="alert('clic a l'element');">
  Fes clic aquí!
</button>
```

Fixeu-vos que s'ha afegit a l'element `button` un atribut, `onclick` amb el valor `alert('clic a l'element')`. Aquest valor és un fragment de codi JavaScript que serà executat en detectar-se l'*event* `click` sobre el botó. Aquesta és la manera més simple d'afegir la detecció d'*events* a un element, tot i que no és la més adient.

> Tot i que és possible afegir codi per reaccionar a *events* directament incrustats al codi HTML, **es considera una mala pràctica**. Sempre hem d'intentar separar el codi CSS, HTML i JavaScript en blocs diferents i no fer-los *inline*. A més, és preferible carregar-los com a fitxers externs.

Veieu a continuació un altre exemple més complexe on s'ha substituït el codi JavaScript per una crida a una funció pròpia:

```
<script>
    let i = 0;
    function mostraMissatge() {
      ++i;
      alert('Has fet clic a l'element ' + i + ' vegades');
    }
  </script>
  <button onclick="mostraMissatge();">
    Fes clic aquí!
  </button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/YyeJLP?editors=1010.

Si analitzem aquest exemple pas a pas, el funcionament seria el següent:

1. Quan el navegador processa el codi HTML i troba l'atribut `onclick` a l'element `button`.
2. El navegador enregistra aquesta funció (`mostraMissatge()`) per ser executada quan es dispari l'*event* `click`. És a dir, la funció serà invocada quan es detecti un clic sobre aquest element.

3. Quan el navegador detecta que s'ha disparat un *event* de tipus `click` sobre el botó:

   (a) Recorre la llista d'observadors registrats per l'*event* `click` d'aquest subjecte.

   (b) Els "notifica" invocant a la funció corresponent passada com argument (el *callback*), en aquest cas `mostrarMissatge()`, amb la informació de l'*event* com a paràmetre.

Aquesta és una versió lleugerament modificada per comprovar que, efectivament, la funció `mostrarMissage()` rep la informació que li passa el navegador:

```
<script>
    function mostraMissatge(e) {
      console.log(e);
      alert('Detectat un event de tipus: ' + e.type);
    }
  </script>
  <button onclick="mostraMissatge(event);">
    Fes clic aquí!
  </button>
```

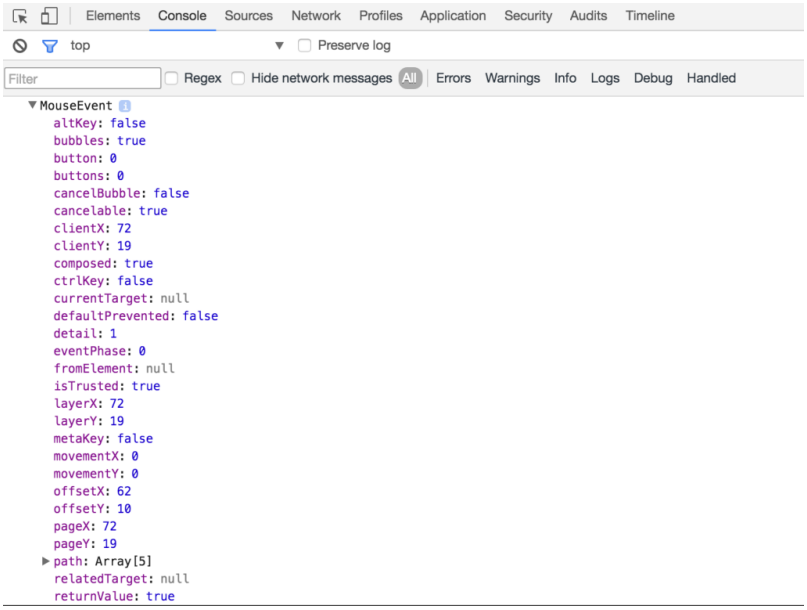
Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/qOxJZL?editors=1011.

Eines per a desenvolupadors

La informació que li passa el navegador es mostra a través de la consola. S'hi accedeix amb les eines per a desenvolupadors que faciliten els navegadors. Cada navegador té les seves pròpies eines, però algunes, com la consola -que és fonamental-, les inclouen tots els navegadors.

En aquest cas hem fet servir `console.log(e)` per mostrar, per la consola del navegador (a la qual es pot accedir des de les eines per a desenvolupadors), el contingut total de l'*event* que hem rebut; com es pot veure a la figura 1.2.

FIGURA 1.2. Exemple d'informació de l'event 'click'



La funció `document.getElementById()` no és pròpia de JavaScript, sinó que és afegida per la interfície DOM que implementen els navegadors.

1.2.1 Afegir funcions als events

Una manera correcta d'afegir aquestes funcions sense haver de contaminar el codi HTML és la següent:

1. El primer pas és obtenir una referència als elements. En aquest cas, per simplificar, farem servir la propietat `id`. Aquesta propietat no pot repetir-se en cap altre element de la pàgina.
2. Podem tractar aquests elements com a objectes i, per tant, podem definir-hi un mètode i assignar-los una funció (anònima en aquest cas). Per exemple, creem un mètode `onclick` tant al primer com al segon element.
3. Quan es dispari un *event* de tipus `click` sobre els subjectes, aquests ho notificaran als seus observadors i s'invocarà la funció associada amb la informació de l'*event*.

```
<script>
  // Guardem els elements en variables
  let primerElement = document.getElementById('primer'),
    segonElement = document.getElementById('segon');

  // Assignem a cada element una funció que serà cridada quan s'hi faci clic
  primerElement.onclick = function (e) {
    alert('Clic al primer element de tipus: ' + e.type);
  }

  segonElement.onclick = function(e) {
    alert('Clic al segon element de tipus: ' + e.type);
  }

</script>
<button id="primer">
  Fes clic al primer element!
</button>
<button>
  Aquí no fa res, no té cap id per referir-nos
</button>
<button id="segon">
  Fes clic al segon element!
</button>
<button id="tercer">
  Aquí sí que tenim id, però no hi hem afegit cap funció
</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ojEQbb?editors=1010.

Fixeu-vos que hem afegit més elements dels necessaris a l'HTML. Això ho hem fet per comprovar que, efectivament, l'element que no té associada cap "id" i l'element que sí que la té (però que no l'hem lligat a cap funció) no responen als clics que fem sobre ells.

Vegeu el mateix exemple, però usant la notació fletxa de les funcions. Aquesta notació és molt útil sobretot quan el cos de les funcions és curt. Al material s'utilitzaran les dues notacions de manera indiferent:

```
<script>
  // Guardem els elements en variables
  let primerElement = document.getElementById('primer'),
    segonElement = document.getElementById('segon');

  // Assignem a cada element una funció que serà cridada quan s'hi faci clic
  primerElement.onclick = e => alert('Clic al primer element de tipus: ' + e.
      type);

  segonElement.onclick = e => alert('Clic al segon element de tipus: ' + e.type
      );

</script>
<button id="primer">
  Fes clic al primer element!
</button>
<button>
  Aquí no fa res, no té cap id per referir-nos
</button>
<button id="segon">
  Fes clic al segon element!
</button>
<button id="tercer">
  Aquí sí que tenim id, però no hi hem afegit cap funció
</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/jOBjByG?editors=1010.

La manera d'assignar events vista als exemples anteriors, encara que sigui correcta, presenta un inconvenient: només hi pot haver un observador per a cada tipus d'*event*. Si intentem afegir-n'hi un altre, l'anterior queda eliminat; com podeu provar amb aquest exemple:

```
<script>
  // Guardem l'element en una variable
  let primerElement = document.getElementById('primer');

  // Assignem a l'element una funció que serà cridada quan s'hi faci clic
  primerElement.onclick = () => alert('Primera funció');

  // Assignem una altra funció l'element que també serà cridada quan s'hi faci
      clic
  primerElement.onclick = e => alert('Segona funció');
</script>
<button id="primer">
  Fes clic aquí!
</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/QjQJpV?editors=1010.

Com podeu veure, no funciona correctament: la segona funció sobreescriu la primera, així que només es notifica la segona.

> Si en lloc d'usar una funció anònima per a l'assignació, voleu assignar-li una funció creada en alguna altra part del codi, cal posar el nom d'aquesta funció sense els parèntesis, ja que si no, en comptes d'assignar la funció a l'esdeveniment, el que faria seria executar la funció i assignar el resultat d'aquesta.

És a dir:

```
<script>
  // Guardem l'element en una variable
  let primerElement = document.getElementById('primer');

  // Assignem a l'element una funció que serà cridada quan s'hi faci clic
  primerElement.onclick = notificacio;

  function notificacio() {
    alert('Primera funció');
  }

</script>
<button id="primer">
  Fes clic aquí!
</button>
```

Afegir múltiples observadors per a un mateix subjecte

És molt possible que necessitem afegir dos o més observadors a un subjecte pel mateix *event* o que ho hàgim de fer més endavant, així que la **forma més recomanable** de fer-ho és **amb el mètode** `addEventListener()`, com us mostrem en l'exemple següent:

addEventListener() a IE 8 i versions anteriors

La funció `addEventListener()` no s'ha afegit al navegador de Microsoft fins a la versió 9. Per a versions anteriors cal fer servir `attachEvent()`, que té la mateixa funcionalitat, però Internet Explorer 11 ja no l'admet.

```
<script>
  // Guardem l'element en una variable
  let primerElement = document.getElementById('primer');

  // Creem les funcions que volem que siguin cridades
  function mostrarMissatge1() {
    alert("primera funció");
  }

  function mostrarMissatge2() {
    alert("segona funció");
  }

  // Subscrivim l'element perquè respongui amb les dues funcions
  primerElement.addEventListener('click', mostrarMissatge1);
  primerElement.addEventListener('click', mostrarMissatge2);
</script>
<button id="primer">
  Fes clic aquí!
</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/qOxQXZ?editors=1010.

La signatura del mètode `addEventListener()` és la següent:

```
1 target.addEventListener(type, listener[, useCapture]);
```

- `target` és l'element al qual s'ha de disparar l'*event* perquè aquest sigui detectat pel **subjecte**. És a dir, si volem detectar quan s'ha disparat l'*event* `click` sobre un botó, el `target` seria el botó. Fixeu-vos que amb aquest sistema es pot observar qualsevol element del DOM i molts dels elements de les Web API, no només dels elements HTML.
- `type` és una cadena amb el tipus d'*event* que volem escoltar, per exemple: `click`, `dblclick`, `load`.
- `listener` aquesta és la funció *callback* que serà cridada quan es dispari l'*event* observat. Fixeu-vos que aquesta funció és la que és **notificada** quan es produeix l'*event* al subjecte. Correspon a la funció `notificar()` en la figura 1.2.
- `useCapture` és un paràmetre opcional que no farem servir en aquests materials, per tant, el considerem sempre fals. En cas que fos cert, canviaria la forma amb què es despatxen els *events*.

En podeu trobar més informació al següent enllaç: www.goo.gl/a8LqGS.

Ara ja hem vist com podeu afegir la detecció d'*events* a un element i com podeu invocar la funció que us interessi. Segurament us deveu haver adonat que descriure els processos és una mica enrevessat. Per aquest motiu, a partir d'ara ens referirem als elements i a les diferents accions de la següent manera:

- ***Event***. Pot fer referencia tant a un tipus com a a un objecte:
  - Un tipus, com pot ser `click`, `dblclick` o `load`.
  - Un objecte que conté tota la informació de l'esdeveniment que s'ha disparat.
- **Disparar un *event*** (en anglès *trigger* o *fire*). Significa que s'ha produït un esdeveniment. Per exemple, quan el navegador finalitza la càrrega del codi HTML es produeix l'esdeveniment `load`. En aquest cas diríem que s'ha disparat l'*event* `load`, i aquesta informació seria notificada als observadors registrats.
- **Despatxar un *event*** (en anglès *dispatch*). Continuem amb el mateix exemple: quan el navegador finalitza la càrrega de la pàgina, despatxa l'*event* `load`. És a dir, quan quelcom és responsable de comunicar que s'ha produït algun esdeveniment, el que fa és despatxar l'*event*. Quan es despatxa un *event* es passa un objecte com a paràmetre que conté tota la informació corresponent. Per exemple, en el cas dels *events* del ratolí, els botons clicats o la posició del cursor es passen a la pantalla com a coordenades cartesianes. Quan es despatxa un *event* el que es fa és notificar tots els observadors registrats per aquest tipus d'*event*.

Per eliminar la detecció d'*events* d'un element cal invocar el mètode `removeEventListener()` i passar com a arguments el tipus d'*event* i la funció que es vol eliminar.

1.2.2 Events del ratolí ('MouseEvents')

Aquests són els *events* que es disparen quan canvia l'estat del ratolí (vegeu la taula 1.1), per exemple, quan es prem un botó del dispositiu o quan es deixa anar.

Taula 1.1. Events relacionats amb els botons del ratolí

Tipus	Operació	Descripció
click	onclick	Es dispara en fer clic amb qualsevol botó del ratolí
dblclick	ondblclick	Es dispara en fer doble clic amb qualsevol botó del ratolí
mousedown	onmousedown	Es dispara en prémer qualsevol botó del ratolí
mouseup	onmouseup	Es dispara en deixar anar qualsevol botó del ratolí

Per comprovar com funciona podríeu escriure un codi així:

```
<button onclick="console.log('click');">test clic</button>
<button ondblclick="console.log('dblclick');">test doble clic</button>
<button onmousedown="console.log('mousedown');">test prémer botó del ratolí</
    button>
<button onmouseup="console.log('mouseup');">test deixar anar botó del ratolí</
    button>
```

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

Fixeu-vos que cada botó mostra el missatge a la consola sota unes condicions diferents.

Un altre punt interessant és saber en quin ordre es disparen aquests *events*; què passa si hem registrat un observador per a cada *event* d'un sol botó?

En aquest cas, recordeu que haurem de fer servir forçosament `addEventListerner()`, així que els afegirem mitjançant JavaScript.

```
<script>
  let button = document.getElementById('test');

  button.addEventListener('click', notificaObservadorClick);
  button.addEventListener('dblclick', notificaObservadorDblClick);
  button.addEventListener('mousedown', notificaObservadorMouseDown);
  button.addEventListener('mouseup', notificaObservadorMouseUp);

  function notificaObservadorClick() {
    console.log("S'ha disparat l'event Click");
  }

  function notificaObservadorDblClick() {
    console.log("S'ha disparat l'event DblClick");
```

Per detectar el botó del ratolí que s'ha polsat es pot fer servir la propietat `button` de l'objecte `event`, de manera que:

- Botó esquerre: `event.button = 0`
- Botó central: `event.button = 1`
- Botó dret: `event.button = 2`

Aquesta propietat només funciona amb els events `mouseup` i `mousedown`.

```
15    }
16
17    function notificaObservadorMouseDown() {
18      console.log("S'ha disparat l'event Mouse Down");
19    }
20
21    function notificaObservadorMouseUp() {
22      console.log("S'ha disparat l'event Mouse Up");
23    }
24  </script>
25
26  <button id="test">test de múltiples events</button>
```

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

Fixeu-vos en quin ordre es disparen els *events*:

1. El primer de ser disparat és `mousedown`.
2. A continuació es dispara `mouseup`, quan es deixa anar el botó.
3. I finalment es dispara `click`. És a dir, fins que no es deixi anar el botó no es confirmarà que s'ha produït un clic sobre l'element.

Si el que fem és un doble clic, fixeu-vos com canvia la seqüència:

1. `mousedown`
2. `mouseup`
3. `click`
4. `mousedown`
5. `mouseup`
6. `click`
7. `dblclick`

Això s'ha de tenir molt en compte, perquè si tenim un element al qual hem lligat la detecció d'*events* com `click` o `mousedown` al mateix temps que `dblclick`, també es dispararan quan es faci un doble clic, i segurament no és la nostra intenció.

Objectes com a paràmetres

En JavaScript és molt freqüent fer servir objectes que només tenen propietats per passar-les a mètodes o funcions. En altres llenguatges, això s'anomenaria *diccionari de dades*, *array associatiu* o simplement *array*. Un exemple d'aquests tipus d'objectes seria `{x: 10, y:20}`.

Vegem ara una variant del codi anterior. Aquesta vegada, a les funcions de notificació els arriba un paràmetre; en lloc de fer servir una funció per a cada *event*, farem servir la mateixa funció `notificarObservador()` per a totes.

El subjecte sempre ens enviarà la informació de l'*event* com un objecte. És important entendre que aquest paràmetre **l'envia el subjecte que observem**, i que no importa el nom que li posem nosaltres, perquè aquest nom només ens serveix de referència a la nostra funció. **Si no el fem servir, no cal declarar-lo com a paràmetre**, simplement serà ignorat.

```
<script>
  let button = document.getElementById('test');

  button.addEventListener('click', notificaObservador);
  button.addEventListener('dblclick', notificaObservador);
  button.addEventListener('mouseup', notificaObservador);
  button.addEventListener('mousedown', notificaObservador);

  function notificaObservador(e) {
    console.log("S'ha disparat un event de tipus: ", e.type, "amb aquesta
        informació: ", e);
  }
</script>
<html>
<button id="test">test de múltiples events</button>

</html>
```

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

En aquest exemple podeu veure que obtenim el tipus d'*event* de la propietat `type` de l'*event* que ens arriba per paràmetre, i a continuació es mostra tota la informació que conté en aquest objecte (com poden ser les coordenades, en quin element s'ha produït l'*event*, si s'està prement cap tecla modificadora -majúscules o control, per exemple-...).

Variacions segons el navegador

Segons en quin navegador s'executi el codi **alguns comportaments poden ser diferents**. Per exemple, en el cas del navegador Chrome, si es fa clic amb el botó dret del ratolí, només es dispara l'*event* `mousedown`, en canvi, en Firefox es dispara també l'*event* `mouseup`.

Especificació HTML5

Podeu trobar el document actualitzat amb l'especificació d'HTML5 en què es basa la implementació dels navegadors web a: bit.ly/3tMEmTr

Aquesta discrepància es produeix perquè en fer clic amb el botó dret tots dos disparen l'*event* `contextmenu` (vegeu la taula 1.2), com s'indica a l'especificació d'HTML5, però disparar o no l'*event* `mouseup` o `keyup` -segons el cas- depèn del navegador i de la plataforma que faci servir l'usuari. Inclús un mateix navegador, en diferents sistemes operatius, pot tenir comportaments diferents perquè l'especificació només obliga que es dispari l'*event* `contextmenu`.

Taula 1.2. Events relacionats amb el menú contextual

Tipus	Operació	Descripció
contextmenu	oncontextmenu	Es dispara quan es mostra el menú contextual del navegador

Si proveu aquest exemple en Chrome, veureu que efectivament la resta d'*events* es disparen, ja que la seqüència s'interromp quan es llença el menú contextual. En canvi, si ho proveu amb Firefox, l'*event* `mouseup` es dispara a continuació de l'*event* `contextmenu`:

```
<script>
  let button = document.getElementById('test');
```

```
3
4     button.addEventListener('click', notificaObservador);
5
6     button.addEventListener('contextmenu', notificaObservador);
7
8     button.addEventListener('mouseup', notificaObservador);
9
10    button.addEventListener('mousedown', notificaObservador);
11
12    function notificaObservador(e) {
13      console.log("S'ha disparat un event de tipus: ", e.type, "amb aquesta
            informació: ", e);
14    }
15  </script>
16  <button id="test">test de múltiples events</button>
```

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

Aquí, la seqüència per a Chrome:

1. `mousedown`
2. `contexmenu`

Mentre que per a Firefox serà aquesta:

1. `mousedown`
2. `contexmenu`
3. `mouseup`

jQuery

jQuery és una biblioteca molt popular que inclou moltes funcionalitats que s'usen habitualment com a crides AJAX, manipulació del DOM (Document Object Model), enregistrament a *events* i animacions d'elements. Les versions més recents han deixat de banda el suport per a navegadors antics.

S'ha de tenir en compte que l'especificació d'HTML5 no és exhaustiva. Per això hi ha casos com aquests en què el comportament serà diferent segons el navegador que s'utilitzi. En aquests casos hem de decidir entre:

- Evitar fer servir elements que puguin presentar conflictes entre navegadors i plataformes.
- Fer una implementació que cobreixi tots els casos per als navegadors i les plataformes en què vulguem que el nostre codi funcioni correctament.
- Fer servir una llibreria que s'encarregui de normalitzar aquests comportaments, per exemple *jQuery*.

Moviments del ratolí

A banda dels *events* que es disparen en prémer els botons del ratolí, també podem detectar els que es produeixen en moure'l. Aquesta és la llista de tipus d'*events* observables (vegeu la taula 1.3).

Taula 1.3. Altres events relacionats amb els botons del ratolí

Tipus	Operació	Descripció
mouseenter	onmouseenter	Es dispara quan el cursor entra dins de l'àrea ocupada pel subjecte
mouseleave	onmouseleave	Es dispara quan el cursor abandona l'àrea ocupada pel subjecte
mousemove	onmousemove	Es dispara cada vegada que el cursor es mou per sobre del subjecte
mouseout	onmouseout	Es dispara quan el cursor abandona l'àrea ocupada pel subjecte o per algun dels seus fills
mouseover	onmouseover	Es dispara quan el cursor entra dins de l'àrea ocupada pel subjecte o per algun dels seus fills

Com podeu veure, la diferència principal entre `onmouseenter` i `onmouseover` és que el primer no diferencia els fills del subjecte i el segon sí que ho fa, i passa el mateix amb `onmouseout` i `onmouseleave`.

Fixeu-vos en aquest exemple; el quadre més petit és fill del quadre més gran, i hi podeu veure:

```
<style>
  div {
    width: 150px;
    height: 150px;
    background-color: red;
    margin: 50px auto;
  }

  div>div {
    position:relative;
    top:-25px;
    width: 50px;
    height: 50px;
    background-color: blue;
  }
</style>
<script>
  let element = document.getElementById('test');
  element.addEventListener('mouseenter', notificaObservador);
  element.addEventListener('mouseleave', notificaObservador);
  element.addEventListener('mousemove', notificaObservador);
  element.addEventListener('mouseout', notificaObservador);
  element.addEventListener('mouseover', notificaObservador);
  function notificaObservador(e) {
    console.log("S'ha disparat un event de tipus: ", e.type);
  }
</script>
<div id="test">
  <div></div>
</div>
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ZbRRbQ?editors=1111

En entrar al quadre gran o al petit des de fora es disparen:

1. `mouseover`
2. `mouseenter`

3. `mousemove`, múltiples vegades, mentre el cursor estigui en moviment sobre el quadre.
4. `mouseout` en sortir del quadre gran
5. `mouseleave` en sortir del quadre gran

La diferència la trobem quan tenim el cursor a sobre d'un dels dos quadres i passem a l'altre; per exemple, amb el cursor sobre el quadre vermell passem al quadre blau. En aquest cas, veurem a la consola la següent seqüència (ignorant els `mousemove` és clar):

1. `mouseout`
2. `mouseover`

És a dir, es detecta la transició entre els dos elements, cosa que no passa amb `mouseenter` i `mouseleave`.

Events per a pantalla tàctil

Els **TouchEvent** són similars als **MouseEvent** però pertanyen a una altra especificació diferent (vegeu la taula 1.4). Són els *events* disparats quan s'interactua tocant una pantalla tàctil, com podria ser la d'un mòbil o una tauleta.

> **Combinar els EventTouch i MouseTouch**
>
> Com que aquest és un problema molt comú, generalment ja està resolt o es poden trobar llibreries que adaptin el funcionament que necessitem. Per exemple: els dispositius mòbils interpreten els tocs de pantalla com a clics automàticament, però si voleu afegir coses més específiques com el reconeixement de textos, haureu de fer servir una llibreria especialitzada.

Taula 1.4. Events relacionats amb l'especificació TouchEvent

Tipus	Operació	Descripció
touchstart	ontouchstart	Es dispara quan es toca la pantalla encara que ja s'estigui tocant en un altre punt
touchend	ontouchend	Es dispara quan es deixa de tocar la pantalla en un punt, encara que hi hagi altres punts pressionats
touchmove	ontouchmove	Es dispara quan un dels dits es mou sobre la pantalla
touchcancel	ontouchcancel	Es dispara quan es produeix una disrupció d'una manera específica, com podria ser que hi haguessin més punts tocats dels que admet el dispositiu

L'exemple següent canviarà de color quan es disparin els *events* de toc a la pantalla; el `touchstart` farà que es posi verd, el `touchmove` de color groc i `touchend` de color gris. L'*event* `touchcancel` normalment no es dispara.

```
<style>
  div {
    position: absolute;
    top: 0;
```

```
    bottom: 0;
    left: 0;
    right: 0;
    background-color: orange;
  }
</style>
<script>
  let element = document.getElementById('test');

  element.addEventListener('touchcancel', notificaObservador);
  element.addEventListener('touchstart', notificaObservador);
  element.addEventListener('touchend', notificaObservador);
  element.addEventListener('touchmove', notificaObservador);

  function notificaObservador(e) {

    switch (e.type) {
      case 'touchstart':
        element.style['background-color'] = 'green';
        break;

      case 'touchend':
        element.style['background-color'] = 'grey';
        break;

      case 'touchmove':
        element.style['background-color'] = 'yellow';
        break;

      case 'touchcancel':
        element.style['background-color'] = 'red';
        break;
    }

  }
</script>
<div id="test"></div>
```

Podeu provar-ne l'exemple en aquest URL: codepen.io/ioc-daw-m06/pen/avKKJy. Però en aquest cas ho haureu de fer des de un dispositiu amb pantalla tàctil, com un mòbil o una tauleta, ja que a l'ordinador aquests *events* no es disparen.

Podem fer servir una única funció per gestionar totes les notificacions i llavors, depenent del tipus d'*event* (`Event.type`), fer una cosa o altra amb un selector `switch-case`; com per exemple canviar el color de la pantalla.

1.2.3 Events del teclat ('keyboardEvents')

Els *events* de teclat són els que ens permeten reaccionar quan es dispara un *event* relacionat amb el teclat, com per exemple prémer una tecla o deixar-la anar.

Taula 1.5. Events relacionats amb el teclat

Tipus	Operació	Descripció
keydown	onkeydown	Es dispara quan es prem una tecla
keypress	onkeypress	Es dispara quan es prem una tecla, i la tecla produeix un valor corresponent a un caràcter

TAULA 1.5 (continuació)

Tipus	Operació	Descripció
keyup	onkeyup	Es dispara quan es deixa anar la tecla

D'entrada això pot semblar una mica estrany; per exemple, es pot observar un element de tipus `<div>` per l'*event* `keydown`, però en quines circumstàncies es rebria la notificació? Vegem-ho amb un primer exemple amb dos elements, un és un `<div>` i l'altre, un botó:

```
<script>
  let element1 = document.getElementById('test1');
  let element2 = document.getElementById('test2');

  element1.addEventListener('keypress', notificaObservador);
  element1.addEventListener('keydown', notificaObservador);
  element1.addEventListener('keyup', notificaObservador);

  element2.addEventListener('keypress', notificaObservador);
  element2.addEventListener('keydown', notificaObservador);
  element2.addEventListener('keyup', notificaObservador);

  function notificaObservador(e) {
    console.log("S'ha disparat un event de tipus: ", e.type);
  }
</script>
<div id="test1">Aquest element no pot rebre el focus, així que no dispara
    events de teclat</div>
<button id="test2">Quan tinc el focus puc disparar els events de teclat</button
    >
```

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

Com podeu veure no és possible fer que l'element `<div>` dispari els *events* de teclat; en canvi, veiem que quan el botó té el *focus*, aquest sí que els dispara i mostra els missatges a la consola. Però si el *focus* és en un altre element de la pàgina, deixa de disparar-los.

Focus

L'element que té el *focus* és el que està seleccionat en un moment determinat i només pot haver-hi un element amb *focus*. Es distingeix normalment perquè surt envoltat per una brillantor (botons, quadres de text...) o per una línia puntejada (enllaços). L'estil depèn del navegador.

Vegem un altre cas, aquesta vegada amb un element capaç de rebre el *focus* dins del `<div>`; per exemple, un altre botó:

```
<script>
  let element1 = document.getElementById('test1');
  let element2 = document.getElementById('test2');

  element1.addEventListener('keypress', notificaObservador1);
  element1.addEventListener('keydown', notificaObservador1);
  element1.addEventListener('keyup', notificaObservador1);

  element2.addEventListener('keypress', notificaObservador2);
  element2.addEventListener('keydown', notificaObservador2);
  element2.addEventListener('keyup', notificaObservador2);

  function notificaObservador1(e) {
    console.log("Des del <div> s'ha disparat un event de tipus: ", e.type);
  }

  function notificaObservador2(e) {
    console.log("Des del <button> s'ha disparat un event de tipus: ", e.type);
  }
</script>
```

```
<div id="test1">Aquest és el div observat que conté un botó <button>Aquest es
    un botó que no està essent observat</div>
<button id="test2">Aquest és el botó que està essent observat</button>
```

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

En aquest cas, encara que el botó que hi ha a dins del `<div>` no estigui essent observat, sí que es disparen els *events* corresponents al `<div>`: quan aquest botó té el *focus* i es prem alguna tecla (feu un clic a sobre perquè rebi el *focus* i premeu qualsevol tecla). De tot això podem deduir quines són les regles perquè es puguin detectar o no si s'ha premut alguna tecla.

Per poder disparar *events* de teclat, un element ha de complir alguna de les següens condicions:

- Ha de ser capaç de rebre el *focus*.
- Ha de contenir un element fill capaç de rebre el *focus*.

Si s'acompleixen alguna d'aquestes condicions i l'element, o algun dels seus descendents, és el *focus* llavors disparà els *events* de teclat.

Capturar events

Quan un element dispara un *event* i un altre és notificat, direm que l'*event* ha estat capturat. El fet de ser notificat és el mateix que capturar l'*event*.

L'element 'body'

Un cas interessant és el de l'element `<body>`: encara que no tingui el *focus*, si s'afegeix un observador per als *events* de teclat, són capturats igualment.

En l'exemple següent podeu veure que no cal incloure cap altre element per fer que es detectin els *events* de teclat:

```
<script>
  let element = document.getElementById('test');

  element.addEventListener('keypress', notificaObservador);
  element.addEventListener('keydown', notificaObservador);
  element.addEventListener('keyup', notificaObservador);

  function notificaObservador(e) {
    console.log("Des del <body> s'ha disparat un event de tipus: ", e.type);
  }
</script>
<body id="test">
</body>
```

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

Aprofitem aquest exemple per analitzar el comportament d'aquests *events*:

- Si premem la tecla *a* (o qualsevol altre caràcter imprimible) i la deixem premuda, es dispararan alternativament els *events* `keydown` i `keypress` fins que deixem de prémer-la.

- Si premem la tecla *control*, només es dispararà l'*event* `keydown`.
- Quan deixem de prémer qualsevol tecla, es dispararà l'*event* `keyup`.

Fixeu-vos que encara que `keydown` i `keypress` es repeteixin mentre la tecla estigui premuda, no passa el mateix amb `keyup`, que només es dispara una vegada per tecla.

Llavors, què passa si estem observant un element i al mateix temps afegim un altre observador per a algun dels seus fills? Es cridaran tots dos? Es cridarà només un d'ells? Vegem-ho amb un exemple ampliat:

```
<script>
  let element1 = document.getElementById('test1');
  let element2 = document.getElementById('test2');

  element1.addEventListener('keypress', notificaObservador1);
  element1.addEventListener('keydown', notificaObservador1);
  element1.addEventListener('keyup', notificaObservador1);

  element2.addEventListener('keypress', notificaObservador2);
  element2.addEventListener('keydown', notificaObservador2);
  element2.addEventListener('keyup', notificaObservador2);

  function notificaObservador1(e) {
    console.log("Des del <body> s'ha disparat un event de tipus: ", e.type);
  }

  function notificaObservador2(e) {
    console.log("Des del <textarea> s'ha disparat un event de tipus: ", e.type)
        ;
  }
</script>
<body id="test1">
  <textarea id="test2"></textarea>
</body>
```

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

Efectivament, es dispararà l'*event* a tots dos elements, tant al `<textarea>` com al `<body>`, i sempre en aquest ordre. Aquest efecte es produeix gràcies a l'*efervescència* (*event bubbling* en anglès): quan a un element es dispara un *event*, aquest es propaga (per exemple, com les bombolles d'un cava) cap als diferents observadors i cap als elements que els contenen. És a dir, quan es dispara l'*event* al `<textarea>` aquest és enviat també a l'element que el conté, l'element `<body>`.

Per solucionar aquest problema només cal invocar el mètode `Event.stopPropagation()` de l'*event* que arriba com a paràmetre.

El flux d'events

Podeu trobar més informació sobre el flux d'*events* a www.goo.gl/vFVGS3 (DOM2) i www.goo.gl/zJBRcf (Esborrany DOM3)

En aquest exemple veiem que, en interrompre la propagació de l'*event* al `textarea`, no arriba mai a ser notificat l'element `<body>`:

```
<script>
  let element1 = document.getElementById('test1');
  let element2 = document.getElementById('test2');

  element1.addEventListener('keypress', notificaObservador1);
  element1.addEventListener('keydown', notificaObservador1);
  element1.addEventListener('keyup', notificaObservador1);
```

```
8
9     element2.addEventListener('keypress', notificaObservador2);
10    element2.addEventListener('keydown', notificaObservador2);
11    element2.addEventListener('keyup', notificaObservador2);
12
13    function notificaObservador1(e) {
14      console.log("Des del <body> s'ha disparat un event de tipus: ", e.type);
15    }
16
17    function notificaObservador2(e) {
18      console.log("Des del <textarea> s'ha disparat un event de tipus: ", e.type)
            ;
19      e.stopPropagation();
20    }
21  </script>
22  <body id="test1">
23    <textarea id="test2"></textarea>
24  </body>
```

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

Detecció de pulsacions de tecles

A banda de detectar si s'ha pres o s'ha deixat anar una tecla, per saber quina és la tecla en qüestió hem de comprovar la propietat `EventKeyboard.key` a Google Chrome i Safari, o `EventKeyboard.keyIdentifier` a Internet Explorer i Firefox:

Event.keyCode

Abans es podia fer servir la propietat `keyCode` d'un *event* per determinar quina tecla s'havia premut, però no tots els navegadors retornaven el mateix codi; finalment, aquesta propietat s'ha declarat obsoleta.

```
1   <script>
2     let element = document.getElementById('test');
3
4     element.addEventListener('keypress', notificaObservador);
5     element.addEventListener('keydown', notificaObservador);
6     element.addEventListener('keyup', notificaObservador);
7
8     function notificaObservador(e) {
9       console.log("Disparat ", e.type, " per la tecla amb codi: ", e.key || e.
            keyIdentifier);
10    }
11  </script>
12  <body id="test">
13  </body>
```

Operador lògic ||

En JavaScript es pot fer servir l'operador lògic ||, de manera que si el primer valor és `null` o `undefined` ho prova amb el següent valor fins que en troba un que sigui vàlid, i llavors retorna o s'assigna aquest valor.

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

El codi retornat serà, per exemple, en prémer la tecla 'a' us mostrarà per la consola:

1. `Disparat keydown per la tecla amb codi:  a`
2. `Disparat keypress per la tecla amb codi:  a`
3. `Disparat keyup per la tecla amb codi:  a`

Formats de codificació

Un format de codificació és el que ens permet identificar un caràcter a partir d'un codi. Dos dels més utilitzats són el codi ASCII i Unicode.

La versió original de ASCII creada el 1963 admetia 128 caràcters (7 bits) però van crear-se variacions que són conegudes com a *ASCII Estès*, que admeten fins a 256 i inclouen altres caràcters com són les vocals accentuades, la *ç*...

La codificació Unicode va ser creada a finals del 1987 i suporta una quantitat de caràcters molt més gran, fins a 1.114.112 (0x10FFFF) i es representen afegint-hi el prefix `U+` i el codi corresponent en hexadecimal. Aquest codi és compatible amb altres codificacions com ASCII7 o ISO 8859-1.

En realitat tots dos retornan el mateix caràcter, però en el cas de Google Chrome el retorna amb codificació Unicode en lloc de mostrar-lo com a ASCII.

Si ho proveu amb la tecla `control`, veureu que tots dos mostren el mateix:

1. `Disparat keydown per la tecla amb codi: Control`
2. `Disparat keyup per la tecla amb codi: Control`

En el cas de les tecles no representables com a caràcter no es dispara l'*event* `keypress`.

Normalment els dos casos principals en què ens interessarà capturar els *events* de teclat serà dins dels elements que formen els formularis per determinar si s'està introduint un valor vàlid en una caixa de text, per exemple, i quan tractem amb experiències interactives, com poden ser jocs o *tours* virtuals que reaccionin amb el teclat.

En aquest segon cas, però, ens trobarem amb un problema: si us hi fixeu, quan es deixa polsada una tecla només es repeteix aquesta tecla i, per exemple, seria impossible fer un moviment en diagonal.

D'altra banda, els *events* són disparats en qualsevol moment i no en el punt del joc on els necessitem, és per això que interessa tenir un objecte al qual poder consultar l'estat del botó.

Per aquesta raó s'han de fer servir llibreries específiques o s'ha d'implementar un controlador d'entrada propi que guardi les tecles premudes i pugui ser consultat pel motor del joc o l'aplicació.

Evitar que es realitzi l'acció habitual

En alguns casos, amb `Event.preventDefault()` no n'hi ha prou, i pot funcionar fer que la nostra funció retorni el valor `false`, però depèn de l'element i el navegador amb els quals estiguem treballant.

En alguns casos ens pot interessar evitar que es dispari l'*event* que per defecte està associat a una acció; això és possible fent servir el mètode `Event.preventDefault()`. Per exemple, això ens permet aturar l'enviament d'un formulari en prémer el botó d'enviar i, en comptes d'això, realitzar una crida asíncrona.

Es veu més clarament en aquest exemple:

```
<script>
  let element = document.getElementById('test');

  element.addEventListener('keypress', notificaObservador);
  element.addEventListener('keydown', notificaObservador);
  element.addEventListener('keyup', notificaObservador);

  function notificaObservador(e) {
```

```
    console.log("Disparat ", e.type, " per la tecla amb codi: ", e.key || e.
        keyIdentifier);
    e.preventDefault();
  }
</script>
<body id="test">
  <textarea></textarea>
</body>
```

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

Com podeu veure no és possible escriure en el `textarea` i, com que el caràcter no es mostra, tampoc no es dispara l'*event* `keypress`.

1.2.4 Altres events del DOM

Hi ha dos subjectes en què ens interessa observar habitualment l'*event* `load` (vegeu la taula 1.6): són l'element `<body>` i l'element `<img>`. El primer cas és més habitual, ja que no és recomanable fer segons quines operacions abans que s'hagi acabat de carregar tot el fitxer HTML, per exemple si volem afegir o eliminar elements del document.

Taula 1.6. Events relacionats amb la càrrega i descàrrega

Tipus	Operació	Descripció
load	onload	Es dispara quan es finalitza la càrrega de la pàgina o d'algun recurs com per exemple una imatge
unload	onunload	Es dispara quan s'està descarregant el document o un recurs
error	onerror	Es dispara si es produeix un error

```
<body onload="alert('Document carregat');">
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/XmYxye?editors=1000 (s'executarà cada vegada que canvieu alguna cosa en el panell HTML, ja que el recarrega periòdicament).

L'altre cas en què ens interessa detectar quan es dispara aquest *event* és quan treballem amb imatges, ja que encara que el document s'hagi carregat completament i l'element `<body>` dispari l'*event* `load` no vol dir que les imatges s'hagin carregat, això només indica que el codi per carregar-les és present.

Això ens porta a trobar-nos amb casos en què volem realitzar alguna acció només quan les imatges s'hagin carregat completament, per exemple per reorganitzar-les segons la seva mida o per esperar a inicialitzar un joc en HTML5 perquè fins que no acaben de descarregar les imatges no s'ha de poder iniciar.

Hi ha un altre *event* relacionat directament amb el `load`: si quan es carrega una imatge es produeix un error, no es dispararà l'*event* `load` sinó l'*event* `error`, cosa que ens permet reaccionar en conseqüència.

L'exemple següent és força complex: el **gestor de descàrregues** del joc **IOC Puzzle Lite**, que està implementat amb HTML5. No us espanteu, que no cal que entengueu tot el codi ara per ara!, però és interessant que us feu una idea de com funciona, ja que la major part d'informació ja l'hem tractat, i en l'exemple es veu com utilitzar els *events* `load` i `error`:

```
let DownloadManager = function() {
  this.successCount = 0;
  this.errorCount = 0;
  this.downloadQueue = [];
  this.cache = {
    images: {}
  };

  this.queueDownload = function(imageData) {
    if (Array.isArray(imageData)) {
      for (let i = 0; i < imageData.length; i++) {
        this.downloadQueue.push(imageData[i]);
      }
    } else {
      this.downloadQueue.push(imageData);
    }
  };

  this.downloadAll = function(callback, args) {
    for (let i = 0; i < this.downloadQueue.length; i++) {
      let path = this.downloadQueue[i].path,
        id = this.downloadQueue[i].id,
        img = new Image();

      img.addEventListener("load", function() {
        this.successCount += 1;

        if (this.isDone()) {
          callback(args);
        }
      }.bind(this), false);

      img.addEventListener("error", function() {
        this.errorCount += 1;

        if (this.isDone()) {
          callback(args);
        }
      }.bind(this), false);

      img.src = path;
      this.cache.images[id] = img;
    }
  };

  this.isDone = function() {
    return (this.downloadQueue.length == this.successCount + this.errorCount);
  };

  this.getImage = function(id) {
    return this.cache.images[id];
  };
};
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/zvamep?editors=0010 (però com que només és una classe aïllada no fa res per si mateixa).

Es tracta d'una classe JavaScript que pot ser instanciada amb `new` d'aquesta manera: `let downloadManager = new DownloadManager();`

Una vegada instanciada, ens exposa les següents operacions:

- **queueDownload(*imageData*)**: afegeix una imatge (o *array* d'imatges) a la cua per ser descarregada. `imageData` conté la informació de la imatge (amb la `id` que li volem assignar per recuperar-la després) i la ruta, per exemple: `{id:"piece-0", path: "img/puzzle-piece-0.png"}`.
- **downloadAll(*callback*, *args*)**: inicia la descàrrega de tots els elements afegits a la cua i, quan acaba, crida la funció passada com a `callback` i `args` com a arguments. Això permet passar de la fase de descàrrega de recursos a la d'iniciar el joc.
- **isDone()**: aquest es fa servir internament; retorna "cert" si ha finalitzat la descàrrega de totes les imatges.
- **getImage(*id*)**: aquest mètode ens retorna la imatge amb l'identificador passat com a argument. Aquesta imatge la recupera del *cache* (memòria cau), on les hem anat descarregant. D'aquesta manera, encara que hi hagi múltiples imatges iguals no cal descarregar-les més d'una vegada.

Fer servir aquesta classe és molt fàcil:

1. Creem una nova instància: `let donwloadManager = new DownloadManager();`
2. Hi afegim els fitxers que vulguem descarregar: `donwloadManager.queueDownload([{id:'cotxe', patch:'imgs/imatge1.png'}, {id:'tren', patch:'imgs/imatge2.png'}]);`
3. Iniciem la descàrrega indicant la funció que s'ha d'invocar quan s'hagin descarregat totes les imatges: `donwloadManager.downloadAll(startGame)`. El segon paràmetre és opcional, pot ser qualsevol cosa que necessitem, per exemple, el nivell.
4. Per accedir a les imatges descarregades només hem d'invocar `downloadManager.getImage('tren')` i obtindrem la imatge, que podrem afegir a la pàgina o a un element de tipus `<canvas>`.

Ara veiem com funciona internament:

- Quan cridem `queueDownload()` la informació que li passem es guarda en un *array*.
- Quan cridem `downloadAll()` es recorre aquest *array*, crea un objecte de classe `Image` (element d'HTML5) per a cada element de l'*array* i hi afegeix dos observadors: un per a l'*event* `load` i un altre per a l'*event* `error`.
- Per cada imatge que es descarrega amb èxit o error s'augmenta en 1 el comptador intern d'èxits o errors, segons correspongui.

- Quan la suma de descàrregues amb èxit o error és igual al nombre d'elements a l'*array* d'imatges, sabem que la descàrrega de tots els elements ha finalitzat, així que passem a invocar la funció *callback*, que podria ser, per exemple, la càrrega del nivell o l'inici del joc.

> Quan implementem un gestor de descàrregues no s'ha d'oblidar gestionar els errors, ja que si només controléssim els casos d'èxit i una imatge donés error, el joc es bloquejaria perquè mai arribaria a finalitzar les descàrregues.

Això ens porta a un altre problema comú al treballar amb *events* i *callbacks*: el context en què s'executen les funcions no és el mateix en què es van originar; així que, si hem de fer referència a aquest context amb `this`, ens trobem amb problemes. Serà necessari trobar altres solucions, com guardar el context en una altra variable o utilitzar el mètode `bind()` comú a tots els objectes de JavaScript.

Context d'origen ('this') i context d'execució ('that')

Quan treballem amb *events* i *callbacks*, pot ocórrer que el context en què s'executen les funcions no sigui el mateix en què es van originar. En aquest cas, si hem de fer referència a aquest context amb `this`, ens trobarem amb problemes.

Per exemple, en el cas de la detecció d'*events* el context d'execució serà l'element en que s'ha disparat l'*event* i no pas la funció (o objecte) en el que es va afegir el detector. És a dir, si s'afegeix un detector per l'*event* `load` d'una imatge, el context de la funció que serà invocada quan la imatge acabi de carregar serà l'element imatge que l'ha disparat.

```
1  function ClasseA() {
2    // Aquí this fa referència a la instància de ClasseA
3    let img = new Image();
4      img.addEventListener("load", function () {
5        // En aquest exemple this fa referència a la imatge creada
6      });
7    }
8
9  function ClasseB() {
10   // Aquí this fa referència a la instància de ClasseB
11   let img = new Image();
12     img.addEventListener("load", function () {
13       // En aquest exemple this fa referència a la instància de ClasseB
14     }.bind(this));
15   }
```

Una altra solució habitual per a aquest problema és crear una variable en què guardem la referència a `this`. Aquesta variable acostuma a anomenar-se `that`:

```
1  function ClasseC() {
2    let that = this;
3    // Aquí this i that fan referència a la instància de ClasseB
4    let img = new Image();
5      img.addEventListener("load", function () {
6        // En aquest exemple this fa referència a la imatge creada
7        // Però that conserva la referència a la instància de ClasseB, així que
             podem fer-la servir si dins d'aquesta funció ens cal accedir a
             ClasseB.
8      });
9    }
```

En altres casos és precisament aquest el comportament que volem: que la referència a `this` sigui a l'objecte on s'està executant la funció, per exemple, per modificar algun atribut d'aquest objecte.

L'event 'unload'

L'*event* `unload` té la característica que és disparat quan es descarrega la pàgina, així que és una mica més difícil veure'l en funcionament.

Una particularitat és que **no es poden mostrar alertes** quan es vol reaccionar a aquest *event*, per tant, farem servir els missatges per la consola (que pot visualitzar-se activant les eines de desenvolupador). Això, però, també té un inconvenient: generalment, en recarregar una pàgina s'esborra el *log* de la consola, així que hem de marcar la casella *'preserve log'* (o l'opció de configuració equivalent segons el navegador) a les eines per a desenvolupadors. Vegem-ho:

```
<script>
  let element = document.getElementById('test');

  element.onunload = () => console.log("descarregant el document");
</script>
<body id="test"></body>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/LprXVM?editors=1011 (Recordeu que heu d'activar l'opció per preservar el *log* o s'esborrarà abans de carregar la pàgina següent).

Per disparar l'*event* només heu de recarregar la pàgina o modificar el codi HTML a l'entorn codepen.io (provoca que només es recarregui la visualització del resultat).

> Una vegada s'ha disparat l'*event* `unload`, **el procés és irreversible**; per exemple, no es pot mostrar un missatge d'avís per indicar que no s'han guardat els canvis. Per altra banda, és possible mostrar un avís abans de recarregar la pàgina observant l'*event* `beforeunload`.

A la taula 1.7 veureu una llista dels *events* relacionats amb els formularis.

Taula 1.7. Events relacionats amb els formularis

Tipus	Operació	Descripció
select	onselect	Es dispara quan se selecciona un text
change	onchange	Es dispara quan ha canviat el valor del subjecte i aquest ha perdut el focus
submit	onsubmit	Es dispara quan es pressiona un botó per enviar un formulari
reset	onreset	Es dispara quan es pressiona un botó de reinicialitzar un formulari

S'ha de tenir en compte que l'*event* `change` no es dispara en fer canvis a l'element, sinó **quan perd el *focus***. És a dir, mentre escrivim text en un `textarea`, aquest *event* no es dispara, només es dispararà quan canviem a un altre element; per exemple, quan cliquem en un botó o polsem la tecla de tabulació per canviar a l'element següent.

Un fet que haurem de tenir en compte és que quan es dispara l'*event* `submit` serà el moment que podrem fer canvis en la informació que s'enviarà en el formulari, ja sigui afegint més dades, validant les que hi ha o cancel·lant l'enviament si les dades no són correctes.

Els events 'focus', 'blur' i 'resize'

Tal com mostra la taula 1.8, els *events* `focus` i `blur` són complementaris.

Taula 1.8. Altres events destacables del DOM

Tipus	Operació	Descripció
focus	onfocus	Es dispara quan un element rep el *focus*
blur	onblur	Es dispara quan un element perd el *focus*
resize	onresize	Es dispara quan es canvia la mida de l'element

Quan un element rep el *focus*, just abans l'element que el tenia dispara l'*event* `blur` i a continuació l'element que l'ha rebut dispara l'*event* `focus`; la seqüència serà la següent:

1. Cliquem un element que no té el focus
2. L'element que té el *focus* dispara l'*event* `blur`
3. L'element que hem clicat rep el focus i dispara l'*event* `focus`

Com es rep el focus?

Les dues maneres més habituals per les quals un element rep el *focus* són: perquè s'ha clicat l'element o perquè s'ha premut la tecla de tabulació. Amb la tecla de tabulació es van recorrent tots els elements de la pàgina capaços de rebre el *focus*.

Es pot veure en aquest exemple amb dos quadres de text:

```
<script>
  let text1 = document.getElementById('test1');
  let text2 = document.getElementById('test2');

  text1.onfocus = e => console.log(e.type, "El primer quadre de text té el
      focus");

  text2.onfocus = e => console.log(e.type, "El segon quadre de text té el focus
      ");

  text1.onblur = e => console.log(e.type, "El primer quadre de text ha perdut
      el focus");

  text2.onblur = e => console.log(e.type, "El segon quadre de text ha perdut el
       focus");
</script>
<input type="text" id="test1" />
<input type="text" id="test2" />
```

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

Hi ha dos quadres de text, inicialment cap dels dos elements té el focus, així que no es dispara l'*event* focus de cap d'ells; i si anem alternant entre l'un i l'altre veurem a la consola:

1. `focus El primer quadre de text té el focus`
2. `blur El primer quadre de text ha perdut el focus`
3. `focus El segon quadre de text té el focus`
4. `blur El segon quadre de text ha perdut el focus`
5. etc.

Si cliquem a l'espai en blanc de la pàgina, es pot veure que es llença un últim `blur` quan el quadre de text perd el focus.

> **Els objectes window i document**
>
> Els elements `window` i `document` no són propis de JavaScript sinó que els proporciona el navegador. L'objecte `document` representa el document HTML i l'objecte `window`, la finestra on es mostra.
>
> A més a més, `document` és part de l'objecte `window` i és possible accedir-hi també com a `window.document` (encara que no té sentit fer-ho perquè podem accedir-hi directament com a `document`).

Només hi ha un element al qual podem registrar-nos com a observadors per l'*event* `resize`, i ser notificats d'alguna cosa, es tracta de l'element `window`, que representa la finestra del navegador on es mostra la pàgina.

Vegem-ho en aquest exemple:

```
<style>
  div {
    position:absolute;
    top: 0;
    bottom:0;
    left:0;
    right:0;
    background-color:red
  }
</style>
<script>
  window.onresize = e => {
    let width = window.innerWidth;
    let height = window.innerHeight;
    console.log(e.type, "nova mida: " + width + "px x " + height + "px");
  };

  let element = document.getElementById('test');

  element.onresize = e => console.log("S'ha disparat l'esdeveniment resize del
      <div>!");
</script>
<div id="test"></div>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/meKvMQ?editors=1111.

Com deveu haver vist, no obtenim la nova mida a partir de l'*event* sinó que el traiem del mateix objecte `window`. Ho fem així perquè aquesta informació no ens la proporciona l'*event*, així que no hi ha cap altra manera de fer-ho.

No tots els navegadors proporcionen aquesta informació de la mateixa manera, tot i que les versions actuals de Mozilla Firefox i Chrome sí que ho fan. Si volem assegurar-nos que obtenim la mida correcta, hem de fer una implementació més complexa tenint en compte les diferents opcions o recórrer a alguna llibreria com **jQuery**, que ja afegeix la nova mida dins de l'objecte *event*,

També deveu haver notat que tot i que la mida de l'element `div` sí que canvia (perquè ocupa tota la finestra), no es dispara l'*event* `resize`. Recordeu que aquest *event* només es dispara per l'objecte `window`.

No hi ha cap *event* que es dispari quan un element del document canvia de mida; per exemple, si no s'ha definit una mida fixa per a una imatge, quan es descarrega, canviarà la mida que té per la mida real, i aquest canvi no és detectable.
L'*event* `resize` només es dispara quan canvia la mida de la finestra; és a dir, quan redimensionem la finestra o canviem l'amplada de la secció de les eines del desenvolupador, perquè només es considera com a finestra del navegador la secció on es mostra la pàgina web.
Així que, en cas de necessitar fer aquestes comprovacions, l'única manera de fer-ho és **crear un temporitzador** que periòdicament comprovi la mida dels elements que ens interessi i, si ha canviat, reaccioni en conseqüència.

1.2.5 Events d'HTML5 i HTML5 media

Exposem aquí una llista d'*events* categoritzats com a HTML5 i HTML5 *media*, i només entrarem en detalls en aquells que presentin alguna peculiaritat. Comencem amb els tipus d'*events* generals que podeu trobar a la taula 1.9.

TAULA 1.9. Events d'HTML5 generals

Tipus	Operació	Descripció
beforeunload	onbeforeunload	Es dispara abans de descarregar la pàgina i ens permet cancel·lar l'operació

L'*event* `beforeunload` ens permet evitar que una pàgina sigui descarregada. Es dispara abans que l'*event* `unload` i, si dins de la funció que és notificada que s'ha disparat aquest *event* retornem un missatge, el mostrarà i demanarà confirmació abans de continuar.

Trobareu més informació sobre l'*event* `beforeundload` en aquest mateix apartat.

Vegem-ne un primer exemple molt senzill perquè quedi clar:

```
<script>
  window.onbeforeunload = e => 'Estàs segur que vols abandonar la pàgina?';
</script>
<p>Recarrega la pàgina per veure el missatge de confirmació.</p>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/YyvgzM?editors=1010.

Si proveu de recarregar la pàgina, us mostrarà un missatge demanant confirmació abans de sortir. Tingueu en compte que als navegadors més moderns no es pot configurar el text del missatge i en surt un per defecte.

> Segons la versió del navegador, **el missatge pot ser que no es mostri**, ja que el fet de mostrar o no el missatge de confirmació abans de sortir no forma part de l'especificació d'HTML5. Així doncs, hi ha navegadors com Chrome que mostren el missatge, mentre que d'altres com Mozilla Firefox no el mostren.

S'ha de tenir en compte que el valor que es retorna és irrellevant, perquè **sempre que es retorni quelcom** es mostrarà el diàleg, encara que sigui el valor *fals*.

Comprovareu que això és cert amb aquest exemple:

```
<script>
  window.onbeforeunload = e => false;
</script>
<p>Recarrega la pàgina per veure el missatge de confirmació.</p>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/XmYGbZ?editors=1010.

En canvi recordeu que, si no es retorna res, no es mostrarà el diàleg:

```
<script>
window.onbeforeunload = function(e) {
  console.log("No es mostra el diàleg");
};
</script>
<p>Recarrega la pàgina, però no veuràs cap diàleg.</p>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ojyVXJ?editors=1011 (Recordeu que heu d'activar l'opció per preservar el *log* o s'esborrarà abans de carregar la pàgina següent).

Això ens dóna molt de joc, perquè ens permet fer diverses coses, com comprovar si s'ha fet alguna modificació quan volem recarregar la pàgina o demanar confirmació només si s'han produït canvis, tal com es pot veure en l'exemple següent:

```
<script>
  let comptador = 0;
  let element = document.getElementById('test');

  element.onclick = () => {
    comptador++;
    console.log("Comptador:", comptador);
  }
```

```
9
10
11    window.onbeforeunload = e => {
12      if (comptador>0) {
13        return "El comptador ha canviat, segur que vols sortir?";
14      }
15    };
16  </script>
17  <button id="test">Augmenta comptador</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/pjKYjP?editors=1010.

Si proveu de recarregar la pàgina sense prémer el botó, no us demanarà confirmació perquè el comptador serà igual a 0; en canvi, en el moment en què el cliqueu, el comptador augmentarà i sí que us demanarà confirmació per recarregar la pàgina.

Vegem un exemple de quan es dispara l'*event* `input` (vegeu la taula 1.10) amb un quadre de text:

Taula 1.10. Events d'HTML5 relacionats amb els formularis

Tipus	Operació	Descripció
input	oninput	Es dispara cada vegada que es produeix un canvi a l'element
invalid	oninvalid	Es dispara quan s'intenta enviar un formulari i l'element no compleix amb els requisits que s'han imposat

```
1  <script>
2    let element = document.getElementById('test');
3
4    element.addEventListener('input', e => console.log("S'han produït canvis al
         quadre de text", e));
5  </script>
6  <input type="text" id="test"/>
```

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

Fixeu-vos que cada vegada que premeu una tecla o enganxeu, elimineu o retalleu el text, aquest *event* es dispara.

> L'*event* `input` és molt similar a `change`, però `input` es dispara quan es produeix qualsevol canvi, mentre que `change` només es dispara si el contingut ha canviat en perdre el focus. Així que **l'*event* `input` es disparà molt més sovint que** `change`.

S'ha de tenir en compte que no tots els elements disparen els *events* `input`, només ho fan aquells que tenen la propietat `contenteditable` i aquesta propietat té el valor de *cert*.

L'*event* `invalid` el veurem detalladament a l'apartat "Programació amb formularis", junt amb la resta d'*events* relacionats amb els formularis.

HTML5 media

Amb HTML5 es van afegir molts elements per millorar la gestió d'elements multimèdia (àudio i vídeo), i, juntament amb aquests elements, es van afegir també una gran quantitat d'*events* que poden servir-nos per millorar-ne el control; vegeu a la taula 1.11 un petit extracte d'aquests *events*:

Taula 1.11. Events d'HTML5 media

Tipus	Operació	Descripció
canplay	oncanplay	Es dispara quan és possible començar la reproducció del recurs d'àudio o vídeo
ended	onended	Es dispara quan finalitza la reproducció del recurs
pause	onpause	Es dispara quan es posa en pausa la reproducció
play	onplay	Es dispara quan s'inicia la reproducció del recurs
waiting	onwaiting	Es dispara quan es pausa la reproducció perquè hi ha una falta temporal de dades (per exemple, si la reproducció és en *streaming* i encara no s'ha acabat de descarregar el recurs sencer)

Com que el tractament d'elements multimèdia queda fora de l'abast d'aquests materials no en veurem cap exemple.

Per acabar, és interessant saber que hi ha una sèrie d'*events* relacionats amb la funcionalitat **Drag&Drop**, però no n'hem afegit detalls ni exemples perquè no està completament implementada de manera homogènia a tots els navegadors.

Drag&Drop

La funcionalitat Drag&Drop ens permet arrossegar elements per reorganitzar-los a la pàgina, per exemple, per canviar l'ordre d'una sèrie de fotos. És recomanable fer servir llibreries com jQuery si necessitem aquesta funcionalitat, ja que ens permeten aplicar-la amb molt poc esforç.

1.2.6 Events de les Web API estàndard

Encara que la llista de les Web API estàndard és força extensa, en aquesta secció ens centrarem a comentar els *events* relacionats amb algunes de les Web API més utilitzades.

Les quatre Web API que veurem són:

- **XMLHttpRequest**: permet fer peticions asíncrones al servidor.
- **Web Storage**: permet emmagatzemar dades de forma persistent al navegador.
- **WebSocket**: permet crear una connexió TCP amb un servidor, per exemple, per crear una aplicació de xat o un joc multijugador.

- **Web Workers**: tot i que els navegadors no permeten fer programes que aprofitin les possibilitats multifil dels ordinadors amb més d'un processador, gràcies a aquesta API sí que és possible treballar amb programació multifil, creant tasques que es realitzaran de forma paral·lela sense bloquejar el fil principal de la interfície.

Programació multifil

Programació que crea tasques que es realitzaran de forma paral·lela.

API XMLHttpRequest

Normalment, quan es realitza una petició via **AJAX**, el servidor ens retorna una resposta amb una sèrie de dades. Quan parlem de *descàrrega de dades* ens referim a aquestes dades (vegeu la taula 1.12).

Taula 1.12. Events de l'API XMLHttpRequest

Tipus	Operació	Descripció
abort	onabort	Es dispara quan es cancel·la la descàrrega de dades
error	onerror	Es dispara quan es produeix un error durant la descàrrega
load	onload	Es dispara quan finalitza la descàrrega amb èxit
loadend	onloadend	Es dispara quan s'atura la descàrrega, sigui quina sigui la raó (`error`, `abort` o `load`)
loadstart	onloadstart	Es dispara quan s'inicia la descàrrega de dades
readystatechange	onreadystatechange	Es dispara quan canvia la propietat `readyState`. Aquesta propietat és la que ens indica si s'ha completat amb èxit o no la petició.

API Web Storage

Aquest API permet guardar informació al navegador, de forma semblant a les galetes, però amb una capacitat d'emmagatzemage molt superior (vegeu la taula 1.13). S'ha de tenir en compte que aquest magatzem de dades és gestionat pel navegador i, per consegüent, si obriu la mateixa pàgina amb dos navegadors diferents, cadascun utilitzarà el seu magatzem independent.

Taula 1.13. Events de l'API Web Storage

Tipus	Operació	Descripció
storage	onstorage	Es dispara quan hi ha canvis al magatzem de dades

Cal destacar que la detecció de canvis en el magatzem de dades no es dispara a la pestanya on es produeixen els canvis (ja que l'aplicació responsable de fer canvis ha de saber, forçosament, què ha canviat), sinó que aquests són detectat per altres pestanyes obertes al mateix navegador.

Una altra peculiaritat és que, per escoltar els *events* del `Web Storage`, s'ha d'afegir la detecció d'*events* a l'objecte `window` i no pas al magatzem: `window.addEventListener('storage', callback);`.

API WebSocket

Aquesta API ens permet connectar amb un servidor via TCP. És una opció relativament recent, ja que abans d'HTML5, això no era possible i s'havien de fer servir altres tecnologies, com Flash o Java (vegeu la taula 1.14).

Connexions TCP

TCP és un protocol que permet establir connexions entre aplicacions client i servidor. En podeu trobar més informació al següent enllaç: www.goo.gl/Ibd0k3.

Taula 1.14. Events de l'API WebSocket

Tipus	Operació	Descripció
open	onopen	Es dispara quan s'estableix una connexió
close	onclose	Es dispara quan es tanca la connexió
message	onmessage	Es dispara quan es rep un missatge a través del WebSocket
error	error	Es dispara quan s'ha tancat la connexió amb perjudici, per exemple, amb pèrdua de dades

API Web Workers

L'API Web Workers només té un *event* propi que podem observar (vegeu la taula 1.15).

Taula 1.15. Events de l'API Web Workers

Tipus	Operació	Descripció
message	onmessage	Es dispara al *web worker* quan hi ha un missatge per comunicar a l'aplicació

2. Programació amb formularis

Els aspectes més importants que cal conèixer sobre els formularis són: per a què serveixen i com validar que la informació és correcta abans de fer-la servir. Amb l'arribada d'HTML5 els elements que formen l'estructura dels formularis van adquirir una gran quantitat de noves opcions i, d'aquesta manera, van facilitar-ne la validació i van oferir als navegadors l'opció de mostrar quadres d'entrada de dades més adients, com per exemple els selectors de dates.

Cal destacar la importància de la validació de les dades -aquest és un dels principals usos que es va donar a JavaScript inicialment. Gràcies a l'evolució del llenguatge HTML s'ha simplificat molt aquesta tasca, ja que ara és possible incrustar regles de validació directament a les etiquetes HTML. Tot i així, en alguns casos més complexos, com la càrrega de fitxers, continua requerint implementar la validació mitjançant JavaScript.

Una de les tècniques més potents per realitzar validacions complexes, reemplaçaments i extracció de textos és la utilització *expressions regulars*. Aquestes expressions permeten utilitzar patrons complexos per realitzar cerques a les cadenes de text. Gràcies a aquestes és possible simplificar el tractament de la informació guardada en galetes (*cookies* en anglès), ja que aquestes no utilitzen un format fàcil d'utilitzar.

2.1 Què és un formulari?

Segurament tots us haureu trobat amb formularis en paper en un moment o altre: es tracta d'un document amb uns textos que us demanen determinada informació, i uns espais en blanc per omplir-los. És a dir, la seva funció és **recollir informació** que més endavant **serà processada** d'una manera o altra.

En el cas que ens interessa el seu fi és el mateix, tot i que la seva forma és molt diferent. Pràcticament totes les pàgines que visitem diàriament tenen formularis per una banda o altra:

- Formularis per realitzar cerques com www.google.com.
- Formularis per identificar-nos en una pàgina com el de www.ioc.xtec.cat.
- Pàgines amb *quizzes* que, seleccionant unes opcions o les altres, ens mostren diferents resultats.
- Els comentaris que es publiquen a través de Facebook.
- I, per descomptat, la versió digital dels formularis paper d'ús habitual; com el de la declaració de la renda.

El seu funcionament és molt semblant als formularis en paper: ofereixen unes opcions o quadres de text per marcar o omplir; una vegada s'ha omplert aquesta informació, és **recollida, enviada/entregada i processada**.

S'ha de tenir en compte que un formulari només és un mitjà per introduir informació que serà enviada a un servidor o bé serà utilitzada per l'aplicació en JavaScript.

Exemples de formularis segons la finalitat

- Un exemple de formulari enviat a un servidor seria el de cerca de Google. Aquest formulari recull les dades necessàries per fer la cerca i, un cop l'usuari prem el botó, s'envien al servidor. Aquest fa la cerca amb les dades i retorna el resultat al navegador del client perquè el mostri a la finestra.
- Un exemple -molt simple- de formulari en JavaScript seria una aplicació amb dos quadres de text, en què s'han d'introduir dos nombres, i un botó que en clicar-lo mostri el resultat de la suma. En aquest formulari, quan l'usuari prem el botó, és el propi navegador qui directament fa el processament de les dades i mostra el resultat que obté.

Tots dos fan servir formularis, però amb objectius diferents.

2.2 Estructura d'un formulari

En general, tots els formularis comparteixen una mateixa estructura similar a aquesta:

```
<form>
  Introdueix el teu nom:
  <input type="text" />
  <br>
  Introdueix el teu cognom:
  <input type="text" />
    <br>
  <input type="submit" value="enviar" />
</form>
```

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

D'aquesta manera, tots els elements del formulari han d'anar niats dins del bloc format per l'etiqueta `form`, i tots els elements que es troben dins queden lligats a aquest formulari.

Podeu veure que dintre del formulari hi ha un element `input` que es repeteix 3 vegades. Els dos primers són de tipus `text` i això fa que es mostrin com una capsa per escriure text, mentre que el darrer és de tipus `submit` i es mostra com un botó.

Aquest últim element és imprescindible, ja que és el que ens permet enviar la informació del formulari al servidor.

No només hi ha elements de tipus `text`, sinó que es poden afegir altres tipus d'elements, com ara: llistes, àrees de text, llistes desplegables, botons, caselles de selecció o etiquetes per associar els camps del formulari.

Ara bé, aquest exemple no és funcional perquè el formulari no conté informació sobre on s'ha d'enviar ni com ho ha de fer, ni hi ha manera de saber a quin concepte corresponen els quadres de text. Tota aquesta informació s'afegeix mitjançant els atributs dels elements `form` i `input`.

Un exemple més complet seria el següent:

```
<form action="test.php" method="GET">
  <label>Introdueix el teu nom:
    <input type="text" name="nom"/></label>
  <br>
  <label>Introdueix el teu cognom:
  <input type="text" name="cognom"/>

  </label>
  <br>
  <input type="submit" value="enviar" />
</form>
```

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

Aquest exemple és més interessant; hem afegit a l'element `form` els atributs `action="test.php"` i `method="GET"`. El primer indica que, en clicar el botó d'enviament, s'ha de passar aquesta informació a la pàgina `test.php`, mentre que el segon indica que el mètode que farem servir per passar aquesta informació és `GET`.

Quan les pàgines a les quals dirigim les nostres peticions no existeixen, sempre es retorna una pàgina d'error amb el codi 404.

Si obriu les eines de desenvolupador del vostre navegador, a la pestanya de xarxa (*Network* en anglès), en clicar el botó *enviar* veureu una petició (que retornarà error 404, pàgina no trobada) fent servir el mètode `GET`.

Protocol HTTP1.1

Aquest protocol és clau per al funcionament d'Internet. Entre altres elements, el protocol inclou una llista de verbs (equivalents als mètodes) i codis d'estat que són retornats juntament amb les respostes de peticions, com per exemple el codi 404 quan no es troba una pàgina o el codi 200 quan la pàgina s'ha rebut amb èxit. En podeu trobar més informació a l'enllaç següent: wwww.goo.gl/kUzYf8.

Els formularis poden fer servir els mètodes `POST` o `GET`, que es corresponen amb els mètodes del mateix nom per al protocol HTTP. Tot i que el protocol *HTTP1.1* admet més mètodes com `PUT` i `DELETE`, l'especificació d'HTML5 no els admet. Existeixen *frameworks* que afegeixen aquesta funcionalitat fent l'enviament com a `POST`, però afegint un camp ocult que indica el mètode que es vol fer servir.

Serveis Web RESTful

RESTful és una arquitectura de serveis web que es basa a fer servir un mateix URL amb diferents verbs per realitzar diferents accions. Per exemple, suposant que tenim un servei web que permet gestionar l'usuari, podríem tenir els següents URL:

- `www.exemple.cat/usuaris/joan` (GET): retorna les dades de l'usuari `joan`.
- `www.exemple.cat/usuaris/joan` (PUT): modifica les dades de l'usuari `joan`.
- `www.exemple.cat/usuaris/joan` (DELETE): elimina l'usuari `joan`.
- `www.exemple.cat/usuaris/` (POST): crea un usuari nou.
- `www.exemple.cat/usuaris/` (GET): retorna la llista completa d'usuaris.

Tot i que HTML no admet l'ús de tots els verbs des dels formularis, sí que és possible fer crides *AJAX* que els facin servir.

2.2.1 Elements d'un formulari

Dintre de les etiquetes `form` podem trobar els mateixos elements que en qualsevol document HTML, per aquesta raó només ens centrarem en els que són específics dels formularis, en la seva estructuració i en l'enviament de dades.

Aquesta és la llista dels elements que s'utilitzen més sovint a l'hora de crear un formulari:

- **form**: element principal que forma un formulari; totes les dades definides en elements que es trobin entre l'etiqueta d'apertura i tancament seran enviats en clicar el botó d'enviament.
- **input**: aquest element permet que l'usuari introdueixi dades, i pot representar-se de moltes maneres diferents: botons, quadres de text, selectors de colors...
- **textarea**: serveix únicament per introduir textos. A diferència de l'`input` de tipus `text`, el `textarea` és multilineal i s'utilitza generalment per la introducció de textos llargs.
- **button**: aquest element crea un botó clicable que, per defecte, el seu comportament és enviar el formulari. Pot donar problemes en versions de navegadors antigues.
- **label**: per tal de millorar l'estructura semàntica del document és recomanable fer servir aquest element per afegir les etiquetes corresponents a cada element d'entrada de dades.
- **datalist** (HTML5): aquest element ens permet afegir llistes al nostre codi HTML que poden ser utilitzades per altres elements del formulari.

Si volem elaborar llistes desplegables més complexes, haurem de fer servir els elements `select`, `option` i `optgroup`:

Tenim altres elements que es poden considerar opcionals, ja que no afecten com s'envia la informació sióo com s'estructuren les opcions:

- **select**: embolcalla tots els elements que formen la llista, i inclou l'atribut que determina el nom del paràmetre amb què s'enviarà la informació a l'atribut `name`.
- **option**: aquest element conté el valor que es passarà en enviar el formulari i el text que es veurà en desplegar la llista que correspon al seu atribut `value`.
- **optgroup**: serveix per agrupar les opcions de la llista, separades per un títol no seleccionable que s'especifica al seu atribut `label`.

Al contrari dels elements que es poden utilitzar individualment, aquestes llistes s'han de crear amb una estructura específica; vegem-ho:

```
<form action="test.php" method="GET">
  <select name="selector">
    <option value="SEU_A">Barcelona</option>
    <option value="SEU_B">Tarragona</option>
    <option value="SEU_C">Lleida</option>
    <option value="SEU_D">Girona</option>
</select>
  <input type="submit" />
</form>
```

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

Com es pot veure en la pestanya de xarxa de les eines del desenvolupador, en fer clic sobre el botó d'enviar s'envia al servidor el paràmetre anomenat `selector` amb el valor de l'opció seleccionada, per exemple `SEU-A` si se selecciona *Barcelona*: `test.php?selector=SEU-A`

La diferència que trobem entre aquest tipus de llista i fer servir un `datalist` amb un element `input` és que aquestes llistes obliguen a seleccionar un dels elements (no es pot escriure el que vulguem) i a més permet afegir grups d'opcions gràcies a l'element `optgroup`.

```
<form action="test.php" method="GET">
  <select name="assignatura">
    <optgroup label="Primer Curs">
    <option value="M01">Sistemes operatius</option>
    <option value="M03">Programació</option>
    </optgroup>
    <optgroup label="Segon Curs">
    <option value="M06">Desenvolupament d'aplicacions en entorn client</option>
    <option value="M09">Disseny d'interfícies Web</option>
    </optgroup>
</select>
  <input type="submit" />
</form>
```

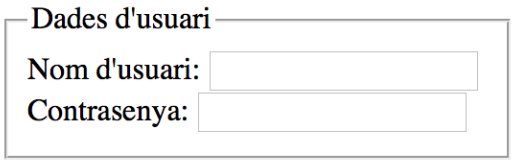
Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/mVvVaW?editors=1000.

En aquest cas es pot comprovar que la llista desplegable inclou les categories *Primer Curs* i *Segon Curs*, que tenen la funció de separar tots dos grups d'assignatures, i no són seleccionables.

També trobem alguns elements que, si bé no tenen efecte sobre l'enviament de dades, sí que milloren l'estructura semàntica del formulari, ja que agrupen els possibles conjunts de dades indicant clarament de què es tracta, en lloc de fer servir capçaleres i elements `div` genèrics; com ara `fieldset` i `legend`:

- **fieldset**: permet agrupar els elements d'un formulari (vegeu la figura 2.1).

FIGURA 2.1. Representació d'un "fieldset"



- **legend**: afegeix un títol dins d'un `fieldset`.

Vegem-ne el funcionament:

```
<style>
fieldset {
  width: 240px;
}
</style>
<fieldset>
  <legend>Dades d'usuari</legend>
  <label>Nom d'usuari:
    <input type="text" name="nom_usuari" /><br>
  </label>
  <label>Contrasenya:
    <input type="password" name="contrasenya" />
  </label>
</fieldset>
```

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

2.2.2 Element 'form'

Atributs de 'form'

Podeu trobar una descripció completa de les propietats de l'element `form` en l'enllaç següent: www.goo.gl/p5DRi8.

Tal com passa amb altres elements, si volem poder identificar fàcilment un formulari, hem d'especificar el seu atribut `id`, que ha de ser únic entre els elements de la pàgina. Això ens permet lligar-lo a elements externs a l'estructura del formulari, i accedir de forma fàcil i ràpida a través de JavaScript.

L'element `form` accepta dos mètodes per enviar les dades: `POST` i `GET`. A banda de la intenció que tinguem quan enviem les dades, cal tenir en compte algunes diferències importants entre tots dos; vegem-les:

Limitació de la mida de l'URL

Si volem assegurar-nos que les peticions `GET` es fan correctament, hem de tenir en compte que els navegadors més antics no admeten URL de més de 255 bytes; per exemple, Internet Explorer només admet fins a 2.083 caràcters (www.goo.gl/Xj7loh).

- **GET**: els valors dels paràmetres es passen al servidor a través de l'URL. El format és l'URL de destí, seguida d'un interrogant (`?`), afegint els parells de paràmetre-valor units amb un signe d'igual (`=`), i encadenant els següents paràmetres amb el símbol (`&`). Per exemple, si tenim els paràmetres `nom` i `cognom`, en enviar el formulari l'URL resultant podria ser semblant a `test.php?nom=john&cognom=doe`. A causa de la manera com es passen les dades hi ha una limitació de longitud imposada pels servidors -no per l'especificació-, i **no s'ha de fer servir `GET` per enviar grans volums de dades o text**.
- **POST**: aquest mètode afegeix els paràmetres dins de la capçalera, de manera que ni és visible a l'URL ni té la mida limitada. Cal tenir en compte que encara que no siguin visibles a l'URL **no estan encriptats ni amagats**: des del panell de xarxa de les eines de desenvolupador dels navegadors es poden veure els paràmetres enviats amb la capçalera.

> Es recomana fer servir el mètode `GET` per l'enviament d'un volum de dades baix o si volem que es generi un URL amb els paràmetres. Els URL generats d'aquesta manera ens permeten fer proves més fàcilment o guardar els enllaços per visitar-los posteriorment.
> En tots els altres casos, com poden ser enviar fitxers o volums importants de dades o si desitgem que no es vegin a l'URL els paràmetres enviats, farem servir `POST`.

Un altre atribut molt important és el `enctype`. Aquest atribut especifica el tipus de format (*media type* o *MIME type*) que s'envia al servidor:

- **application/x-www-form-urlencoded**: aquest és el **valor per defecte** que s'assigna si no s'especifica res.
- **multipart/form-data**: aquest valor és el que s'ha d'especificar **si volem enviar fitxers** juntament amb el nostre formulari. Amb aquest tipus de format sempre s'ha de fer servir el mètode `POST`.

Així doncs, un formulari amb enviament de fitxers quedaria així:

```
1 <form action="test.php" method="POST" enctype="multipart/form-data" id="form-
      principal">
2   <input type="file" />
3   <input type="submit" />
4 </form>
```

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

Amb HTML5 s'han afegit algunes propietats als botons (tant els elements `button` com els `input` de tipus botó) que permeten modificar com s'envia el formulari. Aquestes propietats sobreescriuen el comportament normal en enviar el formulari clicant el botó; són les següents:

- **formaction** especifica on s'envien les dades del formulari.
- **formenctype** especifica el tipus de format per enviar les dades.
- **formmethod** especifica el mètode que es farà servir per enviar les dades.
- **formnovalidate** salta la validació de les dades.

2.2.3 Element 'input'

L'element `input` es va enriquir moltíssim amb HTML5. Mentre que altres elements es mantenen pràcticament igual que en versions anteriors, a aquest se li han afegit una gran quantitat de tipus que no són disponibles als navegadors antics o fins i tot en alguns navegadors actuals. Per aquesta raó, cal distingir entre els elements que són compatibles amb versions anteriors i aquells que es van afegir amb HTML5 (podeu comprovar les compatibilitats amb els diferents navegadors a la pàgina https://caniuse.com/?search=input%20type).

Fixeu-vos que l'element `input`, al contrari que la majoria dels elements HTML, s'autotanca: `type="text" />`. Com que no ha de contenir res al seu interior **no cal afegir una etiqueta de tancament independent**, igual que passa amb l'element `img`.

En general, quan parlem d'un element de tipus `input`, ens referim a un camp del formulari.

Atributs d'input

Podeu trobar una llista completa dels atributs d'`input` al següent enllaç: www.goo.gl/jSaZgH.

Els atributs comuns d'ús més freqüent i, per tant, compatibles amb versions anteriors, són:

- **id**: ens permet identificar unívocament l'element, de manera que el podem manipular fàcilment des de codi, i ens serveix per enllaçar-lo amb elements `label`.
- **name**: aquest atribut és indispensable per poder enviar dades, ja que el seu valor és el que es fa servir com a nom del paràmetre.
- **value**: valor per defecte o valor actual del camp, segons el seu tipus.
- **disabled** (booleà): quan s'aplica aquest atribut el camp estarà desactivat i es mostrarà amb un estil diferent per destacar-ho. Els valors dels camps amb aquest atribut **no s'envien amb el formulari**.
- **readonly**: similar a `disabled`, però no canvia l'estil de representació i

només funciona amb alguns tipus. A diferència de `disabled`, els valors d'aquests camps **sí que s'envien amb el formulari**.

- **size**: permet establir la mida del control, **en caràcters, si conté text**, o en píxels si es tracta d'altres tipus i treballem amb navegadors antics. En HTML5 aquest atribut només s'aplica a elements que contenen text, així que sempre s'aplica com a nombre de caràcters.
- **maxlength**: en els camps de tipus text com `text`, `email`, `password`, etc. especifica el nombre màxim de caràcters que admet.

Proveu l'exemple següent per veure la diferència entre els camps activats i els desactivats:

L'atribut booleà

Un atribut booleà només pot tenir dos valors: `true` i `false` (cert i fals); aquests valors són admesos pels operadors lògics (&&, || i !). En el cas dels formularis, l'existència de l'atribut equival a `true` i l'absència a `false`, independentment del valor assignat.

Quan fer servir la propietat 'size'

No és recomanable fer servir la propietat `size`, ja que per modificar l'estil és més apropiat utilitzar CSS. Però si es fa servir juntament amb la propietat `maxlength` s'aconsegueix un quadre de text de la mida exacta.

```
<input type='text' value="aquest text es troba activat" /><br>
<input type='text' disabled value="aquest text es troba desactivat" /><br>
<label>Casella desactivada
  <input type='checkbox' disabled />
</label><br>
<label>Casella activada
  <input type='checkbox' />
</label><br>
<label>Casella desactivada i marcada
  <input type='checkbox' disabled checked/>
</label><br>
<label>Casella activada i marcada
  <input type='checkbox' checked/>
</label>
```

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

En el següent exemple, podeu veure la diferència entre un camp amb la mida establerta com a propietat `size` o fent servir CSS, i l'efecte de la propietat `maxlength`:

```
<style>
  label {
    display:block; /* això evita haver d'afegir un salt de línia */
  }
  .mida {
    width: 5em;
  }
</style>
<label>Quadre amb size="5"
  <input type="text" size="5" />
</label>
<label>Quadre amb estil CSS width="5em"
  <input type="text" class="mida" />
</label>
<label>Quadre amb size="5" i maxlength="5"
  <input type="text" size="5" maxlength="5" />
</label>
<label>Quadre amb estil CSS width="5em" i maxlength="5"
  <input type="text" class="mida" maxlength="5"/>
</label>
<label>Quadre amb size="10" i maxlength="5"
  <input type="text" size="10" maxlength="5" />
</label>
```

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

Entre els elements que **només estan disponibles per a HTML5** trobem els següents:

- **placeholder**: si s'especifica aquest atribut, es mostra un text quan el camp és buit; generalment s'aprofita per indicar a l'usuari què s'espera que s'introdueixi en aquest camp, per exemple: *introdueix el teu nom*. Aquest atribut només està disponible per als elements que contenen text i no compta com a contingut per al camp, és a dir, si s'envia el formulari i no hem afegit cap text, s'enviarà buit.
- **autofocus** (booleà): només es pot fer servir en un element per document. Fa que aquest element sigui el seleccionat automàticament en carregar la pàgina.
- **form**: permet indicar l'`id` d'un formulari al qual s'enllaçarà aquest element. Gràcies a aquest atribut és possible afegir un element fora de l'estructura del formulari però que les seves dades s'enviïn juntament amb el formulari amb l'`id` especificat.
- **autocomplete**: habilita la funció d'autocompletar del navegador per al tipus que s'indiqui, per exemple: `email`, `tel`, `bday`, `postal-code`, etc.
- **list**: s'utilitza juntament amb l'element `datalist`; afegeix la llista especificada a aquest atribut com a valors per mostrar.
- **required** (booleà): quan hi ha aquest atribut, el camp ha de tenir algun valor abans de poder enviar el formulari. No tots els camps admeten aquest atribut.
- **pattern**: el valor és una expressió regular que es fa servir per validar el contingut del camp. Només s'aplica quan el tipus de camp conté text.

Les expressions regulars es veuran detalladament en la secció "Expressions regulars" d'aquest mateix apartat.

Fixeu-vos com fent servir l'atribut `placeholder` es fa molt més entenedor com s'ha d'omplir el formulari:

```
<style>
  label {
    display: block;
  }

  input {
    width: 20em;
  }
</style>
<label>
  <input type="text" name="dni" placeholder="Introdueix el teu DNI incloent la
      lletra"/>
  DNI
</label>
<label>
  <input type="email" name="email" placeholder="Introdueix una adreça de correu
       vàlida"/>
  Correu electrònic
</label>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/zreXBL?editors=1100.

Vegem un exemple de com s'enllaça un element extern a l'estructura del formulari:

```
<form id="principal" action="test.php" method="GET">
  <input type="text" name="element–intern" value="intern" />
  <input type="submit"/>
</form>
<input type="text" name="element–extern" value="extern" form="principal" />
```

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

Si us fixeu en el panell de xarxa de les eines de desenvolupador, veureu que en fer clic sobre el botó "enviar" la petició que s'envia conté els paràmetres `element-intern` i `element-extern`, tot i que el camp corresponent a `element-extern` es troba fora de l'estructura del `form`.

Vegem, ara, el funcionament d'un element `data-list` juntament amb l'atribut `list` en un altre camp:

```
<label>Selecciona una seu:
  <input list="seus" name="seu" />
</label>
<datalist id="seus">
  <option value="Barcelona">
  <option value="Girona">
  <option value="Lleida">
  <option value="Tarragona">
</datalist>
```

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

Fixeu-vos que l'element `input` conté com a atribut `list` l'identificador de l'element `datalist`.

Tipus de l'element 'input': l'atribut 'type'

Encara que es tracti només d'un atribut, les diferències entre seleccionar un tipus o un altre són tan importants que cal tractar-les en una secció a part.

D'una banda, podem definir dos grans grups:

- Els que tracten amb text; com serien el tipus `text`, `password` o `email`.
- Els que afegeixen un component completament diferent; com són una casella de selecció (`checkbox`), un botó de selecció única (`radiobutton`) o un botó per afegir fitxers (`file`).

D'altra banda, amb HTML5 es van afegir una gran quantitat de components amb noves funcionalitats, com el tipus `time` per introduir hores; `date` per introduir dates; `email`, que es tracta com un camp de text però valida automàticament que sigui una adreça de correu vàlida, o `range`, que mostra una barra per seleccionar un valor entre un mínim i un màxim.

En la llista següent, trobareu alguns dels diferents tipus que admeten tots els navegadors:

- **button**: el camp es tracta com un botó sense cap comportament per defecte.
- **submit**: es mostra com un botó però el seu comportament per defecte és enviar el formulari en fer clic.
- **reset**: en aquest cas també es mostra un botó, i el comportament per defecte és restablir tots els camps del formulari.
- **image**: en cas de voler fer servir una imatge personalitzada en lloc d'un botó d'enviament es pot fer servir aquest tipus.
- **text**: quadre de text bàsic per introduir qualsevol tipus de text; es mostra en una sola línia.
- **password**: és com el tipus `text`, però no mostra els caràcters introduïts per pantalla.
- **hidden**: els camps d'aquest tipus no són visibles a la pàgina, tot i que el seu valor s'enviarà juntament amb el formulari.
- **checkbox**: el camp serà una casella de selecció.
- **radio**: afegeix un botó de selecció única que només permet seleccionar un dels valors; si se'n selecciona un altre, es desselecciona l'anterior. Es considera que **tots els botons de selecció única que tenen la propietat** `name` **igual pertanyen al mateix grup**.
- **file**: afegeix un component d'enviament de fitxers que consta d'un botó i una etiqueta que indica el fitxer (o fitxers en HTML5) que s'enviaran juntament amb el formulari.

Camps ocults (hidden)

Els camps ocults ens permeten afegir informació que s'ha d'enviar amb el formulari, però que no ha de ser modificada per l'usuari. Per exemple, en un formulari per demanar més informació d'un producte que es trobi en una pàgina cal incloure l'identificador únic del producte com a valor ocult, ja que aquest no ha de ser modificat per l'usuari.

Recordeu que per poder enviar fitxers juntament amb el formulari, cal fer servir el mètode `POST` i l'`enctype` ha de ser `multipart/form-data`.

L'estil de l'element de tipus `file` **no es pot modificar mitjançant CSS**; ja que la implementació d'aquest component no forma part de l'especificació d'HTML i cada fabricant la fa d'una manera diferent. Les solucions per donar format a aquest element impliquen implementar el nostre propi component en JavaScript o fer servir biblioteques externes.

Vegeu, en aquest exemple, l'aspecte de cadascun d'aquests tipus:

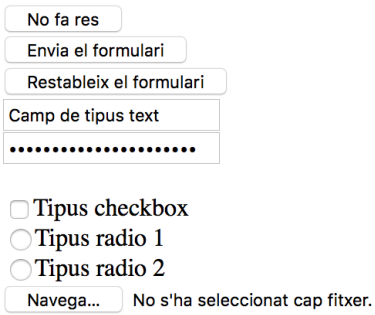
```
<form enctype="multipart/form-data" method="POST" action="test.php">
  <input type="button" value="No fa res" /><br>
  <input type="submit" value="Envia el formulari" /><br>
  <input type="reset" value="Restableix el formulari" /><br>
  <input type="text" value="Camp de tipus text" /><br>
  <input type="password" value="Camp de tipus password" /><br>
  <input type="hidden" value="Camp ocult" /><br>
  <label>
```

```
    <input type="checkbox" />Tipus checkbox<br>
  </label>
  <label>
    <input type="radio" name="exemple"/>Tipus radio 1<br>
  </label>
  <label>
    <input type="radio" name="exemple"/>Tipus radio 2<br>
  </label>
  <input type="file" />
</form>
```

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

El resultat que obtindreu serà similar al de la figura 2.2.

FIGURA 2.2. Representació d'elements 'input' d'HTML



Podeu comprovar que com que els dos botons de selecció única tenen el mateix nom (`exemple`), només n'hi pot haver un de seleccionat en un moment donat.

Amb HTML5 es van afegir molts més tipus, amb funcionalitats que abans requerien ser programades pel nostre compte en JavaScript o fer servir llibreries externes. Així, s'ha de tenir en compte que no existeixen en navegadors antics o que fins i tot poden no ser compatibles amb alguns dels navegadors actuals. Per aquesta raó, si voleu que funcioni correctament en tot tipus de navegadors, s'ha de proporcionar a l'usuari una alternativa.

Incongruències entre l'atribut 'value' i el contingut d'un camp

En alguns casos ens trobarem que hi ha discrepàncies entre l'un i l'altre. Això és a causa del fet que l'especificació d'HTML indica un format que és el que s'utilitza internament, mentre que a la representació del navegador se n'utilitza una altra de basada en la configuració de llenguatge i idioma. Per exemple, ens trobem amb aquests casos quan treballem amb nombres reals i dades.

En la següent llista, trobareu alguns dels diferents tipus que no admeten els navegadors antics:

Alguns dels nous tipus són versions modificades dels originals, que afegeixen capacitats de validació, com per exemple el tipus `email`.

- **email**: obliga a introduir una adreça de correu vàlida per enviar el formulari.
- **number**: només permet escriure nombres, i per passar la validació cal que sigui un nombre vàlid.
- **url**: és un quadre de text que obliga a introduir una adreça URL vàlida,

incloent-hi el protocol.

- **range**: aquest tipus mostra una barra per seleccionar el valor desplaçant el marcador. Es pot configurar fent servir els atributs `max`, `min` i `step` per indicar els valors màxims, mínims i els increments respectivament.
- **date**: serveix per introduir dates, i pot mostrar un selector de dates que ens permet seleccionar-les en lloc d'afegir-les manualment.
- **time**: és similar a l'anterior però per introduir hores i minuts; en aquest cas no es mostra cap component extra per seleccionar-les.
- **color**: permet seleccionar un color de dintre d'una paleta de colors.

Vegeu en aquest exemple l'aspecte de cadascun d'aquests tipus:

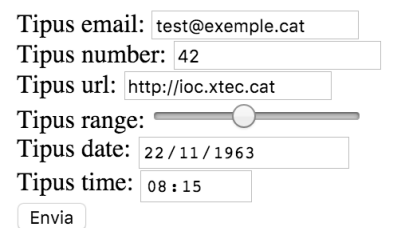
```
<form>
  <label>Tipus email:
    <input type="email" value="test@exemple.cat"/>
  </label><br>
  <label>Tipus number:
    <input type="number" value="42"/>
  </label><br>
  <label>Tipus url:
    <input type="url" value="http://ioc.xtec.cat"/>
  </label><br>
  <label>Tipus range:
    <input type="range" value="42" min="1" max="100" step="2"/>
  </label><br>
  <label>Tipus date:
    <input type="date" value="1963-11-22"/></label><br>
  <label>Tipus time:
    <input type="time" value="08:15"/></label><br>
  <input type="submit" />
</form>
```

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

Fixeu-vos en el camp de tipus `date`: tot i que el valor s'ha afegit amb el format `aaaa-mm-dd` (el corresponent a l'especificació d'HTML), el format que es mostra al navegador és `dd/mm/aaaa`, el corresponent a la nostra llengua. Una cosa semblant passa amb els decimals: en el codi hem de fer servir el punt com a separador, mentre que per introduir les dades es fa servir la coma.

A la figura 2.3 podeu veure un exemple de representació de diferents elements afegits a HTML5.

Figura 2.3. Representació d'elements input d'HTML5



Tot i que l'element `input` accepta més tipus, en aquests materials ens hem limitat a enumerar els que s'utilitzen més habitualment.

2.2.4 Element 'textarea'

Aquest element té un funcionament similar al d'un element `input` de tipus text, però la diferència principal és que aquest és multilineal, fet que permet escriure extensions més grans de text i incloure salts de línia.

