Orientació a objectes en JavaScript

Marcos González Sancho
Vicent Moncho Mas
Jordi Ustrell Garrigós

PID_00220469

UOC
Universitat Oberta de Catalunya

Índex

1. POO en JavaScript

1.1. Introducció

Actualment, la programació orientada a objectes és el paradigma de programació més adoptat entre els desenvolupadors i pels nous llenguatges que van sorgint.

La base d'aquest paradigma és l'ús de l'abstracció per a crear un model basat en el món real, i es tracta d'un model organitzat entorn dels objectes en lloc de les accions, i de les dades en lloc de la lògica.

Històricament, un programa era vist com un conjunt de processos lògics, que a partir d'unes dades d'entrada, es processaven i produïen unes dades de sortida. La programació orientada a objectes utilitza objectes (estructures de dades i mètodes) i les seves interaccions per a dissenyar aplicacions. En aquest paradigma, cada objecte s'ha de veure com una entitat amb una sèrie de propietats que la fan única i que és responsable de dur a terme una sèrie de processos (mètodes) per als quals ha estat creada sobre la base d'un patró o tipus comú per a tots els objectes de la seva mateixa classe.

JavaScript implementa els quatre principis bàsics de la programació orientada a objectes (abstracció, encapsulació, herència i polimorfisme), però a diferència d'altres llenguatges com Java, és possible programar sense utilitzar exclusivament aquestes característiques.

> JavaScript implementa els quatre principis bàsics de la programació orientada a objectes: **abstracció**, **encapsulació**, **herència** i **polimorfisme**.

Si bé històricament l'ús de JavaScript s'ha aplicat habitualment a desenvolupaments destinats a complementar una determinada funcionalitat i interacció en les pàgines HTML sobre les quals s'executava, amb el canvi que ha experimentat el desenvolupament web, avui dia el seu ús s'ha professionalitzat notablement, i ha passat d'emprar-se pràcticament sense aplicar principis de POO (mitjançant l'ús de funcions i codi aïllat) a no solament treballar amb POO, sinó a millorar els editors, IDE, eines complementàries, entorns de treball i biblioteques, i fins i tot ha avançat molt notablement en l'aplicació de patrons de disseny i bones pràctiques a l'hora d'abordar aplicacions completes i complexes.

La pròpia naturalesa del llenguatge basada en prototips, el fet de ser feblement tipificat i el seu dinamisme, al costat de la seva evolució que s'ha produït de la mà de l'augment del protagonisme que el desenvolupament web ha anat adquirint amb la implantació d'Internet en tots els àmbits de la nostra vida quotidiana, ens ha portat a tenir un panorama relativament confús a l'hora d'abordar el tema de la POO en JavaScript.

És molt habitual trobar diferents maneres d'abordar-la, des d'una aproximació clàssica a enfocaments més moderns que han anat essent possibles arran de nous mètodes i funcions suportades pels navegadors i el llenguatge. Això no obstant, no ha de veure's com un problema, sinó com a part de la riquesa d'aquest llenguatge que avui dia és pràcticament omnipresent.

1.2. Tipus d'objectes en JavaScript

Com en tots els llenguatges de programació orientats a objectes, es pot fer una classificació dels tipus suportats, establint com a criteri l'origen d'aquests:

Objectes natius de JavaScript

Dins d'aquest conjunt d'objectes, s'inclouen:

- Objectes associats als tipus de dades primitives com *String*, *Number* i *Boolean*, que proporcionen propietats i mètodes per a aquests tipus de dades.

- Objectes associats a tipus de dades complexes/compostes com *Array* i *Object*.

- Objectes complexos amb un enfocament clar d'utilitat, com per exemple *Date* per al treball amb dates, *Math* per al treball amb operacions matemàtiques i *RegExp* per al treball amb expressions regulars.

Les característiques d'aquests objectes estan especificades per l'estàndard ECMA-262, encara que, com sol ocórrer amb els estàndards, les diferents implementacions pels creadors de navegadors poden afegir algunes particularitats.

Objectes del navegador. BOM (*browser object model*)

Atès que JavaScript té l'origen en un entorn basat en un navegador, hi ha alguns objectes predefinits que el navegador "exposa" perquè siguin accessibles des de JavaScript, de manera que tingui accés a alguns elements d'aquest com per exemple per mitjà dels objectes `Window` i `Navigator`, que permeten el treball amb la finestra i la interacció amb el navegador respectivament.

El conjunt d'objectes d'aquest tipus se sol agrupar sota el nom BOM (*browser object model*). No hi ha un estàndard per a això, la qual cosa i el fet d'estar íntimament lligats al navegador fan que no siguin estàndard i disposin de variacions depenent del navegador i fins i tot de la versió que s'estigui utilitzant.

Objectes del document

Formen una part de l'especificació estàndard del DOM (*document object model*) i defineixen l'estructura de la interfície de la pàgina HTML. El consorci W3C ha dut a terme aquesta especificació estàndard del DOM per ajudar a fer convergir les diferents estructures implementades.

De manera semblant als objectes del navegador, els objectes del DOM "exposen" els elements i determinades funcionalitats d'aquests perquè aquest DOM es pugui manipular des de JavaScript, essent l'element principal l'objecte `document`.