Primerament, veurem algunes diferències de funcionament entre aquest element i l'element `input`. A diferència d'aquest últim, el `textarea` no s'autotanca: s'ha d'afegir l'etiqueta de tancament, i el valor del camp és el que es troba entre l'etiqueta d'obertura i de tancament, per exemple:

```
<script>
  let textarea = document.getElementById('test');

  function mostrarValor() {
    alert(textarea.value);
  }
</script>

<textarea id="test">Aquest es el valor del camp</textarea>
<br>
<button onclick="mostrarValor();">Mostrar valor</button>
```

Fixeu-vos que el valor inicial s'obté del text contingut entre les etiquetes d'apertura i tancament del `textarea`, però una vegada que el modifiqueu, el valor que es mostra en clicar el botó és el que conté el `textarea`.

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/LGoowV?editors=1010.

Vegem-ne ara un altre exemple molt semblant:

```
<script>
  let textarea = document.getElementById('test');

  function mostrarValor() {
    alert(textarea.value);
  }
</script>

<textarea id="test" value="Aquest es el valor del camp"></textarea>
<br>
<button onclick="mostrarValor();">Mostrar valor</button>
```

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

En aquest cas hem establert el mateix text que abans com a valor de la propietat del `textarea`, però **tant el `textarea` com el missatge d'alerta mostrat pel**

navegador són buits. És a dir, si ho establim com a valor des de la declaració HTML, aquest valor és ignorat.

Però què passa si intentem establir el valor del `textarea` per codi?

```
<script>
  let textarea = document.getElementById('test');

  textarea.value = "Valor establert via codi";

  function mostrarValor() {
    alert(textarea.value);
  }
</script>

<textarea id="test"></textarea>
<br>
<button onclick="mostrarValor();">Mostrar valor</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/vLwqNg?editors=1010.

Com es pot veure, es mosta el text introduït per codi. Per tant, podem concloure que:

- Si necessitem afegir un text per defecte a un `textarea` des de l'HTML, ho hem de fer entre les etiquetes d'apertura i tancament.
- Si ho hem de fer mitjançant codi, ho farem a través de la propietat `value`.

Finalment ens queda un dubte per resoldre: com reacciona el `textarea` en establir el seu contingut amb la propietat `innerHTML`?

```
<script>
  let textarea = document.getElementById('test');

  textarea.innerHTML = "Valor establert via codi amb innerHTML";

  function mostrarValor() {
    alert(textarea.value);
  }
</script>

<textarea id="test" value="Aquest és el valor del camp"></textarea>
<br>
<button onclick="mostrarValor();">Mostrar valor</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/xZNoEG?editors=1010.

L'efecte és similar, però s'ha de tenir en compte una diferència important: si inspeccionem l'element `textarea` amb les eines de desenvolupador, veurem que si establim l'atribut `value` d'aquest element el codi HTML inspeccionat no ha canviat, continua buit. En canvi, si ho fem mitjançant `innerHTML`, en inspeccionar l'element veurem que el valor és entre les etiquetes del `textarea` perquè amb `innerHTML` **es modifica l'estructura del document**.

Tot i que la funció del `textarea` és la mateixa que la del `input` de tipus `text`, té unes diferències de funcionament importants que s'han de tenir en compte.
Podem canviar el contingut d'un `textarea` tant a través de la propietat `value` com de la propietat `innerHTML`, però **es recomana fer servir** `value`, ja que és més coherent amb el tractament de dades, mentre que `innterHTML` en modifica l'estructura.

El `textarea` compta amb moltes menys opcions que l'element `input`, ja que aquest element acompleix una funció molt específica. Tot i així s'ha afegit noves propietats a HTML5 i cal diferenciar-les; si més no, per ser conscient de quines propietats no són compatibles amb navegadors antics. Vegem, primer, les propietats de les versions anteriors:

- **id**: idèntic que en el cas de l'element `input`. Ens permet identificar-lo unívocament i enllaçar-lo amb elements `label`.
- **name**: igual que en l'element input, aquest és el nom del paràmetre que s'enviarà amb el formulari.
- **value**: aquesta propietat no té cap efecte si l'apliquem com a etiqueta HTML, en canvi, ens permet obtenir el contingut del `textarea` quan hi accedim des de JavaScript.
- **disabled** (booleà): quan s'aplica aquest atribut el contingut no serà modificable, es presentarà amb un estil diferent, i els seus continguts no s'enviaran amb el formulari.
- **readonly** (booleà): similar a l'anterior, però en aquest cas les dades sí que són enviades.
- **cols**: especifica l'amplada de l'àrea de text en caràcters, de manera similar al funcionament de l'atribut `size` amb els camps de text.
- **rows**: nombre de files de text que es veuran. S'ha de tenir en compte que encara que `cols` i `row` tracten amb l'aparença en lloc de l'estructura, no es pot fer servir CSS en aquest cas perquè CSS no treballa amb caràcters.
- **selectionStart**: indica l'índex del primer caràcter seleccionat.
- **selectionEnd**: indica l'índex de l'últim caràcter seleccionat. Això, combinat amb l'anterior propietat, ens permet capturar el valor de la selecció mitjançant JavaScript.

Aquest és un petit exemple de com capturar el text seleccionat en un `textarea`:

```
<script>
  let textarea = document.getElementById('test');

  function mostrarSeleccio() {
    let seleccio,
      start,
      length;
```

Mètode String.substr(start, length)

El mètode `substr()`, dels objectes de tipus `String`, permet obtenir el fragment de la cadena que va des de la posició indicada pel primer paràmetre fins a aquesta posició més el valor del segon paràmetre. Per exemple: `"hola mon".substr(5,3)` retorna `mon`.

```
8
9       start = textarea.selectionStart;
10      length = textarea.selectionEnd - textarea.selectionStart;
11      seleccio = textarea.value.substr(start, length);
12      alert(seleccio);
13    }
14  </script>
15  <textarea id="test">Contingut de prova</textarea>
16  <br>
17  <button onclick="mostrarSeleccio();">Mostrar seleccio</button>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/XXwLVJ?editors=1010J.

D'altra banda, també podem fer servir JavaScript per fer la selecció d'un text, tal com podeu veure en aquest exemple: permet cercar dins del `textarea` i seleccionar el primer mot que coincideixi amb la cerca; vegem-ho:

Una altra opció per establir la selecció d'un `textarea` és fer servir el mètode `setSelectionRange(selectionStart, selectionEnd)`, proporcionat per la interfície de l'element `textarea`.

```
1   <script>
2     let text = document.getElementById('text'),
3         cerca = document.getElementById('mot');
4
5     function cercar() {
6
7       let mot = cerca.value,
8         start = text.value.indexOf(mot);
9
10      if (start > 0) {
11        text.selectionStart = start;
12        text.selectionEnd = start + mot.length;
13        text.focus();  //necessari en alguns navegadors (p.e. a Firefox) perquè
              es visualitzi la selecció
14
15      } else {
16        alert("No s'ha trobat el mot");
17      }
18    }
19  </script>
20  <label>Mot a cercar:
21    <input type="search" id="mot"/>
22  </label><button onclick="cercar();">cercar</button><br>
23  <textarea id="text" readonly cols="150" rows="7">--Ja era passat l'any e lo dia
      , e les festes eren complides de solemnizar, com la magestat del senyor
      rey tramés a preguar a tots los stats que·s volguessen esperar alguns dies
       per ço com la magestat sua volia fer publicar una fraternitat, la qual
      novament havia instituïda, de XXVI cavallers, sens que negú no fos
      reproche. E tots de bon grat foren contents de aturar. E la causa e
      principi de aquesta fraternitat, senyor, és stada aquesta, ab tota veritat
      , segons yo e aquests cavallers que açí són havem hoïda reçitar per boca
      del senyor rey mateix: com, un dia de solaç que·s fehien moltes danças e
      lo rey, havent dançat, restà per reposar al cap de la sala, e la reyna
      restà a l'altre cap ab les sues donzelles, e los cavallers dançaven ab les
       dames; e fon sort que una donzella, dançant ab un cavaller, apleguà fins
      en aquella part hon lo rey era e, al voltar que la donzella féu, caygué-li
       la liguacama de la calça, e al parer de tots devia ésser de la sinestra
      cama, e era de çimolça.
24  </textarea>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/BjegbG?editors=1010.

Vegem ara les propietats afegides amb HTML5, molt similars a les que trobem a l'element `input`:

- **placeholder**: com en el cas de l'element `input`, afegeix un text per indicar què s'espera que s'introdueixi en aquest camp. Aquest text no s'envia amb el formulari, i si no s'introdueix res al `textarea`, s'enviarà buit.
- **required**: serveix per validar el camp i ens obliga a introduir-hi quelcom abans de poder enviar el formulari.
- **maxlength**: limita el nombre de caràcters que es poden introduir.
- **minlength**: estableix el nombre mínim de caràcters per poder enviar-lo. **No s'aplica si el camp es deixa buit**; si volem controlar que no sigui buit hem d'afegir també la propietat `required`.
- **form**: igual que en el cas de l'element `input` aquesta propietat permet associar el `textarea` a un formulari en el qual no està inclòs.

En aquest exemple podeu veure aquestes propietats en funcionament:

```
<form method="GET" action="test.phpt" id="principal">
  <input type="submit" />
</form>
<textarea name="text" form="principal" minlength="5" maxlength="50" cols="100"
    rows="5" required>Text de prova que passa la validació</textarea>
```

Podeu veure aquest exemple en aquest enllaç: codepen.io/ioc-daw-m06/pen/adreVV?editors=1010; i el resultat a la figura 2.4.

Figura 2.4. Formulari amb textarea

2.2.5 Element 'button'

Aquest element, com es pot deduir pel seu nom, serveix per representar botons; tal com els que es mostren amb l'element `input`, amb els tipus: `button`, `submit`, i `reset`.

En aquest cas el comportament per defecte de l'element és enviar el formulari, igual que fa el tipus `submit`; tot i que es pot especificar el tipus mitjançant les seves propietats, que són les següents:

En cas de fer servir l'element `button` fora de l'estructura de formulari, el comportament per defecte és ignorat.

- **name**: com en altres elements ja vistos, és el nom del paràmetre que s'enviarà amb el formulari.
- **id**: identificador únic habitual de tots els elements HTML.
- **type**: els botons poden ser de tres tipus: `submit`, `reset` i `button`, amb el comportament equivalent als mateixos tipus de l'element `input`.

- **value**: valor del botó que s'afegirà com a valor en enviar el formulari.

Fixeu-vos en aquest exemple: en fer clic al botó, com que no s'ha especificat cap tipus, es realitza l'enviament del formulari i, com podeu veure a la pestanya *Network* de les eines de desenvolupador, s'ha afegit el paràmetre `esborrar` amb el valor `true`:

```
<form method="GET" action="test.php">
  <button name="esborrar" value="true">Esborrar</button>
</form>
```

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

> En navegadors antics, **el funcionament de l'element `button` no sempre és correcte**, per aquesta raó és preferible fer servir l'element `input` amb un tipus de botó.

2.2.6 Element 'label'

La finalitat d'aquest element és purament semàntica, ja que no afecta l'enviament de les dades. Serveix per indicar clarament que el text que inclou acompanya un dels elements del formulari.

Aquest element es pot aplicar de dues formes:

- Afegint l'element `input` entre les etiquetes d'apertura i tancament:

```
<label>Etiqueta:
  <input type="text" />
</label>
```

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

- Fent servir l'atribut `for` i assignar com a valor l'`id` del camp pel qual serveix d'etiqueta:

```
<label for="camp1">Etiqueta:</label>
<input id="camp1" type="text" />
```

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

2.2.7 Element 'datalist'

Aquest element es va afegir amb HTML5 i serveix per definir una llista de valors que després pot ser utilitzada per l'element `input` especificant el nom de la llista a l'atribut `list`.

A diferència de les llistes generades amb l'element `select`, aquestes permeten introduir valors propis, i el valor no queda limitat als valors del `datalist`. Vegem-ne el funcionament:

```
<label>Selecciona un navegador de la llista:
  <input list="navegadors" name="meuNavegador" />
</label>
<datalist id="navegadors">
  <option value="Chrome">
  <option value="Firefox">
  <option value="Internet Explorer">
  <option value="Opera">
  <option value="Safari">
</datalist>
```

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

2.3 Modificació d'aparença i comportament

A l'hora de crear un formulari, s'ha de tenir en compte que molt rarament n'hi haurà prou amb l'aparença original. Aquests formularis estaran incrustats en una pàgina o aplicació que tindrà un estil ja definit per l'empresa i, per tant, el formulari haurà de fer servir els mateixos colors i estils.

D'altra banda, també serà habitual fer algun tipus de personalització del comportament, ja sigui fer una validació de dades abans d'enviar el formulari, fer alguns càlculs amb la informació que serà enviada o simplement actualitzar el valor d'alguns camps a partir dels canvis produïts en uns altres.

2.3.1 Modificació de l'aparença d'un formulari

Un formulari molt rarament es trobarà de forma independent, sempre serà part d'un altre lloc web o aplicació i, per aquest motiu, la seva aparença ha de concordar amb la resta de l'aplicació.

Aquesta concordança d'estil la farem fent servir CSS, raó per la qual segurament ja tindrem la major part de la feina feta.

És important recordar que, per agrupar els elements del formulari, en lloc de fer servir l'element `div` i etiquetar fent servir text directament o capçaleres, disposem d'elements com ara:

- **fieldset**: per agrupar els continguts en lloc de fer servir *divs*.
- **label**: per etiquetar els camps.
- **legend**: per fer servir com a capçalera dins d'un `fieldset`.
- **placeholder**: propietat per donar pistes als usuaris de com s'ha d'emplenar el formulari.

S'ha de tenir compte que si no mantenim amb un estil idèndtic els elements `input` de tipus botó i els elements `button`, l'aparença pot resultar incoherent.

Malauradament, alguns dels elements que es fan servir per als formularis, com els botons i el tipus de camp `file`, no es poden personalitzar, ja que el CSS no s'aplica correctament. La seva implementació depèn del navegador i, si volem poder modificar-ne l'aparença, hem de recórrer a fer servir una biblioteca externa o hem de crear el nostre propi component.

Fulls d'estil CSS

Podeu trobar més informació sobre els fulls d'estil CSS en aquest enllaç: www.developer.mozilla.org/es/docs/Web/CSS.

En l'exemple següent, podeu veure com canvia l'aspecte d'un formulari quan s'hi aplica l'estil CSS. Primer, vegem el formulari sense aplicar-hi cap estil:

```
<form>
  <fieldset>
    <legend>Connectar</legend>
    <label>Usuari
      <input type="text" name="usuari" />
    </label>
    <label>Contrasenya
      <input type="password" name="password" />
    </label>

    <button>Connectar</button>
</form>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/YwoPLR?editors=1000; i el resultat a la figura 2.5.

Figura 2.5. Formulari sense estils CSS

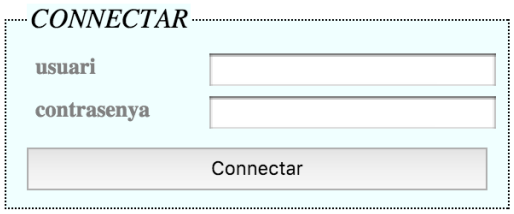
I ara, vegem el mateix formulari, però afegint-hi el següent codi CSS:

```
<style>
  form {
    margin: 20px auto;
    width: 300px;
  }

  label {
    text-transform: lowercase;
    font-weight: bold;
    font-size: small;
    color: grey;
    width: 100%;
    display: block;
```

```
    padding: 5px;
  }

  input {
    width: 60%;
    float: right;
  }

  button {
    margin-top: 10px;
    width: 100%;
    height: 25px;
  }

  legend {
    text-transform: uppercase;
    font-style: italic;
    background: azure;
  }

  fieldset {
    background: azure;
    border: 1px dotted ;
  }
</style>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/yedyyy?editors=1100; i el resultat a la figura 2.7.

FIGURA 2.6. Formulari amb estils CSS



2.3.2 Modificació del comportament d'un formulari

Quan treballem amb formularis i amb els elements relacionats (com són les llistes, els quadres de text i els botons), no sempre voldrem fer un enviament d'aquesta informació i, en molts casos, voldrem fer un tractament d'aquest abans d'enviar-la.

Potser hem de fer alguns càlculs amb la informació, formatar-la o validar-la abans de fer l'enviament. En qualsevol d'aquests casos, el que ens interessa és interrompre l'execució de l'enviament, comprovar o processar les dades i, si tot és correcte, continuar amb l'enviament.

A continuació podeu trobar un exemple en el qual es comprova si el nombre introduït és parell en enviar el formulari i, si no ho és, s'interromp l'enviament:

```
<script>
  let principal = document.getElementById('principal'),
    nombre = document.getElementById('nombre')

  principal.addEventListener('submit', event => {
```

```
    if (nombre.value % 2 === 0) {

      alert("Correcte, es continua amb l'enviament");

  } else {

      alert("El nombre ha de ser parell");
      event.preventDefault();

    }
  });
</script>
<form action="test.php" method="GET" id="principal">
  <label>Introdueix un numero parell:<input type="number" name="nombre" id ="
      nombre" required/>
  </label>
  <input type="submit" />
</form>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/GobJzN?editors=1010.

Si us fixeu en el panell *Network* de les eines de desenvolupador, podreu comprovar que, en efecte, si el nombre no és parell, no es produeix l'enviament, perquè `event.preventDefault()` evita el comportament per defecte de l'*event*, que en aquest cas és l'enviament.

D'aquesta manera podem evitar que es produeixi l'enviament del formulari fins que es compleixin totes les condicions necessàries.

> Cal recordar que en el cas dels elements `form`, `preventDefault()` **és un mètode clau si estem treballant amb AJAX**, perquè ens permet aturar l'enviament de la informació i invocar, en el seu lloc, les funcions que ens permeten fer la petició asíncrona.

Un altre exemple d'ús interessant és preprocessar la informació que s'enviarà al servidor; vegem-ho:

```
<script>
  let principal = document.getElementById('principal'),
    nom = document.getElementById('nom'),
    cognom = document.getElementById('cognom'),
    nomcomplet = document.getElementById('nomcomplet')

  principal.addEventListener('submit', event => {
    nomcomplet.value = (nom.value + ' ' + cognom.value).toLowerCase();
  });
</script>
<form action="test.php" method="GET" id="principal">
  <label>Nom:<input type="text" name="nom" id="nom" required/>
  </label>
  <label>Cognom:<input type="text" name="cognom" id ="cognom" required/>
  </label>
  <input type="hidden" name="nomcomplet" id="nomcomplet"/>
  <input type="submit" />
</form>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/jWjPjx?editors=1010.

En aquest cas hem afegit un camp ocult que originalment no conté res; però, abans d'executar l'enviament, es processa la informació del `nom` i el `cognom` per crear el `nomcomplet`, que està compost per la concatenació del `nom` i el `cognom` amb totes les lletres en minúscula.

No obstant això, fer servir els botons de tipus `submit` no és l'única manera de provocar l'enviament de dades; també es pot fer manualment, cridant el mètode `submit` del formulari. Per exemple, en fer clic sobre un element personalitzat per emular el funcionament d'un botó:

```
<style>
  #botoFals {
    display: inline;
    border: 2px solid black;
    border-radius: 15px;
    padding: 5px;
    font-style: italic;
    background: azure;
  }
</style>
<script>
  let principal = document.getElementById('principal'),
      boto = document.getElementById('botoFals');

  boto.addEventListener('click', event => principal.submit());

</script>
<form action="test.php" method="GET" id="principal">
  <input type="text" name="nom" />
  <div id="botoFals">
    Enviar!
  </div>
</form>
```

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

Com es pot veure, hem afegit un *listener* a l'element `div` per detectar quan es clica, i una vegada es dispara l'*event* `click` fem l'enviament del formulari cridant el mètode `submit`.

2.4 Validació

Un dels primers usos que es va donar a JavaScript va ser el de validar la informació del navegador abans d'enviar els formularis. Imagineu els problemes que suposava enviar un formulari amb informació errònia o incompleta el 1995, amb una velocitat de connexió de 28,8 kbit/s com a màxim i pagant pel temps de connexió. Aquest formulari s'havia d'enviar al servidor; allà es comprovava si era correcte i, en cas contrari, es retornava al client (el navegador de l'usuari) que havia de corregir-lo i tornar-lo a enviar fins que finalment passava la validació.

Cal remarcar que no es pot confiar la validació de les dades al client, les dades **també han de ser validades al servidor**. Tot i així, el fet de poder validar-les abans de fer l'enviament, ens estalvia temps i consum de dades; temps, perquè

no s'ha de comunicar amb el servidor per realitzar la validació; i consum de transferència en els plans de dades, si ens connectem mitjançant una xarxa de dades com 4G.

> **Les dades sempre s'han de validar i sanejar al servidor**, ja que no podem comptar que la informació que ens arribi sigui sempre correcta. Sense aquests controls, seria molt fàcil que un usuari maliciós pogués accedir-hi i manipular o corrompre les dades del servidor.

2.4.1 Validació d'HTML5

Amb HTML5 es van afegir moltes millores als elements que formen els formularis. Gairebé tots els elements inclouen propietats que permeten fer validacions simples sobre les dades entrades, sense necessitat de codificar res.

> A les especificacions anteriors d'HTML **no s'incloïa la capacitat de validar les dades automàticament**, l'única limitació que es podia afegir era la llargària d'una entrada de text. Aquest fet obligava a fer servir JavaScript per realitzar qualsevol tipus de validacions.

Vegem una llista d'exemples de com aplicar aquestes propietats per afegir la validació als nostres formularis:

- Fer que un camp sigui obligatori; hem d'afegir-hi la propietat `required`:

```
<form action="test.php" method="GET">
  <input type="text" placeholder="Aquest camp no pot ser buit" size="50"
      required>
  <button>Enviar</button>
</form>
```

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

- Afegir una validació de correu electrònic, el tipus ha de ser `email`; vegem-ho:

```
<form action="test.php" method="GET">
  <input type="email" placeholder="Aquest camp ha de ser un email correcte"
      size="50" required>
  <button>Enviar</button>
</form>
```

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

- Acceptar només nombres enters:

```
<form action="test.php" method="GET">
  <input type="number" placeholder="Només nombres enters" required>
  <button>Enviar</button>
</form>
```

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

- Acceptar també nombres reals:
  - Només haurem d'afegir l'atribut `step` amb el valor que representarà la diferència entre un nombre i el següent. Per exemple, si volem només una precisió d'un decimal, hi afegirem `step="0.1"`:

```
<form action="test.php" method="GET">
  <input type="number" placeholder="Accepta nombres reals" required step="0.1">
  <button>Enviar</button>
</form>
```

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

- Establir un valor mínim i màxim (contingut numèric):
  - Continuant amb els exemples numèrics, ens trobem que en molts casos necessitarem establir un valor mínim i un de màxim. Això també es pot fer mitjançant les propietats de l'element `input`, en aquest cas amb `max` i `min`:

```
<form action="test.php" method="GET">
  <input type="number" placeholder="Acecepta nombres reals" required step="0.1"
       min="1" max="42">
  <button>Enviar</button>
</form>
```

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

- Establir un valor mínim i màxim (cadenes de text):
  - Seria possible aplicar els valors `max` i `min` a un camp de text? Per exemple, per requerir que una contrasenya tingui com a mínim 5 caràcters? Vegem-ho:

```
<form action="test.php" method="GET">
  <input type="password" placeholder="Entre 5 i 10 caracters" required min="5"
      max="10">
  <button>Enviar</button>
</form>
```

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

- No funciona, els valors de `min` i `max` només s'apliquen a continguts numèrics. Si volem treballar amb cadenes de text, hem de fer servir les propietats `minlength` i `maxlength`:

```
<form action="test.php" method="GET">
  <input type="password" placeholder="Entre 5 i 10 caracters" required
      minlength="5" maxlength="10">
  <button>Enviar</button>
</form>
```

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

Validació per 'pattern'

La propietat de validació més potent és `pattern`, introduïda a HTML5 i, per consegüent, no disponible en els navegadors més antics. Aquesta propietat ens permet introduir una expressió regular que es farà servir per validar el contingut del camp. Per exemple, si l'entrada de text ha de contenir només una paraula i ha de començar per majúscula, podríem fer-ho així:

Les *expressions regulars* es troben detallades a la secció "Expressions regulars" d'aquest mateix apartat.

```
<form action="test.php" method="GET">
  <input type="text" placeholder="Només una paraula que comenci per majúscula"
      required size=50 pattern='^\b[A-Z]\w*\b$'>
  <button>Enviar</button>
</form>
```

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

Si proveu aquest exemple, veureu que només us deixa enviar el formulari si es compleix la condició indicada per l'expressió regular, que es pot interpretar de la manera següent:

- Començant al principi del text: `^`
- Fins al final del text: `$`
- Ha de contenir només una paraula, que és envoltada per `\b` que marca l'inici i el final
- El primer caràcter ha d'estar comprès entre A i Z, ambdós inclosos: `[A-Z]`
- A continuació pot haver-hi qualsevol quantitat (0 o més) de lletres i números: `\w*`

2.4.2 Validació per a JavaScript

Tot i que HTML5 ens permet validar els camps en una gran quantitat de situacions, pot ocórrer que necessitem programar les nostres pròpies validacions; per exemple, en el cas d'haver de donar suport a navegadors antics o quan tractem amb un tipus de validació més complexa.

Detectant quan es dispara l'*event* `submit` de l'element `form`, és possible aturar l'enviament d'un formulari i afegir les nostres pròpies validacions . Vegem una variació per simplificar el codi, afegint directament el nom de la funció que serà invocada en detectar-se aquest *event*:

```
<form action="test.php" method="GET" onsubmit="return validar();">
  <input name="dni" id="dni" type="text" minlength="9" maxlength="9" size="9"
      placeholder="DNI" pattern="^\d{8}[a-zA-Z]"/>
  <button>Enviar</button>
</form>

<script>
  function validar() {
    let dni = document.getElementById('dni');
    return comprovarDni(dni.value);
  }

  function comprovarDni(dni) {
    let nombre,
      lletraIntroduida,
      lletraEsperada,
      correcte = false;

    nombre = dni.substr(0, dni.length - 1); // Extraiem el nombre
    lletraIntroduida = dni.substr(dni.length - 1, 1); // Extreiem la lletra
    nombre = nombre % 23;
    lletraEsperada = 'TRWAGMYFPDXBNJZSQVHLCKET'; // Possibles valors de la
        lletra ordenats per la posició corresponent
    lletraEsperada = lletraEsperada.substring(nombre, nombre + 1);

    if (lletraEsperada === lletraIntroduida.toUpperCase()) {
      correcte = true;
      alert('DNI correcte: ' + dni);
    } else {
      alert('DNI incorrecte: ' + dni);
    }

    return correcte;
  }
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/XXLKGX?editors=1010.

Com es pot apreciar, a l'element `input` del bloc HTML s'ha especificat:

- La mida mínima ha de ser 9 caràcters (`minlength`).
- La mida màxima ha de ser 9 caràcters (`maxlength`).
- El valor s'ha de correspondre amb una expressió regular (`pattern`) que requereix 8 nombres `\d{8}` seguits d'una lletra en minúscules o majúscules `[a-zA-Z]`.

Com que la lletra corresponent a cada DNI es pot calcular en funció dels nombres, és possible comprovar que el DNI introduït sigui correcte abans d'enviar el formulari. Per afegir aquesta comprovació només cal afegir el nom de la funció encarregada de la validació a la detecció de l'*event* `submit` del formulari perquè aquesta funció sigui invocada quan el formulari sigui enviat.

Fixeu-vos que en lloc de simplement invocar la funció `validar` en disparar-se l'*event* `submit`, el que hem fet és retornar el valor d'aquesta funció: `onsumbit="return validar();"`. Si aquest valor és `true`, el formulari s'enviarà, i si és `false`, s'aturarà l'enviament. Aquesta és una alternativa a invocar el mètode `preventDefault()` per aturar el comportament per defecte.

Validació del DNI

Podeu trobar més informació sobre com es calcula la lletra del DNI en el següent enllaç: www.goo.gl/VrlzYQ.

Fer la validació de la lletra és més complicat, així que per fer-ho hem de recórrer a JavaScript. En aquest exemple hem invocat la funció `validar()`, cosa que ens permetria centralitzar la validació i afegir més comprovacions, si fos necessari. Només s'ha de vigilar que ha de retornar `true`, si es passen totes les comprovacions, o a `false`, si alguna falla.

Fixeu-vos que la funció `validar()` retorna el valor de la funció `comprovarDni()`, ja que en aquest cas només ha de fer una comprovació. El procés a seguir per determinar si s'ha de retornar cert o fals és el següent:

1. Separem la part de la cadena que correspon als nombres i la que correspon a la lletra.
2. Calculem quina és la lletra que hi hauria de correspondre.
3. Si la lletra introduïda correspon amb l'esperada, llavors és correcta. Fixeu-vos que fem la comprovació del valor en majúscules, així és indiferent si l'hem introduït d'una manera o de l'altra, ja que la lletra esperada és majúscula.
4. Si és correcte, retornarem `true`; si no, retornarem `false`.

Un altre cas que us podeu trobar és haver de validar la mida màxima d'un o més fitxers abans de fer l'enviament al servidor. Aquest cas és molt interessant perquè, en cas contrari, no sabríeu si s'ha superat la mida màxima fins que falli el servidor, cosa que pot provocar llargues esperes abans de conèixer l'error. Vegem-ho:

Web API File

La Web API que permet obtenir la informació dels fitxers pertany a HTML5. Podeu trobar-ne més informació a l'enllaç següent: www.goo.gl/7dtSKT.

```
<script>
  let fitxers = document.getElementById('fitxers'),
    MAX_MIDA_FITXERS = 2048; // Mida màxima en bytes

  function validarFitxers() {
    let fitxer, midaTotal;

    if (fitxers.files.length > 0) {

      midaTotal = 0;

      for (let i = 0; i < fitxers.files.length; i++) {
        fitxer = fitxers.files[i];
        console.log("Fitxer: " + fitxer.name + " te una mida de " + fitxer.size
             + " bytes");
        midaTotal += fitxer.size;
      }
```

```
17
18      console.log("Mida total dels fitxers: ", midaTotal);
19
20      if (midaTotal > MAX_MIDA_FITXERS) {
21        fitxers.value = [];
22        alert("La mida màxima dels fitxers és " + MAX_MIDA_FITXERS +
23          " bytes, però els fitxers seleccionats tenen una mida de " +
24          midaTotal + " bytes.");
25      }
26    }
27  };
28 </script>
29 <form action="test.php" method="POST" enctype="multipart/form-data">
30   <input type="file" id="fitxers" name="fitxer[]" multiple="true" onchange="
         validarFitxers();">
31   <button>Enviar</button>
32 </form>
```

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

`MAX_MIDA_FITXERS` és una variable definida per nosaltres que ens facilita ajustar la mida segons els requeriments, en lloc de fer servir un nombre. La posem en majúscules perquè la seva funcionalitat coincideix amb la de les constants d'altres llenguatges.

En aquest exemple, en lloc de fer la validació en enviar el formulari, l'hem afegit a l'*event* `change` de l'element *input*. En el cas dels fitxers, l'*event* `change` es produeix sempre tan bon punt es tanca el diàleg de selecció de fitxers, així que és el punt ideal per fer la validació. Tampoc no cal retornar cap valor; si la mida no és correcta, el que fem és buidar la selecció establint l'*array* de fitxers a un *array* buit.

Com que és un exemple per a HTML5 es pot fer servir l'atribut `multiple` de l'element *input*, cosa que ens permet fer una selecció múltiple de fitxers. Fixeu-vos que en aquest cas l'atribut `name` és un *array*: això permet enviar tota la llista de fitxers seleccionats en lloc d'un de sol.

En el codi de la funció `validarFitxers()`, veureu que primer hem afegit una sèrie de missatges que es mostren per consola per facilitar el seguiment de què està passant. Aquests missatges, però, no són necessaris per al bon funcionament del programa. El procés seria el següent:

1. El primer pas és comprovar si la mida de l'*array* de fitxers `file` (exposat per la Web API de fitxers) conté algun element.
2. Si és així, inicialitzem la mida total a 0 i recorrem l'*array*.
3. Mostrem la informació de cada fitxer per la consola i afegim la mida al total mitjançant la seva propietat `size`.
4. Una vegada els hem recorregut tots, mostrem per la consola la mida total.
5. Procedim a comprovar si la mida és superior al límit (en el nostre cas, 2.048 bytes) i, si és així, buidem el contingut de l'element `input` que conté els fitxers, i mostrem l'alerta.

Les bases per realitzar validacions directament amb JavaScript són molt simples. Això sí, s'ha de tenir sempre a mà una bona pàgina de consulta com www.developer.mozilla.org, sobretot quan treballem amb les Web API, ja que poden oferir-nos possibilitats que van més enllà de l'especificació bàsica d'HTML.

2.5 Expressions regulars

Es pot fer servir l'atribut `pattern` dels elements `input` per realitzar validacions, aplicant una expressió regular. En aquesta secció veurem què són exactament les expressions regulars i com s'ha d'utilitzar-les.

Comprovador d'expressions regulars en línia

En molts casos pot ser interessant comprovar que l'expressió regular que volem aplicar és correcta. Per comprovar-ho existeixen eines en línia com *Regex Pal* (www.regexpal.com). Aquest web permet introduir una expressió regular i un text, de manera que podem comprovar si la selecció es correspon amb el que s'esperava.

L'ús de l'atribut `pattern` dels elements `input` es tracta a la secció "Validació d'HTML5" d'aquest apartat.

Una expressió regular és una seqüència de caràcters que defineixen un patró de cerca. Aquest patró es pot utilitzar en operacions del tipus *cercar i reemplaçar*, oferint inclús capacitat de captura de grups per realitzar uns reemplaçaments molt potents.

Expressions regulars per a JavaScript

Es pot trobar més informació sobre totes les opcions que ofereixen les expressions regulars per a JavaScript a l'enllaç següent: www.goo.gl/p3Ejc3.

Es poden trobar processadors d'expressions regulars tant en eines dels sistemes operatius Unix i Linux, com en editors i processadors de textos. A banda de les aplicacions externes, pràcticament tots els llenguatges de programació ofereixen la possibilitat d'utilitzar-les. Així que encara que a primera vista siguin inintel·ligibles, dominar-les ens pot facilitar molt la manipulació de cadenes de text, sigui quin sigui el llenguatge amb el qual treballem.

Exemple d'usos de les expressions regulars

A banda de validar dades, a les expressions regulars se'ls poden donar molts altres usos, per exemple, els següents:

- Normalitzar textos: eliminar espais en blanc duplicats, salts de línia erronis (la següent línia comença en minúscula), etc.
- Canviar els formats: substituir cometes normals per tipogràfiques, que requereixen conèixer on comença i on acaba l'element.
- Convertir un text d'un tipus en un altre: extreure la informació d'una consulta SQL i convertir-la en una cadena de text en format JSON.
- Realitzar cerques de fitxers amb patrons concrets (utilitats del sistema operatiu Unix/Linux)
- Creació de *parsers* per extreure dades de fitxers amb formats coneguts de text pla (XML, JSON...) i convertir-les en objectes per poder manipular-los.
- Creació d'intèrprets de nous llenguatges de programació.
- Creació de plantilles on se substituiran valors clau pels valors de les variables que ens interessin.

En el següent exemple podeu veure com es fa servir un patró molt simple per eliminar tots els salts de línia i espais de sobra al text original, i reemplaçar-los per un sol espai:

```
1 <script>
2   let original = document.getElementById('original'),
3     net = document.getElementById('net');
```

```
4
5   function netejar() {
6     let text = original.innerHTML,
7       patro = /\s+/g;
8     net.innerHTML = text.replace(patro, ' ');
9   }
10 </script>
11 <pre id="original">»E  en aquell
12 punt
13 se
14 féu  desliguar
15 la çimolsa, que   no  la    volgué més   portar,   ab molta   malenconia que
          li restà, emperò no·n
16 féu demostració alguna.
17 </pre>
18 <button onclick="netejar()">Netejar</button>
19 <pre id="net"></pre>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/qbzrQY?editors=1010.

> Tot i que la majoria dels *metacaràcters* es poden trobar en totes les implementacions, en cas de dubte, és recomanable fer servir una eina com ***RegexPal*** (www.regexpal.com) per comprovar que la nostra expressió funciona com esperem, o bé consultar la documentació especial. Per exemple, no tots els processadors d'expressions regulars ofereixen la capacitat de fer ***backtracing***.

Un *metacaràcter* és un caràcter o conjunt de caràcters que tenen un significat especial a l'expressió regular; per exemple ^ que coincideix la primera posició del text o . que coincideix amb qualsevol caràcter.

En aquests materials ens centrarem en la implementació de les expressions regulars per a JavaScript, tot i que no farem una anàlisi en profunditat de totes les seves opcions.

En JavaScript es poden crear expressions regulars com a representació literal:

```
1 let patro = /ioc/;
```

Les barres inicial i final d'una expressió regular no formen part del patró sino que són els delimitadors de l'expressió regular.

També es poden crear expressions regulars com a objecte `RegExp`, fent servir el constructor i passant l'expressió com a cadena de text:

```
1 let patro = new RegExp("ioc");
```

> Es prefereix **l'ús de la creació literal** quan coneixem el patró a priori; mentre que **l'ús de l'objecte** és el mètode preferit quan l'expressió és desconeguda inicialment, per exemple, quan es crea a partir de les dades entrades per l'usuari.

Totes dues formes admeten opcions addicionals (*flags*), que s'afegeixen de la següent manera en la representació literal:

```
1 let patro = /ioc/flags;
```

I d'aquesta si fem servir el objecte:

```
let patro = new RegExp("ioc", "flags");
```

Els ***flags*** formen part de l'expressió regular, i no poden afegir-se ni eliminar-se després de la seva creació. A JavaScript es poden utilitzar els següents:

- **g**: *Global*, cerca tots els resultats a la cadena de text; si no s'afegeix, es para la cerca quan troba la primera coincidència.
- **i**: *Case-insensitive*, no distingeix entre majúscules i minúscules.
- **m**: *Multi-line*, el metacaràcter de final del text coincideix amb el de salt de línia, per exemple, el següent patró coincideix amb el punt i a part de tots els paràgrafs d'un text: `/\.$/gm`

El caràcter \ és u caràcter d'escapament; això vol dir que altera la interpretació del següent caràcter:

- Si es tracta d'un caràcter amb un significat especial, com `.`, es tractarà com un caràcter normal.
- Si es tracta d'un caràcter normal, i és interpretable com a especial com `b`, s'interpretarà segons la seva funció especial, en aquest cas com el delimitador de paraula `\b`.

Si no es fa servir cap metacaràcter de tipus quantificador, cada caràcter del patró només se cerca una vegada. Per exemple, `Adéu` valida *Adéu*, però no *AAAdéu*, el patró correcte per al segon cas seria `A+deu`.

Podeu trobar els metacaràcters més utilitzats a la taula 2.1.

TAULA 2.1. Taula de metacaràcters per a expressions regulars

Metac.	Descripció	Patró exemple	Vàlid	No vàlid
`^`	Coincideix només amb el començament de la cadena	`^a`	**a**vió	campana
`$`	Coincideix amb el final de la cadena	`a$`	Samarret**a**	avió
`*`	Cerca el caràcter o grup precedent 0 o més vegades, equival a {0,}	`c*`	**c**amió	avió
`+`	Cerca el caràcter o grup precedent 1 o més vegades, equival a {1,}	`c+`	cole**cc**ió	camió
`?`	Aquest caràcter pot tenir diferents significats segons on es trobi. En situacions normals cerca el caràcter precedent 0 o 1 vegada, i equival a {0,1}	`bici?`	**bici**, **bic**	col
`.`	Coincideix amb tots els caràcters excepte salt de línia	`bi.i`	**bici**, **bili**	billi
`{n}`	Repeteix la cerca del caràcter o grup precedent exactament `n` vegades	`o{2}`	**oo**, **oo**o	o
`{n,m}`	Repeteix la cerca del caràcter o grup precedent entre `n` i `m` vegades	`o{2,4}`	oo, ooo, oooo, **oooo**o	o

TAULA 2.1 (continuació)

Metac.	Descripció	Patró exemple	Vàlid	No vàlid
`( i )`	Obren i tanquen un grup, el fragment contingut dins dels parèntesis es tracta com un bloc i pot ser capturat.	`o(na){2}`	**onana**, **onana**na, **onanaonana**	ona
`[ i ]`	Creen un joc de caràcters, el valor d'aquesta posició pot ser qualsevol dels que en formen part. Dins del joc de caràcters el guió `-` serveix per indicar un rang de caràcters, per exemple `[A-Z]` forma un joc de caràcters que inclou totes les lletres des de la A majúscula fins a la Z majúscula	`coordenada[XYZ]`	**coordenadaX**, **coordenadaY**, **coordenadaZ**	coordenadaB
`[ i ]` començant per `^`	Si el joc de caràcters comença per `^` nega el joc de caràcters	`coordenada[^XYZ]`	**coordenadaB**	coordenadaY, coordenadaZ
\b	Coincideix amb el principi o final de la paraula	`\bBarcelona`	Som a **Barcelona**	AutoBacerlona
\d	Coincideix amb un digit, equival a `[0-9]`	\d\d\d	**123**, **123**4	12
\D	L'invers a \d, equival a `[^0-9]`	\D\D\D	**abc**	123
\w	Coincideix amb un caràcter alfanumèric anglès, inclosa la barra baixa. Equival a `[A-Za-z-0-9_]` **no inclou caràcters d'accentuació, ni propis d'altres llengües com la ç**	\w+	**pala_72**	.?!
\W	L'invers a \w, equival a `[^A-Za-z-0-9_]`	\W	42**%**	pala_72
\s	Coincideix amb els caràcters d'espai en blanc, incloent-hi tabulacions i salts de línia	`\w\s\w`	sens**e s**ortida	-
\S	L'invers a \s, coincideix amb tot el que no siguin espais en blanc	`\w\S\w`	**cot**xe	c o t x e

En negreta es mostra la coincidència del patr; cal fixar-se que sovint només és parcial.

2.5.1 Grups de captura

L'ús de parèntesis ens serveix per agrupar seleccions, que poden ser extretes o reemplaçades, dins del patró. A aquests grups se'ls coneix com a **grups de captura**.

Vegem-ne un exemple fent servir grups de captura. Cerquem totes les ocurrències de `lo` i afegirem les etiquetes `<b>` i `</b>` a cadascuna per ressaltar el canvi:

```
<script>
  let original = document.getElementById('original'),
    nou = document.getElementById('nou');

  function canviar() {
    let text = original.innerHTML,
```

```
      patro = /(\b[lL]o\b)+/g;
    nou.innerHTML = text.replace(patro, '<b>$1</b>');
  }
</script>
<p id="original">»Aprés diré a la senyoria vostra les cerimònies que s'an de
    fer en la capella. Ara diré los cavallers qui foren elets. Primerament, lo
     rey elegí XXV cavallers e, ab lo rey foren XXVI. Lo rey fon lo primer qui
     jurà de servar totes les ordinacions en los capítols contengudes e que no
     fos cavaller negú qui demanàs aquest orde que·l pogués haver. Tirant fon
    elet lo primer de tots los altres cavallers, per ço com fon lo millor de
    tots los cavallers. Aprés fon elet lo príncep de Gales, lo duch de
    Betafort, lo duch de Lencastre, lo duch d'Atçètera, lo marquès de Sófolch,
     lo marqués de Sant Jordi, lo marqués de Belpuig, Johan de Varoych, gran
    conestable, lo comte de Nortabar, lo comte de Salasberi, lo comte d'
    Estafort, lo comte de Vilamur, lo comte de les Marches Negres, lo comte de
     la Joyosa Guarda, lo senyor d'Escala Rompuda, lo senyor de Puigvert, lo
    senyor de Terranova, miçer Johan Stuart, miçer Albert de Riuçech. Aquests
    foren del regne. Los strangers foren: lo duch de Berrí, lo duch d'Anjou,
    lo comte de Flandes. Foren tots en nombre de XXVI cavallers.
</p>
<button onclick="canviar()">Canviar</button>
<p id="nou"></p>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/jWjmYL?editors=1010.

Com es pot apreciar, en el segon paràmetre de la invocació a `replace`, l'expressió `$1` indica que en aquest lloc anirà la primera -i en aquest cas única- cadena capturada. Si capturéssim una segona cadena, podríem referir-nos-hi amb `$2`, a una tercera amb `$3`, etc.

Fixeu-vos que el patró utilitzat ha estat `(\b[lL]o\b)+`; això és així pels motius següents:

- Volem capturar els grups que continguin el patró envoltat pels parèntesis: `(` i `)`.
- Només volem fer el reemplaçament si n'hi ha 1 o més: `+`.
- Ha de ser la paraula completa, no ho volem substituir en els casos què formi part d'una altra paraula, com pot ser **los** o **millor**, així que ho hem delimitat al principi i final de les paraules amb `\b`.
- Volem fer el canvi tant en el cas que la `l` sigui majúscula com si no, per això afegim la selecció :`[lL]`.

A partir d'aquest codi, només canviant el patró, podem veure com és molt fàcil afegir més elements a la selecció; per exemple, si volem que la substitució afecti "la", "les", "lo" i "los", només hem de tenir en compte els elements comuns i les variacions; el patró seria el següent: `(\b[Ll][aoe]s?\b)`. Per tant:

- Hem afegit un joc de selecció de caràcters nou: `[aoe]`.
- Hem afegit una `s` opcional: `s?`.

A continuació podeu trobar un exemple que cerca totes les coincidències de "la", "las", "le", "les", "lo" i "los" tant amb "l" minúscula com majúscula i les

reemplaça per la mateixa coincidència però afegint les etiquetes per mostrar-les en negreta (element `<b>`). És a dir, si es troba la coincidència "lo" ho substituirà per `<b>lo</b>`. Vegem-ho:

```
<p id="original">»Aprés diré a la senyoria vostra les cerimònies que s'an de
    fer en la capella. Ara diré los cavallers qui foren elets. Primerament, lo
     rey elegí XXV cavallers e, ab lo rey foren XXVI. Lo rey fon lo primer qui
     jurà de servar totes les ordinacions en los capítols contengudes e que no
     fos cavaller negú qui demanàs aquest orde que·l pogués haver. Tirant fon
    elet lo primer de tots los altres cavallers, per ço com fon lo millor de
    tots los cavallers. Aprés fon elet lo príncep de Gales, lo duch de
    Betafort, lo duch de Lencastre, lo duch d'Atcètera, lo marquès de Sófolch,
     lo marquès de Sant Jordi, lo marquès de Belpuig, Johan de Varoych, gran
    conestable, lo comte de Nortabar, lo comte de Salasberi, lo comte d'
    Estafort, lo comte de Vilamur, lo comte de les Marches Negres, lo comte de
     la Joyosa Guarda, lo senyor d'Escala Rompuda, lo senyor de Puigvert, lo
    senyor de Terranova, miçer Johan Stuart, miçer Albert de Riuçech. Aquests
    foren del regne. Los strangers foren: lo duch de Berrí, lo duch d'Anjou,
    lo comte de Flandes. Foren tots en nombre de XXVI cavallers.
</p>
<button onclick="canviar()">Canviar</button>
<p id="nou"></p>

<script>
  let original = document.getElementById('original'),
    nou = document.getElementById('nou');

  function canviar() {
    let text = original.innerHTML,
      patro = /(\b[Ll][aoe]s?\b)/g;
    nou.innerHTML = text.replace(patro, '<b>$1</b>');
  }
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/YwoQKq?editors=1010.

La resta del patró és idèntic, en canvi, ara s'aplica a moltes més combinacions.

Com podeu veure, encara que a primera vista les expressions regulars poden semblar inintel·ligibles, si se'n divideixen els components, són més fàcils d'entendre. Ara bé, hi ha excepcions, com per exemple en el cas de les substitucions, que es poden fer servir per localitzar en altres idiomes diferents de l'anglès.

Captura de caràcters accentuats

El metacaràcter `\w` no captura ni la `ç` ni els caràcters accentuats, per tant, si volem seleccionar tots aquests caràcters, el patró equivalent a `\w` és: `[A-Za-z_çàèéíïóòúüÇÀÉÈÍÏÓÒÚÜ]`

2.5.2 Altres mètodes d'utilització de les expressions regulars ('String' i 'RegExp')

Cal destacar que la utilització de patrons d'expressions regulars no es troba limitada als objectes de tipus `RegExp`, sinó que són utilitzats també per altres objectes. Per exemple, els objectes de tipus `String` ofereixen dos mètodes que accepten expressions regulars com a paràmetres, el que permet fer reemplaçaments i divisions de cadenes complexes.

Vegem a continuació alguns mètodes dels objectes de tipus `String` i els objectes de tipus `RegExp` que permeten realitzar diferents operacions amb expressions regulars:

- `exec(cadena)`: mètode de `RegExp` que executa una cerca sobre el paràmetre i retorna un *array* amb la informació de la cadena trobada (posició 0) i, si n'hi ha, grups de captura (resta de posicions); si no troba la cadena, retorna `null`.
- `test(cadena)`: mètode de `RegExp` que cerca una coincidència a `cadena` i retorna `true` si la troba o `false` en cas contrari.
- `match(regexp)`: mètode de `String` amb funcions similars a `exec`.
- `search(regexp)`: mètode de `String` que cerca a la cadena de text el patró passat com argument. En cas de trobar-se alguna coincidència retorna la posició (dintre de la cadena) on comença la coincidència o el valor -1 si no s'ha trobat. En cas de trobar-se múltiples coincidències només es té en compte la primera d'aquestes i la resta són ignorades.
- `replace(regexp,cadena)`: mètode de `String` que executa la cerca en base a la expressió regular indicada al primer paràmetre i la reemplaça per la cadena de reemplaçament especificada al segon paràmetre (que pot contenir referències als grups de captura).
- **split(regexp)**: mètode de `String` que divideix una cadena en fragments i els col·loca com a *array*. L'expressió regular identifica els separadors dels diferents fragments.

Tant `replace` com `split` poden treballar directament amb cadenes, no és obligatori fer servir expressions regulars.

Vegem amb un exemple com funcionen tots aquests mètodes; obriu la consola de CodePen o les eines de desenvolupador per veure'n la informació extreta:

```
<script>
  let original = document.getElementById('original'),
    nou = document.getElementById('nou');

  function canviar() {
    let text = original.innerHTML;

    provaTest(text);
    provaSearch(text);
    provaExec(text);
    provaMatch(text);
    provaSplit(text);

    nou.innerHTML = provaReplace(text);
  }

  function provaTest(text) {
    let patro = /(cavallers)|(rey)/g;
    console.log("———RegExp.test()———");
    console.log("Es troba la paraula cavallers o rey?", patro.test(text));
  }

  function provaSearch(text) {
    let patro = /(cavallers)|(rey)/g;
    console.log("———String.search()———");
    console.log("Es troba la paraula cavaller o cavallers, i si és així en
        quina posició?", text.search(patro));
  }

  function provaExec(text) {
    let patro = /((cavallers)|(rey))+/g;
    console.log("———RegExp.exec()———");
```

```
32      console.log("Quines coincidències de la paraula cavallers o rey s'han
            trobat?", patro.exec(text));
33    }
34
35    function provaMatch(text) {
36      let patro = /((cavallers)|(rey))+/g;
37      console.log("———String.match()———");
38      console.log("Quines coincidències de la paraula cavallers o rey s'han
            trobat?", text.match(patro));
39    }
40
41    function provaSplit(text) {
42      let patro = /[:,\.]? /g;
43      console.log("Array amb tots els mots del text:", text.split(patro));
44    }
45
46    function provaReplace(text) {
47      let patro = /([A-Za-z_çàéèíïóòúüÇÀÉÈÍÏÓÒÚÜ]+)([\.:,·]?\s)([A-Za-z_çàéèíïóòú
            üÇÀÉÈÍÏÓÒÚÜ]+)/gm;
48
49      return text.replace(patro, '$3$2$1');
50    }
51  </script>
52  <p id="original">Aprés diré a la senyoria vostra les cerimònies que s'an de fer
        en la capella. Ara diré los cavallers qui foren elets. Primerament, lo
      rey elegí XXV cavallers e, ab lo rey foren XXVI. Lo rey fon lo primer qui
      jurà de servar totes les ordinacions en los capítols contengudes e que no
      fos cavaller negú qui demanàs aquest orde que·l pogués haver. Tirant fon
      elet lo primer de tots los altres cavallers, per ço com fon lo millor de
      tots los cavallers. Aprés fon elet lo príncep de Gales, lo duch de
      Betafort, lo duch de Lencastre, lo duch d'Atçètera, lo marquès de Sófolch,
       lo marqués de Sant Jordi, lo marqués de Belpuig, Johan de Varoych, gran
      conestable, lo comte de Nortabar, lo comte de Salasberi, lo comte d'
      Estafort, lo comte de Vilamur, lo comte de les Marches Negres, lo comte de
       la Joyosa Guarda, lo senyor d'Escala Rompuda, lo senyor de Puigvert, lo
      senyor de Terranova, miçer Johan Stuart, miçer Albert de Riuçech. Aquests
      foren del regne. Los strangers foren: lo duch de Berrí, lo duch d'Anjou,
      lo comte de Flandes. Foren tots en nombre de XXVI cavallers.
53  </p>
54  <button onclick="canviar()">Canviar</button>
55  <p id="nou"></p>
```

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

Fixeu-vos que fent servir el mateix patró, **el mètode `RegExp.exec()` no retorna els valors esperats**; en canvi, el mètode `String.match()` ens retorna un *array* amb tots els elements, com és d'esperar.

Com es pot apreciar, s'ha fet servir el mètode `String.replace()` amb un patró una mica més complex per capturar correctament totes les paraules. Com a segon paràmetre s'ha utilitzat la cadena de reemplaçament `$3$2$1`, de manera que es posa primer el tercer grup capturat (`$3`), a continuació el segon grup (`$2`) i finalment el primer (`$1`). Així doncs, la primera coincidència del patró serien els grups *Aprés*, i *diré*; per tant, en reemplaçar el resultat és *diré Aprés*.

2.6 Utilització de galetes i Web Storage

Com que *HTTP* és un protocol sense estat, cada vegada que es demana una pàgina al servidor, el servidor no "ens recorda". Això, com podeu imaginar, suposa un greu inconvenient, ja que no és possible, per exemple, canviar de pàgina sense perdre tota la informació, com pot ser:

- Estar autenticats a un lloc web
- La cistella de la compra en un negoci de venda per internet
- Les preferències seleccionades en visitar un lloc web

Tampoc no es pot accedir als següents serveis:

- Fer el seguiment de les pàgines visitades per generar analítiques
- Mostrar vídeos (Youtube)
- Mostrar mapes (Google Maps)
- Mostrar anuncis publicitaris
- Serveis de xat

La solució que es va donar a aquest problema va ser la invenció de les galetes que són uns petits fragments d'informació que el navegador guarda en fitxers i s'envien juntament amb la capçalera al servidor; de manera que es pot simular que el servidor ens recorda, tot que en realitat sempre es tracta cada petició com si fos nova.

Mètode alternatiu a l'ús de galetes per autenticar-se

En cas de no tenir activada l'opció d'acceptar galetes, el que es fa habitualment per mantenir sessions *obertes* al servidor és passar la informació d'autenticació com a paràmetre d'una petició GET; és a dir, es codifica al mateix *URL*.

Com que aquestes galetes poden ser llegides i escrites des de JavaScript, es poden utilitzar per a altres propòsits que no estan relacionats amb el servidor; per exemple, per guardar les preferències d'un usuari en visitar una pàgina. S'ha de tenir en compte que la mida màxima per a cada galeta és de 4.095 bytes.

Amb HTML5 es va afegir una nova Web API per emmagatzemar informació al navegador anomenada Web Storage que, a diferència de les galetes, no s'envien mai al servidor.

Així doncs, si necessitem compartir la informació guardada amb el servidor, farem servir galetes, però, si s'hi ha d'accedir només localment, es recomana fer servir Web Storage, ja que accepta fins a 5 MB per domini i s'estalvia amplada de banda perquè no s'envia aquesta informació al servidor (al contrari del que passa amb les galetes).

Si es desactiva l'ús de galetes al navegador, queden inhabilitats tant el Web Storage com les galetes.

Les galetes poden visualitzar-se des de les eines de desenvolupador dels navegadors, però la seva localització canvia segons el navegador amb el qual treballeu. A Google Chrome podeu trobar-les a la pestanya **Resources**, en el desplegable **Cookies** del panell esquerre, tal com es veu en la imatge figura 2.7. En canvi, si feu servir Mozilla Firefox, les podeu trobar a la pestanya **Emmagatzematge**, en el desplegable **Galetes** del panell esquerre, tal com es veu en la imatge figura 2.8.

FIGURA 2.7. Visualització de galetes a Google Chrome

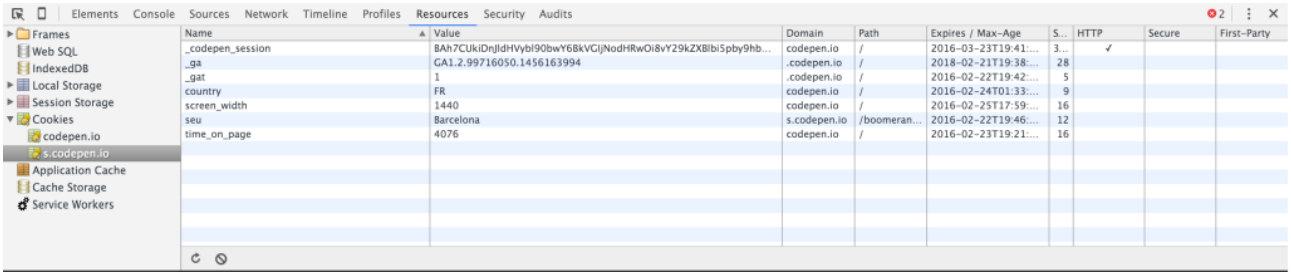
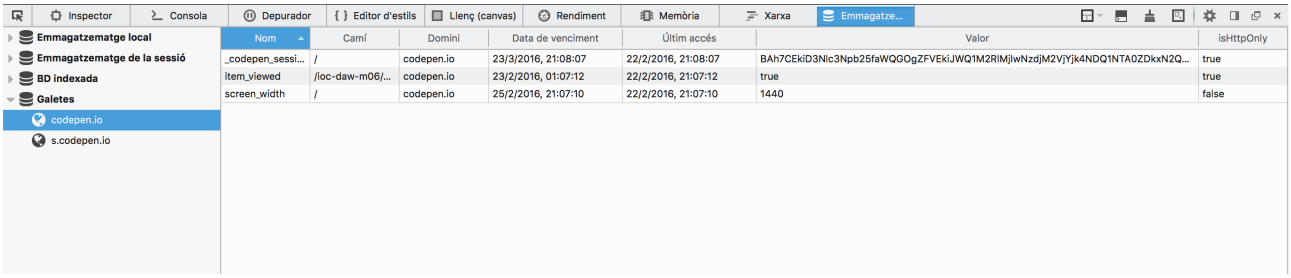
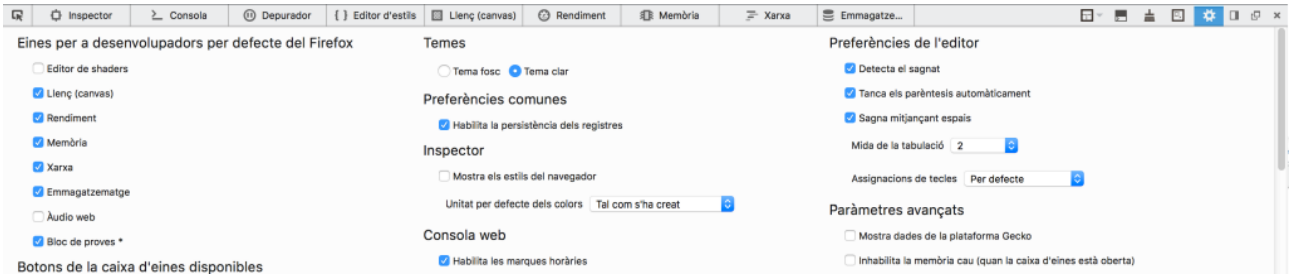


FIGURA 2.8. Visualització de galetes a Mozilla Firefox



En cas que no us aparegui aquesta pestanya a Mozilla Firefox, la podeu habilitar a les opcions de les eines de desenvolupador (símbol de la roda dentada a la dreta), marcant la casella **Emmagatzematge** (vegeu la figura 2.9).

FIGURA 2.9. Visualització d'opcions a Mozilla Firefox



La informació sobre els **Web Storage** es troba a la mateixa pestanya, però a la secció corresponent: **Local Storage** o **Session Storage**.

2.6.1 Llei de cookies

Abans d'endinsar-nos en la gestió de galetes hem de parlar de la normativa que en regula l'ús, ja que si no s'informa adequadament l'usuari que les fem servir i amb quina finalitat, podem ser sancionats. En aquests materials no tractarem el tema en

profunditat, però és important conèixer-les i saber que sempre que treballem amb galetes haurem d'aplicar aquesta llei. Recomanem la lectura de la **Guia sobre les normes d'ús de les galetes**.

Guia sobre les normes d'ús de les galetes

Podeu trobar la guia oficial de l'*Agencia española de protección de datos* amb tota la informació sobre la legislació de galetes a l'enllaç següent: www.goo.gl/akFmH.

Exemple de sanció econòmica per vulnerar la llei de galetes

En aquest enllaç podeu trobar informació sobre el primer cas de **sancions econòmiques per vulnerar la llei de galetes**: www.goo.gl/Ej7bFQ.

Si us hi fixeu, no se n'està fent cap ús maliciós, totes les galetes que es fan servir tenen una finalitat legítima. Pertanyen a components que fan funcionar la pàgina correctament, o a publicitat, o bé a serveis externs, com Google Maps, YouTube... Tot i així, com que el text amb la política de galetes no està ben definit, les empreses involucrades van ser sancionades.

Les sancions per aquests motius poden arribar fins als 30.000€.

A l'hora de treballar amb galetes s'ha de tenir molt present la legislació vigent, que ens obliga a informar l'usuari que el nostre lloc web fa servir galetes i amb quins fins, i amb un enllaç que expliqui com pot desactivar-les al seu navegador.

Declinació de l'ús de galetes

Com que per guardar la informació sobre l'acceptació de les galetes cal fer servir galetes, si un usuari en declina l'ús, no es pot guardar aquesta preferència. Haurà de declinar-la a cada pàgina que visiti o resignar-se a veure el missatge.

A la pràctica és molt difícil trobar un lloc web que no faci servir galetes, ja que són imprescindibles fins i tot per generar analítiques. Per aquesta raó podeu donar per descomptat que haureu d'informar l'usuari que el vostre lloc web fa servir galetes i respectar la normativa.

S'ha de tenir en compte també que si les dades guardades en les galetes són de caràcter personal, s'ha d'aplicar també la Llei orgànica de protecció de dades (LOPD).

Llei orgànica de protecció de dades (LOPD)

Podeu trobar més informació sobre aquesta llei al següent enllaç: www.goo.gl/Fmpuzv i www.goo.gl/zHqQIY.

2.6.2 Utilització de les galetes

El funcionament de les galetes o *cookies* és molt senzill, però utilitzar-les no ho és tant. S'hi accedeix a través de la propietat `document.cookie`, tant per recuperar-les com per guardar-les. Podeu veure-ho en el següent exemple:

```
<script>
  // Afegim una nova galeta
  document.cookie="seu=Barcelona";
  document.cookie="paf=1";

  // Recuperem totes les galetes
  let galetes = document.cookie;

  // Les mostrem a la consola
  console.log(galetes);
</script>
```

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

Si comproveu la consola, veureu el contingut de la vostra galeta. En cas de provar-lo en CodePen, veureu també les galetes afegides per CodePen separades per un ;

unes de les altres: "`seu=Barcelona; paf=1; grid_layout=list; _gat=1; _ga=GA1.2.197952496.1453793789`"

El primer que veieu en aquest exemple és com es guarda una informació a una galeta; sempre ha de ser amb el següent format: `clau=valor`, i si existeixen més galetes, estaran separades amb un `;`.

Segurament us ha estranyat aquest fragment de codi:

```
1 document.cookie="seu=Barcelona";
2 document.cookie="paf=1";
```

> Es podria esperar que en sobreescriure el valor de la propietat `document.cookie`, aquest valor fos "`paf=1`" i el valor `seu=Barcelona` fos eliminat. Això no passa, perquè aquesta propietat té uns *setter* i *getter* natius que en modifiquen el comportament i, per tant, **no és el mateix el que s'escriu i el que es llegeix**.

Es coneixen com a *setter* les funcions (o mètodes) encarregades d'establir un valor a una propietat i *getter* les que els recuperen.

Provem ara de guardar múltiples galetes:

```
1  <script>
2  // Afegim una nova galeta
3  document.cookie="seu=Barcelona";
4  document.cookie="seu=Tarragona";
5  document.cookie="seu=Lleida";
6  document.cookie="seu=Girona";
7
8  // Recuperem totes les galetes
9  let galetes = document.cookie;
10
11 // Les mostrem a la consola
12 console.log(galetes);
13 </script>
```

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

Com podeu apreciar, només s'ha guardat el valor `seu=Girona`. En realitat s'han guardat tots, però els següents els han sobreescrit, ja que només hi pot haver un valor associat a cada clau.

Una possible solució seria guardar les dades com un *array*, però **les galetes només poden guardar cadenes de text**, i per tant, si necessitem un comportament més avançat, l'hem d'implementar nosaltres mateixos o fer servir alguna llibreria externa.

Després d'entendre com guardar i recuperar les galetes, se'ns presenta un problema nou: com ho fem per recuperar només la galeta que ens interessa i no totes? Malauradament ja heu vist tot el suport proporcionat per l'especificació, és a dir: com afegir una galeta, com sobreescriure'n el valor i com recuperar-les totes. Si volem recuperar-ne només una, hem d'implementar la nostra pròpia solució, per exemple:

```
1 <script>
```

```
// Afegim una nova galeta
document.cookie = "seu=Barcelona";
document.cookie = "paf=1";

// Extraiem la galeta corresponent a la seu
let patro = new RegExp("(?:seu)=(.+?)(?=;|$)", "g"),
  seu = patro.exec(document.cookie)[1];

// La mostrem a la consola
console.log(seu);
</script>
```

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

El primer que fem és definir un patró que ens capturarà el valor de la galeta amb el nom `seu`, i a continuació l'executem. El resultat d'executar l'expressió regular és un *array* que conté a la primera posició (índex 0) el text coincident complet i a les següents posicions el contingut dels grups de captura ignorant el primer, ja que té el format de grup no capturable: `(?:)`. És a dir, a l'índex 1 es troba el contingut del grup que es troba a continuació del signe igual, ja que és el primer grup de captura vàlid i, per consegüent, es correspon amb el valor de la seu.

Tot això és força enrevessat, així que, per simplificar-ho, podem crear una funció que ens facilitarà la tasca, com en aquest exemple:

```
<script>
// Afegim una nova galeta
document.cookie = "seu=Barcelona";
document.cookie = "paf=1";

// Recuperem les galetes
let paf = getCookie('paf'),
  seu = getCookie('seu');

// Mostrem les dades per consola
console.log("Totes les galetes:", document.cookie);
console.log("Seu:", seu);
console.log("PAF:", paf);

function getCookie(key) {
  let patro = new RegExp("(?:"+key + ")=(.+?)(?=;|$)", "g");
  return patro.exec(document.cookie)[1];
}
</script>
```

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

Com podeu comprovar, en fer servir una funció que encapsuli la lògica per recuperar una galeta, se'n simplifica molt la utilització; només hem d'invocar `getCookie()` i passar el nom de la clau que volem recuperar.

A banda d'aquest mètode -més simple- per guardar una galeta, les galetes admeten altres opcions per limitar-ne l'ús i la caducitat; són les següents:

- **path**: per exemple `/`, `/actualitat`. Si no s'especifica a la galeta, s'hi podrà accedir des de totes les pàgines del lloc web.

- **domain**: per exemple `exemple.com`, `subdomini.exemple.com`. Si s'especifica el domini principal, també es podrà llegir des dels subdominis; si no s'especifica, només es podrà llegir des del domini principal.
- **max-age**: durada màxima en segons fins que la galeta caduqui.
- **expires**: data en format GMT en què expirarà la galeta. Si no s'especifica, la galeta caduca quan finalitza la sessió (generalment quan es tanca la pestanya concreta o el navegador)
- **secure**: la galeta només es podrà transmetre sobre protocols segurs com HTTPS. Aquest tipus de galetes no necessita valor. Totes les galetes enviades a través de HTTPS són automàticament segures.

Tot i que existeix una opció `secure`, **les galetes són inherentment insegures**, i no heu de fer-les servir mai per guardar informació sensible ni confidencial.

Cada opció i el seu valor anirà separat per un punt i coma (`;`) i un espai en blanc, i es separaren els parells *opció* i *valor* per un signe igual (`=`).

Aprofitant que ja coneixem totes les opcions possibles, implementarem la funció `setCookie()`:

```
<script>
function setCookie(key, value, options) {
  let cookie = encodeURIComponent(key) + '=' + encodeURIComponent(value);

  if (options) {
    for (option in options) {
      if (option === 'secure') {
        cookie += '; ' + encodeURIComponent(option);
      } else {
        cookie += '; ' + encodeURIComponent(option) + '=' + encodeURIComponent(
            options[option]);
      }
    }
  }

  document.cookie = cookie;
}

function getCookie(key) {
  let patro = new RegExp("(?:" + key + ")=(.+?)(?=;|$)", "g");
  return patro.exec(document.cookie)[1];
}

// Afegim una nova galeta
setCookie('seu', 'Barcelona', {
  'max-age': 300,
  path: '/',
  domain: 'codepen.io'
});

// Mostrem les dades per consola
console.log("Totes les galetes:", document.cookie);
console.log("Seu:", getCookie('seu'));
</script>
```

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

Fixeu-vos que, per passar les opcions de les galetes, hem fet servir un objecte literal de JavaScript, d'aquesta manera el codi queda molt més net i compacte. La propietat `max-age`, en incloure el signe `-`, s'ha de posar entre cometes.

Si proveu de passar el paràmetre `secure` no us funcionarà perquè requereix que la connexió es faci sobre HTTPS, i ni l'exemple en local ni en CodePen corren sobre aquest protocol.

Tot i que l'especificació no requereix fer la codificació completa dels continguts de la galeta (només els espais, `,` i `;` han de ser codificats) és més simple codificar tots els valors i les claus fent servir la funció `encodeUriComponent()`. D'aquesta manera ens assegurem que els valors guardats seran correctes.

Temps i durada de les galetes

Tot i que la nostra funció per guardar galetes rutlli i contempli totes les opcions disponibles, encara podem tenir problemes amb un apartat: establir la data en la qual volem que expiri.

L'especificació ens diu que el format ha de ser el següent: `Wdy, DD-Mon-YYYY HH:MM:SS GMT` com per exemple: `Sat, 02 May 2009 23:38:25 GMT`. Però si fem servir l'objecte `Date` de JavaScript el que ens retorna serà semblant a això:

```
Date.now();
// 1456167356375
```

L'objecte `Date` ens retorna la data en mil·lisegons. Afortunadament, disposa de mètodes per poder modificar aquesta data; vegem-ne alguns exemples:

```
<script>
  let ara = new Date(),
      mes,
      proximMes,
      nadal;

  console.log("La data d'avui en UTC", ara.toUTCString());

  mes = ara.getMonth();
  console.log("Quin és el mes actual?", mes);

  proximMes = (mes + 1) % 12;
  ara.setMonth(proximMes);
  console.log("Quina és la data corresponent al pròxim mes?", ara.toUTCString()
      );

  nadal = new Date(2016, 11, 25);
  console.log("Quin dia és Nadal el 2016?", nadal.toUTCString());

  console.log("Desfasament horari per la nostra localització (en minuts):",
      nadal.getTimezoneOffset()
)
</script>
```

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

El resultat d'invocar el mètode `toUTCString()` produeix una data exactament amb el format que necessitem, com aquesta: `Tue, 22 Mar 2016 19:08:01 GMT`. Fixeu-vos que, en l'últim exemple, creem una data en concret, però el resultat pot ser inesperat, ja que:

- Gener correspon al mes 0, i per tant desembre és el mes 11.
- La data mostrada serà aquesta: `Sat, 24 Dec 2016 23:00:00 GMT`
- El desfasament horari serà de -60 minuts, això explica per què la data mostrada sembla errònia per una hora. Quan es crea l'objecte Date es fa servir l'ús horari del client, que en el nostre cas és `GMT+1`, i en convertir-la en `GMT` es descompta aquesta hora.

Treballar amb dates afegeix una complicació extra a la creació de galetes. Per aquesta raó és més recomanable fer servir l'opció `max-age`, que ens permet especificar l'edat màxima que pot assolir la galeta, expressada en segons. Així, per exemple, podem crear-ne una que expiri en un mes així:

```
setCookie('seu', 'Barcelona', {
  'max-age': 60*60*24*30
});
```

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

Per tant, podem fer que les galetes persisteixin en el temps, encara que es finalitzi la sessió del navegador. Ara bé, com podem esborrar-les? La manera de fer-ho és molt simple, només hem de guardar una galeta amb la mateixa clau i amb l'opció `max-age=-1` (o amb l'opció `expires` amb una data ja passada). D'aquesta manera la galeta expira immediatament:

```
setCookie('seu', 'Barcelona', {
  'max-age': -1
});
```

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

2.6.3 Utilització de Web Storage

Aquesta Web API permet emmagatzemar dades en l'ordinador client amb més possibiltats que les galetes. Distingeix dues maneres de realitzar aquest emmagatzematge:

- LocalStorage: les dades no expiren mai i, per tant, es requereix esborrat manual o per programa

- SessionStorage: les dades es guarden només per una sessió i, en tancar la finestra del navegador s'esborra.

A banda de l'anterior, el funcionament és idèntic a tots dos tipus de Web Storage.

Web Storage

Podeu trobar l'especificació del *Living Standard* al següent enllaç: www.goo.gl/JVCEHl. I informació sobre el funcionament detallat en aquest: www.goo.gl/mPwJxl.

El **LocalStorage** comparteix la informació entre totes les pàgines del mateix domini, en canvi, el **SessionStorage** és diferent per a cada finestra; de manera que si tenim tres finestres obertes amb la mateixa pàgina, la informació del **SessionStorage** serà diferent per a cadascuna, mentre que la del **LocalStorage** serà la mateixa.

A diferència de l'ús de galetes, aquesta Web API ofereix una interfície molt clara per interactuar amb el magatzem de dades:

- **Storage.setItem(key, value)**: per guardar un valor amb la clau especificada.
- **Storage.getItem(key)**: per recuperar el valor corresponent a la clau.
- **Storage.removeItem(key)**: per eliminar l'element corresponent a la clau del magatzem.
- **Storage.clear()**: elimina totes les claus del magatzem.

Vegem com funcionen tots aquests mètodes:

```
<script>
  localStorage.setItem('seus', ['Barcelona', 'Girona', 'Lleida', 'Tarragona']);
  localStorage.setItem('seu seleccionada', 'Barcelona');
  localStorage.setItem('paf', 1);
  localStorage.setItem('confirmat', false);


  console.log("Comprovem que s'han creat:"); console.log("seus:", localStorage.
      getItem('seus'));
  console.log("seleccionada:", localStorage.getItem('seu seleccionada'));
  console.log("paf:", localStorage.getItem('paf'));
  console.log("confirmat:", localStorage.getItem('confirmat'));


  console.log("Eliminem la seu seleccionada.")
  localStorage.removeItem('seu seleccionada');

  console.log("Comprovem el valor de la seu seleccionada:", localStorage.
      getItem('seu seleccionada'));

  console.log("Eliminem tots els valors");
  localStorage.clear();

  console.log("Comprovem que s'han eliminat:")
  console.log("seus:", localStorage.getItem('seus'));
  console.log("seleccionada:", localStorage.getItem('seu seleccionada'));
  console.log("paf:", localStorage.getItem('paf'));
  console.log("confirmat:", localStorage.getItem('confirmat'));
</script>
```

Per fer servir el **SessionStorage** en lloc del **LocalStorage**, només hem de canviar `localStorage` per `sessionStorage`.

Com podeu veure, a diferència de les galetes, els valors emmagatzemats poden ser de diferents tipus: enters, cadenes, *booleans* i fins i tot *arrays* (només caldrà fer servir `String.split()` per recuperar l'*array*).

No cal crear cap objecte ni configurar res per accedir als magatzems de dades, són accessibles directament a través dels objectes globals `localStorage` i `sessionStorage` que ens proporciona el navegador, i la informació es guarda automàticament associada al domini on s'ha creat.

La persistència de les dades es basa en el tipus de magatzem: si fem servir el *LocalStorage* aquestes dades persistiran fins que les esborrem nosaltres mateixos via codi o quan l'usuari esborri les dades de navegació; en canvi, si fem servir el *SessionStorage* , totes les dades s'esborraran tan bon punt es tanqui la finestra del navegador.

Model d'objectes del document

Xavier Garcia Rodríguez

Índex

Introducció

Avui dia és molt difícil trobar una pàgina o aplicació web que no requereixi modificar, afegir o eliminar alguns dels seus elements per respondre a les accions dels usuaris. Tant pot ser un menú desplegable, un diàleg d'entrada de dades o un panell que mostra el contingut d'un carretó electrònic: en tots els casos cal manipular aquests elements. Aquestes modificacions es fan a través del model d'objectes del document (DOM), i es poden fer treballant directament amb les seves interfícies o fent servir alguna biblioteca com jQuery.

En aquesta unitat, "Model d'objectes del document", s'expliquen les característiques principals del DOM, les principals interfícies que el componen i com s'hi ha de treballar per modificar el contingut de qualsevol pàgina o aplicació web. L'assimilació dels continguts d'aquesta unitat és fonamental, atès que es faran servir en major o menor grau en el desenvolupament de totes les vostres aplicacions a la banda del client.

A l'apartat **"Programació amb el DOM (Document Object Model)"** trobareu una introducció a les interfícies del DOM, més concretament a la utilització de les interfícies principals: `Node`, `Document` i `Element`. Cal destacar que no només es tracta la creació i eliminació, sinó també la modificació i especialment la cerca d'elements, tant a partir del document com entre els elements descendents d'un node concret. Per altra banda. També es tractarà la detecció d'*events* lligada a aquests elements.

A l'apartat **"Programació amb el DOM i la biblioteca jQuery"** s'introdueix la biblioteca jQuery, que destaca especialment per la facilitat amb la qual es poden manipular conjunts d'elements del DOM i assignar-hi o eliminar-ne *events*. Com que és palesa la importància de cercar elements i conjunts d'elements, es recalcarà la utilització de selectors (tant de CSS com de propis) i com refinar aquestes seleccions. Finalment es farà una introducció al patró model-vista-vontrolador i als entorns de treball (*frameworks*) i les biblioteques més populars que l'implementen, juntament amb els components web, una component experimental que forma part de l'especificació del W3C.

Com que els continguts d'aquesta unitat són aplicables a la majoria de les aplicacions web, podeu aplicar aquests coneixements a la vostra feina (si us dediqueu al desenvolupament web) o als vostres projectes personals. Com més practiqueu, més fàcil serà assimilar aquests continguts.

Per assolir els objectius d'aquesta unitat cal que proveu tots els exemples i que hi feu modificacions per entendre com funcionen; també cal que feu totes les activitats proposades. En cas de dubte, primer es recomana consultar a internet les múltiples fonts d'informació (Mozilla Developer Network, StackOverflow, lloc oficial de jQuery...) i preguntar al fòrum de l'assignatura, ja que així us podran ajudar els vostres companys o el professor.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions web analitzant i aplicant les característiques del model d'objectes del document.

- Reconeix el model d'objectes del document d'una pàgina web.
- Identifica els objectes del model, les seves propietats i els seus mètodes.
- Identifica les diferències que presenta el model en diferents navegadors.
- Crea i verifica un codi que accedeixi a l'estructura del document.
- Crea nous elements de l'estructura i en modifica elements ja existents.
- Associa accions als esdeveniments del model.
- Programa aplicacions web de manera que funcionin en navegadors amb diferents implementacions del model.
- Independitza les tres facetes (contingut, aspecte i comportament) en aplicacions web.

1. Programació amb el DOM (Document Object Model)

La manipulació del **model d'objectes del document** –més conegut per les seves sigles en anglès: DOM (Document Object Model)– és fonamental per al desenvolupament d'aplicacions web, perquè sense aquesta capacitat no és possible alterar la visualització de les pàgines dinàmicament.

Internament, els navegadors treballen amb els documents web com si es tractés d'un arbre, i és a partir d'aquest arbre que es pot modificar la representació de la pàgina, tant afegint nous elements (paràgrafs, capçaleres, taules...) com modificant els atributs dels nodes que ja es troben al document o eliminant-los.

A més a més, és possible cercar tant elements concrets com llistes d'elements fent servir diferents propietats i mètodes segons les vostres necessitats: a través de relacions (el primer element, el pròxim element...), cercant segons el tipus d'element, el seu identificador o fent una cerca més complexa gràcies als selectors de CSS.

A banda de manipular aquests elements, també és possible accedir-hi per fer operacions de consulta, com per exemple per extreure informació dels atributs o del text contingut.

1.1 Estructura del DOM. Interfícies principals

El model d'objectes del document és una interfície que facilita treballar amb documents **HTML, XML i SVG**. És a dir, independentment del llenguatge de programació que es faci servir, el nom dels mètodes i paràmetres per manipular-lo són idèntics, tant si es tracta de PHP, com de JavaScript o de Python, per exemple.

Aquesta interfície defineix els mètodes i propietats que permeten accedir i manipular el document com si es tractés d'un arbre de nodes, en el qual l'arrel és el node `document`, que pot contenir qualsevol quantitat de nodes fills, i les fulles són els nodes que no tinguin cap descendent (generalment el text dels elements HTML).

Especificació del DOM segons el W3C

Podeu trobar l'especificació del DOM del W3C (que és el consorci internacional que treballa per desenvolupar els estàndards utilitzats per internet) en l'enllaç següent: www.w3.org/DOM.

Tot i que el més habitual és accedir al DOM a través de JavaScript, el DOM no forma part de l'especificació de JavaScript, sinó que es troba especificat pel W3C com una sèrie d'**interfícies independents** de la plataforma i el llenguatge.
Per tant, els mètodes i les propietats que formen part d'aquestes interfícies no són propis de JavaScript, sinó que es troben **disponibles a qualsevol llenguatge** que ofereixi la capacitat de manipular el DOM, com per exemple Python, tant directament com a través de biblioteques.

Al llarg dels anys s'han produït molts canvis a l'especificació, i ha evolucionat i s'ha simplificat. Algunes de les interfícies antigues han estat declarades obsoletes i no s'han de fer servir perquè els navegadors moderns poden deixar d'admetre-les.

És important tenir-ho en compte a l'hora de consultar materials de referència, ja que és fàcil trobar informació desactualitzada. És recomanable consultar directament la documentació del web Mozilla Developer Network (www.developer.mozilla.org) o l' 'especificació viva' (*DOM Living Standard*; www.dom.spec.whatwg.org) per aclarir els dubtes que pugueu tenir.

WHATWG

El *Web Hypertext Application Technology Working Group* (WHATWG) és una comunitat fundada per integrants d'Apple, la fundació Mozilla i Opera. En trobareu més informació en l'enllaç següent: www.goo.gl/AFxeCb.

Actualment hi ha dos grups treballant en les especificacions per al web, el *W3C* i el *WHATWG*. Encara que tots dos van col·laborar temporalment fins al 2012, van deixar de fer-ho a causa del fet que els objectius de WHATWG (a favor dels estàndards vius) i del W3C (especificacions estàtiques) eren contraris. Actualment el W3C actualitza la seva versió de l'especificació a partir d'*instantànies* de l'especificació viva, centrant-se en la correcció d'errors.

Les especificacions proporcionades pel W3C sobre el DOM estan dividides en nivells i cadascun d'aquests nivells ho està, al seu torn, en diferents especificacions. D'aquesta manera es poden conèixer les capacitats quant a manipulació del DOM dels diferents navegadors. Per exemple, el *DOM nivell 1* és admès per pràcticament tots els navegadors, en canvi, durant l'any 2016 l'especificació *DOM3 carregar i guardar* encara no l'admetia cap navegador.

Per aquesta raó, trobareu que a la documentació sobre diferents webs API s'indica el nivell de DOM necessari per utilitzar-les, ja que no funcionaran en navegadors que no admetin aquest nivell.

1.1.1 Tipus de models DOM

En desenvolupar aplicacions web es treballa amb la interfície HTMLDocument, que s'aplica als elements HTML, però l'especificació del DOM consta de més interfícies que permeten manipular diferents tipus de documents. Són les següents:

- **Interfície HTMLDocument** (documents HTML): és una extensió de la interfície *Document* i afegeix l'accés a característiques pròpies dels navegadors, com són la finestra (`window`) o les pestanyes.
- **Interfície XMLDocument** (documents XML): aquesta interfície es troba en fase experimental i a finals de l'any 2016 encara no l'admet cap navegador.
- **Interfícies SVG**: es tracta d'un conjunt d'interfícies que permeten manipular directament des de JavaScript els elements d'una imatge vectorial de tipus SVG, per afegir formes, canviar colors... No es fa servir gaire perquè és molt complexa.

Les interfícies SVG

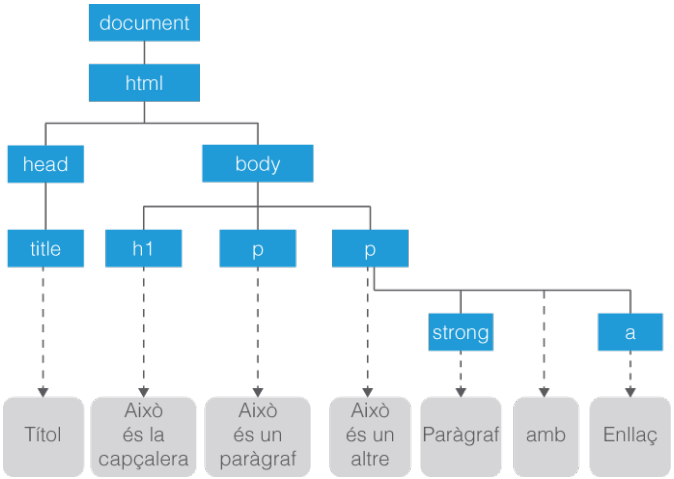
Podeu trobar un exemple d'ús de les interfícies SVG juntament amb JavaScript en l'enllaç següent: goo.gl/gz6Lg6.

1.1.2 Estructura del DOM

Per entendre més fàcilment com s'estructura el DOM fixeu-vos en el següent codi HTML i contrasteu-lo amb la figura 1.1:

```
<html>
  <head>
    <title>Títol</title>
  </head>
  <body>
    <h1>Això és la capçalera</h1>
    <p>Això és un paràgraf</p>
    <p>Això és un altre <strong>paràgraf</strong> amb <a>enllaç</a></p>
  </body>
</html>
```

Figura 1.1. Estructura d'arbre corresponent a un document HTML



Com es pot apreciar, l'arrel de l'arbre és el node `document`, tot i que no forma part del codi HTML. A continuació trobem el node `html`, que conté els nodes `head` i `body`; aquests contenen altres nodes: `title`, `h1` i `p`. Tots aquests nodes són de tipus `Element`. En canvi, l'últim element de cada branca (les fulles) són de tipus `Text` i tenen una consideració diferent.

Pareu atenció a l'últim paràgraf del document: en aquest cas el text inclou els elements `strong` i `a`, però aquests elements no pengen del text sinó que pengen del node `p`. Per tant, es pot concloure que **un element sempre és descendent d'un altre element** i mai d'un text.

1.1.3 Adaptacions de codi per diferents navegadors

S'ha de tenir en compte que encara que s'amplien les especificacions de les interfícies i dels llenguatges de programació contínuament, els navegadors antics no les admeten.

IE 6 al mercat xinès

A l'agost del 2012, Internet Explorer 6 encara era el segon navegador més utilitzat a la Xina (22.41% dels usuaris).

Actualment, tots els navegadors s'actualitzen automàticament de forma predeterminada. Per aquesta raó, es parla de Google Chrome o de Firefox sense indicar-ne la versió. En canvi, els navegadors més antics s'havien d'actualitzar manualment i en alguns països l'ús de navegadors obsolets (especialment Internet Explorer) ha estat molt estès fins fa pocs anys.

Afortunadament, cada vegada hi ha menys usuaris que facin servir navegadors obsolets, i molts grups han deixat fer-los compatibles perquè consideren que mentre hi hagi compatibilitat amb aquests navegadors obsolets, la gent continuarà fent-los servir i això perjudica tant els desenvolupadors com els usuaris amb navegadors actualitzats.

En cas de requerir compatibilitat amb aquests navegadors, al mateix temps que s'aprofiten les noves tecnologies, teniu a la vostra disposició dues opcions:

- **Biblioteques**: fer servir una biblioteca que implementi les característiques que necessiteu i que doni compatibilitat amb navegadors antics (per exemple, la branca 1.12 de jQuery). En aquest cas, sempre que sigui possible, es recomana fer servir la biblioteca: les biblioteques implementen el comportament correcte per a tots els navegadors sense haver de modificar el vostre codi.
- **Polyfill**: consisteix a afegir, juntament amb el vostre codi, una implementació pròpia de la característica que necessiteu i que no és disponible en tots els navegadors antics. Per exemple, abans que HTML5 fos disponible per a tots els navegadors moderns, era possible fer servir *polyfills* per implementar algunes de les noves característiques del llenguatge sense haver de preocupar-se pels navegadors dels usuaris.

Al web de Mozilla Developer Network es pot trobar el *polyfill* per implementar moltes de les característiques d'HTML5.

Per exemple, si voleu fer una implementació per treballar amb JSON en navegadors antics, faríeu servir un codi similar al següent:

Implementació 'polyfill' per a JSON

Es pot trobar la implementació completa del *polyfill* per a l'objecte JSON en l'enllaç següent: goo.gl/p4nm6l.

```
1 if (!window.JSON) {
2   window.JSON = {
3     parse: function(text) { // Codi per retornar un objecte a partir del text},
4     stringify: function(obj) { // Codi per retornar una cadena de text a partir
          de l'objecte}
5   }
6 }
```

Cal destacar que la implementació pròpia dels navegadors d'aquestes funcions és molt més eficient que els *polyfills*. Per aquesta raó, és molt important comprovar primer si el navegador ja les implementa i, si no és així, cal afegir-les. Per fer-ho només cal comprovar si es troben definides a l'objecte o al prototipus, segons el cas.

Tenint en compte la complexitat afegida d'utilitzar *polyfills* i les seves limitacions, és més recomanable fer servir biblioteques que ja implementin aquests *polyfills* o incloguin les funcionalitats que requeriu en lloc de fer la vostra pròpia implementació.

1.2 Interfícies del DOM

L'especificació del DOM està dividida en múltiples interfícies que determinen les possibles accions que es poden realitzar amb cada element; per exemple, per conèixer la classe o classes aplicades a un element, caldrà utilitzar la interfície `Element`; en canvi, per consultar o modificar el valor d'un atribut de l'element, s'utilitza la interfície `Attr`.

Llistat d'interfícies

Podeu trobar una llista de totes les interfícies DOM en l'enllaç següent: goo.gl/X3rGDj.

A continuació podeu trobar una llista de les interfícies més destacables per a la manipulació de documents HTML:

- **Node**: molts dels elements del DOM hereten d'aquest tipus i, per tant, ofereixen aquestes funcionalitats. Aquesta interfície permet consultar i manipular els nodes, com per exemple navegar a través dels nodes fills, pares i contigus, cercar nodes, afegir nodes nous o eliminar-los.
- **Document** (herència de `Node`): aquesta interfície és la que s'aplica a l'arrel del document. Permet, entre altres coses, conèixer l'element actiu, manipular les galetes (més conegudes com a *cookies*) i obtenir informació global del document.
- **Element** (herència de `Node`): s'aplica a tots els elements del document, és a dir, a les etiquetes que apareixen com a `body`, `h1` o `p`. A partir d'aquesta interfície es pot obtenir una llista d'atributs, la classe de l'element, el seu `id` i afegir observadors per escoltar diferents *events*.
- **Attr**: aquesta interfície permet consultar i modificar els atributs d'un element. En versions anteriors a DOM4 la interfície `Attr` derivava de `Node`, però no es recomana fer servir aquestes funcionalitats perquè no hi hagi discrepàncies amb els navegadors més actuals.
- **Event**: aquesta interfície és implementada per altres interfícies de l'especificació i conté tots els mètodes comuns per representar un esdeveniment.

Cal destacar que existeix una interfície per a `text`, però no es fa servir gaire. El seu objectiu principal és representar el text contingut en un `element` o `attr` (atribut).

1.3 Interficie 'Node'

D'entre totes les interfícies la més important és `Node`, ja que `Document` i `Element` deriven d'aquesta. És a dir, totes les propietats i mètodes de `Node` són accessibles tant per a `Document` com per a `Element`.

Propietats de 'Node'

Podeu trobar informació més detallada sobre la interficie `Node` en l'enllaç següent: www.http://goo.gl/Rmwm4h.

Cal destacar que tot i que això implica que `Document` i `Element` inclouen la funcionalitat per navegar entre nodes, no s'acostuma a fer-les servir. En el cas

de la primera interfície s'acostuma a cercar directament l'element que interessi (a través del tipus d'element, el seu identificador o la seva classe), mentre que el segon, a més de disposar de la capacitat de cerca, inclou propietats específiques per navegar entre els nodes de tipus *element*, filtrant-ne, així, la resta (per exemple, els de tipus *text*).

1.3.1 Informació bàsica dels nodes: 'nodeType' i 'nodeName'

La propietat `nodeType` permet determinar el tipus d'un node. Conté un valor numèric corresponent a les pseudoconstants (recordeu que realment a JavaScript no existeixen les constants), que podeu veure a la taula taula 1.1.

TAULA 1.1. Correspondència de valors de la propietat nodeType

Pseudoconstant	Valor
ELEMENT_NODE	1
TEXT_NODE	3
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11

Per exemple, per comprovar el valor de la propietat nodeType de `document` només cal mostrar-la per la consola:

```
1 console.log(document.nodeType);
```

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

Com es pot apreciar, el valor retornat és `9`, ja que és el valor associat a `DOCUMENT_NODE`.

En canvi, si es consulta el valor de la propietat d'un element, el valor associat serà `1`, com correspon a `ELEMENT_NODE`:

`getElementById` és un mètode de la interfície document que permet obtenir un element per al seu `id`.

```
1 <div>
2   <p id="primer">Primer paràgraf</p>
3 </div>
4
5 <script>
6 var primer = document.getElementById('primer');
7 console.log(primer.nodeType);
8 </script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/QKWKNg?editors=1012.

Per altra banda, la propietat `nodeName` permet saber el nom del node, i això permet distingir-los entre d'altres. Aquest nom no correspon necessàriament amb el de l'etiqueta que crea el node, com es pot comprovar en l'exemple següent:

```
<div>
  <p id="primer">Primer paràgraf</p>
</div>

<script>
  var primer = document.getElementById('primer');
  console.log(document.nodeName);
  console.log(primer.nodeName);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/EgxgjZ?editors=1012.

Fixeu-vos que el nom del node `document` és `#document`, mentre que el de l'element `p` és `P`.

1.3.2 Contingut textual: 'textContent'

Aquesta propietat retorna el contingut textual:

```
<p id="primer">Primer paràgraf</p>

<script>
  var primer = document.getElementById('primer');
  console.log(primer.textContent);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/GjRjOq?editors=1012.

1.3.3 Relacions entre nodes: 'parentNode', 'firstChild', 'lastChild', 'previousSibling' i 'nextSibling'

La propietat `parentNode` permet accedir al pare del node o retorna `null` si no existeix (per exemple, si el node no s'ha afegit al document).

```
<div id="contenidor">
  <p id="primer">Primer paràgraf</p>
</div>

<script>
  var primer = document.getElementById('primer');
  console.log(primer.parentNode.nodeName);
</script>
```

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

Com es pot apreciar, el nom del node retornat és `DIV`, el corresponent al node pare. Aquest comportament permet recórrer l'arbre ascendentment; per exemple, es podria accedir al pare del pare: `primer.parentNode.parentNode.nodeName`.

Les propietats `firstChild` i `lastChild` permeten accedir al primer i a l'últim fill d'un node, com es pot comprovar en l'exemple següent:

```
<div id="contenidor"><p id="primer">Primer paràgraf</p><p id="segon">Segon parà
    graf</p><p id="tercer">Tercer paràgraf</p></div>

<script>
  var contenidor = document.getElementById('contenidor');
  console.log(contenidor.firstChild.textContent);
  console.log(contenidor.lastChild.textContent);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/VKwKbw?editors=1012.

> Els **salts de línia** són interpretats com a nodes de tipus text i, per consegüent, en cas d'afegir salts de línia, tant el `firstChild` com el `lastChild` serien nodes de tipus text.
> En el cas de `previousSibling` i `nextSibling` es produeix el mateix comportament, i per aquesta raó cal tenir en compte els salts de línies i tabulacions quan es tracta amb aquests mètodes.

Per altra banda, `previousSibling` i `nextSibling` permeten accedir als nodes germans, anterior i posterior respectivament, d'un mateix node:

```
<div id="contenidor"><p id="primer">Primer paràgraf</p><p id="segon">Segon parà
    graf</p><p id="tercer">Tercer paràgraf</p>
</div>

<script>
  var segon = document.getElementById('segon');
  console.log(segon.previousSibling.textContent);
  console.log(segon.nextSibling.textContent);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/mAdAxE?editors=1012.

En executar aquest exemple es mostrarà correctament el contingut textual dels nodes anterior i posterior, és a dir, `Primer paràgraf` i `Tercer paràgraf`.

1.3.4 Obtenció de nodes descendents: 'childNodes' i 'hasChildNodes'

Per altra banda, la propietat `childNodes` ens permet accedir a la llista de nodes continguts que es pot tractar com un *array* i conèixer la quantitat de nodes a través de la propietat `length`. Per altra banda, si només es vol saber si conté altres nodes o no, es pot invocar el mètode `hasChildNodes`, que retornarà `true` si en conté o `false` en cas contrari.

Com que la llista de nodes es pot tractar com un *array*, es pot recórrer normalment:

```
<div id="contenidor"></div>
  <p id="primer">Primer paràgraf</p>
  <p id="segon">Segon paràgraf</p>
  <p id="tercer">Tercer paràgraf</p>
</div>

<script>
  var contenidor = document.getElementById('contenidor');

  console.log('El contenidor conté altres nodes: ' + contenidor.hasChildNodes()
      );

  console.log('Conté i + contenidor.childNodes.length + ' nodes');

  for (var i = 0; i < contenidor.childNodes.length; i++) {
    console.log('Trobat un node de tipus ' + contenidor.childNodes[i].nodeType
        + ': ' + contenidor.childNodes[i].nodeName);
  }
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/vXYXbX?editors=1012.

Fixeu-vos que en aquest cas s'han fet servir la indentació habitual, i en recórrer la llista de nodes s'han trobat quatre nodes de tipus text, corresponents als nodes generats pels salts de línia i les indentacions.

1.3.5 Inserció de nodes: 'appendChild' i 'insertBefore'

Aquests dos mètodes permeten afegir un node com a fill d'un altre. La diferència és que `appendChild` afegeix el node al final de la llista de fills, mentre que amb `insertBefore` el node s'afegeix abans del node de referència passat com a argument.

El mètode `insertBefore` requereix que es passin dos arguments, el node que s'ha d'afegir i el node de referència o `null`, obligatòriament.

El mètode `createElement` forma part de la interfície `Element`.

```
<div id="contenidor"></div>

<script>
  var contenidor = document.getElementById('contenidor');

  contenidor.appendChild(document.createElement('p'));
  contenidor.appendChild(document.createElement('div'));
  contenidor.insertBefore(document.createElement('h1'), contenidor.firstChild);

  console.log('El contenidor conté altres nodes: ' + contenidor.hasChildNodes()
      );

  console.log('Conté ' + contenidor.childNodes.length + ' nodes');

  for (var i = 0; i < contenidor.childNodes.length; i++) {
    console.log('Trobat un node de tipus ' + contenidor.childNodes[i].nodeType
        + ': ' + contenidor.childNodes[i].nodeName);
  }
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/WGNooa?editors=1012.

Com es pot apreciar en aquest exemple, els nodes afegits amb `appendChild` s'han afegit en ordre, mentre que el node afegit amb `insertBefore` s'ha afegit davant del primer fill del contenidor i, per tant, ha quedat en primera posició.

S'ha de tenir en compte que en el cas d'invocar aquests mètodes passant com a paràmetre un node que ja es trobi al document, en lloc d'afegir-se, el node es mourà, és a dir, s'eliminarà de la seva posició anterior i s'afegirà a la nova:

```
<div id="contenidor1">
  <h1 id="titol1">Aquest és el títol 1<h1>
</div>
<div id="contenidor2">
    <h1 id="titol2">Aquest és el títol 2<h1>
</div>

<script>
  var contenidor2 = document.getElementById('contenidor2');
  var titol1 = document.getElementById('titol1');
  contenidor2.appendChild(titol1);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/ALBOjL?editors=1010.

Com es pot apreciar, una vegada s'afegeix el `titol1` al `contenidor2` s'elimina automàticament del `contenidor1`.

1.3.6 Eliminació i substitució de nodes: 'removeChild' i 'replaceChild'

El mètode `removeChild` permet eliminar un node fill. S'ha de tenir en compte que per poder eliminar un node s'ha d'accedir al pare d'aquest, per tant, cal recordar que la propietat `parentNode` hi dona accés, tal com es pot veure a l'exemple següent:

```
<div id="contenidor1">
  <h1 id="titol1">Aquest és el títol 1<h1>
</div>
<div id="contenidor2">
    <h1 id="titol2">Aquest és el títol 2<h1>
</div>

<script>
  var titol1 = document.getElementById('titol1');
  titol1.parentNode.removeChild(titol1);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/rrNWpX?editors=1010.

Fixeu-vos que el `titol1` ha desaparegut i només es mostra el `titol2`. El `contenidor1` (el node pare) continua existint, però ara es troba buit.

Tot i que el node ja no forma part del contenidor, es pot conservar una referència (per exemple, la retornada pel mètode `removeChild`) i afegir-lo a un altre node:

```
<div id="contenidor1">
  <h1 id="titol1">Aquest és el títol 1<h1>
</div>
<div id="contenidor2">
    <h1 id="titol2">Aquest és el títol 2<h1>
</div>

<script>
  var contenidor2 = document.getElementById('contenidor2');
  var titol1 = document.getElementById('titol1');

  var refNode = titol1.parentNode.removeChild(titol1);
  contenidor2.appendChild(refNode);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/yaLVmN?editors=1010.

Com es pot apreciar, ara el `contenidor2` conté tots dos títols. Cal tenir en compte que en aquest cas concret no caldria guardar la referència perquè la variable `titol1` ja la manté. Si reemplaceu el codi JavaScript pel següent, el resultat és idèntic:

```
var contenidor2 = document.getElementById('contenidor2');
var titol1 = document.getElementById('titol1');

titol1.parentNode.removeChild(titol1);
contenidor2.appendChild(titol1);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/WGNRZV?editors=1010.

Per altra banda, `replaceChild` permet reemplaçar un node per un altre. Per exemple, es poden intercanviar els continguts dels dos contenidors de l'exemple anterior:

```
<div id="contenidor1">
  <h1 id="titol1">Aquest és el títol 1<h1>
</div>
<div id="contenidor2">
    <h1 id="titol2">Aquest és el títol 2<h1>
</div>

<script>
  var contenidor1 = document.getElementById('contenidor1');
  var contenidor2 = document.getElementById('contenidor2');

  var titol1 = document.getElementById('titol1');
  var titol2 = document.getElementById('titol2');

  var refNode = contenidor1.replaceChild(titol2, titol1);
  contenidor2.appendChild(refNode);
</script>
```

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

En primer lloc s'obté una referència als contenidors i els nodes, seguidament es fa el reemplaça de `titol1` per `titol2`, i finalment s'afegeix el `titol1` al segon contenidor.

Com que `replaceChild` retorna una referència al node reemplaçat, aquesta referència es pot fer servir per afegir aquest node al `contenidor2`. Cal tenir en compte que no és possible invocar el mètode `replaceChild` per fer l'intercanvi al `contenidor2` perquè aquest ja no conté cap node que pugui ser reemplaçat.

1.3.7 Copia de nodes: 'cloneNode'

Aquest mètode permet clonar un node, de manera que es genera un nou node idèntic. Es pot passar `true` com a paràmetre, si volem que es faci una còpia profunda del node (és a dir, que cloni també tots els seus nodes descendents); o `false`, en cas contrari.

Per assegurar la compatibilitat amb navegadors que no admetin el DOM4 és recomanable passar `true` com argument per fer còpies profundes.

En el següent exemple es pot veure com es clona la capçalera de tipus `h1` i s'afegeix cinc vegades:

```
<div id="contenidor"><h1>Això és un títol<h1></div>

<script>
  var contenidor = document.getElementById('contenidor');

  for (var i=0; i<5; i++) {
    var clonedNode = contenidor.firstChild.cloneNode(true);
    console.log(clonedNode);
    contenidor.appendChild(clonedNode);
  }
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/GjRApw?editors=1010.

En cas de passar `false` com a argument, s'haurien afegit cinc elements de tipus `h1`, però sense cap contingut, ja que el node amb el contingut textual `Això és un títol` no s'hauria clonat.

1.4 Interficie 'Document'

Aquesta interfície és herència de `Node` i d'`EventTarget` (aquesta darrera permet la utilització d'*events*), així doncs, és possible manipular el document com si es tractés d'un node i gestionar *events*.

Mètodes i propietats de 'Document'

Podeu trobar més informació sobre la interfície `Document` en l'enllaç següent: www.goo.gl/Rmwm4h.

Les propietats que ofereix aquesta interfície no són gaire utilitzades, en canvi, és fonamental conèixer els seus mètodes, perquè permeten cercar i crear nous nodes. En aquests materials només es tractarà un subconjunt d'aquests mètodes, els més utilitzats, ja que molts d'aquests tenen aplicacions molt concretes i no cal conèixer-los.

1.4.1 Creació de nodes: 'createTextNode' i 'createElement'

Aquests mètodes permeten crear nodes de text i elements, respectivament. Com que totes dues implementen la interfície `Node`, es poden afegir al document, eliminar-los, reemplaçar-los o clonar-los:

```
<h1 id="titol1">Aquest és el primer node de text.<h1>

<script>
  var titol1 = document.getElementById('titol1');
  var nouNodeDeText = document.createTextNode('I aquest és el segon, afegit din
      àmicament');
  titol1.appendChild(nouNodeDeText);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/BLamyb?editors=1010.

Com es pot apreciar, el codi HTML només inclou un fragment de text, però s'afegeix el segon dinàmicament en crear un nou node i afegint-lo al contenidor (l'element `h1`).

Per afegir-lo al principi del contenidor en lloc del final només cal substituir l'última línia per:

```
titol1.insertBefore(nouNodeDeText, titol1.firstChild);
```

El mètode `createElement` també crea nodes, però en aquest cas són de tipus element i, per consegüent, implementen la interfície 'Element'. Això permet crear elements d'HTML, com es pot apreciar en l'exemple següent:

```
<div id="contenidor"></div>

<script>
  var contenidor = document.getElementById('contenidor');

  var nouElement = document.createElement('h1');
  var nouText = document.createTextNode('Això és la capçalera')

  nouElement.appendChild(nouText);
  contenidor.appendChild(nouElement);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/pEodaE?editors=1010.

L'argument que es passa en invocar el mètode és una cadena de text amb l'etiqueta (*tag*), és a dir, si l'argument és `h1` es crearan les etiquetes HTML`<h1></h1>`. Aquesta etiqueta és important perquè també permet fer cerques a l'arbre de tots els nodes amb la mateixa etiqueta, per exemple: cercar tots els títols principals (`h1`), tots els enllaços (`a`)...

Fixeu-vos que no cal afegir els nous nodes al document immediatament, es pot crear una branca a la memòria (en aquest cas el node referenciat per `nouelement`, que conté el node de text) i, una vegada afegits tots els nodes, afegir-la a l'arbre.

Aquesta és la millor manera de fer-ho, perquè si s'afegeixen els nodes al document d'un en un, el navegador renderitzarà l'arbre una vegada per cada node. En canvi, si s'afegeix tota la branca, només ha de renderitzar l'arbre de nou un cop.

1.4.2 Cerca d'elements simple: 'getElementsByClassName', 'getElementsByTagName' i 'getElementById'

Aquests mètodes permeten fer **cerques d'elements** al document. Cal recalcar que es tracta d'elements (implementen la interfície `Elements`) i no de qualsevol node. És a dir, es poden cercar tots els paràgrafs d'una pàgina però no els nodes de text que contingui.

L'atribut `id` de qualsevol element d'un document sempre ha de ser únic.

Els mètodes `getElementsByClassName` i `getElementsByTagName` retornen una llista de nodes de tipus element, mentre que `getElementById` retorna sempre un únic node de tipus element.

Cadascun realitza la cerca segons un aspecte diferent:

- **getElementsByClassName**: nodes de tipus element que continguin la classe passada com a argument.
- **getElementsByTagName**: nodes de tipus element amb l'etiqueta passada com a argument; per exemple: `p`, per obtenir una llista de paràgrafs, o `h1` per obtenir totes les capçaleres de primer nivell.
- **getElementById**: node de tipus element amb l'atribut `id` que coincideix amb el valor passat com a argument. Permet accedir de forma més directa a qualsevol element, però requereix afegir, abans, l'atribut `id` als elements que s'han de manipular.

```
<h1>Primera capçalera</h1>
<ul>
  <li>Primer element de la primera llista</li>
  <li>Segon element de la primera llista</li>
  <li>Tercer element de la primera llista</li>
</ul>
<h1 class="secundari">Segona capçalera</h1>
<p id="contingut" class="secundari paragraf">Un paràgraf</p>
<ul>
  <li>Primer element de la segona llista</li>
  <li>Segon element de la segona llista</li>
  <li>Tercer element de la segona llista</li>
</ul>

<script>
  var elementsSecundaris = document.getElementsByClassName('secundari');

  console.log('Contingut textual dels elements amb classe="secundari":');
  for (var i = 0; i < elementsSecundaris.length; i++) {
    console.log(elementsSecundaris[i].textContent);
  }

  var elementsLlista = document.getElementsByTagName('li');
  console.log('Contingut textual dels elements de la llista (<li></li>):');
  for (i = 0; i < elementsLlista.length; i++) {
```

```
    console.log(elementsLlista[i].textContent);
  }

  var elementContingut = document.getElementById('contingut');
  console.log('Contingut textual del element amb id="contingut": ' +
      elementContingut.textContent);
</script>
```

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

Fixeu-vos que el mètode `getElementsByClassName` retorna dos elements, ja que comprova totes les classes de cada element, de manera que tant `secundari` com `secundari paràgraf` són coincidències vàlides.

També cal destacar que en el cas de `getElementsByTagName` es retornen tots els elements amb l'etiqueta `li` del document, és a dir, inclou tant els elements de la primera llista com els de la segona.

Com és d'esperar, és possible fer modificacions sobre aquests nodes, per exemple per moure'ls o eliminar-los. En cas de voler eliminar-los, es pot presentar un problema inesperat: com que en eliminar el node canvia la seva posició a la llista, es produirà un error en intentar accedir als elements posteriors. Una possible solució seria la següent: fer servir un bucle amb `while` que elimini el primer element de la llista i es repeteixi fins que la longitud de la llista sigui 0.

```
<h1>Primera capçalera</h1>
<ul>
  <li>Primer element de la primera llista</li>
  <li>Segon element de la primera llista</li>
  <li>Tercer element de la primera llista</li>
</ul>
<h1 class="secundari">Segona capçalera</h1>
<p id="contingut" class="secundari paragraf">Un paràgraf</p>
<ul>
  <li>Primer element de la segona llista</li>
  <li>Segon element de la segona llista</li>
  <li>Tercer element de la segona llista</li>
</ul>

<script>
  var elementsLlista = document.getElementsByTagName('li');

  while (elementsLlista.length>0) {
    elementsLlista[0].parentNode.removeChild(elementsLlista[0]);
  }
</script>
```

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

1.4.3 Cerca d'elements amb selectors ('querySelector' i 'querySelectorAll')

Aquests mètodes permeten fer una cerca específica d'elements, passant com a argument una cadena de text amb un selector CSS. La diferència entre tots dos és que el primer només retorna el primer element coincident, mentre que el segon retorna una llista amb totes les coincidències.

Selectors CSS

Podeu trobar informació detallada sobre els selectors CSS en l'enllaç següent: goo.gl/o8ELvX.

Entre els selectors que es poden utilitzar hi ha:

- `#identificador`: que contingui `id=‘‘identificador’’`.
- `.nom_classe`: que contingui `class=‘‘nom_classe’’`.
- `element`: que sigui un element amb l'etiqueta `element`.
- `element1 element2`: que sigui un element amb l'etiqueta `element2` descendent d'un element amb l'etiqueta `element1`. Cal destacar que **no cal que sigui descendent directe**.
- `element1>element2`: que sigui un element amb el tag `element2` **descendent directe** d'un element amb l'etiqueta `element1`.
- `[type=‘‘button’’]`: elements que tinguin l'atribut `type` i el seu valor sigui exactament `button`.
- ':first-child' (exemple de pseudoclasse): elements que siguin el primer fill de qualsevol altre element.
- 'element:first-child' (exemple de pseudoclasse): elements que siguin el primer fill de l'element pare.

Un element és **descendent directe** d'un altre quan el seu pare és aquest element.

Cal remarcar que el nombre de possibles selectors i les seves combinacions és molt gran i el seu estudi queda fora de l'abast d'aquest mòdul.

A continuació podeu veure un exemple d'ús del mètode `querySelectorAll`:

```
<h1>Primera capçalera</h1>
<ul>
  <li>Primer element de la primera llista</li>
  <li>Segon element de la primera llista</li>
  <li>Tercer element de la primera llista</li>
</ul>
<h1 class="secundari">Segona capçalera</h1>
<p id="contingut" class="secundari paragraf">Un paràgraf</p>
<ul>
  <li>Primer element de la segona llista</li>
  <li>Segon element de la segona llista</li>
  <li>Tercer element de la segona llista</li>
</ul>

<script>
  console.log('-- Capçaleras h1 amb classe secundari --');
  var elementsTitols = document.querySelectorAll('h1.secundari');
  for (var i=0; i<elementsTitols.length; i++) {
    console.log(elementsTitols[i].textContent);
```

```
  }

  console.log('—— Tots els elements de la primera llista ——');
  var elementsLlista = document.querySelectorAll('li');
  for (var i=0; i<elementsLlista.length; i++) {
    console.log(elementsLlista[i].textContent);
  }

  console.log('—— Primers elements de les llistes ——');
  var elementsPrimers = document.querySelectorAll('li:first-child');
  for (var i=0; i<elementsPrimers.length; i++) {
    console.log(elementsPrimers[i].textContent);
  }
</script>
```

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

Com es pot apreciar, en el primer cas se seleccionen tots els elements amb l'etiqueta `h1` amb la classe `secundari`. Així doncs, només troba una coincidència, ja que tot i que hi ha dos elements amb l'etiqueta `h1` i dos elements amb la classe `secundari`, només hi ha un element que compleixi les dues condicions.

En el segon cas s'han seleccionat tots els elements amb l'etiqueta `li`, de manera que el seu efecte resultat és idèntic al que retornaria el mètode `document.getElementsByTagName('li');`.

A l'últim cas, en canvi, s'ha afegit al selector `:first-child`, de manera que en lloc de retornar tots els elements de les llistes, només retorna el primer element de cadascuna.

1.4.4 L'extensió 'HTMLDocument'

En el cas dels documents HTML, a banda de totes les propietats i mètodes de la interfície `Document`, s'aplica l'extensió `HTMLDocument`, que afegeix algunes propietats i mètodes extres que no es troben quan es treballa amb documents XML o SVG. Entre les més destacables es troben:

- **activeElement**: referència a l'element enfocat.
- **cookie**: cadena de text que permet consultar o modificar les galetes del document.
- **forms** (només lectura): llista d'elements de tipus `form` (formularis).
- **images** (només lectura): llista d'elements de tipus `img` (imatges).
- **getElementsByName(String name)**: retorna una llista d'elements amb el nom (atribut `name`, habitualment utilitzat en formularis) passat com a argument.
- **getSelection()**: retorna el text seleccionat al document.

- **write(String text)**: escriu el text al document.
- **writeln(String text)**: escriu el text al document i afegeix un salt de línia.

1.4.5 Altres propietats: 'body', 'documentElement' i 'forms'

La interfície `document` ofereix també dues propietats que permeten accedir directament als elements `body` i `html`: `body` i `documentElement` respectivament.

L'accés a l'element `body` és especialment útil per afegir la detecció de l'*event* `load`, ja que això permet detectar quan s'ha acabat de carregar el DOM. En cas contrari, es poden produir errors, per exemple si es volen realitzar modificacions al DOM abans que aquest s'hagi acabat de carregar completament.

```
document.body.onload = function() {
  console.log("DOM carregat");
}
```

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

Fixeu-vos que `onload` és una drecera per l'*event* `load`, així doncs es podria haver fet servir també el mètode `addEventListener` per afegir la detecció de l'*event*.

Per altra banda, l'extensió `HTMLDocument` afegeix `forms` a la propietat, que permet accedir a una llista d'objectes que conté la informació de tots els formularis del document i, al seu torn, cadascun d'aquests formularis permet accedir als seus elements:

```
<form id="primer">
  <input type="text" />
</form>

<form id="segon">
  <input type="text" />
</form>

<script>
  console.log('Nombre de formularis al document: ', document.forms.length);

  for (var i = 0; i < document.forms.length; i++) {
    console.log('id del formulari amb índex' + i + ':', document.forms[i].id);
  }
</script>
```

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

Tot i que en determinats casos pot ser útil accedir a aquesta propietat per recórrer tots els formularis del document, és poc habitual. Per una banda, els documents no acostumen a contenir múltiples formularis i, per altra banda, quan s'ha de treballar amb múltiples formularis s'acostuma a requerir un control més precís (s'obtenen a través del seu `id`).

1.5 La interfície 'Element'

Aquesta interfície, com també `Document`, és herència de `Node` i d'`EventTarget`, per tant, permet manipular els elements i gestionar esdeveniments. A més a més, afegeix mètodes per enregistrar i desenregistrar observadors del mateix element.

Mentre que `Document` facilita la creació i manipulació del document que es mostra al navegador, la interfície `Element` permet manipular els elements concrets, manipulant les classes que els afecten, la llista d'atributs i afegint o eliminant detectors d'*events* específics.

Propietats d'"Element'

Podeu trobar més informació sobre la interfície `Element` en l'enllaç següent: goo.gl/0bCjXA.

1.5.1 Informació bàsica dels elements: 'tagName', 'className', 'classList' i 'id'

Aquestes propietats permeten conèixer la informació bàsica d'un element. Fixeu-vos que aquesta és la que ens permet fer una cerca simple des del document, a través de la seva etiqueta, de les classes o de l'`id`. Vegem-les detingudament:

- **tagName** (només lectura): nom de l'etiqueta d'aquest element.
- **className**: cadena de caràcters separats per espais que inclou tots els noms de classes que afecten l'element (per exemple, en aplicar un full d'estils CSS). Es pot modificar, per tant, permet afegir noves classes i establir una nova cadena de text com a valor.
- **classList** (només lectura): llista amb el nom de les classes que es pot recórrer com un *array*.
- **id**: identificador únic de l'element dintre del document.

Com es pot apreciar a la llista anterior, no es pot modificar ni el `tagName` ni la llista de classes, però aquesta darrera pot modificar-se amb dos mètodes que aquesta proporciona:

- **add(String classe)**: afegeix a l'element la classe indicada pel paràmetre. Si ja hi era a l'element, no fa res. Té una versió que admet diferents classes, separades per comes, i que fa exactament el mateix, però per a cada classe que rep com a paràmetre.
- **remove(String nomClasse)**: suprimeix de l'element la classe indicada pel paràmetre. Si l'element no té la classe que es demana suprimir, no fa res. Té també una versió que admet diferents classes, separades per comes, i que suprimeix de l'element cadascuna de les classes rebudes com a paràmetre de la mateixa manera que es feia amb una.

Podem veure com utilitzar-les al següent exemple:

```
<style>
  .vermell{
    color:red;
  }

  .negreta{
    font-weight:bold;
  }
</style>
<html>
  <p id="para">paràgraf de prova</p>
</html>
<script>
var paragraf=document.getElementById("para");

    paragraf.classList.add("vermell");
    paragraf.classList.add("negreta");

    alert("Hem afegit els estils vermell i negreta.\n\n Ara eliminarem l'estil
        vermell.\n\n Prem una tecla per continuar");

    paragraf.classList.remove("vermell");

    alert("Hem suprimit l'estil vermell.\n\n Ara eliminarem també negreta.\n\n
        Prem una tecla per continuar.");
    paragraf.classList.remove("negreta");
</script>
```

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

El següent exemple treballa amb les propietats `tagName` i `className`:

```
<h1 class="principal">Primera capçalera</h1>
<p id="contingut" class="principal paragraf" title="Aqui va un paràgraf">Un par
    àgraf</p>

<script>
  var elementH1 = document.getElementsByTagName('h1')[0];
  var elementP = document.getElementsByTagName('p')[0];

  elementH1.tagName = 'H2';

  console.log('-- Tag dels elements --');
  console.log(elementH1.tagName);
  console.log(elementP.tagName);

  console.log('-- Classes dels elements --');
  console.log(elementH1.className);
  console.log(elementP.className);

  console.log('-- Modificació de els classes dels elements --');
  elementH1.className = elementH1.className + ' ampliat';
  elementP.className = 'canvi per noves classes ';

  console.log('-- Classes dels elements ampliats--');
  console.log(elementH1.className);
  console.log(elementP.className);

  console.log('-- Llista de classes del paràgraf --');
  for (var i = 0; i < elementP.classList.length; i++) {
    console.log("Classe " + i + ":" + elementP.classList[i]);
  }

  console.log('-- Identificadors dels elements --');
  console.log('Capçalera: ' + elementH1.id);
```

```
  console.log('Paràgraf: ' + elementP.id);
  elementH1.id = 'actualitzat';
  elementP.id = elementP.id + ' actualitzat';

  console.log('-- Identificadors dels elements actualitzats--');
  console.log('Capçalera: ' + elementH1.id);
  console.log('Paràgraf: ' + elementP.id);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/xExYXV?editors=1012.

Fixeu-vos que tant en el cas de `className` com a `id` es pot assignar com a valor una cadena de text, sigui una de nova o concatenant el valor anterior. En tots dos casos el valor per defecte que prenen és una cadena buida, de manera que es poden concatenar sense fer cap consideració especial.

Relació entre "className" i "classList"

El següent exemple mostra com, tal com és d'esperar, en modificar `className` també es modifica `classList`. Concretament, implementa els mètodes `afegirClasse` i `eliminarClasse`, que fan el mateix, respectivament, que `classList.add` i `classList.remove`, però sense utilitzar aquests. En la pràctica, aquest codi només té utilitat si es fa una aplicació per navegadors antics, que no proporcionen la propietat `classList`.

Una opció seria dividir la cadena de caràcters assignada a `className` fent servir el mètode `split`, i seguidament recórrer l'*array* generat per fer l'acció desitjada, però això no és necessari perquè la propietat `classList` ja conté aquest *array*:

```
<style>
  .principal {
    background-color: grey;
  }

  .vermell {
    color: red;
  }

  .important {
    font-weight: bold;
  }
</style>

<h1 class="principal">Primera capçalera</h1>
<p id="contingut" class="principal paragraf" title="Aquí hi va un
     paràgraf">Un paràgraf</p>

<script>
  var afegirClasse = function(element, classe) {
    var trobada = false;
    for (var i=0; i <element.classList.length; i++) {
      if (element.classList[i] == classe) {
        trobada = true; // Ja es troba a l'element, no cal afegir
            -la
      }
    }
    if (!trobada) {
      element.className += ' ' + classe;
    }
  }

  var eliminarClasse = function(element, classe) {
    var nouClassName = '';
```

```
    for (var i=0; i<element.classList.length; i++) {
      if (element.classList[i] != classe) {
        nouClassName += element.classList[i] + ' ';
      }
    }
    element.className = nouClassName;
  }

  var elementH1 = document.getElementsByTagName('h1')[0];
  var elementP = document.getElementsByTagName('p')[0];

  afegirClasse(elementH1, 'vermell');
  afegirClasse(elementH1, 'vermell');
  console.log(elementH1.className); // No es duplica

  eliminarClasse(elementH1, 'principal');
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/VKwQGz?editors=1111.

Com es pot comprovar, la funció `afegirClasse` afegeix la classe i evita possibles repeticions, i la funció `eliminarClasse` elimina la classe sense haver d'editar manualment la cadena de text i crea una nova cadena de text a partir de la llista de classes que no coincideixin amb la que es vol eliminar.

Per simplificar més la utilització d'aquestes funcions es poden afegir al prototipus d'`Element`, d'aquesta manera són accessibles directament per tots els elements. Modifiqueu el codi JavaScript de l'exemple anterior pel següent:

```
Element.prototype.afegirClasse = function(classe) {
  var trobada = false;
  for (var i = 0; i < this.classList.length; i++) {
    if (this.classList[i] == classe) {
      trobada = true; // Ja es troba a l'element, no cal afegir-
          la
    }
  }
  if (!trobada) {
    this.className += ' ' + classe;
  }
}

Element.prototype.eliminarClasse = function(classe) {
  var nouClassName = '';
  for (var i = 0; i < this.classList.length; i++) {
    if (this.classList[i] != classe) {
      nouClassName += this.classList[i] + ' ';
    }
  }
  this.className = nouClassName;
}

var elementH1 = document.getElementsByTagName('h1')[0];
var elementP = document.getElementsByTagName('p')[0];

elementH1.afegirClasse('vermell');
elementH1.afegirClasse('vermell');
console.log(elementH1.className); // No es duplica

elementH1.eliminarClasse('principal');
```

Podeu veure aquest exemple en l'enllaç següent: www.http://codepen.io/ioc-daw-m06/pen/yaLvWL?editors=1111.

Fixeu-vos que la funcionalitat és idèntica, però en lloc d'haver d'invocar una funció i passar cada vegada l'element que s'ha de modificar i la classe, ara es

poden invocar directament a partir de l'element concret que es vulgui manipular: `elementH1.afegirClasse('vermell');`.

1.5.2 Obtenció dels descendents com HTML: 'innerHTML'

Aquesta propietat és molt interessant perquè permet obtenir i establir el codi HTML dels nodes descendents de l'element. És a dir, en lloc de crear una branca d'un arbre i afegir-la com a nodes, és possible assignar a aquesta propietat el codi HTML que correspondria:

```
<div id="contingut"></div>

<script>
  var contingut = document.getElementById('contingut');
  contingut.innerHTML = '<h1>Això és una capçalera</h1>';
</script>
```

Podeu veure aquest exemple en l'enllaç següent: http://codepen.io/ioc-daw-m06/pen/VKwdGz?editors=1010.

Com es pot apreciar, és fàcil de fer servir, però la cadena de codi HTML es pot complicar ràpidament si s'hi han d'afegir múltiples elements:

```
<div id="contingut"></div>

<script>
  var contingut = document.getElementById('contingut');
  contingut.innerHTML = '<h1>Això és una capçalera</h1><p>I això un paràgraf
      amb un <a href="#">enllaç</a>.</p>';
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/NRWzJJ?editors=1010.

Fixeu-vos que només s'ha afegit un paràgraf amb un enllaç a continuació de la capçalera, però el codi és molt menys entenedor i és més difícil detectar els errors.

Per altra banda, com que es tracta d'una cadena de text, es pot manipular de la mateixa manera, per exemple: fent servir expressions regulars per fer canvis al text o concatenant valors per generar la cadena i seguidament assignant-la com a propietat:

```
<div id="contingut">Carregant dades...</div>

<script>
var estudiants = ['Josep', 'Maria', 'Carles', 'Montserrat'];
var codiHtml = '';

codiHtml += '<ul>';

for (var i = 0; i < estudiants.length; i++) {
  codiHtml += '<li>' + estudiants[i] + '</li>';
}

codiHtml += '</ul>';

```

```
document.getElementById('contingut');
contingut.innerHTML = codiHtml;
</script>
```

Podeu veure aquest exemple en l'enllaç següent:codepen.io/ioc-daw-m06/pen/yaLEdL?editors=1010.

Cal destacar que tot el contingut de l'element és reemplaçat, per aquesta raó no es mostra el text "Carregant dades": és reemplaçat per la llista de noms una vegada s'executa el codi JavaScript.

Per descomptat, és possible consultar la propietat per obtenir el codi HTML corresponent als descendents del node:

```
<div id="contingut">
  <ul>
    <li>Josep</li>
    <li>Maria</li>
    <li>Carles</li>
    <li>Montserrat</li>
  </ul>
</div>

<script>
  document.getElementById('contingut');
  console.log(contingut.innerHTML);
</script>
```

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

1.5.3 Atributs: 'attributtes', 'getAttribute', 'removeAttribute', 'setAttribute'

La interfície `Elements` ofereix una propietat per consultar la llista completa d'atributs i tres mètodes per afegir, actualitzar o eliminar un atribut complet.

La propietat `attributes` (de només lectura) permet consultar la llista completa d'atributs. Els atributs es retornen com un objecte que es pot recórrer com si es tractés d'un *array*. Per consultar el nom i el valor de cadascun d'aquests atributs es pot consultar la propietat `name` i `value` respectivament:

```
<h1 id="titol" class="vermell principal">Aquesta és la capçalera</h1>

<script>
var titol = document.getElementById('titol');

for (var i = 0; i < titol.attributes.length; i++) {
  console.log(titol.attributes[i].name + ': ' + titol.attributes[i].value);
}
</script>
```

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

Per altra banda, a diferència de la llista de noms de classes, amb els atributs no cal fer una implementació pròpia dels mètodes d'addició i eliminació, ja que la mateixa interfície els inclou:

- **getAttribute**: retorna el valor de l'atribut.
- **removeAttribute**: elimina l'atribut.
- **setAttribute**: estableix el valor de l'atribut.

```
<style>
  .vermell {
    color:red;
  }
</style>

<h1 id="titol" class="vermell principal">Aquesta és la capçalera</div>

<script>
  var titol = document.getElementById('titol');

  titol.setAttribute('title', 'Això es mostra en posar el cursor a sobre');
  titol.removeAttribute('class');
  console.log('L\'identificador és: ' + titol.getAttribute('id'));
</script>
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/QKWBvw?editors=1111.

Cal destacar que a HTML5 es va afegir la possibilitat de definir atributs personalitzats; d'aquesta manera és possible, per exemple, emmagatzemar identificadors de productes o tipus especials d'elements:

```
<div data-id-producte="42">...</div>
```

Tot i que els navegadors admeten la definició d'atributs amb qualsevol nom, l'especificació requereix que incloguin el prefix `data-` i que no continguin majúscules, per exemple: `data-id-producte`. Aquests nous atributs es poden fer servir després per obtenir el seu valor o per seleccionar els elements fent servir selectors.

1.5.4 Modificar estils CSS: 'style'

A HTML és possible modificar els estils concrets d'un element a través de l'atribut `style`. Normalment els estils aplicats d'aquesta forma tenen prioritat sobre qualsevol altre que afecti l'element.

Tot i així, quan es vol tractar amb aquest atribut des de JavaScript no és tan simple com caldria esperar, perquè s'han de tenir en compte dos aspectes fonamentals:

- El valor final de la propietat no és l'indicat a l'atribut, sinó el calculat, que pot estar modificat per altres regles de diferents orígens (fulls d'estil, navegador...).

Propietat 'style'

Podeu trobar més informació sobre la propietat `style` en l'enllaç següent: goo.gl/BYa2IC.

- En consultar el valor de l'atribut `style` d'un element des de JavaScript, s'obté una col·lecció i no una cadena de text.

Per altra banda, si es vol treballar amb el contingut textual de la propietat es pot fer o bé fent servir el mètode `setAttribute` o modificant la propietat `cssText` de la col·lecció retornada per `style`:

```
<p id="paragraf1" style="color: red; font-weight: bold">Paràgraf 1</p>
<p id="paragraf2" style="color: red; font-weight: bold">Paràgraf 2</p>

<script>
  var paragraf1 = document.getElementById('paragraf1');
  var paragraf2 = document.getElementById('paragraf2');

  paragraf1.setAttribute('style', 'color:green;');
  paragraf2.style.cssText = 'color: blue';

  console.log(paragraf1.style.cssText);
  console.log(paragraf2.style.cssText);
</script>
```

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

Com es pot apreciar, tant fent servir el mètode `setAttribute` com assignant el valor directament a la propietat `cssText`, el valor de la propietat és substituït; per aquesta raó cap dels dos textos es mostra en negreta.

És a dir, en cas de voler modificar alguna de les propietats d'estil CSS s'hauria de crear una nova cadena de text amb el contingut correcte i reemplaçar el valor de la propietat `style.cssText`.

Una manera més adequada de treballar amb els estils concrets és fer-ho a través de la col·lecció. Fixeu-vos en l'exemple següent, partint d'un element HTML que conté dos estils (canvi de color a vermell i font gruixuda), s'elimina el color, es modifica el gruix de la font i s'afegeix un nou estil per augmentar-ne la mida. Per facilitar la reutilització i fer-lo més entenedor s'ha aplicat el disseny descendent i s'ha creat una funció per a cada acció:

```
<p id="paragraf" style="color: red; font-weight: bold">Paràgraf</p>

<script>
  var mostrarEstils = function (element) {
    var estil = element.style;
    var css = window.getComputedStyle(element, null);

    for (var i=0; i<estil.length; i++) {
      console.log (estil[i] + ':' +css[estil[i]] + ";");
    }
  }

  var afegirEstil = function (element, clau, valor) {
    element.style[clau] = valor;
  }

  var eliminarEstil = function (element, clau) {
    element.style[clau]=null;
  }

  var actualitzarEstil = function (element, clau, valor) {
```

```
    afegirEstil(element, clau, valor);
  }

  var paragraf = document.getElementById('paragraf');

  console.log("-- Estil original --");
  mostrarEstils(paragraf);
  afegirEstil(paragraf, 'font-size', '30px');
  eliminarEstil(paragraf, 'color');
  actualitzarEstil(paragraf, 'font-weight', 'lighter');

  console.log("-- Estil modificat --");
  mostrarEstils(paragraf);
</script>
```

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

El primer que us cridarà l'atenció és la complexitat que té mostrar els estils, ja que s'ha d'invocar el mètode `window.getComputedStyle` per obtenir un objecte amb la informació de tots els valors calculats per l'objecte.

S'ha de tenir en compte que no només s'apliquen els estils de la propietat, sinó que s'apliquen també els propis del navegador i els dels fulls d'estil carregats. És possible obtenir només els valors aplicats al propi estil, però aquests valors no són finals: pot ser que una altra regla CSS ho hagi sobreescrit, per exemple, si s'ha aplicat el modificador `!important` a una regla que l'afecti.

Una vegada s'ha obtingut la llista de propietats calculades, es recorren els noms dels estils establerts a la propietat `style` com si es tractés d'un *array*, ja que la propietat `style` és una col·lecció que té una propietat `length`, i cada element es troba referenciat per un enter que es fa servir com a índex.

D'aquesta manera, combinant els valors calculats amb els estils aplicats a l'element, es pot mostrar una llista dels valors reals aplicats a l'element.

Alternativament, imitant el comportament de la biblioteca jQuery, és possible modificar la interfície `Element` per afegir al seu prototip els mètodes `css` i `mostrarElements`, de manera que tots els elements tinguin accés a aquesta funcionalitat:

```
<p id="paragraf" style="color: red; font-weight: bold">Paràgraf</p>

<script>
  Element.prototype.mostrarEstils = function() {
    var estil = this.style;
    for (var i = 0; i < estil.length; i++) {
      console.log(estil[i] + ':' + this.css(estil[i]) + ";");
    }
  }

  Element.prototype._mostrarEstil = function(clau) {
    var css = window.getComputedStyle(this, null);
    return css[clau];
  }

  Element.prototype.css = function(clau, valor) {
    this.style[clau] = valor;

    return this._mostrarEstil(clau);
  }

```

```
  var paragraf = document.getElementById('paragraf');

  console.log("-- Estil original --");
  paragraf.mostrarEstils();

  paragraf.css('font-size', '30px');
  paragraf.css('color', '');
  paragraf.css('font-weight', 'lighter');

  console.log("-- Estil modificat --");
  paragraf.mostrarEstils();
</script>
```

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

Com es pot apreciar, a partir d'aquesta implementació es molt fàcil accedir i modificar els estils d'un node, tant si és per afegir-hi un estil nou, modificar-ne algun o eliminar-lo.

1.5.5 Relacions entre elements: 'previousElementSibling', 'nextElementSibling', 'firstElementChild' i 'lastElementChild'

Tot i que la interfície `Element` deriva de `Node` i, consegüentment, disposa de les propietats `previousSibling` i `nextSibling`, aquestes no retornen els elements, sinó els nodes. És a dir, inclouen tot tipus de nodes, com per exemple els nodes de tipus text que es generen en afegir un salt de línia o un tabulador.

En canvi, gràcies a `previousElementSibling` i `nextElementSibling` (totes dues propietats són de només lectura), es poden consultar els nodes anterior i posterior directament:

```
<ul>
  <li>Primer element de la llista</li>
  <li id="central">Segon element de la llista</li>
  <li>Tercer element de la llista</li>
</ul>

<script>
  var central = document.getElementById('central');

  console.log('El contingut de l\'element anterior és: ', central.
      previousElementSibling.textContent);

  console.log('El contingut de l\'element següent és: ', central.
      nextElementSibling.textContent);
</script>
```

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

Com es pot comprovar, a la consola es mostra correctament el contingut textual del primer i el tercer element, ignorant els nodes de tipus text.

De la mateixa manera, es pot accedir al primer i l'últim element a través de les propietats `firstElementChild` i `lastElementChild`:

```
<ul id="llista">
  <li>Primer element de la llista</li>
  <li>Segon element de la llista</li>
  <li>Tercer element de la llista</li>
</ul>

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

  console.log('El contingut del primer element és: ', central.firstElementChild
      .textContent);

  console.log('El contingut del darrer element és: ', central.lastElementChild.
      textContent);
</script>
```

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

1.5.6 Cerca d'elements descendents simple: 'getElementsByClassName', 'getElementsByTagName'

La funcionalitat dels mètodes `getElementsByClassName` i `getElementsByTagName` és idèntica a la que proporciona la interfície `Document` amb la peculiaritat que la cerca només es fa entre els descendents del mateix element.

Cal destacar que no s'inclou un mètode `getElementById`, ja que els identificadors són únics i no és rellevant si és o no descendent d'un element concret, com es pot comprovar en l'exemple següent:

```
<ul>
  <li>Primer element de la llista</li>
  <li id="central">
    <ul>
      <li>Primer element de la subllista</li>
      <li>Segon element de la subllista</li>
    </ul>
  </li>
  <li>Tercer element de la llista</li>
</ul>

<script>
  var central = document.getElementById('central');
  var elementsSubLlista = central.getElementsByTagName('li');

  for (var i=0; i<elementsSubLlista.length; i++) {
    console.log(elementsSubLlista[i].textContent);
  }
  </script>
```

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

Cal destacar que tot i que inicialment s'invoca `getElementById` per obtenir la referència a l'element amb identificador `central`, a continuació se cerquen els

elements a partir d'aquest i no pas de `document`. Així doncs, es limita la cerca als seus descendents i el resultat obtingut són els elements de la subllista.

1.5.7 Cerca d'elements descendents amb selectors: 'querySelector', 'querySelectorAll'

Igual que els mètodes `getElementsByTagName` i `getElementsByClassName`, la funcionalitat d'aquests és molt similar a la dels mètodes amb el mateix nom de la interfície `Document`, però en aquest cas la cerca també es limita als elements descendents del mateix element.

En l'exemple següent podeu comprovar com es fa una selecció a partir d'un element, de manera que només retorna la llista d'elements descendents i no pas totes les coincidències del document:

```
<ul>
  <li>Primer element de la llista</li>
  <li id="central">
    <ul data-quantaitat="0">
      <li data-quantitat="0">Primer element de la subllista</li>
      <li data-quantitat="19">Segon element de la subllista</li>
      <li data-quantitat="0">Tercer element de la subllista</li>
    </ul>
  </li>
  <li data-quantitat="0">Tercer element de la llista</li>
</ul>

<script>
var central = document.getElementById('central');
var elementsSubLlista = central.querySelectorAll('li[data-quantitat="0"]');

for (var i = 0; i < elementsSubLlista.length; i++) {
  console.log(elementsSubLlista[i].textContent + ' (data-quantitat=' +
      elementsSubLlista[i].getAttribute('data-quantitat') + ')');
}
</script>
```

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

Fixeu-vos que s'ha fet servir un atribut propi, `data-quantitat`, i que el selector `li['data-quantitat=''0'']` ha filtrat els resultats de manera que només es mostren a la consola el primer i tercer element de la subllista, sense incloure ni l'element `ul` ni el segon element, que té com a valor de l'atribut `19`. A més a més, com és d'esperar, tampoc no s'hi ha inclòs el tercer element de la llista, ja que no és descendent de l'element `central`.

1.6 Integració de la detecció d'events amb el DOM

La programació d'*events* permet interactuar amb l'aplicació web i afegeix la detecció d'*events* als elements (o al mateix document), de manera que l'aplicació pot reaccionar de diferents maneres, per exemple: afegint files a una taula, eliminant-les, modificant les classes CSS per canviar-ne els colors o aplicar efectes...

Podeu trobar més informació sobre els *events* a la unitat "Programació d'events".

S'ha de tenir en compte que quan es dispara un *event* en un element fill, si no s'especifica el contrari, aquest travessa tots els nodes pare, de manera que finalment són rebuts pel node arrel, que és `document`.

És a dir, si s'implementa la gestió de l'esdeveniment `submit` al document, quan s'enviï un formulari (es dispara l'*event* `submit`) contingut en un element fill, el document detectarà aquest *event*.

A continuació podeu trobar una llista dels *events* més destacables a l'hora de tractar amb un document o element:

- **blur**: es dispara quan es perd el focus.
- **focus**: es dispara quan l'element rep el focus.
- **click**: es dispara quan es produeix un clic sobre el document.
- **dblclick**: es dispara quan es produeix un doble clic sobre el document.
- **keydown**, **keypress**, **keyup**: es disparen quan es detecta que s'ha premut una tecla.
- **load**: es dispara quan es completa la càrrega del document.
- **scroll**: es dispara quan es desplaça verticalment o horitzontalment el document.
- **submit**: es dispara quan s'envia un formulari.
- **input**: es dispara quan es modifica el valor d'un camp d'entrada (per exemple, un quadre de text).
- **change**: es dispara quan es modifica el valor d'un camp d'entrada però, a diferència d'`input`, no es dispara l'*event* per cada canvi produït, només es dispara en determinades circumstàncies, com per exemple en canviar el focus a un altre element.

Això permet, per exemple, controlar quan es fa clic sobre qualsevol element d'una pàgina web, sense haver de gestionar aquest *event* en cadascun dels elements que formen la pàgina.

Quan es parla d'*events*, el terme 'escoltar' (*listening* en anglès) fa referència a la detecció de l'*event* al document o element.

La interfície ofereix mètodes per "escoltar" quan es dispara un *event* (`addEventListener`), per deixar d'escoltar-lo (`removeEventListener`) i tot un

seguit de dreceres en forma de mètodes que permeten realitzar les dues accions per *events* concrets.

La utilització de les dreceres comporta més limitacions, ja que no es pot afegir més d'una funció per a cada *event*, però pot simplificar el codi en casos molt simples (per exemple, per controlar la càrrega de documents o imatges).

1.6.1 Afegir detecció d'esdeveniments: 'addEventListener'

El mètode `addEventListener` permet enregistrar una funció que serà invocada quan es dispari l'*event* passat com a argument a l'element, per exemple quan es faci un doble clic sobre ell o es detecti un canvi en un camp del formulari.

En l'exemple següent podeu veure com s'ha afegit un comptador de caràcters que indica el nombre de caràcters introduït en una àrea de text i que, a més a més, en modifica el color segons els següents paràmetres:

- Menys de 100 caràcters o més de 156: text en vermell, el text és massa curt o massa llarg.
- Menys de 150 caràcters: text en taronja, el text és curt.
- Entre 150 i 156: text en verd, text amb llargària òptima.

```
<style>
  label,
  small {
    display: block;
  }

  .massa_curt_o_llarg {
    color: red;
  }

  .curt {
    color: orange;
  }

  .correcte {
    color: green;
  }
</style>

<div>
  <label>Introduex el contingut de la descripció meta</label>
  <textarea name="meta_description" id="meta_description" placeholder="
      Introdueix fins a 156 caràcters" / cols="80" rows="3"></textarea>
  <small>Nombre de caràcters: <span id="comptador" className="curt">0</span></
      small>
</div>

<script>
  var camp = document.getElementById('meta_description'),
    comptador = document.getElementById('comptador');

  var actualitzarComptador = function() {
    var llargaria = camp.value.length
    comptador.textContent = llargaria;
```

```
    if (llargaria > 156 || llargaria < 100) {
      comptador.className = 'massa-curt-o-llarg';
    } else if (llargaria < 150) {
      comptador.className = 'curt';
    } else {
      comptador.className = 'correcte';
    }
  }

  camp.addEventListener('input', actualitzarComptador);
</script>
```

L'etiqueta 'meta-description'

L'etiqueta `meta-description` conté el text que mostren els cercadors a tall de resum quan surt el llistat de pàgines trobades. La llargària màxima recomanada de l'etiqueta per millorar el SEO és de 156 caràcters.

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

Com podeu apreciar, afegir un detector d'*events* és molt senzill, ja que només cal invocar el mètode `addEventListener` passant com a arguments el nom de l'*event* al qual es vol reaccionar i la funció que s'ha d'executar quan es dispari: `camp.addEventListener('input', actualitzarComptador);`.

A continuació podeu comprovar, modificant el mateix exemple, la diferència més important entre els *events* `input` i `change`. Mentre que el primer es dispara cada vegada que es modifica el contingut del `textarea`, el segon només es dispara en canviar el focus (per exemple, fent clic en un altre punt del document):

```
camp.addEventListener('change', actualitzarComptador);
```

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

Alternativament, es podria fer servir la drecera `oninput`, modificant la línia en què s'afegeix la detecció d'*events* per la següent:

```
camp.oninput = actualitzarComptador;
```

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

Aquest format és més concís, però podria provocar errors si es fes servir aquest codi en un projecte més avançat en el qual s'haguessin de fer diferents accions quan es detectessin canvis (per exemple, guardar un esborrany): si s'ha fet servir aquest format en tots dos casos, només s'executaria la funció afegida en darrer lloc.

Per altra banda, les dreceres permeten incrustar el codi JavaScript directament al codi HTML com si es tractés d'un atribut:

```
<button onclick="escriureMissatge();">Escriure missatge a la consola</button>

<button onclick="alert('Alerta!');">Mostra una alerta</button>

<script>
  var escriureMissatge=function() {
    console.log('Aquesta funció ha estat invocada des del botó')
  }
</script>
```

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

Com es pot apreciar, es poden invocar tant funcions pròpies com funcions predefinides de JavaScript.

1.6.2 Eliminar detecció d'esdeveniments: 'removeEventListener'

En alguns casos pot interessar-vos eliminar un detector d'*events*, per exemple, per deshabilitar un botó d'enviar formulari si els camps de text són buits o no s'ha passat la validació.

En el següent exemple podeu veure com s'afegeixen i eliminen els detectors dinàmicament, de manera que només es pot clicar sobre el quadre blau:

```
<style>
div {
  width: 100px;
  height: 100px;
  background-color: grey;
  float: left;
  margin: 5px;
}

span {
  padding: 0 3px;
}

.seleccionat {
  background-color: green;
  color:white;
}

.proper {
  background-color: blue;
    color:white;
}
</style>

<p>El quadre clicable es mostra de color <span class="proper">blau</span> i l'ú
     ltim quadre clicat de color <span class="seleccionat">verd</span></p>
<div id="a"></div>
<div id="b"></div>
<div id="c"></div>

<script>
var quadreA = document.getElementById('a');
var quadreB = document.getElementById('b');
var quadreC = document.getElementById('c');

var seleccionarA = function() {
  quadreA.className = 'seleccionat';
  quadreB.className = 'proper';
  quadreC.className = '';
  quadreA.removeEventListener('click', seleccionarA);
  quadreB.addEventListener('click', seleccionarB);
  console.log(listenerB);
}

var seleccionarB = function() {
  quadreB.className = 'seleccionat';
  quadreC.className = 'proper';
```

```
  quadreA.className = '';
  quadreB.removeEventListener('click', seleccionarB);
  quadreC.addEventListener('click', seleccionarC);
}

var seleccionarC = function() {
  quadreC.className = 'seleccionat';
  quadreA.className = 'proper';
  quadreB.className = '';
  quadreC.removeEventListener('click', seleccionarC);
  quadreA.addEventListener('click', seleccionarA);
}

var inicialitzar = function() {
  quadreA.className = 'proper';
  listenerA = quadreA.addEventListener('click', seleccionarA);
}

inicialitzar();
</script>
```

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

Com es pot apreciar, només cal indicar el nom de l'*event* (el seu tipus) i la funció que s'ha d'eliminar. És a dir, en cas que hi hagués múltiples funcions lligades al mateix *event*, només s'eliminaria la indicada.

En cas de fer servir dreceres, per eliminar un detector només cal assignar el valor `nul` al mètode:

```
var quadreA = document.getElementById('a');
var quadreB = document.getElementById('b');
var quadreC = document.getElementById('c');

var seleccionarA = function() {
  quadreA.className = 'seleccionat';
  quadreB.className = 'proper';
  quadreC.className = '';
  quadreA.onclick = null;
  quadreB.onclick = seleccionarB;
}

var seleccionarB = function() {
  quadreB.className = 'seleccionat';
  quadreC.className = 'proper';
  quadreA.className = '';
  quadreB.onclick = null;
  quadreC.onclick = seleccionarC;
}

var seleccionarC = function() {
  quadreC.className = 'seleccionat';
  quadreA.className = 'proper';
  quadreB.className = '';
  quadreC.onclick = null;
  quadreA.onclick = seleccionarA;
}

var inicialitzar = function() {
  quadreA.className = 'proper';
  quadreA.onclick = seleccionarA;
}

inicialitzar();
```

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

Com es pot veure, independentment de si es fan servir dreceres o no, crear una funció per a cada element que cal detectar no és gens pràctic. A l'exemple anterior si és volgués augmentar el nombre de quadres, caldria modificar les tres funcions existents i afegir-n'hi una altra. A més a més, totes les funcions són pràcticament idèntiques. Aquest és un senyal molt clar que cal refactoritzar el codi.

Una opció és aprofitar les relacions entre nodes: en aquest cas el proper node sempre és el següent, i un cop no n'hi ha més es pot navegar directament al primer fill del node pare:

```
<style>
div#quadres div {
  width: 100px;
  height: 100px;
  background-color: grey;
  float: left;
  margin: 5px;
}

span {
  padding: 0 3px;
}

#quadres div.seleccionat {
  background-color: green;
  color:white;
}

#quadres div.proper {
  background-color: blue;
    color:white;
}
</style>

<p>El quadre clicable es mostra de color <span class="proper">blau</span> i l'ú
    ltim quadre clicat de color <span class="seleccionat">verd</span></p>
<div id="quadres">
  <div></div>
  <div></div>
  <div></div>
  <div></div>
  <div></div>
  <div></div>
</div>

<script>
var quadres = document.getElementById('quadres');
var seleccionat;
var proper;

var seleccionar = function() {
  // S'elimina l'estil i el detector del seleccionat anterior
  console.log(seleccionat, proper);
  seleccionat.className = '';
  seleccionat.removeEventListener('click', seleccionar);

  // S'actualitza el seleccionat al proper quadre
  seleccionat = proper;
  seleccionat.className = 'seleccionat';

  // Es determina el proper quadre i s'afegeix el detector
  proper = seleccionat.nextElementSibling;
  if (!proper) {
```

```
    proper = quadres.firstElementChild;
  }
  proper.className = 'proper';
  proper.addEventListener('click', seleccionar);
}

var inicialitzar = function() {
  seleccionat = quadres.firstElementChild;
  proper = seleccionat.nextElementSibling;
  seleccionar();
}

inicialitzar();
</script>
```

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

Com es podeu veure, s'ha modificat el codi HTML per afegir un element contenidor (`quadres`); ja no cal fer servir un `id` per a cada quadre i s'ha refinat el codi CSS perquè els estils només han d'afectar els elements `div` descendents de `quadres`.

Ara bé, si no es poden fer servir les relacions entre elements, es pot aprofitar el context en el qual s'invoquen les funcions. S'ha de tenir en compte que dintre de la funció invocada, el seu context és el node al qual s'ha lligat el detector. Així doncs, és possible generalitzar aquestes funcions per fer servir el node com a context.

En l'exemple següent podeu comprovar com es genera aleatòriament quin serà el proper element, només cal reemplaçar el codi JavaScript pel següent:

```
var quadres = document.getElementById('quadres');
var darrerSeleccionat;

var seleccionar = function() {
  var proper;

  // S'elimina l'estil i el detector del seleccionat anterior
  if (darrerSeleccionat) {
    darrerSeleccionat.className = '';
  }

  // S'actualitza l'element actual
  this.removeEventListener('click', seleccionar);
  this.className = 'seleccionat';
  darrerSeleccionat = this;

  // Es determina el proper quadre i s'afegeix el detector
  proper = seleccionarQuadreAleatori();
  proper.className = 'proper';
  proper.addEventListener('click', seleccionar);
}

var seleccionarQuadreAleatori = function() {
  var maxIndex = quadres.childNodes.length - 1;

  // S'han de descartar el darrer node seleccionat per no repetir i els nodes
      de text
  do {
    index = Math.floor(Math.random() * (maxIndex + 1));
    proper = quadres.childNodes[index];
  } while (proper.nodeName != 'DIV' || proper === darrerSeleccionat);

  return proper;
```

```
}

seleccionar();
```

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

Fixeu-vos que en aquest exemple no cal controlar quin és l'element actual ni el proper globalment, només es guarda la referència al darrer seleccionat per poder esborrar-lo quan es cliqui el següent element i evitar que el mateix element es repeteixi. També s'ha eliminat la funció d'inicialització: no cal inicialitzar cap variable, només cal invocar `seleccionar`.

Per altra banda, s'ha afegit la funció `seleccionarQuadreAleatori`, que genera un nombre entre 0 i el nombre de nodes descendents de l'element pare menys 1 (recordeu que els índexs dels *arrays* es compten començant per 0) i es van comprovant els nodes fins que se'n troba un que sigui de tipus `DIV` i no sigui el darrer seleccionat. Heu de tenir en compte que `childNodes` retorna una llista de nodes i, consegüentment, s'inclouen també els nodes amb text que es troben entre els elements: salts de línia i tabulacions.

1.7 Cas pràctic: generador de factures

A continuació trobareu un exemple pràctic, un generador de factures. Hi podeu veure com s'integren les característiques principals de les interfícies del DOM, com pot ser la cerca, la consulta, la creació i l'eliminació d'elements.

Primerament, afegiu el següent codi HTML i CSS en un nou document per crear l'estructura del generador de factures, que està dividit en dues seccions:

- Una àrea d'entrada de dades formada per 3 quadres de text i un botó.
- Una taula que contindrà les línies de la factura i el peu que mostra els totals.

```
<style>
  div {
    margin-bottom: 15px;
  }

  label {
    width: 100px;
    display: inline-block;
  }

  input {
    width: 100px;
  }

  th {
      width:136px;
  }

  td {
```

```
    text-align:right;
  }

  td:first-child {
    text-align:left;
  }

  tfoot th {
    text-align:left;
  }

  tfoot th:last-child {
    text-align:right;
  }

  tfoot, thead, tbody, table {
    border: 1px solid black;
  }

  thead, tfoot {
    background-color: grey;
  }

  tbody tr:hover {
    background-color:#2897E8;
  }
</style>

<div>
  <h3>Introducció de productes a la factura</h3>
  <label for="producte">Producte:</label>
  <input type="text" id="producte" />
  <label for="quantitat">Quantitat:</label>
  <input type="number" id="quantitat" value="0" />
  <label for="preu-unitari">Preu unitari:</label>
  <input type="number" id="preu-unitari" value="0" />
  <button id="afegir">Afegir</button>
</div>


<table rules="groups">
  <thead>
    <tr>
      <th>Producte</th>
      <th>Quantitat</th>
      <th>Preu unitari</th>
      <th>Preu total</th>
      <th>Accions</th>
    </tr>
  </thead>
  <tfoot>
    <tr>
      <th colspan="4">Base imposable</th>
      <th><span id="base-imposable">0</span>€</th>
    </tr>
    <tr>
      <th colspan="4" data-iva="0.21">IVA 21%</th>
      <th><span id="iva">0</span>€</th>
    </tr>
    <tr>
      <th colspan="4">Total factura</th>
      <th><span id="total">0</span>€</th>
    </tr>
  </tfoot>
  <tbody>
  </tbody>
</table>
```

Quant al codi HTML, cal destacar que s'ha afegit la propietat `id` per als següents elements:

- Entrada de text `producte` per introduir el nom del producte.
- Entrada de text `quantitat` per introduir la quantitat de productes facturats.
- Entrada de text `preu-unitari` per introduir el preu unitari del producte.
- Botó `afegir` per afegir la nova línia a la factura.
- Element `base-imposable` per mostrar la base imposable total.
- Element `iva` per mostrar el total calculat per l'IVA.
- Element `total` per mostrar el total a pagar.

Addicionalment, s'ha afegit l'atribut propi `data-iva` amb el valor `0.21` de manera que és possible modificar el percentatge d'IVA que s'ha d'aplicar a la factura modificant, directament, aquest atribut. Així doncs, és possible modificar aquest valor per aplicar, per exemple, un 7% d'IVA (0.07).

El codi CSS només s'utilitza per donar estil al document. S'ha aplicat un fons de color gris a la capçalera i al peu de la taula. A més a més, per destacar quina és la fila a la qual afectaran les accions, aquesta canvia de color quan el cursor és a sobre.

El codi JavaScript, tot i que a primera vista pot semblar complicat, és força senzill. No és res més que l'aplicació de les funcionalitats i l'accés a les propietats dels elements i el document.

Primer de tot hi ha la funció `inicialitzar`, que és l'encarregada d'inicialitzar tots els elements necessaris de l'aplicació. Aquesta funció és invocada automàticament quan es completa la càrrega del DOM:

```
1 var inicialitzar = function() {
2   var boto = document.getElementById('afegir');
3   boto.onclick = afegirLinia;
4 }
5
6 // Inicialització de l'aplicació quan es carregui el DOM
7 document.body.onload = inicialitzar;
```

Dins de la funció `inicialitzar` se cerca l'element amb identificador `afegir`, que correspon al botó i assigna a l'*event* `click` la funció `afegirLinia`, fent servir la drecera `onclick`. Seguidament, s'assigna aquesta funció a l'*event* `load` de `document.body`, de manera que aquesta funció s'executa una vegada s'acaba de carregar el DOM.

Fixeu-vos que si en canvieu l'ordre, l'aplicació no funcionarà correctament. Primer s'ha de declarar la funció o el valor que s'assignarà a `document.body.onload`, que serà `null`.

La següent funció és `afegirLinia`, que és invocada quan es fa clic al botó `afegir`:

```
1 var afegirLinia = function() {
2   var nomProducte = document.getElementById('producte').value;
3   var quantitatProducte = document.getElementById('quantitat').value;
```

```
  var preuUnitari = document.getElementById('preu_unitari').value;
  var totalProducte = quantitatProducte * preuUnitari;

  afegirFilaTaula(nomProducte, quantitatProducte, preuUnitari, totalProducte);
  recalcularTotal();
  netejarLinia();
}
```

Com es pot apreciar, en la implementació d'aquest programa s'ha aplicat el disseny descendent, de manera que s'ha dividit el codi en múltiples funcions específiques.

En primer lloc, s'obté el valor dels tres camps de text. Com que no cal guardar la referència a l'element, s'accedeix directament a la seva propietat `value`: `document.getElementById('producte').value`. Així, doncs, nomProducte contindrà el valor de l'element amb identificador `producte`. Una manera menys concisa d'implementar-lo seria la següent:

```
var elementNomProducte = document.getElementById('producte');
var nomProducte = elementNomProducte.value;
```

Una vegada s'han obtingut la `quantitatProducte` i el `preuUnitari`, es calcula el preu total i es passen aquests quatre valors a la funció `afegirFilaTaula` per afegir les dades a la taula. Seguidament s'invoca `recalcularTotal` (que recalcula la base imposable, l'IVA i l'import total), i finalment `netejarLinia`, per eliminar les dades de l'entrada de text.

La funció `afegirFilaTaula`, tot i que és la més llarga de l'aplicació, és molt simple: bàsicament es repeteix el mateix codi per a cada columna. Es podria haver fet servir un *array* alternativament, però per simplificar el codi i fer-lo més entenedor s'ha optat per duplicar-lo:

```
var afegirFilaTaula = function(nomProducte, quantitatProducte, preuUnitari,
    totalProducte) {
  var cosTaula = document.querySelector('tbody');

  var fila = document.createElement('tr');

  var col1 = document.createElement('td');
  var col2 = document.createElement('td');
  var col3 = document.createElement('td');
  var col4 = document.createElement('td');
  var col5 = document.createElement('td');

  col1.innerHTML = nomProducte + ' (detall)';
  col2.innerHTML = quantitatProducte;
  col3.innerHTML = preuUnitari + '€';
  col4.innerHTML = totalProducte + '€';
  col5.innerHTML = '(eliminar)';

  col1.addEventListener('click', mostrarDetall);
  col5.addEventListener('click', eliminarFila);

  fila.appendChild(col1);
  fila.appendChild(col2);
  fila.appendChild(col3);
  fila.appendChild(col4);
  fila.appendChild(col5);

  cosTaula.appendChild(fila);
}
```

Primer de tot, s'obté la referència al cos de la taula (l'element `tbody`) amb el mètode `document.querySelector`, perquè aquest permet accedir directament a l'element com si fos un selector CSS i sempre retorna un únic element (al contrari de `document.getElementsByTagName`, que retorna un *array*).

A continuació, es creen els nous elements que s'afegiran:

- Una nova fila: `document.createElement('tr')`.
- Cinc noves cel·les, una per a cada columna: `document.createElement('td')`.

Una vegada s'han creat les columnes (recordeu que es tracta de nodes de tipus `element`) s'assigna el contingut corresponent a cadascuna d'elles, i s'estableix la seva propietat `innerHTML`.

Tot seguit, s'afegeix la detecció de l'*event* `click` a les columnes 1 i 5, de manera que en clicar sobre les columnes s'invoca les funcions `mostrarDetall` o `eliminarFila`, respectivament.

Amb els continguts i la detecció d'*events* afegida, només resta afegir les columnes a la fila (`fila.appendChild(col1)`) i, seguidament, la fila a la taula (`cosTaula.appendChild(fila)`).

El següent mètode, `recalcularTotal`, és l'encarregat de recalcular els totals que es mostren al peu de la taula. Aquest mètode és més complex, perquè es treballa amb selectors, accés als elements descendents i germans i s'accedeix a propietats.

```
1  var recalcularTotal = function() {
2    var files = document.querySelectorAll('tbody tr');
3    var valorBase = 0;
4
5    for (var i=0; i<files.length; i++) {
6      var columnaUltima = files[i].lastElementChild;
7      var columnaPenultima = columnaUltima.previousElementSibling;
8      var valorTotalFila = parseFloat(columnaPenultima.textContent);
9      valorBase += valorTotalFila;
10   }
11
12   var elementBase = document.getElementById('base-imposable');
13   elementBase.innerHTML = valorBase;
14
15   var elementPercentatgeIVA = document.querySelector('[data-iva]');
16   var valorPercentatgeIVA = elementPercentatgeIVA.getAttribute('data-iva');
17   var valorIVA = parseFloat(valorPercentatgeIVA * valorBase);
18
19   var elementIVA = document.getElementById('iva');
20   elementIVA.innerHTML = valorIVA;
21
22   var elementTotal = document.getElementById('total');
23   elementTotal.innerHTML = valorBase + valorIVA;
24 }
```

Fixeu-vos que per obtenir totes les files s'invoca `document.querySelectorAll('tbody tr')`, de manera que s'obtenen totes les files (elements de tipus `tr`) que es trobin dins de l'element `tbody`. És a dir, s'exclouen les files de la capçalera i del peu de la taula. Cal destacar

que a diferència de `querySelector`, que retorna només un únic element, `querySelectorAll` retorna una col·lecció d'elements.

Seguidament es recorre aquesta col·lecció per extreure la informació de cada línia de la factura i calcular-ne el valor base total. Com que la columna que interessa és la cinquena, s'ha optat per accedir a l'últim element descendent de la fila (`files[i].lastElementChild`) i a continuació a l'element anterior (`columnaUltima.previousElementChild`).

Per assegurar que el valor que s'afegeix és un nombre real (és a dir, no s'interpreta com una cadena de text), s'invoca la funció `parseFloat` i aquest valor s'afegeix a la variable `valorBase`. Una vegada acaba l'execució del bucle, la variable `valorBase` correspondrà al total de la base imposable que s'afegeix al document a través de la propietat `innerHTML` de l'element amb l'identificador `base-imposable`.

El següent pas és calcular l'IVA. Per fer-ho primer s'obté la referència a l'element que conté la propietat `data-iva` invocant `document.querySelector('[data-iva]')`. Seguidament, s'invoca `elementPercentatgeIVA.getAttribute('data-iva')` per obtenir el valor d'aquest atribut que indica el percentatge que s'ha d'aplicar. Una vegada s'ha calculat, s'afegeix al document assignant-lo a la propietat `innerHTML` de l'element `iva`.

Amb tots dos valors calculats, només queda sumar-los i afegir-los al document com a contingut de l'element `total`.

La funció `netejarLinia` és molt més simple que l'anterior. De la mateixa manera com s'obtenen els valors a la funció `afegirLinia`, s'estableixen els valors per defecte de les entrades de text: una cadena buida per al producte i 0 per a la quantitat i el preu unitari.

```
1  var netejarLinia = function() {
2    document.getElementById('producte').value = '';
3    document.getElementById('quantitat').value = 0;
4    document.getElementById('preu_unitari').value = 0;
5  }
```

Seguidament es troba el codi que s'invoca en detectar-se l'*event* `click` sobre la primera i l'última columna. En el primer cas només cal destacar que s'obtenen tots els elements amb l'etiqueta `td` del node pare, i després es recorre aquesta llista per construir el text que es mostrarà com a detall:

```
1  var mostrarDetall = function() {
2    var missatge = 'Detall de la factura:\n';
3    var elementsFila = this.parentNode.getElementsByTagName('td');
4    var etiquetes = ['Producte', 'Quantitat', 'Preu unitari', 'Preu total'];
5
6    for (var i = 0; i < elementsFila.length; i++) {
7      missatge += '\t' + etiquetes[i] + ': ' + elementsFila[i].textContent + '\n'
           ;
8    }
9
10   alert(missatge);
11 }
```

Fixeu-vos que es fa servir `this` per accedir a les propietats pròpies de l'element clicat. Tot i que totes les cel·les de la primera columna criden aquesta funció, el context en el qual s'executen sempre és l'element concret on s'ha disparat l'*event*.

Per acabar, hi ha el codi de la funció `eliminarFila`. En primer lloc s'elimina la mateixa fila i per fer-ho s'accedeix al node pare del pare. És a dir, es travessa el node ascendentment fins a l'element `tr` (la fila), seguidament, fins a l'element `tbody` que la conté, i a partir d'aquest, s'elimina. Seguidament, s'invoca la funció `recalcularTotal()` per actualitzar els valors del peu de la taula.

```
var eliminarFila = function() {
  this.parentNode.parentNode.removeChild(this.parentNode);
  recalcularTotal();
}
```

A continuació podeu trobar el codi JavaScript complet del generador de factures:

```
var inicialitzar = function() {
  var boto = document.getElementById('afegir');
  boto.onclick = afegirLinia;
}

// Inicialització de l'aplicació quan es carregui el DOM
document.body.onload = inicialitzar;

var afegirLinia = function() {
  var nomProducte = document.getElementById('producte').value;
  var quantitatProducte = document.getElementById('quantitat').value;
  var preuUnitari = document.getElementById('preu_unitari').value;
  var totalProducte = quantitatProducte * preuUnitari;

  afegirFilaTaula(nomProducte, quantitatProducte, preuUnitari, totalProducte);
  recalcularTotal();
  netejarLinia();
}

var afegirFilaTaula = function(nomProducte, quantitatProducte, preuUnitari,
    totalProducte) {
  var cosTaula = document.querySelector('tbody');

  var fila = document.createElement('tr');

  var col1 = document.createElement('td');
  var col2 = document.createElement('td');
  var col3 = document.createElement('td');
  var col4 = document.createElement('td');
  var col5 = document.createElement('td');

  col1.innerHTML = nomProducte + ' (detall)';
  col2.innerHTML = quantitatProducte;
  col3.innerHTML = preuUnitari + '€';
  col4.innerHTML = totalProducte + '€';
  col5.innerHTML = '(eliminar)';

  col1.addEventListener('click', mostrarDetall);
  col5.addEventListener('click', eliminarFila);

  fila.appendChild(col1);
  fila.appendChild(col2);
  fila.appendChild(col3);
  fila.appendChild(col4);
  fila.appendChild(col5);

  cosTaula.appendChild(fila);
}

```

```
var recalcularTotal = function() {
  var files = document.querySelectorAll('tbody tr');
  var valorBase = 0;

  for (var i=0; i<files.length; i++) {
    var columnaUltima = files[i].lastElementChild;
    var columnaPenultima = columnaUltima.previousElementSibling;
    var valorTotalFila = parseFloat(columnaPenultima.textContent);
    valorBase += valorTotalFila;
  }

  var elementBase = document.getElementById('base-imposable');
  elementBase.innerHTML = valorBase;

  var elementPercentatgeIVA = document.querySelector('[data-iva]');
  var valorPercentatgeIVA = elementPercentatgeIVA.getAttribute('data-iva');
  var valorIVA = parseFloat(valorPercentatgeIVA * valorBase);

  var elementIVA = document.getElementById('iva');
  elementIVA.innerHTML = valorIVA;

  var elementTotal = document.getElementById('total');
  elementTotal.innerHTML = valorBase + valorIVA;
}

var netejarLinia = function() {
  document.getElementById('producte').value = '';
  document.getElementById('quantitat').value = 0;
  document.getElementById('preu-unitari').value = 0;
}


var mostrarDetall = function() {
  var missatge = 'Detall de la factura:\n';
  var elementsFila = this.parentNode.getElementsByTagName('td');
  var etiquetes = ['Producte', 'Quantitat', 'Preu unitari', 'Preu total'];

  for (var i = 0; i < elementsFila.length; i++) {
    missatge += '\t' + etiquetes[i] + ': ' + elementsFila[i].textContent + '\n'
        ;
  }

  alert(missatge);
}

var eliminarFila = function() {
  this.parentNode.parentNode.removeChild(this.parentNode);
  recalcularTotal();
}
```

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

2. Programació amb el DOM i la biblioteca jQuery

Tot i que les versions anteriors de jQuery eren compatibles amb navegadors antics, les noves versions només admeten navegadors actuals.

La biblioteca jQuery és una de les més populars, ja que inclou tota una sèrie de **funcionalitats** que són bàsiques per a qualsevol aplicació web amb un mínim de complexitat; en concret:

- D'una banda, simplifiquen la realització de tasques com gestionar el DOM, els *events* o les peticions asíncrones.
- De l'altra, evita que hàgiu de fer servir implementacions alternatives (per a navegadors actuals) per característiques que potser no estan definides de forma estricta a l'especificació del W3C.

Concretament, pel que fa a la manipulació del DOM, jQuery ofereix una gran quantitat de mètodes que ajuden a gestionar els elements del DOM i permeten crear-ne de nous o modificar els actuals d'una manera més simple. Cal destacar especialment la potència del seu **sistema de selecció**, que va molt més enllà del que permeten les interfícies del DOM.

A més a més, donada la seva popularitat, és fàcil trobar centenars de **connectors lliures**, així com privatius, per ampliar les seves funcionalitats o per afegir fàcilment nous elements, com poden ser galeries d'imatges, carrusels... És a dir, en molts casos podreu recórrer a components completament desenvolupats i testats per utilitzar-los directament o adaptar-los a les vostres necessitats en lloc de començar des de zero.

2.1 Introducció a jQuery

El primer pas a l'hora d'utilitzar jQuery és **carregar la biblioteca**. La forma de fer-ho variarà segons l'entorn en el qual s'hagi d'utilitzar i les vostres preferències quant al seu allotjament.

Cal recordar que no és segur manipular el DOM abans que acabi de carregar, per aquesta raó jQuery inclou els seus propis sistemes per detectar-ho i permetre que el desenvolupador escrigui codi, que començarà a executar una vegada finalitzi la càrrega.

El primer que trobareu en començar a treballar amb jQuery és que cal distingir entre la **funció** `jQuery` (habitualment substituïda pel símbol `$`) i els **objectes** `jQuery`, que són generats o bé per la mateixa funció o per la crida d'algun mètode sobre altres objectes `jQuery`.

Generalment treballareu amb la funció `jQuery` per obtenir un objecte que contingui els elements seleccionats, i a partir de llavors les crides a tots els mètodes

les fareu sobre l'objecte que s'ha de manipular, per exemple, cridant el mètode `addClass` per afegir una classe nova als elements.

Cal recordar que tot i que l'estructura dels programes que fan servir jQuery poden semblar molt diferents dels programes en JavaScript, es tracta del mateix llenguatge. Segurament la major part de l'estranyesa es deu a la utilització del símbol `$` com a àlies de la funció `jQuery` i la utilització d'interfícies fluides.

Gairebé tots els mètodes dels objectes jQuery tenen un comportament fluid, és a dir, retornen el mateix objecte sobre el qual s'han invocat. Així doncs us podeu trobar amb estructures semblants a aquesta:

```
$('p')
  .find('span')
  .addClass('vermell')
  .addClass('negreta')
  .css('font-size', '18px')
  .css('background-color', 'green');
```

Cal destacar que el mètode `find` retorna un nou objecte `jQuery` que inclouria només els elements de tipus `span` descendents dels paràgrafs seleccionats inicialment i, per tant, els mètodes següents s'invocarien a partir d'aquest nou objecte i no de l'original.

Fixeu-vos també que només l'última línia inclou el punt i coma; és a dir, aquest codi es podria interpretar com si fos una sola línia:

```
$('p').find('span').addClass('vermell').addClass('negreta').css('font-size', '
    18px').css('background-color', 'green');
```

Com es pot apreciar, el primer cas és més fàcil de llegir i d'entendre. Una altra manera d'escriure el mateix codi seria guardant el valor de la selecció inicial (un objecte `jQuery` en una variable):

```
var $paragraf = $('p');
var $span = $paragraf.find('span');

$span.addClass('vermell');
$span.addClass('negreta');
$span.css('font-size', '18px');
$span.css('background-color', 'green');
```

Tot i que el resultat és el mateix en tots tres casos, quan es treballa amb jQuery, per convenció s'acostuma a utilitzar el format del primer exemple, que genera un codi més concís i més clar.

2.1.1 Carregar la biblioteca jQuery

A l'hora d'incloure la biblioteca jQuery a les vostres aplicacions, el primer pas és carregar la biblioteca. Per fer-ho, les dues opcions més habituals són descarregar la biblioteca i afegir-la juntament amb la resta de fitxers JavaScript de la vostra aplicació o fer que els clients la descarreguin a través d'un CDN.

Xarxes de lliurament de continguts (CDN)

Un CDN és una xarxa de servidors localitzats en diferents punts geogràfics però amb els mateixos continguts, de manera que quan es rep una petició del fitxer aquest és enviat des del servidor més proper fins a l'usuari. D'aquesta manera, s'augmenta la velocitat de la resposta. Podeu trobar més informació sobre les xarxes de lliurament de continguts en l'enllaç següent: www.ca.wikipedia.org/wiki/Xarxa_de_lliurament_de_continguts.

Per descarregar la biblioteca jQuery heu de visitar el web següent: www.jquery.com/download/. Entre d'altres descàrregues, hi trobareu la versió més actual comprimida i sense comprimir. Només cal descarregar-ne un dels dos, segons les vostres necessitats.

La diferència entre fer servir la versió comprimida o sense comprimir és que la primera ocupa menys espai però és inintel·ligible: com que són fitxers de text pla, no es tracta exactament d'una compressió sinó d'una *minimització*. Així doncs, a l'hora de desplegar l'aplicació s'ha de fer servir la versió comprimida, mentre que durant el desenvolupament és recomanable utilitzar la versió sense *minimitzar* per poder depurar millor el codi.

La **minimització** (de l'anglès *minification*) consisteix a eliminar tots els espais i salts de línies i substituir els noms de variables i funcions per altres amb el mínim nombre de caràcters possibles; així, `totalProductes = preUnitari * quantitatProducte;` es convertiria en alguna cosa semblant a `a=b*c`.

Una vegada descarregada la biblioteca podeu afegir-la al vostre programa com qualsevol altre fitxer de codi JavaScript. Suposant que esteu treballant amb la versió 3.1 (aquesta és la versió que es farà servir per als exemples) i que l'heu copiat a un directori anomenat `js` dintre del vostre projecte, el codi seria el següent:

```
1 <script src="js/biblioteques/jquery-3.1.0.js"></script>
```

Per **evitar el bloqueig de la pàgina** es recomana que tots els fitxers amb codi JavaScript, propi o biblioteques, s'afegeixin al final de la pàgina abans de tancar l'element `body`, en comptes de fer-ho al principi dintre de l'element `head`.

Les eines d'automatització per treballar amb JavaScript requereixen tenir instal·lat Node.js i s'utilitzen a través del terminal.

Un avantatge de treballar directament amb la biblioteca és que podeu utilitzar eines d'automatització com Gulp o Grunt per concatenar tots els fitxers JavaScript en un de sol i a continuació minimitzar-lo abans de fer el desplegament de l'aplicació. D'aquesta manera es redueix el nombre de fitxers i s'optimitza el temps de descàrrega de la pàgina.

D'altra banda, si feu servir un CDN, no cal descarregar el fitxer: el fitxer es descarregarà per a cada client des del servidor.

```
1 <script src="//code.jquery.com/jquery-3.1.0.js"></script>
```

Independentment del sistema que feu servir, sempre s'ha de carregar la biblioteca abans d'utilitzar-la, perquè si ho feu en l'ordre contrari, és molt possible que es produeixi un error quan la vulgueu fer servir.

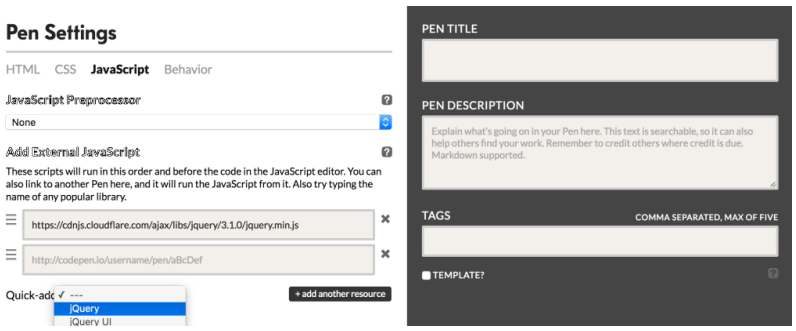
2.1.2 Carregar jQuery a CodePen

Atès que l'ús de la biblioteca jQuery està tan estès, moltes plataformes d'edició de codi en línia, com CodePen, ofereixen la possibilitat d'incloure'l directament sense haver d'afegir la càrrega manualment.

> **Fitxers externs a CodePen**
>
> A CodePen és possible afegir tant algunes de les biblioteques de JavaScript més conegudes com els vostres propis fitxers externs amb codi JavaScript. A més a més, es poden fer servir preprocessadors com Babel o TypeScript per utilitzar ES6 o TypeScript en lloc de JavaScript.

Per accedir a aquestes opcions en els vostres propis *pens* (nom que rep cada demostració a CodePen) només heu de fer clic al botó *Settings*, a continuació, en el panell de configuració, feu clic a la pestanya *JavaScript* i a la llista desplegable *Quick-add*, que es troba a la part inferior seleccioneu **jQuery**. Podeu veure les opcions de configuració de CodePen a la figura 2.1

FIGURA 2.1. Opcions de configuració de CodePen



Cal destacar que quan s'afegeix la biblioteca d'aquesta manera, no cal afegir el codi per carregar-la perquè es carrega automàticament; per tant, es pot procedir a treballar directament amb la biblioteca. Per aquesta raó, la càrrega de la biblioteca no es mostra en cap dels exemples: es carrega automàticament gràcies a la configuració del *pen*.

2.1.3 Primers passos amb jQuery

A banda d'indicar que s'ha de carregar la biblioteca, us heu d'assegurar que el document (més concretament el DOM) ha carregat completament. En cas contrari, és possible que es produeixin errors. Per aquesta raó la primera línia que trobareu habitualment en el codi que fa servir jQuery és el següent:

```
1 $(document).ready(function() {
2   // Aquí va el nostre codi
3   console.log('Llestos!');
4 });
```

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

D'aquesta manera us assegureu que és segur començar a treballar amb el document, ja que el DOM estarà completament carregat.

Recordeu que en cas de fer la prova en un fitxer local, heu d'incloure la càrrega de la biblioteca. En primer lloc, haureu d'afegir la càrrega de la biblioteca i, seguidament, el vostre codi d'inicialització. Així doncs, si feu servir el CDN de jQuery, el vostre codi quedaria així:

```
<script src="//code.jquery.com/jquery-3.1.0.js"></script>

<script>
  $(document).ready(function() {
    // Aquí va el nostre codi
    console.log('Llestos!');
  });
</script>
```

> L'efecte d'utilitzar el mètode `ready` de jQuery és molt similar al detector de l'*event* `load` de JavaScript però no és el mateix. **No s'han d'utilitzar mai tots dos sistemes al mateix temps** perquè són incompatibles, en cas de fer servir jQuery, sempre s'ha d'utilitzar el mètode `ready` que facilita la biblioteca.

Atès que a JavaScript les funcions són objectes de primera classe, es poden referenciar fent servir variables i passar-les com a arguments a altres funcions.

Fixeu-vos en la primera línia: `$(document).ready()`. Aquesta línia fa dues accions diferenciades: d'una banda, es crida la funció `jQuery` fent servir la drecera `$`, i de l'altra, sobre l'objecte retornat s'invoca el mètode `ready` passant com a argument una funció, que serà executada una vegada es carregui el DOM.

És a dir, es podria haver escrit `jQuery(document).ready()` i hauria estat el mateix. Recordeu que el símbol `$` és utilitzable com a nom de variable, tot i que no és recomanable fer-lo servir, perquè algunes biblioteques (com aquesta) el fan servir amb fins específics.

D'altra banda, per convenció, sí que s'acostuma a prefixar el nom de les variables que emmagatzemen objectes de tipus `jQuery` amb el símbol `$`. Així doncs, si una variable conté un objecte jQuery que conté, al seu torn, una llista de matrícules, és correcte que el seu nom sigui `$matricules`.

La funció que es passa com a argument –i serà cridada en carregar el DOM–, no cal que estigui definida com a funció anònima, es podria haver passat una variable que referenciés una funció:

```
var inicialitzacio = function() {
    console.log('Llestos!');
}

$(document).ready(inicialitzacio);
```

Podeu veure aquest exemple en el següent enllaç: www.codepen.io/ioc-daw-m06/pen/zKvrdx?editors=0012.

Fixeu-vos en un detall important: en cas que la funció estigui referenciada per una variable en lloc d'estar declarada directament, s'ha d'assignar la funció a aquesta variable abans d'invocar el mètode `ready`. Si no es fa així, el valor de la variable es passarà com a `null` (a causa del *hoisting*: internament, la declaració de les variables es mou al principi del bloc però sense el valor assignat).

En canvi, si es passa el nom de la funció, l'ordre no importa perquè s'inclou el cos de la funció quan es produeix el *hoisting*:

```
$(document).ready(inicialitzacio);

function inicialitzacio() {
    console.log('Llestos!');
}
```

Podeu veure aquest exemple en el següent enllaç: www.codepen.io/ioc-daw-m06/pen/BLoGdx?editors=0012.

Cal destacar que només cal incloure dins de la funció d'inicialització les crides a les funcions; la declaració, en canvi, es pot realitzar en qualsevol altre lloc (per exemple, en altres fitxers) sempre que us assegureu que l'ordre de declaració i assignació és el correcte.

Per exemple, es podrien invocar múltiples funcions dintre de la funció `inicialització` mentre que es tingui la garantia que aquestes han estat invocades una vegada s'ha completat la càrrega del DOM (com succeeix quan s'invoca `inicialització`):

```
var inicialitzacio = function() {
  mostrarMissatge('DOM carregat');
  iniciarAplicacio();
}

var mostrarMissatge = function(missatge) {
  console.log(missatge);
}

var iniciarAplicacio = function() {
  console.log("Iniciant l'aplicació...");
}

$(document).ready(inicialitzacio);
```

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

2.1.4 L'objecte 'jQuery'

Quan es parla de jQuery cal diferenciar entre la **biblioteca**, la **funció** i els **objectes** de tipus `jQuery`.

En el primer cas, es fa referència a la biblioteca en conjunt, per exemple: 'carregar jQuery' significa que s'ha de carregar la biblioteca, ja sigui com un fitxer de codi JavaScript o a través d'un CDN.

En el segon cas, fa referència a la funció `jQuery`, que pot invocar-se com a `jQuery()` o `$()`. Aquesta funció permet, d'una banda, crear objectes de tipus `jQuery`, i de l'altra, accedir a una sèrie de mètodes genèrics, com per exemple `each`, que permet iterar sobre un *array*.

En el tercer cas, un objecte de tipus `jQuery` fa referència a un objecte retornat per la funció, sigui generat o com a resultat d'una cerca i que hi pot tenir associats un o més elements. En qualsevol cas es tracta sempre com una col·lecció d'elements.

Una peculiaritat dels objectes de tipus `jQuery` és que quan s'invoquen els seus mètodes afecten, generalment, tota la col·lecció. Per exemple, si un objecte conté una col·lecció amb tots els enllaços de la pàgina i s'hi afegeix la detecció de l'*event* `click`, aquesta detecció afectarà tots els elements d'una tacada, no cal recórrer la col·lecció i afegir-lo d'un en un. Fixeu-vos en l'exemple següent:

```
<h1>Primer títol</h1>
<p>Primer paràgraf</p>
<h1>Segon títol</h1>
<p>Segon paràgraf</p>
<h1>Tercer títol</h1>
<p>Tercer paràgraf</p>

<script>
  var inicialitzacio = function() {
    var $paragrafs = $('p');
    $paragrafs.hide();
  }

  $(document).ready(inicialitzacio);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/amvxrY?editors=1010.

Com es pot apreciar, dins de la funció `inicialitzacio` s'invoca la funció `jQuery` passant com a paràmetre `p`; això indica que s'ha de fer una cerca de tots els elements de tipus `p`. Així doncs, el retorn de la funció és un nou objecte `jQuery` que conté tots els elements de tipus `p`.

El mètode `hide` dels objectes jQuery amaguen tots els objectes continguts, mentre que el mètode `show` els mostra.

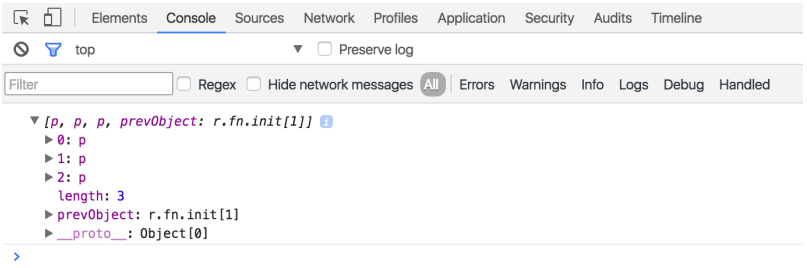
Seguidament, es crida al mètode `hide` d'aquest objecte per amagar tots els paràgrafs. Fixeu-vos que no cal iterar sobre els elements de la col·lecció, simplement cridant el mètode ja s'aplica el canvi a tots ells.

Si proveu de mostrar l'objecte jQuery per la consola, veureu que té algunes característiques similars als *arrays*, ja que es tracta d'una col·lecció d'elements. Afegiu el codi següent al codi anterior, dins de la funció `inicialitzacio`:

```
console.log($paragrafs);
```

Podeu veure aquest exemple en l'enllaç següent: codepen.io/ioc-daw-m06/pen/rrxvgj?editors=1010 i el resultat de la consola a la figura 2.2.

FIGURA 2.2



Objecte jQuery a la consola

Per visualitzar el resultat s'ha de fer servir la consola de les eines de desenvolupador del navegador, ja que la consola de CodePen no mostra correctament la informació.

Fixeu-vos que hi ha els tres elements seleccionats amb el seu índex corresponent: '0', '1', '2' i, seguidament, el valor de la propietat `length`. Si desplegueu qualsevol dels objectes niats, veureu tota l'estructura interna de cadascun d'aquests elements.

Tot i que aquestes propietats i mètodes es gestionen a través de la interfície proporcionada per la biblioteca, és interessant saber com es pot consultar aquesta informació directament al vostre navegador.

En alguns casos necessitareu accedir directament a algun dels elements seleccionats per l'objecte jQuery. Es poden obtenir tant tots els elements com un element en particular. En tots dos casos s'accedeix a través del mètode `get`:

- Si es passa un nombre com a argument, es retornarà l'element en aquesta posició.
- Si no es passa cap argument, es retornarà un *array* amb tots els elements.

Alternativament, es pot accedir directament a l'element com si es tractés d'un *array* (per exemple, `$paragrafs[1]`). D'altra banda, no hi ha cap garantia que l'accés directe a l'índex d'un objecte `jQuery` continuï funcionant en versions futures, per consegüent, **es desaconsella fer-lo servir**.

Vegeu a continuació un exemple que inclou els tres mètodes d'accés i els mostra per la consola:

```
<p>Primer paràgraf</p>
<p>Segon paràgraf</p>
<p>Tercer paràgraf</p>

<script>
  var inicialitzacio = function() {
    var $paragrafs = $('p');
    console.log("Element (primer paràgraf):", $paragrafs.get(0));
    console.log("Element (últim paràgraf):", $paragrafs[2]);
    console.log("Col·lecció d'elements (tots els paràgrafs):", $paragrafs.get()
        );
  }

  $(document).ready(inicialitzacio);
```

```
</script>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/gwAOVv?editors=1010.

Malauradament alguns dels objectes mostrats són massa complexos per a la consola de CodePen, així que haureu de consultar el resultat a la consola de les eines de desenvolupador del vostre navegador.

Com es pot comprovar, tant `$paragrafs.get(0)` com `$paragrafs[2]` retornen un element del DOM, mentre que `paragrafs.get()` retorna un *array* amb tots els elements. Cal destacar que el retorn del mètode `get` (o l'accés directe) són elements del DOM, no objectes `jQuery`.

Així doncs, no es poden invocar mètodes de jQuery sobre aquests elements. Per poder manipular-los individualment utilitzant la biblioteca s'hauran de convertir en objectes `jQuery`, passant-los com argument a la funció `jQuery`:

```
<style>
  .vermell {
    color: red;
  }
</style>

<p>Primer paràgraf</p>
<p>Segon paràgraf</p>
<p>Tercer paràgraf</p>

<script>
  var inicialitzacio = function() {
    var $paragrafs = $('p');
    var $paragraf2 = $($paragrafs.get(1));
    $paragraf2.addClass('vermell');
  }

  $(document).ready(inicialitzacio);
</script>
```

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

Fixeu-vos que primer s'han cercat tots els paràgrafs i, seguidament, s'ha tornat a cridar la funció `jQuery` passant com a argument el segon element dels seleccionats. El resultat és un nou objecte `jQuery` que conté només aquest element i, per tant, es pot manipular.

2.2 Selectors

Per generar un objecte `jQuery` que contingui una llista d'elements ja existents, s'ha d'invocar la funció `jQuery` passant com a argument un *selector*. Aquest selector pot ser **de diferents tipus**, no està limitat als selectors de CSS com en el cas dels mètodes que formen part de l'especificació del DOM.

Tot i que la funcionalitat és similar a la dels mètodes `querySelector` i `querySelectorAll` de les interfícies del DOM `Document` i `Elements`, en comptes d'un element o una col·lecció d'elements, aquest mètode sempre retorna un objecte `jQuery`. Aquest objecte conté tots els elements que coincideixen amb la selecció. Fins i tot en cas que no hi hagi cap element coincident, es retornarà un objecte però amb longitud 0 (propietat `length`).

Llista de selectors

Podeu trobar una llista completa dels selectors en l'enllaç següent: www.goo.gl/jn30Ta.

Els tipus de selectors disponibles es poden dividir en dos grans grups:

- **Selectors CSS**: aquests selectors es corresponen exactament amb els selectors CSS, inclosos els selectors afegits a CSS3.
- **Selectors propis de jQuery**: aquests són selectors específics de jQuery, que funcionen com dreceres per a seleccions molt comunes, com per exemple `:image` –per seleccionar totes les imatges– o `:input` –per seleccionar tots els elements d'entrada de dades (`input`, `textarea`, `select` i `button`).

Cal destacar que mentre que la funció `jQuery` fa la cerca a tot el document, hi ha un altre mètode que forma part de tots els objectes `jQuery` i permet cercar, només, entre els elements descendents: és el mètode `find`.

2.2.1 Selectors CSS

Podeu trobar més informació sobre els selectors CSS a l'apartat "Programació amb el DOM (Document Object Model)" d'aquesta unitat.

Aquest tipus de selectors són els més utilitzats perquè ofereixen les mateixes possibilitats que CSS i, per tant, la majoria de desenvolupadors web les coneixen.

Per demostrar el funcionament d'aquests selectors s'afegirà una classe CSS que pintarà el fons de l'element de color vermell. Vegeu-ne un primer exemple, en el qual se seleccionen tots els elements amb el *tag* p:

El mètode `addClass` de la biblioteca jQuery afegeix la classe passada com a argument als elements seleccionats.

```
1  <style>
2  .vermell {
3    background-color: red
4  }
5  </style>
6
7  <h1>Primer títol</h1>
8  <p>Primer paràgraf</p>
9  <h1>Segon títol</h1>
10 <p>Segon paràgraf</p>
11 <h1>Tercer títol</h1>
12 <p>Tercer paràgraf</p>
13
14 <script>
15   var inicialitzacio = function() {
16     var $paragrafs = $('p');
17     $paragrafs.addClass('vermell');
18   }
19
20   $(document).ready(inicialitzacio);
21 </script>
```

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

Com es pot apreciar, el fons de tots els paràgrafs es pinta de color vermell, perquè s'ha afegit la classe `vermell` a cadascun. A més a més, podeu fer servir les eines de desenvolupador per inspeccionar el codi HTML de la pàgina i comprovar que, efectivament, s'ha afegit la classe.

Per descomptat, la selecció no es limita als elements, és possible combinar els selectors per filtrar per classes o identificadors:

```
<style>
.vermell {
  background-color: red
}

.verd {
  background-color: green
}
</style>

<div>
  <h1>Primer títol</h1>
  <p>Primer paràgraf</p>
  <h1 class="central">Segon títol</h1>
  <p class="central">Segon paràgraf</p>
  <h1>Tercer títol</h1>
  <p id="tercer">Tercer paràgraf</p>
</div>
<p class="central">Aquest no canvia de color</p>

<script>
var inicialitzacio = function() {
  var $central = $('div .central');
  $central.addClass('vermell');

  var $tercer = $('#tercer');
  $tercer.addClass('verd');
}

$(document).ready(inicialitzacio);
</script>
```

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

Fixeu-vos que el selector `div .central` ha seleccionat tots els elements amb la classe `central` descendents d'un element de tipus `div` i, per consegüent, l'últim paràgraf no s'ha inclòs. D'altra banda, com era d'esperar, el fons de l'element amb identificador `tercer` s'ha pintat de color verd.

De la mateixa manera, es poden utilitzar els selectors d'atribut; això permet seleccionar elements segons quines siguin les característiques dels seus atributs:

- Si un atribut es troba present: `[atribut]`.
- Si l'atribut conté un valor concret: `[atribut=''valor'']`.
- Si el seu valor comença pel valor: `[atribut^=''valor'']`.
- Si el seu valor acaba pel valor: `[atribut$=''valor'']`.
- Si l'atribut conté el valor: `[atribut*=''valor'']`.

A continuació podeu comprovar el funcionament de dos d'aquests selectors. Primerament se cerquen tots els enllaços a la Wikipedia del document via el seu atribut `href`, i seguidament se cerquen només els de llengua anglesa (inclouen el subdomini `en`):

```
<style>
  .vermell {
    background-color: red
  }

  .resaltat {
    font-weight: bold;
  }
</style>

<div>
  <h1>Primer títol</h1>
  <p>Primer paràgraf <a href="https://ca.wikipedia.org/wiki/Aut%C3%B2mat_finit"
      >Enllaç a Automàt finit</a></p>
  <h1 class="central">Segon títol</h1>
  <p class="central">Segon paràgraf <a href="https://en.wikipedia.org/wiki/
      Finite-state_machine">Enllaç a Finite-state machines</a></p>
  <h1>Tercer títol</h1>
  <p id="tercer">Tercer paràgraf</p>
</div>
<p class="central">Aquest no canvia de color <a href="#">Un altre enllaç</a></p
    >

<script>
  var inicialitzacio = function() {
    var $wikipedia = $('[href*="wikipedia"]');
    $wikipedia.addClass('vermell');

    var $angles = $('[href^="https://en.wikipedia.org"]');
    $angles.addClass('resaltat');
  }

  $(document).ready(inicialitzacio);
</script>
```

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

Com es pot apreciar, el primer selector (`[href*=''wikipedia'']`), selecciona només els enllaços que contenen la paraula `wikipedia` als quals s'afegeix la classe `vermell`, mentre que el segon (`[href^='https://en.wikipedia.org']`) selecciona només els que pertanyen al subdomini lligat a la llengua anglesa de la Wikipedia.

D'altra banda, l'enllaç `Un altre enllaç` no forma part de cap dels dos objectes `jQuery`, ja que encara que conté l'atribut `href` el seu valor no compleix les condicions de cap dels dos selectors.

Cal destacar que l'element que enllaça amb la pàgina anglesa forma part de les col·leccions d'elements de tots dos objectes (referenciats per `$wikipedia` i `$angles`), de manera que qualsevol mètode cridat sobre un o altre l'afectarà.

Per exemple, si afegiu el codi següent al final de la funció d'inicialització, s'amagaran tots dos enllaços:

```
$wikipedia.hide();
```

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

En canvi, si es crida el mateix mètode sobre `$angles`, s'amagarà només l'enllaç a la pàgina anglesa:

```
$angles.hide();
```

Podeu veure aquest exemple en el següent enllaç: www.codepen.io/ioc-daw-m06/pen/VKZLqY.

Així doncs, es pot concloure que l'element que enllaça amb la pàgina anglesa forma part de la col·lecció d'elements seleccionats per tots dos objectes `jQuery`.

També es poden fer servir pseudoclasses com `first-child` o `last-child`, com es pot comprovar en l'exemple següent:

```
<style>
  .vermell {
    background-color: red;
  }

  .verd {
    background-color: green;
  }
</style>

<ul>
  <li>Primer element</li>
  <li>Segon element</li>
  <li>
    <ul>
      <li>Primer subelement</li>
      <li>Segon subelement</li>
      <li>Últim subelement</li>
    </ul>
    </li>
  <li>Últim element</li>
</ul>

<script>
  var inicialitzacio = function() {
    var $primers = $('li:first-child');
    $primers.addClass('vermell');

    var $ultims = $('li:last-child');
    $ultims.addClass('verd');
  }

  $(document).ready(inicialitzacio);
</script>
```

Podeu veure aquest exemple en l'ellaç següent: www.codepen.io/ioc-daw-m06/pen/GjoJZj.

Cal destacar que, tot i que no s'han mostrat exemples complexos, jQuery admet tot tipus de combinacions, incloent-hi la combinació amb els seus propis selectors.

2.2.2 Selectors propis de jQuery: seleccions especials

Llista de selectors de jQuery

Podeu trobar una llista completa dels selectors propis de jQuery en l'enllaç següent: www.goo.gl/zzvN1g.

L'ús de selectors a jQuery no està limitat als selectors de CSS, ja que disposa d'un joc propi de selectors que simplifiquen encara més la selecció dels elements.

Entre les dreceres genèriques que s'apliquen a tots els elements, la biblioteca disposa de les següents:

- **:first** i **:last**: selecciona només el primer element o només l'últim, respectivament, de la selecció. Cal distingir entre la funcionalitat d'aquestes dreceres i la dels selectors de CSS `first-child` i `last-child`, ja que en aquest cas es tracta del primer o últim element que formaria part de la selecció, i no pas dels descendents concrets d'algun altre.
- **:header**: selecciona totes les capçaleres, per exemple `h1`, `h2`, etc.
- **:even** i **:odd**: selecciona els elements parells i els senars, respectivament.

Vegeu a continuació un exemple amb els selectors `first` i `last` que us ajudarà a entendre la diferència amb els selectors de CSS `first-child` i `last-child`:

```
<style>
  .vermell {
    color: red;
  }

  .verd {
    color: green;
  }

  .negreta {
    font-weight: bold;
  }

  .cursiva {
    font-style: italic;
  }
</style>

<ul>
  <li>Primer element</li>
  <li>Segon element</li>
  <li>
    <ul>
      <li>Primer sub-element</li>
      <li>Segon sub-element</li>
      <li>Últim sub-element</li>
    </ul>
  </li>
  <li>Últim element</li>
</ul>

<script>
  var inicialitzacio = function() {
    var $primers = $('li:first-child');
    $primers.addClass('vermell');

    var $ultims = $('li:last-child');
    $ultims.addClass('verd');

```

```
    var $primer = $('li:first');
    $primer.addClass('negreta');

    var $ultim = $('li:last');
    $ultim.addClass('cursiva');
  }

  $(document).ready(inicialitzacio);
</script>
```

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

Com es pot apreciar, mentre que els selectors CSS han seleccionat els elements que són primer i últim element respecte al seu element pare, els selectors propis només han seleccionat el primer i l'últim element de la selecció, que en aquest cas correspon a tots els elements amb l'etiqueta `li`.

A continuació podeu veure un exemple d'utilització del selector `header`. Fixeu-vos que és possible tant seleccionar totes les capçaleres com combinar-lo amb un altre selector CSS per obtenir una selecció més concreta:

```
<style>
  .vermell {
    color: red;
  }

  .cursiva {
    font-style: italic;
  }
</style>

<h1>Títol 1</h1>
<p>Paràgraf 1</p>
<div>
  <h2>Títol 2</h1>
  <p>Paràgraf 2</p>
  <p>Paràgraf 3</p>
  <h3>Títol 3</h3>
  <p>Paràgraf 4</p>
</div>

<script>
  var inicialitzacio = function() {
    var $titols = $(':header');
    $titols.addClass('vermell');

    var $titolsEnContenidor = $('div :header');
    $titolsEnContenidor.addClass('cursiva')
  }

  $(document).ready(inicialitzacio);
</script>
```

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

Els selectors `odd` i `even` serveixen per afegir efectes o comportaments diferents d'elements alterns, per exemple les files o columnes d'una taula. En aquest exemple s'han fet servir diferents colors per mostrar-ne el comportament, però cal tenir en compte que si només necessiteu canviar el format, el més recomanable

és fer-ho directament a través de fulls d'estil i no pas modificant el document dinàmicament com s'ha fet aquí:

```
<style>
  table {
    border-collapse: collapse;
  }

  td {
    text-align: center;
    padding: 2px;
  }

  th {
    padding: 5px;
  }
  .blau {
    background-color: #1b95e0;
  }

  .gris {
    background-color: gray;
  }

  .negreta {
    font-weight: bold;
  }
</style>

<table>
  <tr>
    <th>Columna 1</th>
    <th>Columna 2</th>
        <th>Columna 3</th>
        <th>Columna 4</th>
  </tr>
  <tr>
    <td>Fila 1</td>
    <td>Fila 1</td>
    <td>Fila 1</td>
    <td>Fila 1</td>
  </tr>
  <tr>
    <td>Fila 2</td>
    <td>Fila 2</td>
    <td>Fila 2</td>
    <td>Fila 2</td>
  </tr>
  <tr>
    <td>Fila 3</td>
    <td>Fila 3</td>
    <td>Fila 3</td>
    <td>Fila 3</td>
  </tr>
  <tr>
    <td>Fila 4</td>
    <td>Fila 4</td>
    <td>Fila 4</td>
    <td>Fila 4</td>
  </tr>
</table>
</script>
```

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

Fixeu-vos que tot i que `odd` correspon als elements senars, el primer element marcat com a senar no és la primera fila ni la primera columna. Això es deu

al fet que els elements comencen a comptar-se a partir de 0 i, per tant, la primera fila i columna corresponen a `even` (parell).

2.2.3 Selectors propis de jQuery: formularis

Un altre tipus de selectors que també ofereix la biblioteca faciliten treballar amb formularis:

- **:input**: selecciona tots els elements de tipus `input`, `textarea`, `select` i `button`.
- **:text**: selecciona tots els elements `input` de tipus `text`.
- **:checkbox**: selecciona tots els elements amb tipus `checkbox`.
- **:radio**: selecciona tots els elements de tipus `radio`.
- **:button**: selecciona tots els botons i elements amb tipus `button`.

El mètode `val` dels objectes `jQuery` permet consultar o establir la propietat `value` d'un element.

```
<style>
  .tabular {
    margin-left: 25px;
  }

  div {
    margin: 0 auto;
    width: 300px;
    border: 1px solid black;
  }

  .caselles {
    float: right;
  }
</style>

<div>
  <h1>Formulari</h1>
  <input type="text" name="nom" />Nom<br> <input type="text" name="cognom" />
      Cognom<br>
  <h2>Descripció</h2>
  <textarea name="descripcio"></textarea><br> <input type="checkbox" name="
      acceptar" />Acceptar condicions<br>
  <input type="radio" name="color" value="blau" checked>blau<br>
  <input type="radio" name="color" value="vermell">vermell
</div>

<script>
  var inicialitzacio = function() {
    var $texts = $(':text');
    $texts.val(12345678);

    var $input = $(':input');
    $input.addClass('tabular');

    var $radio = $(':radio');
    $radio.addClass('caselles');

   var $checkbox = $(':checkbox');
    $checkbox.addClass('caselles');
  }
```

```
40
41    $(document).ready(inicialitzacio);
42  </script>
```

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

En aquest exemple es modifica –via codi– el valor dels camps de text (invocant el mètode `val`), s'afegeix un marge a l'esquerra de tots els elements englobats pel selector `:input` (`input` i `textarea`) i, finalment, es converteixen les caselles i els botons de selecció en flotants a la dreta afegint la classe `caselles`.

Fixeu-vos que per afegir la classe `caselles` tant a les caselles com als botons de ràdio s'ha hagut de fer individualment. Afortunadament jQuery ofereix el mètode `add`, que permet concatenar seleccions. Canvieu el codi JavaScript de l'exemple anterior pel següent per comprovar-ho:

```
1  var inicialitzacio = function() {
2    var $texts = $(':text');
3    $texts.val(12345678);
4
5    var $input = $(':input');
6    $input.addClass('tabular');
7
8    var $caselles = $(':radio').add(':checkbox');
9    $caselles.addClass('caselles');
10 }
11
12 $(document).ready(inicialitzacio);
```

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

Com es pot apreciar, l'objecte `jQuery` referenciat per `$caselles` ara conté tant la selecció de `:radio` com la selecció de `:checkbox` i, conseqüentment, quan s'invoca el mètode `addClass`, aquest afecta tots dos tipus d'elements.

Però es pot simplificar encara més. Els objectes jQuery tenen una interfície fluida, és a dir, molts dels seus mètodes retornen sempre el mateix objecte `jQuery`, de manera que es poden invocar tots els mètodes un darrere de l'altre. A més a més, no cal fer cap operació addicional amb l'objecte, ni tan sols cal emmagatzemar la referència. Vegeu una implementació més concisa del mateix exemple; substituïu el codi JavaScript pel següent:

```
1  var inicialitzacio = function() {
2    $(':text').val(12345678);
3    $(':input').addClass('tabular');
4
5    $(':radio')
6      .add(':checkbox')
7      .addClass('caselles');
8  }
9
10 $(document).ready(inicialitzacio);
```

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

Fixeu-vos que no s'ha fet servir cap variable, s'han invocat els mètodes necessaris directament. En el cas del selector per a les caselles, s'ha posat la invocació a cada mètode en una línia diferent, en aquest cas només hi ha dues invocacions i es podria haver escrit tot en la mateixa línia, però en casos més complexos la visibilitat millora molt si es fa d'aquesta manera.

Cal destacar els dos elements següents –especialment interessants–, ja que cap dels dos es pot substituir per selectors CSS, ni requereixen que sigui descendent directe:

- **:has(element)**: selecciona els elements que continguin com a descendent `element`.
- **:contains(text)**: selecciona els elements que continguin el `text`.

Vegeu com funciona en l'exemple següent:

```
<style>
  .vermell {
    color: red;
  }

  .negreta {
    font-weight: bold;
  }

  span {
    font-style: italic;
  }
</style>

<div>Que deu ésser feta al cavaller. Volgué anar havent dolor e contricció de. La Comtessa e als servidors la. Ésser atesa sens mitjà de virtuts Los cavallers. Actes frescs de nostres dies.</div>
<p> D'or ab les armes sues e de la Comtessa. <a href="#">Lo virtuós Comte</a> en edat avançada de cinquanta-cinc. Als servidors la sua partida En la fè rtil. Ho pres ab molta impaciència.</p>
<p> Romans: d'Escipió d'Anibal de Pompeu d'Octovià. Longitud de molts dies E com entre los altres insignes.</p>
<p> Comte en edat avançada de cinquanta-cinc anys. Fama d'aquell no deu preterir per longitud de. Havia rei o fill de rei.</p>
<p> Experiència mostra la debilitat de la nostra memòria sotsmetent fàcilment. Cavaller pare de cavalleria lo comte Guillem. De <span><a href="#">Marc Antoni</a></span> e de molts altres.</p>

<script>
  var inicialitzacio = function() {
    $('p:has(a)').addClass('vermell');
    $('p:contains(Comte)').addClass('negreta');
  }

  $(document).ready(inicialitzacio);
</script>
```

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

Primerament, se seleccionen només els elements de tipus paràgraf (`p`) que continguin un element de tipus `a` i, seguidament, s'afegeix la classe `vermell`, de manera que el text d'aquests elements passa a ser vermell. Fixeu-vos que en l'últim

paràgraf l'enllaç no és un descendent directe però, tal com s'espera, se selecciona correctament.

A continuació, se cerquen tots els paràgrafs que continguin la paraula `Comte`. Com que s'ha indicat expressament que només s'ha de cercar als paràgrafs, la primera línia no se'n veu afectada, ja que es tracta d'un element de tipus `div`. Com en el cas anterior, no és necessari que sigui descendent directe, i per això s'aplica correctament a la quarta línia.

En tots dos casos es pot passar el paràmetre amb cometes o sense; així doncs, es podrien substituir pel codi següent:

```
1 $('p:has("a")').addClass('vermell');
2   $('p:contains("Comte")').addClass('negreta');
```

S'ha de tenir en compte que aquests selectors no són tan eficients com els selectors CSS. Tot i així, en la majoria dels casos, aquesta pèrdua d'eficiència és inapreciable.

El mètode 'filter'

Podeu trobar més informació sobre el mètode `filter` en l'enllaç següent: www.api.jquery.com/filter/.

En els casos en què l'eficiència sigui crítica es recomana descompondre la selecció en dues parts. Primerament, s'invoca la funció `jQuery` amb el selector CSS (que està optimitzat) i sobre l'objecte retornat s'invoca el mètode `filter`, passant com a argument els selectors propis, que actuaran com a filtre. Podeu comprovar-ho canviant el codi JavaScript de l'exemple anterior pel següent:

```
1 var inicialitzacio = function() {
2   $('p').filter(':has("a")').addClass('vermell');
3   $('p').filter(':contains("Comte")').addClass('negreta');
4 }
5
6 $(document).ready(inicialitzacio);
```

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

Com es pot apreciar, el comportament és el mateix però internament el càlcul és més eficient perquè en lloc de fer la selecció a partir de tots els elements de la pàgina, es realitza sobre un petit subconjunt, el dels paràgrafs.

2.2.4 Cercar entre els elements seleccionats: 'filter' i 'find'

Quan s'utilitza la funció `jQuery` la selecció es fa a partir de l'arrel del document; en moltes situacions el que interessa és obtenir una selecció a partir d'un objecte `jQuery` ja existent. Per portar a terme aquesta acció els objectes `jQuery` disposen de dos mètodes: `filter` i `find`.

La **diferència entre els mètodes** `filter` i `find` és que el primer aplica el selector només entre els elements que formen part de la selecció actual; mentre que el segon aplica el selector als elements descendents.

Tots dos mètodes retornen un nou objecte `jQuery` amb el resultat de la selecció. Vegeu una demostració d'aquesta diferència en l'exemple següent:

```
<style>
  .filtrat {
    color: red;
  }

  .cercat {
    font-weight: bold;
  }
</style>

<ul class="principal">
  <li>Element 1</li>
  <ul>Subllista
    <li>Element 1.1</li>
    <li>Element 1.2</li>
  </ul>

  <li>Element 2</li>
</ul>
<ul class="principal">
  <li>Element 1</li>
  <ul>Subllista
    <li>Element 1.1</li>
    <li>Element 1.2</li>
  </ul>

  <li>Element 3</li>
</ul>

<script>
  var inicialitzacio = function() {
    var $llista = $('ul');

    $filtrat = $llista.filter('li');
    $filtrat.addClass('filtrat');

    $cercat = $llista.find('li');
    $cercat.addClass('cercat');

    console.log('Elements a la llista:', $llista.length);
    console.log('Elements \'li\' filtrats:', $filtrat.length);
    console.log('Elements \'li\' cercats:', $cercat.length);
  }

  $(document).ready(inicialitzacio);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/KgVRBd?editors=1111.

Fixeu-vos en els elements seleccionats per a cadascun dels objectes `jQuery`:

- `$llista`: 4 elements seleccionats, corresponents als 4 elements `ul` del document.
- `$filtrat`: 0 elements seleccionats. Encara que pot sorprendre, cal tenir en compte que a `$llista` no es troba cap element de tipus `li`; així doncs, el resultat és correcte.
- `$cercat`: 8 elements seleccionats. En aquest cas, se cerca entre cadascun dels elements descendents d'`ul` i selecciona els elements de tipus `li` que contenen.

Cal destacar que aquests mètodes són disponibles a tots els objectes `jQuery` i, per consegüent, no és necessari que els elements es trobin afegits al DOM. És a dir, es poden crear nous elements lligats a objectes `jQuery` i fer operacions de cerca sobre ells:

```
1  var inicialitzacio = function() {
2    $nousElements = $('<h1>Títol 1</h1><p>Paràgraf 1</p><p>Paràgraf 2</p><h2>Tí
         tol 2</h2><p>Paràgraf 3</p>');
3
4    $titols = $nousElements.filter(':header');
5    $paragrafs = $nousElements.filter('p');
6
7    console.log('Nombre de títols:', $titols.length);
8    console.log('Nombre de paràgrafs:', $paragrafs.length);
9  }
10
11 $(document).ready(inicialitzacio);
```

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

Quan a la funció `jQuery` es passa una cadena de codi HTML com a argument, es crea un nou objecte que conté com a selecció els elements corresponents a aquest codi.

Com podeu veure en aquest exemple, tot i que els nodes generats per la funció `jQuery` només es troben a la memòria, és possible invocar el mètode `filter` per cercar les capçaleres i els paràgrafs.

2.3 Crear i manipular elements

Un dels punts forts de jQuery és la facilitat amb la qual es poden crear nous elements i modificar-los. De la mateixa manera que les interfícies del DOM permeten crear branques i manipular-les abans d'afegir-les al document, jQuery permet treballar amb aquests objectes mentre es troben a la memòria.

D'aquesta manera, és possible generar un objecte `jQuery` a partir d'un fragment de codi HTML (per exemple, enviat des d'un servidor), fer modificacions als seus atributs o classes, clonar-lo, afegir-lo a un altre objecte jQuery i, finalment, afegir aquesta nova branca a l'arbre que forma el document.

Cal destacar que, al contrari de quan es treballa directament amb les interfícies del DOM, jQuery permet afegir nous estils i atributs d'una forma molt intuïtiva a través dels seus propis mètodes.

2.3.1 Crear nous elements

La creació de nous elements amb jQuery és molt simple, només cal passar com a paràmetre de la funció la cadena de codi HTML i la biblioteca retornarà un nou objecte `jQuery` que contindrà com a selecció aquests elements generats correctament.

El mètode `append` permet afegir els elements continguts en un objecte `jQuery` a un altre objecte `jQuery`.

Per exemple, per crear un nou paràgraf només cal passar el seu codi corresponent:

```
var inicialitzacio = function() {
  var $paragraf = $('<p>Paràgraf generat dinàmicament</p>');
  $(document.body).append($paragraf);
}

$(document).ready(inicialitzacio);
```

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

Com es pot apreciar, primer s'ha creat un nou objecte jQuery referenciat per `$paragraf`, i seguidament s'ha seleccionat l'element `body` (obtingut a través de la propietat del document) i s'hi ha afegit aquest nou element.

Vegeu-ne a continuació un exemple una mica més complex, però que està fet aplicant la mateixa mecànica:

```
var inicialitzacio = function() {
  var $taula = $('<table><tr><th>nom</th><th>cognom</th></tr><tr><td>Josep</td
      ><td>Campmany</td></tr><tr><td>Maria</td><td>Torres</td></tr></table>');
  $(document.body).append($taula);
}

$(document).ready(inicialitzacio);
```

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

Tot i que s'ha fet servir el mateix sistema, aquest cop la codificació és molt menys entenedora. En cas d'haver d'afegir-hi més files, el codi s'aniria fent cada vegada més complicat i més difícil de depurar. Atès que aquest paràmetre no és més que una cadena de text, podem confeccionar-la pas a pas.

Una possible solució seria aplicar el disseny descendent, creant tot un seguit de funcions que permetin confeccionar el codi per generar la taula:

```
var generarTaula = function(alumnes) {
  var html = '<table>' + generarCapcalera();

  for (var i = 0; i < alumnes.length; i++) {
    html += generarFila(alumnes[i]);
  }

  html += '</table>';
  return html;
}

var generarCapcalera = function() {
  var capcalera = '<tr>';
  capcalera += '<th>nom</th>';
  capcalera += '<th>cognom</th>';
  capcalera += '</tr>';
  return capcalera;
}

var generarFila = function(alumne) {
  var fila = '<tr>';
  fila += '<td>' + alumne.nom + '</td>';
  fila += '<td>' + alumne.cognom + '</td>';
  fila += '</tr>';
```

```
  return fila;
}

var inicialitzacio = function() {
  var alumnes = [
    {nom: 'Josep', cognom: 'Campmany'},
    {nom: 'Maria', cognom: 'Torres'},
    {nom: 'Alex', cognom: 'Puig'},
    {nom: 'Ana', cognom: 'Perez'}
  ];

  var html = generarTaula(alumnes);
  var $taula = $(html);
  $(document.body).append($taula);
}

$(document).ready(inicialitzacio);
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ALEdkJ?editors=0010.

Com podeu veure, la llargària del codi ha augmentat considerablement, però la complexitat s'ha reduït i s'ha millorat l'escalabilitat. Per exemple, si en lloc de quatre alumnes en fossin quaranta, el codi seria el mateix modificant només el diccionari de dades. A més a més, el codi de cada funció és molt clar, de manera que és molt fàcil modificar-lo.

En primer lloc s'ha afegit un diccionari de dades que conté les dades dels alumnes, d'aquesta manera es podria canviar l'origen d'aquestes dades, per exemple, carregant-les d'un fitxer extern sense haver de modificar la resta del programa.

Dintre de la funció d'inicialització es genera el codi HTML cridant la funció `generarTaula` i passant-hi com a argument el diccionari de dades. Dintre d'aquesta funció es genera una cadena de text amb el codi per generar una taula, al qual s'afegeix el codi de la capçalera cridant la funció `generarCapcalera`.

Seguidament es recorren tots els elements del diccionari de dades afegint el codi per cada fila cridant la funció `generarFila` amb les dades de cada alumne. Finalment es tanca l'etiqueta de la taula i es retorna el codi complet, que és convertit en un objecte `jQuery` i afegit al cos del document.

Aquesta és només una de les tècniques que ofereix jQuery per compondre una branca d'elements; en cada cas s'ha de valorar quin és el sistema que millor s'adapta a la vostra aplicació.

Un altre sistema per generar nous objectes `jQuery` és passar com a argument elements del DOM:

```
<style>
  .vermell {
    color: red;
  }

  .negreta {
    font-weight: bold;
  }
</style>

<p>Primer paràgraf</p>
```

```
<p id="segon">Segon paràgraf</p>
<p>Tercer paràgraf</p>

<script>
  var inicialitzacio = function() {
    var paragrafs = document.getElementsByTagName('p');
    var paragraf2 = document.getElementById('segon');

    var $paragrafs = $(paragrafs);
    var $paragraf2 = $(paragraf2);
    $paragrafs.addClass('vermell');
    $paragraf2.addClass('negreta');
  }

  $(document).ready(inicialitzacio);
</script>
```

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

Com es pot apreciar, primer de tot s'han obtingut els nodes fent servir els mètodes de les interfícies del DOM (una col·lecció de nodes en el primer cas, i un únic node en el segon). A continuació, s'han generat els dos objectes `jQuery` passant els elements com a argument, i finalment, s'ha afegit la classe corresponent a cadascun: `vermell` a tots els paràgrafs i `negreta` al segon.

2.3.2 Manipulació de classes: 'addClass', 'removeClass' i 'toggleClass'

A banda de l'addició de classes als elements, jQuery ofereix mètodes per eliminar-les i per activar-les/desactivar-les. Tots tres casos són fàcils de fer servir: només cal invocar el mètode sobre l'objecte `jQuery` passant com a argument el nom de la classe:

```
<style>
  .vermell {
    color: red;
  }

  .verd {
    color: green;
  }

  .negreta {
    font-weight: bold;
  }
</style>

<p class="negreta">Paràgraf 1: originalment en negreta</p>
<p class="vermell">Paràgraf 2: originalment vermell</p>

<script>
  var inicialitzacio = function() {
    var $paragrafs = $('p');

    $paragrafs.removeClass('vermell');
    $paragrafs.addClass('verd');
    $paragrafs.toggleClass('negreta');
  }
```

```
26
27    $(document).ready(inicialitzacio);
28  </script>
```

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

Com es pot apreciar no es produeix cap error en eliminar la classe `vermell` de tots els paràgrafs, tot i que el primer paràgraf no conté aquesta classe. Com és d'esperar, quan s'afegeix la classe `verd`, s'afegeix correctament a tots dos paràgrafs.

Cal parar especial atenció al comportament d'invocar `toggleClass`: es determina si s'ha d'activar o desactivar segons l'estat concret de cada element, és a dir, en el primer cas estava activat i s'elimina, mentre que en el segon cas s'afegeix.

2.3.3 Modificar continguts: 'val', 'html' i 'text'

A l'hora de modificar el contingut d'un element s'ha de distingir entre modificar el seu valor (aplicable a elements de formularis), el contingut textual o el codi HTML (quan conté altres elements).

Per consultar i modificar aquests continguts, jQuery ofereix els mètodes `val` per als elements de formularis, `html` com a equivalent a la propietat `innerHTML` i `text` per manipular els continguts textuals.

El funcionament és molt simple: aquests mètodes s'invoquen a partir de l'objecte `jQuery` que contingui els elements que es volen modificar. Si es vol consultar el valor, s'han d'invocar sense passar cap argument; en canvi, si es volen modificar, s'ha de passar com a argument el valor, codi o text per substituir.

```
1  <label>Nom: </label>
2  <input type="text" value="Pere" />
3  <p>Paràgraf 1</p>
4  <p>Paràgraf 2</p>
5
6  <script>
7    var inicialitzacio = function() {
8      var $text = $('input');
9      var $paragraf1 = $('p:first');
10     var $paragraf2 = $('p:last');
11
12     console.log($text.val());
13     $text.val('Maria');
14
15     console.log($paragraf1.html());
16     $paragraf1.html("<ul><li>Element 1</li><li>Element 2</li></ul>");
17
18     console.log($paragraf2.text());
19     $paragraf2.text("El contingut textual d'aquest paràgraf ha estat modificat"
           );
20   }
21
22   $(document).ready(inicialitzacio);
23 </script>
```

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

Fixeu-vos que el resultat mostrat per la consola es correspon amb el valor original, mentre que al document es mostren els continguts actualitzats: `Maria` com a valor del quadre de text, una llista de dos elements com a contingut del primer paràgraf, i una frase diferent com a contingut del segon paràgraf.

2.3.4 Modificar estils: 'css'

A banda de treballar amb classes, la biblioteca jQuery també permet manipular directament els estils CSS d'un element de forma molt més simple que si ho heu de fer directament amb les interfícies del DOM.

Per afegir, modificar o eliminar un estil només cal invocar el mètode `css` de l'objecte `jQuery` que contingui els elements que cal modificar:

- **Afegir estils**: es passa com a argument el nom de la propietat CSS i el valor que s'ha d'assignar.
- **Eliminar estils**: es passa com a argument el nom de la propietat CSS i una cadena buida com a valor.
- **Consultar el valor d'una propietat CSS**: es passa com a argument el nom de la propietat CSS.

A continuació podeu veure un exemple en què es consulten les propietats CSS d'un element, se n'afegeixen de noves i se n'eliminen:

```
<p id="paragraf" style="color: red; font-weight: bold">Paràgraf</p>

<script>
  var mostrarEstils = function (element) {
    var estil = element.style;
    var $element = $(element)
    for (var i=0; i<estil.length; i++) {
      console.log (estil[i] + ':' + $element.css(estil[i]) + ";");
    }
  }

  var inicialitzacio = function() {
    var paragraf = document.getElementById('paragraf');
    var $paragraf = $(paragraf);

    console.log('-- Estil original --');
    mostrarEstils(paragraf);

    $paragraf.css('font-size', '30px');
    $paragraf.css('color', '');
    $paragraf.css('font-weight', 'lighter');

    console.log('-- Estil modificat --');
    mostrarEstils(paragraf);
  }

  $(document).ready(inicialitzacio);
```

```
</script>
```

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

Com es pot apreciar, jQuery no ofereix una solució simple per mostrar els estils. En lloc d'utilitzar el mètode `getComputedStyle` s'ha cridat el mètode `css` per a cadascuna de les propietats, que retorna el mateix resultat.

Fixeu-vos que tant per afegir mètodes com per actualitzar només cal passar el nom i el valor de la propietat, mentre que per eliminar-la s'ha de passar una cadena buida.

2.3.5 Manipular atributs: 'attr' i 'prop'

La biblioteca jQuery ofereix dos mètodes diferents per tractar amb atributs:

- El mètode `attr`, que serveix per consultar i manipular els atributs en general.
- El mètode `prop`, que serveix per consultar i modificar propietats del DOM com `checked`, `selected` o `disabled`.

Una diferència important entre `attr` i `prop` és que si es fa servir el primer per obtenir el valor d'una propietat com `checked`, el valor retornat és una cadena de text; en canvi, amb el segon mètode el valor de retorn és un booleà, que correspon a l'estat de l'element (marcat o no marcat).

S'ha de tenir en compte que en cas que l'objecte `jQuery` tingui seleccionats múltiples elements, aquests mètodes retornen el valor del primer element.

```
<label id="1"><input type="checkbox" checked="checked"/>Casella A</label>
<label id="2"><input type="checkbox" />Casella B</label>

<script>
  var $etiquetes = $('label');
  var $etiqueta1 = $('#1');
  var $etiqueta2 = $('#2');
  var $caselles = $(':checkbox');

  console.log('Atribut id de les etiquetes:', $etiquetes.attr('id'));

  console.log('Canviant id per "A"...');
  $etiquetes.attr('id', 'A');

  console.log('Atribut id de la etiqueta1:', $etiqueta1.attr('id'));
  console.log('Atribut id de la etiqueta2:', $etiqueta2.attr('id'));

  console.log('Es troben les caselles marcades?', $caselles.prop('checked'));

  console.log('Es troben les caselles marcades?', $caselles.attr('checked'));

  console.log('Desmarquem totes les caselles');
  $caselles.prop('checked', '');
</script>
```

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

Com es pot apreciar, tant el mètode `attr` com el mètode `prop` retornen només un valor, els corresponents al primer element seleccionat. D'altra banda, en canviar l'`id` per `A`, aquest s'aplica a totes les etiquetes.

Fixeu-vos com el valor retornat per `prop` és diferent d'`attr` quan es passa `checked` com a argument. En el primer cas retorna un booleà, mentre que en el segon retorna el valor. Com en el cas de les etiquetes, el valor retornat en tots dos casos és el corresponent al primer element de la selecció, i en canviar el valor de la propietat, aquest canvi s'aplica a totes les caselles.

Així doncs, si es vol obtenir el valor de cadascun o es volen manipular els elements individualment s'ha d'iterar sobre ells, per exemple fent servir el mètode `each` dels objectes `jQuery`:

```
<label id="1"><input type="checkbox" checked="checked"/>Casella A</label>
<label id="2"><input type="checkbox" />Casella B</label>

<script>
  var $etiquetes = $('label');
  var $caselles = $(':checkbox');

  var mostrarId = function ($items) {
    $items.each(function(index) {
      console.log('Atribut id de l\'etiqueta ' + index + ':', this.id);
    });
  }

  var afegirTextAId = function ($items, text) {
      $items.each(function() {
        this.id += text;
    });
  }

  var mostrarMarcada = function ($items) {
      $items.each(function(index) {
      console.log('Casella ' + index + ' marcada?', this.checked);
    });
  }

  mostrarId($etiquetes);
  console.log('Afegint text als Ids...');
  afegirTextAId($etiquetes, 'casella');
  mostrarId($etiquetes);

  console.log('Estat de les caselles');
  mostrarMarcada($caselles);
</script>
```

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

La biblioteca ofereix **dos mètodes** `each` amb funcions diferents. Si es crida a partir de la funció jQuery (`jQuery.each()`), aquest itera sobre l'*array* o la col·lecció passada com a argument; en canvi, si es crida aquest mètode a un objecte `jQuery`, el mètode `each` itera sobre cadascun dels elements seleccionats.

Gràcies al mètode `each` dels objectes `jQuery`, és molt fàcil iterar sobre tots els elements. Es passa al mètode `each` una funció com argument i aquesta serà invocada una vegada per cada element, fent servir com a context de la funció l'element corresponent. Opcionalment es pot afegir un paràmetre a la funció, que rebrà l'índex de l'element (`0` pel primer, `1` pel segon, `2` pel tercer...).

Recordeu que cada element és el context en el que es realitza la acció; ni executa la acció (això ho fa la funció) ni és el destinatari (encara que es pot modificar l'element).

Cal recalcar que el context d'execució de la funció és l'element seleccionat, per consegüent, `this` fa referència a aquest element, que és un node del DOM. És a dir, **no és un objecte jQuery**.

A continuació podeu trobar un exemple en què es modifica l'identificador dels elements de tipus `label` i es mostra per la consola del navegador si les caselles es troben o no marcades. En aquest exemple s'ha accedit directament a les seves propietats per simplificar, però és possible convertir aquests nodes en objectes `jQuery`:

```
<label id="1"><input type="checkbox" checked="checked"/>Casella A</label>
<label id="2"><input type="checkbox" />Casella B</label>

<script>
  var $etiquetes = $('label');
  var $caselles = $(':checkbox');

  var mostrarId = function($items) {
    $items.each(function(index) {
      var $element = $(this);

      console.log('Atribut id de l\'etiqueta ' + index + ':', $element.attr('id
          '));
    });
  }

  var afegirTextAId = function($items, text) {
    $items.each(function() {
      var $element = $(this);
      $element.attr('id', $element.attr('id') + text);
    });
  }

  var mostrarMarcada = function($items) {
    $items.each(function(index) {
      var $element = $(this);
      console.log('Casella ' + index + ' marcada?', $element.prop('checked'));
    });
  }

  mostrarId($etiquetes);
  console.log('Afegint text als Ids...');
  afegirTextAId($etiquetes, 'casella');
  mostrarId($etiquetes);

  console.log('Estat de les caselles');
  mostrarMarcada($caselles);
</script>
```

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

Recordeu que una de les maneres de crear un objecte `jQuery` és passar un element directament a la funció; així doncs, es passa l'element que serveix de context a la funció (`this`) i es genera un nou objecte `jQuery` a partir del qual es poden cridar els mètodes pertinents.

Fixeu-vos que en aquest cas concret la implementació s'ha complicat més i l'eficiència és menor que a l'exemple anterior, però pot haver-hi situacions en les quals sigui més pràctic treballar dintre de la funció amb objectes `jQuery`. S'ha de valorar cada cas i fer servir una implementació o altra segons les vostres necessitats.

2.4 Manipular el DOM

La biblioteca jQuery destaca tant per la facilitat amb la qual es poden seleccionar elements com per la facilitat per crear-ne nous. Concretament, per generar nous elements, ofereix les següents opcions, segons el tipus de paràmetre passat:

- **cadena de text que *sembli* HTML**: `$('<p>')` o `$('<p id="paragraf2"><em>Paràgraf amb èmfasi</em></p>')`.
- **Node HTML existent** (clona un element ja existent node): `$(element)`.

```
<div id="contenidor"></div>

<script>
  var $contenidor = $('#contenidor');

  var $nouParagraf = $('<p>');
  $nouParagraf.html('Paràgraf normal');

  var $nouParagrafEmfasi = $('<p id="paragraf2"><em>Paràgraf amb èmfasi</em></p>');

  $contenidor.append($nouParagraf);
  $contenidor.append($nouParagrafEmfasi);

  var paragraf3= document.createElement('p');
  paragraf3.innerHTML = 'Paràgraf per clonar';

  var $paragrafClonat = $(paragraf3);
  $contenidor.append($paragrafClonat);
</script>
```

Podeu veure aquest exemple l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ALLaAm?editors=1010.

En tots dos casos el retorn de la funció és un objecte `jQuery` que embolcalla el nou element (o elements). S'ha de tenir en compte que aquests nous elements encara no formen part del document, sinó que només es troben a la memòria.

Per afegir-los al DOM, la biblioteca ofereix diversos mètodes. Aquests mètodes s'han de cridar a partir d'un objecte que serà el contenidor dels elements, passant com a paràmetre el nou element que s'ha d'afegir:

- **prepend**: afegeix l'element al principi de la llista de descendents directes del contenidor.

- **append**: afegeix l'element al final de la llista descendents directes del contenidor.
- **before**: afegeix l'element al mateix nivell que el contenidor, però davant seu.
- **after**: afegeix l'element al mateix nivell que el contenidor, però darrere seu.

És a dir, els mètodes `prepend` i `append` col·loquen el nou element **dins** del contenidor, mentre que els mètodes `after` i `before` el col·loquen **fora** d'aquest, davant i darrere respectivament.

```
<ul>
  <li>Element 1</li>
  <li>Element 2</li>
  <li>Element 3</li>
</ul>

<script>
  var $llista = $('ul');

  var $nouItem1 = $('<li>Element append</li>');
  $llista.append($nouItem1);

  var $nouItem2 = $('<li>Element prepend</li>');
  $llista.prepend($nouItem2);

  var $nouItem3 = $('<ul><li>Element before</li></ul>');
  $llista.before($nouItem3);

  var $nouItem4 = $('<ul><li>Element after</li></ul>');
  $llista.after($nouItem4);
</script>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/vXXjpx?editors=1010.

Fixeu-vos que una vegada s'ha generat l'element, encara que no s'hagi afegit al document ja es troba a la memòria. I com que es tracta d'un objecte `jQuery`, es poden invocar pràcticament tots els seus mètodes, per exemple per afegir classes:

```
<style>
.vermell {
  color: red;
}
</style>

<div id="contenidor"></div>

<script>
  var $contenidor = $('#contenidor');
  var $paragraf = $('<p>Paragraf <span>creat</span> i <span>manipulat</span> a
      la memòria</p>');

  $paragraf.find('span').addClass('vermell');

  $contenidor.append($paragraf);
</script>
```

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

Com podeu veure, s'afegeix la classe `vermell` als elements de tipus `span` abans d'afegir-lo al document; d'aquesta manera, l'operació s'executa correctament.

Quant a l'eliminació d'elements, jQuery ofereix dos mètodes: `remove` i `empty`. El primer elimina tots els elements seleccionats per l'objecte `jQuery` a partir del qual s'invoca, mentre que el segon elimina els descendents d'aquests nodes (és a dir, els *buida*):

```
<ul id="llista1">
  <li>Element A</li>
  <li>Element B</li>
  <li>Element C</li>
</ul>

<ul id="llista2">
  <li>Element 1</li>
  <li>Element 2</li>
  <li>Element 3</li>
</ul>

<script>
  var $llista1 = $('#llista1');
  var $llista2 = $('#llista2');

  $llista1.remove();
  $llista2.empty();

  console.log("Existeix l'element amb id llista1:", document.getElementById('
      llista1'));

  console.log("Existeix l'element amb id llista2:", document.getElementById('
      llista2'));
</script>
```

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

Fixeu-vos que mentre que la primera llista ha deixat d'existir, la segona continua a la pàgina (tot i que no es veu perquè és buida) però s'han eliminat tots els seus descendents.

2.5 Utilitzar Events amb jQuery

Entre els avantatges d'utilitzar jQuery a l'hora d'afegir la detecció d'esdeveniments cal destacar-ne dos:

- Es pot assignar la detecció de múltiples esdeveniments amb una sola línia.
- La detecció d'esdeveniments s'aplica a tots els elements seleccionats.

Per afegir un detector, s'ha de generar primer un objecte jQuery: se seleccionen els elements als quals voleu afegir-lo, i seguidament s'invoca el mètode `on`, passant com a arguments una cadena amb el nom dels elements que cal detectar i la funció que es farà servir de *callback* (funció que serà cridada internament).

Cal que recordeu que tot i que als exemples s'acostuma a fer servir funcions anònimes, es pot passar qualsevol tipus de funció i, per tant, podria ser una funció amb nom definida en altre punt de l'aplicació o una variable que referenciés una funció o un mètode.

```
<ul id="llista1">
  <li>Element A</li>
  <li>Element B</li>
  <li>Element C</li>
</ul>

<ul id="llista2">
  <li>Element 1</li>
  <li>Element 2</li>
  <li>Element 3</li>
</ul>

<script>
  var $llista1 = $('#llista1 li');
  var $llista2 = $('#llista2 li');

  $llista1.on('click mouseenter mouseout', function () {
    $(this).toggleClass('vermell');
  });

  $llista2.dblclick(function () {
    $(this).remove();
  });
</script>
```

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

Com es pot apreciar, s'ha afegit la detecció per als *events* `click`, `mouseenter` i `mouseout` als elements de la primera llista, de manera que aquests es posen de color vermell quan hi entra el cursor o quan es fa clic sobre ells, mentre que en sortir es fa l'acció inversa.

D'altra banda, a la segona llista s'ha afegit la detecció de l'*event* `dblclick`, que es dispara en fer doble clic sobre algun dels elements, i elimina aquest element de la llista en disparar-se.

Mètode remove() al DOM

Existeix un mètode `remove` a l'especificació del DOM que es pot cridar directament sobre l'element, però no és compatible amb navegadors antics, particularment amb versions d'Internet Explorer anteriors a Edge.

Cal destacar que el context de la funció (`this`) fa referència a l'element en el qual s'ha detectat l'esdeveniment; així doncs, a la primera llista es genera un objecte `jQuery` a partir d'aquest i s'activa o desactiva la classe `vermell`. De la mateixa manera, en el segon cas, per evitar incompatibilitats, es genera l'objecte i s'invoca el mètode `remove` per eliminar-lo.

Dreceres d'events

Podeu trobar una llista completa de les dreceres d'*events* en l'enllaç següent: goo.gl/BJMUZh.

Fixeu-vos que en el segon cas, en lloc d'invocar el mètode `on` i passar una cadena de text amb el nom de l'*event*, s'ha invocat `dblclick`. Això és possible perquè jQuery ofereix una llarga llista de mètodes que poden invocar-se com dreceres (`click`, `dblclick`, `load`, etc.), és a dir, pot invocar-se aquest mètode passant com a únic argument la funció que serà invocada en detectar-se l'esdeveniment. Com és d'esperar, quan es fan servir dreceres hi ha una limitació important: només es pot detectar un únic *event*.

En cas que necessiteu eliminar la detecció d'esdeveniments només cal fer servir el mètode `off`:

```
<style>
  div {
    width: 100px;
    height: 100px;
    float: left;
    background-color: grey;
    margin: 1px;
  }
  .vermell {
    background-color:red;
  }
</style>

<h1>Selecciona un quadre</h1>
<div></div>
<div></div>
<div></div>
<div></div>
<div></div>

<script>
  $quadres = $('div');

  $quadres.click(function() {
    $(this).addClass('vermell');
    $quadres.off();
    $('h1').html('Quadre seleccionat');
  });
</script>
```

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

Com es pot apreciar, una vegada es clica qualsevol dels quadres, el quadre en qüestió es mostra de color vermell (s'aplica la classe `vermell`), i s'elimina la detecció d'esdeveniments de tots els quadres invocant el mètode `off` sense passar-hi cap argument.

Una altra opció que ofereix el mètode `off` és eliminar només alguns dels *events* de la detecció, com es pot comprovar en l'exemple següent:

```
<style>
  .vermell {
    color:red;
  }
</style>

<ul>
  <li>Element A</li>
  <li>Element B</li>
  <li>Element C</li>
</ul>

<script>
  var $elements = $('li');

  $elements.on('click mouseenter mouseout', function () {
    $(this).toggleClass('vermell');
  });

  $elements.on('click', function(function() {
    $(this).off('mouseenter mouseout');
  })
</script>
```

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

Fixeu-vos que **s'han afegit dos detectors diferents per a l'*event*** `click` i tots dos es criden en fer clic sobre qualsevol dels elements. El primer s'encarrega de canviar el color de la classe, mentre que el segon elimina la detecció d'*events* per a `mouseenter` i `mouseout`. És a dir, una vegada es fa clic sobre un element aquest només respondrà a l'*event* `click`.

Alternativament es podria haver fet servir la drecera `click` en lloc del mètode `on` per al segon detector; el resultat hauria estat el mateix:

```
1 $elements.click( function() {
2   $(this).off('mouseenter mouseout');
3 })
```

2.6 Crear connectors per jQuery

En el context de la biblioteca jQuery, un connector (*plug-in*, en anglès) és un component que afegeix noves funcionalitats a la biblioteca. És a dir, una vegada tenim el connector carregat,S aquest s'afegeix automàticament a la biblioteca i és accessible per a tots els objectes jQuery.

Les galeries d'imatges o els carrusels d'anuncis o productes són exemples habituals de l'ús d'aquests connectors, però es poden trobar tot tipus de connectors en diferents repositoris, a més a més dels que podeu crear vosaltres mateixos.

El gestor de paquets **npm** està basat en Node.js i per fer-lo servir s'ha de tenir istal·lat.

Node.js és una tecnologia que permet utilitzar JavaScript al servidor.

Un dels espais web on es poden trobar aquests repositoris és al lloc web d'npm (www.npmjs.com). Aquí es pot localitzar fàcilment tot tipus de programari en JavaScript, tant per als navegadors com per al servidor (per utilitzar amb Node.js).

Fixeu-vos que només una part d'aquests paquets són connectors de jQuery; per localitzar-los es recomana fer servir aquest enllaç: www.goo.gl/Lc9c6Z, que filtra directament els components que inclouen la paraula `jquery-plugin`. Si proveu d'utilitzar el cercador del mateix lloc, veureu que el resultat és diferent.

Per descomptat, l'ús d'aquest gestor de paquets és opcional i podeu utilitzar els vostres propis connectors directament o descarregar-los de qualsevol altre repositori de tercers sense cap problema. Però en aquest últim cas és preferible fer servir una font fiable per evitar trobar-vos que s'està manipulant la vostra aplicació de forma malintencionada.

A continuació veureu com es pot crear el vostre propi connector per a jQuery, de manera que pugui ser utilitzat per tots els components de la vostra aplicació sense haver de fer-lo global.

En primer lloc, cal tenir clar quin serà el comportament que ha de tenir. En aquest cas s'ha decidit que les característiques del connector seran les següents:

- L'objectiu del connector és facilitar l'addició (i eliminació) d'elements a una o més llistes.
- S'afegiran dos botons (+ i -) i un títol que es mostrarà a la part superior de la llista.
- En clicar el botó + s'afegirà un quadre de text en l'última posició de la llista que indicarà que s'ha d'escriure un text.
- Quan es perdi el focus, si es troba algun valor, s'establirà aquest com a text de la llista.
- El quadre de text s'eliminarà o es reemplaçarà en qualsevol cas.
- Si s'ha entrat un valor, s'afegirà automàticament un nou quadre de text al final de la llista.
- Sempre que s'afegeixi un nou quadre, aquest rebrà el focus.
- Si es clica el botó -, s'eliminarà l'últim element de la llista.

Fent una primera implementació d'aquesta funcionalitat es pot obtenir un codi similar al següent:

```
<style>
  ul {
    list-style-type: none;
    padding: 0;
    width: 200px;
    float: left;
    margin: 5px;
  }

  ul div {
    border-bottom: 1px gray dotted;
    padding-bottom: 2px;
  }
</style>

<ul>
</ul>
<ul>
</ul>
<ul>
</ul>

<script>
  $llistes = $('ul');

  $llistes.each(function() {
    var $barraEines = $('<div>Gestor Llistes </div>');
    var $botoMes = $('<button>+</button>');
    var $botoMenys = $('<button>-</button>');

    $barraEines.append($botoMes);
    $barraEines.append($botoMenys);

    var $contenidor = $(this);
    $contenidor.prepend($barraEines);

    $botoMes.click(function() {
      var $quadre = $('<li><input type="text" placeholder="Escriu aquí..."/></
          li>');

      $quadre.on('change focusout', function() {
```

```
        var valor = $quadre.find('input').val();

        if (valor) {
          $quadre.html(valor);
          $botoMes.trigger('click');
        } else {
          // Si no hi ha cap contingut, s'elimina
          $quadre.remove();
        }
      });
      $contenidor.append($quadre);
      $quadre.find('input').focus();
    });

    $botoMenys.click(function() {
      $contenidor.find('li:last-child').remove();
    });
  });
</script>
```

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

Primerament es fa una selecció de tots els elements de tipus llista, i a continuació s'itera sobre aquests elements per afegir la funcionalitat a totes les llistes trobades.

Es crea un contenidor per a la barra d'eines que inclou el títol `Gestor Llistes` seguit dels dos botons `+` i `-`, als quals se'ls afegeix la detecció de l'*event* `click`.

En clicar el primer botó es crea un nou element a la llista que inclou un quadre d'entrada de text i que detecta els *events* `change` i `focusout`, de manera que si aquest canvia o perd el focus és substituït per un text pla –si s'ha entrat algun valor– o l'elimina –en cas contrari–.

El mètode `trigger` permet disparar un esdeveniment manualment.

Addicionalment, en crear-se l'element de la llista, s'estableix el focus en el quadre d'entrada i això permet introduir dades de forma més fluida. D'altra banda, una vegada es detecta el canvi i s'estableix com a text, es genera automàticament un nou element a la llista per omplir (per simplificar s'ha fet disparant l'*event* `click` sobre el botó `+`).

En clicar sobre el botó `-` s'elimina l'última fila de la llista fins que no en queda cap. No cal fer cap comprovació addicional.

Fixeu-vos que també s'ha afegit el codi CSS per formatar les llistes. Aquest codi s'hauria d'incloure amb el connector, de manera que es concretés més per no interferir en l'estructura de la pàgina on es faria servir el connector, per exemple, afegint-hi alguna classe pròpia. Substituïu les declaracions de `$contenidor` i de la `$barraEines` respectivament al codi JavaScript per les següents:

```
var $contenidor = $(this);
$contenidor.addClass('gestor-llistes');

var $barraEines = $('<div>Gestor Llistes </div>');
$barraEines.addClass('barra-eines');
```

I substituïu el codi CSS pel següent:

```
ul.gestor-llistes {
  list-style-type: none;
  padding: 0;
```

```
  width: 200px;
  float: left;
  margin: 5px;
}

ul.gestor-llistes div.barra-eines {
  border-bottom: 1px gray dotted;
  padding-bottom: 2px;
}
```

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

Tot i que el comportament és el mateix, ja no hi haurà conflictes quan es faci servir el connector. El següent pas és convertir-lo en un connector de jQuery. Per fer-ho només s'ha d'afegir una funció anomenada *arrodonir* a `$.fn`, i a partir d'aquest moment aquesta funció passarà a estar disponible a tots els objectes `jQuery`. Reemplaceu el codi JavaScript pel següent:

```
$(document).ready(function() {
  $('ul').gestionar();
});

$.fn.gestionar = function() {

  this.each(function() {
    var $contenidor = $(this);
    $contenidor.addClass('gestor-llistes');

    var $barraEines = $('<div>Gestor Llistes </div>');
    $barraEines.addClass('barra-eines');

    var $botoMes = $('<button>+</button>');
    var $botoMenys = $('<button>-</button>');

    $barraEines.append($botoMes);
    $barraEines.append($botoMenys);

    $contenidor.prepend($barraEines);

    $botoMes.click(function() {
      var $quadre = $('<li><input type="text" placeholder="Escriu aquí..."/></
          li>');

      $quadre.on('change focusout', function() {
        var valor = $quadre.find('input').val();

        if (valor) {
          $quadre.html(valor);
          $botoMes.trigger('click');
        } else {
          // Si no hi ha cap contingut s'elimina
          $quadre.remove();
        }
      });
      $contenidor.append($quadre);
      $quadre.find('input').focus();
    });

    $botoMenys.click(function() {
      $contenidor.find('li:last-child').remove();
    });
  });
}
```

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

Com es pot apreciar, només hi ha tres canvis molt lleugers:

- Es crida el mètode `gestionar` directament sobre l'objecte `jQuery` que conté la selecció de les llistes.
- S'ha posat tot el codi del connector dins de la funció `$.fn.gestionar`: això és el que possibilita cridar la funció des de qualsevol objecte `jQuery`.
- En lloc d'iterar sobre `$llistes` es fa sobre `this`, ja que en el context del connector `this` fa referència a l'objecte jQuery des del qual s'ha cridat, és a dir, el que selecciona totes les llistes en el nostre cas.

Per millorar el connector es pot implementar de forma que accepti arguments per flexibilitzar encara més la seva utilitat. Per exemple, per afegir un títol i missatge del quadre de text personalitzat:

```
<style>
  ul.gestor-llistes {
    list-style-type: none;
    padding: 0;
    width: 200px;
    float: left;
    margin: 5px;
  }

  ul.gestor-llistes div.barra-eines {
    border-bottom: 1px gray dotted;
    padding-bottom: 2px;
  }
</style>

<ul id="Nom">
</ul>
<ul id="Generic">
</ul>
<ul id="Curs">
</ul>

<script>
  $(document).ready(function() {
    $('ul#Nom').gestionar('Noms', 'Introdueix un nom');
    $('ul#Curs').gestionar('Curs', 'Introdueix un curs');
    $('ul#Generic').gestionar();
  });

  $.fn.gestionar = function(titol, placeholder) {
    if (!titol) {
      titol = "Gestor de llistes";
    }
    if (!placeholder) {
      placeholder = "Escriu aquí..."
    }

    this.each(function() {
      var $contenidor = $(this);
      $contenidor.addClass('gestor-llistes');

      var $barraEines = $('<div>' + titol + ' </div>');
      $barraEines.addClass('barra-eines');

```

```
45      var $botoMes = $('<button>+</button>');
46      var $botoMenys = $('<button>–</button>');
47
48      $barraEines.append($botoMes);
49      $barraEines.append($botoMenys);
50
51      $contenidor.prepend($barraEines);
52
53      $botoMes.click(function() {
54        var $quadre = $('<li><input type="text" placeholder="' + placeholder +
              '"/></li>');
55
56        $quadre.on('change focusout', function() {
57          var valor = $quadre.find('input').val();
58
59          if (valor) {
60            $quadre.html(valor);
61            $botoMes.trigger('click');
62          } else {
63            // Si no hi ha cap contingut s'elimina
64            $quadre.remove();
65          }
66        });
67        $contenidor.append($quadre);
68        $quadre.find('input').focus();
69      });
70
71      $botoMenys.click(function() {
72        $contenidor.find('li:last-child').remove();
73      });
74    });
75
76  }
77 </script>
```

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

Només ha calgut afegir els paràmetres a la funció `gestionar` i utilitzar aquests valors internament: el paràmetre `titol` per substituir el títol que mostra el connector i el paràmetre `placeholder` com a atribut `placeholder` del quadre de text.

Un altre canvi que s'ha fet és afegir un identificador per a cada llista al codi HTML. Això no és cap requisit, s'ha fet així per simplificar la selecció individual i mostrar com cadascuna de les llistes aplica el mateix connector d'una manera personalitzada, gràcies a la parametrització.

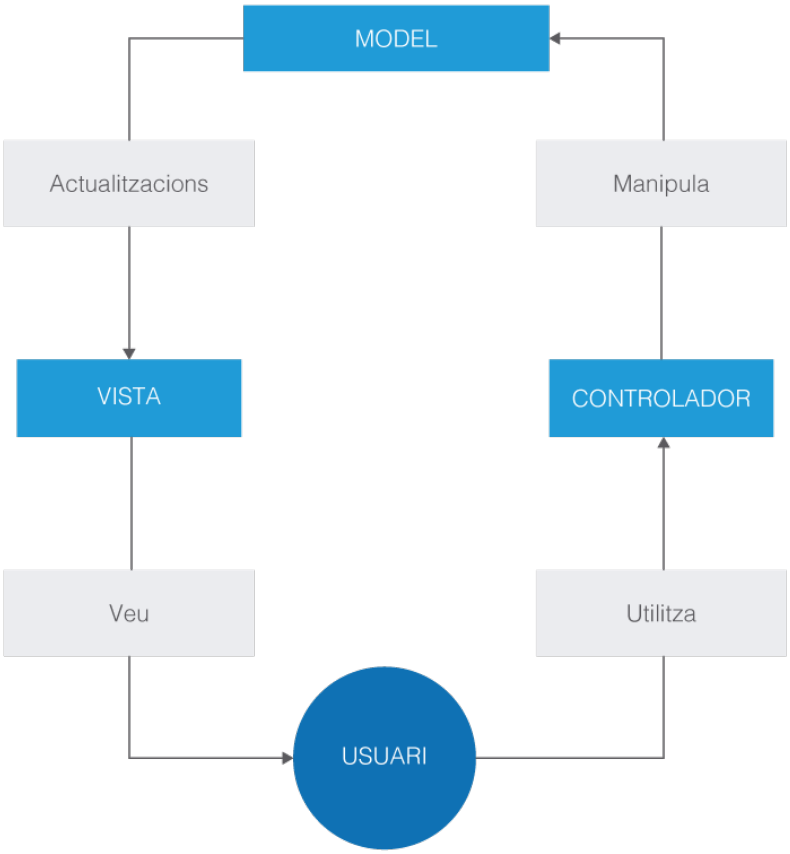
2.7 Model-vista-controlador (MVC)

El model-vista-controlador és un patró de disseny complex utilitzat per implementar interfícies d'usuari, que separa l'aplicació en tres parts, com es pot apreciar a la figura 2.3:

- El **model**: encarregat de gestionar les dades i la lògica de l'aplicació.
- Les **vistes**: representacions de la informació que es mostra a l'usuari. Hi pot haver diferents vistes per a una mateixa informació.

- Els **controladors**: encarregats de gestionar l'entrada de dades des de les vistes i fer les operacions necessàries sobre el model.

Figura 2.3. Reprentació típica dels elements del patró MVC



Fixeu-vos que mentre que la **vista** generalment estarà programada amb HTML, CSS i JavaScript i s'executa en el client (el navegador), el **model** i el **controlador** poden trobar-se en un servidor remot i fer servir un llenguatge diferent de JavaScript, per exemple PHP o Java. Per tant, la implementació d'aquest patró pot requerir la utilització de diferents tecnologies.

Tot i que a cop d'ull sembla senzill, és molt fàcil confondre quin component pertany a cada part, ja que en programació es treballa amb conceptes força abstractes. Vegeu els següents dos exemples d'aplicació d'aquest patró:

Exemple de sistema d'autenticació

Un exemple molt simple seria aplicar aquest patró a un sistema d'autenticació; la divisió de les tres parts seria la següent:

- **Model**: encarregat de comprovar si la informació d'un usuari i la seva contrasenya són correctes, si l'usuari és actiu, etc.
- **Controlador**: rep l'entrada del client (nom d'usuari i contrasenya) i passa aquestes dades al model que retornarà si són correctes o no. En qualsevol cas es retornarà una resposta al model (o es generarà una nova vista) amb aquesta informació (accés concedit o denegat).
- **Vista**: un formulari per autenticar l'usuari amb dos quadres de text (un per al nom d'usuari i un altre per a la contrasenya), i un botó per enviar la petició al controlador.

D'aquesta manera, es podrien afegir nous sistemes d'autenticació sense haver de tocar el **model**, només afegint nous mètodes al controlador (o nous controladors) i les **vistes** necessàries per interactuar amb el sistema. Per exemple, des de la **vista** es podria fer una petició asíncrona al **controlador** i que aquest, com a resposta, retornés un objecte JSON amb la informació de l'intent (èxit o error) i, en conseqüència, la vista mostraria el missatge corresponent.

Un exemple de com es podrien utilitzar diferents vistes a partir d'una mateixa informació seria el següent:

Exemple d'utilització de múltiples vistes

Suposeu que treballeu en una aplicació que gestiona les dades d'un institut i entre aquestes dades es troba la informació de tots els alumnes i els cursos.

Aquesta informació és proporcionada pel **model**, es realitzen les consultes a través del **controlador** i com a resposta es podrien generar diferents **vistes** (segons l'entrada enviada al controlador). Per exemple, es podrien generar les següents vistes:

- Una vista que mostri el llistat de tots els alumnes de l'institut ordenats alfabèticament.
- Una vista que mostri la mateixa informació però agrupada per cursos.
- Una vista que mostri una gràfica amb la informació demogràfica dels alumnes.
- Una vista que mostri un mapa amb xinxetes indicant la localització del domicili de cada alumne.

Cal destacar que aquest és un concepte avançat i no és recomanable que els programadors novells intentin implementar-lo a les seves aplicacions (excepte a les més senzilles). Afortunadament hi ha molts entorns de treball (*frameworks*) que implementen aquest patró i són molt utilitzats en el desenvolupament d'aplicacions complexes com Angular o Ember i biblioteques especialitzades com React.

L'avantatge d'utilitzar un entorn de treball és que permet començar a treballar més ràpidament, ja que inclouen tots els elements necessaris per crear una aplicació. En canvi, si es fa servir una biblioteca com React, serà necessari afegir altres biblioteques per afegir certes funcions (o desenvolupar-les vosaltres mateixos).

2.7.1 Angular

Angular (www.angular.io) és un dels entorns de treball més utilitzats i el matenen, principalment, enginyers de Google. Una vegada s'ha afegit ja es pot començar a treballar-hi directament, ja que inclou totes les funcionalitats per desenvolupar aplicacions complexes, sense requerir cap altra biblioteca.

La versió Angular 2 i posteriors són incompatibles amb la versió 1 i, per consegüent, la documentació relativa a versions anteriors està obsoleta i s'ha d'ignorar.

Typescript

Podeu trobar més informació sobre TypeScript en l'enllaç següent: www.typescriptlang.org.

Gulp i Grunt

Gulp (www.gulpjs.com) i Grunt (www.gruntjs.com) són eines d'automatització que treballen sota Node.js.

Tot i que es pot programar amb ES5, els desenvolupadors d'Angular recomanen utilitzar TypeScript (una variant de ES6). Així doncs, cal fer servir alguna eina d'automatització com Gulp o Grunt per generar el codi final (que ha d'estar en ES5 per fer que sigui compatible amb els navegadors actuals).

Aquest entorn de treball forma part del que es coneix com a *MEAN stack* (un *stack* és un 'conjunt de tecnologies') que consisteix en l'ús de **M**ongoDB (base de dades NoSQL), **E**xpress.js (servidor web per a Node.js), **A**ngular a l'entorn client i **N**ode.js com a entorn d'execució al servidor.

Popularitat de les diverses tecnologies

Podeu trobar més informació sobre quines són les tecnologies més populars i quins llocs web les utilitzen en l'enllaç següent: www.goo.gl/9nkkNN.

Entre les aplicacions webs desenvolupades amb Angular hi ha www.youtube.com, www.freelancer.com, www.telegram.org i www.udemy.com.

2.7.2 Ember

Ember (www.emberjs.com) és un altre entorn de treball molt utilitzat, tot i que molt lluny de la popularitat d'Angular. Aquest fa servir un sistema de *convenció sobre configuració*, és a dir, en lloc d'haver de configurar tota l'aplicació, s'han de respectar una sèrie de normes (noms de fitxers, rutes, controladors, etc.) i automàticament són reconeguts.

Un dels inconvenients és que la seva corba d'aprenentatge és més gran que en el cas d'Angular, i un altre és que el sistema de plantilles que utilitza incrusta etiquetes `script` per tot el document, ja que és així com fa la substitució de codi per continguts de text (tot i que això està previst que canviï en el futur).

Entre les aplicacions webs desenvolupades amb aquest entorn de treball es troben www.twitch.tv, www.digitalocean.com, www.heroku.com i www.sitepoint.com.

2.7.3 React

A diferència d'Angular i Ember, React (facebook.github.io/react) no és un entorn de treball, sinó una biblioteca desenvolupada per Facebook. El seu punt fort és el desenvolupament d'aplicacions d'una sola pàgina (precisament, Facebook el va desenvolupar amb aquest objectiu).

És a dir, en cas de requerir comportaments més complexos (com per exemple l'encaminament de pàgines), s'ha de recórrer a altres biblioteques complementàries. D'altra banda, com que es tracta d'una biblioteca, és molt més fàcil d'integrar en altres projectes o fins i tot combinar-la amb altres entorns de treball.

Entre les aplicacions webs desenvolupades amb aquesta biblioteca hi ha www.facebook.com, www.whatsapp.com, www.imdb.com, www.instagram.com i www.netflix.com.

2.7.4 Components Web

La idea al darrere dels components web és poder crear nous elements de la interfície que siguin reutilitzables i s'integrin perfectament al DOM. Aquests components inclouen tant el format com els detectors d'esdeveniments i qualsevol codi necessari per funcionar.

Especificació del W3C dels components web

Podeu trobar tota la informació referent a l'especificació del W3C dels components web al següent enllaç: www.goo.gl/OOsGNx.

> Tot i que els components web actualment no són viables, és important conèixer-ne les característiques i saber com s'han implementat components similars.

Per exemple, seria possible crear un component "FitxaAlumne" que mostrés tota la informació d'un alumne correctament formatada a partir de les seves dades, i que aquest es mostrés al document HTML només afegint-hi l'etiqueta corresponent (similar a `<fitxaalumne></fitxaalumne>`).

Cal destacar que mentre molts entorns de treball permeten crear elements personalitzats, en inspeccionar el codi es pot comprovar que el que s'ha fet és reemplaçar o embolcallar l'etiqueta original amb tota una sèrie d'elements incrustats per l'entorn de treball per donar format i afegir els detectors d'esdeveniments. En canvi, utilitzant components web només es mostraria l'etiqueta corresponent.

Estat de les tecnologies

Podeu trobar l'estat de les tecnologies utilitzades pels components web als navegadors més populars en l'enllaç següent: www.goo.gl/5aUF9E.

Aquests components consisteixen en l'aplicació de quatre tecnologies diferents; malauradament, no estan disponibles en tots els navegadors d'escriptori i menys encara en els navegadors de dispositius mòbils. Les tecnologies són les següents (totes es consideren experimentals):

- **Custom elements**: capacitat de crear nous elements i etiquetes HTML personalitzats.
- **HTML Templates**: utilització de plantilles directament al codi HTML, que no són mostrades al navegador però es poden afegir via JavaScript.
- **Shadow DOM**: permet encapsular el codi JavaScript i CSS d'un component web per evitar que aquest es barregi amb el de la resta de l'aplicació.
- **HTML Imports**: capacitat d'importar altres fitxers HTML.

Així doncs, per poder utilitzar qualsevol d'aquestes tecnologies és molt probable que hàgiu d'activar el mode experimental del vostre navegador i, conseqüentment, no es poden utilitzar en aplicacions webs que hagin d'emprar altres usuaris.

Cal destacar que encara que fa anys que s'hi treballa, encara no existeix una definició definitiva i per aquesta raó se n'han fet diferents interpretacions.

Google Polymer

Podeu trobar més informació sobre Google Polymer en l'enllaç següent: www.polymer-project.org/1.0.

Per exemple, Mozilla Firefox no implementarà *HTML Imports* perquè els consideren insegurs. Google va presentar Google I/O 2013 Polymer, la primera versió del seu entorn de treball basat en les tecnologies dels components web. I d'altra

banda, els entorns de treball Angular (directrius), Ember (components) i React (components) ofereixen una funcionalitat similar a la dels components web.

Mecanismes de programació asíncrona client-servidor

Xavier Garcia Rodríguez

Desenvolupament web en l'entorn client

Índex

Introducció

Originalment, per actualitzar els continguts d'una pàgina s'havia de recarregar completament: no era possible afegir nous continguts dinàmicament. Això provocava una gran despesa de recursos tant a la banda del client com a la del servidor. La solució a aquest problema és la utilització d'AJAX, un conjunt de tecnologies que permeten fer peticions asíncrones al servidor, processar-ne les respostes i actualitzar la pàgina.

Aquesta tecnologia permet crear aplicacions i pàgines web que només cal carregar una vegada (per exemple Facebook) i afegeixen nous elements a mesura que l'usuari selecciona diferents opcions o es desplaça pels continguts.

En aquesta unitat aprendreu a implementar les vostres pròpies solucions AJAX per connectar amb serveis web o llegir fitxers tant en dominis propis com de tercers, a fer servir la biblioteca jQuery per simplificar el vostre codi i a connectar amb fonts de dades obertes.

Primerament, a l'apartat **"Comunicació asíncrona amb JavaScript"**, es discutiran les dificultats que presenta provar el codi AJAX. Seguidament, es descriuran les característiques i funcionalitats de l'objecte `XMLHttpRequest` abans de passar a la descripció de les dues arquitectures de serveis web més utilitzades: SOAP i REST. A continuació, es tractaran mètodes alternatius per fer peticions sobre altres dominis, s'explicarà com accedir a dades obertes i es descriurà una altra tecnologia que permet establir connexions a través de la xarxa. A més a més, es mostraran i compararan les opcions de depuració de peticions de Google Chrome i Mozilla Firefox.

A continuació, a l'apartat **"Programació de la comunicació asíncrona amb jQuery"**, es tractarà la creació i enviament de peticions utilitzant la biblioteca jQuery, que exposa una interfície més clara, permet utilitzar-la per l'enviament de peticions AJAX i JSONP i simplifica la utilització de paràmetres. Se'n descriuran les opcions, com connectar a serveis web tant propis com de tercers i alguns mètodes d'ajuda per codificar els paràmetres que cal enviar.

Per assimilar correctament els coneixements que comprenen aquesta unitat és imprescindible fer els exercicis i activitats, així com provar els exemples allotjats en CodePen i consultar la documentació de l'objecte `XMLHttpRequest` i la biblioteca jQuery en cas de dubte.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions web dinàmiques, reconeixent i aplicant mecanismes de comunicació asíncrona entre client i servidor.

- Avalua els avantatges i els inconvenients d'utilitzar mecanismes de comunicació asíncrona entre client i servidor web.
- Analitza els mecanismes disponibles per a l'establiment de la comunicació asíncrona.
- Utilitza els objectes relacionats.
- Identifica les seves propietats i els seus mètodes.
- Utilitza comunicació asíncrona en l'actualització dinàmica del document web.
- Utilitza diferents formats en l'enviament i en la recepció d'informació.
- Programa aplicacions web asíncrones de manera que funcionin en diferents navegadors.
- Classifica i analitza llibreries que facilitin la incorporació de les tecnologies d'actualització dinàmica a la programació de pàgines web.
- Crea i depura programes que utilitzin aquestes llibreries.

1. Comunicació asíncrona amb JavaScript

Als inicis d'internet, cada vegada que un usuari realitzava una acció, la pàgina on es trobava s'havia de recarregar completament perquè reflectís aquests canvis (per exemple, quan s'afegia un ítem en un carretó electrònic).

Això suposava una mala experiència a l'usuari, perquè s'havia d'esperar que la pàgina es descarregués un altre cop sense poder interactuar-hi, per mínim que fos el canvi.

Descàrrega de fitxers i data d'expiració

Habitualment, en obrir una pàgina web, els navegadors emmagatzemen fitxers descarregats per no haver de tornar-los a recarregar en visites posteriors (imatges, codi CSS, codi JavaScript, etc.). La data d'expiració –si n'hi ha– depèn de la configuració del servidor, tot i que aquests fitxers es poden eliminar manualment del navegador.

Per altra banda, la càrrega del servidor també s'incrementava perquè havia de tornar a enviar tots els continguts per a cada petició que es feia, i en molts casos s'havien de fer consultes a la base de dades per tornar a generar els mateixos continguts que ja havia descarregat l'usuari inicialment.

1.1 Introducció a AJAX

Per resoldre el problema de la càrrega dinàmica de dades es fa servir AJAX (Asynchronous JavaScript and XML), un conjunt de tecnologies que permeten actualitzar els continguts d'una pàgina o aplicació web sense necessitat de recarregar-la. Les tecnologies que la componen són les següents:

- HTML i CSS per a la representació.
- El model d'objectes del document (DOM) per mostrar i manipular les dades dinàmicament.
- L'objecte `XMLHttpRequest`, que permet la comunicació asíncrona.
- Els formats JSON o XML per a l'intercanvi de dades.
- JavaScript per lligar totes aquestes tecnologies.

Com es pot apreciar, totes aquestes tecnologies estan integrades amb JavaScript i, consegüentment, no és necessari fer servir cap llibreria per treballar-hi (tot i que és habitual fer servir jQuery perquè d'aquesta manera no cal implementar la gestió de les peticions).

"Parsejar" i "Analitzar sintàcticament"

La utilització del terme *parsejar* és incorrecte però és possible trobar materials que ho fan servir en lloc d'*analitzar sintàcticament*. Cal tenir en compte que en tots dos casos es fa referencia a la mateixa operació.

Cal destacar que, tot i que originalment es va utilitzar XML per a l'intercanvi de dades, no és un requisit: es poden fer servir diferents formats perquè, com a resposta, admeten tant continguts en format XML com continguts en text pla que es poden analitzar sintàcticament (aquesta acció es coneix com a *parse* en anglès) per convertir-los a altres formats, per exemple, a JSON o fins i tot HTML.

L'objecte `XMLHttpRequest` permet realitzar peticions en segon pla (són asíncrones), de manera que l'aplicació no queda bloquejada quan es fa la petició, i una vegada es rep la resposta (amb dades o amb un missatge d'error) és processada per l'aplicació.

Fer servir aquesta tecnologia també permet crear aplicacions `single-page`, que consisteixen en una única pàgina a la qual es mostren uns continguts o uns altres segons les accions de l'usuari, sense haver de sortir-ne, o les pàgines amb desplaçament vertical (*scroll*) infinit, en què una vegada s'arriba al final de la pàgina es carreguen més continguts (com per exemple el llistat de notícies de Facebook o la cerca d'imatges de Google).

Un ús molt freqüent és la càrrega dinàmica de les dades de les llistes desplegables; per exemple, quan se selecciona una província d'un desplegable és habitual que es realitzi una petició al servidor que retorna el llistat de localitats que són afegides en una altra llista. D'aquesta manera, s'evita haver de descarregar totes les localitats cada vegada que un usuari visita la pàgina.

Exemple de carrega de fitxers mitjançant AJAX

El joc IOC Invaders (www.goo.gl/fMVSbs) fa servir AJAX per carregar la informació dels nivells i els enemics. Podeu trobar el codi font a www.goo.gl/LNiK94.

Un altre exemple d'ús seria la càrrega de nivells a un joc HTML, en el qual la informació s'obté d'un fitxer amb format JSON o XML, de manera que el client descarrega el motor del joc i aquest realitza les peticions necessàries per carregar els nivells, la informació sobre enemics o el rànquing. Això facilita molt la tasca de desenvolupar aquest tipus de continguts, ja que es poden crear editors independents (fins i tot en altres llenguatges) i fa possible ampliar el joc sense haver de tocar-ne la implementació.

Tot i que la connexió entre l'aplicació i el servidor s'inicia i acaba amb la petició, és possible crear sistemes de notificacions o de comunicació (tipus xat) realitzant peticions de forma continuada, fent servir els temporitzadors `setTimeout` i `setInterval`. Cal tenir en compte que aquest sistema no és gens eficient i en aquests casos és recomanable fer servir altres tecnologies com els Web Sockets, que permeten establir una connexió directa amb el servidor.

A banda de les peticions que podeu fer vosaltres directament, algunes llibreries i paquets de desenvolupament (com l'API de Google Maps) poden realitzar les seves pròpies peticions: per aquesta raó, en carregar una pàgina web no és gens estrany que hi hagi peticions fetes per tercers.

1.2 Dificultats a l'hora de provar el codi AJAX

Abans de començar a desenvolupar aplicacions que utilitzen AJAX s'ha de tenir en compte que hi ha algunes dificultats a l'hora de comprovar-ne el funcionament que cal conèixer.

Política del mateix origen

Es pot trobar més informació a l'enllaç següent: www.goo.gl/KgoIFn.

Com a mesura de seguretat, els navegadors apliquen la *política del mateix origen* (*same-origin policy*), que només permet realitzar peticions asíncrones dintre del mateix origen, és a dir, tant l'aplicació que realitza la petició com l'URL al qual s'envia han d'originar-se al mateix domini. No està permès fer aquestes peticions, ni tan sols entre subdominis, per tant, si una aplicació que es troba a `www.example.com` fa una petició a `api.example.com` el navegador la bloquejarà.

Primer de tot creeu un fitxer amb el següent codi HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script>
    // Escriviu aquí el vostre codi
    </script>
  </head>
  <body>
  </body>
</html>
```

I seguidament afegiu el següent codi dintre de les etiquetes `script`:

```
if (window.XMLHttpRequest) {
  httpRequest = new XMLHttpRequest();
} else if (window.ActiveXObject) {
  httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
  console.log("Objecte creat a partir d'ActiveXObject.");
} else {
  console.error("Error: Aquest navegador no admet AJAX.");
}

httpRequest.onreadystatechange = function () {
  console.log("Ha canviat l'estat de la petició");
}

httpRequest.open('GET', 'http://www.example.com', true);
httpRequest.send(null);
```

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

Aquest codi és completament correcte, però l'adreça de petició (establerta a httpRequest.open) indica que el domini al qual s'envia és `www.example.com`; així doncs, qualsevol petició enviada des d'altres dominis llançarà una excepció que es pot consultar a la consola de les eines de desenvolupador:

```
XMLHttpRequest cannot load http://www.example.com. No 'Access-Control-Allow-
    Origin' header is present on the requested resource. Origin 'http://s.
    codepen.io' is therefore not allowed access.
```

És possible evitar aquesta limitació fent servir altres mecanismes com CORS (*cross-origin resource sharing*; www.en.wikipedia.org/wiki/Cross-origin_resource_sharing), que afegeix noves capçaleres a la petició per restringir-ne l'accés als usuaris, o JSONP (JSON amb *padding*; www.ca.wikipedia.org/wiki/JSONP) per carregar la informació som si es tractés d'un script que seguidament és processat.

Cal destacar que alguns navegadors, com Google Chrome, inclouen el `localhost` dins de la política del mateix origen i, per tant, no funcionen correctament. Per aquest motiu **es recomana fer servir Mozilla Firefox** quan es desenvolupin aplicacions que utilitzen AJAX.

Així doncs, heu de tenir en compte que pràcticament cap dels exemples d'aquest material pot funcionar a CodePen, ja que tant el vostre codi JavaScript com el servidor al qual s'envia la petició han de trobar-se en el mateix domini i, segons la versió del navegador utilitzat, les vostres peticions locals també poden ser bloquejades. De totes maneres s'ha inclòs l'enllaç a CodePen dels exemples per facilitar "copiar i enganxar" el codi.

1.3 L'Objecte XMLHttpRequest

L'objecte `XMLHttpRequest` forma part de les especificacions del W3C i exposa una sèrie de mètodes i propietats que permeten gestionar les crides asíncrones i les respostes obtingudes.

XMLHttpRequest: propietats i mètodes

Podeu trobar una llista completa de propietats i mètodes de la interfície XMLHttpRequest a l'enllaç següent: www.goo.gl/1WpgA3.

Cal destacar que la major part de les propietats que formen part de la interfície `XMLHttpRequest` només són de lectura, ja que contenen informació que s'hi afegeix externament com a resultat de la petició. A continuació podeu trobar una llista de les més utilitzades:

- **onreadystatechange**: tot i que es tracta d'una propietat, la seva funció és assignar-n'hi una que serà cridada automàticament quan es produeixi un canvi a la propietat `readyState`.
- **readyState**: retorna l'estat de la petició.
- **responseText**: retorna la resposta de la petició com a text.
- **responseXML**: retorna la resposta de la petició com a XML.

Fixeu-vos que la resposta es pot obtenir tant en text pla com en XML. Així doncs, si el format de la resposta és diferent a XML, s'haurà d'obtenir de la propietat `responseText` i seguidament analitzar-la sintàcticament per convertir-la en el format correcte (per exemple a JSON).

Quant als mètodes disponibles per realitzar les peticions bàsiques hi ha els següents:

- **open**: inicialitza la petició.
- **overrideMimeType**: permet sobreescriure el tipus multimèdia de la resposta rebuda (`text/html`, `text/plain`, `application/xml`, etc.).
- **send**: envia la petició amb les dades passades com a paràmetre que poden ser, entre d'altres, una cadena de text, un diccionari de dades o tot el document.

Respecte a la detecció d'*events*, els navegadors moderns admeten l'ús del mètode `addEventListener`, així com l'ús de les dreceres següents:

- **onload**: drecera per afegir un detector de l'*event* `load`, que es dispara en rebre la resposta.
- **onerror**: drecera per afegir un detector de l'*event* `error`, que es dispara si es produeix algun error.
- **onprogress**: drecera per afegir un detector de l'*event* `progress`, que es dispara quan s'actualitza el progrés de la resposta i permet controlar la proporció rebuda respecte al total.

1.3.1 Creació i enviament de peticions

El primer pas per poder realitzar una petició AJAX és determinar si el navegador admet l'objecte XMLHttpRequest o no. En cas que no l'admeti es comprova si existeix l'objecte `ActiveXObject` (el seu antecessor, creat per Microsoft) i, si tampoc es troba, es mostra un missatge d'error:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script>
      if (window.XMLHttpRequest) { // Mozilla, Safari, IE7+
        httpRequest = new XMLHttpRequest();
        console.log("Creat l'objecte a partir de XMLHttpRequest.");
      } else if (window.ActiveXObject) { // IE 6 i anteriors
        httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
        console.log("Creat l'objecte a partir d'ActiveXObject.");
      } else {
        console.error("Error: Aquest navegador no admet AJAX.");
      }
    </script>
  </head>

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

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

Una vegada s'ha creat l'objecte, a partir de *XMLHttpRequest* o d'*ActiveXObject*, es pot assignar una funció al mètode `onreadystatechange`, que serà invocada automàticament quan canvia l'estat de la petició; és on es gestiona què s'ha de fer amb la resposta o si es produeix un error.

Una vegada assignada la propietat `onreadystatechange` es poden fer servir els mètodes `open` i `send` per crear la petició i enviar-la:

- **open** (mètode, URL, asíncrona): proporciona la informació per crear la petició. El primer paràmetre determina el mètode que es farà servir per a la petició (`GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`), el segon és l'URL en el qual es realitzarà la petició, i el tercer (opcional) indica si es vol fer la invocació asíncronament (true) o síncronament (false), cosa que bloquejaria l'execució fins a rebre'n la resposta. Aquest últim paràmetre és opcional, i si s'omet, la petició es fa de manera asíncrona.
- **send** (params): envia la petició prèviament preparada amb els paràmetres especificats. Aquests paràmetres han d'estar en un format que pugui interpretar el servidor de destí.

Per realitzar proves bàsiques amb AJAX no cal tenir un servidor funcionant, és possible crear els fitxers de prova manualment en una carpeta que sigui accessible des de la pàgina (com les imatges i el codi CSS i JS).

Afegiu dins de la mateixa carpeta del vostre codi un fitxer anomenat `provincies.xml` amb el contingut següent:

```
<provincies>
  <provincia>Barcelona</provincia>
  <provincia>Girona</provincia>
  <provincia>Lleida</provincia>
  <provincia>Tarragona</provincia>
</provincies>
```

A més a més, cal inserir el següent codi a l'exemple anterior:

```
httpRequest.onreadystatechange = processarCanviEstat;

httpRequest.open('GET', 'provincies.xml', true);
httpRequest.send(null);

function processarCanviEstat() {
  console.log("Ha canviat l'estat de la petició");
}
```

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

Com es pot apreciar, una vegada es rep la resposta s'invoca la funció `processarCanviEstat`. És aquí on s'ha d'implementar la lògica per realitzar una acció o altra en funció de si s'ha finalitzat amb èxit o s'hi ha produït un error.

Alguns navegadors mostren un error a la consola si no s'especifica el joc de caràcters (`charset`) que ha d'utilitzar la pàgina, i si s'utilitzen caràcters no anglesos, no es mostren correctament, per tant, és recomanable especificar-ho.

En cas que el fitxer no tingui un format correcte també es mostrarà error, ja que per defecte s'espera que la resposta siguin dades en format XML. Així doncs, cal assegurar-se, per al bon funcionament dels exemples, que el fitxer XML és correcte.

El primer que s'ha de fer quan l'estat canvia és comprovar si ja s'ha completat o no. L'objecte XMLHttpRequest es pot trobar en 5 estats diferents i el seu valor s'obté a partir de la propietat `readyState`. Els noms dels valors possibles (i el seu valor) són els següents:

- **UNSENT** (0). Encara no s'ha obert la petició amb `open`.
- **OPENED** (1). S'ha obert la petició però encara no s'ha enviat.
- **HEADERS_RECEIVED** (2). S'ha enviat la petició.
- **LOADING** (3). S'està descarregant la resposta. La propietat `responseText` conté el contingut parcial.
- **DONE** (4). S'ha completat l'operació.

Cal destacar que a Internet Explorer els noms són diferents i, per consegüent, a l'hora de desenvolupar qualsevol aplicació, es recomana fer servir els valors enters i assignar-los com a pseudoconstants pròpies en lloc dels noms.

Substituïu la funció `processarCanviEstat` per la següent:

```
function processarCanviEstat() {
  if (httpRequest.readyState === XMLHttpRequest.DONE) {
    console.log("S'ha rebut la resposta. Estat actual:", httpRequest.readyState
        );
  } else {
    console.log("Encara no s'ha rebut la resposta... Estat actual:",
        httpRequest.readyState);
  }
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/mAajqy?editors=1012.

Com es pot apreciar, l'estat va canviant des de l'1 (`OPENED`) fins al 4 (`DONE`). Una vegada arribat a aquest punt es pot consultar la propietat `status` per comprovar si s'ha tingut èxit amb la petició o no. El valor d'aquesta propietat serà 200 quan no s'ha produït cap error o altres valors específics com 404 (pàgina no trobada) o 500 (error intern del servidor).

Substituïu la funció `processarCanviEstat` per la següent, a la qual s'ha afegit el codi per discriminar entre peticions amb èxit i peticions errònies:

```
function processarCanviEstat() {
  if (httpRequest.readyState === XMLHttpRequest.DONE) {
    console.log("S'ha rebut la resposta. Estat actual:", httpRequest.readyState
        );

    if (httpRequest.status === 200) {
      console.log("S'ha rebut la resposta correctament. Estat de resposta:",
          httpRequest.status);
```

```
    } else {
      console.log("S'ha produït un error en obtenir la resposta. Estat de
          resposta:", httpRequest.status)
    }

  } else {
    console.log("Encara no s'ha rebut la resposta... Estat actual:",
        httpRequest.readyState);
  }
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/QKzBZa?editors=1012.

Com es pot apreciar, primer s'ha de comprovar que el valor de `readyState` sigui 4 (`DONE`) i que l' `status` valgui 200 abans de processar la resposta.

Per acabar, només cal processar les dades i mostrar-les per pantalla, afegint la crida a la funció `processaResposta`, en cas d'èxit, i afegint-hi la nova funció:

```
function processarCanviEstat() {
  if (httpRequest.readyState === XMLHttpRequest.DONE) {
    console.log("S'ha rebut la resposta. Estat actual:", httpRequest.readyState
        );

    if (httpRequest.status === 200) {
      console.log("S'ha rebut la resposta correctament. Estat de resposta:",
          httpRequest.status);

      processarResposta(httpRequest.responseText);
    } else {
      console.log("S'ha produït un error en obtenir la resposta. Estat de
          resposta:", httpRequest.status)
    }

  } else {
    console.log("Encara no s'ha rebut la resposta... Estat actual:",
        httpRequest.readyState);
  }
}

function processarResposta(resposta) {
  var elementPre = document.createElement('pre');
  elementPre.innerHTML = resposta;
  document.body.appendChild(elementPre);
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/yaGxaR?editors=1012.

Com es pot apreciar, la funció `processarResposta` simplement crea un element i l'afegeix al cos del document, assignant la resposta com a contingut sense realitzar cap tractament, només per comprovar que funciona correctament.

Alternativament, en cas de treballar només amb navegadors moderns, es pot afegir la detecció de l'*event* `load` en lloc de controlar els canvis d'estat. Aquest mètode es va afegir a la segona versió de l'especificació i es dispara automàticament quan es completa la descàrrega. De la mateixa manera, es poden escoltar els *events* `error` i `progress` per determinar si s'ha produït un error o per actualitzar el progrés de la transmissió de dades, respectivament.

Cal recordar que com que es tracta d'*events* es pot utilitzar el mètode `addEventListener` de l'objecte `XMLHttpRequest` o les dreceres `onload`, `onerror` i `onprogress` per afegir la funció que serà cridada en detectar-se l'*event* corresponent.

> En la majoria dels casos és recomanable treballar amb els *events* `load`, `error` i `progress` en lloc de controlar els canvis d'estat.

Vegeu en l'exemple següent com el codi se simplifica molt:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script>
    if (window.XMLHttpRequest) { // Mozilla, Safari, IE7+
          httpRequest = new XMLHttpRequest();
          console.log("Creat l'objecte a partir de XMLHttpRequest.");
        } else if (window.ActiveXObject) { // IE 6 i anteriors
          httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
          console.log("Creat l'objecte a partir d'ActiveXObject.");
        } else {
          console.error("Error: Aquest navegador no admet AJAX.");
        }

        httpRequest.onload = processarResposta;

        httpRequest.open('GET', 'provincies.txt', true);
        httpRequest.send(null);

        function processarResposta() {
          var resposta = httpRequest.responseText;
          var elementPre = document.createElement('pre');
          elementPre.innerHTML = resposta;
          document.body.appendChild(elementPre);
        }
  </script>
</head>

<body>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ZpVrZP?editors=1012.

Cal destacar que les etiquetes `provincies` i `provincia` no són visibles, com és d'esperar, ja que els navegadors no les representen. En canvi, sí que es pot visualitzar l'estructura del document fent servir les eines de desenvolupador per comprovar que s'han afegit correctament.

Un altre avantatge d'aquest sistema és que ofereix un mètode senzill per controlar el progrés de la petició, ja que mitjançant la detecció de l'*event* `progress` és possible controlar el progrés de la petició. Proveu d'afegir el següent codi a continuació de l'assignació d' `onload`:

```
httpRequest.onprogress = mostrarProgres;

```

```
function mostrarProgres(event) {
  if (event.lengthComputable) {
    var progres = 100 * event.loaded / event.total;
    console.log("Completat: " + progres + "%")
  } else {
    console.log("No es pot calcular el progrés");
  }
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/amPaYO?editors=1012.

En aquest exemple, com que es tracta d'una resposta molt curta, no es pot apreciar el progrés i automàticament retorna el 100%, però en cas d'enviament de formularis o pujada de fitxers al servidor es pot utilitzar per mostrar barres de progres a l'usuari.

1.3.2 Enviament de paràmetres

Habitualment, en fer una petició AJAX cal afegir-hi un o més paràmetres. Cal distingir entre dos tipus de formats per enviar els paràmetres:

- Cadena de consulta (*query string*): els paràmetres es passen codificats a l'URL, per exemple: `http://www.example.com?nom=Maria&cognom=Campmany`.
- Dades de formulari (*form data*): els paràmetres s'inclouen a la capçalera de la petició.

Quan el mètode d'enviament és `GET` s'han de codificar els paràmetres directament a la petició (és a dir, es tracta d'una *cadena de consulta*), de manera que en cridar el mètode `open` de l'objecte `XMLHttpRequest`, l'URL ja inclou aquests paràmetres.

Per exemple, serien paràmetres vàlids els que es mostren a l'exemple següent:

```
var httpRequest = new XMLHttpRequest();
var url = 'www.example.com';
var dades = {
  nom: 'Maria',
  cognom: 'Campmany Roig'
}

var params = convertirEnParametres(dades);

httpRequest.open('GET', url + "?" + params , true);

httpRequest.onload = function() {
  // Aquí es processaria la resposta de les peticions
};

httpRequest.send(null);

function convertirEnParametres(dades) {
  var params = '';
  for (var dada in dades) {
```

```
    if (params.length > 0) {
      params += '&'
    }

    params += encodeURIComponent(dada);
    params += '=';
    params += encodeURIComponent(dades[dada]);
  }

  return params;
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/zKgxqN?editors=0010.

Com es pot apreciar, s'ha creat una funció per convertir el diccionari de dades en una cadena de text amb el format adequat per afegir-la a l'URL. Fixeu-vos que s'ha utilitzat la funció `encodeURIComponent` per convertir les dades (tant la clau com el valor) en caràcters acceptats per la codificació de l'URL, com són els accents, les titlles i els espais.

Si examineu la capçalera de petició (o la pestanya "paràmetres" de la petició a Mozilla Firefox) amb les eines de desenvolupador del navegador, veureu que es reconeixen els paràmetres `nom` i `cognom` com a paràmetres de tipus cadena de consulta.

Podeu trobar més informació sobre les eines de desenvolupador al punt "Depuració de crides AJAX" d'aquest mateix apartat.

Enviament de contingut binari mitjançant AJAX

Entre els tipus d'arguments que accepta el mètode `send` es troben `Blob` i `ArrayBufferView`, que s'utilitzen per enviar contingut binari, com per exemple una imatge.

Cal destacar que tot i que `send` és l'encarregat de fer l'enviament i passar els paràmetres a la petició, en el cas del mètode d'enviament `GET` no és ben bé així, s'ha de codificar a l'URL.

En el cas d'altres mètodes com `POST`, `PUT/PATCH` i `DELETE` la petició s'ha de fer d'una manera diferent, i hi inclou modificar la capçalera de la petició per especificar el format en el qual s'envien les dades.

Per exemple, per afegir l'enviament de dades com a cadena de text fent servir el mètode `POST` no s'ha de modificar l'URL, però una vegada oberta la petició, cal establir a la capçalera el tipus de contingut com `application/x-www-form-urlencoded`:

```
var httpRequest = new XMLHttpRequest();
var url = 'www.example.com';
var dades = {
  nom: 'Maria',
  cognom: 'Campmany Roig'
}

var params = convertirEnParametres(dades);

httpRequest.open('POST', url, true);
httpRequest.setRequestHeader('Content-type', 'application/x-www-form-urlencoded
    ');

httpRequest.onload = function() {
  // Aquí es processaria la resposta de les peticions
};

httpRequest.send(params);

function convertirEnParametres(dades) {
  var params = '';
```

```
  for (var dada in dades) {
    if (params.length > 0) {
      params += '&'
    }

    params += encodeURIComponent(dada);
    params += '=';
    params += encodeURIComponent(dades[dada]);
  }

  return params;
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/VPXXqQ?editors=0010.

Fixeu-vos que només cal especificar que el mètode d'enviament serà `POST`; una vegada oberta la petició, s'ha d'especificar el tipus de contingut com a `application/x-www-form-urlencoded` i passar els paràmetres (convertits en una cadena codificada) com a argument al mètode `send`.

En alguns casos el servei web pot esperar rebre la informació en un altre format. Per fer-ho cal especificar aquest format a la capçalera i codificar les dades de la manera apropiada. Per exemple, si el servidor accepta els paràmetres en format JSON, el codi seria el següent:

```
var httpRequest = new XMLHttpRequest();
var url = 'www.example.com';
var dades = {
  nom: 'Maria',
  cognom: 'Campmany Roig'
}

httpRequest.open('POST', url, true);
httpRequest.setRequestHeader('Content-type', 'application/json');

httpRequest.onload = function() {
  // Aquí es processaria la resposta de les peticions
  var respostaJSON = JSON.parse(httpRequest.responseText);
};

httpRequest.send(JSON.stringify(dades));
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/KNwdWv?editors=0010.

Fixeu-vos que en aquest cas el tipus de contingut s'ha establert com a `application/json` i no és necessari codificar les dades perquè es pot convertir el diccionari de dades en una cadena de text amb format JSON mitjançant `JSON.stringify`. A la resposta s'ha afegit l'operació inversa, suposant que el retorn ha de ser una cadena de text també en format JSON, de manera que `respostaJSON` contindria un diccionari de dades amb la resposta.

Cal destacar que, utilitzant aquest sistema, el navegador Mozilla Firefox no detecta correctament els paràmetres; en canvi, Google Chrome ho mostra com a *Request Payload* i es poden visualitzar els paràmetres tant com a cadena de text en format JSON com a parells de dades clau-valor.

1.3.3 Processament de respostes com a HTML

Tot i que la interfície de `XMLHttpRequest` no incorpora cap mètode específic per recuperar la resposta en format HTML, és molt fàcil fer-ho directament a partir de la resposta textual.

Primerament creeu un nou fitxer anomenat `provincies.html` amb el codi següent:

```
<h1>Provincies</h1>
<ul>
  <li>Barcelona</li>
  <li>Girona</li>
  <li>Lleida</li>
  <li>Tarragona</li>
</ul>
```

Com que aquesta estructura no es correspon amb el format XML vàlid (hauria d'haver-hi només un element arrel) i en lloc d'enviar la petició al servidor s'està fent servir un fitxer local, és necessari sobreescriure el tipus multimèdia del fitxer de manera que el navegador pugui interpretar la resposta, ja que per defecte considera que totes les respostes són de tipus XML.

Per fer-ho s'ha d'invocar el mètode `overrideMimeType` i passar el tipus `text/html` com a argument, de manera que el codi per realitzar una petició simple quedaria així:

```
<html>
  <head>
    <meta charset="utf-8">

    <script>
      if (window.XMLHttpRequest) { // Mozilla, Safari, IE7+
        httpRequest = new XMLHttpRequest();
        console.log("Creat l'objecte a partir de XMLHttpRequest.");
      } else if (window.ActiveXObject) { // IE 6 i anteriors
        httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
        console.log("Creat l'objecte a partir d'ActiveXObject.");
      } else {
        console.error("Error: Aquest navegador no admet AJAX.");
      }

      httpRequest.onload = processarResposta;
      httpRequest.onprogress = mostrarProgres;

      function mostrarProgres(event) {
        if (event.lengthComputable) {
          var progres = 100 * event.loaded / event.total;
          console.log("Completat: " + progres + "%")
        } else {
          console.log("No es pot calcular el progrés");
        }
      }

      httpRequest.open('GET', 'provincies.html', true);
      httpRequest.overrideMimeType('text/html');
      httpRequest.send(null);

      function processarResposta() {
        var resposta = httpRequest.responseText;
```

```
34          var contenidor = document.createElement('div');
35          contenidor.innerHTML = resposta;
36          document.body.appendChild(contenidor);
37        }
38      </script>
39
40    </head>
41    <body>
42    </body>
43  </html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/wzRLvB?editors=1012.

Gràcies a aquesta funcionalitat és possible enviar peticions al servidor per mostrar diàlegs que seguidament són renderitzats al client o fitxes de productes generades al servidor mitjançant plantilles (tot i que aquesta tasca es pot realitzar al client mitjançant entorns de treball com React, Ember o Angular).

1.3.4 Processament de respostes XML

XML

Podeu trobar més informació sobre XML ('llenguatge d'etiquetatge extensible') a l'enllaç següent: www.goo.gl/kBf1kz.

El format XML és un dels formats més utilitzats per intercanviar informació. Com que és el format pel qual, originalment, es transferia la informació utilitzant AJAX, l'objecte `XMLHttpRequest` disposa del mètode `responseXML` per obtenir la resposta directament en aquest format.

S'ha de tenir en compte que en cas que el servidor no especifiqui cap tipus multimèdia per obtenir la resposta (com passa amb els fitxers locals), el navegador pot esperar que es tracti de dades en format XML vàlid. Si no és així, llançarà una excepció i aturarà l'execució de l'aplicació.

Una resposta XML vàlida seria la següent:

```
1  <provincies>
2    <provincia cp="08">Barcelona</provincia>
3    <provincia cp="17">Girona</provincia>
4    <provincia cp="25">Lleida</provincia>
5    <provincia cp="43">Tarragona</provincia>
6  </provincies>
```

Perquè una resposta sigui considerada amb format XML vàlid ha de complir (entre d'altres) les condicions següents:

- Cada element ha d'estar format per una etiqueta d'apertura i una de tancament: .
- Un element pot incloure altres elements (l'element `provincies` conté 4 elements `provincia`).
- Tots els elements s'han de tancar en un ordre jeràrquic correcte, per exemple, no es pot tancar l'element `provincies` mentre algun dels elements `provincia` sigui obert.

- Ha d'existir un element arrel que engloba la resta d'elements del document (en aquest cas l'element arrel és `provincies`).
- Es poden especificar atributs assignant un valor entre cometes (`cp=‘‘08’’`).

Opcionalment, una resposta XML pot incloure un pròleg en el qual s'indiqui la versió XML i la codificació del document, entre d'altres. Per exemple, per indicar que correspon a la versió 1.0 d'XML i la codificació `ISO-859-1` (la corresponent a l'alfabet llatí, incloent-hi els diacrítics) al començament de la resposta es trobaria el codi següent:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

Podeu trobar més informació sobre el DOM a la unitat “Programació amb el DOM” d'aquest mateix mòdul.

Cal recordar que les interfícies del DOM s'apliquen també als documents XML, de manera que es poden fer servir els mateixos mètodes o propietats per tractar tant HTML com XML, tal com podeu veure en l'exemple següent. Primerament, heu de crear un fitxer anomenat `provincies2.xml` amb el codi XML:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<provincies>
  <provincia cp="08">Barcelona</provincia>
  <provincia cp="17">Girona</provincia>
  <provincia cp="25">Lleida</provincia>
  <provincia cp="43">Tarragona</provincia>
</provincies>
```

Seguidament, creeu un fitxer amb el nom “processarXML.html” i enganxeu-hi el codi següent:

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <script>
    if (window.XMLHttpRequest) {
      httpRequest = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
    } else {
      console.error("Error: Aquest navegador no admet AJAX.");
    }

    httpRequest.onload = processarResposta;
    httpRequest.open('GET', 'provincies2.xml', true)
    httpRequest.send(null);

    function processarResposta() {
      var elementArrel = httpRequest.responseXML.documentElement;
      var elements = elementArrel.childNodes;
      var llista = document.createElement('ul');

      for (var i = 0; i < elements.length; i++) {
        if (elements[i].nodeType == Node.ELEMENT_NODE) {
          var item = processarElement(elements[i]);
          llista.appendChild(item);
        }
      }

      document.body.appendChild(llista);
    }

    function processarElement(element) {
      var codiPostal = element.getAttribute('cp');
```

```
34        var provincia = element.textContent;
35        var item = document.createElement('li');
36        item.textContent = provincia + ' (CP ' + codiPostal + ')';
37        return item;
38      }
39    </script>
40  </head>
41
42  <body>
43  </body>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/EgGBEZ?editors=1012.

Com es pot apreciar, s'ha descompost el processament de la resposta en dues funcions. Primer s'invoca la funció `processarResposta`, que obté el node arrel (que correspon a `provincies`) i a partir d'aquest obté la llista de nodes descendents (propietat `childNodes` de la interfície `Node` del DOM).

Cal destacar que la propietat `responseXML` conté un objecte que implementa la interfície `Document` del DOM i això permet obtenir l'element arrel a partir de la propietat `documentElement`.

Seguidament es recorre la llista dels nodes descendents de l'arrel i, una vegada s'ha comprovat que el tipus és `ELEMENT_NODE` (de manera que s'eviten els nodes de text), s'invoca la funció `processarElement` passant com a argument l'element. Aquesta funció extreu l'atribut `cp` i el contingut textual de l'element, crea un nou element de tipus `li` (element d'una llista d'HTML) i li assigna com a contingut una cadena composta pel nom de la província i el prefix del codi postal.

Els elements es processen un per un fins que no en queda cap, llavors s'afegeix la `llista` al cos del document.

Fixeu-vos que el tractament que es fa de la resposta XML és igual al que es pot fer a HTML, de manera que es possible extreure la informació dels atributs amb el mètode `getAttribute` i del contingut textual amb `textContent`.

S'ha de tenir en compte que en casos més complexos pot ser necessari utilitzar funcions recursives per recórrer completament el document. En aquest exemple s'ha considerat que l'arbre només constava de dos nivells i, per tant, només s'ha utilitzat un bucle per iterar sobre els elements. Per aquesta raó, en molts casos necessitareu implementar el vostre propi intèrpret (*parser* en anglès) que recorri tots els nodes iterativament i els processi un per un segons els requeriments de l'aplicació.

1.3.5 Processament de respostes JSON

Per processar una resposta en format JSON (JavaScript Object Notation) s'utilitza la propietat `responseText` de l'objecte `XMLHttpRequest` i seguidament s'analitza sintàcticament per convertir-la en un objecte de JavaScript, que es pot utilitzar com qualsevol altre.

Per comprovar-ho, creeu un fitxer de text anomenat `provincies.json` amb el contingut següent:

```
{
  "provincies": [
    {"nom": "Barcelona", "cp": "08"},
    {"nom": "Girona", "cp": "17"},
    {"nom": "Lleida", "cp": "25"},
    {"nom": "Tarragona", "cp": "43"}
  ]
}
```

L'estructura del format JSON és molt similar a la declaració literal d'objectes en JavaScript, ja que aquest és el seu origen. S'ha de tenir en compte que tant els noms de les propietats com els valors textuals han d'anar entre cometes dobles (no es poden utilitzar cometes simples). Encara que els nombres no és necessari que estiguin entre cometes, en aquest exemple s'han de tractar com cadenes de text per respectar el valor `08`.

El contingut de la resposta s'ha de trobar entre claus, que contenen una o més propietats amb un nom que es pot fer servir com a clau i un valor. Aquest valor pot ser qualsevol dels valors primitius de JavaScript com són els nombres i les cadenes de text, un *array* o un altre objecte literal. En el cas dels *arrays* el contingut ha d'anar entre claudàtors, `[` i `]`, mentre que els objectes han d'anar entre claus `{` i `}`.

Cal destacar que l'element arrel en una resposta en format JSON tant pot ser un objecte com un *array*.

El codi que s'ha de fer servir per realitzar la petició és pràcticament idèntic al de les peticions HTML: primer es crea l'objecte `XMLHttpRequest`, seguidament s'envia i finalment es processa la resposta:

```
<!DOCTYPE html>
<head>
  <meta charset="utf-8">
  <script>
     if (window.XMLHttpRequest) {
       httpRequest = new XMLHttpRequest();
     } else if (window.ActiveXObject) {
       httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
     } else {
       console.error("Error: Aquest navegador no admet AJAX.");
     }

     httpRequest.onload = processarResposta;
     httpRequest.open('GET', 'provincies.json', true)
     httpRequest.overrideMimeType('text/plain');
     httpRequest.send(null);

     function processarResposta() {
       var resposta = JSON.parse(httpRequest.responseText);
       var llista = document.createElement('ul');

       for (var i = 0; i < resposta.provincies.length; i++) {
         var item = processarElement(resposta.provincies[i]);
         llista.appendChild(item);
       }

       document.body.appendChild(llista);
     }

     function processarElement(provincia) {
```

```
31        var item = document.createElement('li');
32        item.textContent = provincia.nom + ' (CP ' + provincia.cp + ')';
33        return item;
34      }
35    </script>
36  </head>
37
38  <body>
39  </body>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/qaLZyk?editors=1012.

Com podeu veure s'ha forçat el tipus multimèdia invocant el mètode `overrideMimeType` amb el valor `text/plain` per assegurar que es pot interpretar com a text sense que el navegador llanci cap excepció.

Un cop s'ha obtingut la resposta, s'analitza sintàcticament fent servir el mètode `parse` de l'objecte `JSON`, de manera que el resultat és un objecte JavaScript funcional que es pot utilitzar com un diccionari de dades.

Es recorren totes les províncies una per una i es retorna un element de tipus `li` amb el contingut textual corresponent a la província especificada com a argument.

Fixeu-vos que la implementació en aquest cas és més simple, ja que la conversió de la resposta en un objecte de JavaScript és directa i, per tant, es pot accedir a les seves propietats fent servir la notació de punt (`provincia.nom`) o la notació de claudàtors (`provincia['nom']`).

Com en el cas del processament de respostes XML, s'ha de tenir en compte que en determinats casos (quan no coneixem la resposta a priori, per exemple) pot ser necessari implementar una funció recursiva per recórrer tots els elements de l'arbre per processar-los.

La utilització d'un format o un altre dependrà del format en el qual el servidor retorna la resposta, tot i que en alguns casos s'ofereix accedir a la mateixa informació tant en XML com en JSON. En aquests casos pot ser més fàcil treballar amb el format JSON, perquè es pot manipular la resposta directament sense necessitat d'utilitzar les interfícies del DOM.

1.4 Serveis web

En moltes ocasions és necessari implementar un sistema informàtic que faciliti l'intercanvi d'informació entre diferents plataformes, així és possible accedir a les mateixes dades des d'una pàgina web o una aplicació mòbil canviant només la interfície en la qual es mostren.

La implementació d'aquests serveis web depèn del servidor i és el servidor el que determina la forma d'accedir a aquestes dades. Les dues arquitectures més utilitzades són SOAP (Simple Object Access Protocol) i REST (Representational

State Transfer); per conseqüent, cal conèixer-les per poder desenvolupar aplicacions web que connectin amb aquests serveis, mitjançant AJAX o bé CORS (Cross-Origin Resource Sharing).

Cal destacar que la implementació del servidor és independent de la implementació dels clients, ja que el servidor permet connectar amb l'aplicació sense que importi la plataforma o el sistema operatiu que s'hagin fet servir. Només cal que els clients implementin correctament els mecanismes de connexió.

Un avantatge dels serveis web és que quan es treballa sobre el protocol HTTP, per accedir-hi no cal fer cap modificació als clients, al contrari d'altres tipus d'aplicacions que requereixen utilitzar ports específics i que poden ser bloquejats pels tallafocs, tant de l'equip com de l'encaminador a través del qual es connecten a internet.

1.4.1 SOAP

SOAP

Podeu trobar més informació sobre el protocol SOAP ('protocol d'accés a objecte simple') a l'enllaç següent: www.goo.gl/Zufz8w.

WSDL

Podeu trobar més informació sobre WSDL ('llenguatge de descripció de serveis web') a l'enllaç següent: www.goo.gl/SU5j3d.

SOAP és un protocol utilitzat per intercanviar informació en una xarxa d'ordinadors. Utilitza el format XML tant per negociar la connexió com per generar la resposta i pot funcionar sobre diferents protocols de xarxa com HTTP, SMTP, TCP o UDP.

Tot i que no és imprescindible, es pot utilitzar un fitxer WSDL (Web Services Description Language) per definir la descripció del *web service*. Aquest fitxer, en format XML, conté la informació necessària perquè els clients puguin conèixer i utilitzar les operacions que ofereix el servei web.

Entre els desavantatges d'aquesta arquitectura es troba la llargària dels missatges, la complexitat d'implementar els serveis, tant a la banda del client com a la del servidor, i la lentitud en analitzar sintàcticament les respostes XML.

1.4.2 REST

REST

Podeu trobar més informació sobre l'arquitectura REST ('transferència d'estat de representació') a l'enllaç següent: www.goo.gl/bUr2TO.

Mentre que SOAP és un protocol i disposa d'una interfície estàndard, REST és un estil d'arquitectura i no s'aplica cap estàndard a si mateix, tot i que sí que en fa servir alguns com HTTP, URI, JSON i XML.

Els serveis web REST són més simples d'implementar que els serveis SOAP i força més lleugers. Per accedir-hi només cal conèixer l'URI (*uniform resource identifier*, 'identificador uniforme de recursos') del punt d'accés per l'acció (*endpoint*) i el verb o mètode que s'ha d'utilitzar.

Els verbs es corresponen amb els mètodes del protocol HTTP són els següents:

- **GET**: per obtenir dades.

- **PUT**: per reemplaçar dades.
- **PATCH**: per actualizar dades.
- **POST**: per afegir una nova dada.
- **DELETE**: per eliminar una dada.

D'aquesta manera, a partir d'un mateix URL, segons el verb emprat el resultat serà diferent. Per exemple, a partir de l'URL (fictícia) `http://api.example.com/recursos/`:

- **GET**: obtindria una llista de tots els recursos.
- **PUT**: reemplaçaria un recurs existent amb les dades enviades com a paràmetres (des d'un formulari, per exemple).
- **PATCH**: actualitzaria les dades d'un recurs existent.
- **POST**: crearia un nou recurs a partir de les dades enviades com a paràmetres.
- **DELETE**: eliminaria la col·lecció `recursos`.

En aquesta arquitectura les accions realitzades utilitzant el mètode `GET` no han de provocar cap modificació a les dades i, en conseqüència, es tracta d'un mètode segur. Per contra, els altres tres sempre les modifiquen.

Un avantatge important sobre SOAP és que no requereix utilitzar XML, així que per l'intercanvi de dades es pot utilitzar JSON o altres formats de representació que poden resultar més pràctics i fàcils de comprovar.

A més a més, el protocol REST és molt lleuger i quan es fan servir altres formats de transferència com JSON, poden minimitzar el consum d'amplada de banda tant per al servidor com per al client.

A diferència de SOAP, els serveis web REST funcionen només sobre el protocol HTTP, ja que depenen de la utilització dels seus mètodes per determinar quina acció s'ha de portar a terme sobre les dades.

S'ha de tenir en compte que tant el format com l'estructura de les dades que s'intercanvien són lliures; així doncs, caldrà comprovar la documentació del servei o analitzar sintàcticament la resposta per poder interpretar-les correctament. Per exemple, és possible que la resposta arribi en format JSON amb dos camps, un amb el codi de la resposta i un altre amb les dades:

```
{
  "codi": 200
  "dades": {
    provincies: ["Barcelona", "Girona", "Lleida", "Tarragona"]
  }
}
```

En altres casos podria arribar només la informació sense cap codi, ja que es pot extreure del codi d'estat de la capçalera (*propietat state*), llavors la resposta podria ser similar a aquesta:

```
{
  provincies: ["Barcelona", "Girona", "Lleida", "Tarragona"]
}
```

Cal destacar que amb aquesta arquitectura és molt fàcil comprovar-ne el funcionament, sobretot amb el mètode `GET`, ja que només s'ha d'introduir al navegador l'URI del punt d'accés amb els paràmetres (amb aquesta forma: `www.example.com/recursos?asignatura=m06&unitat=2`) i es carregarà el resultat al mateix navegador.

1.5 JSONP i CORS

A causa de les limitacions imposades per la política del mateix domini no es poden fer peticions AJAX a dominis diferents del que es troba l'aplicació. Per superar aquesta limitació hi ha dues tècniques alternatives: JSONP (JSON amb *padding*) i CORS (Cross-Origin Resource Sharing).

Malauradament, totes dues requereixen que el servidor estigui preparat per acceptar aquestes peticions de dominis externs i, per consegüent, no totes les fonts de dades són accessibles des de aplicacions web encara que ofereixin una API (interfície de programació d'aplicacions) que exposi aquestes dades.

S'ha de tenir en compte que la política del mateix domini només afecta els navegadors i, per tant, els serveis web que no admeten ni JSONP ni CORS continuen sent accessibles per aplicacions d'escriptori o mòbils.

1.5.1 JSON amb padding (JSONP)

La tècnica del JSON amb *padding* consisteix a carregar la informació com si es tractés d'un script que inclou una invocació a una funció amb totes les dades com a paràmetre (el *padding* es refereix a això).

Aquesta tècnica funciona perquè la càrrega de scripts no està subjecta a la política del mateix origen i permet carregar scripts que invoquin automàticament altres funcions.

API de Flickr

Podeu trobar tota la informació per desenvolupadors de l'API de Flickr a l'enllaç següent: www.goo.gl/AmwqQl.

Per exemple, la resposta obtinguda en carregar l'URL bit.ly/2ocrAKQ incrusta un script a la pàgina amb una invocació a la funció `processar`, que és el valor del paràmetre `jsoncallback`, i que utilitza el servei web de Flickr (www.flickr.com) per generar la resposta JSONP i li passa totes les dades en format JSON com a argument:

```
processar({
```

```
    "title": "Recent Uploads tagged kitten",
    "link": "http://www.flickr.com/photos/tags/kitten/",
    "description": "",
    "modified": "2016-10-22T18:01:41Z",
    "generator": "http://www.flickr.com/",
    "items": [
     {
      "title": "Cats images Beautiful Eyes via http://ift.tt/29KELz0",
      "link": "http://www.flickr.com/photos/dozhub/29860948003/",
      "media": {"m":"http://farm9.staticflickr.com/8677/29860948003
          _31626a7d77_m.jpg"},
      "date_taken": "2016-10-22T11:01:41-08:00",
      "description": " <p><a href=\"http://www.flickr.com/people/dozhub/\">
          dozhub<\/a> posted a photo:<\/p> <p><a href=\"http://www.flickr.com/
          photos/dozhub/29860948003/\" title=\"Cats images Beautiful Eyes via
          http://ift.tt/29KELz0\"><img src=\"http://farm9.staticflickr.com
          /8677/29860948003_31626a7d77_m.jpg\" width=\"240\" height=\"160\"
          alt=\"Cats images Beautiful Eyes via http://ift.tt/29KELz0\" /><\/a
          ><\/p> <p>Cats images Beautiful Eyes -<br /> <br /> Cats images
          Beautiful Eyes - Cats, kittens and kittys, cute and adorable! Aww! (
          via <a href=\"http://ift.tt/29KELz0\" rel=\"nofollow\">ift.tt/29
          KELz0<\/a>)<br /> <br /> - via <a href=\"http://ift.tt/29KELz0\" rel
          =\"nofollow\">ift.tt/29KELz0<\/a>. Cats, kittens and kittys, cute
          and adorable! Aww!<\/p>",
      "published": "2016-10-22T18:01:41Z",
      "author": "nobody@flickr.com (dozhub)",
      "author_id": "143919671@N07",
      "tags": "cat kitty kitten cute funny aww adorable cats"
     }]
  });
```

Així doncs, una vegada es carrega l'script s'executa la funció `processar`, que prèviament haurem definit i rep com a paràmetre totes les dades per processar-les.

Com es pot apreciar, només es pot utilitzar aquesta tècnica si el servidor està preparat per servir les dades en aquest format.

Vegeu a continuació un exemple que extreu la informació de l'agregador (*feed*) de Flickr per mostrar fotos de gatets:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
</body>
<script>
  function processar (dades) {
    var imatges = dades.items;

    for (var i=0; i<imatges.length; i++) {
      var img = document.createElement('img');
      img.setAttribute('src', imatges[i].media.m);
      img.setAttribute('title', imatges[i].title);
      img.setAttribute('alt', imatges[i].title);
      document.body.appendChild(img);
    }
  };
</script>

<script src='http://api.flickr.com/services/feeds/photos_public.gne?
    jsoncallback=processar&tags=kitten&format=json'></script>
</script>
```

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

Fixeu-vos que a l'exemple a CodePen s'ha fet servir només la pestanya de codi HTML. Això s'ha fet així per assegurar que primer es defineix la funció `processar` i seguidament es realitza la petició JSONP. En cas contrari, la resposta de la petició pot arribar abans de definir-se la funció i produir-se un error, ja que la funció a executar encara no es troba definida.

La implementació de la funció `processar` és molt simple: com que la informació ha arribat en format JSON, s'ha convertit automàticament en un objecte JavaScript sense haver d'analitzar-la sintàcticament, així que només cal iterar sobre les dades i crear un element de tipus `img` per a cada imatge establint els atributs `src`, `title` i `imatge` associats.

1.5.2 Cross-Origin Resource Sharing (CORS)

CORS és un mecanisme que permet als navegadors moderns realitzar peticions AJAX a dominis diferents del que es troba l'aplicació web, utilitzant l'objecte `XMLHttpRequest` però assegurant la seguretat de la transmissió de dades en col·laboració amb el servidor.

CORS

Podeu trobar més informació sobre CORS ('control d'accés HTTP') a l'enllaç següent: www.goo.gl/CHQXLa.

En enviar una petició, hi ha un intercanvi de dades entre el client i el servidor per comprovar si la transmissió és acceptable o no. Aquestes comprovacions poden incloure diferents factors com l'origen de la petició o l'acreditació de l'usuari.

Si el servidor envia a la capçalera el paràmetre `Access-Control-Allow-Origin` amb un valor que correspongui al del domini de l'aplicació o `*`, la transmissió es pot dur a terme i la resposta contindrà les dades demanades. En altres casos és necessari comprovar l'acreditació i es necessitarà establir la propietat `withCredentials=true` de l'objecte que envia la petició per habilitar l'enviament de les galetes al servidor, on es comprovarà la validesa de les credencials.

> El mecanisme CORS és **transparent** al codi del client, ja que és gestionat per la implementació del servidor i dels navegadors.

A continuació, podeu veure un exemple en què s'envia una petició a una de les fonts de dades obertes de l'Ajuntament de Barcelona i es recupera la llista de festivals de l'any 2015 en format JSON:

```
<!DOCTYPE html>

<head>
  <meta charset="utf-8">
  <script>
    if (window.XMLHttpRequest) {
      httpRequest = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
    } else {
```

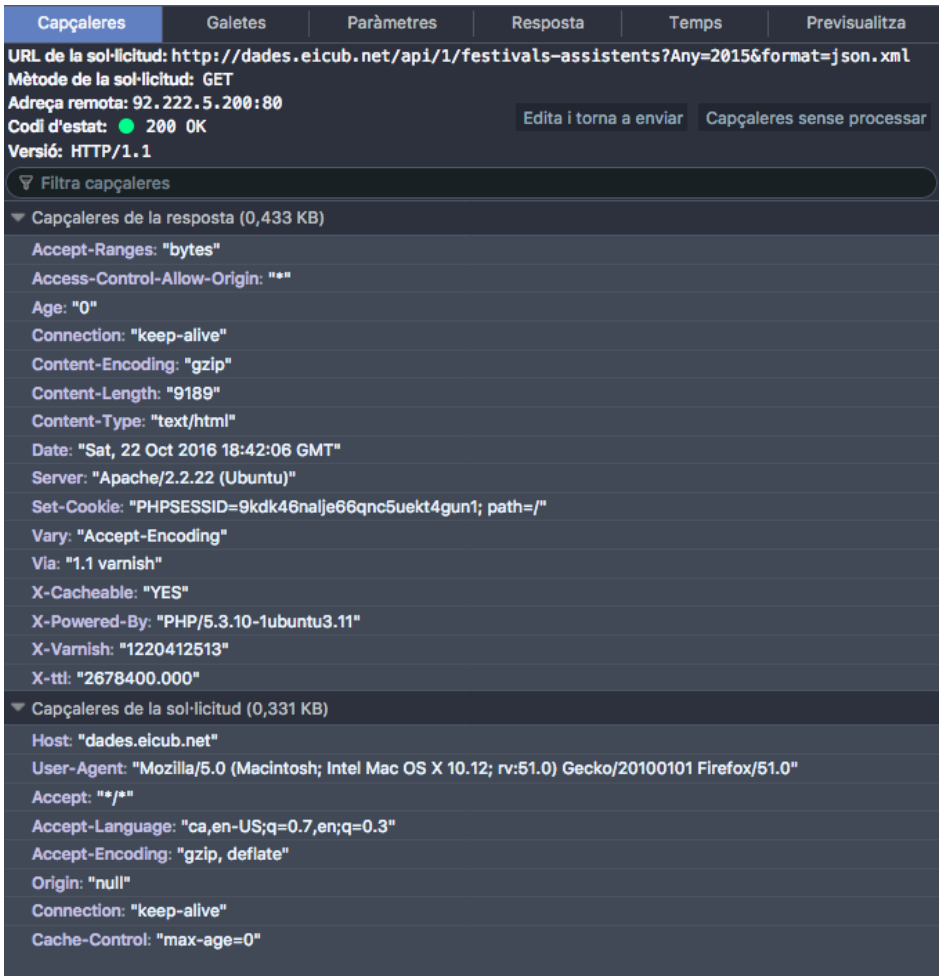
```
      console.error("Error: Aquest navegador no admet AJAX.");
    }

    httpRequest.onload = processarResposta;
    httpRequest.open('GET', 'http://dades.eicub.net/api/1/festivals_assistents?
        Any=2015&format=json.xml', true)
    httpRequest.overrideMimeType('text/plain');
    httpRequest.send(null);

    function processarResposta() {
      var resposta = JSON.parse(httpRequest.responseText);
      var llista = document.createElement('ul');
      console.log(resposta);

      for (var i = 0; i < resposta.length; i++) {

          var item = processarDada(resposta[i]);
          llista.appendChild(item);
      }

      document.body.appendChild(llista);
    }

    function processarDada(dada) {
      var item = document.createElement('li');
      var enllac = document.createElement('a');
      enllac.textContent = dada.NomDelFestival;
      if (dada.Web) {
        enllac.setAttribute('href', dada.Web);
      }
      enllac.setAttribute('title', dada.Organitzador);
      item.appendChild(enllac);

      return item;
    }
  </script>
</head>

<body>
  <h1>Festivals 2015</h1>
</body>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ALNjyG?editors=1010.

Si examineu la capçalera de la petició amb les eines de desenvolupador del navegador, veureu que inclou el paràmetre `Access-Control-Allow-Origin` amb valor `*`. És a dir, admet connexions des de qualsevol domini, com es pot apreciar a la figura 1.1.

CORS inclou altres mecanismes més complexos com són les restriccions de les peticions segons els mètodes empleats, l'ús de credencials o la modificació de les capçaleres, però tots treballen en conjunció amb el servidor i, consegüentment, si no es té accés al servidor o aquest no ha estat preparat per treballar amb aquests mecanismes no es podrà accedir a les dades des d'altres dominis.

FIGURA 1.1. Capçaleres d'una petició CORS



1.6 Accés a dades obertes

Les dades obertes són conjunts de dades posades a disposició del públic i que es poden fer servir sense restriccions. Habitualment aquestes dades poden ser consultades al mateix lloc web, descarregades o consumides per altres serveis.

Aquesta darrera opció permet el desenvolupament d'aplicacions que utilitzin dades de tercers; per exemple, per mostrar els pròxims esdeveniments a una ciutat o l'estat del trànsit.

Un conjunt de dades, perquè es considerin obertes, han de poder ser reutilitzables, adaptables i distribuïbles pels usuaris.

S'ha de tenir en compte que per poder accedir a aquestes dades els serveis web que les exposen han d'admetre la utilització de JSONP o de CORS, ja que en cas contrari seran bloquejats per la política del mateix origen dels navegadors.

En aquest aspecte, l'Ajuntament de Barcelona ofereix una gran selecció de dades obertes a les quals es pot accedir des de l'adreça següent: www.opendata.bcn.cat/opendata. Malauradament, la majoria de dades d'aquest

catàleg no admeten ni JSONP ni CORS i, per consegüentment, no es poden accedir des d'aplicacions web.

Per comprovar si una font de dades admet JSONP s'ha de consultar la documentació de l'API del servei web que es vol utilitzar. Aquesta documentació pot ser disponible o no; per exemple, al catàleg de dades obertes de l'Ajuntament de Barcelona no es pot trobar en la majoria de les fonts. Per contra, les dades ofertes per l'Observatori de dades culturals de Barcelona (que forma part del mateix Ajuntament) sí que ofereix documentació per accedir a les seves dades, que es poden trobar a l'enllaç següent: www.barcelonadadescultura.bcn.cat/dades-obertes.

Comprovar si un lloc admet la transmissió de dades mitjançant el mecanisme CORS bàsic (sense cap requisit) és molt més simple: només cal consultar a les eines de desenvolupador la capçalera de la resposta en accedir directament a l'adreça. Si la capçalera inclou el paràmetre `Access-Control-Allow-Origin=''*''` (l'asterisc vol dir que accepta qualsevol origen), voldrà dir que s'hi pot accedir utilitzant AJAX directament, perquè el mecanisme CORS està habilitat al servidor.

Per exemple, si accediu a www.dades.eicub.net/api/1/museusexposicions-visitants (que forma part del catàleg de dades obertes de l'Observatori de dades culturals de Barcelona) i a les eines de desenvolupador seleccioneu la pestanya per inspeccionar el tràfic a la xarxa (pestanya *Network* tant a Google Chrome com a Mozilla Firefox), en fer clic sobre la petició, podeu comprovar que, efectivament, a la capçalera es troba el paràmetre `Access-Control-Allow-Origin` amb valor `*` i, per tant, es poden realitzar peticions des de diferents dominis.

Podeu comprovar-ho amb l'exemple següent, que realitza una petició al catàleg de dades, processa aquestes dades afegint-les a un diccionari de dades propi i les incorpora a una llista desplegable que, en seleccionar l'identificador, mostra les dades enllaçades:

```
<head>
  <meta charset="utf-8">

  <style>
    h1 {
      text-align:center;
    }

    fieldset {
      width: 300px;
      margin: 0 auto;
    }
    label {
      display: inline-block;
      width: 150px;
      font-weight: bold;
    }
    span {
      display: inline-block;
      width: 150px;
    }
    ul {
      list-style-type: none;
      padding: 0;
    }
  </style>
```

```
27
28  </head>
29
30  <body>
31    <h1>Activitats de difusió de les fàbriques de creació</h1>
32    <fieldset>
33      <select id="identificador">
34      <option>Selecciona un identificador</option></select>
35      <ul>
36        <li><label>Id:</label><span id="id"></span></li>
37        <li><label>Any:</label><span id="any"></span></li>
38        <li><label>Equipament:</label><span id="equipament"></span></li>
39        <li><label>Districte:</label><span id="districte"></span></li>
40        <li><label>Àmbit:</label><span id="ambit"></span></li>
41        <li><label>Assistents:</label><span id="asistents"></span></li>
42        <li><label>Notes:</label><span id="notes"></span></li>
43        <li><label>Tipus d'equipament:</label><span id="tipusEquipament"></span><
              /li>
44
45        <li><label>Titularitat:</label><span id="titularitat"></span></li>
46        <li><label>Activitats de difusió dins dels centres:</label><span id="
              activitats"></span></li>
47      </ul>
48    </fieldset>
49
50  <script>
51    if (window.XMLHttpRequest) {
52      httpRequest = new XMLHttpRequest();
53    } else if (window.ActiveXObject) {
54      httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
55    } else {
56      console.error("Error: Aquest navegador no admet AJAX.");
57    }
58
59    var dades = {}; // Diccionari on es guardaran les dades
60
61    httpRequest.onload = processarResposta;
62    httpRequest.open('GET', 'http://dades.eicub.net/api/1/fabriquescreacio-
          difusio', true)
63    httpRequest.overrideMimeType('text/plain');
64    httpRequest.send(null);
65
66    function processarResposta() {
67      var resposta = JSON.parse(httpRequest.responseText);
68      var llistaDesplegable = document.getElementById('identificador');
69
70      for (var i = 0; i < resposta.length; i++) {
71        var item = processarDada(resposta[i]);
72        dades[resposta[i].Id] = resposta[i];
73        llistaDesplegable.appendChild(item);
74      }
75
76      llistaDesplegable.onchange = actualitzarDadesMostrades;
77
78    }
79
80    function processarDada(dada) {
81      var item = document.createElement('option');
82      item.setAttribute('value', dada.Id);
83      item.textContent = dada.Id;
84      return item;
85    }
86
87    function actualitzarDadesMostrades(event) {
88        var llistaDesplegable = document.getElementById('identificador');
89      var dada = dades[llistaDesplegable.value]
90
91      console.log(dada);
92      actualitzarDadaMostrada('id', dada.Id);
93      actualitzarDadaMostrada('any', dada.Any);
```

```
94      actualitzarDadaMostrada('equipament', dada.Equipament);
95      actualitzarDadaMostrada('districte', dada.Districte);
96      actualitzarDadaMostrada('ambit', dada.Ambit);
97      actualitzarDadaMostrada('asistents', dada.Assistents);
98      actualitzarDadaMostrada('notes', dada.Notes || 'No hi ha cap nota');
99      actualitzarDadaMostrada('tipusEquipament', dada.TipusEquipament);
100     actualitzarDadaMostrada('titularitat', dada.Titularitat);
101     actualitzarDadaMostrada('activitats', dada.
            Activitats_de_difusio_dins_dels_centres);
102
103   }
104
105   function actualitzarDadaMostrada(nom, valor) {
106     document.getElementById(nom).textContent = valor;
107   }
108
109 </script>
110 </body>
```

Podeu veure aquest exemple en l'enllaç següent: www.http://codepen.io/ioc-daw-m06/pen/jrdVNd?editors=1111.

Com es pot apreciar, com que es tracta d'un servei web que admet CORS, només cal fer una petició AJAX convencional i processar les dades.

Cal tenir en compte que una mateixa pàgina pot fer peticions a múltiples fonts de dades. Per exemple, seria possible fer consultes a un catàleg de dades d'esdeveniments culturals on s'incloguessin les coordenades de geolocalització i utilitzar aquestes dades per afegir marques a un mapa de Google Maps (que fa servir les seves pròpies crides AJAX).

1.7 API Web Sockets

En determinades situacions, la transmissió de dades mitjançant AJAX no és viable, per exemple, en el cas de jocs multijugador en temps real o les aplicacions de missatgeria.

Cal recordar que les peticions al servidor fent servir el protocol HTTP no conserven l'estat, sinó que cada petició es considera nova i s'han d'utilitzar altres mecanismes com les galetes i l'enviament de paràmetres perquè el servidor reconegui l'usuari.

La solució és fer servir altres protocols com TCP i UDP, que permeten mantenir la comunicació oberta entre el client i el servidor per enviar i rebre missatges.

Web Sockets

Podeu trobar més informació sobre l'API Web Sockets i l'objecte `WebSocket` als enllaços següents, respectivament: www.goo.gl/eJgu0q i www.goo.gl/OF78BK

Per implementar aquesta funcionalitat a JavaScript s'ha d'utilitzar l'API Web Sockets, introduïda a HTML5, que permet realitzar connexions a servidors utilitzant internament el protocol TCP. Els protocols utilitzats per connectar amb el servidor són `ws` o `wss` per a sòcols segurs.

Mentre que fent servir peticions AJAX s'han de fer peticions periòdiques al servidor per comprovar si hi ha noves dades, això no cal fer-ho quan s'utilitzen Web Sockets, perquè una vegada hi ha dades noves al servidor, s'envien al client, i així s'estalvien recursos i amplada de banda tant al client com al servidor.

S'ha de tenir en compte que aquesta tecnologia és experimental, així doncs, abans de començar la implementació de qualsevol aplicació s'ha de consultar documentació actualitzada, ja que és possible que canviïn els noms de propietats, mètodes o els seus paràmetres.

Per descomptat, per poder realitzar una connexió amb un servidor, ha d'acceptar connexions TCP. Cada llenguatge de programació ofereix les seves alternatives; així doncs, es poden trobar servidors per Web Sockets en pràcticament qualsevol llenguatge.

Tot i així, s'ha de tenir en compte que no tots els llenguatges tenen el mateix nivell de compatibilitat en aquest àmbit, de manera que si voleu implementar un servidor de Web Sockets en PHP, el nombre de fonts que caldrà consultar serà molt menor que si ho feu amb Node.js (JavaScript a la banda del servidor), que té un grau de compatibilitat molt superior i API especialitzades com Socket.IO (www.socket.io) per simplificar aquestes tasques.

A l'hora de desenvolupar aplicacions segures utilitzant Web Sockets s'ha de tenir en compte que els navegadors no admeten determinats comportaments. Per exemple, si la pàgina on s'executa l'aplicació utilitza el protocol HTTP, l'aplicació pot fer servir el protocol `ws` o `wss`. Per contra, si la pàgina fa servir el protocol HTTPS (HTTP segur), s'haurà d'utilitzar forçosament el protocol `wss` o la connexió serà bloquejada pel navegador.

Per altra banda, tampoc no està permesa la utilització de certificats (SSL o TLS) autosignats, sinó que han d'utilitzar-se certificats aprovats per una autoritat de certificació i lligats a un nom de domini.

1.8 Depuració de crides AJAX

Per desenvolupar aplicacions és imprescindible saber les eines de depuració que teniu al vostre abast. Afortunadament, les eines més importants de les que es disposa a l'hora de depurar una aplicació web estan incorporades en els navegadors més populars, com Google Chrome i Mozilla Firefox.

Concretament, aquests dos navegadors fan servir interfícies d'usuari molt similars, com es pot apreciar a la figura 1.2, corresponent a les eines de xarxa de Google Chrome, i la figura 1.3, que mostra les de Mozilla Firefox.

Com podeu veure en els dos casos, la pestanya que conté la informació sobre les peticions s'anomena `xarxa` (*network* en anglès). En tots dos casos es disposa d'una barra superior on es poden filtrar les peticions que cal visualitzar: XHR, JS, CSS, Img, WS, etc. Això facilita localitzar ràpidament el recurs que volem consultar, ja que una pàgina pot realitzar desenes de peticions.

FIGURA 1.2. Eines de desenvolupador a Google Chrome

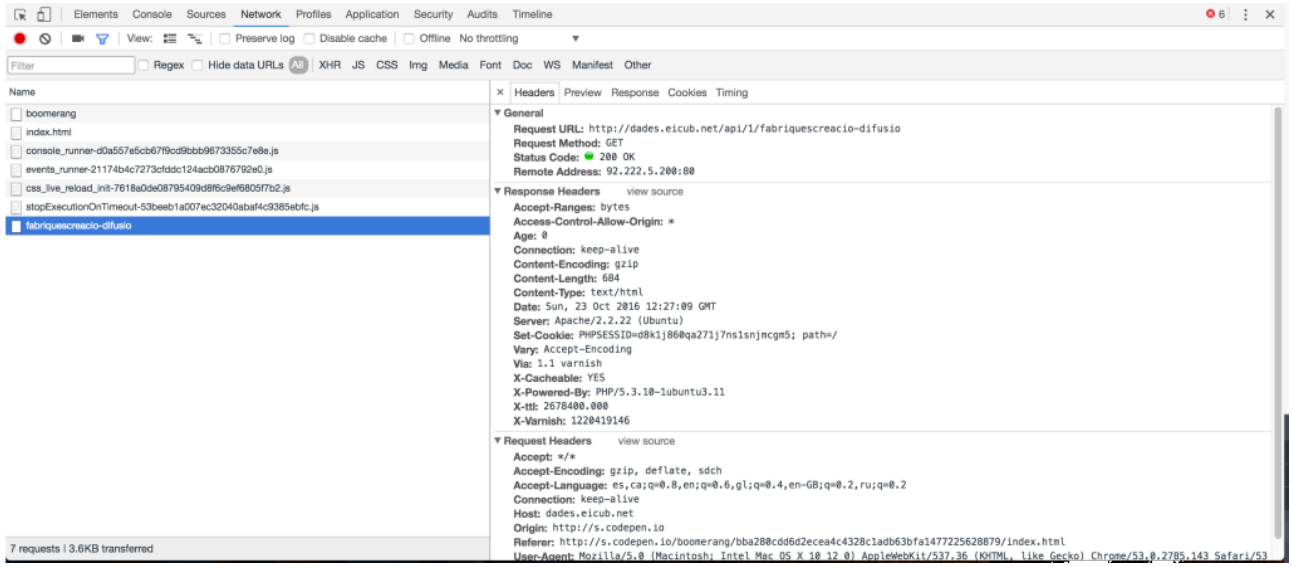
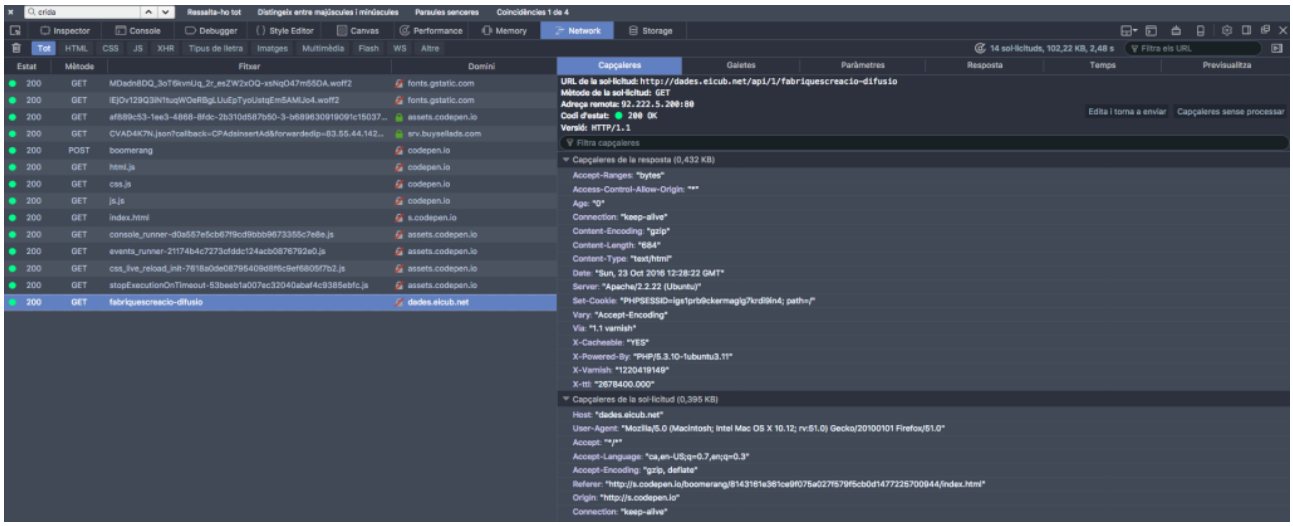


FIGURA 1.3. Eines de desenvolupador a Mozilla Firefox



Una vegada es clica sobre una de les peticions, la pestanya es divideix en dues zones: a l'esquerra es continua veient la informació general de les peticions, i a la dreta la informació concreta, dividida al seu torn en cinc pestanyes (capçaleres, previsualització, resposta, galetes i temps).

Per comprovar les dades de la petició i la resposta podeu fer servir el codi següent, que envia una petició a un servei web que admet peticions de diferents dominis:

```
1  <html>
2  <head>
3    <meta charset="utf-8">
4    <script>
5    httpRequest = new XMLHttpRequest();
6
7    httpRequest.onload = processarResposta;
8    httpRequest.open('GET', 'http://dades.eicub.net/api/1/fabriquescreacio-
         difusio', true)
9    httpRequest.overrideMimeType('text/plain');
10   httpRequest.send(null);
11
12   function processarResposta() {
13     console.log("Dades carregades correctament");
```

```
14    }
15    </script>
16  </head>
```

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

A la pestanya de la capçalera es pot veure tant la informació d'enviament com la resposta del servidor. Entre la informació que es mostra en aquesta pestanya cal destacar l'URL al qual s'ha enviat la petició, el mètode, el codi d'estatus (200 per a una petició realitzada amb èxit, 404 per a una petició no trobada, 500 per a un error intern del servidor, etc.) i en el cas de peticions a altres dominis, la presència del paràmetre `Access-Control-Allow-Origin` si el servidor admet CORS.

Cal destacar que Mozilla Firefox inclou a aquesta pestanya el botó *Edita i torna a enviar*, que permet editar manualment les dades de la capçalera de la petició i reenviar-la.

Podeu veure a la figura 1.4 la mostra d'una capçalera en Google Chrome i a la figura 1.5 la mateixa capçalera en Mozilla Firefox.

FIGURA 1.4. Capçaleres d'una petició a Google Chrome

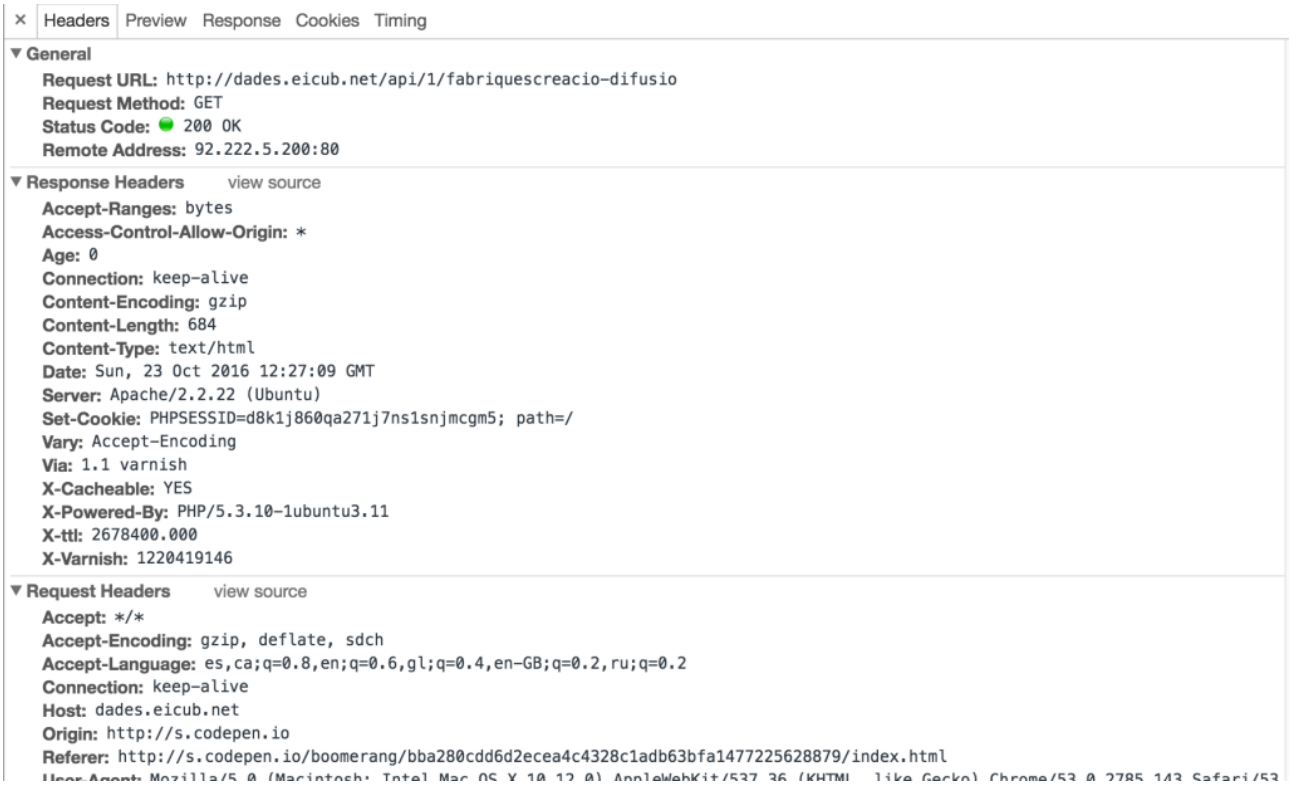
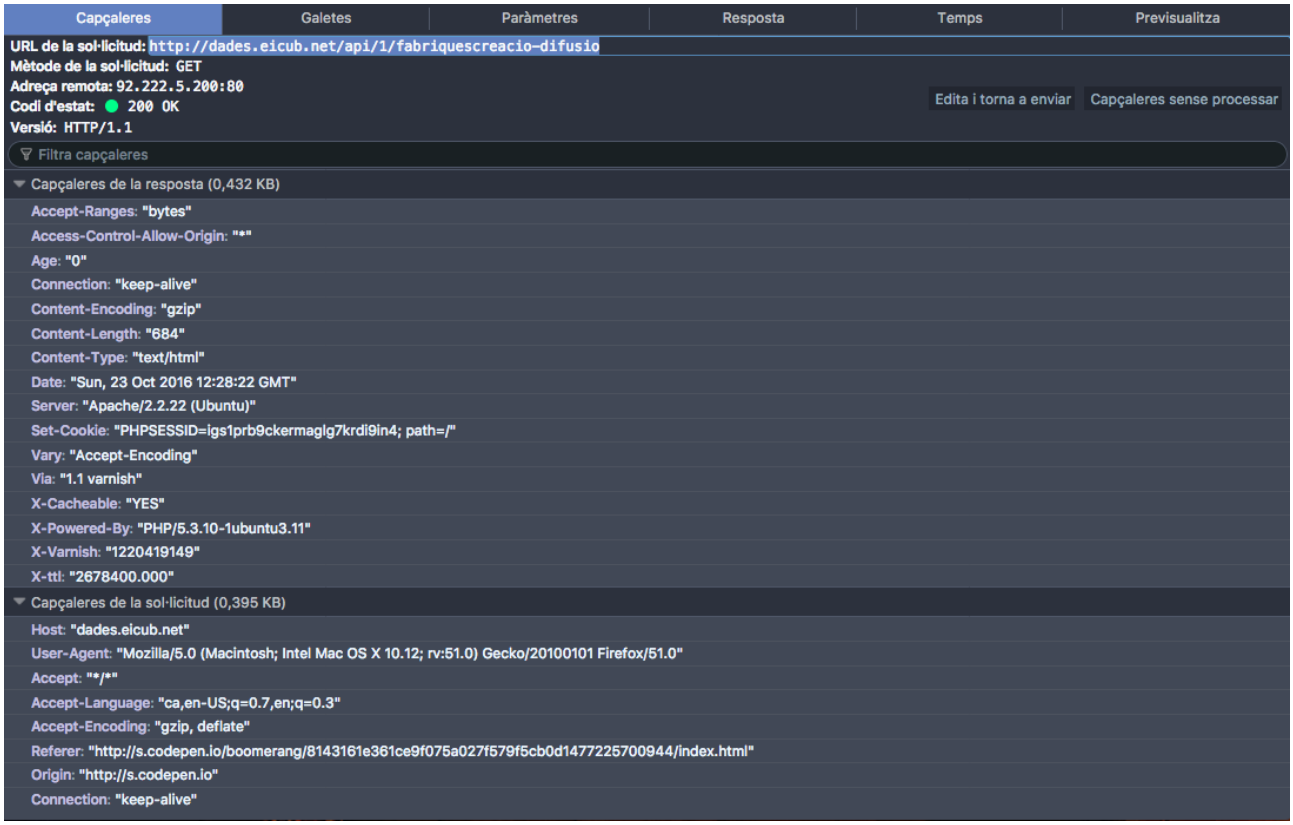


FIGURA 1.5. Capçaleres d'una petició a Mozilla Firefox



La pestanya de previsualització és la que mostra més diferències entre els dos navegadors: mentre que a Google Chrome es mostra la informació correctament estructurada com a objectes de JavaScript (com es pot apreciar a la figura 1.6), a Mozilla Firefox es mostra com a text pla (vegeu la figura 1.7). Cal destacar que en el cas del codi HTML el més habitual és que es mostri una previsualització de la pàgina.

FIGURA 1.6. Previsualització d'una petició a Google Chrome

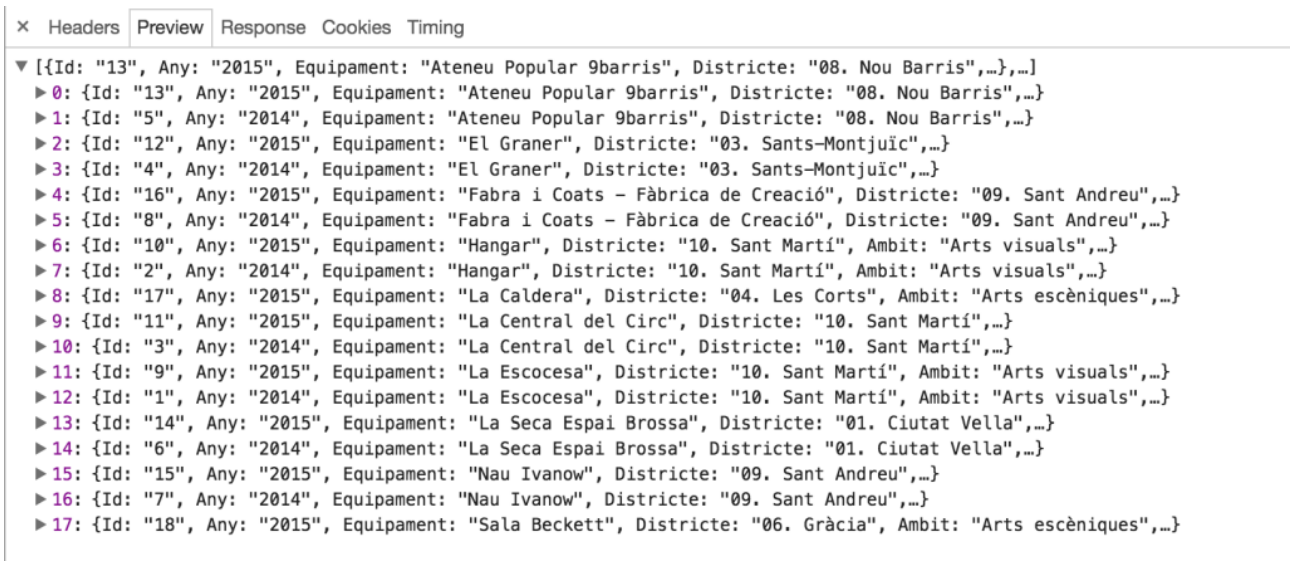
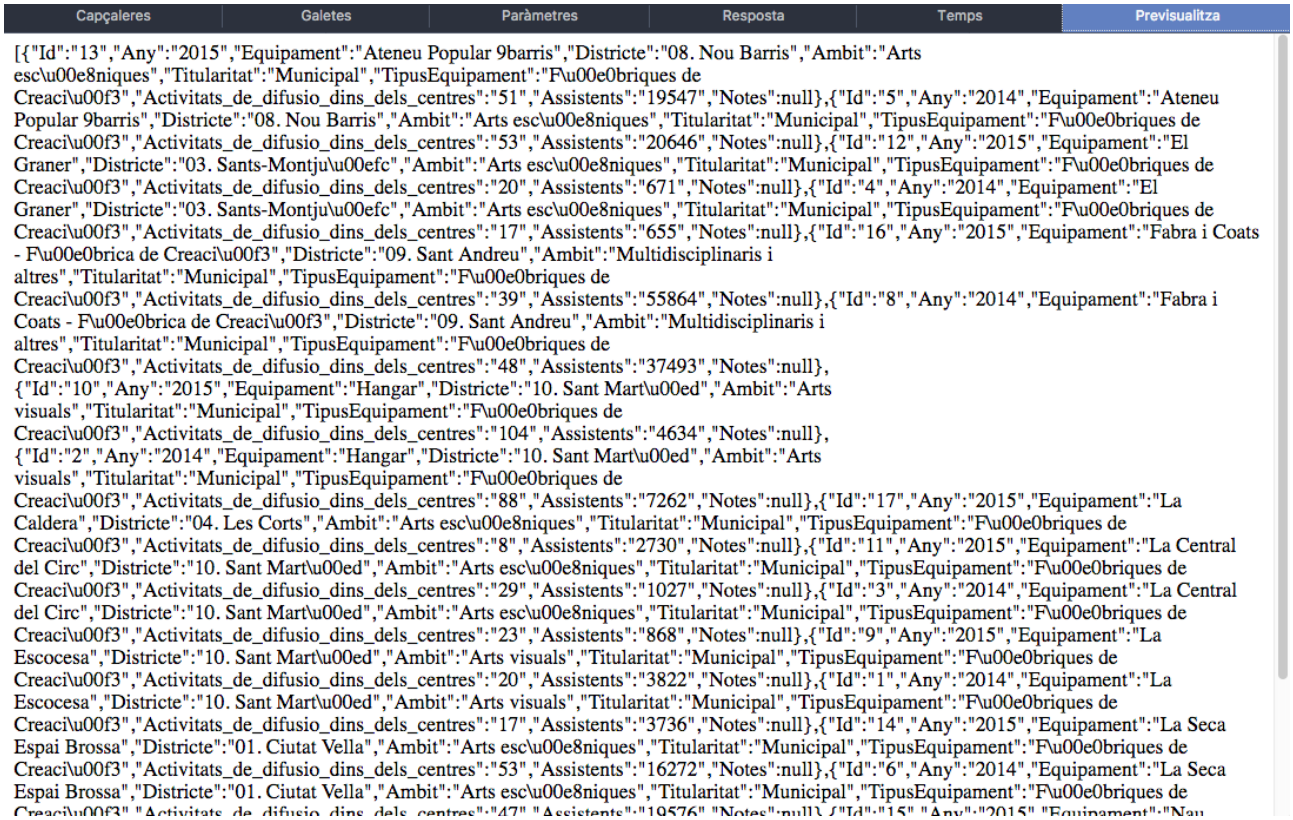


FIGURA 1.7. Previsualització d'una petició a Mozilla Firefox



Per altra banda, la pestanya "Response" ('resposta') de Google Chrome (vegeu la figura 1.8) mostra la resposta en text pla, mentre que la pestanya "Resposta" de Mozilla Firefox (vegeu la figura 1.9) mostra les dades estructurades per facilitar-ne la inspecció. És interessant recordar-ho perquè a l'hora de depurar una aplicació que utilitzi AJAX o Web Sockets, segons quin navegador utilitzeu us interessarà més consultar la informació a la pestanya de resposta o a la de previsualització.

FIGURA 1.8. Resposta d'una petició a Google Chrome

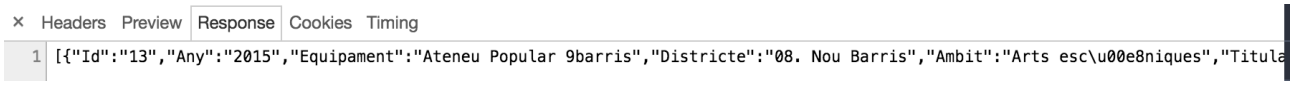
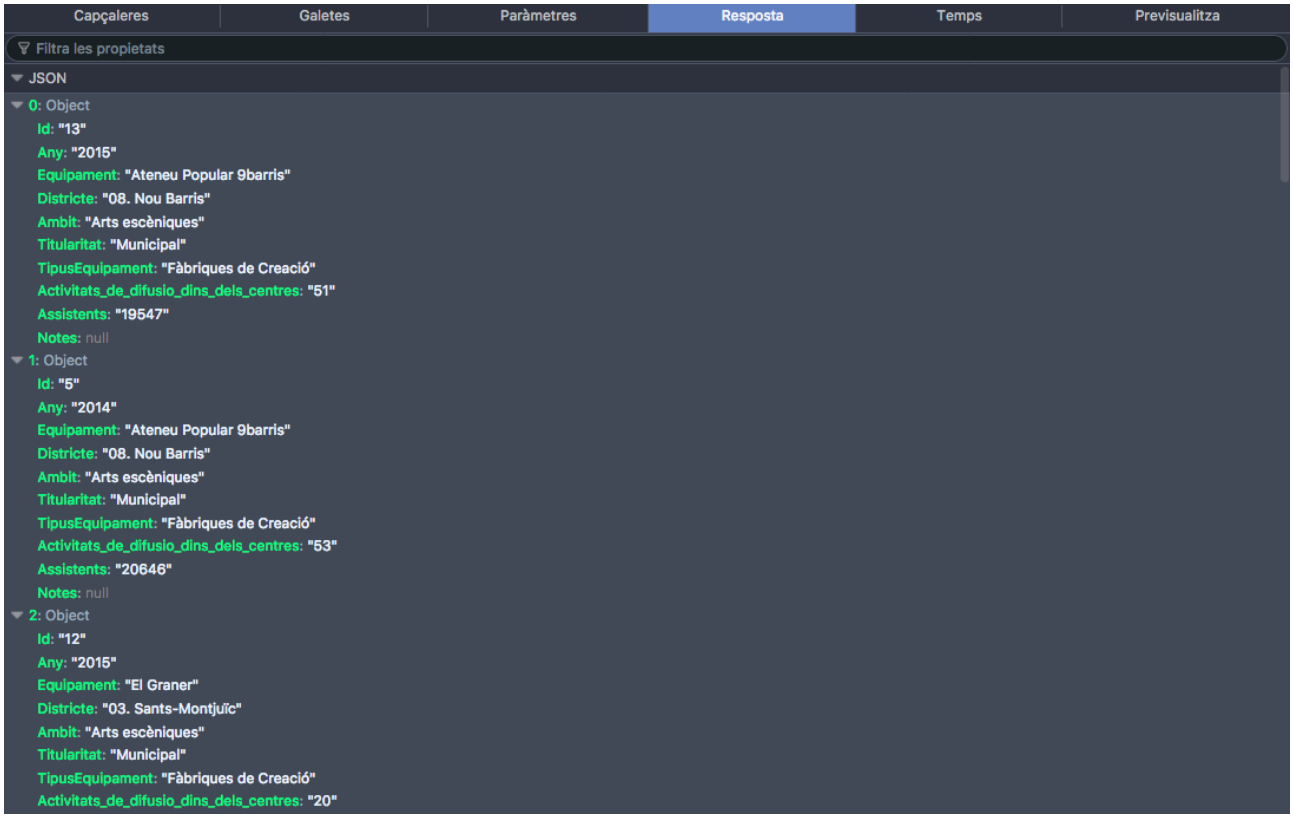


FIGURA 1.9. Resposta d'una petició a Mozilla Firefox



A la pestanya amb la informació de les galetes es poden veure les galetes de la sol·licitud d'una manera pràcticament idèntica a tots dos navegadors (vegeu la figura 1.10 i la figura 1.11). Aquesta pestanya només es mostra quan la petició inclou les galetes.

Cal destacar que es pot trobar més informació sobre les galetes a la pestanya *Application* a Google Chrome i a la pestanya *Storage* a Mozilla Firefox.

FIGURA 1.10. Galetes d'una petició a Google Chrome

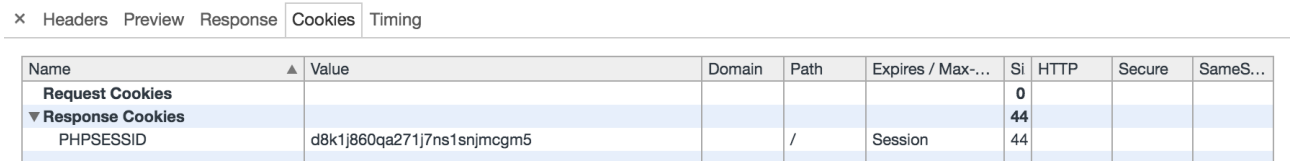
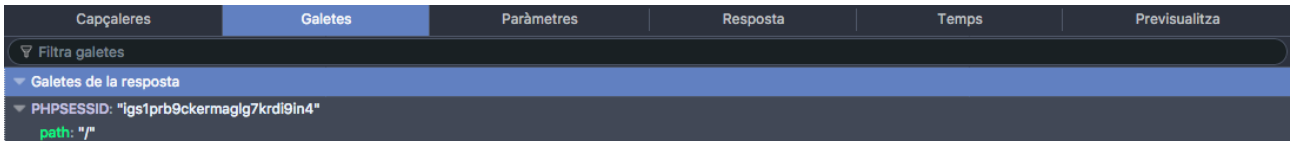


FIGURA 1.11. Galetes d'una petició a Mozilla Firefox



Finalment, es troba la pestanya *Temps*, que mostra com s'ha invertit el temps que ha trigat a realitzar-se la petició i en retornar la resposta. Aquesta pestanya mostra la mateixa informació tant a Google Chrome (vegeu la figura 1.12) com a Mozilla Firefox (vegeu la figura 1.13).

Figura 1.12. Temps invertit d'una petició a Google Chrome

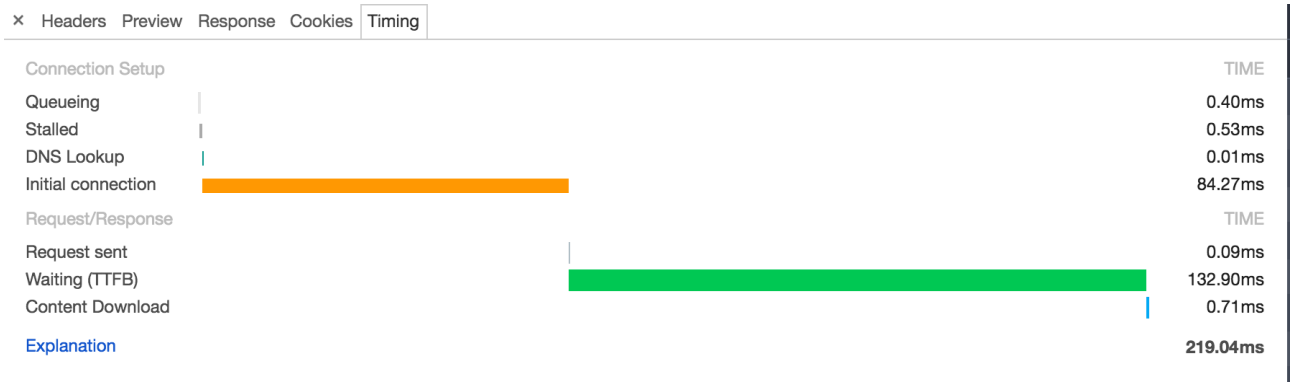
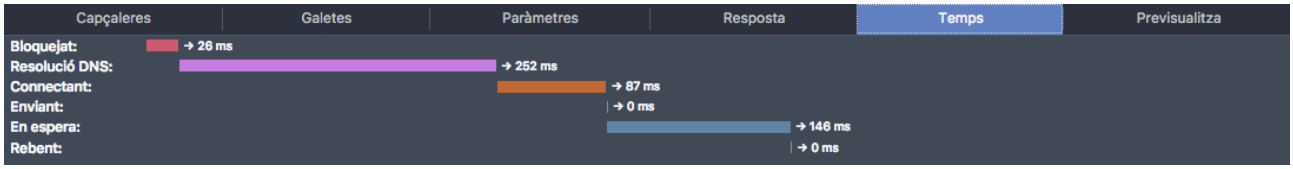


Figura 1.13. Temps invertit d'una petició a Mozilla Firefox



Cal destacar que Mozilla Firefox inclou una pestanya extra en la qual es mostren els paràmetres enviats, mentre que a Google Chrome es poden trobar a la mateixa pestanya de la capçalera.

Conèixer aquestes eines i les opcions que ofereixen, tant per l'enviament com per la resposta, és imprescindible per poder depurar correctament aplicacions que facin servir AJAX, ja que ens permet comprovar, entre altres coses, si la petició ha retornat un codi d'èxit o d'error, si el servidor admet CORS o no, o si els paràmetres enviats i la resposta són els esperats.

2. Programació de la comunicació asíncrona amb jQuery

A banda de la gestió d'*events* i la manipulació del DOM, una de les funcionalitats més importants de jQuery és la gestió de peticions AJAX (Javascript asíncron i XML).

Una de les diferències és que no cal fer cap comprovació prèvia per discriminar entre les versions antigues del navegador Internet Explorer i els navegadors actuals, la biblioteca s'encarrega de fer-ho, i la gestió de la resposta se simplifica mitjançant mètodes propis en lloc d'haver d'afegir-hi la detecció d'*events* (que els navegadors antics potser no admeten).

Cal destacar que per poder utilitzar aquestes funcionalitats s'ha de carregar la biblioteca jQuery. Per exemple, a partir del codi local (suposem que es troba dins del directori /js/biblioteques i que el nom del fitxer és `jquery-3.1.0.js`):

```
1 <script src="js/biblioteques/jquery-3.1.0.js"></script>
```

En cas d'utilitzar una xarxa de lliurament de continguts (en anglès, *content delivery network*, CDN) el codi seria el següent:

```
1 <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
```

S'ha de tenir en compte que tant si s'utilitza la biblioteca com si es fa servir una implementació pròpia, els navegadors apliquen la política del mateix origen i, per consegüent, només es poden realitzar peticions al mateix domini en què es troba l'aplicació, motiu pel qual la majoria dels exemples que es troben a CodePen no s'executaran correctament.

Aquesta limitació no s'aplica si el servei web al qual s'envia la petició admet CORS (peticions amb diferent origen) i, en cas de requerir-les, les capçaleres i les dades d'autenticació són correctes, o si admet JSONP (JSON amb *padding*) i es realitza la petició utilitzant aquesta tècnica.

A diferència de les implementacions pròpies, amb jQuery s'utilitza el mètode `ajax` de l'objecte `jQuery` tant per a les peticions ordinàries com per a les peticions mitjançant JSONP.

2.1 Creació i enviament de peticions amb jQuery

Com que la biblioteca jQuery s'encarrega de gestionar les diferències entre versions no cal comprovar si el navegador admet o no la utilització de l'objecte `XMLHttpRequest` o si es tracta d'una versió antiga d'Internet Explorer.

Només cal afegir la càrrega de la biblioteca (en el següent exemple mitjançant un CDN):

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
  </head>

  <body>
  </body>

</html>
```

Per evitar la complexitat de crear un servidor per servir les peticions als exemples s'utilitzaran fitxers amb format XML, que s'utilitzaran per llegir la informació. El primer d'aquests fitxers ha de dur el nom `provincies.xml` i el contingut següent:

```
<provincies>
  <provincia>Barcelona</provincia>
  <provincia>Girona</provincia>
  <provincia>Lleida</provincia>
  <provincia>Tarragona</provincia>
</provincies>
```

Una vegada tingueu l'entorn preparat (el fitxer HTML i el fitxer XML) només heu d'afegir les següents línies al final de l'element `body` per enviar la petició i mostrar el resultat per la consola:

```
<script>
  $.ajax('provincies.xml', {
    success: processarResposta,
    error: processarError
  });

  function processarResposta(dades, statusText, jqXHR) {
    console.log(dades, statusText);
  }

  function processarError(jqXHR, statusText, error) {
    console.log(error, statusText);
  }
</script>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/zKVzwG?editors=1010.

Configuració de la funció 'ajax' de jQuery

Podeu consultar una llista completa d'opcions de configuració per la funció `ajax` a l'enllaç següent: www.goo.gl/fDsgT5.

Com es pot apreciar, el codi és molt simple. S'invoca el mètode `ajax` de l'objecte `jQuery` passant com a paràmetres l'URL on s'enviarà la petició i un objecte de JavaScript amb una propietat anomenada `success`, a la qual s'ha assignat la funció `processarResposta`, i una altra propietat anomenada `error`, a la qual s'ha assignat la funció `processarError`.

Aquest objecte conté les opcions que s'utilitzen per configurar la petició. En aquest cas se n'han vist dues:

- **success**: s'executa si la petició s'ha realitzat correctament i rep la resposta, el text d'estat i l'objecte `jqXHR` creat per la petició.

- **error**: s'executa quan s'ha produït un error. En aquest cas els paràmetres són l'objecte `jqXHR`, el text d'estat i l'error produït.

Objectes diferits

Podeu trobar més informació sobre els objectes diferits a l'enllaç següent: www.goo.gl/5oxmwa.

L'objecte `jqXHR` (una versió ampliada de l'objecte XMLHttpRequest dels navegadors) és retornat per la invocació del mètode `ajax` i permet realitzar o encadenar altres accions a la invocació mitjançant una interfície anomenada *objecte diferit* (*deferred object*). Per aquesta raó, es passa també com a argument les funcions `success` i `error`, de manera que es pot obtenir més informació o realitzar altres operacions sobre la petició.

Tot i que la petició es realitza automàticament en invocar el mètode `ajax`, cal recordar que (per defecte) és asíncrona i, per tant, la resposta no s'obté immediatament; l'aplicació continuarà executant-se sense bloquejar-se fins que es rebi la resposta, llavors s'executarà el mètode pertinent (assignat a `success` o `error`).

Alternativament, es pot invocar el mètode `ajax` utilitzant, només, l'objecte amb les opcions com a paràmetre i assignant l'URL a la propietat `url`:

```
$.ajax({
  url: 'provincies.xml',
  success: processarResposta,
  error: processarError
});
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/Bprxvj?editors=1010.

2.1.1 Paràmetres

Habitualment en fer una petició AJAX s'han de passar un o més paràmetres al servidor. Aquests paràmetres són visibles a l'URL en el cas d'utilitzar el mètode `GET` per a l'enviament (el mètode per defecte).

Per exemple, la següent petició s'efectuaria sobre l'URL `http://example.com` i els paràmetres enviats són `nom` i `cognom`. Com que no s'especifica el mètode, s'utilitzarà `GET`. Fixeu-vos que en aquest cas s'afegiran a l'URL: `http://example.com/?nom=Maria&Cognom=Campmany`.

```
$.ajax({
   url: 'http://example.com',
   data: { nom:"Maria", cognom:"Campmany"}
 });
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/jMgOpJ?editors=0010.

Podeu trobar més informació sobre les eines de desenvolupador a l'apartat "Comunicació asíncrona amb JavaScript" d'aquesta mateixa unitat.

Obriu les eines de desenvolupadors i a la pestanya xarxa (*network*) trobareu una línia corresponent a aquesta petició. Quan hi feu clic, tindreu accés a tota la informació sobre la petició. Pareu especial atenció a la capçalera (*header*).

El símbol `?` indica que el que hi ha a continuació són els paràmetres de la consulta, que estarà formada per parells de clau i valor separats pel signe igual. Aquests parells, al seu torn, són separats d'altres paràmetres pel símbol `&`.

Així doncs, analitzant l'URL, es pot dividir en diferents parts:

- **http**: correspon al protocol HTTP.
- **example.com**: correspon al domini al qual s'envia la petició.
- **/**: correspon a la ruta, que en aquest cas és l'arrel del domini.
- **nom=Maria**: primer paràmetre, corresponent a la clau `nom` amb valor `Maria`.
- **cognom=Campmany**: segon paràmetre, corresponent a la clau `cognom` amb valor `Campmany`.

Una alternativa a fer servir un objecte per passar els paràmetres és fer servir una cadena de text amb els parells clau-valor separats pel signe igual i aquests separats d'altres parells pel símbol `&`:

```
1 $.ajax({
2    url: 'http://example.com',
3    data: 'nom=Maria&cognom=Campmany'
4  });
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/MJVGMr?editors=0010.

Quan es fa servir un altre mètode, com `POST` o `PUT`, aquesta informació només és visible a la capçalera. Cal destacar que això no és cap garantia de privadesa ni seguretat, ja que aquesta capçalera pot ser inspeccionada per qualsevol node de la xarxa per on passi la informació.

```
1 $.ajax({
2    url: 'http://example.com',
3    method: 'post',
4    data: { nom:"Maria", cognom:"Campmany"}
5  });
```

Podeu veure aquest exemple en l'enllaç següent: www.http://codepen.io/ioc-daw-m06/pen/PGMovd?editors=0010.

En aquest cas, en analitzar la petició amb les eines de desenvolupador, podeu comprovar que l'URL no s'ha modificat i aquests paràmetres s'han afegit a la capçalera com a dades de formulari.

Cal destacar que la biblioteca jQuery permet assignar la propietat `data` de les opcions com a cadena de text per a tots els mètodes i no només per al mètode `GET`, ja que jQuery fa les conversions necessàries. Així doncs, és possible fer servir com a `data` la cadena de text `nom=Maria&cognom=Campmany` i en enviar la petició aquests paràmetres s'afegeixen com a dades a la capçalera:

```
$.ajax({
   url: 'http://example.com',
   method: 'post',
   data: 'nom=Maria&cognom=Campmany'
 });
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ORKPJa?editors=0010.

2.1.2 Tipus de dades: dataType

Si no s'especifica una altra cosa, s'espera que el format de la resposta sigui XML vàlid. És a dir, si proveu de fer una petició demanant un fitxer de text pla amb el contingut que trobareu a continuació, es produirà un error:

```
Barcelona
Girona
Lleida
Tarragona
```

Com que no s'ha especificat el tipus de dades, es produeix un error: el format del fitxer no és vàlid. Així doncs, cal especificar el tipus de dades que s'espera rebre del servidor assignant la propietat `dataType` a les opcions:

```
$.ajax({
  url: 'provincies.text',
  dataType: 'text',
  success: processarResposta,
  error: processarError
});
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/MjMvoJ?editors=1010.

Els tipus de dades admesos per les peticions AJAX de jQuery són:

- **xml** (tipus per defecte): retorna un document XML que pot ser processat per jQuery.
- **html**: retorna el codi HTML com a text pla.
- **script**: avalua la resposta com a JavaScript.
- **json**: avalua la resposta com a JSON i retorna un objecte de JavaScript.
- **jsonp**: utilitzat per obtenir les dades d'un altre domini fent servir la tècnica JSONP, retorna un objecte de JavaScript amb les dades.
- **text**: resposta com a text pla.

> En cas de carregar fitxers, com en aquests exemples, o si el servidor no especifica el tipus multimèdia de la resposta, s'ha de sobreescriure mitjançant el mètode `overrideMimeType` de l'objecte `jqXHR`.

Per assegurar que el tipus de contingut multimèdia és correcte es pot utilitzar l'opció `beforeSend`, que permet especificar una funció que serà invocada abans de realitzar la petició. Aquesta funció rep com a paràmetre l'objecte `jqXHR` i sobre aquest es pot invocar el mètode `overrideMimeType` per forçar el tipus de la resposta:

```
$.ajax({
  url: 'provincies.txt',
  dataType: 'text',
  beforeSend: function (jqXHR) {
    jqXHR.overrideMimeType('text/plain');
  },
  success: processarResposta,
  error: processarError
});
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ozremJ?editors=1010.

Cal destacar que el més habitual és que el servidor web retorni el tipus multimèdia correcte i no s'hagi de sobreescriure.

Com es pot apreciar, un altre dels tipus admesos és HTML.

Suposem que s'ha d'afegir dinàmicament a una pàgina el codi HTML següent (podria ser la resposta enviada pel servidor, per exemple):

```
<html>
  <h1>Provincies</h1>
  <ul>
    <li>Barcelona</li>
    <li>Girona</li>
    <li>Lleida</li>
    <li>Tarragona</li>
  </ul>
</html>
```

Per fer-ho, només caldria especificar el tipus com `html` i afegir-lo utilitzant el mètode `append` de l'objecte jQuery:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
  <script>
    $.ajax({
      url: 'provincies.html',
      dataType: 'html',
      success: processarResposta,
      error: processarError
```

```
    });

    function processarResposta(dades, statusText, jqXHR) {
        $dades = $(dades);
        $('body').append($dades);

    }

    function processarError(jqXHR, statusText, error) {
      console.log(error, statusText);
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/XjLaVe?editors=1010.

Errors habituals quan no s'especifica el tipus de dades

Habitualment es produeixen errors si no s'especifica el tipus de dades mitjançant la propietat `dataType`, ja que a l'hora de processar la resposta la biblioteca jQuery intentarà interpretar-lo com XML i si no ho aconsegueix llença una excepció.

En aquest cas, com que el codi HTML utilitzat té el mateix format que XML (que és el valor per defecte de dataType) es considera correcte i no cal sobreescriure el tipus multimèdia.

En canvi, si la resposta fos en format JSON, caldria canviar el `dataType` per JSON i sobreescriure el tipus multimèdia. A diferència de les implementacions manuals, no cal analitzar la resposta perquè ja és retornada com un objecte de JavaScript. Proveu de crear un fitxer anomenat `provincies.json` amb el contingut següent:

```
{
  "provincies": [
    {"nom": "Barcelona", "cp": "08"},
    {"nom": "Girona", "cp": "17"},
    {"nom": "Lleida", "cp": "25"},
    {"nom": "Tarragona", "cp": "43"}
  ]
}
```

I substituir el codi JavaScript pel següent:

```
$.ajax({
  url: 'provincies.json',
  dataType: 'json',
  beforeSend: function (jqXHR) {
    jqXHR.overrideMimeType('application/json');
  },
  success: processarResposta,
  error: processarError
});

function processarResposta(dades, statusText, jqXHR) {
  console.log(dades, statusText);
}

function processarError(jqXHR, statusText, error) {
  console.log(error, statusText);
}
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/wzLqOM?editors=1010.

Com es pot apreciar, en aquest cas la resposta és un objecte JavaScript que es pot processar de la forma habitual.

2.1.3 Mètodes d'enviament

De la mateixa manera que a les implementacions pròpies d'AJAX, la biblioteca jQuery permet especificar el mètode amb el qual s'envien les peticions. Només cal afegir, a l'objecte amb les opcions, la propietat `method` i assignar-li el mètode pertinent: `GET` (per defecte), `POST`, `PUT` o `DELETE`. Per exemple, es pot especificar el mètode `GET` per llegir un fitxer JSON:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
  <script>
    $.ajax({
      method: 'get',
      url: 'provincies.json',
      dataType: 'json',
      beforeSend: function (jqXHR) {
        jqXHR.overrideMimeType('application/json');
      },
      success: processarResposta,
      error: processarError
    });

    function processarResposta(dades, statusText, jqXHR) {
      console.log(dades, statusText);
    }

    function processarError(jqXHR, statusText, error) {
      console.log(error, statusText);
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en el següent enllaç: www.codepen.io/ioc-daw-m06/pen/wzLQOR?editors=1010.

Cal recordar que aquests mètodes es corresponen amb els del protocol HTTP 1.1, i als serveis web REST són utilitzats sobre un mateix URL per realitzar diferents accions:

- **GET**: consultar les dades.
- **POST**: afegir noves dades.

- **PATCH**: actualitzar dades ja existents.
- **PUT**: reemplaçar dades ja existents.
- **DELETE**: eliminar dades.

'get' i 'post'

Podeu trobar totes les opcions dels mètodes `$.get` i `$.post` als següents enllaços: www.goo.gl/DVzxg8 i www.goo.gl/zRbp5A; respectivament.

A més a més, jQuery ofereix dues dreceres per simplificar l'enviament de peticions a través dels mètodes `GET` i `POST`. Tots dos mètodes s'invoquen a partir de l'objecte jQuery:

- `$.get`: envia una petició AJAX fent servir el mètode `GET`.
- `$.post`: envia una petició AJAX fent servir el mètode `POST`.

Aquests mètodes ofereixen un nombre més limitat d'opcions, però suficient en la major part de les situacions:

- **url**: una cadena amb l'URL al qual s'envia la petició.
- **data**: un objecte o cadena de text que s'enviaran com a paràmetres de la petició.
- **sucess**: funció de tipus *callback* que serà cridada si la petició es realitza amb èxit.
- **dataType**: el tipus de dades esperat pel servidor (`xml`, `json`, `script`, `text` i `html`).

Aquestes opcions es poden passar com a paràmetres individuals:

```
$.get(url, data, success, dataType);
```

O com un objecte en el qual totes les propietats, excepte `url`, són opcionals:

```
$.get({
  url: 'provincies.html',
  data: {nom: 'Josep', cognoms: 'Torres Blanc'},
  success: processarResposta},
  dataType: 'html'
});
```

Així doncs, per fer una petició amb el mètode `GET` i obtenir una resposta en HTML el codi complet seria el següent:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
  <script>
    $.get({
      url: 'provincies.html',
```

```
      data: {nom: 'Josep', cognoms: 'Torres Blanc'},
      success: processarResposta,
      dataType: 'html'
    });

    function processarResposta(dades, statusText, jqXHR) {
      $dades = $(dades);
      $('body').append($dades);
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en el següent enllaç: www.codepen.io/ioc-daw-m06/pen/RGzvgP?editors=1010.

Com es pot apreciar, en aquest cas no s'utilitzen els paràmetres enviats, ja que no hi ha cap servidor servint la resposta, però a les eines de desenvolupador podeu comprovar que la capçalera ha inclòs aquests paràmetres. En tractar-se del mètode `get` s'ha convertit l'URL de la petició en quelcom similar al següent: `/provincies.html?nom=Josep&cognoms=Torres%20Blanc`.

És a dir, jQuery ha convertit l'objecte passat com a argument en paràmetres vàlids per enviar com a petició, fent servir el nom de les propietats com a clau i el valor assignat com a valor del paràmetre.

S'ha de tenir en compte que, tal com passa amb el mètode `ajax`, si l'URL de la petició no correspon a un servidor que envia una resposta amb un tipus especificat, l'aplicació esperarà que es tracti d'una resposta XML i donarà error. En el cas de les dreceres no existeix l'opció `beforeSend` i, per consegüent, no es pot sobreescriure el tipus multimèdia de la resposta.

2.1.4 Interpretar la resposta

Segons el tipus de data (`dataType`) de la resposta i si aquesta és correcta, la resposta serà retornada de diferents maneres:

- **XML**: un objecte `XMLDocument`.
- **HTML**, **TEXT**: una cadena de text amb el contingut.
- **JSON**: un objecte de JavaScript.
- **Script**: un bloc de codi JavaScript que serà avaluat (executat).

`XMLDocument` és un tipus d'objecte que és herència de la interfície del DOM `document` i no afegeixen cap altra propietat o mètode nous. Això permet utilitzar jQuery per manipular aquest document:

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

```
3
4  <head>
5    <meta charset="utf-8">
6    <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
7  </head>
8
9  <body>
10   <script>
11     $.get({
12       url: 'provincies.xml',
13       success: processarResposta,
14     });
15
16     function processarResposta(dadesXML, statusText, jqXHR) {
17         var $provincies = $(dadesXML).find('provincia');
18         var $llista = $('<ul>');
19
20
21         $provincies.each(function() {
22             var nomProvincia = $(this).html();
23             $provincia = $('<li>');
24             $provincia.html(nomProvincia);
25             $llista.append($provincia);
26         })
27
28         $('body').append($llista);
29     }
30   </script>
31 </body>
32
33 </html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/MjMLRP?editors=1010.

Una vegada s'ha obtingut la resposta, es converteix en un objecte jQuery i se seleccionen tots els elements amb el nom `provincia`. A continuació, es crea una nova llista sense numeració (`<ul>`) i es procedeix a recórrer-los fent servir el mètode `each`, que invoca la funció passada com a argument amb l'element corresponent com a context. És a dir, l'element és referenciat per `this` (en aquest cas, cadascun dels elements de `provincies`).

De cadascun d'aquests elements s'obté el nom de la província a partir del contingut intern de l'element mitjançant el mètode `html`. Seguidament es crea un nou element de tipus element de llista (`<li>`) i s'assigna com a contingut d'aquest element el nom de la província que, finalment, és afegit a la llista que, al seu torn, és afegida al cos de la pàgina quan es finalitza el recorregut.

Com que XML es el tipus per defecte, no cal especificar-lo quan es treballa amb aquest format.

En el cas de `HTML` es pot utilitzar la funció `jQuery` per convertir aquest text en un objecte jQuery que, al seu torn, converteix la cadena de text en elements del DOM: `$branca = $(dades)`. Aquesta nova branca es pot afegir a altres elements: `$('body').append($branca)`.

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <meta charset="utf-8">
```

```
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
  <script>
    $.ajax({
      url: 'provincies.html',
      beforeSend: function (jqXHR) {
        jqXHR.overrideMimeType('text/html');
      },
      success: processarResposta,
      dataType: 'html'
    });

    function processarResposta(dadesHTML, statusText, jqXHR) {
        var $branca = $(dadesHTML);
        $('body').append($branca);
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en lenllaç següent: www.codepen.io/ioc-daw-m06/pen/EgBMXj?editors=1010.

Com es pot apreciar, s'ha fet servir el mètode `ajax` per incloure l'opció `beforeSend` i canviar el tipus multimèdia a `text/html`. D'aquesta manera s'eviten possibles errors de format, ja que el sistema de fitxers no especifica el tipus de contingut.

Per analitzar un fitxer de tipus textual s'han de fer servir les funcions de tractament de cadenes, com per exemple `split`, o expressions regulars. Per exemple, si el contingut de la resposta fos la següent:

```
Barcelona,Girona,Lleida,Tarragona
```

Es podria processar la resposta dividint la cadena per la coma, mitjançant el mètode `split`:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
  <script>
    $.ajax({
      url: 'provincies.txt',
      beforeSend: function(jqXHR) {
        jqXHR.overrideMimeType('text/plain');
      },
      success: processarResposta,
      dataType: 'text'
    });

    function processarResposta(dadesText, statusText, jqXHR) {
      var provincies = dadesText.split(',');
      var $llista = $('<ul>');

      for (var i = 0; i < provincies.length; i++) {
```

```
        var $item = $('<li>');
        $item.html(provincies[i]);
        $llista.append($item);
      }

      $('body').append($llista);
    }
  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/ZpdPME?editors=1010.

En el cas que la resposta sigui un objecte JavaScript és molt fàcil treballar-hi, ja que es pot accedir directament a tota la informació fent servir la notació de claudàtors (`provincia['nom']`) o de punt (`provincia.nom`).

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
  <script>
    $.ajax({
      url: 'provincies.json',
      beforeSend: function(jqXHR) {
        jqXHR.overrideMimeType('application/json');
      },
      success: processarResposta,
      dataType: 'json'
    });

    function processarResposta(dadesJSON, statusText, jqXHR) {
      var provincies = dadesJSON.provincies;
      var $llista = $('<ul>');

      for (var i = 0; i < provincies.length; i++) {
        var $item = $('<li>');
        $item.html(provincies[i].nom + " (" + provincies[i].cp + ")");
        $llista.append($item);
      }

      $('body').append($llista);
    }

  </script>
</body>

</html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/PGrLVd?editors=1010.

Per acabar, si la resposta és de tipus `script` s'ha d'anar amb molt de compte, perquè s'executarà. Afegiu un fitxer anomenat `provincies.js` amb el codi següent:

```
var provincies = ['Barcelona', 'Girona', 'Lleida', 'Tarragona'];
```

```
2
3  for (var i=0; i<provincies.length; i++) {
4    alert(provincies[i]);
5  }
```

I comproveu l'efecte amb el codi següent, que realitzarà una petició de tipus `script` i, per consegüent, la resposta serà carregada i executada immediatament:

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <meta charset="utf-8">
6    <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
7  </head>
8
9  <body>
10   <script>
11     $.ajax({
12       url: 'provincies.js',
13       beforeSend: function(jqXHR) {
14         jqXHR.overrideMimeType('application/javascript');
15       },
16       success: processarResposta,
17       dataType: 'script'
18     });
19
20     function processarResposta(dadesJS, statusText, jqXHR) {
21       console.log("Dades rebudes:", dadesJS);
22     }
23
24   </script>
25 </body>
26
27 </html>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/pRLKZO?editors=1010.

Com es pot apreciar, l'script s'executa automàticament i, per tant, no es pot controlar els efectes que tindrà aquest tipus de resposta, ja que depèn completament del servidor. Per aquest motiu s'ha d'estar molt segur que aquest comportament és el desitjat perquè permet executar el possible codi malintencionat rebut com a resposta del servidor.

2.1.5 Events AJAX

Durant l'enviament d'una petició AJAX mitjançant la biblioteca jQuery es disparen tot un seguit d'*events*. Aquests *events* poden ser locals (propis d'una petició concreta) o globals (disparats al document).

L'ordre en què es disparen aquests *events* és el següent:

- **ajaxStart** (global): es dispara quan comença la petició i no hi ha cap altra petició en execució.

- **beforeSend** (local): es dispara abans de realitzar-se l'enviament i permet modificar la petició abans d'enviar-la (per exemple, per sobreescriure el tipus multimèdia).
- **ajaxSend** (global): disparat globalment quan es realitza l'enviament.
- **success** (local): es dispara només si la petició retorna amb una resposta exitosa.
- **ajaxSuccess** (global): disparat en les mateixes condicions que l'anterior, però globalment.
- **error** (local): disparat només quan es produeix un error.
- **ajaxError** (global): igual que l'anterior però disparat globalment.
- **complete** (local): es dispara en finalitzar la petició independentment de si ha estat exitosa o errònia.
- **ajaxComplete** (global): igual que l'anterior però disparat globalment.
- **ajaxStop** (global): es dispara quan no hi ha més peticions AJAX processant-se.

Fixeu-vos que els *events* locals es corresponen amb les opcions a les quals es pot afegir una funció de tipus *callback*: `beforeSend`, `success`, `error` i `complete`.

Per altra banda, a excepció de l'*event* `ajaxStart` i `ajaxStop`, la resta d'*events* globals són rèpliques dels *events* locals i es disparen a continuació d'aquests.

Cal destacar que per detectar quan es dispara un *event* global s'ha d'especificar la detecció al document, tal com es mostra en l'exemple següent:

```
$(document).ajaxSend(function() {
  console.log("S'ha enviat una petició AJAX");
```

Per contra, la detecció d'*event* local s'especifica a la petició concreta que hi està lligada, com es pot apreciar en l'exemple següent:

```
$.ajax({
      url: 'provincies.xml',
      beforeSend: function(jqXHR) {
        console.log("Disparat abans d'enviar una petició concreta");
      });
```

Fixeu-vos que en el primer cas el missatge es mostraria a la consola quan s'enviés qualsevol petició AJAX, mentre que en el segon cas només es mostraria quan s'enviés la petició concreta on s'ha especificat.

2.1.6 Altres opcions del mètode Ajax

El mètode `ajax` ofereix múltiples opcions per crear una petició, a banda dels més utilitzats com `method`, `url` i `data`, podeu trobar els següents, que permeten crear

peticions més avançades afegint dades d'autenticació o modificant les capçaleres per adequar-les als requeriments dels serveis web.

- **async**: per defecte el seu valor és `true`; si s'estableix com a `false`, la petició serà síncrona.
- **contentType**: permet canviar el tipus de contingut enviat en les peticions a altres dominis; si és diferent d'`application/x-www-form-urlencoded`, `multipart/form-data` o `text/plain` s'activaran mesures de comprovació extres al navegador.
- **crossDomain**: permet forçar l'enviament de la petició com si es tractés d'un altre domini, tot i que el destí sigui el mateix en què es troba l'aplicació.
- **headers**: aquesta propietat permet afegir parelles de claus i valors addicionals a la capçalera de la petició. Per exemple, és possible que alguns serveis web requereixin que a la capçalera es trobi un *token* d'autenticació.
- **password**: contrasenya que serà utilitzada en una petició HTTP d'autenticació.
- **username**: nom d'usuari que serà utilitzat en una petició HTTP d'autenticació.

Addicionalment, **tot i que no es recomana fer-lo servir**, jQuery ofereix el mètode `ajaxSetup`, que permet establir uns valors per defecte que seran utilitzats per totes les peticions realitzades pel mètode `ajax` i els seus derivats, com els mètodes `get` i `post`.

Així, per exemple, el següent codi canviaria el mètode d'enviament per defecte (`GET`) per `POST`, cosa que podria fer que altres peticions de l'aplicació deixessin de funcionar o tinguessin efectes no desitjats.

```
1 $.ajaxSetup({
2   method: 'post'
3 });
```

2.2 JSONP i CORS amb jQuery

Sovint és necessari l'accés a dades en serveis webs amb origen diferent. Per poder accedir a aquestes dades depenem del servidor, que ha d'oferir les dades o bé en format JSONP (JSON amb *padding*) o implementar mecàniques per habilitar el CORS (Cross-Origin Resource Sharing).

En tots dos casos jQuery utilitza la mateixa interfície, mitjançant el mètode AJAX, de manera que les tècniques emprades són pràcticament transparents a l'usuari.

API de Flickr

Podeu trobar tota la informació per desenvolupadors de l'API de Flickr a l'enllaç següent: www.goo.gl/tAVTBD.

Aquesta estandardització es veu molt clarament en la implementació de les peticions amb JSONP, ja que en lloc d'afegir una etiqueta `script` amb l'origen de

les dades i passant els paràmetres com una cadena de text, es pot crear una petició AJAX amb les opcions corresponents. Fixeu-vos en una aplicació que consulta l'agregador (*feed*) de l'aplicació `flickr` enviant com a paràmetres l'etiqueta `kittens` i el format `json`.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>
  <script>
    $.ajax({
      url: 'http://api.flickr.com/services/feeds/photos_public.gne',
      success: processarResposta,
      dataType: "jsonp",

      // Nom del paràmetre que indica al servidor el nom de la funció de
          callback, definit a la seva documentació
      jsonp: "jsoncallback",

      // Paràmetres que es passen al servidor, definits a la seva documentació
      data: {
        tags: 'kitten',
        format: 'json'
      }
    });

    function processarResposta(dades) {
      var imatges = dades.items;

      for (var i = 0; i < imatges.length; i++) {
        var $img = $('<img>');
        $img.attr('src', imatges[i].media.m);
        $img.attr('title', imatges[i].title);
        $img.attr('alt', imatges[i].title);
        $('body').append($img);
      }
    };
  </script>
</body>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/PGrLVd?editors=1010.

Com es pot apreciar, només es diferencia d'una crida AJAX estàndard en el fet que s'ha especificat l'opció `jsonp` per indicar el nom del paràmetre corresponent a la funció *callback*, que es farà servir com a *padding* a la resposta, i l'opció `dataType` a la qual s'ha assignat el valor `jsonp`.

Quant a les crides a serveis web que admetin CORS, com també en les implementacions pròpies d'AJAX, no canvia res quan el servidor inclou a la capçalera el paràmetre `Access-Control-Allow-Origin` i el seu valor es correspon amb el del domini on s'executa l'aplicació o el seu valor és `*` i, per consegüent, accessible per a tothom.

Vegeu en l'exemple següent com es consulta una font de dades obertes de l'Ajuntament de Barcelona per recuperar la llista de festivals de l'any 2015 en format JSON:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>
</head>

<body>
<h1>Festivals 2015</h1>

  <script>

    $.ajax({
      url: 'http://dades.eicub.net/api/1/festivals-assistents',
      beforeSend: function(jqXHR) {
        jqXHR.overrideMimeType('application/json');
      },
      success: processarResposta,
      dataType: 'json',
      data: {
        format : 'json.xml',
        Any : 2015,
      }
    });

    function processarResposta(dades, statusText, jqXHR) {
        var $llista = $('<ul>');

        for (var i=0; i<dades.length; i++) {
            var $dada = processarDada(dades[i]);
            $llista.append($dada)
        }

        $('body').append($llista);
    }

    function processarDada(dada) {
      var $item = $('<li>');
      var $enllac = $('<a>');
      $enllac.html(dada.NomDelFestival);
      $enllac.attr('href', dada.Web);
      $enllac.attr('title', dada.Organitzador);
      $item.append($enllac);

      return $item;
    }

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

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/RGXbBr?editors=1010.

En cas que el servei web requereixi autenticació o altres paràmetres, a les opcions que es passen al mètode `ajax` es pot especificar la propietat `xhrFields` per incloure-les en la petició (per exemple, per afegir l'ús de credencials si el servidor les demana), i es faria de la manera següent:

```
$.ajax({
  url: url_en_un_domini_diferent,
  xhrFields: {
    withCredentials: true
  }
});
```

Com que les dades addicionals que pot requerir un servei web que implementi les mecàniques per habilitar CORS poden ser molt diferents, cal consultar-ne la documentació per poder configurar correctament les peticions.

2.3 Accés a dades obertes amb jQuery

L'accés a dades obertes com les que ofereix l'Observatori de dades culturals de Barcelona (www.barcelonadadescultura.bcn.cat/dades-obertes) es realitza de la mateixa manera que altres peticions AJAX mitjançant jQuery, amb l'avantatge que en cas que sigui necessari enviar paràmetres, la tasca se simplifica, ja que no cal modificar la informació de la capçalera i es tracta igual si el mètode d'enviament és `GET` o qualsevol altre.

Vegeu a continuació un exemple d'una consulta que carrega les dades de les "activitats de difusió de les fàbriques de creació" i permet seleccionar, segons el seu identificador, de quina activitat es volen visualitzar les dades.

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>

  <style>
    h1 {
      text-align:center;
    }

    fieldset {
      width: 300px;
      margin: 0 auto;
    }
    label {
      display: inline-block;
      width: 150px;
      font-weight: bold;
    }
    span {
      display: inline-block;
      width: 150px;
    }
    ul {
      list-style-type: none;
      padding: 0;
    }
  </style>

</head>

<body>
  <h1>Activitats de difusió de les fàbriques de creació</h1>
  <fieldset>
    <select id="identificador">
    <option>Selecciona un identificador</option></select>
    <ul>
      <li><label>Id:</label><span id="id"></span></li>
      <li><label>Any:</label><span id="any"></span></li>
      <li><label>Equipament:</label><span id="equipament"></span></li>
      <li><label>Districte:</label><span id="districte"></span></li>
```

```
      <li><label>Ambit:</label><span id="ambit"></span></li>
      <li><label>Assistents:</label><span id="asistents"></span></li>
      <li><label>Notes:</label><span id="notes"></span></li>
      <li><label>Tipus d'equipament:</label><span id="tipusEquipament"></span><
          /li>

      <li><label>Titularitat:</label><span id="titularitat"></span></li>
      <li><label>Activitats de difusió dins dels centres:</label><span id="
          activitats"></span></li>
    </ul>
  </fieldset>

  <script>
     var dades = {};

     $.ajax({
       url: 'http://dades.eicub.net/api/1/fabriquescreacio_difusio',
       beforeSend: function(jqXHR) {
         jqXHR.overrideMimeType('application/json');
       },
       success: processarResposta,
       dataType: 'json',
     });

     function processarResposta(resposta, statusText, jqXHR) {
       var $llistaDesplegable = $('#identificador');

       for (var i = 0; i < resposta.length; i++) {
         var $item = processarDada(resposta[i]);
         dades[resposta[i].Id] = resposta[i];
         $llistaDesplegable.append($item);
       }

       $llistaDesplegable.on('change', actualitzarDadesMostrades);
     }

     function processarDada(dada) {
       var $item = $('<option>');
       $item.attr('value', dada.Id);
       $item.html(dada.Id);
       return $item;
     }

     function actualitzarDadesMostrades(event) {
       var $llistaDesplegable = $('#identificador');
       var dada = dades[$llistaDesplegable.val()]

       actualitzarDadaMostrada('id', dada.Id);
       actualitzarDadaMostrada('any', dada.Any);
       actualitzarDadaMostrada('equipament', dada.Equipament);
       actualitzarDadaMostrada('districte', dada.Districte);
       actualitzarDadaMostrada('ambit', dada.Ambit);
       actualitzarDadaMostrada('asistents', dada.Assistents);
       actualitzarDadaMostrada('notes', dada.Notes || 'No hi ha cap nota');
       actualitzarDadaMostrada('tipusEquipament', dada.TipusEquipament);
       actualitzarDadaMostrada('titularitat', dada.Titularitat);
       actualitzarDadaMostrada('activitats', dada.
           Activitats_de_difusio_dins_dels_centres);

     }

     function actualitzarDadaMostrada(nom, valor) {
       $('#' + nom).html(valor);
     }

  </script>
</body>

</html>
```

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

Com es pot apreciar, en aquest cas no s'han especificat paràmetres perquè la font de dades no els requeria. En els casos que és necessari només cal especificar a l'objecte d'opcions la propietat `data` i assignar els paràmetres necessaris, o en forma de cadena de text o d'objecte, i la biblioteca s'encarregarà d'afegir-los a la capçalera de la forma correcta segons el mètode emprat.

2.4 Mètodes d'ajuda: serialize, serializeArray i param

La biblioteca jQuery ofereix un seguit de mètodes per facilitar l'obtenció i manipulació de dades a l'hora de realitzar peticions, entre els quals cal destacar `serialize`, `serializeArray` i `params`.

El mètode `serialize` converteix la informació dels controls d'un objecte `jQuery` que encapsula un formulari en una cadena de text codificada per fer servir com a paràmetre. Aquesta cadena estarà formada pels controls que disposin de la propietat `name` i el seu valor.

```
<form>
  <input name="nom" />
  <input name="cognom" />
</form>

<div></div>

<script>
  $('input').on('input change', function() {
    var serialitzat = $('form').serialize();
    $('div').html(serialitzat);
  });
</script>
```

Podeu veure aquest exemple en l'enllaç següent: www.codepen.io/ioc-daw-m06/pen/vXoExz?editors=1010.

Com es pot apreciar, això permet establir l'opció `data` d'una petició AJAX directament a partir d'aquesta cadena, ja que inclou tota la informació del formulari sense haver de recórrer els elements i extreure'n les dades una per una.

De forma similar, el mètode `serializeArray` també obté la informació del formulari, però en aquest cas genera un *array* d'objectes JavaScript amb tota la informació del formulari.

```
<form>
  <input name="nom" value="Maria"/>
  <input name="cognom" value="Campmany"/>
</form>
<button>Comprovar</button>

<script>
  $('button').on('click', function() {
    var serialitzat = $('form').serializeArray();
```

```
    console.log(serialitzat);
  });
</script>
```

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

L'*array* d'objectes generats per la invocació del mètode `serializeArray` seria el següent:

```
[
  Object {
    name: "nom",
    value: "Maria"
  },
  Object {
    name: "cognom",
    value: "Campmany"
  }
]
```

Fixeu-vos que aquest objecte es pot utilitzar directament com a valor assignat a l'opció `data`: aquesta opció accepta tant objectes de JavaScript (sense mètodes) com cadenes de text.

Cadena de consulta és una mena de cadena de text amb un format especial utilitzat per enviar peticions i que, inclús, pot portar paràmetres.

Mentre que `serialize` i `serializeArray` faciliten la tasca de convertir les dades introduïdes als controls d'un formulari per fer-les servir com a **dades** en una petició AJAX, el mètode `param` permet convertir objectes de JavaScript, objectes de jQuery i *arrays* en una **cadena de consulta** codificada:

```
var dades = [{
    name: 'nom',
    value: 'Maria'
  }, {
    name: 'cognom',
    value: 'Campmany'
  }, ]

  console.log($.param(dades));
```

Podeu veure aquest exemple en el següent enllaç: www.codepen.io/ioc-daw-m06/pen/xEvbpp?editors=0012.

La cadena de consulta obtinguda en invocar el mètode `param` serà: `nom=Maria&cognom=Campmany`.

Fixeu-vos que mentre que `serialize` i `serializeArray` són mètodes que es criden sobre una instància d'objecte `jQuery` que conté la referència al formulari, el mètode `param` s'invoca directament a partir de la funció `jQuery` passant les dades que s'han de convertir com a argument.

S'ha de tenir en compte que en el cas dels objectes JavaScript, han de tenir forçosament el format següent:

```
[
  {
    name: nom_del_parametre
    value: valor_corresponent
```

```
  }
]
```

Els objectes jQuery han de contenir referències a objectes de tipus `input` en què els controls tinguin definit l'atribut `name` (com en el cas de `serialize` i `serializeArray`), en cas contrari no funcionarà correctament.

Desenvolupament de casos pràctics

Xavier Garcia Rodríguez

Índex

Introducció

Un desenvolupador d'aplicacions no ha de conèixer, només, les paraules clau d'un llenguatge i les estructures bàsiques, sinó que ha de ser capaç d'utilitzar aquests coneixements per implementar aplicacions complexes i saber com integrar-les en serveis de tercers, tant per fer consultes a través de l'API d'un tercer com per integrar biblioteques externes en una aplicació o desenvolupar els propis servidors de proves.

En aquesta unitat, **"Desenvolupament de casos pràctics"**, aprendreu com utilitzar Node.js per crear dos servidors senzills i poder comprovar el correcte funcionament de les vostres aplicacions que requereixin interactuar amb serveis web. Per una banda s'implementarà un servidor RESTful per comprovar les crides AJAX, i per l'altra, s'implementarà un servidor de xat que permeti connectar múltiples usuaris simultàniament.

Seguidament, veureu com utilitzar la biblioteca de Google Maps per crear un mapa simple amb marcadors i finestres d'informació, que es combinarà –amb la connexió mitjanant AJAX– amb un servei web que ofereix dades obertes per mostrar la localització de tots els cinemes de la ciutat de Barcelona.

Per acabar, es farà un repàs pas a pas dels components necessaris per crear un joc en HTML5, incloent-hi com a exemple el codi complet del joc *IOC Invaders* que podeu descarregar, modificar i utilitzar com a base per als vostres propis jocs.

Com que aquesta unitat és completament pràctica, és imprescindible seguir tots els exemples pas a pas, amb excepció del desenvolupament del joc *IOC Invaders*, que és força complicat i només s'ha d'entendre com funciona.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Desenvolupa aplicacions web dinàmiques, reconeixent i aplicant mecanismes de comunicació asíncrona entre client i servidor.

- Avalua els avantatges i els inconvenients d'utilitzar mecanismes de comunicació asíncrona entre client i servidor web.
- Analitza els mecanismes disponibles per a l'establiment de la comunicació asíncrona.
- Utilitza els objectes relacionats.
- Identifica les seves propietats i els seus mètodes.
- Utilitza comunicació asíncrona en l'actualització dinàmica del document web.
- Utilitza diferents formats en l'enviament i en la recepció d'informació.
- Programa aplicacions web asíncrones de manera que funcionin en diferents navegadors.
- Classifica i analitza llibreries que facilitin la incorporació de les tecnologies d'actualització dinàmica a la programació de pàgines web.
- Crea i depura programes que utilitzin aquestes llibreries.

1. Desenvolupament de casos pràctics

Tenir una bona base dels coneixements teòrics és fonamental, però practicar aplicant aquests coneixements és el que us permetrà desenvolupar les vostres pròpies aplicacions per molt complicades que arribin a ser.

S'ha de tenir en compte també que, en programació, és indispensable saber cercar informació a internet, tant documentació sobre el llenguatge com sobre una biblioteca o API concreta o alguna tecnologia amb la qual hàgiu de treballar (com sensors o realitat virtual, per exemple).

Tenir uns coneixements amplis en programació us permetrà entendre com es poden adaptar fàcilment les biblioteques i les noves tecnologies a les vostres aplicacions, per crear un programa de xat, una guia turística amb mapes o un joc en HTML5.

Convé destacar que en el dia a dia dels desenvolupadors d'aplicacions és habitual haver d'utilitzar eines mitjançant la línia d'ordres (compiladors o preprocessadors de CSS) i que moltes d'aquestes eines estan desenvolupades en Node.js. És a dir, estan desenvolupades en JavaScript, però a la banda del servidor.

Conèixer el funcionament de Node.js permet als desenvolupadors crear les seves pròpies eines per a la línia d'ordres, així com desenvolupar servidors tan simples o complexos com sigui necessari per comprovar el funcionament tant d'aplicacions web com dels programes de xat o dels jocs multijugador.

També és molt habitual haver d'utilitzar biblioteques i API de tercers, com per exemple Google Maps o fonts de dades externes. Sovint, per accedir a aquestes API s'ha de crear un compte al lloc web dels proveïdors i per fer-lo servir poden requerir algun tipus de pagament. En el cas de Google molts dels seus serveis són gratuïts fins a un cert punt, però per superar la quota cal activar els pagaments.

Les dades obertes són accessibles per a tothom i l'únic inconvenient per implementar una aplicació que les utilitzi és que la documentació sigui disponible i que admetin CORS o JSONP.

Cal no oblidar el desenvolupament de jocs en HTML5, ja que amb la desaparició de Flash i Java dels navegadors tots els jocs del web han passat a estar desenvolupats en JavaScript. A més a més, amb les millores en la potència dels equips és possible desenvolupar jocs força complexos que utilitzen gràfics en 3D i, fins i tot, preparats per a realitat virtual.

Realitat virtual al navegador

WebVR (webvr.info) és una API de JavaScript que permet utilitzar dispositius de realitat virtual al navegador.

1.1 Desenvolupament amb Node.js

Node.js (nodejs.org) està basat en el motor V8 de JavaScript de Chrome i permet desenvolupar aplicacions en JavaScript que són executades al servidor. Gràcies a això es poden reaprofitar molts dels coneixements adquirits com a desenvolupadors d'aplicacions a la banda del client.

Podeu trobar la documentació de Node.js a l'enllaç següent: goo.gl/40ZGhQ.

Cal destacar que Node.js és una tecnologia relativament recent, però amb molta demanda. Entre les aplicacions més habituals desenvolupades amb Node.js es troben els servidors webs, servidors de xat, jocs multijugador i eines per a la línia d'ordres (com per exemple els preprocessadors de CSS).

1.1.1 Introducció a Node.js

Per desenvolupar una aplicació amb Node.js només cal tenir-lo instal·lat i un editor de text pla. Cal tenir en compte que les aplicacions programades són executades a servidors remots, per posar en marxa un servei o com a línia d'ordres; per consegüent, els missatges que a JavaScript es mostren a la consola de les eines de desenvolupador es mostraran per la terminal. És a dir, els missatges d'error o els missatges que utilitzeu per depurar amb el mètode `console.log` es mostraran a la vostra terminal.

S'ha de tenir en compte que a Node.js es programa amb JavaScript, però hi ha algunes diferències:

- No hi ha disponible cap de les funcions dels navegadors ni del DOM.
- El sistema de mòduls i requeriments és similar al d'ES6 (ES5 no té sistema de mòduls).
- Cada fitxer que es carrega amb `require` és un mòdul que encapsula totes les seves funcionalitats i només exposen alguns mètodes o propietats; no és el mateix que carregar múltiples fitxers JavaScript al navegador.

Hi ha prou similituds per desenvolupar algunes aplicacions senzilles, però es recomana consultar la documentació oficial de Node.js i els tutorials si voleu aprofundir en les seves possibilitats.

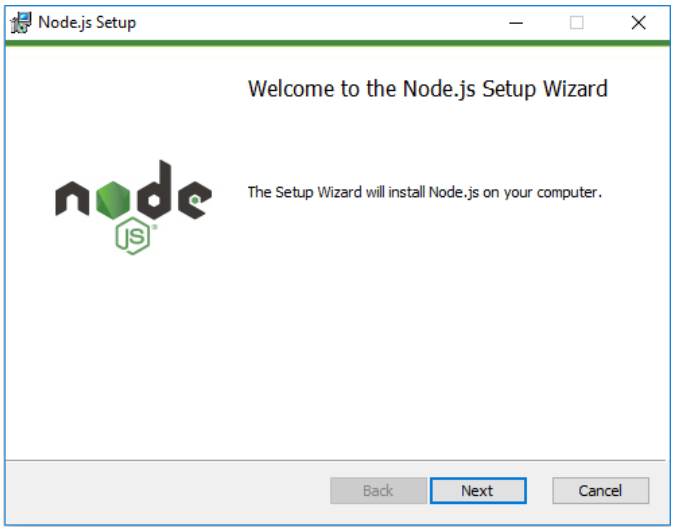
Instal·lació de Node.js

Abans de poder desenvolupar qualsevol aplicació amb Node.js cal instal·lar-lo. El procés és molt senzill en totes les plataformes, tot i que s'ha d'anar amb compte amb la versió que necessiteu: en alguns sistemes operatius, per defecte, s'instal·len versions molt desactualitzades.

Per instal·lar Node.js per a Windows s'han de seguir els passos següents:

- Entreu a la pàgina de descàrregues de Node.js: nodejs.org/en/download.
- Seleccioneu el vostre sistema operatiu: a la banda superior us apareixerà per defecte la darrera versió de l'instal·lable per a Windows 64 bits i Mac. En cas que vulgueu algun altre format, el podeu trobar a sota.
- Una vegada descarregat l'instal·lable per a Windows heu de fer doble clic sobre el fitxer i s'obrirà l'instal·lador, que es mostra a la figura 1.1.

FIGURA 1.1. Instal·lador de Node.js per a Windows

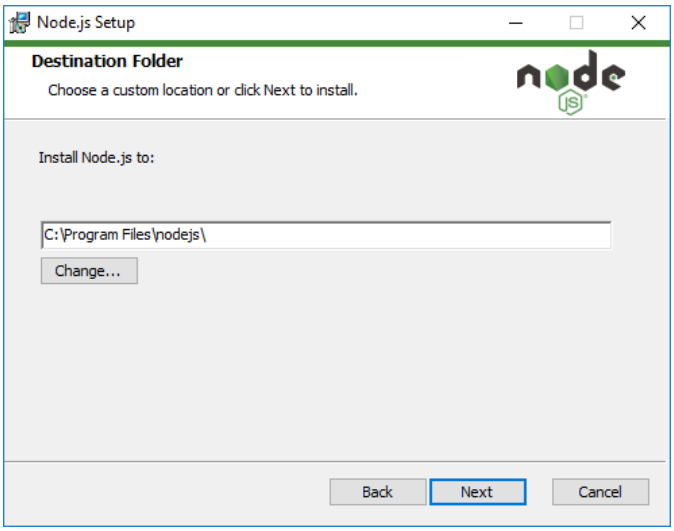


- S'ha d'acceptar la llicència d'ús, com es mostra a la figura 1.2.

FIGURA 1.2. Llicència de Node.js per a Windows

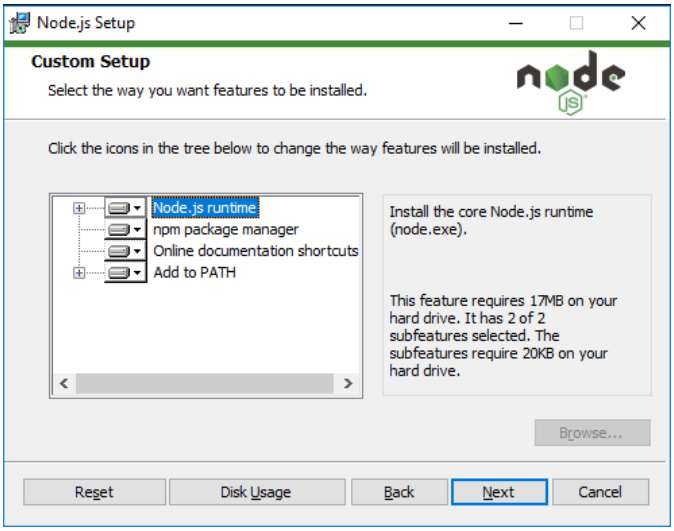
- Seguidament s'ha de seleccionar la carpeta de destí, com es mostra a la figura 1.3.

FIGURA 1.3. Selecció de la carpeta d'instal·lació de Node.js



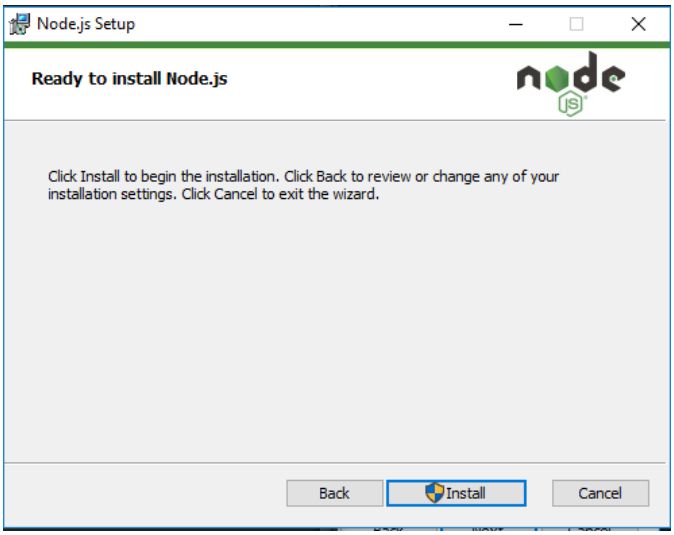
- El següent pas és seleccionar les característiques a instal·lar. L'instal·lador inclou el mateix Node.js, accessos directes a la documentació, l'instal·lador de paquets npm i la configuració automàtica de la ruta, tal com es mostra a la figura 1.4.

FIGURA 1.4. Selecció de característiques d'instal·lació de Node.js



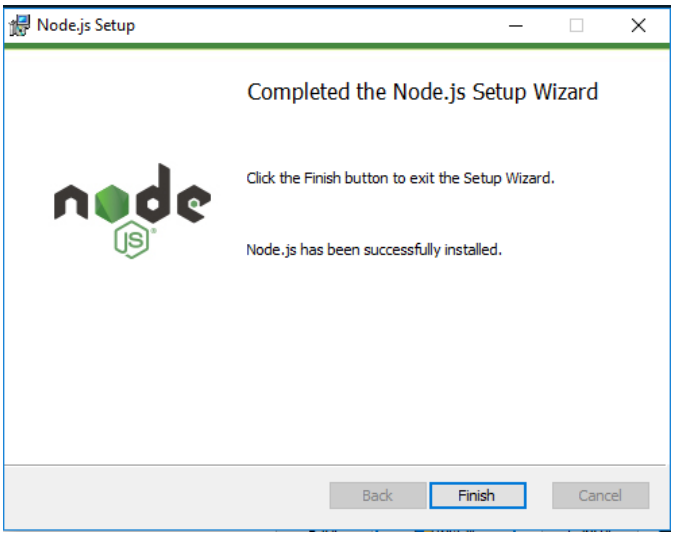
- L'últim pas és confirmar la instal·lació, com es mostra a la figura 1.5.

Figura 1.5. Confirmació d'instal·lació



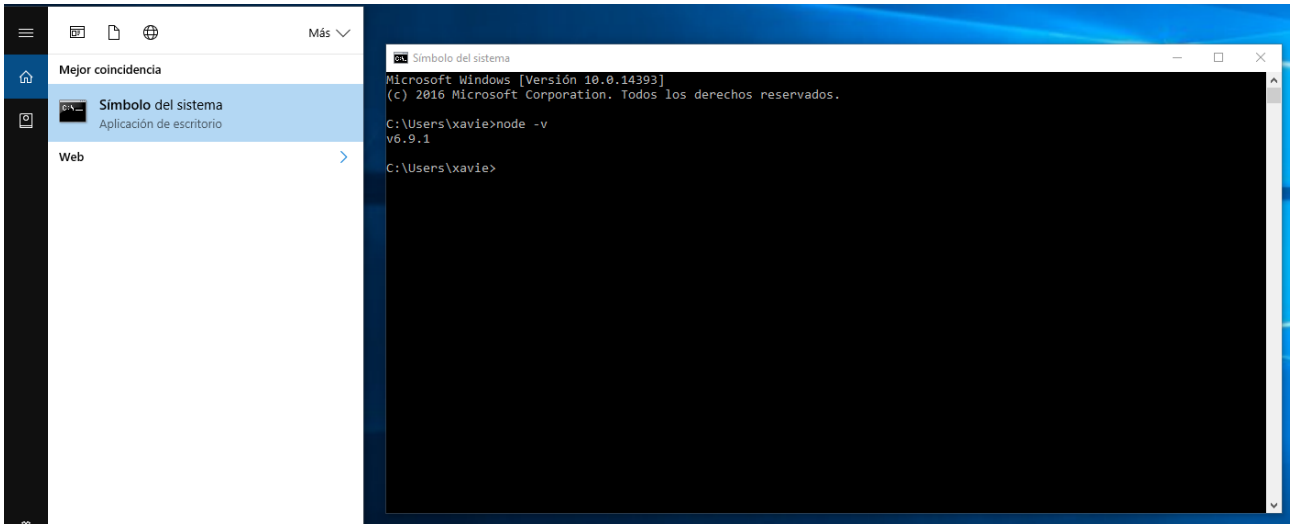
- Una vegada finalitzada la instal·lació rebreu la confirmació d'èxit, que es pot veure a la figura 1.6.

Figura 1.6. Finalització d'instal·lació



Per comprovar que s'ha instal·lat correctament heu d'accedir al símbol del sistema, com es pot apreciar a la figura 1.7, o a la terminal millorada de Windows (anomenada PowerShell).

FIGURA 1.7. Comprovació de la versió de Node.js al símbol del sistema



En tots dos casos amb l'ordre `node -v` se us ha de mostrar el número de la versió. En cas contrari s'haurà produït un error a la instal·lació o a la configuració; proveu de reinstal·lar-lo amb totes les opcions per defecte i pareu atenció a qualsevol possible missatge d'error.

Addicionalment s'haurà instal·lat el gestor de paquets npm. Per comprovar-ho podeu escriure a la terminal `npm -v` i us n'ha de mostrar la versió. Aquest gestor us permetrà instal·lar nous mòduls i actualitzar la versió de node i del mateix gestor de paquets.

npm

npm (www.npmjs.com) és, per defecte, el gestor de paquets per a Node.js. Alguns ho interpreten com a un acrònim de *Node Package Manager*.

Els usuaris de Mac, una vegada finalitzada la instal·lació, poden comprovar que s'ha instal·lat correctament escrivint a la terminal:

```
1 ~ $ node –v
```

APT

APT, acrònim d'*Advanced Packaging Tool* (Eina Avançada d'Empaquetat), és un sistema de gestió de paquets creat pel projecte Debian (inclòs a Ubuntu) que permet instal·lar i eliminar programes mitjançant la línia d'ordres.

La instal·lació amb la distribució de Linux Ubuntu es pot fer utilitzant el gestor de paquets APT des de la terminal:

```
1 sudo apt–get update
2 sudo apt–get install nodejs
```

Altres versions de Node.js a Ubuntu

La versió de node que s'instal·la utilitzant el gestor de paquets d'Ubuntu acostuma a estar molt desactualitzada. Si necessiteu fer servir una versió més recent, podeu seguir els passos descrits a l'enllaç següent: goo.gl/oXkiu9.

Cal destacar que a Ubuntu l'executable de node es diu `nodejs` per evitar conflictes amb altres paquets, en canvi, a Mac i Windows (o altres distribucions de Linux) el nom de l'executable és `node`.

```
1 nodejs –v
```

Fixeu-vos que la instal·lació de Node.js mitjançant el gestor de paquets també inclou la instal·lació d'npm:

```
1 npm –v
```

1.1.2 Primer programa amb Node.js: Hola món

Els programes desenvolupats amb Node.js s'anomenen *mòduls* i en desenvoluparlos s'utilitzen altres mòduls que formen part de Node.js; com per exemple el mòdul `http`, per crear servidors que processin peticions HTTP.

Per altra banda, és habitual haver d'utilitzar codi de tercers, siguin mòduls simples, biblioteques senceres o *frameworks* (entorns de treball). En aquest cas, es parla de *dependències*, ja que el mòdul principal depèn d'aquests mòduls per funcionar.

'Framework' o entorn de treball

Un *framework*, en català 'entorn o marc de treball', és una infraestructura de programari que, en la programació orientada a objectes, facilita la concepció de les aplicacions mitjançant la utilització de biblioteques de classes o generadors de programes.

Creeu un fitxer anomenat `hola-mon.js`, dins d'un directori propi que servirà de contenidor per al vostre mòdul, amb el contingut següent:

```
1 var http = require('http');
2 http.createServer(function(peticio, resposta) {
3         resposta.writeHead(200, {'Content-Type':'text/plain;charset=utf-8'});
4         resposta.end('Hola món');
5 }).listen(8080, '127.0.0.1');
6 console.log('Servidor executant-se a http://127.0.0.1:8080/');
```

Tingueu en compte que, en cas que treballeu amb un servidor remot, en lloc de la IP 127.0.0.1 heu de posar la IP del vostre servidor.

Per posar-lo en marxa, haureu d'escriure a la terminal del sistema operatiu `node hola-mon.js` o, si treballeu amb Ubuntu, `nodejs hola-mon.js`.

Si entreu amb el navegador a l'adreça del servidor especificant el port 8080, veureu el missatge `Hola Món`.

Aplicacions web complexes

Per al desenvolupament d'aplicacions web complexes amb Node.js es recomana utilitzar el *framework* Express (expressjs.com).

Podeu trobar tota la documentació del mòdul `HTTP` a l'enllaç següent: goo.gl/C9GPNk.

A la primera línia es demana carregar el mòdul `http`. Aquest mòdul exposa el mètode `createServer`, que serveix per crear un nou servidor HTTP. Aquest servidor gestionarà les peticions rebudes.

Aquesta funció rebrà dos objectes com a paràmetres: la petició que s'ha enviat des del navegador (`http.ClientRequest`) i la resposta (`http.ServerResponse`), que serà retornada al client.

En aquest cas, s'utilitza el mètode `writeHead` de la resposta per escriure el codi de retorn a la capçalera i s'informa que s'ha processat correctament (200) i el tipus de contingut: un text pla codificat com a UTF-8. Això permet que els signes d'accentuació es mostrin correctament.

Seguidament, s'afegeix el cos de la resposta a enviar, les dades, que en aquest cas només inclouen la cadena `Hola món\n`. Si accediu a la pàgina des del navegador, probablement s'haurà afegit automàticament un codi HTML similar al següent:

```
1 <html>
2   <head>
3     <link rel="alternate stylesheet" type="text/css" href="resource://gre-
          resources/plaintext.css" title="Ajusta les línies llargues">
4   </head>
5   <body>
6     <pre>Hola Món</pre>
7   </body>
```

```
</html>
```

En canvi, si inspeccioneu la resposta, veureu que el seu contingut és `Hola Món`. Això es deu al fet que els navegadors, en rebre determinats tipus de continguts (en aquest cas text pla), el representen de la forma que consideren més adient.

Cal destacar que la funció `createServer` retorna un objecte de tipus `http.Server`, i és a partir d'aquest que s'invoca el mètode `listen`, que indica que s'han d'escoltar les peticions de l'adreça `127.0.0.1` pel port `8080`.

Un mètode alternatiu d'implementar el mateix exemple seria el següent:

```
var http = require('http');
var server = http.createServer();
server.on('request', function(peticio, resposta) {
        resposta.writeHead(200, {'Content-Type':'text/plain;charset=utf-8'});
        resposta.end('Adeu Món');
});
server.listen(8080,"127.0.0.1");
```

Fixeu-vos que en aquest cas primer s'ha creat el servidor sense passar cap paràmetre. Seguidament s'ha cridat el mètode `on` del servidor, que permet detectar diferents *events*; concretament es demana detectar l'*event* `request` (que és disparat en rebre una petició) i que es cridi la funció passada com a argument. Finalment s'indica que s'han d'escoltar les peticions pel port `8080` a l'adreça IP `127.0.0.1`.

1.1.3 Gestor de paquets npm

Node.js treballa amb un gestor de paquets que permet instal·lar nous mòduls molt fàcilment. A més a més, permet enregistrar els vostres mòduls al seu repositori públic, de manera que altres usuaris puguin trobar-los i utilitzar-los (si es vol restringir-los a determinats usuaris, cal un compte de pagament).

El gestor de paquets npm s'instal·la automàticament amb l'instal·lador de Node per a Windows i Mac; en canvi, a Linux cal instal·lar-lo individualment:

```
sudo apt-get install npm
```

Podeu trobar el cercador de mòduls d'npm a la seva pàgina principal: www.npmjs.com.

Una vegada instal·lat, es pot actualitzar així mateix, com si es tractés de qualsevol altre mòdul. Per exemple, a Windows i Mac es pot actualitzar així:

```
npm install npm -g
```

Mentre que a Linux s'han de demanar permisos d'administrador:

```
sudo npm install npm -g
```

En tots dos casos, l'opció `-g` indica que l'actualització s'ha d'aplicar globalment.

De la mateixa manera, per instal·lar qualsevol mòdul es fa servir la mateixa ordre. Per exemple, per instal·lar el mòdul `express` es faria de la manera següent:

```
1 npm install express
```

Si no s'especifica el nom del mòdul i es troba el fitxer `package.json`, s'instal·laran les dependències indicades en aquest fitxer. Gràcies a aquesta característica no heu d'incloure els mòduls i dependències externes (és a dir, la carpeta `node-modules`) al vostre sistema de control de versions.

Sistemes de control de versions

Els sistemes de control de versions, com GIT o Mercurial, permeten gestionar tots els canvis realitzats en un projecte. Podeu trobar més informació en l'enllaç següent: goo.gl/fo2T51.

De la mateixa manera, si descarregueu un programa de tercers, només haureu d'utilitzar l'ordre `npm install` perquè totes les dependències necessàries es descarreguin i s'instal·lin.

Per altra banda, quan es vol actualitzar un mòdul s'ha de fer servir l'ordre `update`. Per exemple, per actualitzar el mòdul `express` s'utilitzaria l'ordre següent:

```
1 npm update express
```

En cas de no especificar el nom del mòdul, s'actualitzaran tots els mòduls.

L'ordre que cal utilitzar per eliminar un mòdul és `remove`; aquesta ordre elimina tant el mòdul indicat com les seves dependències.

```
1 npm remove express
```

Cal destacar que, si s'utilitza sense indicar el mòdul que s'ha d'eliminar, es demanaran permisos d'administrador.

Per resumir, les ordres més importants del gestor de paquets npm són les següents:

- **install**: per instal·lar nous mòduls o actualitzar el mateix npm.
- **update**: per actualitzar mòduls.
- **remove**: per eliminar un mòdul instal·lat.
- **ls**: llista tots els mòduls.
- **-g**: per indicar que l'ordre s'aplica a l'entorn global.

Instal·lació global i local de mòduls

El gestor de paquets npm permet instal·lar els mòduls de forma global o local. La diferència és que si es fa localment, es copiarà dins del directori `node_modules` del mòdul i serà utilitzable només per aquest mòdul concret. En canvi, si s'utilitza l'opció `-g`, s'afegirà dins del directori `node_modules` del directori d'instal·lació de node i serà accessible globalment, és a dir, per a tots els projectes.

Com que el gestor de paquets npm s'ha d'utilitzar globalment, a l'hora d'actualitzar-lo sempre s'haurà de fer globalment. En canvi, quan requeriu un

mòdul per un projecte específic només cal afegir-lo a l'entorn local del projecte (per exemple, el mòdul `express`).

Altres mòduls que s'acostumen a instal·lar globalment són els preprocessadors de CSS, que converteixen el codi LESS o SASS en CSS, els automatitzadors com gulp i grunt o els compiladors d'ES6, que compilen el codi ES6 en ES5 (JavaScript 5), admès pràcticament per tots els navegadors.

gulp i grunt

gulp i grunt són dues eines que permeten automatitzar l'assemblatge d'una aplicació web: compilació d'ES6, precompilació de LESS o SAAS, optimització de fitxers CSS i JS.

Per llistar tots els mòduls instal·lats en un projecte es pot utilitzar l'ordre `node ls`, afegint la opció `-g` si es volen llistar els mòduls globals.

Descripció del mòdul: 'package.json'

Proveu d'afegir el mòdul `express` dins del directori on esteu treballant amb Node.js amb l'ordre següent:

```
1 npm install express
```

En acabar la instal·lació veureu els següents missatges d'advertència (lleugerament diferents segons el sistema operatiu utilitzat):

```
1 npm WARN enoent ENOENT: no such file or directory, open '/home/xavier/hola-mon/
      package.json'
2 npm WARN helloworld No description
3 npm WARN helloworld No repository field.
4 npm WARN helloworld No README data
5 npm WARN helloworld No license field.
```

La primera advertència es produeix perquè s'espera que el mòdul inclogui un fitxer anomenat `package.json` amb la informació del mòdul, i no l'hem afegit. La resta d'avisos es produeixen per la mateixa raó: com que no troba el fitxer, no pot llegir cap dels camps que es requereixen.

Aquest fitxer es pot generar manualment amb qualsevol editor de text pla o es pot inicialitzar amb l'ordre `npm init` (si premeu enter, agafarà el valor mostrat entre parèntesis):

```
1  xavier@Ubuntu-Node:~/hola-mon$ npm init
2  This utility will walk you through creating a package.json file.
3  It only covers the most common items, and tries to guess sensible defaults.
4
5  See 'npm help json' for definitive documentation on these fields
6  and exactly what they do.
7
8  Use 'npm install <pkg> --save' afterwards to install a package and
9  save it as a dependency in the package.json file.
10
11 Press ^C at any time to quit.
12 name: (hola-mon)
13 version: (1.0.0)
14 description: Primer programa amb Node.js
15 entry point: (hola-mon.js)
16 test command:
17 git repository:
18 keywords:
19 author: Xavier Garcia
20 license: (ISC)
21 About to write to /home/xavier/hola-mon/package.json:
```

```
{
  "name": "hola-mon",
  "version": "1.0.0",
  "description": "Primer programa amb Node.js",
  "main": "hola-mon.js",
  "dependencies": {
    "express": "^4.14.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "el vostre nom",
  "license": "ISC"
}


Is this ok? (yes)
```

El resultat és el mateix que si creeu un fitxer manualment amb el contingut en format JSON:

```
{
  "name": "hola-mon",
  "version": "1.0.0",
  "description": "Primer programa amb Node.js",
  "main": "hola-mon.js",
  "dependencies": {
    "express": "^4.14.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "el vostre nom",
  "license": "ISC"
}
```

Si proveu d'instal·lar el mòdul `express` (tot i que ja es troba instal·lat), veureu que han desaparegut tots els avisos, excepte un que indica que no s'ha especificat cap camp per al repositori. Aquest avís el podeu ignorar perquè només és rellevant si publiqueu el vostre codi i feu servir un sistema de control de versions.

El fitxer 'package.json'

Podeu trobar tota la documentació sobre el fitxer `package.json` a l'enllaç següent: goo.gl/Q0HlTj.

Com es pot veure, en el fitxer generat s'hi afegeix la següent informació del mòdul:

- **name**: nom del mòdul.
- **version**: número de la versió.
- **description**: descripció del mòdul.
- **main**: fitxer principal del mòdul.
- **dependencies**: llista de mòduls amb la corresponent versió que utilitza aquest mòdul. Aquesta informació permet instal·lar un mòdul mitjançant npm i, alhora, que es descarreguin les seves dependències en la versió apropiada.
- **scripts**: aquesta és una propietat especial que permet afegir scripts specials; en aquest cas l'escript s'executaria en cridar l'ordre `npm test`. Podeu trobar-ne més informació a l'enllaç següent: goo.gl/QmpQ0s.

- **license**: llicència que s'aplica al vostre mòdul.
- **author**: nom del desenvolupador o desenvolupadors. Aquest camp admet més informació: per exemple, es pot utilitzar el format següent per indicar el nom i el correu electrònic del desenvolupador:

```
{
  "name": "el vostre nom",
  "email": "el vostre correu electrònic"
}
```

Cal tenir en compte que en eliminar un mòdul amb `npm remove`, no s'elimina del llistat de dependències, sinó que s'ha d'editar el fitxer `package.json` i eliminar manualment la línia que correspongui.

1.1.4 Exemples avançats

Com us podeu imaginar, crear un servidor que mostri sempre el mateix missatge a l'usuari no és gaire útil, normalment serà necessari analitzar quin és l'URL demanat al servidor i el mètode emprat per fer la petició (`GET`, `POST`, `PUT`...).

Combinant l'URL, el mètode d'enviament i els paràmetres es pot implementar un encaminador (*router*, en anglès), un component que s'encarregui de gestionar quin component generarà la resposta de la petició.

Cal tenir en compte que Node.js, al contrari que el JavaScript executat al navegador, sí que pot llegir i escriure del disc del servidor. Així doncs, es poden realitzar operacions de lectura i escriptura, per exemple, per generar un registre d'errors o d'usuaris connectats.

Obtenció de l'URL i el mètode

Obtenir l'URL i el mètode de la petició és molt senzill. Quan es dispara l'*event* `request` al servidor, s'invoca la funció associada a la detecció de l'*event*, que rep com a primer paràmetre un objecte. Aquest conté la informació de la petició i, com a segon paràmetre, un altre objecte que permet gestionar la resposta. A partir de l'objecte que conté la informació de la petició es pot extreure tant el mètode utilitzat com l'URL de la petició. A continuació podeu veure un exemple:

```
var http = require('http');
var server = http.createServer();
server.on('request', function(peticio, resposta) {
  var metode = peticio.method;
  var url = peticio.url;
  resposta.writeHead(200, {'Content-Type':'text/plain;charset=utf-8'});
  resposta.end('Rebuda petició per l\'URL ' + url + ' i el mètode ' + metode);
});
server.listen(8080,"127.0.0.1");
```

Si proveu d'accedir des del vostre navegador a l'adreça `127.0.0.1:8080/hola` veureu per pantalla el missatge següent:

```
Rebuda petició per l'URL /hola i el mètode GET
```

Com es pot apreciar, l'objecte `peticio` (que és una instància d'`http.ClientRequest`) permet accedir tant al mètode d'enviament com a l'URL a través dels mètodes `method` i `url` respectivament.

A partir d'aquests dos paràmetres es podria implementar un servei web REST molt simple, ja que només cal discriminar l'URL i el mètode de la petició per portar a terme una acció o una altra.

Per comprovar el funcionament amb el mètode `POST`, podeu crear un fitxer nou anomenat `formulari-node.html` amb el codi següent:

```
<html>
<body>
  <select id="metode">
    <option value="GET" selected>GET</option>
    <option value="POST">POST</option>
    <option value="PATCH">PATCH</option>
    <option value="DELETE">DELETE</option>
  </select>
  <button id="peticio">Enviar Petició</button>
  <p>Resposta:</p>
  <div id="resposta"></div>

  <script>
    var boto = document.getElementById('peticio');
    var resposta = document.getElementById('resposta');
    var metode = document.getElementById('metode');

    boto.addEventListener("click", function() {
      var httpRequest = new XMLHttpRequest();

      httpRequest.open(metode.value, 'http://127.0.0.1:8080/prova', true);
      httpRequest.send(null);

      httpRequest.onload = function () {
        resposta.innerHTML = httpRequest.responseText;
      }
    });
  </script>
</body>
</html>
```

Com es pot apreciar, s'ha creat una llista desplegable amb HTML que permet seleccionar el mètode d'enviament i un botó per realitzar la petició. Una vegada es fa clic al botó s'envia una petició AJAX, utilitzant el mètode seleccionat a la llista, a l'adreça `127.0.0.1`, port `8080` i URL `prova`.

Una vegada arriba la resposta es dispara l'*event* `load` i es crida la funció assignada a `onload`. Aquesta funció estableix com a contingut HTML la resposta en mode text. Fixeu-vos que no cal sobreescriure el tipus de contingut perquè el servidor l'enviarà correctament.

Malauradament, en molts casos aquest exemple no funcionarà a causa de la 'política del mateix origen' (*same-origin policy*). Com que en aquest cas teniu accés directament a l'aplicació del servidor, només cal incloure els mecanismes

CORS adequats per permetre l'accés des de qualsevol adreça i utilitzant qualsevol mètode:

Política del mateix origen

Es tracta d'una normativa que implementen tots els navegadors; podeu trobar-ne més informació a l'enllaç següent: goo.gl/qMDYhF.

```
var http = require('http');
var server = http.createServer();
server.on('request', function(peticio, resposta) {
  var metode = peticio.method;
  var url = peticio.url;

  resposta.setHeader('Access-Control-Allow-Origin', '*');
  resposta.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT,
      PATCH, DELETE');
  resposta.writeHead(200, {'Content-Type':'text/plain;charset=utf-8'});

  resposta.end('Rebuda petició per l\'URL ' + url + ' i el mètode ' + metode);
});
server.listen(8080,"127.0.0.1");
```

En aquesta versió, el servidor afegeix a la capçalera de la resposta el paràmetre `Access-Control-Allow-Origin=*`, fet que permet realitzar peticions AJAX des de qualsevol servidor; i el paràmetre `Access-Control-Allow-Mehtods=GET, POST, OPTIONS, PUT, PATCH, DELETE`, de manera que permet fer servir directament tots els mètodes HTTP.

Fixeu-vos en una diferència important: s'ha fet servir el mètode `setHeader` en lloc de `writeHead`. Aquest últim el que fa és escriure la capçalera amb el codi de resposta i el contingut passat com a paràmetre (opcionalment), de manera que no es pot modificar; en canvi, el mètode `setHeader` permet afegir paràmetres a la capçalera sense escriure-la, de manera que es poden afegir tants paràmetres com sigui necessari (per exemple, pot ser que algunes capçaleres només calgui enviar-les en alguns casos concrets).

Obtenció de paràmetres

Per obtenir els paràmetres de la petició teniu dues opcions: realitzar una implementació pròpia per dividir la cadena formada pels paràmetres o utilitzar mòduls de Node.js `url` i `querystring`. El primer d'aquests mòduls permet descompondre un URL en fragments i el segon converteix una cadena de consulta en un objecte de JavaScript.

En l'exemple següent podeu veure com s'han afegit dos paràmetres a la petició i com es fa el tractament dels dos casos possibles: l'enviament utilitzant el mètode `GET` (paràmetres codificats en l'URL) o els altres mètodes (paràmetres integrats al cos de la petició).

Primerament s'afegeixen dos camps de text per enviar la informació al servidor. Fixeu-vos que s'ha de fer un tractament especial en el cas del mètode `GET`, ja que les dades s'envien afegint-les a l'URL i no pas com a dades:

```
<html>

<body>
  <h1>Petició</h1>
  <select id="metode">
    <option value="GET">GET</option>
```

```
    <option value="POST" selected>POST</option>
    <option value="PATCH">PATCH</option>
    <option value="DELETE">DELETE</option>
  </select>
  <label>Nom:<input type="text" id="nom" /></label>
  <label>Cognom:<input type="text" id="cognom" /></label>
  <button id="peticio">Enviar Petició</button>
  <h1>Resposta</h1>
  <div id="resposta"></div>

  <script>
    var boto = document.getElementById('peticio');
    var resposta = document.getElementById('resposta');
    var metode = document.getElementById('metode');
    var nom = document.getElementById('nom');
    var cognom = document.getElementById('cognom');

    boto.addEventListener("click", function(e) {
      var url = "http://127.0.0.1:8080/proves";
      var dades = "nom=" + encodeURI(nom.value);
      dades += "&cognom=" + encodeURI(cognom.value);

      if (metode.value == "GET") {
        url +="?" + dades;
        dades = null;
      }

      var httpRequest = new XMLHttpRequest();
      httpRequest.open(metode.value, url, true);
      httpRequest.send(dades);
      httpRequest.onload = function (e) {
        resposta.innerHTML = httpRequest.responseText;
      }
    });
  </script>
</body>

</html>
```

Pel que fa al codi del servidor cal afegir-hi més canvis. Per una banda, s'han carregat els mòduls `url` i `querystring` per analitzar els URL i les cadenes de consulta respectivament; i per altra, s'ha de processar el cos del missatge per obtenir els paràmetres en el cas dels mètodes diferents de `GET`.

```
var http = require('http');
var queryString = require('querystring');
var url = require('url');

var server = http.createServer();
server.on('request', function(peticio, resposta) {
  resposta.setHeader('Access-Control-Allow-Origin', '*');
  resposta.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT,
      PATCH, DELETE');
  resposta.writeHead(200, {'Content-Type':'text/html;charset=utf-8'});

  var cos = '';
  peticio.on('data', function(dades) {
    if (cos.length>1e6) { // Limita la mida a 1.000.000 de bytes = 1e6
      peticio.connection.destroy();
    } else {
      cos += dades;
    }

  }).on('end', function() {
    var params;
    var dades;

    if (peticio.method === 'GET') {
```

```
24        dades = url.parse(peticio.url).query;
25      } else {
26        dades = cos;
27      }
28
29      params = queryString.parse(dades);
30
31      resposta.write('Rebuda petició per l\'URL ' + peticio.url + ' i el mètode '
            + peticio.method +"<br>");
32      resposta.write("<b>Nom:</b> " + decodeURI(params.nom) + "<br>");
33      resposta.write("<b>Cognom:</b> " + decodeURI(params.cognom) + "<br>");
34      resposta.end();
35    });
36  });
37
38  server.listen(8080,"127.0.0.1");
```

S'ha de tenir en compte que el cos de la petició s'obté d'un flux de dades, ja que la petició pot incloure elements molt pesats (per exemple, la pujada de fitxers o imatges) i no es llegeix d'una tacada. Per aquesta raó cal detectar l'*event* `data` de la petició, que indica que han arribat noves dades, i concatenar-la per obtenir el cos complet i l'*event* `end`, que indica que s'ha llegit completament la petició.

Quan la resposta conté codi HTML, és més adient utilitzar com a `Content-Type` el tipus `text/html`.

En aquest exemple s'ha afegit una mesura de seguretat extra: es comprova la mida del cos actual i, si supera el milió de bytes, interromp la connexió. En cas contrari, concatena el nou fragment al cos fins que es detecta l'*event* `end`, que indica que ha finalitzat la recepció de la petició.

Una vegada s'han obtingut totes les dades cal processar-les perquè els paràmetres arriben com una cadena de consulta. En aquest cas també cal distingir entre el mètode `GET` i la resta, ja que el mètode GET inclou la cadena de consulta a l'URL. Per obtenir-la s'ha d'analitzar i accedir a la propietat `query`, com es pot veure en aquest exemple: `url.parse(peticio.url).query`.

Per altra banda, en el cas dels mètodes `POST`, `PATCH`, `DELETE`, etc., la cadena de consulta és el cos del missatge. Una vegada s'ha obtingut la cadena de consulta, d'una manera o altra, cal analitzar-la utilitzant el mòdul `querystring`, com es pot veure a l'exemple següent: `params = queryString.parse(dades)`.

Una vegada obtinguts els paràmetres com un objecte de JavaScript, ja s'hi pot treballar accedint als valors i utilitzant la notació de claudàtors o la notació de punts (per exemple: `params.nom`).

Cal destacar que, tant en el fitxer del client com del servidor, s'han codificat i descodificat els paràmetres utilitzant les funcions `encodeURI` i `decodeURI` respectivament; cal fer-ho així per evitar problemes en incloure caràcters que poden corrompre la informació.

Fixeu-vos que, en aquest exemple, en lloc d'enviar el contingut de la resposta utilitzant el mètode `end`, s'ha afegit per parts utilitzant el mètode `write`. La diferència principal entre l'un i l'altre (com passa amb `setHeader` i `writeHead`) és que una vegada s'ha invocat el mètode `end` ja no es pot afegir res més al cos de la resposta; en canvi, es pot invocar el mètode `write` tantes vegades com sigui necessari per construir la resposta.

Encaminament ('routing')

En aquest context, el terme *encaminament* es refereix a l'acció que s'ha de realitzar (o pàgina que s'ha de carregar) segons l'URL i el mètode d'enviament d'una petició. Per realitzar aquest encaminament, en els casos més simples es pot implementar una solució pròpia, tot i que és recomanable fer servir un *framework* com Express o el mòdul de tercers `router`.

A continuació podeu veure un exemple, amb una implementació pròpia, que retorna diferents respostes segons l'URL seleccionat del menú desplegable. Per una banda, s'ha de generar un fitxer HTML que s'executa al navegador del client, i per altra, el codi del servidor HTPP de Node.js.

```
<html>
<body>
  <h1>Petició</h1>
  <select id="metode">
    <option value="GET">GET</option>
    <option value="POST" selected>POST</option>
    <option value="PATCH">PATCH</option>
    <option value="DELETE">DELETE</option>
  </select>
<select id="provincia">
    <option value="Barcelona" selected>Barcelona</option>
    <option value="Girona" selected>Girona</option>
    <option value="Lleida">Lleida</option>
    <option value="Tarragona">Tarragona</option>
  </select>

  <button id="peticio">Enviar Petició</button>
  <h1>Resposta</h1>
  <div id="resposta"></div>
  <script>
    var boto = document.getElementById('peticio');
    var resposta = document.getElementById('resposta');
    var metode = document.getElementById('metode');
    var provincia = document.getElementById('provincia');

    boto.addEventListener("click", function(e) {
      var url = "http://127.0.0.1:8080/" + provincia.value;
      var httpRequest = new XMLHttpRequest();

      httpRequest.open(metode.value, url, true);
      httpRequest.send(null);

      httpRequest.onload = function (e) {
        resposta.innerHTML = httpRequest.responseText;
      }
      });
  </script>
</body>
</html>
```

En aquest cas, com que no s'utilitzaran paràmetres, només cal especificar el mètode i l'URL de destí. Per aquest mateix motiu no cal carregar el mòdul `querystring`, però sí que és necessari carregar el mòdul `url` per obtenir la ruta que s'ha de processar.

```
var http = require('http');
var url = require('url');

var server = http.createServer();
server.on('request', function(peticio, resposta) {
```

```
  resposta.setHeader('Access-Control-Allow-Origin', '*');
  resposta.setHeader('Access-Control-Allow-Methods', 'GET, POST, OPTIONS, PUT, PATCH, DELETE');
  resposta.writeHead(200, {'Content-Type':'text/html;charset=utf-8'});

  var urlEncaminament = url.parse(peticio.url).path;
  encaminar(urlEncaminament, peticio.method, resposta);
  resposta.end();
});

server.listen(8080,"127.0.0.1");

function encaminar(url, metode, resposta) {
  var vista;

  switch (url) {
    case '/Barcelona':
      vista = controladorBarcelona(metode);
      break;

    case '/Girona':
      vista = controladorGirona(metode);
      break;

    case '/Lleida':
      vista =  controladorLleida(metode);
      break;

    case '/Tarragona':
      vista = controladorTarragona(metode);
      break;
    }

    resposta.write(vista);
}


function controladorBarcelona(metode) {
  var vista = "<h2>Barcelona</h2>";

    switch (metode) {
      case 'GET':
        vista += "Simulació: dades de Barcelona obtingudes";
        break;

      case 'POST':
        vista += "Simulació: dades de Barcelona afegides";
        break;

      case 'DELETE':
        vista += "Simulació: dades de Barcelona eliminades";
        break;

      case 'PATCH':
        vista += "Simulació: dades de Barcelona actualitzades";
        break;
    }

    return vista;
}

function controladorGirona(metode) {
        return "<h2>Girona</h2>No hi ha cap acció especial per a <b>Girona</b>";
}

function controladorLleida(metode) {
        var vista = "<h2>Lleida</h2>";
        if (metode === 'GET') {
                vista += "A Lleida s'envia un missatge específic si s'utilitza
```

```
                    el mètode GET";
74          } else {
75                  vista += "A Lleida tots els mètodes excepte GET retornen un
                        missatge genèric";
76          }
77
78          return vista;
79  }
80
81  function controladorTarragona(metode)  {
82          return "<h2>Tarragona</h2>Tampoc s'ha implementat cap acció especial
                per a <b>Tarragona</b>";
83  }
```

Una vegada rebuda la petició, s'invoca la funció `encaminar`, passant com a paràmetres el mètode, la resposta i la ruta de l'URL. Dintre de la funció, segons el valor de la ruta, s'invocarà una funció o una altra, que farà el paper de controlador. Aquests controladors s'encarreguen de generar una vista (en aquest cas es tracta simplement d'una cadena de text formatat com a HTML) que s'escriu a la resposta.

S'ha fet servir `controlador` com a nom de les funcions, perquè és el funcionament habitual d'aquest tipus d'aplicacions. L'encaminador invoca els mètodes corresponents segons el mètode i ruta rebuts, i retorna o envia la vista generada. En aquest cas, com que no existeix cap model, la vista es genera directament fent servir textos estàtics.

Habitualment els sistemes d'encaminament accepten paràmetres com a cadena de consulta (tant al cos del missatge com incrustats a l'URL) i com a fragments de l'URL. Per exemple en el *framework* Express es pot definir una ruta de la manera següent:

```
1  aplicacio.get('/provincies/:provincia/poblacions/:poblacio', function (peticio,
       resposta) {
2    resposta.send(peticio.params)
3  })
```

Aquest encaminament acceptaria un URL com `/provincies/Barcelona/poblacions/Martorell`, que generaria com a paràmetres l'objecte que podeu veure a continuació. És a dir, tant `:provincia` com `:població` són interpretats com a paràmetres d'encaminament:

```
1  {
2    provincia: Barcelona,
3    poblacio: Martorell
4  }
```

Una característica comuna a tots els sistemes d'encaminament és que també permeten utilitzar expressions regulars, de manera que es poden encaminar les rutes que comencin, acabin o continguin una combinació concreta de caràcters, per exemple.

Els sistemes d'encaminament més avançats també inclouen l'opció d'afegir `middleware`, funcions que són cridades amb la resposta i la petició i que permeten modificar-la abans (o després) de passar-les als controladors. D'aquesta manera es poden afegir sistemes d'autenticació, validacions de dades, adaptació de les dades, etc.

Com es pot comprovar, les implementacions pròpies dels sistemes d'encaminament acostumen a ser molt enrevessades i limitades en comparació amb els mòduls i *frameworks* especialitzats. Per aquesta raó només és recomanable utilitzar-les en els casos més simples.

Lectura i escriptura de fitxers

Les aplicacions desenvolupades amb Node.js, al contrari que les desenvolupades pel navegador, poden llegir i escriure fitxers al disc. Això permet utilitzar sistemes de persistència propis o guardar un registre d'esdeveniments.

Vegeu a continuació un exemple que permet afegir el nom i el cognom d'un usuari a un fitxer que més endavant serà recuperat per poder realitzar cerques i mostrar llistats. A la banda del client només es necessita el codi per realitzar l'enviament utilitzant el mètode `POST` i afegir les dades `nom` i `cognom`:

```
<html>
<body>
  <h1>Petició</h1>
  <label>Nom:<input id="nom" /></label>
  <label>Cognom:<input id="cognom" /></label>
  <button id="afegir_boto">Afegir nom</button>
  <h1>Resposta</h1>
  <div id="resposta"></div>
  <script>
    var afegirBoto = document.getElementById('afegir_boto');
    var resposta = document.getElementById('resposta');
    var metode = "POST"

    var nom = document.getElementById('nom');
    var cognom = document.getElementById('cognom');

    afegirBoto.addEventListener("click", function(e) {
      var url = 'http://127.0.0.1:8080';
      var dades = 'nom=' + encodeURI(nom.value);
        dades += '&cognom=' + encodeURI(cognom.value);

      var httpRequest = new XMLHttpRequest();
      httpRequest.open('POST', url, true);
      httpRequest.send(dades);

      httpRequest.onload = function (e) {
        resposta.innerHTML = httpRequest.responseText;
      }
    });
  </script>
</body>
</html>
```

A la banda del servidor, cal carregar el mòdul `fs` (sigles de *file system*, 'sistema de fitxers' en anglès). Per guardar la informació s'ha optat per utilitzar un fitxer de text pla en el qual el nom i el cognom estaran separats per una barra vertical `|`, mentre que cada parell de `nom` i `cognom` estarà separat per un asterisc `*`.

El mòdul 'FileSystem'

Podeu trobar tota la documentació sobre el mòdul `FileSystem` a l'enllaç següent: goo.gl/tpurw0.

Com que només s'accepten peticions de tipus `POST` no cal cercar la cadena de consulta a l'URL, sempre es trobarà al cos de la petició. Per aquest mateix motiu s'ha restringit l'accés a aquest servei només a les peticions `POST`, i s'estableix la limitació a la capçalera de la resposta.

```
var http = require('http');
var queryString = require('querystring');
var fs = require('fs');

var server = http.createServer();
server.on('request', function(peticio, resposta) {
  resposta.setHeader('Access-Control-Allow-Origin', '*');
  resposta.setHeader('Access-Control-Allow-Methods', 'POST');
  resposta.setHeader('Content-Type', 'text/plain;charset=utf-8');

  var cos = '';
  peticio.on('data', function(dades) {
    if (cos.length > 1e6) {
      peticio.connection.destroy();
    } else {
      cos += dades;
    }

  }).on('end', function() {
    var params = queryString.parse(cos);
    var nom = decodeURI(params.nom);
    var cognom = decodeURI(params.cognom);

    var data = Math.floor(Date.now() / 1000);
    fs.appendFile('noms.txt', nom + "|" + cognom + "*", function(error) {
      var missatge;

      if (error) {
        throw error; // No continuarà l'execució
      }

      resposta.statusCode = 200;
      missatge = 'Dades afegides al fitxer correctament: ' + nom + ' ' + cognom
          ;
      resposta.write(missatge);
      console.log(missatge);
      resposta.end();
    });
  });
});

server.listen(8080, '127.0.0.1');
```

Fixeu-vos que en lloc d'utilitzar `writeHead` per establir el codi d'estat, s'ha modificat directament la propietat, ja que aquest valor variarà segons si s'ha produït o no un error, i en el seu lloc s'ha establert el tipus de contingut utilitzant el mètode `setHeader`.

En aquest cas, no es considera l'opció d'acceptar altres mètodes, així que s'ha restringit l'accés al servei web de manera que només accepta peticions enviades mitjançant el mètode `POST`.

Com es pot apreciar, per afegir nou contingut al fitxer, s'utilitza el mètode `fs.appendFile` passant com a paràmetres el nom del fitxer (en aquest cas `noms.txt`), el contingut que s'ha d'afegir i una funció que es cridarà en acabar d'escriure les dades.

Aquesta funció pot rebre un error com a argument si s'ha produït en desar el fitxer (per exemple, per manca de permisos d'escriptura o si el disc fos ple). Per aquesta raó, es comprova si s'ha rebut un error i es genera el codi de la resposta i un missatge adient que es mostrarà per la consola.

Si proveu d'enviar dades a través del navegador, veureu que a la consola apareixeran els continguts de les variables `nom` i `cognom` que s'han afegit al client, i es crearà el fitxer amb el format especificat: parells de `nom` i `cognom` separats per una barra vertical, i aquests separats d'altres parells per un asterisc.

Cal tenir en compte que el mètode `appendFile` permet afegir dades a un fitxer existent, però en cas que aquest no existeixi en crearà un de nou. Un mètode similar del mòdul `FileSystem`, anomenat `writeFile`, permet escriure dades al disc, però en aquest cas sempre se sobreescriu el fitxer.

Aquesta última opció pot interessar si es treballa amb altre tipus de continguts. Per exemple, si en lloc de desar el fitxer com a text pla el guardem com a JSON, s'haurà de guardar l'objecte complet, que contindrà tota la informació. Això obligarà a llegir el fitxer en arrencar el servidor (si no, s'esborraria tota la informació anterior). Vegeu com es podria implementar en l'exemple següent:

```
var http = require('http');
var queryString = require('querystring');
var fs = require('fs');

var dadesAplicacio;

const NOM_FITXER = 'noms.json';

carregarDadesJSON(NOM_FITXER);

var server = http.createServer();

server.on('request', function(peticio, resposta) {
  resposta.setHeader('Access-Control-Allow-Origin', '*');
  resposta.setHeader('Access-Control-Allow-Methods', 'POST');
  resposta.setHeader('Content-Type', 'text/plain;charset=utf-8');

  var cos = '';
  peticio.on('data', function(dades) {
    if (cos.length > 1e6) {
      peticio.connection.destroy();
    } else {
      cos += dades;
    }

  }).on('end', function() {
    var params = queryString.parse(cos);
    var nom = decodeURI(params.nom);
    var cognom = decodeURI(params.cognom);

    dadesAplicacio[dadesAplicacio.length] = {
      nom: nom,
      cognom: cognom
    };
    desarDadesJSON(NOM_FITXER, dadesAplicacio);
    missatge = 'Afegit ' + nom + ' ' + cognom + ' al fitxer';
    resposta.write(missatge);
    console.log(missatge);
    resposta.end();
  });
});

function desarDadesJSON(fitxer, dades) {
  fs.writeFile(fitxer, JSON.stringify(dades), function(error) {
    if (error) {
      console('Error en desar el fitxer');
      throw err;
    } else {
      console.log('Dades desades');
```

```
    }
  });
}

function carregarDadesJSON(fitxer) {
  fs.readFile(fitxer, 'utf8', function(err, dades) {
    if (err) {
      console.log('No existeix el fitxer, creant nou conjunt de dades');
      dadesAplicacio = [];
    } else {
      console.log('Dades carregades:', dades);
      dadesAplicacio = JSON.parse(dades);
    }
  });
}

server.listen(8080, '127.0.0.1');
```

Fixeu-vos que s'ha definit una variable global per al mòdul anomenada `dadesAplicacio`. Aquesta variable contindrà l'*array* amb les dades. Cal destacar que mentre el servidor continuï funcionant, aquestes dades continuaran en memòria, és a dir, no es perden entre peticions. Per aquesta raó, només cal llegir el fitxer en iniciar el servidor, ja que cada vegada que es rep una nova petició s'afegeix un nou element a l'*array* amb aquesta informació i es desa al fitxer.

Com es pot apreciar, a diferència de JavaScript, Node.js sí que disposa de constants; en aquest cas s'ha definit la constant `NOM_FITXER` amb el nom del fitxer que contindrà les dades.

Cal destacar que la lectura i l'escriptura de dades utilitzant els mètodes `readFile`, `appendFile` i `writeFile` és asíncrona: això vol dir que mentre s'estan realitzant les operacions, el programa continua executant-se. Per aquesta raó no es pot retornar el contingut del fitxer en carregar-lo, ja que el valor de retorn és `undefined` en aquell moment. Per aquesta raó, el que s'ha fet és establir el valor de la variable global `dadesAplicacio` dintre de la funció que és cridada en finalitzar la lectura.

Com que s'ha utilitzat el format JSON només cal fer servir els mètodes `JSON.stringify` per convertir les dades en una cadena de text per desar i `JSON.parse` per convertir la cadena de text en un objecte de JavaScript. Aquest format simplifica molt la manipulació de dades, ja que no cal que implementeu els vostres propis analitzadors i permet treballar directament amb les dades llegides.

1.1.5 Cas pràctic: implementació d'un servei web REST

Una vegada ja se sap com fer l'encaminament dels URL, com es llegeixen i s'escriuen els fitxers i com es generen les respostes, implementar un servei web és relativament simple. Com a exemple s'implementarà un servei web per gestionar esdeveniments que inclourà l'opció de filtratge (podeu trobar el codi a l'enllaç següent: goo.gl/tZMFpu). Al llarg d'aquest exemple es practicaran les competències següents:

- Creació d'un servei web REST amb Node.js que permet realitzar el conjunt d'operacions CRUD (creació, lectura, actualització i eliminació).
  - Generació d'una resposta aplicant mecanismes CORS per permetre l'accés dels mètodes acceptats des de qualsevol domini.
  - Obtenció dels paràmetres de la petició sigui quin sigui el mètode emprat (`POST`, `GET`, `PUT` o `DELETE`).
  - Encaminament de la petició al controlador de recurs adequat segons l'URL.
  - Gestió de les operacions sobre el recurs.
  - Lectura i escriptura de fitxers en format JSON.
- Creació d'una aplicació d'una sola pàgina que permet veure el llistat d'esdeveniments, filtrar els resultats, afegir nous esdeveniments, actualitzar-los i eliminar-los.
  - Utilització de la biblioteca jQuery per simplificar el codi.
  - Utilització de formularis.
  - Utilització d'atributs propis (HTML5).
  - Enviament de peticions AJAX.
  - Manipulació del DOM per crear nous elements i modificar-ne la visibilitat mitjançant CSS.
  - Aplicació del patró mòdul per encapsular l'aplicació.

En anglès, les sigles CRUD es corresponen amb les inicials de *create, read, update, delete*.

Tot i que en aquest exemple només es farà servir un fitxer CSS i un fitxer JavaScript, és recomanable crear una carpeta per a cada tipus, ja que és una bona pràctica a l'hora de desenvolupar aplicacions més complexes.

El contingut del fitxer CSS és força simple; podeu crear un fitxer anomenat `principal.css` amb el codi que trobareu a continuació i desar-lo al directori `css`. A banda de donar estil a la pàgina, s'han afegit les classes `error`, `info`, `exit` i `alerta`, que es fan servir per donar diferents colors a les notificacions segons el seu tipus, i la classe `amaga`, per ocultar els continguts.

```
1  body {
2      width: 700px;
3      margin: 0 auto;
4  }
5
6  h1, h2 {
7      text-align: center;
8  }
9
10 label {
11     display: block;
12     font-weight: bold;
13 }
14
15 input, select, button, textarea {
16     display: block;
17     width: 100%;
18 }
19
20 table {
```

```
    border: 1px solid black;
    width: 100%;
    margin-top: 15px;
    margin-bottom: 15px;
}

table button {
    width: 33%;
    display: inline-block;
    overflow: hidden;
}

.columna-doble {
    display: inline-block;
    width: 49%;
}

.amaga {
    display: none;
}

.buit {
    text-align: center;
    font-style: italic;
}

#notificacions {
    border: 0;
    padding: 5px;
}

#notificacions.error {
    color: #D8000C;
    background-color: #FFBABA;
    border-color: #D8000C;
    border: 1px solid;
}

#notificacions.info {
    color: #00529B;
    background-color: #BDE5F8;
    border-color: #00529B;
    border: 1px solid;
}

#notificacions.exit {
    color: #4f8A10;
    background-color: #DFF2BF;
    border-color: #4f8A10;
    border: 1px solid;
}

#notificacions.alerta {
    color: #9F6000;
    background-color: #FEEFB3;
    border-color: #9F6000;
    border: 1px solid;
}
```

El codi HTML inclou la càrrega del full d'estil, la biblioteca jQuery i el fitxer amb el codi JavaScript. Cal destacar que es divideix en dues parts importants. Per una banda, la capçalera (etiqueta `header`), que inclou el títol de l'aplicació i l'àrea on es mostraran les notificacions i, per altra, la zona principal (etiqueta `main`), que mostrarà els diferents continguts.

```
<!DOCTYPE html>
<html lang="ca">
```

```html
<head>
    <meta charset="UTF-8">
    <title>Gestor d'esdeveniments</title>
    <link rel="stylesheet" type="text/css" href="css/principal.css">
</head>
<body>

<header>
    <h1>Gestor d'esdeveniments</h1>
    <div id="notificacions"></div>
</header>

<main>
    <div id="paginaLlistat">
        <h2>Llistat d'esdeveniments</h2>

        <input class="columna-doble" type="text"
               placeholder="Introdueix una paraula o deixa-ho buit per eliminar el filtre"
               name="filtre"/>
        <button class="columna-doble" data-accio="filtrar">Filtrar</button>

        <table>
            <thead>
            <tr>
                <th>Nom</th>
                <th>Data</th>
                <th>Organitzador</th>
                <th>Accions</th>
            </tr>
            </thead>
            <tbody>
            <tr>
                <td colspan="4" class="buit">No s'ha trobat cap esdeveniment</td>
            </tr>
            </tbody>
        </table>
        <button data-accio="mostrar-alta">Afegir nou esdeveniment</button>
    </div>

    <div id="paginaAlta" class="amaga">
        <h2>Alta d'esdeveniment</h2>
        <form id="formulariAlta">
            <label>Nom</label>
            <input type="text" name="nom" placeholder="Nom de l'esdeveniment..." required/>

            <label>Descripció</label>
            <textarea name="descripcio" placeholder="Descripció de l'esdeveniment..."></textarea>

            <label>Data</label>
            <input type="date" name="data"/>

            <label>Organitzador</label>
            <select name="organitzador">
                <option value="Ajuntament de Barcelona" selected>Ajuntament de Barcelona</option>
                <option value="Associació Cultural">Associació Cultural</option>
                <option value="Associació de Botiguers">Associació de Botiguers</option>
                <option value="Organitzador privat">Organitzador privat</option>
            </select>
            <button class="columna-doble" data-accio="alta">Alta</button>
            <button class="columna-doble" data-accio="mostrar-llistat">Tornar al llistat</button>
        </form>
```

```
64      </div>
65
66      <div id="paginaActualitzar" class="amaga">
67          <h2>Modificació d'esdeveniment</h2>
68          <form id="formulariActualitzar">
69              <label>Nom</label>
70              <input type="text" name="nom" placeholder="Nom de l'esdeveniment...
                    " required/>
71
72              <label>Descripció</label>
73              <textarea name="descripcio" placeholder="Descripció de l'
                    esdeveniment..."></textarea>
74
75              <label>Data</label>
76              <input type="date" name="data"/>
77
78              <label>Organitzador</label>
79              <select name="organitzador">
80                  <option selected>Ajuntament de Barcelona</option>
81                  <option>Associació Cultural</option>
82                  <option>Associació de Botiguers</option>
83                  <option>Organitzador privat</option>
84              </select>
85              <button class="columna-doble" data-accio="actualitzar">Actualitzar<
                    /button>
86              <button class="columna-doble" data-accio="mostrar-llistat">Tornar
                    al llistat</button>
87          </form>
88      </div>
89
90      <div id="paginaDetall" class="amaga">
91          <h2>Detall d'esdeveniment</h2>
92          <div>
93              <label>Nom</label>
94              <span id="detallNom">xx</span>
95              <label>Descripció</label>
96              <span id="detallDescripcio">xx</span>
97              <label>Data</label>
98              <span id="detallData">xx</span>
99              <label>Organitzador</label>
100             <span id="detallOrganitzador">xx</span>
101         </div>
102         <button class="columna-doble" data-accio="mostrar-actualitzar">
                Actualitzar</button>
103         <button class="columna-doble" data-accio="mostrar-llistat">Tornar al
                llistat</button>
104     </div>
105 </main>
106
107 <script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
108 <script src="js/gestor-esdeveniments.js"></script>
109 </body>
110 </html>
```

Fixeu-vos que la zona principal conté quatre contenidors `div`: `paginaLlistat`, `paginaAlta`, `paginaActualitzar`, `paginaDetall`. Aquests contenidors representen les diferents pantalles de l'aplicació i, inicialment, la classe `amaga` s'aplica a totes excepte a `paginaLlistat`. Mitjançant JavaScript es fa la transició entre pantalles afegint aquesta classe a la pàgina anterior i eliminant-la de la pàgina nova.

Cal destacar que als botons s'ha aplicat l'atribut personalitzat `data-accio` i en alguns casos, mitjançant JavaScript, l'atribut `data-id`. Això permet processar tots els botons amb el mateix detector d'*events* i segons el valor que tinguin aquests atributs es realitzarà una acció o una altra. Per exemple, en fer clic sobre el botó

Detall es processarà l'acció `mostrar-detall` amb l'identificador corresponent a l'esdeveniment (indicat per `data-id`).

Com es pot apreciar, la taula és buida en el codi HTML i les files es generen mitjançant JavaScript una vegada es rep el llistat d'esdeveniments del servidor.

Un detall a tenir en compte és que el tipus de dades `date` no forma part de l'especificació d'HTML5, sinó que es tracta d'una implementació pròpia de Google Chrome i, per tant, no és disponible en altres navegadors (com Mozilla Firefox). En aquests casos es tractarà com un camp de text normal.

El codi del client el podeu guardar en un fitxer anomenat `gestor-esdeveniments` dintre del directori `js`. Com podeu veure a continuació, s'ha aplicat el patró mòdul per encapsular-lo, de manera que tota l'aplicació es troba a la variable `GESTOR_ESDEVENIMENTS` i només és accessible externament el mètode `iniciarAplicació`. D'aquesta manera es pot assegurar que no interferirà amb altres aplicacións ni serà manipulada externament.

S'ha de tenir en compte que a `GESTOR_ESDEVENIMENTS` no s'assigna una funció, sinó que **se n'assigna el retorn**, ja que és una funció autoinvocada i, per consegüent, el contingut de `GESTOR_ESDEVENIMENTS` és un objecte amb un únic mètode (`iniciarAplicació`) que té accés a l'aplicació gràcies a la clausura.

```
1  var GESTOR_ESDEVENIMENTS = (function () {
2
3      var paginaMostradaActualment;
4      var llistatEsdeveniments;
5
6      function enviarPeticio(metode, url, params) {
7          $.ajax({
8              url: url,
9              data: params,
10             dataType: 'json',
11             type: metode,
12
13             error: function () {
14                 accioMostrarNotificacio('error', 'S\'ha produït un error en fer
                       la petició al servidor');
15             },
16
17             success: function (dades) {
18                 processarResposta(dades);
19             }
20         });
21     }
22
23     function processarAccio(accio, dades) {
24         switch (accio) {
25             case 'establir_llistat':
26                 accioEstablirLlistat(dades.esdeveniments);
27                 break;
28
29             case 'mostrar_notificacio':
30                 accioMostrarNotificacio(dades.notificacio.tipus, dades.
                       notificacio.missatge);
31                 break;
32
33             case 'mostrar_pagina':
34                 accioMostrarPagina(dades.pagina);
35                 break;
36
37             case 'mostrar_actualitzar':
```

```
                establirEsdevenimentActualitzar(llistatEsdeveniments[dades.id],
                     'actualitzar');
                accioMostrarPagina('paginaActualitzar');
                break;

            case 'mostrar_detall':
                establirEsdevenimentDetall(llistatEsdeveniments[dades.id], '
                    detall');
                accioMostrarPagina('paginaDetall');
                break;

            case 'mostrar_llistat':
                netejarFormularis();
                accioMostrarPagina('paginaLlistat');
                break;

            case 'mostrar_alta':
                netejarFormularis();
                accioMostrarPagina('paginaAlta');
                break;

            case 'eliminar':
                enviarPeticio('DELETE', 'http://127.0.0.1:8080/esdeveniments/'
                    + dades.id);
                break;

            case 'actualitzar':
                dades.esdeveniment = obtenirEsdevenimentDelFormulari('
                    formulariActualitzar');
                dades.esdeveniment.id = dades.id;
                enviarPeticio('PUT', 'http://127.0.0.1:8080/esdeveniments/' +
                    dades.id, dades.esdeveniment);
                break;

            case 'alta':
                dades.esdeveniment = obtenirEsdevenimentDelFormulari('
                    formulariAlta');
                enviarPeticio('POST', 'http://127.0.0.1:8080/esdeveniments',
                    dades.esdeveniment);
                break;

            case 'filtrar':
                accioFiltrar();
                break;

            default:
                console.error('Acció desconeguda: ', accio);
        }
    }

    function processarResposta(dades) {

        for (var i = 0; i < dades.accions.length; i++) {
            processarAccio(dades.accions[i].accio, dades.accions[i])
        }
    }

    function processarBoto(e) {
        e.preventDefault();

        var $boto = $(this);
        var accio = $boto.attr('data_accio');
        var id = $boto.attr('data_id');
        var dades = {
            $boto: $boto,
            id: id
        };

        processarAccio(accio, dades);
    }
```

```
101
102     function accioMostrarNotificacio(tipus, missatge) {
103         var $notificacions = $('#notificacions');
104         $notificacions.removeClass();
105         $notificacions.addClass(tipus);
106         $notificacions.html(missatge);
107     }
108
109     function accioMostrarPagina(novaPagina) {
110         $('#' + paginaMostradaActualment).addClass('amaga');
111         paginaMostradaActualment = novaPagina;
112         $('#' + paginaMostradaActualment).removeClass('amaga');
113     }
114
115     function accioEstablirLlistat(dades) {
116         llistatEsdeveniments = dades;
117         omplirTaulaLlistat(dades);
118         accioMostrarPagina('paginaLlistat');
119     }
120
121     function accioFiltrar() {
122         var $filtre = $('input[name="filtre"]');
123         var terme = $filtre.val();
124
125         if (terme.indexOf(' ') !== -1) {
126             accioMostrarNotificacio('error', 'Només es pot introduir una
                    paraula per filtrar');
127         } else if (!terme) {
128             accioMostrarNotificacio('info', 'Desactivat el filtre');
129             omplirTaulaLlistat(llistatEsdeveniments);
130         } else {
131             filtrarEsdeveniments(terme);
132             $filtre.val('');
133         }
134     }
135
136     function omplirTaulaLlistat(dades) {
137         var $cosTaula = $('tbody');
138
139         // Es neteja el cos del llistat
140         $cosTaula.empty();
141
142         // Es recorren les dades
143         for (var i in dades) {
144
145             // Es genera la fila i les columnes
146             var $fila = $('<tr>');
147             var $col1 = $('<td>');
148             var $col2 = $('<td>');
149             var $col3 = $('<td>');
150             var $col4 = $('<td>');
151
152
153             // S'afegeix el contingut a cada columna a partir de la informació
                    del llistat de dades
154             $col1.html(dades[i].nom);
155             $col2.html(dades[i].data);
156             $col3.html(dades[i].organitzador);
157
158             var $botoDetall = $('<button>Detall</button>');
159             $botoDetall.attr('data-id', dades[i].id);
160             $botoDetall.attr('data-accio', 'mostrar-detall');
161
162             var $botoActualitzar = $('<button>Actualitzar</button>');
163             $botoActualitzar.attr('data-id', dades[i].id);
164             $botoActualitzar.attr('data-accio', 'mostrar-actualitzar');
165
166             var $botoEliminar = $('<button>Eliminar</button>');
167             $botoEliminar.attr('data-id', dades[i].id);
168             $botoEliminar.attr('data-accio', 'eliminar');
```

```
169
170             $col4.append($botoDetall);
171             $col4.append($botoActualitzar);
172             $col4.append($botoEliminar);
173
174             // S'afegeixen les columnes a la fila
175             $fila.append($col1);
176             $fila.append($col2);
177             $fila.append($col3);
178             $fila.append($col4);
179
180             // S'afegeix la fila al cos de la taula
181             $cosTaula.append($fila);
182         }
183
184         // S'afegeix un detector d'events a tots els botons
185         $('table button').on('click', processarBoto);
186     }
187
188     function filtrarEsdeveniments(terme) {
189         var llistatFiltrat = [];
190
191         for (var i in llistatEsdeveniments) {
192             if (llistatEsdeveniments[i].nom.indexOf(terme) !== -1
193                 || llistatEsdeveniments[i].descripcio.indexOf(terme) !== -1
194                 || llistatEsdeveniments[i].organitzador.indexOf(terme) !== -1
195             ) {
196                 llistatFiltrat.push(llistatEsdeveniments[i]);
197             }
198         }
199
200         omplirTaulaLlistat(llistatFiltrat);
201
202         if (llistatFiltrat.length === 0) {
203             accioMostrarNotificacio('alerta', 'No s\'ha trobat cap resultat');
204         } else {
205             accioMostrarNotificacio('exit', 'Llistat filtrat pel terme "' +
                    terme + '".');
206         }
207     }
208
209     function obtenirEsdevenimentDelFormulari(idFormulari) {
210         var $form = $('form#' + idFormulari);
211
212         return {
213             nom: $form.find('input[name="nom"]').val(),
214             descripcio: $form.find('textarea').val(),
215             data: $form.find('input[name="data"]').val(),
216             organitzador: $form.find('select').val()
217         }
218     }
219
220     function establirEsdevenimentActualitzar(esdeveniment) {
221         var $form = $('form');
222
223         $form.find('input[name="nom"]').val(esdeveniment.nom);
224         $form.find('textarea').val(esdeveniment.descripcio);
225         $form.find('input[name="data"]').val(esdeveniment.data);
226         $form.find('select').val(esdeveniment.organitzador);
227         $form.find('[data-accio="actualitzar"]').attr('data-id', esdeveniment.
                id);
228     }
229
230     function establirEsdevenimentDetall(esdeveniment) {
231         $('#detallNom').html(esdeveniment.nom);
232         $('#detallDescripcio').html(esdeveniment.descripcio);
233         $('#detallData').html(esdeveniment.data);
234         $('#detallOrganitzador').html(esdeveniment.organitzador);
235
```

```
236         $('[data-accio="mostrar-actualitzar"]').attr('data-id', esdeveniment.id
                );
237     }
238
239     function netejarFormularis() {
240         $('form').trigger('reset');
241     }
242
243     return {
244         iniciarAplicacio: function () {
245             $('button').on('click', processarBoto);
246             enviarPeticio('GET', 'http://127.0.0.1:8080/esdeveniments');
247         }
248     }
249
250 })();
251
252 $(document).ready(function () {
253     GESTOR_ESDEVENIMENTS.iniciarAplicacio();
254 });
```

Fixeu-vos que s'ha centralitzat el processament d'accions en la funció `processarAccio`, que rep el nom de l'acció que cal portar a terme i un objecte amb les dades que s'han de processar. Aquesta funció és cridada tant pel processament de les respostes que arriben del servidor com pel processament dels botons, de manera que aquestes funcions només s'han d'encarregar d'enviar les dades correctes per realitzar el processament.

Les accions en conjunt es poden dividir en dos grups: les responsables d'enviar peticions al servidor i les responsables de modificar-ne la vista (la representació de les dades a la pantalla). Cal tenir en compte que, per simplificar, no s'ha creat una acció per a cada cas, ja que moltes només requereixen un parell de línies de codi. En projectes més complexos, però, és una bona pràctica (o fins i tot utilitzar diferents objectes).

La funció `processarResposta` és l'encarregada de processar les respostes a les peticions AJAX. Aquestes respostes han de ser en format JSON i han de contenir un *array* amb el nom `accions`, que contindrà una o més accions per processar. D'aquesta manera una resposta pot contenir múltiples accions, com poden ser afegir una notificació, actualitzar el llistat i mostrar una pàgina diferent, i són portades a terme per la funció `processarAcció`.

Per simplificar l'enviament de peticions AJAX s'ha creat una funció anomenada `enviarPeticio`, que és l'encarregada de crear-la, ja que el codi d'aquestes peticions és idèntic en tots els casos i només canvia el mètode d'enviament, l'URL i els paràmetres. Si es produeix cap error, es mostrarà una notificació i en cas d'èxit es processarà la resposta invocant la funció `processarResposta`.

Per altra banda, la funció `processarBoto` és invocada quan es fa clic sobre qualsevol dels botons, tant els inclosos en el codi HTML com els creats dinàmicament en generar la taula. Aquesta funció s'encarrega d'obtenir la informació necessària i passar-la a la funció `processarAcció`.

Les funcions `accioMostrarNotificacio` i `accioMostrarPagina` s'encarreguen de modificar la vista. La primera reemplaça l'estil i contingut de la notificació activa amb la nova, de manera que serà de color verd en cas d'èxit, blau en cas de ser una notificació informativa, groc en cas d'alerta i vermell en

cas d'error. La segona amaga la pàgina anterior i en mostra la nova, afegint i eliminant la classe CSS `amaga`, respectivament.

Les funcions `accioFiltrar` i `filtrarEsdeveniments` són les encarregades de filtrar els resultats. La primera comprova que només s'hagi utilitzat una paraula i discrimina entre aplicar el filtratge o eliminar el filtre (omplint la taula amb tots els esdeveniments). La segona és l'encarregada de realitzar l'acció, recorrent tots els esdeveniments i creant una llista amb els que continguin el terme a les propietats `nom`, `descripcio` o `organitzador`.

Per gestionar els formularis i recopilar-ne les dades es fan servir les funcions `obtenirEsdevenimentDelFormulari`, `establirEsdevenimentActualitzar` i `netejarFormularis`. El primer es fa servir tant en actualitzar un esdeveniment com en donar-ne d'alta un de nou i serveix per obtenir un esdeveniment a partir de les dades del formulari. El segon es fa servir per establir les dades de l'esdeveniment per actualitzar al formulari d'actualització. Finalment, el tercer és una funció auxiliar utilitzada per netejar tota la informació de tots els formularis de la pàgina.

La funció `omplirTaulaLlistat` és l'encarregada de generar la taula amb la informació resumida dels esdeveniments. El primer pas és buidar-la utilitzant el mètode `empty` de jQuery; seguidament es recorren les dades (que corresponen a un objecte contenidor d'esdeveniments), es crea una fila amb 4 columnes i s'afegeix la informació de l'esdeveniment a les 3 primeres i els botons d'acció a l'última.

Una vegada s'ha generat la taula, s'afegeix un detector d'*events* a tots els botons de la taula per invocar el mètode `processarBoto` en detectar-se l'*event* `click`.

Cal tenir en compte un detall important. Fixeu-vos que el llistat d'esdeveniments i les dades no és un *array*, sinó un objecte de JavaScript. Aquesta decisió s'ha pres per simplificar la gestió d'esdeveniments, perquè una vegada s'elimina un element de l'*array*, l'índex dels elements ja no es correspon amb el seu identificador (que s'ha decidit fer seqüencial, començant pel 0). D'aquesta manera, l'identificador de l'esdeveniment s'utilitza com a nom de la propietat a l'objecte i l'eliminació no l'afecta perquè es guarda sempre la referència del pròxim identificador a assignat (gestionat pel servidor).

És a dir, l'estructura de dades que es guarda al servidor és similar a la següent (en format JSON):

```
{
  "proximIdentificador":9,
  "llistatEsdeveniments":{
    "5":{
      "nom":"Primer esdeveniment",
      "descripcio":"Aquesta es la descripció del primer esdeveniment.",
      "data":"2016-12-15",
      "organitzador":"Associació de Botiguers",
      "id":5
    },
    "8":{
      "nom":"Segon esdeveniment",
      "descripcio":"Aquest esdeveniment es realitzarà en una data posterior al
          primer",
      "data":"2017-12-20",
      "organitzador":"Ajuntament de Barcelona",
      "id":8}
```

```
  }
}
```

A continuació podeu trobar el codi del servidor, que fa servir els mòduls de Node.js `http` per crear el servidor, `url` per tractar els URL, `querystring` per gestionar els paràmetres i `fs` per treballar amb el sistema de fitxers.

```
var http = require('http');
var url = require('url');
var queryString = require('querystring');
var fs = require('fs');

const NOM_FITXER = 'esdeveniments.json';
const INDEX_RECURS = 0;
const INDEX_IDENTIFICADOR = 1;

var server = http.createServer();
var dadesAplicacio;

inicialitzar();

function inicialitzar() {
    carregarDadesJSON(NOM_FITXER);

    server.on('request', function (peticio, resposta) {
        resposta.setHeader('Access-Control-Allow-Origin', '*');
        resposta.setHeader('Access-Control-Allow-Methods', 'GET, PUT, POST,
            DELETE');
        resposta.writeHead(200, {'Content-Type': 'application/json;charset=utf
            -8'});

        var cos = '';
        peticio.on('data', function (dades) {
            cos += dades;

        }).on('end', function () {
            var params;
            var dades;

            if (peticio.method === 'GET') {
                dades = url.parse(peticio.url).query;
            } else {
                dades = cos;
            }
            params = queryString.parse(dades);

            var urlEncaminament = (url.parse(peticio.url).path).substr(1);
            encaminar(urlEncaminament, peticio.method, resposta, params);
            resposta.end();

        });
    });

    server.listen(8080, "127.0.0.1");
}

function encaminar(url, metode, resposta, params) {
    var contingutResposta = {};
    var fragmentsUrl = url.split('/');

    switch (fragmentsUrl[INDEX_RECURS]) {
        case 'esdeveniments':
            contingutResposta.accions = controladorEsdeveniments(metode,
                fragmentsUrl[INDEX_IDENTIFICADOR], params);
            break;

        default:
            contingutResposta.accions = [];
            afegirNotificacio(contingutResposta.accions, 'error', 'No es
                reconeix el recurs ' + fragmentsUrl[INDEX_RECURS]);
```

```
60      }
61
62      resposta.write(JSON.stringify(contingutResposta));
63  }
64
65  function controladorEsdeveniments(metode, id, params) {
66      var accions = [];
67
68      switch (metode) {
69          case 'GET':
70              afegirNotificacio(accions, 'info', 'Carregat llistat d\'
                    esdeveniments.');f
71              afegirLlistat(accions);
72              break;
73
74          case 'POST':
75              params.id = dadesAplicacio.proximIdentificador;
76              dadesAplicacio.llistatEsdeveniments[dadesAplicacio.
                    proximIdentificador++] = params;
77
78              desarDadesJSON('esdeveniments.json', dadesAplicacio);
79              afegirNotificacio(accions, 'exit', 'Nou esdeveniment afegit.');
80              afegirLlistat(accions);
81              break;
82
83          case 'DELETE':
84              delete(dadesAplicacio.llistatEsdeveniments[id]);
85              desarDadesJSON('esdeveniments.json', dadesAplicacio);
86              afegirNotificacio(accions, 'info', 'Esdeveniment eliminat.');
87              afegirLlistat(accions);
88              break;
89
90          case 'PUT':
91              dadesAplicacio.llistatEsdeveniments[id] = params;
92              desarDadesJSON('esdeveniments.json', dadesAplicacio);
93              afegirNotificacio(accions, 'exit', 'Esdeveniment actualitzat.');
94              afegirLlistat(accions);
95              break;
96
97          default:
98              afegirNotificacio(accions, 'error', 'No es reconeix el mètode ' +
                    metode);
99      }
100
101     return accions;
102 }
103
104 function afegirNotificacio(contingutResposta, tipus, missatge) {
105     contingutResposta.push({
106         accio: 'mostrar_notificacio',
107         notificacio: {
108             tipus: tipus,
109             missatge: missatge
110         }
111     })
112 }
113
114 function afegirLlistat(contingutResposta) {
115     contingutResposta.push({
116         accio: 'establir_llistat',
117         esdeveniments: dadesAplicacio.llistatEsdeveniments
118     });
119 }
120
121
122 function desarDadesJSON(fitxer, dades) {
123     fs.writeFile(fitxer, JSON.stringify(dades), function (error) {
124         if (error) {
125             console.error('Error en desar el fitxer');
126             throw err;
```

```
127         }
128     });
129 }
130 
131 function carregarDadesJSON(fitxer) {
132     fs.readFile(fitxer, 'utf8', function (err, dades) {
133         if (err) {
134             console.log('No existeix el fitxer de dades, creant nou conjunt de
                    dades...');
135             dadesAplicacio = {
136                 proximIdentificador: 0,
137                 llistatEsdeveniments: {}
138             };
139         } else {
140             dadesAplicacio = JSON.parse(dades);
141         }
142     });
143 }
```

Cal recordar que a Node.js, al contrari que a JavaScript, el concepte de constant sí que existeix i, per consegüent, s'ha utilitzat per definir el nom del fitxer de dades i la correspondència dels índexs en dividir l'URL (el primer correspondrà al tipus de recurs i el segon, si n'hi ha, a l'identificador).

Una vegada s'han importat els mòduls necessaris i s'han definit les constants i variables globals del mòdul, s'invoca la funció `inicialitzar`. Aquesta funció és l'encarregada de carregar les dades i afegir el detector d'*events*, que es dispararà quan el servidor rebi una petició (*event* `request`).

Quan es detecta la petició, la funció executada és l'encarregada d'analitzar l'URL i els paràmetres, i de passar-los a l'encaminador, definit com la funció `encaminar`, que escriurà una resposta o una altra segons les dades de la petició.

Dintre d'aquesta funció, l'URL es divideix en fragments. El primer fragment (amb índex 0) correspon al nom del recurs i el segon fragment (si n'hi ha) correspon a l'identificador. En aquest exemple només es treballa amb un tipus de recurs i, per tant, només hi ha un cas vàlid.

Per generar la resposta, s'ha decidit utilitzar un objecte amb una propietat anomenada *accions*, que contindrà un *array* d'accions per realitzar en el client. Aquestes accions són generades i retornades pel controlador, anomenat `controladorEsdeveniments`, i seran determinades a partir dels paràmetres passats (mètode, identificador i paràmetres).

La funció `controladorEsdeveniments` sempre retornarà un *array* d'accions que, com a mínim, contindrà una notificació d'error. A banda de la resposta, segons el mètode emprat es realitzaran diferents accions sobre les dades:

- **GET**: cap acció al servidor, es retorna el llistat complet i una notificació.
- **POST**: s'afegeix l'identificador a les dades rebudes com a paràmetre i s'afegeix una nova notificació (incrementant posteriorment el valor de `proximIdentificador`). Seguidament es desen les dades en format JSON, s'afegeix una notificació d'èxit i es retorna el llistat.
- **DELETE**: s'elimina l'esdeveniment del conjunt de dades amb l'identificador passat com a paràmetre, a continuació es desen les dades, s'afegeix una

notificació informativa i es retorna el nou llistat (que no inclou l'element eliminat).

- **PUT**: actualitza el conjunt de dades amb la informació de l'esdeveniment, desa les dades i finalment afegeix una notificació.

Fixeu-vos que s'ha fet servir `PUT` per a l'actualització, ja que les dades són completament reemplaçades. En canvi, si l'actualització fos només de determinats valors, llavors seria més apropiat utilitzar el mètode `PATCH`.

Cal destacar que els mètodes `afegirNotificacio` i `afegirLlistat` no retornen res, manipulen directament l'*array* passat com a argument: com que es tracta d'un *array*, es passa per referència i no pas per valor. És a dir, qualsevol canvi realitzat a l'*array* dintre de les funcions `afegirNotificacio` i `afegirLlistat` afecta l'*array* original.

Les funcions `afegirNotificacio` i `afegirLlistat` s'encarreguen de generar correctament l'acció pertinent i d'afegir-la al contingut de la resposta. La primera funció afegeix el tipus de missatge (que afectarà l'estil CSS a aplicar) i el missatge a mostrar, mentre que la segona afegeix el llistat complet d'esdeveniments.

Per acabar, hi ha les funcions `desarDadesJSON` i `carregarDadesJSON`, que són les responsables de carregar el fitxer de dades i desar-lo. En cas de detectar un error en carregar les dades, es genera una nova estructura de dades que serà desada quan es rebi una nova alta d'esdeveniments. Per exemple, en cas de produir-se un error la primera vegada que s'inicia el servidor, es retornarà una estructura de dades buida, que serà desada quan es rebi una petició `POST`; en cas contrari, retornarà l'estructura de dades carregada.

Recordeu que les invocacions a `fs.writeFile` i `fse.readFile` són asíncrones i, per tant, no es bloqueja l'execució mentre es carreguen o es desen les dades (tot i que si és necessari, hi ha una versió síncrona de totes dues).

1.1.6 Cas pràctic: creació d'una aplicació de xat amb sòcols

Aquest exemple mostra com es pot crear una aplicació de xat utilitzant els *frameworks* Express i Socket.io de Node.js (està basat en un dels exemples de la documentació oficial de Socket.io). Podeu trobar el codi complet a l'enllaç següent: goo.gl/jWh4Sb.

El *framework* Express és utilitzat per simplificar les operacions habituals d'un servidor web, com és l'encaminament i l'enviament de fitxers; mentre que Socket.io s'encarrega de la gestió dels sòcols de connexió tant al servidor com al client.

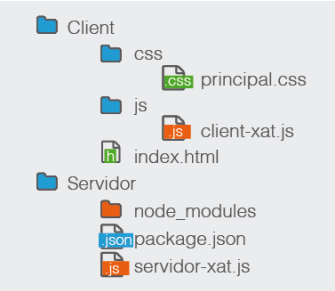
La biblioteca Socket.io utilitza internament l'API WebSockets a la banda del client per realitzar aquestes connexions, facilitant la utilització d'un sistema d'*events* propis que són enviats i gestionats pels sòcols.

En aquest exemple es tractaran els punts següents:

- Instal·lació de *frameworks* de tercers amb Node.js (Express i Socket.io).
- Utilització de *frameworks* per simplificar l'encaminament (Express).
- Encaminament de la petició de fitxers (Express).
- Creació d'un servidor de connexions de sòcols (Socket.io).
- Enviament i detecció d'*events* mitjançant sòcols (Socket.io).

Per reproduir aquest exemple en el vostre entorn, cal crear l'estructura de directoris que es pot veure a la figura 1.8, ja que s'utilitzarà Express per servir els fitxers.

FIGURA 1.8. Estructura de fitxers del xat amb Socket.io



El directori `node_modules` es crearà automàticament en instal·lar els mòduls necessaris utilitzant el gestor de paquets npm.

El primer pas és crear un fitxer anomenat `package.json` amb el següent contingut dintre del directori `Servidor`, que contindrà el servidor de xat:

```
{
  "name": "exemple-web-sockets",
  "version": "0.0.1",
  "description": "Exemple de Xat amb Express i Socket.io",
  "dependencies": {}
}
```

Seguidament, per instal·lar el *framework* Express utilitzant el gestor de paquets npm executeu la següent instrucció:

```
npm install --save express
```

L'opció `-save` fa que la dependència s'afegeixi automàticament al fitxer `package.json` de l'aplicació.

La integració amb el *framework* `Socket.IO` s'ha de fer tant en el client (afegint la biblioteca `socket.io-client`) com en el servidor (carregant el mòdul `socket.io`).

Per afegir el mòdul al servidor s'ha d'executar la comanda següent a la terminal:

```
npm install --save socket.io
```

Una vegada instal·lats els *frameworks* Express i Socket.io, es pot crear dintre de la mateixa carpeta `Servidor` el fitxer `servidor-xat.js`, que contindrà la implementació del servidor de xat.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);
var path = require('path');

// Encaminament
app.get('/', function(peticio, resposta){
    resposta.sendFile(path.resolve(__dirname + '/../Client/index.html'));
});

app.get('/css/:fitxer', function(peticio, resposta){
    resposta.sendFile(path.resolve(__dirname + '/../Client/css/'+ peticio.
        params.fitxer));
});

app.get('/js/:fitxer', function(peticio, resposta){
    resposta.sendFile(path.resolve(__dirname + '/../Client/js/' + peticio.
        params.fitxer));
});

// Gestió de les connexions
io.on('connection', function(socol){
    socol.broadcast.emit('missatge xat', 'Un nou usuari s\'ha connectat');

    console.log('Un usuari connectat');

    socol.on('disconnect', function(){
        socol.broadcast.emit('missatge xat', 'Un usuari s\'ha desconnectat');
        console.log('usuari desconnectat');
    });

    socol.on('missatge xat', function(msg){
        console.log('missatge: ' + msg);
        io.emit('missatge xat', msg);

    });
});

// S'inicia l'escolta del servidor pel port 3000
http.listen(3000, function(){
    console.log('Escoltant a *:3000');
});
```

Primerament, cal importar els mòduls necessaris per a l'aplicació: `express`, `http`, `socket.io` i `path`. Fixeu-vos que en el cas d'`express` i `io` no es carrega el mòdul, sinó que s'invoca com una funció, sense passar cap paràmetre en el cas d'`express` i passant el servidor `http` en el cas de `socket.io`.

Com que `express` no permet utilitzar directament els dos punts per accedir a un directori anterior perquè el considera codi maliciós, per construir la ruta s'ha d'utilitzar el mòdul `path`.

El mòdul `Routing` conté la implementació de l'encaminament d'Express adaptada per funcionar amb el mòdul `Http`.

A continuació es pot veure com funciona l'encaminament d'`Express`, molt més simple que les implementacions manuals. A partir del mètode `get` es passen com a arguments una ruta i la funció que serà cridada en detectar-la.

Es pot trobar més informació sobre els encaminaments d'Express a l'enllaç següent: goo.gl/pgXQuq.

Aquests encaminaments també faciliten treballar amb paràmetres de ruta, prefixats amb el símbol dels dos punts (`:`), als quals es pot accedir com a propietats a partir de l'objecte `peticio.params`. És a dir, quan es demana el fitxer corres-

ponent a la ruta `/js/client-xat.js` el valor de `peticio.params.fitxer` és `client-xat.js`.

Les respostes d'Express ofereixen la possibilitat d'enviar un fitxer directament com a resposta utilitzant el mètode `sendFile`, d'aquesta manera se serveix el fitxer HTML, el fitxer JavaScript i el fitxer CSS corresponent.

Com es pot apreciar, Express reemplaça completament la funcionalitat del mòdul `http`, i fa que la implementació sigui molt més simple i entenedora.

Seguidament es pot veure la implementació de la gestió de connexions. Primerament s'afegeix un detector per a l'*event* `connection` per detectar quan s'ha establert una nova connexió mitjançant l'*event* `connection` del *framework* Socket.io.

Una vegada s'ha rebut una connexió es crida una funció (que rep com a paràmetre el sòcol) perquè realitzi les acions següents:

- S'envia el missatge `Un nou usuari s'ha connectat` a tots els usuaris, excepte al que ha connectat, mitjançant el mètode `socol.broadcast.emit`.
- S'afegeix al sòcol un detector per a l'event `disconnect`, que avisarà els altres usuaris quan un usuari s'hagi desconnectat.
- S'afegeix un detector per a l'*event* `missatge xat` (un *event* propi) que envia a tots els usuaris connectats al servidor (emissor inclòs) el missatge rebut.
- Per facilitar la depuració s'ha afegit un missatge a la consola que permet determinar el moment en què un usuari s'uneix o abandona la sala de xat.

Cal destacar que l'enviament de missatges es tracta, en realitat, d'una emissió d'*events* que farà que l'*event* `missatge xat` es dispari al client amb el missatge enviat com a dades.

Fixeu-vos en les diferències dels dos tipus d'enviament mostrat. Per una banda, hi ha missatges globals que són rebuts per tots els usuaris, i emesos pel servidor (`io.emit`) i, per altra, hi ha els *events* que no ha de rebre l'emissor (com la connexió i la desconnexió) i que són emesos a partir del sòcol (socol.broadcast.emit).

Port de connexió dels sòcols

El port que s'utilitza per connectar amb l'aplicació mitjançant sòcols és el mateix port pel qual s'escolten les peticions HTTP. Així doncs, si es vol canviar aquest port, s'haurà de canviar al servidor modificant la invocació al mètode `http.listen`.

Cal tenir en compte que, per defecte, el client de Socket.io farà servir el mateix port utilitzat per obrir la pàgina, de manera que si la pàgina s'ha obert al port 3000, la connexió dels sòcols es realitzarà a través d'aquest port.

El codi HTML del client ha d'anar en un fitxer anomenat `index.html` dins del directori `Client` amb el contingut següent:

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Client Xat amb Socket.io</title>
    <link rel="stylesheet" type="text/css" href="css/principal.css">
</head>
<body>
<h1>Client Xat</h1>
<ul id="missatges"></ul>
<form action="">
    <input id="missatge" autocomplete="off" /><button>Enviar</button>
</form>

<script src="/socket.io/socket.io.js"></script>
<script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>
<script src="/js/client-xat.js"></script>

</body>
</html>
```

Com es pot apreciar, només es tracta d'una llista en què es mostraran els missatges, un formulari per entrar el text per enviar al servidor i la càrrega dels fitxers amb el codi JavaScript i les biblioteques necessàries.

Cal destacar que el fitxer `socket.io.js` no es troba realment al servidor, sinó que és generat pel *framework* i, per consegüent, sempre serà servit utilitzant la ruta `/socket.io/socket.io.js`.

Dintre del directori `css` heu de crear un fitxer amb el nom `principal.css` i el codi següent:

```
* {
    margin: 0;
    padding: 0;
    box-sizing: border-box;
}

body {
    font-family: Arial, "Helvetica Neue", Helvetica, sans-serif
}

h1 {
    text-align: center;
}

form {
    padding: 3px;
    position: fixed;
    bottom: 0;
    width: 100%;
}

form, h1 {
    background: #ccc;
}

form, ul {
    border-top: 1px solid black;
}

form input {
    border: 1px solid black;
    border-radius: 5px;
    padding: 10px;
```

```
    width: 90%;
    margin-right: .5%;
}

form button {
    width: 9%;
    font-weight: bold;
    background: #1b95e0;
    border-radius: 5px;
    border: 1px solid black;
    padding: 10px;
}

#missatges {
    font-family: "Courier New", Courier, "Lucida Sans Typewriter", "Lucida
        Typewriter", monospace    list-style-type: none;
    margin: 0;
    padding: 0;

}

#missatges li {
    padding: 5px;
}

#missatges li:nth-child(odd) {
    background: #eee;
}

.estat {
    color:blue;
}
```

En aquest cas, el codi CSS compleix una funció estrictament decorativa, ja que la funcionalitat de l'aplicació seria la mateixa si no s'hi apliqués cap estil.

Finalment, creeu un fitxer anomenat `client-xat.js` dintre del directori `js` amb el contingut següent per habilitar la connexió del client al servidor:

```
var socol = io();

$('form').submit(function(){
    socol.emit('missatge xat', $('#missatge').val());
    $('#missatge').val('');
    return false;
});

socol.on('missatge xat', function(msg){
    $('#missatges').append($('<li>').text(msg));
});
```

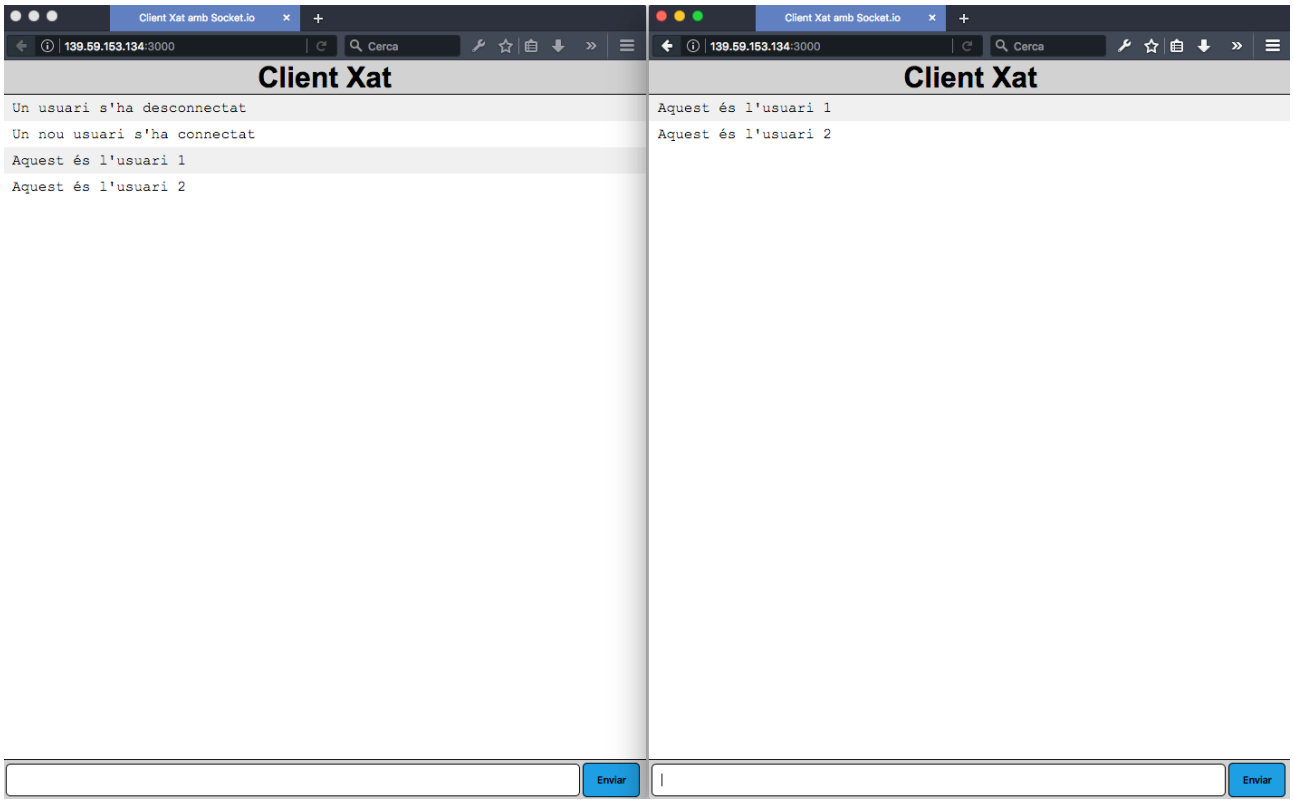
Per iniciar la connexió amb el servidor a través d'un sòcol només cal invocar la funció `io` (o `io.connect`), que pertany a la biblioteca `socket.io.js`. En cas que el client i el servidor es trobessin en diferents dominis o ports caldria especificar l'URL i el port com a paràmetres, però en aquest cas no és necessari.

Una vegada creat el sòcol s'afegeix un detector per a l'*event* `submit` del formulari que detectarà quan s'ha premut el botó d'enviament i, en lloc d'enviar el formulari, emetrà un *event* de tipus `missatge xat` amb el contingut del quadre de text amb l'identificador `missatge` mitjançant el sòcol (que serà rebut pel servidor). Seguidament s'establirà el contingut d'aquest quadre com a buit, a l'espera que s'introdueixi un text nou.

Per altra banda, s'afegeix un detector per a l'*event* `missatge xat` al sòcol, de manera que quan es rep un *event* d'aquest tipus s'executa la funció que s'ha passat com a paràmetre, afegint un nou element de tipus `li` al llistat `missatges` amb el contingut de l'*event*.

A la figura 1.9 es mostra l'aspecte del client funcionant en dues pestanyes diferents.

FIGURA 1.9. Client xat amb dues pestanyes



Fixeu-vos que és molt fàcil crear un protocol propi de comunicació per utilitzar en aplicacions més complexes. Per exemple, es podria discriminar entre els missatges d'estat del sistema i els missatges dels usuaris fent que l'*event* enviat fos diferent, i utilitzar un altre detector per gestionar-los.

Per comprovar-ho substituïu el contingut de la gestió de connexions del fitxer `servidor-xat.js` pel següent:

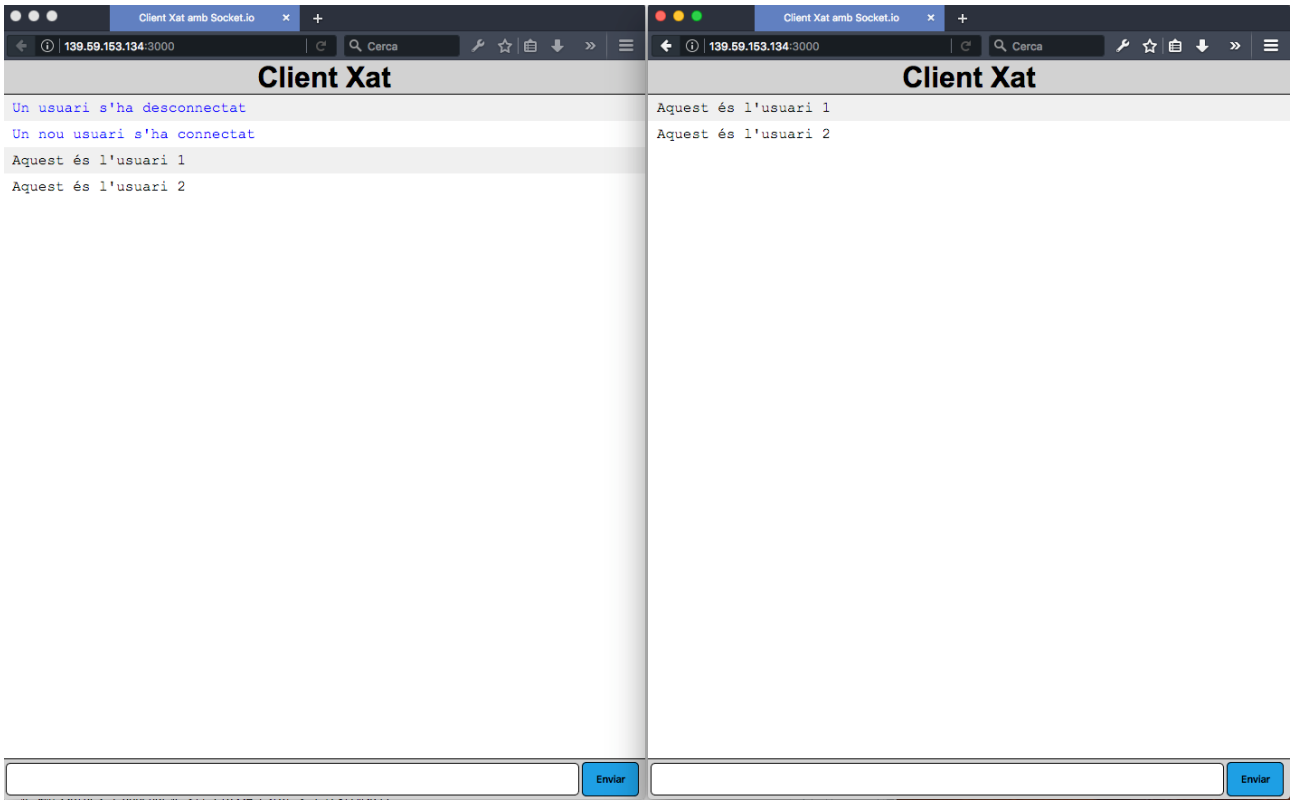
```
// Gestió de les connexions
io.on('connection', function(socol){
    socol.broadcast.emit('missatge estat', 'Un nou usuari s\'ha connectat');

    console.log('Un usuari connectat');

    socol.on('disconnect', function(){
        socol.broadcast.emit('missatge estat', 'Un usuari s\'ha desconnectat');
        console.log('usuari desconnectat');
    });

    socol.on('missatge xat', function(msg){
        console.log('missatge: ' + msg);
        io.emit('missatge xat', msg);

    });
});
```

I afegiu el següent detector el contingut del fitxer `client-xat.js`:

```
1 socol.on('missatge estat', function(msg){
2     $('#missatges').append($('<li class="estat">').text(msg));
3 });
```

Amb aquest canvi, quan es rep un *event* de tipus `missatge estat` s'afegeix un element de tipus `li` amb la classe `estat`, que fa que el missatge es mostri de color blau, tal com podeu veure a la figura 1.10.

FIGURA 1.10. Client xat ampliat amb missatges d'estat



Utilitzant aquest sistema, es podria ampliar l'aplicació per gestionar un llistat d'usuaris o distingir entre missatges globals i missatges privats entre usuaris, per exemple.

Com s'ha pogut comprovar, utilitzant els *frameworks* Express i Socket.io és molt senzill implementar aplicacions, tant a la banda del client com del servidor, que utilitzin sòcols per mantenir una connexió oberta.

1.2 Aplicacions amb la biblioteca Google Maps

Google ofereix moltes API per accedir als seus serveis. La majoria són gratuïtes, però tot i així es necessita crear una clau que permet identificar quines aplicacions estan fent servir cadascun dels serveis.

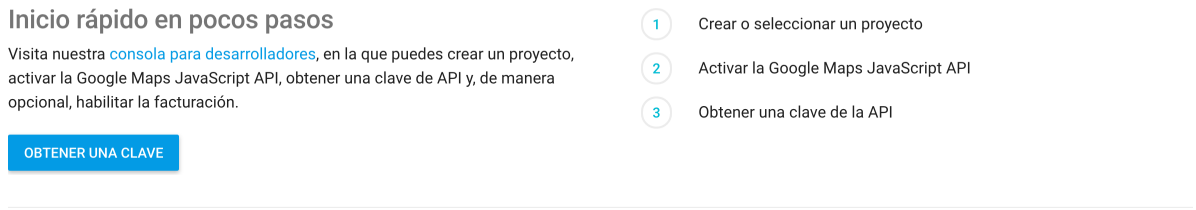
Algunes d'aquestes API, per exemple la de traducció, es poden fer servir fins que s'arriba a uns límits diaris predeterminats. D'altres permeten fer un determinat

nombre de consultes de forma gratuïta i se'n poden contractar més o sol·licitar-ne gratuïtament per a alguns serveis en fase de proves. Aquest nombre d'usos és conegut com a quota.

1.2.1 Obtenció d'una clau per utilitzar l'API Google Maps

En el cas de l'API Google Maps, tot i que per als usos més habituals és gratuïta, és necessari obtenir una clau. Per generar-la s'ha de fer des del panell de desenvolupador de Google, al qual es pot accedir seguint els passos des d'enllaç goo.gl/NnOJk4, i fent clic al botó ***Obtenir una clau***, com es pot veure a la figura 1.11.

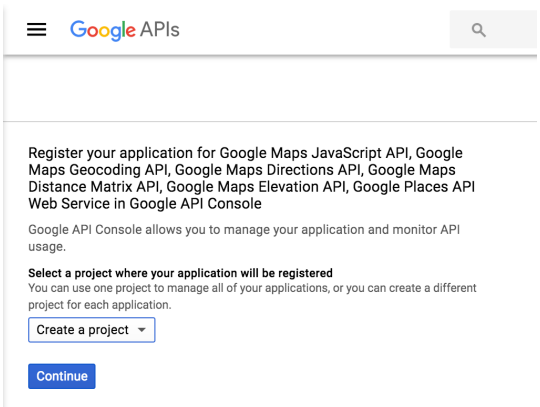
Figura 1.11. Obtenció de la clau per a l'API Google Maps



En cas de no estar autenticat amb un compte de Google, abans de continuar us demanarà les dades de connexió, ja que les vostres aplicacions, les claus i la consola de desenvolupador quedaran lligades al vostre compte.

El següent pas serà crear una aplicació (o seleccionar-ne una d'existent) a la qual s'afegirà la clau, com es mostra a la figura 1.12.

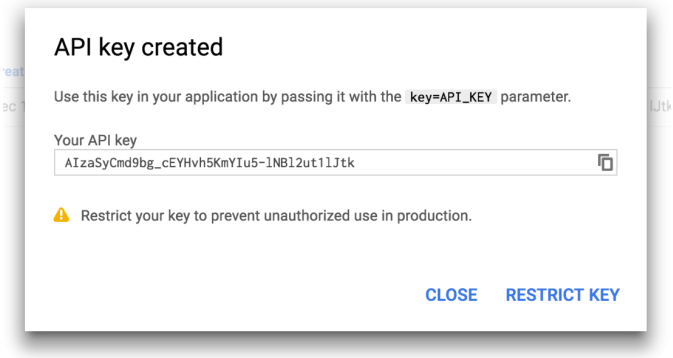
Figura 1.12. Creació d'un nou projecte



Cal tenir en compte que la creació del projecte pot trigar uns quants minuts i en finalitzar es mostrarà el taulell principal de la vostra aplicació automàticament.

A continuació, heu de procedir a generar la clau per a l'aplicació clicant sobre el botó ***Generar clau*** (opcionalment podeu canviar el nom que es mostra per defecte). La nova clau es mostrarà en un diàleg com el que es mostra a la figura 1.13, que podeu procedir a tancar.

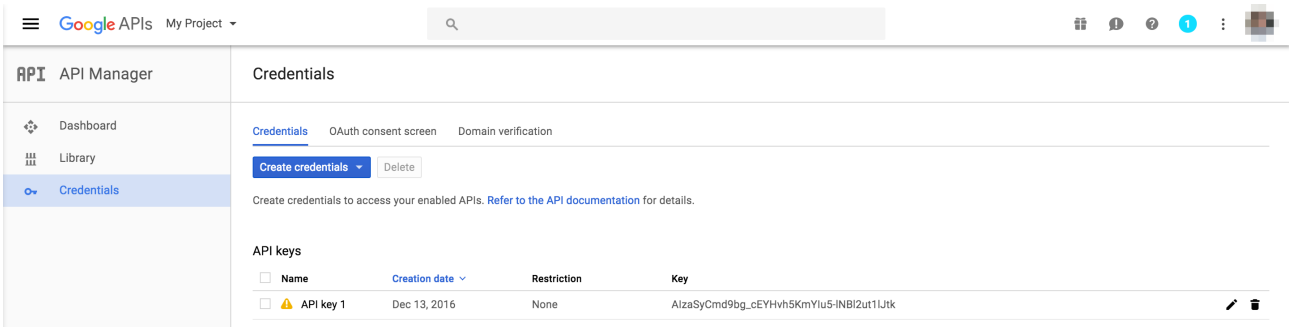
FIGURA 1.13. Diàleg que mostra la clau generada



Com es pot apreciar en el diàleg, us ofereixen l'opció de restringir aquesta clau perquè només sigui utilitzada des de determinats dominis o entorns d'execució. En aquest exemple no s'aplicarà cap restricció, però és un factor que s'ha de tenir en compte per evitar abusos de tercers que facin servir la vostra clau (recordeu que està lligada al vostre compte).

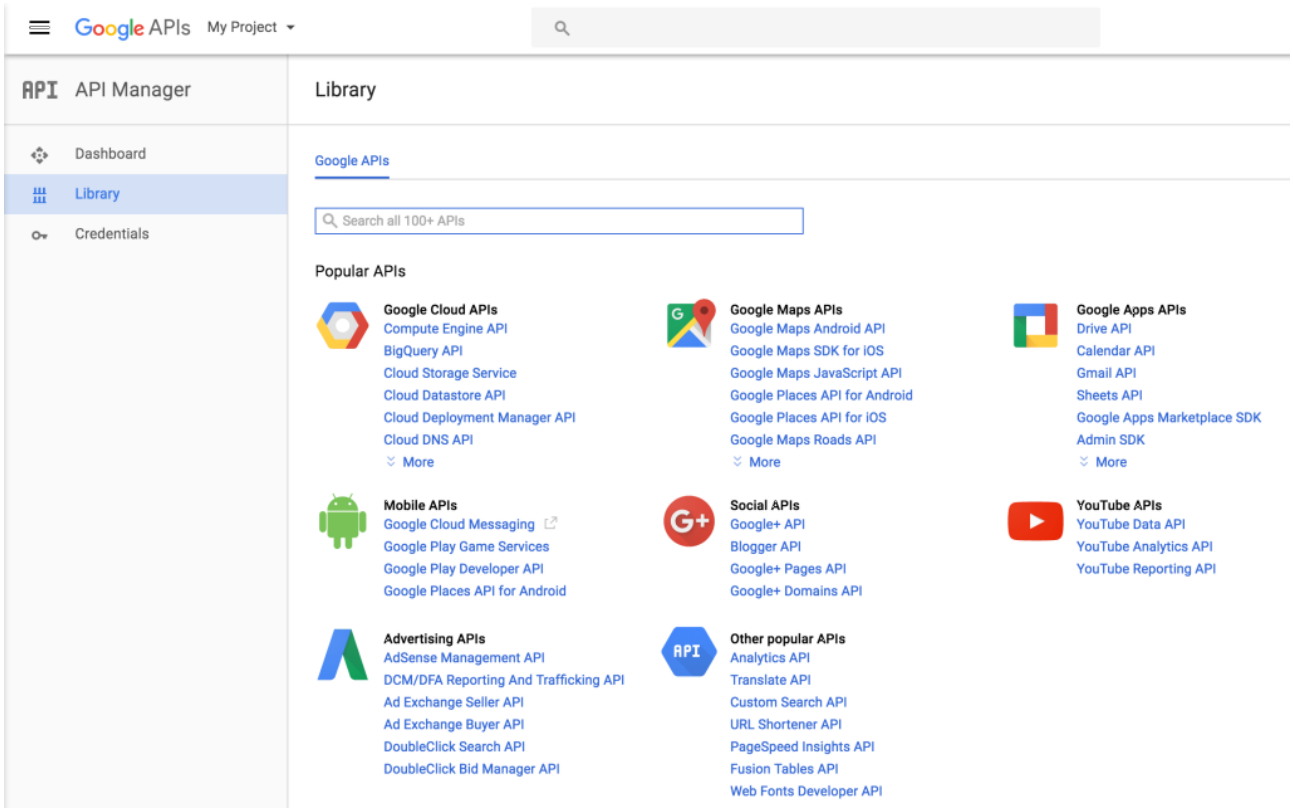
Finalment, la vostra clau apareixerà al taulell, com es mostra a la figura 1.14 a la secció de credencials.

FIGURA 1.14. Taulell de desenvolupadors que mostren la llista de claus



Cal destacar que seguint aquests passos s'activa automàticament l'API Google Maps per a JavaScript i es genera la clau apropiada. En cas de saltar-se algun pas o accedir directament al taulell s'haurà d'habilitar l'API manualment des de l'opció del menú ***Library*** (es pot veure el llistat a la figura 1.15) i generar les credencials des de l'opció del menú lateral ***Credentials***.

FIGURA 1.15. Taulell de desenvolupadors que mostren la biblioteca d'API



1.2.2 Creació d'un mapa simple

Un cop disposeu d'una clau, podeu començar a desenvolupar la vostra aplicació. Primerament haureu d'afegir la funció que serà l'encarregada de generar el mapa. Aquesta funció serà cridada automàticament en carregar-se la biblioteca de Google Maps.

Un exemple de funció d'inicialització, que generaria un nou mapa dins de l'element amb identificador `mapa`, centrat a la ciutat de Barcelona (que correspon a les coordenades especificades) i amb un zoom mitjà (el valor màxim per a la propietat `zoom` és 22), seria el següent:

```
new google.maps.Map(document.getElementById('mapa'), {
  center: {
    lat: 41.390205,
    lng: 2.154007
  },
  zoom: 11
});
```

Com es pot apreciar, el constructor rep dos arguments. El primer correspon al node on es vol dibuixar el mapa i el segon és un objecte amb les dades de configuració. En aquest cas s'ha especificat on s'ha de centrar el mapa (`center`) i quin nivell de zoom es desitja (`zoom`).

Podeu trobar més informació sobre l'element `script` i els seus atributs a l'enllaç següent: goo.gl/A7oxbg.

Seguidament s'ha de carregar la biblioteca de GoogleMaps, preferiblement utilitzant l'atribut `defer` ('diferir') per indicar que s'ha d'executar una vegada finalitzi

la càrrega del DOM. Juntament amb l'URL s'han d'incloure els paràmetres `callback` i `key`, que corresponen al nom de la funció a cridar per inicialitzar el mapa, i la vostra clau per a l'API Google Maps, generada a la consola de desenvolupadors de Google.

> La **biblioteca Google Maps** s'ha de carregar després que el vostre codi d'inicialització per evitar que sigui invocat per la biblioteca abans que s'hagi carregat.

Per exemple, si la funció que inicialitza els mapes s'anomena `iniciarAplicacioMapes` i la vostra clau és `AIzaSyDaYt85ZtjO2YDL-w94ZJLJE7F42HIr6Q0`, el codi per carregar la biblioteca seria el següent:

```
<script src="https://maps.googleapis.com/maps/api/js?key=AIzaSyDaYt85ZtjO2YDL-
    w94ZJLJE7F42HIr6Q0&callback=iniciarAplicacioMapes"></script>
```

Quant al codi HTML i CSS cal tenir en compte que caldrà utilitzar algun contenidor (per exemple, un element `div`) i especificar la mida que ha de tenir, tal com es pot veure en l'exemple següent, que mostra un mapa centrat a la ciutat de Barcelona que ocupa tota la pàgina:

```
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Aplicació amb Google Maps</title>
  <style>
    html,body {
      height: 100%;
      margin: 0;
      padding: 0;
    }

    #mapa {
      height: 100%;
    }
  </style>
</head>

<body>

  <div id="mapa"></div>

  <script>
    iniciarMapa = function() {
    new google.maps.Map(document.getElementById('mapa'), {
      center: {
        lat: 41.390205,
        lng: 2.154007
      },
      zoom: 11
    });
  }
  </script>

  <script defer src="https://maps.googleapis.com/maps/api/js?key=
      AIzaSyDaYt85ZtjO2YDL-w94ZJLJE7F42HIr6Q0&callback=iniciarMapa"></script>

</body>
```

```
39
40 </html>
```

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

Fixeu-vos que aquest exemple és molt limitat: com que no s'ha guardat cap referència al mapa no és possible manipular-lo programàticament. Afortunadament, només cal referenciar el mapa generat amb una variable per poder accedir-hi, per exemple:

```
var mapa = new google.maps.Map(document.getElementById('mapa'), {
  center: {
    lat: 41.390205,
    lng: 2.154007
  },
  zoom: 11
});
```

1.2.3 Marcadors

Una de les accions més habituals en els mapes és afegir-hi marcadors. Per afegir-los-hi es disposa de dos sistemes:

- Especificar a quin mapa està associat el constructor i assignar-lo a la propietat `map`.
- Invocar el mètode `setMap` del marcador generat prèviament i passar el mapa com a argument.

Es pot trobar més informació sobre els marcadors a l'enllaç següent: goo.gl/dgAJVs.

A continuació podeu veure un exemple de creació d'un marcador en què s'ha especificat la posició del marcador, el mapa on s'ha de visualitzar i un títol que es mostra quan s'hi posa el cursor del ratolí:

```
var marcador = new google.maps.Marker({
    position: {
      lat: 41.375106,
      lng: 2.168342
    },
    map: mapa,
    title: "Seu central"
  });
```

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

Una altra opció és crear primer els marcadors i assignar el mapa només en el moment en què es vulguin mostrar. Aquesta és l'opció utilitzada en el següent exemple, en el qual s'ha aplicat el disseny descendent per dividir l'aplicació en diferents funcions. Per una banda es crea el mapa i, per l'altra, un conjunt de

marcadors que són emmagatzemats en un *array*. Una vegada s'acaben de generar es recorre l'*array* i s'afegeixen al mapa:

```
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Aplicació amb Google Maps</title>

  <style>
    html,
    body {
      height: 100%;
      margin: 0;
      padding: 0;
    }

    #mapa {
      height: 100%;
    }
  </style>
</head>

<body>

  <div id="mapa"></div>

  <script>
    var mapa;
    var marcadors = [];

    var iniciarMapa = function() {
      mapa = new google.maps.Map(document.getElementById('mapa'), {
        center: {
          lat: 41.390205,
          lng: 2.154007
        },
        zoom: 7
      });
    }

    var crearMarcadors = function() {
      marcadors.push(crearMarcador("IOC - Seu central", 41.375106, 2.168342));
      marcadors.push(crearMarcador("Proves Barcelona", 41.3860669, 2.1145104));
      marcadors.push(crearMarcador("Proves Girona", 41.961293, 2.829387));
      marcadors.push(crearMarcador("Proves Lleida", 41.623023, 0.6236748));
      marcadors.push(crearMarcador("Proves Tarragona", 41.120293, 1.247892));
    }

    var crearMarcador = function(titol, latitud, longitud) {
      var marcador = new google.maps.Marker({
        position: {
          lat: latitud,
          lng: longitud
        },
        title: titol
      })

      return marcador;
    }

    var afegirMarcadors = function() {
      for (var i = 0; i < marcadors.length; i++) {
        console.log("Afegir marcador: " + i);
        afegirMarcador(marcadors[i]);
      }
    }

    var afegirMarcador = function(marcador) {
```

```
      marcador.setMap(mapa);
    }

    var iniciarAplicacio = function() {
      iniciarMapa();
      crearMarcadors();
      afegirMarcadors();
    }
  </script>

  <script defer src="https://maps.googleapis.com/maps/api/js?key=
      AIzaSyDaYt85ZtjO2YDL-w94ZJLJE7F42HIr6Q0&callback=iniciarAplicacio"></
      script>
</body>

</html>
```

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

Com es pot apreciar, una vegada es carrega la biblioteca de Google Maps, s'invoca la funció `iniciarAplicació`, que al seu torn crida les funcions `iniciarMapa` per afegir el mapa a la pàgina, `crearMarcadors` per generar els marcadors que s'emmagatzemmen a l'*array* `marcadors` i, finalment, la funció `afegirMarcadors`, que els afegeix a la pàgina.

S'ha embolcallat la creació de marcadors dins de la funció `crearMarcador`, que espera com a paràmetres un títol, la latitud i la longitud. D'aquesta manera es pot invocar la funció passant només aquests tres arguments per generar-lo (per exemple: `crearMarcador(''IOC - Seu central'', 41.375106, 2.168342)`.

Fixeu-vos que, partint d'aquesta implementació, és força senzill modificar-la per generar els marcadors a partir d'una estructura de dades, tal com es pot apreciar en l'exemple següent, en el qual s'ha modificat la funció `crearMarcadors` per utilitzar l'*array* `dades`, que conté la informació necessària.

```
var dades = [{
  seu: "IOC – Seu Central",
  lat: 41.375106,
  lon: 2.168342
}, {
  seu: "Proves Barcelona",
  lat: 41.3860669,
  lon: 2.1145104
}, {
  seu: "Proves Girona",
  lat: 41.961293,
  lon: 2.829387
}, {
  seu: "Proves Lleida",
  lat: 41.623023,
  lon: 0.6236748
}, {
  seu: "Proves Tarragona",
  lat: 41.120293,
  lon: 1.247892
}];

var crearMarcadors = function() {
  for (var i = 0; i < dades.length; i++) {
    marcadors.push(crearMarcador(dades[i].seu, dades[i].lat, dades[i].lon));
  }
}
```

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

Cal tenir en compte que en aquest cas l'origen de les dades és un *array* d'objectes JavaScript que s'ha assignat directament al codi, però el funcionament de l'aplicació seria el mateix si les dades s'obtinguessin d'una altra font, per exemple, com a resposta d'una petició AJAX.

1.2.4 Finestres d'informació

Habitualment els marcadors mostren informació ampliada quan s'hi clica, però aquesta informació no es pot afegir directament als marcadors. Per una banda cal crear la finestra d'informació (`InfoWindow`) i, per l'altra, s'hi ha d'afegir un detector d'esdeveniments.

La creació de les finestres d'informació és força simple, només cal especificar-ne el contingut: una cadena de text en format HTML. És a dir, una finestra d'informació pot contenir enllaços, imatges i qualsevol tipus de contingut HTML.

En cas de treballar amb un únic marcador, n'hi hauria prou amb un codi com el següent:

```
1  var crearMarcador = function(dada) {
2    var finestraInfo = new google.maps.InfoWindow({
3      content: '<h3>' + dada.seu + '</h3><p><b>Adreça:</b>' + dada.adreca + '</p>
           '
4    });
5
6    var marcador = new google.maps.Marker({
7      position: {
8        lat: dada.lat,
9        lng: dada.lon
10     },
11     map: mapa,
12     title: dada.seu,
13   });
14
15   marcador.addListener('click', function() {
16     finestraInfo.open(mapa, this);
17   });
18 }
```

Les clausures es tracten a la unitat "Estructures definides pel programador" d'aquest mateix mòdul.

Cal destacar que, en afegir el detector d'esdeveniments, es crea una clausura. És a dir, quan es crida la funció associada a l'*event* clic, aquesta continua tenint accés al context concret en el qual s'ha generat i, per consegüent, té accés a l'objecte `finestraInfo` (que conté la informació de la finestra d'informació) que li correspon independentment de què s'hagin creat múltiples marcadors.

Vegeu a continuació com s'ha afegit l'adreça al conjunt de dades i com es guarda la referència de cada finestra per poder fer que es tanquin totes abans d'obrir-ne una de nova:

```
1  var mapa;
2  var dades = [{
```

```
  seu: "IOC – Seu Central",
  lat: 41.375106,
  lon: 2.168342,
  adreca: "Avinguda del Paral·lel, 71. Barcelona"
}, {
  seu: "Proves Barcelona",
  lat: 41.3860669,
  lon: 2.1145104,
  adreca: "C/ John M. Keynes, 1–11. Barcelona"
}, {
  seu: "Proves Girona",
  lat: 41.961293,
  lon: 2.829387,
  adreca: "Carrer de la Universitat de Girona, 10. Girona",
}, {
  seu: "Proves Lleida",
  lat: 41.623023,
  lon: 0.6236748,
  adreca: "Pi i Margall, 51. Lleida",
}, {
  seu: "Proves Tarragona",
  lat: 41.120293,
  lon: 1.247892,
  adreca: "Av. de Catalunya, 35. Tarragona",
}];
var marcadors = [];
var finestresInfo = [];

var iniciarMapa = function() {
  mapa = new google.maps.Map(document.getElementById('mapa'), {
    center: {
      lat: 41.390205,
      lng: 2.154007
    },
    zoom: 7
  });
}

var crearMarcadors = function() {
  for (var i = 0; i < dades.length; i++) {
    var marcador = crearMarcador(dades[i]);
    marcadors.push(marcador);
  }
}

var crearMarcador = function(dada) {
  var finestraInfo = new google.maps.InfoWindow({
    content: '<h3>' + dada.seu + '</h3><p><b>Adreça:</b>' + dada.adreca + '</p>
        '
  });

  var marcador = new google.maps.Marker({
    position: {
      lat: dada.lat,
      lng: dada.lon
    },
    title: dada.seu,
  });

  marcador.addListener('click', function() {
    tancarTotesLesFinestres();
    finestraInfo.open(mapa, this);
  });

  finestresInfo.push(finestraInfo);

  return marcador;
}

var tancarTotesLesFinestres = function() {
```

```
72    for (var i = 0; i < finestresInfo.length; i++) {
73      finestresInfo[i].close();
74    }
75  }
76
77  var afegirMarcadors = function() {
78    for (var i = 0; i < marcadors.length; i++) {
79      console.log("Afegir marcador: " + i);
80      afegirMarcador(marcadors[i]);
81    }
82  }
83
84  var afegirMarcador = function(marcador) {
85    marcador.setMap(mapa);
86  }
87
88  var iniciarAplicacio = function() {
89    iniciarMapa();
90    crearMarcadors();
91    afegirMarcadors();
92  }
```

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

S'ha afegit l'*array* `finestresInfo` per guardar la referència de totes les finestres creades i la funció `tancarTotesLesFinestres`. Aquesta funció es crida en fer clic sobre un marcador i s'encarrega de recórrer aquest *array* per invocar el mètode `close` de cadascuna de les finestres d'informació.

Fixeu-vos que a la funció `crearMarcador` s'ha implementat la creació de la finestra i s'ha afegit la invocació a la funció `tancarTotesLesFinestres` a la funció invocada en detectar-se l'*event* `click`. A més a més, cada finestra creada es guarda a l'*array* `finestresInfo`.

Si en lloc d'afegir la creació de les finestres d'informació a la funció `crearMarcador` ho feu a la funció `crearMarcadors`, us trobareu amb un error important, ja que en crear-se la clausura entre la funció invocada en fer clic i la funció `crearMarcador` sempre s'accedirà a l'última finestra generada. Això es pot apreciar en l'exemple següent, en el qual s'han modificat només les funcions `afegirMarcador` i `afegirMarcadors`:

```
1   var crearMarcadors = function() {
2     for (var i = 0; i < dades.length; i++) {
3       var marcador = crearMarcador(dades[i]);
4       marcadors.push(marcador);
5
6       var finestraInfo = new google.maps.InfoWindow({
7         content: '<h3>' + dades[i].seu + '</h3><p><b>Adreça:</b>' + dades[i].
              adreca + '</p>'
8       });
9
10      marcador.addListener('click', function() {
11        window.tancarTotesLesFinestres();
12        finestraInfo.open(mapa, this);
13      });
14
15      finestresInfo.push(finestraInfo);
16    }
17  }
18
19  var crearMarcador = function(dada) {
```

```
  var marcador = new google.maps.Marker({
    position: {
      lat: dada.lat,
      lng: dada.lon
    },
    title: dada.seu,
  });

  return marcador;
}
```

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

Com es pot apreciar, totes les finestres d'informació mostren les dades de l'últim element creat. Com que el context de totes les funcions de detecció és el mateix (la funció `crearMarcadors` només ha estat invocada una vegada) totes accedeixen a la mateixa variable `finestraInfo`, que contindrà l'últim valor assignat; és a dir, la finestra amb la informació `Proves Tarragona`.

Fixeu-vos que, en cas d'afegir la finestra d'informació a la funció `crearMarcador`, s'invoca una vegada per a cada marcador i, conseqüentment, el context d'execució de cada funció de detecció és únic.

S'ha d'anar amb compte quan es treballa amb clausures i detecció d'*events*: s'hi poden cometre fàcilment i són difícils de depurar.

1.2.5 Cas pràctic: integrar una font de dades obertes amb Google Maps

En aquest exemple es treballaran els punts següents:

- Ús de la biblioteca jQuery.
- Ús de la biblioteca Google Maps.
- Cerca de serveis Open Data.
- Realització de consultes AJAX.
- Càrrega d'un mapa de Google Maps.
- Creació de marcadors.
- Creació de finestres d'informació.

Quan es necessita treballar amb mapes, el més habitual és que la informació que s'hagi de mostrar s'obtingui d'una font de dades externa, com per exemple un servei web propi o dades obertes ofertes per tercers mitjançant AJAX.

Cal recordar que en el cas de dades de tercers, només seran accessibles si el servidor implementa mecanismes CORS o JSONP, ja que en cas contrari els navegadors bloquejaran la petició AJAX.

Una d'aquestes fonts de dades obertes es troba a la pàgina de l'Observatori de dades culturals de Barcelona (barcelonadadescultura.bcn.cat/dades-obertes), on podeu trobar la documentació de l'API (dades.eicub.net/doc).

A partir de la informació que es troba en aquests enllaços, és possible realitzar consultes a l'API per obtenir diferents llistats d'informació; per exemple, per obtenir informació sobre els cinemes només cal cercar a la documentació en format XML (dades.eicub.net/doc) la paraula *cinemes* i trobareu les dades següents:

```
<dataset>
  <name>cinemes_dadesglobals</name>
  <url>http://dades.eicub.net/api/1/cinemes_dadesglobals</url>
  <title>Dades globals dels cinemes</title>
  <description>Dades globals dels cinemes de la ciutat</description>
  <columns>
    <column>Any</column>
    <column>Dada</column>
    <column>Valor</column>
    <column>Nota</column>
  </columns>
</dataset>
<dataset>
  <name>cinemes_inventari</name>
  <url>http://dades.eicub.net/api/1/cinemes_inventari</url>
  <title>Inventari Cinemes</title>
  <description>Inventari de les sales de cinema</description>
  <columns>
    <column>Districte</column>
    <column>Equipament</column>
    <column>Titularitat</column>
    <column>Latitud</column>
    <column>Longitud</column>
    <column>Web</column>
  </columns>
</dataset>
```

És a dir, hi ha dos conjunts de dades referents a cinemes: un que mostra les dades globals de cada cinema de la ciutat i un altre que mostra el llistat de sales de cinema. Com es pot apreciar, per accedir a aquestes dades cal utilitzar l'URL que s'indica, que per al llistat de cinemes és http://dades.eicub.net/api/1/cinemes-inventari.

Fixeu-vos que a la documentació s'indica una sèrie de columnes per a cada conjunt de dades. Aquestes dades es corresponen amb la informació proporcionada pel llistat corresponent per a cadascun dels cinemes. Així doncs, a l'"Inventari Cinemes" es poden trobar les coordenades per localitzar els cinemes sobre el mapa.

A partir d'aquesta informació es podria decidir implementar una aplicació que mostrés a un mapa els punts on es troben tots els cinemes de Barcelona, utilitzant la informació proporcionada de la manera següent:

- **Equipament**: com a títol del marcador i capçalera de la finestra d'informació.
- **Districte**, **Titularitat** i **Web**: llistada a la finestra d'informació.
- **Latitud** i **Longitud**: utilitzades per crear el marcador.

Així doncs, una possible implementació d'aquest mapa seria la que es troba a continuació. Les dades es carreguen mitjançant AJAX des de l'URL dades.eicub.net/api/1/cinemes-inventari i una vegada descarregades es creen els marcadors i finestres d'informació:

```
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Localització de cinemes a la ciutat de Barcelona</title>

  <style>
    html, body {
    height: 100%;
    margin: 0;
    padding: 0;
  }

  #mapa {
    height: 100%;
  }

  ul {
    list-style-type: none;
    padding: 0;
  }

  li b {
    display: inline-block;
    width: 75px;
  }
</style>
</head>

<body>

  <div id="mapa"></div>
  <script src="https://code.jquery.com/jquery-3.1.1.min.js"></script>

  <script>
    var mapa;
    var dades;
    var marcadors = [];
    var finestresInfo = [];

    var iniciarMapa = function() {
      mapa = new google.maps.Map(document.getElementById('mapa'), {
        center: {
          lat: 41.390205,
          lng: 2.154007
        },
        zoom: 13
      });
    }

    var crearMarcadors = function() {
      for (var i = 0; i < dades.length; i++) {
        var marcador = crearMarcador(dades[i]);
        marcadors.push(marcador);
      }
    }

    var tancarTotesLesFinestres = function() {
      for (var i = 0; i < finestresInfo.length; i++) {
        finestresInfo[i].close();
      }
    }

```

```
    var crearMarcador = function(dada) {
      var finestraInfo = new google.maps.InfoWindow({
        content: '<h3>' + dada.Equipament + '</h3>' +
          '<ul>' +
          '<li><b>Districte:</b>' + dada.Districte + '</li>' +
          '<li><b>Titularitat:</b>' + dada.Titularitat + '</li>' +
          '<li><b>Web:</b><a href="' + dada.Web + '">' + dada.Web + '</a></li>'
               +
          '</ul>'
      });

      var marcador = new google.maps.Marker({
        position: {
          lat: parseFloat(dada.Latitud),
          lng: parseFloat(dada.Longitud)
        },
        title: dada.Equipament,
      });

      marcador.addListener('click', function() {
        tancarTotesLesFinestres();
        finestraInfo.open(mapa, this);
      });

      finestresInfo.push(finestraInfo);

      return marcador;
    }

    var afegirMarcadors = function() {
      for (var i = 0; i < marcadors.length; i++) {
        afegirMarcador(marcadors[i]);
      }
    }

    var afegirMarcador = function(marcador) {
      marcador.setMap(mapa);
    }

    var iniciarMarcadors = function() {
      crearMarcadors();
      afegirMarcadors();
    }

    var iniciarAplicacio = function() {
      iniciarMapa();
      carregarDades();
    }

    var carregarDades = function() {
      $.ajax('http://dades.eicub.net/api/1/cinemes_inventari', {
        dataType: 'json',
        success: function(resposta) {
          dades = resposta;
          iniciarMarcadors();
        }
      });
    }
  </script>
  <script defer src="https://maps.googleapis.com/maps/api/js?key=
      AIzaSyDaYt85ZtjO2YDL_w94ZJLJE7F42HIr6Q0&callback=iniciarAplicacio"></
      script>
</body>

</html>
```

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

Fixeu-vos que aquesta aplicació té unes fortes dependències i si es canviés l'ordre d'execució, l'aplicació no funcionaria. Per una banda, la biblioteca jQuery s'ha de carregar abans que s'iniciï l'aplicació, perquè la càrrega de dades en depèn, ja que s'ha utilitzat per simplificar la crida AJAX. Per altra banda, la funció `iniciarAplicació` ha d'estar disponible abans que acabi de carregar la biblioteca de Google Maps, ja que per inicialitzar els mapes cridarà aquesta funció.

Les dades han de ser carregades abans de poder inicialitzar els marcadors i després que s'hagi carregat la biblioteca de Google Maps. Per aquesta raó, s'invoca `carregarDades` dintre de la funció `iniciarAplicació`, que és invocada per la biblioteca Google Maps i, una vegada es rep la resposta a la crida AJAX, els marcadors s'inicialitzen invocant la funció `iniciarMarcadors`.

Quant a l'aspecte de l'aplicació, s'ha modificat el codi CSS per estendre el mapa a tota la pantalla i donar un format més atractiu a les finestres d'informació. Tingueu en compte que les finestres accepten tot tipus de codi HTML i, per tant, és possible afegir altres elements com, per exemple, imatges.

Les possibilitats que ofereix l'API de Google Maps és molt extensa i es recomana consultar els tutorials i la documentació oficial que podeu trobar a goo.gl/B8veWw en cas de necessitar funcionalitats més complexes, com per exemple afegir altres controls al mapa o personalitzar la icona del marcador.

1.3 Desenvolupament de jocs amb HTML5

A finals de l'any 2016 la majoria de fabricants de navegadors (Google i Mozilla entre altres) van deixar d'admetre la utilització de connectors per a Flash o per a les miniaplicacions (*applets*) de Java als navegadors. Per una banda, no es podia garantir la seguretat d'aquests connectors i, per l'altra, amb les característiques afegides a HTML5 ja no eren necessaris.

'Applet' o miniaplicació

Un *applet*, en català 'miniaplicació', és un petit programa desenvolupat en Java perquè s'executi al navegador.

S'ha de tenir en compte que quan es parla d'**HTML5** no només es tracta del llenguatge de marques HTML, sinó que engloba també CSS, JavaScript i les noves Web API afegides al llenguatge (com per exemple els elements `àudio`, `vídeo`, `canvas`, `API WebSockets`...).

Quant al desenvolupament de jocs amb HTML5, l'element clau va ser la inclusió de l'element `canvas` i, en menor mesura, els elements `àudio` i `vídeo`. El primer permet dibuixar formes i imatges dintre de l'element, mentre que els altres dos permeten reproduir àudio i vídeo directament al navegador.

Tot i que hi ha motors molt populars per desenvolupar jocs amb HTML5, com Construct 2 (www.scirra.com) o ImpactJs (impactjs.com), és relativament senzill desenvolupar un joc només utilitzant JavaScript, HTML i CSS.

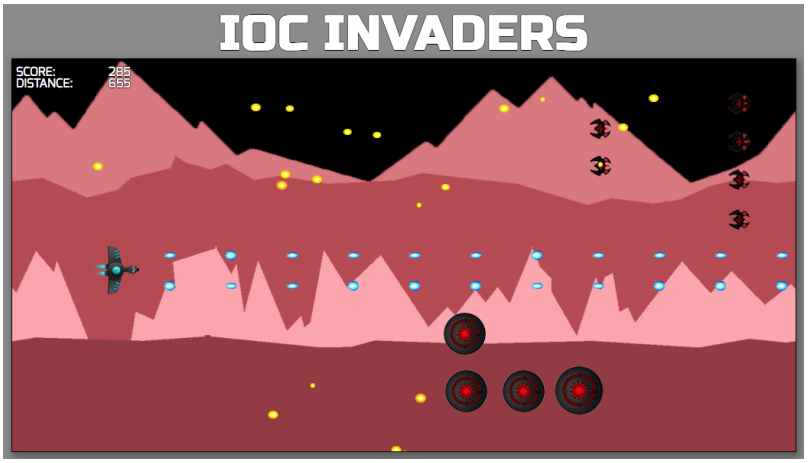
Cal destacar que molts d'aquests motors prometen exportar els jocs a diferents plataformes (Android, iOS, tvOS, etc.); en realitat, el que fan és generar una aplicació que conté una finestra de navegador sense cap botó, on executen el joc en JavaScript.

Per altra banda, hi ha altres motors de joc multiplataforma que es programen en altres llenguatges però exporten a JavaScript. Per exemple, el motor Unity (unity3d.com/es) permet desenvolupar jocs amb C# o JavaScript i exportar-los a HTML5 mitjançant un reproductor que utilitza l'API WebGL per renderitzar gràfics en 2D i 3D.

1.3.1 Introducció

Un exemple relativament senzill de com desenvolupar un joc amb JavaScript és l'*IOC Invaders* (vegeu la figura 1.16). Aquest joc permet il·lustrar com es pot estructurar el codi i aprofitar les característiques afegides a HTML5 (concretament els elements `canvas` i `àudio`) per crear un joc de naus amb múltiples nivells, tipus d'enemics i armes.

Figura 1.16. Joc "IOC Invaders" en funcionament



Podeu trobar el codi complet d'aquest joc a l'enllaç següent: github.com/XavierGaro/ioc-invaders. Tots els recursos són lliures, tant el codi com els recursos gràfics i d'àudio, però si els utilitzeu en els vostres projectes, heu d'incloure els enllaços a les fonts i mencionar-ne els autors.

Cal destacar la utilització del `canvas` tant per dibuixar la nau del jugador, els enemics, les bales i les explosions com per simular un desplaçament horitzontal (*scroll*, en anglès) amb quatre nivells de profunditat. Tot i que es fa servir contínuament la mateixa imatge, aquest sistema es podria millorar per utilitzar un *array* d'imatges per concatenar.

Un punt a tenir molt en compte quan es desenvolupen jocs és que s'han d'optimitzar per obtenir-ne el millor rendiment possible. Per exemple, en altres tipus d'aplicacions gestionar la creació i destrucció d'objectes no és crític perquè

l'execució del recol·lector de brossa és inapreciable. En canvi, quan s'han de gestionar centenars de naus, bales, efectes de fons i control del jugador, l'execució del recol·lector provoca una baixada de rendiment crítica i molt evident.

A *IOC Invaders*, per optimitzar l'ús de la memòria i reduir les crides al recol·lector de brossa, s'ha utilitzat una tècnica coneguda com a `pool`, que consisteix a reutilitzar els mateixos elements en lloc de destruir-los i crear-ne de nous. Això permet mantenir un mínim de 60 FPS (*frames* o fotogrames per segon), és a dir, l'element `canvas` es redibuixa com a mínim 60 vegades per segon.

Quant a la gestió de nivells i recursos s'han utilitzat fitxers de configuració en format JSON que es carreguen mitjançant AJAX. D'aquesta manera, sense modificar el codi del joc es poden modificar els nivells i els enemics, afegir-ne de nous o actualitzar-ne els recursos (imatges i sons).

Regles del joc

Abans de començar a programar cal tenir clares quines seran les regles del joc. Quan s'acaba la partida? Com s'arriba al final del nivell? Què passa si s'arriba al final del joc?

En el cas de l'*IOC Invaders* s'ha decidit utilitzar el següent conjunt de regles:

- Tant les naus de l'enemic com la del jugador són destruïdes si xoquen amb l'adversari.
- Tant les bales de l'enemic com les del jugador són destruïdes si impacten en un adversari.
- Qualsevol nau impactada per una bala de l'adversari és destruïda.
- Quan la nau del jugador és destruïda el joc s'acaba.
- Quan la nau d'un enemic és destruïda augmenta la puntuació del jugador.
- Els enemics apareixen quan s'ha recorregut una distància determinada del mapa que s'estableix per a cada nivell.
- El nivell es considera per finalitzat quan s'arriba a una distància determinada i no és visible cap enemic.
- Quan es finalitza l'últim nivell es torna al primer (conservant la puntuació).

Presentació del joc

Tot i que l'acció de qualsevol joc es trobarà representada dins del `canvas`, cal recordar que l'entorn d'execució és el navegador i, per consegüent, es té accés tant a HTML com a CSS. Això permet, per exemple, mostrar el títol del joc a la capçalera, els crèdits i les instruccions sota del joc i fins i tot utilitzar les puntuacions com etiquetes HTML sobre el `canvas`.

Per descomptat, es poden utilitzar efectes i animacions de CSS, per exemple, per mostrar un efecte de *fade out* (fos a negre) o desplaçament de lletres o nombres (puntuació actual, vides restants, missatges de felicitació, etc.).

A *IOC Invaders* s'ha aprofitat el fitxer HTML per afegir la capçalera del joc, els crèdits corresponents als recursos gràfics i d'àudio emprats i la llicència corresponent al peu, com es pot veure en el codi corresponent al fitxer `ioc-invaders.html`:

```
<!DOCTYPE html>
<html lang="ca">
<head>
    <meta charset="UTF-8">
    <title>IOC Invaders</title>
    <link href="https://fonts.googleapis.com/css?family=Russo+One" rel="
        stylesheet">
    <link href="https://fonts.googleapis.com/css?family=Exo" rel="stylesheet">
    <link href="css/style.css" rel="stylesheet">
</head>
<body>

<h1>IOC INVADERS</h1>

<div id="game-background">
</div>

<div class="credits">
    <h2>CRÈDITS</h2>
    <div>
        <h3>Música</h3>
        <ul>
            <li>Defense Line: <a target="_blank" href="https://www.dl-sounds.
                com/royalty-free/defense-line/">Background
                Loop</a></li>
            <li>Star Commander1(<a target="_blank" href="https://www.dl-sounds.
                com/royalty-free/star-commander1/"> DL
                Sounds</a>)
            </li>
        </ul>
    </div>

    <div>
        <h3>Efectes de so</h3>
        <ul>
            <li>Laser: <a target="_blank" href="http://soundbible.com/1087-
                Laser.html"> SoundBible.com</a></li>
            <li>Explosions: inferno (<a target="_blank" href="http://www.
                freesound.org/people/inferno/sounds/18384/">
                freesound.org</a>)
            </li>
            <li>Altres – Xavier Garcia Rogríguez</li>
        </ul>
    </div>

    <div>
        <h3>Gràfics</h3>
        <ul>
            <li>Naus: sujit1717 (<a target="_blank"
                                    href="http://opengameart.org/content/
                                        complete-spaceship-game-art-pack">
                                        Space Odyssey
                Pack</a>)
            </li>
            <li>Fons i trets: Xavier Garcia Rogríguez</li>
        </ul>
    </div>
</div>

<div class="footer">
```

```
    Xavier Garcia Rodríguez 2017. <a href="http://www.apache.org/licenses/
        LICENSE-2.0" target="_blank">Llicencia Apache
    2.0</a>. Podeu trobar el codi font d'aquest joc a <a target="_blank"
                                                        href="https://github.
                                                            com/XavierGaro/
                                                            ioc-invaders">
                                                            GitHub</a>.
</div>

<script src="js/ioc-invaders.js">
</script>

</body>
</html>
```

També s'han aprofitat les característiques de CSS3 per afegir alguns efectes de transició que són activats pel joc, i s'han utilitzat dues fonts externes:

- Vora negra al voltant de tot el text.
- Ombra al voltant del `canvas`.
- Missatges que es fan visibles i s'esvaeixen, així com pantalles de transició (classe `fadeable`).
- Utilització de les fonts Russo One i Exo facilitades per Google Fonts.
- Utilització de diferents colors per a diferents elements.

A continuació podeu trobar el codi corresponent al fitxer "css/style.css" que inclou tots els estils emprats:

```
body {
    color: white;
    background-color: gray;
    text-align: center;
    font-family: 'Russo One', sans-serif;
    text-shadow: 1px 0 0 #000, 0 -1px 0 #000, 0 1px 0 #000, -1px 0 0 #000;
}

canvas {
    margin: 0 auto;
    background: transparent;
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
}

h1, h2, h3 {
    margin: 0;
    line-height: 1;
}

h1 {
    font-size: 64px;
}

h2 {
    color: red;
    padding: 15px;
    font-size: 50px;
}
```

```
33
34  h3 {
35      font-size: 40px;
36      color: #FFD700;
37  }
38
39  ul {
40      list-style: none;
41      padding: 0;
42      text-align: center;
43      color: white;
44  }
45
46  li {
47      font-family: 'Exo', sans-serif;
48  }
49
50  a {
51      color: cornflowerblue;
52      text-decoration: none;
53      border-bottom: 1px dotted;
54  }
55
56  #game-background {
57      position: relative;
58      margin: 0 auto;
59      width: 1024px;
60      height: 512px;
61      border: 1px solid black;
62      -webkit-box-shadow: 3px 5px 10px 2px rgba(0, 0, 0, 0.39);
63      -moz-box-shadow: 3px 5px 10px 2px rgba(0, 0, 0, 0.39);
64      box-shadow: 3px 5px 10px 2px rgba(0, 0, 0, 0.39);
65      background-color: black;
66  }
67
68  #background {
69      z-index: -2;
70  }
71
72  .ui {
73      margin: 0 auto;
74      width: 1024px;
75      position: relative;
76      top: 0;
77  }
78
79  .score, .distance {
80      position: absolute;
81      top: 5px;
82      left: 5px;
83      color: #FFF;
84      cursor: default;
85      width: 150px;
86      text-align: left;
87      font-family: 'Exo', sans-serif;
88  }
89
90  .score span, .distance span {
91      float: right;
92  }
93
94  .distance {
95      top: 20px;
96  }
97
98  .game-over, .messages {
99      position: absolute;
100     top: 180px;
101     left: 0;
102     right: 0;
```

```
    color: red;
    font-size: 40px;
    cursor: default;
}

.game-over span, .messages span {
    font-size: 20px;
    position: relative;
}

.game-over span {
    cursor: pointer;
}

.game-over {
    z-index: 100;
    display: none;
    cursor: default;
}

.game-over span:hover, .messages {
    color: #FFD700;
}

.fadeable {
    -webkit-transition: opacity 3s ease-in-out;
    -moz-transition: opacity 3s ease-in-out;
    -ms-transition: opacity 3s ease-in-out;
    -o-transition: opacity 3s ease-in-out;
    opacity: 0;
}

.credits {
    margin: 0 auto;
    width: 1024px;
}

.credits div {
    width: 33%;
    float: left;
}

.footer {
    font-size: smaller;
    font-family: 'Exo', sans-serif;
    position: fixed;
    bottom: 0;
    width: 100%;
    text-align: center;
}
```

El resultat d'aplicar aquesta estructura HTML i els estils la podeu veure a la figura 1.17.

FIGURA 1.17. Representació de l'estructura HTML i CSS del joc "IOC Invaders"

Per facilitar la portabilitat del joc és recomanable que tots els elements imprescindibles per al seu funcionament es creïn mitjançant JavaScript. Així només cal comprovar que l'identificador del contenidor en el qual s'afegirà el joc és correcte. Per exemple, en el joc *IOC Invaders* només cal afegir un element amb l'identificador `game-background` i la resta d'elements es generaran automàticament.

Concretament, els elements necessaris per al funcionament del *IOC Invaders* s'afegeixen en finalitzar la càrrega de la pàgina mitjançant el codi següent, corresponent al final del fitxer `ioc-invader.js`:

```
1  window.onload = function () {
2      var gameContainer = document.getElementById('game-background'),
3          canvas,
4          game;
5
6      gameContainer.innerHTML = '' +
7          '<canvas id="game-canvas" width="1024" height="512" class="fadeable">'
               +
8          'Your browser does not support canvas. Please try again with a
               different browser.' +
9          '</canvas>' +
10         '<div class="ui">' +
11         '   <div class="score">SCORE: <span id="score"></span></div>' +
12         '   <div class="distance">DISTANCE: <span id="distance"></span></div>'
               +
13         '</div>' +
14         '<div class="game-over" id="game-over">GAME OVER<p><span>Restart</span
               ></p></div>' +
15         '<div class="messages fadeable" id="messages"></div>';
16
17     canvas = document.getElementById('game-canvas');
18
19     IOC_INVADERS.start({
20         "asset_data_url": "asset-data.json",
21         "entity_data_url": "entity-data.json",
22         "levels_data_url": "level-data.json"
```

```
    }, canvas);

    document.getElementById('game-over').addEventListener('click', IOC_INVADERS
        .restart);
};
```

Com es pot apreciar, s'obté la referència a l'element `game-background` i s'estableix com a contingut intern d'aquest element el codi HTML que conté el `canvas` i els elements que mostraran les puntuacions i els missatges.

A més a més s'obté la referència al `canvas` amb el qual s'inicialitza el joc, juntament amb la ruta de les dades corresponents als recursos i les entitats i els nivells que formaran part del joc. Per acabar, s'afegeix un detector de l'*event* `click` a l'element `game-over` per reinicialitzar el joc.

1.3.2 Encapsulament del joc

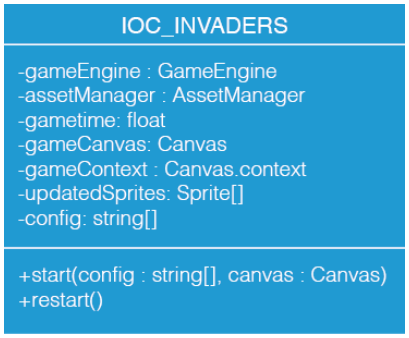
Per encapsular la informació és recomanable fer servir el patró de disseny *mòdul*. D'aquesta manera el joc no interfereix amb altres aplicacions i totes les seves funcions, mètodes i objectes queden aïllats de l'usuari, que no podrà manipular-lo, ja que només tindrà accés a les funcions públiques exposades pel mòdul.

En el cas del joc *IOC Invaders*, el mòdul disposa d'una sèrie de propietats internes, funcions constructores per a pràcticament tots els components del joc i dues funcions públiques que són accessibles externament i permeten iniciar el joc i reiniciar la partida: `start` i `restart`. A continuació, podeu trobar el codi del mòdul sense incloure els constructors de classes internes, i a la figura **??** podeu veure el diagrama UML corresponent:

```
var IOC_INVADERS = function () {
    var GameState = {
            GAME_OVER: "GameOver",
            LOADING_NEXT_LEVEL: "LoadingNextLevel",
            RUNNING: "Running"
        },

        MAX_TIME_OUTSIDE_BOUNDARIES = 180,
        gameCanvas,
        config,
        gameEngine,
        assetManager,
        gametime,
        gameContext,
        updatedSprites = [];

  // Resta de constructors i funciones internes

  return {
    start: function (conf, canvas) {
      config = conf;
      gameCanvas = canvas;
      gameManager = gameEngineConstructor();
      assetManager = assetManagerConstructor(function (current, total) {
        console.log("Carregats:" + current + "/" + total);
      });

      gameEngine.init();
```

```
29      },
30
31      restart: function () {
32        gameEngine.restart();
33      }
34    }
35  }();
```

Figura 1.18. Diagrama UML corresponent al mòdul IOC_INVADERS



Cal destacar que un dels objectes declarats és `GameState` i té un format que pot resultar-vos peculiar. Tot i que a JavaScript les enumeracions no existeixen (`enums` en altres llenguatges), és possible emular-les amb estructures com aquesta.

Podeu trobar més informació sobre els *enums* a l'enllaç següent: goo.gl/qnhKSC.

En programació no es recomana mai fer servir *nombres màgics* al codi (valors literals sense explicacions o en múltiples llocs): per una banda, és difícil saber a què fan referència i, per l'altra, és possible que estiguin duplicats en diversos punts del codi i, llavors, si cal canviar-los, resulta complicat.

Per resoldre aquest problema s'acostumen a fer servir constants o enumeracions. Quan els valors no estan relacionats entre ells és recomanable utilitzar constants (a *IOC Invaders* s'utilitza la constant `MAX_TIME_OUTSIDE_BOUNDARIES`, per exemple) però quan hi ha una relació entre ells és recomanable utilitzar una enumeració (`GameState` fa referència als estats possibles del joc).

A JavaScript les constants no existeixen, però es poden simular posant els noms en majúscules com a convenció.

Utilitzant constants i enumeracions és molt fàcil fer el canvi dels valors, ja que només s'ha de fer en un punt. A més a més, la majoria d'entorns de desenvolupament habiliten l'autocompletar i el ressaltat d'errors, si no s'ha escrit correctament el nom, cosa que no passa quan es treballa amb nombres i cadenes de text. A més a més, el codi és més entenedor; per exemple: no aporta la mateixa informació `var a = 100;` (a què fa referència 100?, és molt?, és poc?, és correcte?) que `var a = MAX_SPEED;`, que genera molts menys dubtes.

Tornant al tema de l'encapsulament, en algunes ocasions ens pot interessar fer tot el contrari: com per exemple estendre objectes propis de JavaScript per afegir funcionalitats que no es troben en el llenguatge. A continuació podem veure com s'amplia el prototip dels `arrays` afegint el mètode `contains` perquè comprovi si un element es troba dins de l'*array*:

```
1  /**
2   * @param {*} needle – objecte a cercar dins de l'array
3   * @returns {boolean} – cert si es troba o false en cas contrari
4   */
```

```
Array.prototype.contains = function (needle) {
    for (var i in this) {
        if (this[i] == needle) return true;
    }
    return false;
};
```

Aquesta funció s'utilitza dintre de l'*IOC Invaders* per comprovar si un `sprite` ja ha estat actualitzat (es troba dins de l'*array* `updatedSprites`) o no.

Un altre exemple d'extensió seria afegir la funció `clamp` (disponible en altres llenguatges) que permet *encaixonar* un valor entre un mínim i un màxim, de manera que el resultat estigui sempre dintre d'aquests límits:

```
/*
 * @param {number} min – valor mínim
 * @param {number} max – valor màxim
 * @returns {number} – El nombre si està dins del limit o el valor corresponent
      al límit
 */
Number.prototype.clamp = function (min, max) {
    return Math.min(Math.max(this, min), max);
};
```

Un exemple d'utilització d'aquesta funció es troba en l'objecte `player`, que es fa servir per evitar que la nau del jugador surti de la pantalla:

```
that.position.x = that.position.x.clamp(0, gameCanvas.width – that.sprite.size.
    width);
that.position.y = that.position.y.clamp(0, gameCanvas.height – that.sprite.size
    .height);
```

Ara compareu-lo amb el codi sense utilitzar el mètode `clamp`; s'entén fàcilment però és molt llarga:

```
if (that.position.x <0) {
  that.position.x = 0;
} else if (that.position.x > gameCanvas.width – that.sprite.size.width) {
  that.position.x = gameCanvas.width – that.sprite.size.width;
}

if (that.position.y < 0) {
  that.position.y = 0;
} else if (that.position.y > gameCanvas.height – that.sprite.size.height) {
  that.position.y = gameCanvas.height – that.sprite.size.height;
}
```

I, finalment, compareu-lo amb la implementació fent servir els mètodes `Math.max` i `Math.min`, no és tan curta com amb el mètode `clamp` i és més complicada d'entendre:

```
that.position.x = Math.min(Math.max(that.position.x, 0), gameCanvas.width –
    that.sprite.size.width);
that.position.y = Math.min(Math.max(that.position.y, 0), gameCanvas.height –
    that.sprite.size.height);
```

S'ha de tenir en compte que habitualment no es recomana modificar el prototipus dels objectes del llenguatge, però hi ha ocasions en què és molt útil i permet escriure un codi més concís i més entenedor.

1.3.3 Gestió de les dades

Habitualment, quan es desenvolupa un joc s'ha de treballar amb dades que han de ser manipulades i reutilitzades. Per exemple, la informació sobre els tipus d'enemics o els tipus d'armes que es poden trobar, la informació sobre les peces d'un puzle... Aquesta informació es pot tractar com un catàleg només de consulta: a partir d'aquesta informació es generen els elements del joc i no s'ha de modificar.

Aquestes dades poden provenir de fonts externes (per exemple, carregades mitjançant AJAX) o poden estar incrustades al codi. Es pot considerar que aquestes dades són un dipòsit o un catàleg, a partir del qual es construeixen els objectes que s'utilitzaran en el joc.

En molts casos un dipòsit de dades no es limita a emmagatzemar dades, sinó que n'acostuma a realitzar algun tipus de tractament abans d'afegir-les o recuperar-les. El seu funcionament és molt similar al del patró de disseny factoria, però amb una implementació més simple.

Podeu trobar més informació sobre el patró de disseny factoria a l'enllaç següent: goo.gl/r3sBkz.

Per exemple, a *IOC Invaders* s'han implementat dos dipòsits de dades (*respository*, en anglès) diferents. Per una banda, s'ha creat un dipòsit d'entitats que s'encarrega de gestionar les dades de les entitats que formen el joc (naus, bales, explosions...) i, per l'altra, un dipòsit d'estratègies que permet assignar diferents funcions de moviment a les entitats (concretament a les bales i les naus enemigues).

A continuació podeu veure com s'ha implementat el dipòsit d'*entitats* (es tracta d'un nom escollit arbitràriament que fa referència a les dades que s'utilitzen per crear instàncies d'objectes al joc, vegeu la figura 1.19), que pot rebre un *array* d'entitats (per exemple, carregades des d'un fitxer d'entitats) i en cas necessari assigna la funció `move` obtinguda del dipòsit d'estratègies:

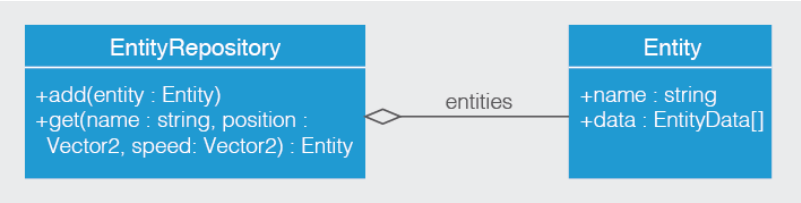
Crear instàncies

En el context de JavaScript, *crear instàncies* o en alguns llocs 'instanciar' (de l'anglès *instance*), consisteix a crear un objecte nou (una instància) a partir d'un altre objecte. És a dir, el nou objecte tindrà els mateixos mètodes i propietats públics de l'objecte original.

```
1  var entitiesRepository = (function () {
2    var entities = {};
3
4    function addEntity(name, data) {
5      var entity = data;
6
7      if (data.move) {
8        entity.move = strategiesRepository.get(data.move);
9      }
10
11     entities[name] = entity;
12   }
13
14   return {
15     add: function (entity) {
16       if (Array.isArray(entity)) {
17         for (var i = 0; i < entity.length; i++) {
18           addEntity(entity[i].name, entity[i].data);
19         }
20       } else {
21         addEntity(entity.name, entity.data);
22       }
23     },
24
25     get: function (name, position, speed) {
26       var entity = entities[name];
```

```
        if (!entity) {
          console.error("No es troba la entitat: ", entity);
        }
        entity.position = position;
        entity.speed = speed;

        return entity;
    }
  }
})();
```

Figura 1.19. Diagrama UML dipòsit d'entitats



Fixeu-vos que es tracta d'una funció autoinvocada i, per conseqüent, només exposa els mètodes de l'objecte que retorna (`add` i `get`). Així s'encapsulen les dades dintre del dipòsit i s'assegura que només s'accedirà als elements de la forma prevista.

Al contrari que en el dipòsit anterior, el dipòsit d'estratègies no obté les dades d'una font externa, sinó que conté un diccionari de funcions que correspon a les diverses estratègies que poden assignar-se a les entitats, com es pot veure a continuació:

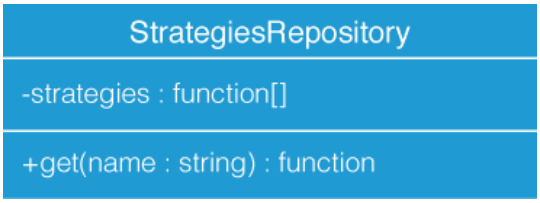
```
var strategiesRepository = (function () {
  var strategies = {

    movement_pattern_a: function () {
      if (!this.extra.ready) {
        this.extra.speed = Math.max(Math.abs(this.speed.x), Math.abs(this.speed
            .y));
        this.extra.leftEdge = this.position.x - 10 * this.extra.speed;
        this.extra.rightEdge = this.position.x + 10 * this.extra.speed;
        this.extra.topEdge = this.position.y + 10 * this.extra.speed;
        this.extra.bottomEdge = this.position.y - 10 * this.extra.speed;
        this.extra.direction = {x: this.speed.x <= 0 ? -1 : 1, y: 1};
        this.extra.ready = true;
      }

      this.position.x += this.speed.x;
      this.position.y += this.speed.y * this.extra.direction.y;


      if (this.position.y > this.extra.topEdge || this.position.y < this.extra.
          bottomEdge) {
        this.speed.x = this.extra.direction.x >= 0 ? this.extra.speed : -this.
            extra.speed;
        this.speed.y = 0;
        this.extra.direction.y = -this.extra.direction.y;
      }

      if (this.position.x <= this.extra.leftEdge) {
        this.speed.x = 0;
        this.speed.y = this.extra.speed;
        this.extra.leftEdge = this.position.x - 10 * this.extra.speed;
      } else if (this.position.x >= this.extra.rightEdge) {
        this.speed.x = 0;
```

```
31          this.speed.y = this.extra.speed;
32          this.extra.rightEdge = this.position.x + 10 * this.extra.speed;
33        }
34
35        this.position.y = this.position.y.clamp(this.extra.bottomEdge, this.extra
              .topEdge);
36      },
37
38      movement_pattern_b: function () {
39        this.position.x += this.speed.x;
40        this.position.y += this.speed.y;
41      },
42
43      movement_pattern_c: function () {
44        if (!this.extra.ready) {
45          this.extra.age = 0;
46          this.extra.speed = Math.max(Math.abs(this.speed.x), Math.abs(this.speed
                .y));
47          this.extra.ready = true;
48          this.extra.vertical = this.speed.x > this.speed.y;
49        }
50
51        if (this.extra.direction === 1) {
52          this.speed.x = this.extra.speed * Math.cos(-this.extra.age * Math.PI /
                64);
53        } else {
54          this.speed.y = this.extra.speed * Math.sin(this.extra.age * Math.PI /
                64);
55        }
56
57        this.extra.age++;
58        this.position.x += this.speed.x;
59        this.position.y += this.speed.y;
60      },
61
62      movement_pattern_d: function () {
63        if (!this.extra.ready) {
64          this.extra.age = 0;
65          this.extra.speed = Math.max(Math.abs(this.speed.x), Math.abs(this.speed
                .y));
66          this.extra.ready = true;
67          this.extra.vertical = this.speed.x > this.speed.y;
68        }
69
70        if (this.extra.direction === 1) {
71          this.speed.x = this.extra.speed * Math.cos(this.extra.age * Math.PI /
                64);
72        } else {
73          this.speed.y = this.extra.speed * Math.cos(this.extra.age * Math.PI /
                64);
74        }
75
76        this.extra.age++;
77        this.position.x += this.speed.x;
78        this.position.y += this.speed.y;
79      }
80    };
81
82    return {
83      get: function (strategy) {
84        return strategies[strategy];
85      }
86    }
87  }());
```

Tot i que el codi pot intimidar una mica, fixeu-vos que només consisteix en un diccionari de dades al qual corresponen funcions que determinen una nova posició. Com és d'esperar, es tracta d'una funció autoinvocada i per tant només exposa el

mètode `get`, que permet obtenir la funció corresponent a l'estratègia de moviment amb aquest nom (vegeu la figura 1.20).

FIGURA 1.20. Diagrama UML dipòsit d'estratègies



Fixeu-vos que tots dos dipòsits disposen d'un diccionari de dades privat (objecte de JavaScript que fa servir una cadena de text com a clau) per emmagatzemar les dades i un mètode `get` públic per recuperar-les.

1.3.4 Interacció amb l'aplicació

Els dos dispositius més habituals per interactuar amb una aplicació són el ratolí i el teclat. Per estrany que pugui semblar, tots dos poden ser problemàtics a l'hora de treballar amb ells donades les discrepàncies que es poden trobar entre els navegadors.

En el cas del ratolí, la dificultat més gran que es presenta en treballar amb l'element `canvas` és determinar sobre quin punt es troba realment el cursor, ja que no hi ha cap mètode 100% fiable per accedir a aquesta informació i cada navegador implementa diferents solucions.

El més recomanable és treballar amb les propietats `offsetX` i `offsetY`, ja que tot i que són experimentals formen part de les recomanacions del W3C. Aquestes propietats retornen la posició del cursor respecte a l'element clicat, que correspondrà al `canvas`.

Quant a l'ús del teclat, hi ha problemes similars. Cada navegador ha realitzat la implementació d'una manera diferent i no es garanteix que totes les tecles responguin de la mateixa manera als *events* `keydown`, `keyup` i `keypress`. Tampoc no es retornen les mateixes propietats amb l'*event*, de manera que alguns navegadors retornen el codi a la propietat `keyCode` mentre que d'altres utilitzen `charCode`.

Podeu trobar més informació sobre les propietats dels *events* disparats pel ratolí a l'enllaç següent: goo.gl/t0ijCe.

Per unificar aquestes entrades, la solució és crear un objecte que centralitzi l'entrada dels dispositius i que l'aplicació consulti a aquest objecte en lloc de gestionar directament els *events*. Aquest objecte també pot encarregar-se de gestionar les diferències entre navegadors, de manera que tots els canvis que calgui fer sobre aquest aspecte es poden fer des d'un mateix punt.

Per exemple, a *IOC Invaders* això s'aconsegueix a través de l'objecte `inputController`, una funció autoinvocada que encapsula la complexitat de detectar quines tecles s'han premut, i exposa només dues propietats que permeten consultar l'estat de les tecles fent servir cadenes de text: `space`, `left`, `up`, `right` i `down`:

```
var inputController = (function () {
  var KEY_CODES = {
    32: 'space',
    37: 'left',
    38: 'up',
    39: 'right',
    40: 'down'
  },

  KEY_STATUS = {};

  for (var code in KEY_CODES) {
    KEY_STATUS[KEY_CODES[code]] = false;
  }

  document.onkeydown = function (e) {
    var keyCode = (e.keyCode) ? e.keyCode : e.charCode;

    if (KEY_CODES[keyCode]) {
      e.preventDefault();
      KEY_STATUS[KEY_CODES[keyCode]] = true;
    }
  };

  document.onkeyup = function (e) {
    var keyCode = (e.keyCode) ? e.keyCode : e.charCode;

    if (KEY_CODES[keyCode]) {
      e.preventDefault();
      KEY_STATUS[KEY_CODES[keyCode]] = false;
    }
  };
  return {
    KEY_CODES: KEY_CODES,
    KEY_STATUS: KEY_STATUS
  }
})();
```

Com es pot apreciar, s'utilitzen dos diccionaris:

- **KEY_CODES**: relaciona el codi corresponent a cada tecla amb un nom més entenedor pel desenvolupador.
- **KEY_STATUS**: emmagatzema l'estat de cada tecla (`true` si s'ha premut o `false` en cas contrari).

Aquests diccionaris són exposats com a propietats públiques de l'objecte, mentre que la implementació que gestiona els canvis queda amagada gràcies a la clausura creada per la funció autoinvocada. D'aquesta manera és possible consultar l'estat de les tecles, però no poden modificar-se externament.

Exposar una propietat o mètode

En programació orientada a objectes, *exposar* fa referència a mètodes o propietats a les quals es pot accedir des d'altres punts del programari, i la resta de la implementació queda inaccessible.

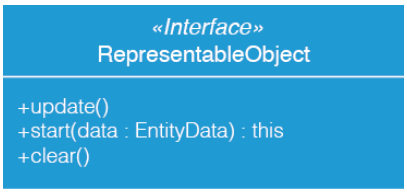
Per consultar l'estat concret d'un botó, només cal consultar el valor del diccionari `KEY_STATUS`. Per exemple, per saber l'estat de la tecla corresponent a la fletxa esquerra (anomenada `left` en el codi), només cal consultar l'estat del diccionari de la manera següent: `inputController.KEY_STATUS.left`. Aquesta consulta retornarà `true` o `false` segons si s'ha premut el botó o no.

1.3.5 Elements representables al joc

Tots els jocs tenen elements que han de ser representats a la pantalla: fons del joc, jugadors, peces d'un puzle, enemics, bales, efectes... Tots aquests elements tenen una característica en comú: han de poder mostrar-se a la pantalla i han de poder-se eliminar.

Per simplificar la implementació pot utilitzar-se una interfície (vegeu la figura 1.21), que serà implementada per cadascun d'aquests elements. S'ha de disposar d'un mètode d'actualització (*update* en anglès), que serà cridat cada vegada que es redibuixa el `canvas`, un mètode per inicialitzar l'element (*start* en anglès) i un mètode per netejar (*clear* en anglès) quan calgui eliminar-lo.

FIGURA 1.21. Diagrama UML interfície RepresentableObject



Cal destacar que JavaScript no admet l'ús d'interfícies i, per tant, `RepresentableObject` no es troba definit al codi del joc, però forma part del disseny de l'aplicació.

Podeu trobar més informació sobre la utilització de l'element `canvas` a l'enllaç següent: goo.gl/GV9awc.

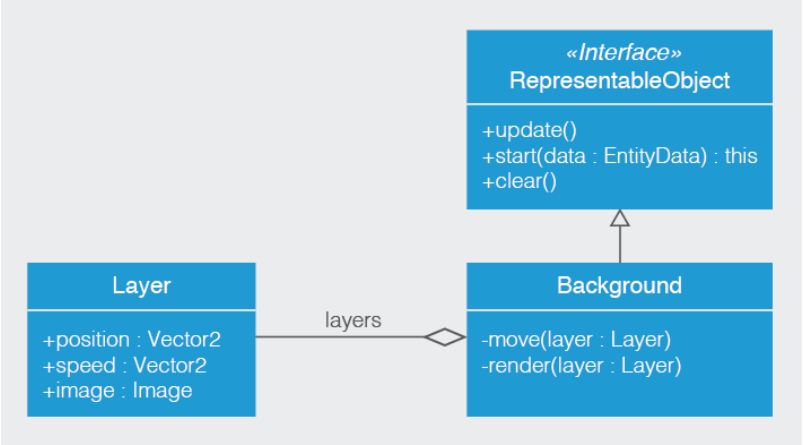
L'element `canvas` permet dibuixar tant gràfics en 2D com en 3D mitjançant l'API WebGL, tot i que aquest últim sistema no es fa servir gaire perquè és molt més complex. Tot i així, hi ha motors de jocs com Unity que permeten exportar jocs en 3D i sí que aprofiten aquesta API.

El context 2D del `canvas` permet escriure textos i dibuixar formes o imatges completes carregades a la memòria (fitxers d'imatge, per exemple). S'ha de tenir en compte que el contingut és estàtic, és a dir, els continguts no són animats, es crea aquesta il·lusió reescrivint-los en diferents posicions o canviant la imatge dibuixada.

`Vector2` és una estructura de dades que representa un vector de dues dimensions amb les propietats: `x` i `y`.

Per exemple, per dibuixar el fons del joc *IOC Invaders*, s'utilitza l'objecte generat per `backgroundConstructor`, que emmagatzema la informació sobre 4 imatges que es dibuixen en ordre com si fossin capes (vegeu la figura 1.22). La primera capa que es dibuixa queda al fons de la pila, la segona capa es dibuixa a sobre, i així contínuament.

Figura 1.22. Diagrama UML objecte de tipus Background i les seves associacions



A continuació podeu veure el codi del generador d'objectes de tipus `Background` utilitzat a *IOC Invaders*, que inclou un fons amb desplaçament infinit amb 4 nivells de profunditat:

```
1  var backgroundConstructor = function () {
2    var that = {},
3      layers = {};
4
5    function move(layer) {
6      layer.position.x += layer.speed.x;
7      layer.position.y += layer.speed.y;
8
9      if (layer.position.x < -layer.image.width || layer.position.x > layer.image
           .width) {
10       layer.position.x = 0;
11     }
12
13     if (layer.position.y < -layer.image.height || layer.position.y > layer.
           image.height) {
14       layer.position.y = 0;
15     }
16
17   }
18
19   function render(layer) {
20     // Imatge actual
21     gameContext.drawImage(
22       layer.image, layer.position.x, layer.position.y, layer.image.width, layer
             .image.height);
23
24       // Imatge següent
25       gameContext.drawImage(
26         layer.image, layer.position.x + layer.image.width - 1,
27         layer.position.y, layer.image.width, layer.image.height);
28     }
29
30     that.update = function () {
31       for (var i = 0; i < layers.length; i++) {
32         move(layers[i]);
33         render(layers[i]);
34       }
35     };
36
37     that.start = function (data) {
38       layers = data.layers;
39
40       for (var i = 0; i < layers.length; i++) {
41         layers[i].image = assetManager.getImage(layers[i].id);
42         layers[i].position = {x: 0, y: 0}
```

```
43        }
44      };
45
46      that.clear = function () {
47        layers = {};
48        speed.reset();
49      };
50
51    return that;
52  };
```

Com podeu veure, per construir l'objecte s'ha fet servir el patró funcional. És a dir, la funció `backgroundConstructor` retorna un objecte "Background". Aquest objecte exposa uns mètodes públics (afegits a l'objecte `that`, que és retornat) i encapsula les propietats (declarades mitjançant la paraula `var` al principi de la funció) i mètodes privats (declarats amb la paraula clau `function`) per distingir-los més clarament.

Cada vegada que s'invoca el mètode `update`, primer s'invoca el mètode privat `move`, que desplaça la posició de cada capa segons la velocitat establerta i, seguidament, s'invoca el mètode `render`, que les redibuixa en la nova posició mitjançant el mètode del context `drawImage`, que requereix unes coordenades, la mida i la imatge a dibuixar.

Fixeu-vos que es dibuixa la mateixa imatge dos cops, una a continuació de l'altra. D'aquesta manera s'aconsegueix un efecte de bucle, ja que no es pot apreciar on acaba una i on comença la següent. Aquest efecte es podria millorar incloent imatges més llargues o fent servir un *array* d'imatges que anés alternant-se.

En aquest cas, com que la imatge de fons ocupa tot el `canvas` no cal netejar-lo, però hi ha casos en què és convenient utilitzar el mètode `clearRect` del context per eliminar el contingut anterior (totalment o parcialment).

Cal destacar que les imatges utilitzades s'obtenen d'un gestor de recursos (l'objecte `assetManager`) mitjançant el mètode `getImage`. D'aquesta manera, l'objecte `Background` no necessita saber com ni on s'ha generat, només ha de passar al mètode l'identificador de la imatge que necessita.

Sprites

Un altre element clau per representar a la pantalla són els *sprites*. S'acostuma a utilitzar el mateix terme, sigui una sola imatge o un conjunt que forma una animació. En el cas de constar de múltiples imatges, poden dibuixar-se seqüencialment per simular diferents moviments. Per exemple, la nau que es mostra a la figura 1.23 simula l'efecte dels propulsors fent augmentar i disminuir la flama.

Figura 1.23. Sprite d'un alien de tipus A del joc "IOC Invaders"

Un ***sprite*** és un mapa de bits bidimensional utilitzats per primera vegada a les consoles i ordenadors de 8 bits.

Es pot trobar més informació sobre els *sprites* a l'enllaç següent: goo.gl/55RIoP.

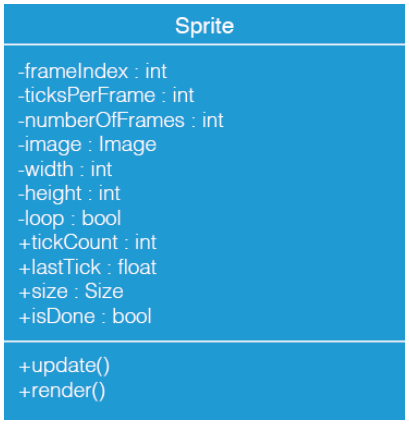
En jocs més avançats, cada personatge o objecte pot tenir associats múltiples *sprites*, de manera que pot haver-hi diferents animacions per a les accions que es poden portar a terme: atacar, caminar, córrer, saltar... I, per descomptat, tots els elements poden utilitzar-los: el personatge del jugador, enemics, bales, explosions, decoracions... Només cal fer una ullada als jocs d'ordinadors i videoconsoles de 8 i 16 bits (fins i tot de màquines recreatives) per veure'n les possibilitats.

Si un joc corre a 60 FPS, es mostrarà una imatge cada 16 mil·lisegons aproximadament.

La gestió d'aquests elements és més complicada que en el cas dels fons perquè requereixen càlculs addicionals per determinar si s'han de reproduir infinitament i quant de temps s'ha de mostrar cada imatge que compon l'animació.

A la figura 1.24 podeu veure el diagrama corresponent a un objecte `Sprite` generat per la funció generadora `spriteConstructor` del joc *IOC Invaders*, corresponent al codi següent:

Figura 1.24. Diagrama UML objecte de tipus Sprite




```
1  var spriteConstructor = function (options) {
2
3    var that = {},
4      frameIndex = 0,
5      ticksPerFrame = options.ticksPerFrame || 0,
6      numberOfFrames = options.numberOfFrames || 1,
7      image = options.image,
8      width = image.width,
9      height = image.height,
10     loop = options.loop === undefined ? true : options.loop;
11
12     that.tickCount = 0;
13     that.lastTick = gametime;
14     that.position = options.position || {x: 0, y: 0};
15     that.size = {width: width / numberOfFrames, height: height};
16     that.isDone = false;
17
18     that.update = function () {
19
20       if (updatedSprites.contains(that)) {
21         return;
22       } else {
23         updatedSprites.push(that);
24       }
```

```
25
26      that.tickCount++;
27      that.lastTick = gametime;
28
29      if (that.tickCount > ticksPerFrame) {
30        that.tickCount = 0;
31
32        if (frameIndex < numberOfFrames - 1) {
33          frameIndex += 1;
34        } else {
35          that.isDone = !loop;
36          frameIndex = 0;
37        }
38      }
39    };
40
41    that.render = function () {
42
43      if (that.isDone) {
44        return;
45      }
46
47      gameContext.drawImage(
48        image,
49        frameIndex * width / numberOfFrames,
50        0,
51        width / numberOfFrames,
52        height,
53        that.position.x,
54        that.position.y,
55        width / numberOfFrames,
56        height);
57    };
58
59  return that;
60 };
```

Com es pot apreciar, la tasca principal d'aquesta classe és determinar si s'ha completat l'animació i si cal repetir-la i dibuixar la imatge que correspongui. Fixeu-vos que en el cas que una imatge contingui una animació, aquesta no es dibuixa completa sinó que només se'n dibuixa el fragment corresponent.

En aquest cas, per simplificar, s'ha considerat que totes les imatges que formen l'animació tenen la mateixa mida; així doncs, només cal saber quin és el `frame` intern que cal dibuixar i calcular-ne la posició dins de la imatge.

Fixeu-vos que en el mètode `update` es comprova primer si aquest *sprite* ja ha estat actualitzat (aquesta propietat correspon al mòdul del joc) i, en cas contrari, s'afegeix a l'*array* abans de continuar amb l'actualització. Aquesta és una optimització que s'ha aplicat a aquest joc per evitar que s'actualitzin múltiples vegades per *frame* alguns elements.

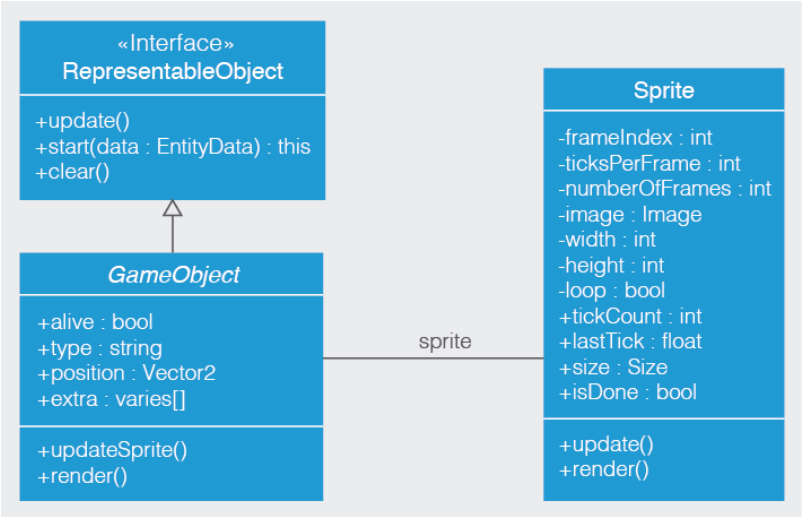
Objectes de joc: 'GameObject'

Alguns motors de desenvolupament de jocs, com Unity, utilitzen una classe especial d'objectes anomenats *objectes de joc* (*game objects* en anglès). Aquests objectes són elements que poden ser representats a la pantalla i que han de mantenir-se actualitzats contínuament. Això s'aconsegueix invocant el seu mètode `update` (entre d'altres) a cada iteració del bucle principal del joc.

Per exemple, un objecte de tipus bala canviarà la seva posició, actualitzarà l'animació de l'*sprite* que el representa, comprovarà si ha impactat a un adversari i si ha sortit de la pantalla; un enemic, a més a més, determinarà si cal disparar de nou, i una explosió només actualitzarà l'animació fins que aquesta acabi i desaparegui.

Tot i que els objectes de tipus `Background` també es poden considerar objectes de joc (corresponents als objectes `GameObject` que podeu veure a la figura 1.25), a *IOC Invaders* s'ha decidit diferenciar-los per generalitzar propietats comunes a: explosions, bales i naus.

Figura 1.25. Diagrama UML objecte de tipus GameObject i les seves relacions



Vegeu a continuació la implementació del generador d'objectes de tipus `Constructor` del joc *IOC Invaders*:

```
1  var gameObjectConstructor = function (options) {
2    var that = {
3      alive: false,
4      type: null,
5      position: null,
6      sprite: null,
7      extra: {}
8    };
9
10   that.updateSprite = function () {
11     that.sprite.position = that.position;
12     that.sprite.update();
13   };
14
15   that.render = function () {
16     that.sprite.render()
17   };
18
19   that.start = function (data) {
20     console.error("Error. Aquest mètode no està implementat");
21     return that;
22   };
23
24   that.update = function () {
25     console.error("Error. Aquest mètode no està implementat");
26   };
27
28   that.clear = function () {
```

```
    console.error("Error. Aquest mètode no està implementat");
  };

  return that;
};
```

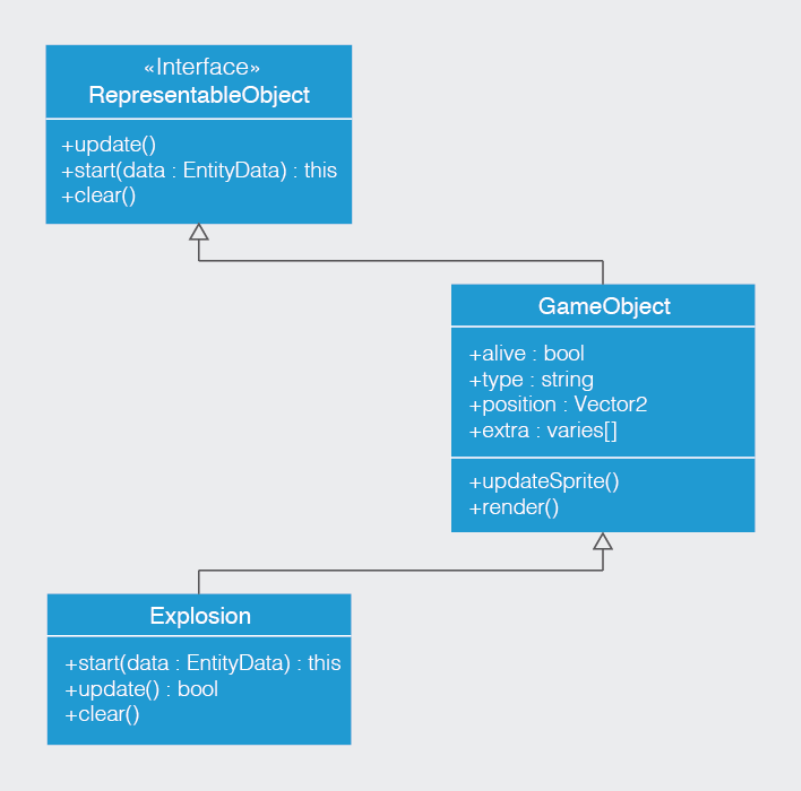
Cal destacar que `GameObject` s'ha de considerar una classe abstracta (tot i que a JavaScript no hi ha aquesta distinció). És a dir, no s'ha d'instanciar directament sinó que altres objectes l'utilitzen com a classe pare o superclasse.

Tipus especialitzats de 'GameObject' a "IOC Invaders"

Tot i que això no sempre és necessari, a *IOC Invaders* s'han generat múltiples generadors d'objectes per aprofitar la reutilització del codi. D'aquesta manera l'estructura és molt més clara i s'evita la duplicació de codi.

Per una banda, es van diferenciar les explosions d'altres elements del joc, ja que les característiques de les explosions són idèntiques a les de `GameObject` i es limita a implementar els mètodes `update`, `start` i `clear` de la interfície `RepresentableObject`, tal com es mostra a la figura 1.26. Podeu veure la implementació en el següent bloc codi:

Figura 1.26. Diagrama UML objecte de tipus Explosion i les seves relacions

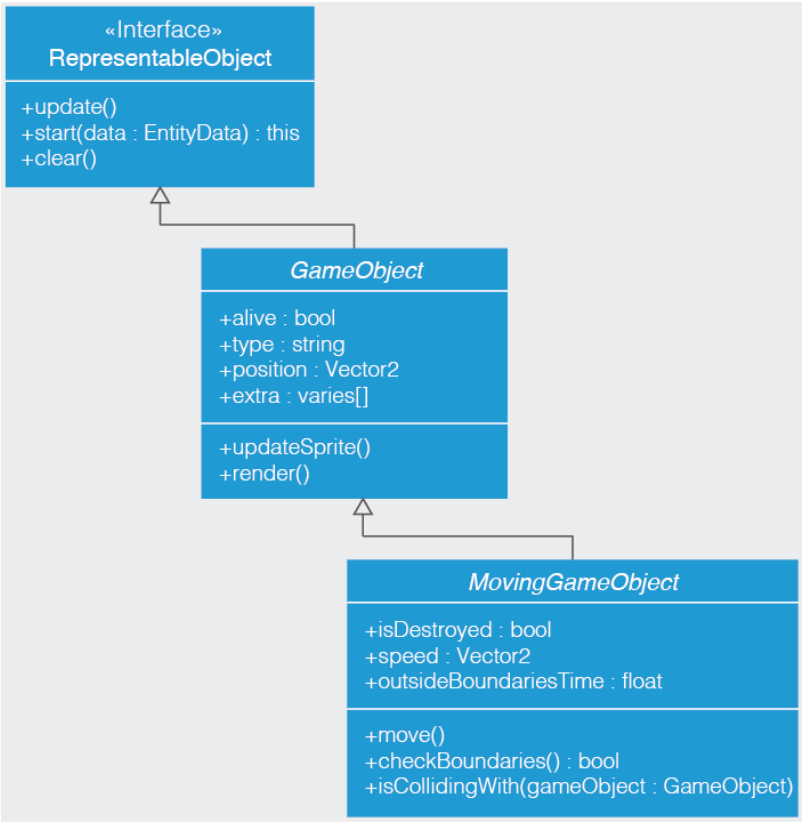



```
var explosionConstructor = function (options) {
  var that = gameObjectConstructor(options);

  that.start = function (data) {
    that.alive = true;
    that.type = data.type;
    that.position = data.position;
```

```
    that.sprite = assetManager.getSprite(data.sprite);
    that.sprite.isDone = false;
    assetManager.getSound(data.sound);
    return that;
  };

  that.clear = function () {
    that.alive = false;
    that.type = null;
    that.position = {x: 0, y: 0};
    that.sprite = null;
  };

  that.update = function () {
    if (that.sprite.isDone) {
      return true;
    }

    that.updateSprite();
    that.render();
  };

  return that;
};
```

Per altra banda, es troben els objectes que s'han de moure i, per conseqüent, han de controlar-se si es troben fora de la pantalla o si han col·lidit. Per generalitzar les característiques comunes d'aquest tipus d'objectes a *IOC Invaders* s'ha afegit un generador d'objectes de tipus `MovingGameObject` (veieu la figura 1.27) amb el codi que podeu trobar a continuació:

Figura 1.27. Diagrama UML objecte de tipus MovingGameObject i les seves relacions




```
var movingGameObjectConstructor = function (options) {
  var that = gameObjectConstructor(options);
```

```
3
4   that.isDestroyed = false;
5   that.speed = null;
6   that.outsideBoundariesTime = 0;
7
8   that.move = function () {
9     console.error("Error. Aquest mètode no està implementat");
10  };
11
12  that.checkBoundaries = function () {
13    if (this.position.x >= gameCanvas.width
14      || this.position.x <= -this.sprite.size.width
15      || this.position.y > gameCanvas.height
16      || this.position.y < -this.sprite.size.height) {
17        this.outsideBoundariesTime++;
18    } else {
19      this.outsideBoundariesTime = 0;
20    }
21
22    return that.outsideBoundariesTime >= MAX_TIME_OUTSIDE_BOUNDARIES;
23  };
24
25  that.isCollidingWith = function (gameObject) {
26    return (this.position.x < gameObject.position.x + gameObject.sprite.size.
          width
27      && this.position.x + this.sprite.size.width > gameObject.position.x
28      && this.position.y < gameObject.position.y + gameObject.sprite.size.
            height
29      && this.position.y + this.sprite.size.height > gameObject.position.y);
30  };
31
32  return that;
33 };
```

Fixeu-vos que els objectes generats s'han de tractar com si fossin abstractes, ja que requereix la implementació de la funció `move` a més a més dels mètodes `update`, `start` i `clear` de la interfície `RepresentableGameObject`.

Com es pot apreciar, els objectes que siguin herència del tipus `MovingGameObject` inclouen, a més del mètode per moure's, els mètodes `checkBoundaries` i `iscollisingWith` per comprovar si l'objecte del joc es troba dins dels límits de la pantalla o si ha col·lidit amb un altre objecte.

La resta de tipus d'objectes segueixen el mateix sistema, només cal destacar que tant els enemics com els jugadors són herència del mateix tipus d'objectes (el tipus `SpaceShip`), amb la diferència que les naus enemigues disparen aleatòriament i la nau del jugador es controla amb el teclat. A la figura 1.28 podeu veure la jerarquia completa d'objectes representables, i a continuació, el codi generador dels objectes de tipus `Shot`, `SpaceShip` i `Player`:

```
1  var shotConstructor = function (options) {
2    var that = movingGameObjectConstructor(options);
3
4    that.start = function (data) {
5      that.alive = true;
6      that.type = data.type;
7      that.position = data.position;
8      that.sprite = assetManager.getSprite(data.sprite);
9      assetManager.getSound(data.sound);
10     that.speed = data.speed;
11
12     // Dades i funcions específiques de cada tipus d'enemic
13     that.extra = data.extra || {};
```

```
    that.move = data.move.bind(that);
    that.outsideBoundariesTime = 0;

    return that;
  };

  that.clear = function () {
    that.isDestroyed = false;
    that.alive = false;
    that.outsideBoundariesTime = 0;
    that.type = null;
    that.position = {x: 0, y: 0};
    that.sprite = null;
    that.speed = {x: 0, y: 0};

    // Dades i funcions específiques de cada tipus d'enemic
    that.extra = {};
    that.move = null;
  };

  that.update = function () {
    if (that.isDestroyed) {
      return true;
    }

    that.updateSprite();
    that.move();

    if (that.checkBoundaries()) {
      return true;
    }

    that.render();
  };

  return that;
};

var spaceshipConstructor = function (options) {
  var that = movingGameObjectConstructor(options);

  that.bulletPool = options.pool.bullet;
  that.explosionPool = options.pool.explosion;

  that.fire = function () { // @protected
    for (var i = 0; i < that.cannon.length; i++) {
      that.shoot(that.cannon[i]);
    }
  };

  that.shoot = function (cannon) { // @protected
    var origin;

    if (Math.random() < cannon.fireRate / 100) {
      origin = {x: that.position.x + cannon.position.x, y: that.position.y +
          cannon.position.y};
      that.bulletPool.instantiate(cannon.bullet, origin, cannon.direction);
    }
  };

  that.start = function (data) {
    that.alive = true;
    that.type = data.type;
    that.position = data.position;
    that.sprite = assetManager.getSprite(data.sprite);
    that.cannon = data.cannon;
    that.explosion = data.explosion;
    that.speed = data.speed;
    that.points = data.points;
    that.outsideBoundariesTime = 0;
```

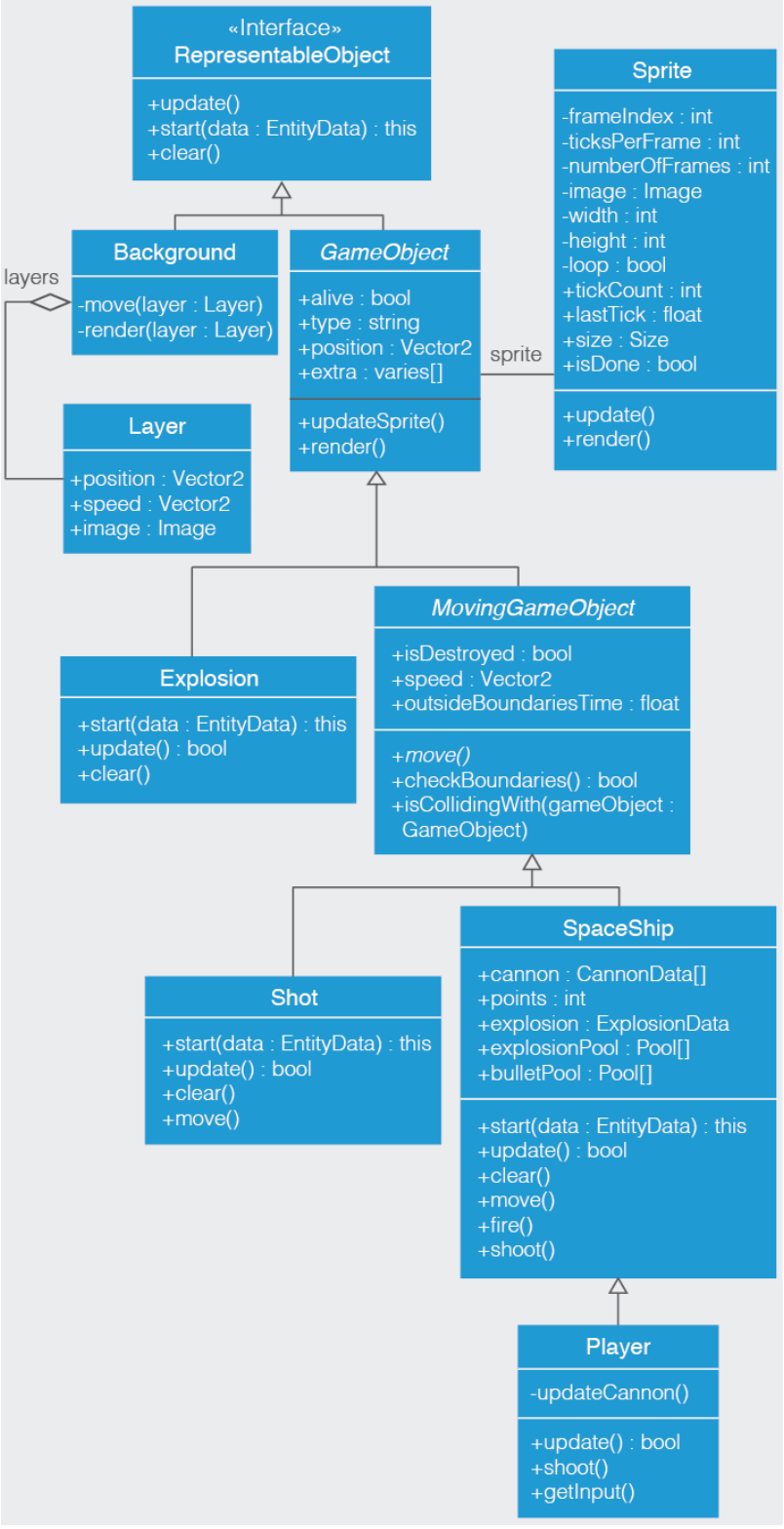
```
83
84      // Dades i funcions específiques de cada tipus d'enemic
85      that.extra = data.extra || {};
86
87      if (data.move) {
88        that.move = data.move.bind(that);
89      }
90
91      return that;
92    };
93
94    that.clear = function () {
95      that.isDestroyed = false;
96      that.alive = false;
97      that.type = null;
98      that.position = {x: 0, y: 0};
99      that.sprite = null;
100     that.speed = {x: 0, y: 0};
101     that.points = 0;
102     that.cannon = null;
103
104     // Dades i funcions específiques de cada tipus d'enemic
105     that.extra = null;
106     that.move = null;
107     that.outsideBoundariesTime = 0;
108   };
109
110   that.update = function () {
111
112     if (that.isDestroyed) {
113       that.explosionPool.instantiate(that.explosion, that.position);
114       return true;
115     }
116
117     that.updateSprite();
118     that.move();
119     that.fire();
120
121     if (that.checkBoundaries()) {
122       return true;
123     }
124
125     that.render();
126   };
127
128   return that;
129 };
130
131 var playerConstructor = function (options) {
132   var that = spaceshipConstructor(options);
133
134   function getInput() {
135     if (inputController.KEY_STATUS.left) {
136       that.position.x -= that.speed.x;
137     } else if (inputController.KEY_STATUS.right) {
138       that.position.x += that.speed.x;
139     }
140
141     if (inputController.KEY_STATUS.up) {
142       that.position.y -= that.speed.y;
143     } else if (inputController.KEY_STATUS.down) {
144       that.position.y += that.speed.y;
145     }
146
147     if (inputController.KEY_STATUS.space && !that.isDestroyed) {
148       that.fire();
149     }
150
151     // S'evita que surti de la pantalla
152     that.position.x = that.position.x.clamp(0, gameCanvas.width - that.sprite.
```

```
        size.width);
153     that.position.y = that.position.y.clamp(0, gameCanvas.height - that.sprite.
            size.height);
154
155   }
156
157   function updateCannon() {
158     for (var i = 0; i < that.cannon.length; i++) {
159       if (that.cannon[i].lastShot === undefined) {
160         that.cannon[i].lastShot = that.cannon[i].fireRate + 1;
161       }
162       that.cannon[i].lastShot++;
163     }
164   };
165
166   that.start(entitiesRepository.get('player', options.position, options.speed))
          ;
167
168   that.shoot = function (cannon) {
169     var origin;
170     if (cannon.lastShot > cannon.fireRate) {
171       cannon.lastShot = 0;
172       origin = {x: that.position.x + cannon.position.x, y: that.position.y +
              cannon.position.y};
173       that.bulletPool.instantiate(cannon.bullet, origin, cannon.direction);
174     }
175   };
176
177   that.update = function () {
178     updateCannon();
179
180     // Si ha sigut impactat, s'elimina. Aquí també es podria afegir l'animació
            de l'explosió
181     if (this.isDestroyed) {
182       that.explosionPool.instantiate(that.explosion, that.position);
183       return true;
184     }
185
186     getInput();
187     this.updateSprite();
188     this.render();
189   };
190
191   return that;
192 };
```

Com podeu veure, aquests objectes inclouen la utilització d'objectes de tipus `PoolGameObject` (`explosionPool` i `bulletPool`). Aquests objectes són un tipus especial de col·lecció que permet reduir la necessitat de crear objectes nous reutilitzant els que es troben al *pool* (conjunt d'objectes) en lloc de crear-ne de nous. Així, quan un objecte ja no és necessari (per exemple, una bala que surt de la pantalla o una nau que és destruïda), aquests objectes són marcats com a disponibles al *pool* i poden tornar a utilitzar-se quan calgui instanciar un nou objecte d'aquest tipus.

Fixeu-vos que la finalitat de cada objecte és molt clara tot i que aquesta jerarquia és força extensa: inclou classes abstractes i una interfície. A més a més, s'ha pogut evitar la duplicació de codi en fer les implementacions dels mètodes comuns a les classes pare o delegant-les a altres objectes (per exemple, tot el comportament relatiu als *sprites* és gestionat pels objectes `Sprite`).

Figura 1.28. Diagrama UML de la jerarquia completa de RepresentableObject



Cal destacar que quan a un diagrama apareix un mètode que ja es troba a una classe pare és perquè aquest mètode el sobreescriu (*override* en anglès). És a dir, quan s'invoca aquest mètode el codi executat serà el de la classe concreta on s'ha fet la invocació i no pas el de la classe pare.

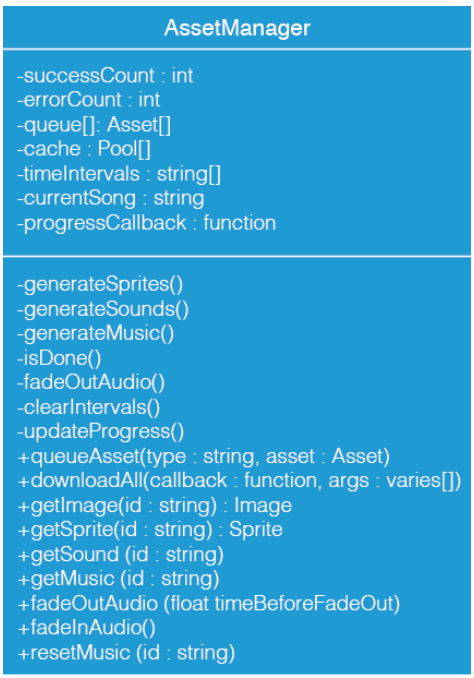
1.3.6 Gestor de recursos

Habitualment, un joc requerirà utilitzar imatges, *sprites*, efectes de so, música... A aquests elements se'ls coneix en conjunt com a recursos. Com que aquests recursos poden ser requerits per molts objectes diferents és recomanable utilitzar un objecte amb la funció de gestionar-los. Per exemple, un mateix efecte de so pot ser utilitzat per múltiples explosions al mateix temps i tots els enemics del mateix tipus faran servir el mateix *sprite*.

A JavaScript és especialment important fer-ho així, perquè s'han d'implementar mecanismes especials (com els gestors de descàrrega) per controlar quan s'han acabat de descarregar i inicialitzar tots els recursos, ja que la descàrrega es fa de manera asíncrona i l'execució del codi podria continuar abans d'haver descarregat totes les imatges per exemple.

A *IOC Invaders* la implementació del gestor de recursos (podeu veure el diagrama UML de l'`AssetManager` a la figura 1.29) inclou la gestió de descàrregues (només es controlen les imatges), la generació dels objectes necessaris, la funcionalitat d'obtenció de recursos i la reproducció d'àudio. Això s'ha fet així per simplificar el codi però és recomanable que cada objecte només tingui una responsabilitat.

FIGURA 1.29. Diagrama UML de la jerarquia completa de RepresentableObject



Com que en aquest objecte s'han barrejat múltiples funcionalitats, a continuació podeu trobar un resum dels mètodes i propietats dividits segons la seva utilitat:

- Gestió de descàrregues:
  - **queue**: llista d'elements per descarregar o generar.

  - **successCount** i **errorCount**: comptadors de recursos descarregats amb èxit o error.
  - **isDone**: mètode que determina si ja s'ha finalitzat la descàrrega dels recursos.
  - **progressCallback**: funció cridada per mostrar el progrés de la descàrrega.
  - **updateProgress**: mètode encarregat d'actualitzar el progrés de la descàrrega.
  - **queuesAsset**: mètode per afegir recursos a la llista de descàrrega.
  - **downloadAll**: mètode per iniciar la descàrrega dels recursos i engegar la generació dels objectes.

- Generació dels objectes:
  - **cache**: és la col·lecció d'objectes generats.
  - **generateSprites**: genera múltiples còpies de cada *sprite* per ser utilitzades com a *pool*.
  - **generateSounds**: genera múltiples còpies de cada efecte de so per generar un *pool* i poder reproduir-lo més d'una vegada simultàniament.
  - **generateMusic**: genera un element d'àudio per a cada objecte (només se'n necessita un en cada moment).

- Servei de recursos:
  - **getImage**: retorna la imatge indicada.
  - **getSprite**: retorna el primer *sprite* lliure del tipus indicat al *pool*.

- Reproducció d'àudio:
  - **currentSong**: identifica la música que es reprodueix en un moment concret.
  - **getSound**: reprodueix el primer so del tipus indicat que es trobi lliure al *pool* corresponent.
  - **getMusic**: reprodueix el fitxer de música indicat i el reinicia si ja estava reproduint-se.
  - **resetMusic**: atura la música i la reinicialitza.
  - **fadeInAudio**: abaixa el volum de tots els efectes de so a 0.
  - **fadeOutAudio**: restaura el volum de tots els efectes de so.
  - **timeIntervals**: són els temporitzadors utilitzats pel sistema de baixada i restauració del volum dels sons actius.
  - **clearInterval**: elimina els temporitzadors actius.

Com que l'única raó per la qual es "demana" un recurs d'àudio és per reproduir-lo, a *IOC Invaders* s'ha decidit reproduir-los automàticament en lloc de retornar-los quan s'invoquen els mètodes `getSound` i `getMusic`.

A continuació podeu trobar el codi complet del generador d'objectes de tipus `AssetManager`:

```
var assetManagerConstructor = function (progressCallback) {
    var that = {},
      successCount = 0,
      errorCount = 0,
      queue = {
        images: [],
        sprites: [],
        sounds: [],
        music: []
      },
      cache = {
        images: {},
        sprites: {},
        sounds: {},
        music: {}
      },
      timeIntervals = {},
      currentSong;

    function updateProgress() {
      if (progressCallback) {
        progressCallback(successCount + errorCount, queue.images.length);
      }
    }

    function generateSprites() {
      var pool, poolSize;

      for (var i = 0; i < queue.sprites.length; i++) {
        pool = [];
        poolSize = 10;

        for (var j = 0; j < poolSize; j++) {
          pool.push(spriteConstructor({
            image: that.getImage(queue.sprites[i].id),
            numberOfFrames: queue.sprites[i].numberOfFrames,
            ticksPerFrame: queue.sprites[i].ticksPerFrame,
            loop: queue.sprites[i].loop === undefined ? true : queue.sprites[i
                ].loop
          }));
        }

        cache.sprites[queue.sprites[i].id] = pool;
      }
    }

    function generateSounds() {
      var pool, poolSize, sound;

      for (var i = 0; i < queue.sounds.length; i++) {
        pool = [];
        poolSize = 10; // nombre màxim de sons idèntics que es reprodueixen al
            mateix temps
        for (var j = 0; j < poolSize; j++) {
          sound = new Audio(queue.sounds[i].path);
          sound.volume = queue.sounds[i].volume;
          pool.push(sound);
        }
        cache.sounds[queue.sounds[i].id] = {
          currentSound: 0,
          pool: pool,
          volume: queue.sounds[i].volume
        }
      }
    }

    function isDone() {
      return (queue.images.length == successCount + errorCount);
    }
```

```
68
69  function fadeOutAudio() {
70    for (var id in cache.sounds) {
71      for (var i = 0; i < cache.sounds[id].pool.length; i++) {
72        var sound = cache.sounds[id].pool[i];
73        sound.volume = 0;
74      }
75    }
76  }
77
78  function generateMusic() {
79    for (var i = 0; i < queue.music.length; i++) {
80      var sound = new Audio(queue.music[i].path);
81      sound.volume = queue.music[i].volume;
82      sound.loop = queue.music[i].loop;
83      cache.music[queue.music[i].id] = sound;
84    }
85  }
86
87  that.queueAsset = function (type, asset) {
88    queue[type].push(asset);
89  };
90
91  that.downloadAll = function (callback, args) {
92    if (queue.images.length === 0) {
93      callback();
94    }
95
96    for (var i = 0; i < queue.images.length; i++) {
97      var path = queue.images[i].path,
98        id = queue.images[i].id,
99        img = new Image();
100
101     img.addEventListener("load", function () {
102       successCount += 1;
103       updateProgress();
104
105       if (isDone()) {
106         generateSprites();
107         callback(args);
108       }
109     }, false);
110
111     img.addEventListener("error", function () {
112       errorCount += 1;
113       updateProgress();
114       if (isDone()) {
115         generateSprites();
116         callback(args);
117       }
118     }, false);
119
120     img.src = path;
121     cache.images[id] = img;
122   }
123
124   generateSounds();
125   generateMusic();
126 };
127
128 that.getImage = function (id) {
129   return cache.images[id];
130 };
131
132 that.getSprite = function (id) {
133   var pool = cache.sprites[id];
134   var sprite = pool[pool.length - 1];
135   pool.unshift(pool.pop());
136   return sprite;
137 };
```

```
138
139   that.getSound = function (id) {
140     var sounds = cache.sounds[id];
141
142     if (sounds.pool[sounds.currentSound].currentTime === 0
143       || sounds.pool[sounds.currentSound].ended) {
144         sounds.pool[sounds.currentSound].play();
145     }
146
147     sounds.currentSound = (sounds.currentSound + 1) % sounds.pool.length;
148   };
149
150   that.fadeOutAudio = function (timeBeforeFadeOut) {
151     that.clearIntervals();
152     if (timeBeforeFadeOut) {
153       timeIntervals.fadeOutAudio = setTimeout(fadeOutAudio, timeBeforeFadeOut);
154     } else {
155       fadeOutAudio();
156     }
157   };
158
159   that.fadeInAudio = function () {
160     that.clearIntervals();
161
162     for (var id in cache.sounds) {
163       for (var i = 0; i < cache.sounds[id].pool.length; i++) {
164         var sound = cache.sounds[id].pool[i];
165         sound.volume = cache.sounds[id].volume;
166       }
167     }
168   };
169
170   that.clearIntervals = function () {
171     clearInterval(timeIntervals.fadeInAudio);
172     clearInterval(timeIntervals.fadeOutAudio);
173   };
174
175   that.getMusic = function (id) {
176     that.resetMusic(currentSong);
177     cache.music[id].play();
178     currentSong = id;
179   };
180
181   that.resetMusic = function (id) {
182     if (!id) {
183       return;
184     }
185
186     if (!cache.music[id].ended) {
187       cache.music[id].pause();
188     }
189
190     if (cache.music[id].currentTime > 0) {
191       cache.music[id].currentTime = 0;
192     }
193   };
194
195   return that;
196 };
```

Podeu trobar més informació sobre el patró de disseny façana a l'enllaç següent: goo.gl/JU6JRh.

Habitualment, quan es necessita centralitzar diferents funcionalitats en un mateix objecte s'acostuma a aplicar el patró de disseny *façana* (*facade* en anglès). Aquest patró consisteix a crear una classe que exposa els mètodes públics necessaris però internament treballa amb múltiples classes diferents. D'aquesta manera el funcionament de la façana és transparent a l'usuari, com en aquest cas, que no ha de preocupar-se d'on prové realment un *sprite*, simplement invoca el mètode `getSprite` de l'`AssetManager` sense haver de saber-ne l'origen.

Cal destacar que aquest objecte utilitza un sistema de *pools* simplificat (es tracta d'un *array*), perquè no s'ha de controlar l'estat dels elements d'aquests *pools*.

1.3.7 Optimització: ús de 'pools'

Un dels objectius principals a l'hora de desenvolupar un joc és que el nombre de *frames* per segon sigui estable. Preferentment han de mantenir-se 60 *frames* per segon per aconseguir que el joc sigui fluid.

Cal tenir en compte que, a JavaScript, la destrucció dels objectes és gestionada pels navegadors, que de tant en tant inicien la recol·lecció de brossa per alliberar la memòria dels objectes que ja no s'estan fent servir. Això pot provocar caigudes crítiques en el nombre de *frames* per segon durant l'execució, i cal evitar-lo tant sí com no perquè durant l'execució del recol·lector de brossa l'aplicació pot quedar completament bloquejada.

Com que durant un joc en pocs minuts es poden crear un gran nombre d'objectes (per exemple les bales en un joc de trets), s'ha d'implementar un sistema que permeti reciclar aquests objectes. El més habitual és fer servir un *pool* (conjunt d'objectes).

Aquesta tècnica és força simple d'entendre i d'implementar. Consisteix a crear el màxim nombre d'objectes que s'hagin de mostrar en pantalla al mateix temps abans de començar el joc i emmagatzemar-los al *pool*. Llavors, en lloc de crear els objectes, s'agafen d'aquest *pool*, i quan no es necessiten es tornen a afegir al *pool*. Així es reciclen i no cal crear-ne més: això evita que es cridi el recol·lector de brossa sigui i que s'aturi l'execució del programa.

És a dir, si es requereix que a la pantalla hi hagi 500 bales actives es crearà un *pool* amb 500 bales. Quan calgui instanciar una bala, s'obté del *pool*, i una vegada surt de la pantalla o impacta en un adversari es retorna al *pool*.

Hi ha molts sistemes per determinar quin és el pròxim objecte a retornar, i depenent de les vostres necessitats podeu fer-ne servir un o altre. Per exemple, suposant que els objectes s'emmagatzemen en un *array* amb nom `pool`, es podria determinar de les maneres següents:

- Guardar en un comptador quin és el proper ítem a utilitzar (aquest és el sistema utilitzat al mètode `getSound` de l'`AssetManager`): `properIndex = (properIndex + 1) % pool.length`.
- Retornar sempre l'últim element i moure'l de l'última posició a la primera (aquest és el sistema utilitzat al mètode `getSprite` de l'`AssetManager` i l'objecte `GameObjectPool`), per exemple:

```
function getElement() {
  var element = pool[pool.length – 1];
  pool.unshift(pool.pop());
  return element;
}
```

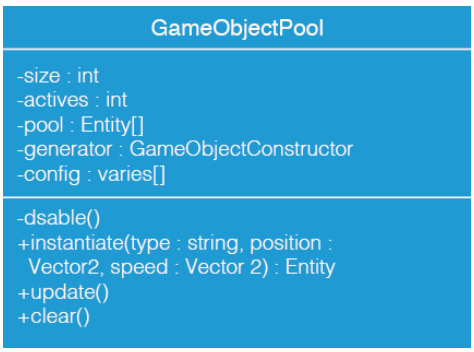
- Afegir una propietat a tots els objectes de tipus booleà `actiu`, canviar el valor a cert o fals (segons s'activi o es desactivi) i ordenar l'*array* segons aquesta propietat, de manera que tots els elements inactius es trobin al final de la llista, per exemple, i només calgui obtenir l'últim element, que sempre serà inactiu.
- Eliminar l'element de l'*array* (amb els mètodes `pop`, `shift` o `splice` segons la posició) quan sigui actiu i tornar a afegir-lo a l'*array* quan es desactivi.

L'objecte `GameObjectPool` del joc *IOC Invaders* combina dos sistemes (tots els mètodes pertanyen als *arrays*):

- En instanciar un objecte es mou de l'última posició (mètode `pop`) a la primera (mètode `unsfhit`).
- En desactivar un objecte s'elimina de la seva posició actual (mètode `splice`) i s'afegeix al final de l'*array* (mètode `push`).

El `GameObjectPool` permet netejar el *pool* (`clear`), instanciar nous objectes (`instantiate`) i actualitzar tots els `GameObjects` emmagatzemats (`update`), com es pot veure a la figura 1.30.

Figura 1.30. Diagrama UML objecte Pool



A continuació podeu trobar el codi del `GameObjectPool` implementat a l'*IOC Invaders*:

```
1  var gameObjectPoolConstructor = function (maxSize, generator, config) {
2    var that = {},
3    size = maxSize;
4
5    function disable(index) {
6      that.pool[index].clear();
7      that.pool.push((that.pool.splice(index, 1))[0]);
8    }
9
10   that.actives = size;
11   that.pool = [];
12
13   for (var i = 0; i < size; i++) {
14     that.pool[i] = generator(config);
15   }
16
17   that.instantiate = function (type, position, speed) {
```

```
18      var instance = that.pool[size - 1].start(entitiesRepository.get(type,
            position, speed));
19      that.pool.unshift(that.pool.pop());
20      return instance;
21    };
22
23    that.update = function () {
24      for (var i = 0; i < size; i++) {
25        // Només dibuixem fins que trobem un objecte que no sigui viu
26        if (that.pool[i].alive) {
27          if (that.pool[i].update()) {
28            // Si update ha retornat cert, és que s'ha de desactivar
29            disable(i);
30          }
31        } else {
32          that.actives = i;
33          break;
34        }
35      }
36    };
37
38    that.clear = function () {
39      for (var i = 0; i < size; i++) {
40        that.pool[i].alive = false;
41      }
42      that.actives = 0;
43    };
44
45    return that;
46  };
```

Com es pot apreciar, el generador d'objectes rep com a paràmetres `maxSize` (mida del *pool*), `generator` (generador d'objectes) i `config` (dades de configuració per generar els objectes). És a dir, es pot passar com a generador la funció `shotConstructor` per crear un *pool* de bales o la funció `spaceshipConstructor` per crear-ne un de naus enemigues.

La funció `clear` permet netejar el *pool* quan es reinicia el joc o es passa de nivell, mentre que la funció `update` és cridada des del bucle principal del joc a cada iteració i s'encarrega d'actualitzar cadascun dels elements del *pool*.

Fixeu-vos que en el cas de `update` només es recorren els elements fins a trobar-ne un que no estigui actiu (corresponent a la propietat `alive`), ja que els elements es troben ordenats i no cal continuar amb el bucle.

1.3.8 Motor del joc

Quan es parla del **motor del joc** cal distingir entre el motor implementat concretament per a un joc (per exemple el motor del *IOC Invaders*), i els motors de joc genèrics utilitzats per desenvolupar jocs (com Construct 2 o Game Canvas). En aquesta secció ens referim als primers, la implementació de motors de joc propis.

El motor del joc és el component responsable de portar a terme les tasques genèriques del joc, com poden ser, entre d'altres, les següents:

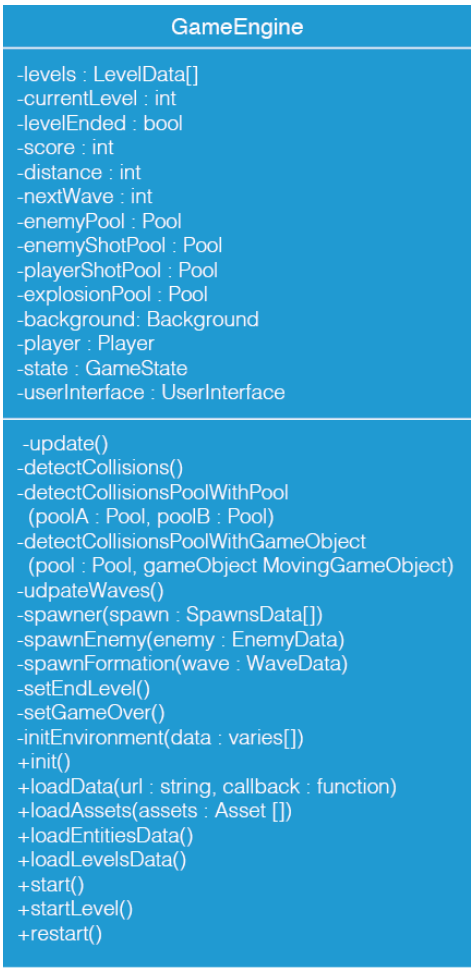
- Inicialitzar el joc.

- Gestionar la detecció de col·lisions.
- Carregar les dades del joc.
- Gestionar els canvis de pantalles.
- Executar el bucle principal del joc.

L'execució del bucle principal és especialment important, perquè una vegada el joc estigui inicialitzat, aquest bucle es repetirà a cada *frame* i serà el responsable d'actualitzar tots els elements del joc.

Com es pot apreciar a la figura 1.31 el motor del joc *IOC Invaders* (`GameManager`) es força complex, ja que inclou múltiples funcionalitats en un sol objecte.

FIGURA 1.31. Diagrama UML objecte GameManager

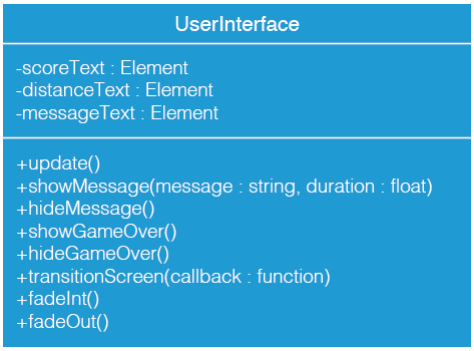


Interfície d'usuari

La interfície d'usuari del joc és l'encarregada de mostrar informació a l'usuari i permet interactuar-hi mostrant un menú d'opcions per iniciar el joc o les puntuacions. També es consideren elements de la interfície els indicadors que es poden fer servir per mostrar els punts de vida d'un enemic o un ressaltat al voltant d'un tresor.

Per simplificar-ho, a *IOC Invaders* només es mostra la puntuació, la distància recorreguda al nivell i un botó per reiniciar el joc quan acaba la partida. Com que es tracta d'un cas tan simple, s'ha optat per afegir l'objecte `UserInterface` (vegeu la figura 1.32) directament a l'objecte `GameEngine`, però no és el més recomanable.

Figura 1.32. Diagrama UML objecte UserInterface



A continuació podeu trobar un resum dels mètodes i la seva funció:

- **update**: actualitza la puntuació i la distància en pantalla.
- **showMessage** i **hideMessage**: mostra i amaga missatges respectivament, per exemple, en començar un nivell.
- **showGameOver** i **hideGameOver**: mostra i amaga el missatge de fi del joc (inclou el botó per tornar a començar mitjançant HTML).
- **transitionScreen**, **fadeIn**, **fadeOut**: fan efectes per enfosquir i tornar a mostrar la pantalla.

Càrrega de dades

Cal diferenciar entre la càrrega de dades i la càrrega de recursos. Normalment els recursos es corresponen amb elements d'HTML (àudio, vídeo i imatges) i es poden controlar quan ha finalitzat la càrrega detectant l'*event* `load`, ja que es tracta d'elements. En canvi, les dades s'han de carregar realitzant peticions AJAX i processant-ne la resposta.

En cas que els fitxers amb les dades es trobin en un servidor diferent, s'ha de tenir en compte que aquest ha d'implementar mecanismes CORS o JSONP per evitar el bloqueig dels navegadors provocat per la política del mateix origen.

Al motor del joc de l'*IOC Invaders* s'utilitzen les següents funcions per portar a terme la càrrega de dades:

- **loadData**: realitza la petició AJAX i invoca la funció passada com a argument per processar les dades rebudes en format JSON.
- **loadEntitiesData**: carrega les dades de les entitats i les afegeix al repositori.

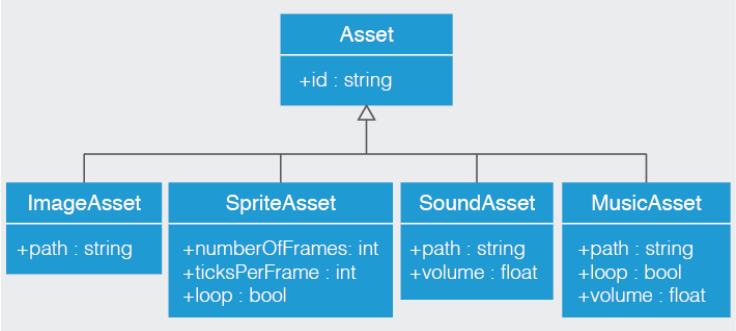
- **loadLevelsData**: carrega les dades del nivell i, en acabar, inicia el joc.
- **init** i **loadAssets**: el primer carrega les dades dels recursos i seguidament n'inicia la descàrrega mitjançant l' `AssetsManager`.

Totes les dades del joc són en format JSON, cosa que facilita la tasca d'editar-los manualment o crear un editor per modificar-los.

A continuació podeu trobar l'estructura del fitxer de recursos (`asset-data.json`) que correspon al diagrama de la figura 1.33:

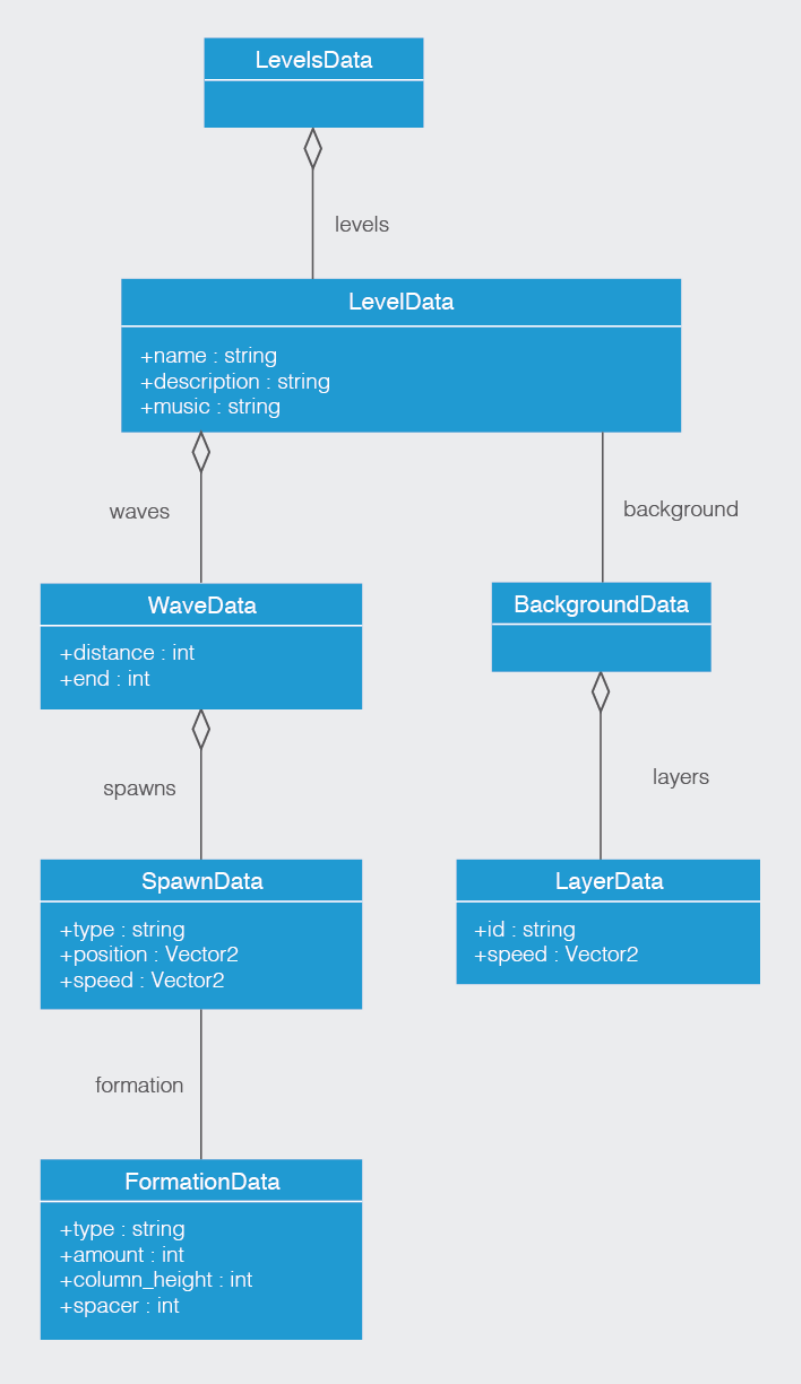
```
{
  "assets": {
      "images": [
        {
          "id": "identificador de la imatge",
          "path": "ruta de la imatge"
        }
      ],
      "sprites": [
        {
          "id": "identificador del sprite",
          "numberOfFrames": "nombre de frames que formen l'animació",
          "ticksPerFrame": "nombre de frames que es mostren a la mateixa imatge
               abans de canviar a la següent"
        }
      ],
      "sounds": [
        {
          "id": "identificador de l'efecte de so",
          "path": "ruta de l'efecte de so",
          "volume": "tipus float. Volumen al qual s'ha de reproduir"
        }
      ],
      "music": [
        {
          "id": "identificador del fitxer de música",
          "path": "ruta del fitxer de música",
          "loop": "tipus booleà. Indica si s'ha de repetir indefinidament"
          "volume": "tipus float. volumen al que s'ha de reproduir"
        }
      ]
  }
}
```

Figura 1.33. Estructura de dades de "asset-data.json"



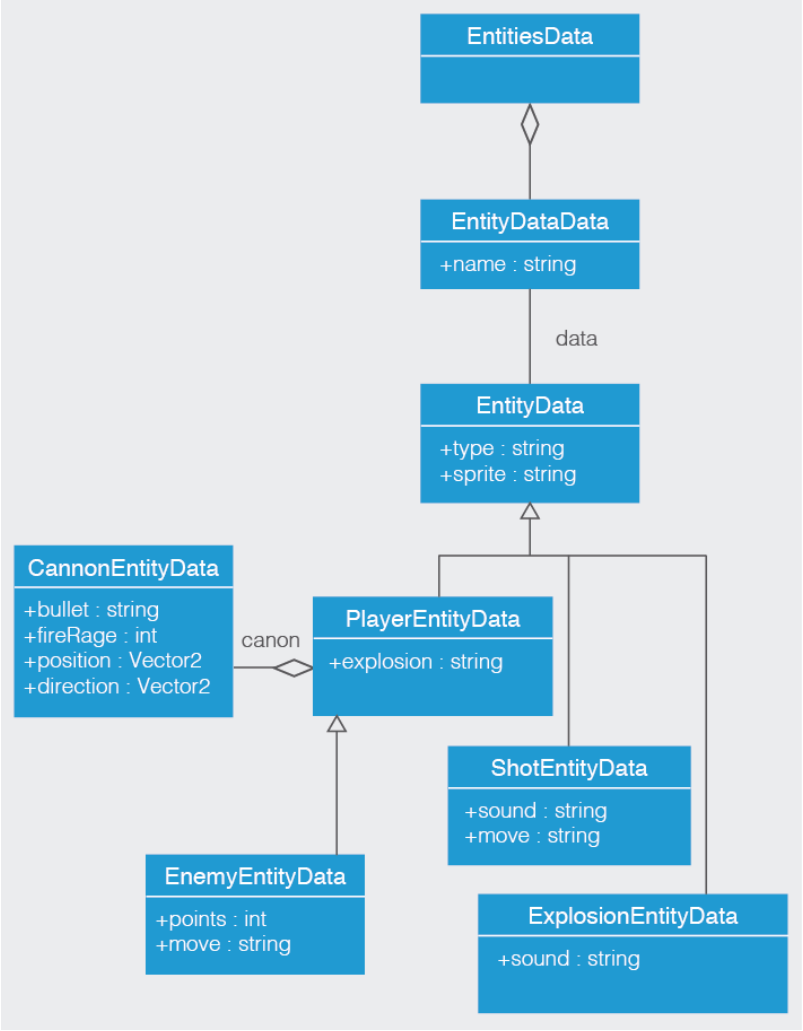
L'estructura dels fitxers `levels-data.json` és més complicada perquè conté múltiples estructures niudes (nivell, fons, onades d'enemics i formacions), però a la figura 1.34 se'n pot veure clarament la composició.

Figura 1.34. Estructura de dades de "levels-data.json"



El fitxer `entity-data.json` també inclou estructures niades. En aquest cas el fitxer inclou la informació del jugador, dels diferents tipus d'enemics i el seu armament, i les bales i les explosions, com es pot apreciar a la figura 1.35.

FIGURA 1.35. Estructura de dades de "entity-data.json"



Fixeu-vos que tot i que el més habitual és que cada nau només porti un canó, és possible afegir-hi qualsevol quantitat, ja que la propietat `cannon` és un *array*. Així, per exemple, la nau del jugador inclou dos canons col·locats a les posicions (35,5) i (35,45).

Com que la bala del jugador té una alçada de 14 píxels, l'alçada de la nau és de 64 píxels i l'origen de les coordenades (0,0) es troba a la cantonada superior esquerra. Per col·locar el segon canó simètric al primer s'ha de fer el càlcul següent: `64 - 14 - 5 = 45`. Per la distància horitzontal s'utilitza el mateix valor a tots dos canons.

Detecció de col·lisions

La detecció de col·lisions s'ha de calcular a cada iteració del bucle principal. Les comprovacions seran més o menys costoses segons la forma dels elements per comprovar. Per exemple, les dues formes de detectar col·lisions més simples són:

- **Forma circular**: requereix calcular si la distància que hi ha entre el centre de tots dos cercles és més gran que la suma dels radis.

- **Rectangles**: cal comprovar si qualsevol de les cantonades del primer rectangle es troba dintre de l'àrea del segon.

Els motors de joc de tercers permeten utilitzar formes més avançades en 2D i 3D per detectar col·lisions.

En cas que es necessités més precisió en la detecció, es poden utilitzar múltiples cercles i rectangles per ajustar la forma utilitzada per fer els càlculs, però –com es pot intuir– el cost de realitzar-los es multiplica.

A *IOC Invaders* la detecció de col·lisions està dividida entre dos components i s'utilitza la detecció de col·lisions per rectangles. Per una banda, els objectes de tipus `MovingGameObject`, com les naus i les bales, implementen `isCollidingWith`, que permet determinar si un objecte està col·lidint amb un altre. Per l'altra, en el motor del joc es troben els mètodes següents:

- **detectCollisions**: comprova totes les col·lisions possibles i marca com a destruïts els elements que hi hagin col·lidit.
- **detectCollisionsPoolWithPool**: detecta totes les col·lisions entre dos *pools* complets, per exemple el *pool* de bales del jugador amb el *pool* de naus enemigues.
- **detectCollisionsPoolWithGameObject**: detecta les col·lisions entre tots els objectes d'un *pool* amb un objecte concret, per exemple, el *pool* de bales enemigues amb el jugador.

1.3.9 Gestió d'enemics i nivells

El motor del joc *IOC Invaders* també és el responsable d'instanciar els enemics a mesura que el jugador arriba als punts marcats (propietat `distance`). Els mètodes responsables de generar els enemics són els següents:

- **updateWaves**: comprova si s'ha arribat a la distància requerida per instanciar un nou grup d'enemics o s'ha arribat al final del nivell.
- **spawner**: instancia un nou enemic o una nova onada d'enemics.
- **spawnEnemy**: instancia un únic enemic.
- **spawnFormation**: instancia una onada d'enemics en formació. Segons el tipus de formació i les dades de l'onada, les naus es col·locaran en diferents posicions.

S'ha de tenir en compte que, per simplificar, aquesta funcionalitat s'ha afegit al `GameManager`, però el més adient seria crear un objecte de tipus `SpawnManager` que fos l'encarregat de gestionar la creació d'instàncies dels enemics.

Aquests enemics, en ser destruïts, augmenten la puntuació del jugador. La puntuació ha de ser gestionada pel `GameManager`, ja que a diferència d'altres elements, la puntuació ha de persistir durant tota la partida.

Quant a la gestió de nivells, a *IOC Invaders* s'utilitzen les propietats i els mètodes següents:

- **currentLevel**: indica el nivell actual.
- **levels**: informa sobre els nivells del joc.
- **startLevel**: inicia el nivell.
- **setEndLevel**: avança al següent nivell. És invocat quan s'arriba al final del nivell i no és visible cap enemic.

Inicialització i bucle principal del joc

La resta de mètodes del `GameEngine` s'encarreguen de gestionar el cicle de vida del joc. És a dir, la inicialització, la finalització i el bucle principal:

- **start**: és invocat en iniciar-se el joc, posa en marxa el bucle principal (`update`).
- **restart**: invocat tant quan s'inicia el joc com quan es reinicia.
- **init**: inicia la càrrega de dades i seguidament invoca `initEnvironment`.
- **initEnvironment**: inicialitza els *pools* per les bales, les explosions i els enemics, estableix el context del canvas (`2D`) i inicia la càrrega de recursos.
- **setGameOver**: atura el funcionament del joc i mostra el missatge de fi del joc.
- **update**: executa el bucle principal del joc.

Quan es treballa amb l'element `canvas` és fonamental invocar el mètode `window.requestAnimationFrame`, ja que és l'encarregat de demanar al navegador que es cridi la funció passada com a argument abans de pintar la finestra.

Com es pot apreciar en el bucle principal de l'*IOC Invaders* (`update`), la primera acció que es porta a terme és invocar `windows.requestAnimationFrame` per tornar a ser cridat quan es torni a pintar la finestra. Seguidament s'actualitzen les onades i es detecta si s'ha produït alguna col·lisió.

Cal destacar que s'utilitza una màquina d'estats molt simple per determinar què s'ha de fer a continuació:

- Si el jugador ha estat destruït i l'estat no és `GAME_OVER`, s'acaba el joc.
- Si no hi ha enemics vius, s'ha arribat al final del nivell i l'estat no és `GAME_OVER`, l'estat passa a `LOADING_NEXT_LEVEL` i s'inicia la transició cap al pròxim nivell.
- Si no s'ha acabat el nivell i no s'ha acabat el joc, s'actualitza el jugador (estat `RUNNING`).

Podeu trobar més informació sobre les màquines d'estats finits a l'enllaç següent: goo.gl/t0j8OA.

Abans de finalitzar el bucle s'actualitza el *pool* d'explosions i finalment la interfície amb la nova distància i la puntuació.

Fixeu-vos que principalment la tasca del bucle principal és cridar el mètode `update` dels altres elements del joc i canviar d'estat. Si estructureu bé el vostre codi, la implementació del bucle principal és molt simple.

A continuació podeu trobar el codi del generador d'objectes de tipus `GameEngine`:

```
var gameEngineConstructor = function () {
  var that = {},
  levels = {},
  currentLevel,
  levelEnded,
  score,
  distance, // Relativa al nivell actual
  nextWave, // Relativa al nivell actual

  enemyPool,
  enemyShotPool,
  playerShotPool,
  explosionPool,

  background,
  player,
  state,

  ui = (function () {
    var scoreText = document.getElementById('score'),
      distanceText = document.getElementById('distance'),
      messageText = document.getElementById('messages');

    return {
      update: function () {
        scoreText.innerHTML = score;
        distanceText.innerHTML = distance;
      },

      showMessage: function (message, duration) { // Temps en mil·lisegons
        messageText.innerHTML = message;
        messageText.style.opacity = 1;

        setTimeout(function () {
          messageText.style.opacity = 0;
        }, duration);
      },

      hideMessage: function () {
        messageText.style.opacity = 0;
      },

      showGameOver: function () {
        document.getElementById('game-over').style.display = "block";
      },

      hideGameOver: function () {
        document.getElementById('game-over').style.display = "none";
      },

      transitionScreen: function (callback) {
        gameCanvas.style.opacity = 0;

        setTimeout(function () {
          gameCanvas.style.opacity = 1;
          callback();
        }, 3000); // la transició dura 3s
      },

```

```
60        fadeIn: function () {
61          gameCanvas.style.opacity = 1;
62        },
63
64        fadeOut: function () {
65          gameCanvas.style.opacity = 0;
66        }
67      };
68    })();
69
70    function initEnvironment(data) {
71      gameContext = gameCanvas.getContext("2d");
72
73      explosionPool = gameObjectPoolConstructor(100, explosionConstructor);
74      enemyShotPool = gameObjectPoolConstructor(500, shotConstructor);
75      enemyPool = gameObjectPoolConstructor(100, spaceshipConstructor, {
76        pool: {
77          bullet: enemyShotPool,
78          explosion: explosionPool
79        }
80      });
81
82      playerShotPool = gameObjectPoolConstructor(100, shotConstructor);
83
84      that.loadAssets(data.assets);
85    }
86
87    function update() {
88      window.requestAnimationFrame(update);
89
90      updatedSprites = [];
91
92      updateWaves();
93      detectCollisions();
94
95      background.update();
96      enemyPool.update();
97      enemyShotPool.update();
98      playerShotPool.update();
99
100
101     if (player.isDestroyed && state != GameState.GAME_OVER) {
102       setGameOver();
103
104     } else if (enemyPool.actives === 0 && levelEnded && state != GameState.
            GAME_OVER
105         && state != GameState.LOADING_NEXT_LEVEL) {
106       ui.transitionScreen(setEndLevel)
107       state = GameState.LOADING_NEXT_LEVEL;
108     } else if (state != GameState.GAME_OVER) {
109       player.update();
110       distance++;
111     }
112
113     explosionPool.update();
114
115     ui.update();
116   }
117
118   function detectCollisions() {
119     var impactInfo,
120       i;
121
122     // bala del jugador amb enemic
123     impactInfo = detectCollisionsPoolWithPool(playerShotPool, enemyPool);
124
125     if (impactInfo.length > 0) {
126       for (i = 0; i < impactInfo.length; i++) {
127         impactInfo[i].source.isDestroyed = true;
128         impactInfo[i].target.isDestroyed = true;
```

```
        score += impactInfo[i].target.points;
      }
    }

    // bala del enemic amb jugador
    impactInfo = detectCollisionsPoolWithGameObject(enemyShotPool, player);

    if (impactInfo.length > 0) {
      for (i = 0; i < impactInfo.length; i++) {
        impactInfo[i].source.isDestroyed = true;
        impactInfo[i].target.isDestroyed = true;
      }
    }

    // enemic amb jugador
    impactInfo = detectCollisionsPoolWithGameObject(enemyPool, player);
    if (impactInfo.length > 0) {
      for (i = 0; i < impactInfo.length; i++) {
        impactInfo[i].source.isDestroyed = true;
        impactInfo[i].target.isDestroyed = true;
      }
    }
  }

  function detectCollisionsPoolWithPool(poolA, poolB) {
    var i = 0,
      j = 0,
      impacts = [];

    while (poolA.pool[i] && poolA.pool[i].alive) {
      while (poolB.pool[j] && poolB.pool[j].alive) {
        if (poolA.pool[i].isCollidingWith(poolB.pool[j])) {
          impacts.push({
            source: poolA.pool[i],
            target: poolB.pool[j]
          });
        }
        j++;
      }
      j = 0;
      i++;
    }

    return impacts;
  }

  function detectCollisionsPoolWithGameObject(pool, gameObject) {
    var i = 0,
      impacts = [];

    while (pool.pool[i] && pool.pool[i].alive) {
      if (pool.pool[i].isCollidingWith(gameObject)) {
        impacts.push({
          source: pool.pool[i],
          target: gameObject
        });
      }
      i++;
    }

    return impacts;
  }

  function updateWaves() {
    var waves = levels[currentLevel].waves,
      currentWave;

    if (nextWave < waves.length && distance >= waves[nextWave].distance) {
      currentWave = waves[nextWave];
      spawner(currentWave.spawns);
```

```
      nextWave++;
    }

    if (distance > levels[currentLevel].end) {
      levelEnded = true;
    }
  }

  function spawner(spawns) {
    for (var i = 0; i < spawns.length; i++) {
      if (!spawns[i].formation) {
        spawnEnemy(spawns[i]);
      } else {
        spawnFormation(spawns[i]);
      }
    }
  }

  function spawnEnemy(enemy) {
    enemyPool.instantiate(enemy.type, enemy.position, enemy.speed);
  }

  function spawnFormation(wave) {
    var i,
      j,
      originPosition,
      spacer,
      nextColumn,
      currentPosition,
      side;

    switch (wave.formation.type) {
      case "columns":
        originPosition = {x: wave.position.x, y: wave.position.y};
        spacer = wave.formation.spacer;
        nextColumn = originPosition.y + wave.formation.column_height;
        currentPosition = {x: wave.position.x, y: wave.position.y};

        for (i = 0; i < wave.formation.amount; i++) {
          enemyPool.instantiate(wave.type, {
            x: currentPosition.x,
            y: currentPosition.y
          }, wave.speed);

          currentPosition.y += spacer;

          if (currentPosition.y >= nextColumn) {
            currentPosition.x += spacer;
            currentPosition.y = originPosition.y;
          }
        }
        break;

      case "grid":
        originPosition = {x: wave.position.x, y: wave.position.y};
        spacer = wave.formation.spacer;
        side = Math.round(Math.sqrt(wave.formation.amount));

        for (i = 0; i < side; i++) {
          for (j = 0; j < side; j++) {
            enemyPool.instantiate(wave.type, {
              x: originPosition.x + (spacer * i),
              y: originPosition.y + (spacer * j)
            }, wave.speed);
          }
        }

        break;

      case "row":
```

```
269         originPosition = {x: wave.position.x, y: wave.position.y};
270         spacer = wave.formation.spacer;
271
272         for (i = 0; i < wave.formation.amount; i++) {
273           enemyPool.instantiate(wave.type, {
274             x: originPosition.x + (spacer * i),
275             y: originPosition.y
276           }, wave.speed);
277         }
278         break;
279
280       case "column":
281         originPosition = {x: wave.position.x, y: wave.position.y};
282         spacer = wave.formation.spacer;
283
284         for (i = 0; i < wave.formation.amount; i++) {
285           enemyPool.instantiate(wave.type, {
286             x: originPosition.x,
287             y: originPosition.y + (spacer * i)
288           }, wave.speed);
289         }
290         break;
291
292       default:
293         console.log("Error, no es reconeix el tipus de formació");
294     }
295   }
296
297   function setEndLevel() {
298     currentLevel++;
299
300     if (currentLevel >= levels.length) {
301       currentLevel = 0;
302     }
303
304     enemyPool.clear();
305     explosionPool.clear();
306     enemyShotPool.clear();
307     playerShotPool.clear();
308
309     that.startLevel(currentLevel);
310     state = GameState.RUNNING;
311   }
312
313
314   function setGameOver() {
315     player.update();
316     state = GameState.GAME_OVER;
317
318     ui.hideMessage();
319     ui.showGameOver();
320     ui.fadeOut();
321     assetManager.fadeOutAudio(2000); // Donem temps perquè es produeixi l'
          explosió
322     setTimeout(function () {
323       assetManager.getMusic("game_over");
324     }, 2000);
325   }
326
327   that.init = function () {
328     that.loadData(config.asset_data_url, initEnvironment);
329   };
330
331   that.loadData = function (url, callback) {
332     var httpRequest = new XMLHttpRequest();
333
334     httpRequest.open("GET", url, true);
335     httpRequest.overrideMimeType('text/plain');
336     httpRequest.send(null);
337
```

```
    httpRequest.onload = function () {
      var data = JSON.parse(httpRequest.responseText);
      callback(data);
    };
  };

  that.loadAssets = function (assets) {
    for (var type in assets) {
      for (var i = 0; i < assets[type].length; i++) {
        assetManager.queueAsset(type, assets[type][i]);
      }
    }

    assetManager.downloadAll(that.loadEntititesData);
  };

  that.loadEntititesData = function () {
    that.loadData(config.entity_data_url, function (data) {
      entitiesRepository.add(data);
      that.loadLevelsData();
    })
  };

  that.loadLevelsData = function () {
    that.loadData(config.levels_data_url, function (data) {
      levels = data.levels;
      that.start();
    });
  };

  that.start = function () {
    background = backgroundConstructor({
      context: gameContext
    });

    that.restart();
    update();
  };

  that.startLevel = function (level) {
    var message = levels[level].name
      + "<p><span>"
      + levels[level].description
      + "</span></p>";

    ui.showMessage(message, 3000);

    background.start(levels[level].background);
    assetManager.getMusic(levels[currentLevel].music);

    levelEnded = false;
    nextWave = 0;
    distance = 0;

    player.position = {x: 10, y: 256};
  };

  that.restart = function () {
    ui.hideGameOver();
    assetManager.fadeInAudio();

    player = playerConstructor(
      {
        pool: {
          bullet: playerShotPool,
          explosion: explosionPool
        },
        position: {x: 10, y: 256},
        speed: {x: 4, y: 4}
      });
```

```
408
409     currentLevel = 0;
410     score = 0;
411     state = GameState.RUNNING;
412
413     enemyPool.clear();
414     explosionPool.clear();
415     enemyShotPool.clear();
416     playerShotPool.clear();
417
418     assetManager.resetMusic(levels[currentLevel].music);
419
420     ui.fadeIn();
421     that.startLevel(currentLevel);
422   };
423
424   return that;
425 };
```

Recordeu que podeu trobar el codi complet del joc *IOC Invaders* a l'enllaç següent: github.com/XavierGaro/ioc-invaders.