Objectes personalitzats o definits per l'usuari

Aquests són definits i creats pel mateix usuari amb l'objectiu d'adaptar el seu programari a la realitat dels tipus de dades i processos que es vol representar mitjançant POO. El segon apartat d'aquest mòdul se centrarà en l'estudi de la creació d'objectes personalitzats.

Com es pot deduir dels punts anteriors, no hi ha un estàndard que imposi tots els aspectes de JavaScript, si bé JavaScript 1.5 és compatible amb ECMA-262 (d'ECMA International[1]), en la seva tercera edició regula els aspectes bàsics del llenguatge, i l'especificació DOM de W3C[2] defineix com s'han de presentar els documents estructurats (pàgines web) en un llenguatge de programació.

(1)ECMA International és una organització basada en grups d'estàndards per a la comunicació i la informació. L'organització es va fundar el 1961 per a estandarditzar els sistemes computeritzats a Europa.

(2)El World Wide Web Consortium, abreujat W3C, és un consorci internacional que produeix recomanacions per a la World Wide Web. Està dirigit per Tim Berners-Lee.

Encara que la tendència actual de les companyies és el compliment dels estàndards internacionals ECMA-262 i DOM de W3C, aquestes continuen definint els seus propis models d'accés a la interfície d'usuari i creant les seves pròpies extensions del DOM.

1.3. Principis bàsics de POO en JavaScript

A continuació presentem un petit recorregut pels principis bàsics de la programació orientada a objectes, fent una anàlisi específica relativa a JavaScript i el suport que els dóna:

1) Abstracció

Aquest concepte, molt lligat amb l'*encapsulació*, es refereix a la representació mitjançant propietats i mètodes d'una realitat en funció de les seves característiques i funcionalitat.

Un dels objectius de l'abstracció és poder gestionar de manera més còmoda la complexitat d'un model o problema real, descomponent el global en components més petits i que per tant siguin més senzills.

2) Encapsulació

Mentre que l'abstracció defineix un model, l'encapsulació oculta la part desitjada de la seva implementació a l'exterior, de manera que des de fora no hàgim de preocupar-nos de com està realitzat sinó de les propietats i mètodes que ens ofereix per interactuar.

Aquest mecanisme ens permet fer fins i tot modificacions sobre una classe, sense que repercuteixi en la resta del programa. Imaginem que refactoritzem un mètode d'una classe que tenia un mal rendiment de manera que acabem disposant-ne d'uns altres tres que fan la seva mateixa funció però de manera més eficient. Atès que la resta del programa es comunicava amb aquesta classe per mitjà dels mètodes exposats gràcies a l'encapsulació, mentre el mètode públic no canviï la resta del programa no queda afectat pels canvis interns de la classe.

Refactoritzar

Dur a terme ajustos de codi interns en els mètodes d'una classe, sense que afecti com es comunica amb la resta del programa i, per tant, de manera que no calgui canviar codi addicional.

En el cas de JavaScript, l'encapsulació s'aconsegueix gestionant les propietats i mètodes que s'exposaran a l'exterior, la qual cosa es pot aconseguir de diferents maneres. La realitat és que la falta de modificadors d'accés fa que no sigui una tasca directa com en el cas d'altres llenguatges, cosa que veurem en el segon apartat d'aquest mòdul.

3) Herència

El concepte d'*herència* es basa a crear objectes més especialitzats a partir d'altres models existents. Aquest principi també està molt lligat amb l'abstracció i fins i tot amb l'encapsulació per mitjà dels modificadors d'accés.

En JavaScript, i a causa de la seva naturalesa basada en prototips, aquesta herència s'aconsegueix per mitjà d'aquests, de manera que un objecte refereixi com el seu prototip un altre (el seu antecessor) per mitjà de la seva propietat `prototype`.

4) Polimorfisme

El concepte de *polimorfisme* està absolutament lligat a l'*herència*, i es basa en el fet que dos objectes de diferent tipus puguin compartir una mateixa interfície, de manera que es pugui treballar amb aquesta interfície abstraient-se independentment del tipus de l'objecte amb el qual es treballa.

Hi ha diferents tipus de polimorfisme:

- **Polimorfisme *ad hoc***
  Per mitjà de la sobrecàrrega (*overloading*) de mètodes que permet disposar de diversos mètodes amb el mateix nom però que per mitjà del context en el qual es criden (per exemple, el tipus del paràmetre que rep) poden variar el comportament.
  Però, en realitat, aquest tipus de polimorfisme se sol resoldre en temps de compilació, i en el cas de JavaScript, en ser un llenguatge sense una tipificació estricta, es pot simular implementant en la funció o mètode de manera manual aquesta distinció, per exemple mitjançant l'ús de la funció `typeof`.

  Un exemple de polimorfisme *ad hoc* incorporat en JavaScript de manera pròpia són alguns dels seus operadors, que permeten la sobrecàrrega, com per exemple l'operador de suma que funciona tant per a nombres com per a cadenes de caràcters.

- **Polimorfisme paramètric (*generics* en Java o *templates* en C++)**
  Permet a una funció o un tipus de dades ser escrits de manera genèrica, de manera que pot manejar valors de manera idèntica sense dependre del seu tipus.
  El fet que en JavaScript no es disposi d'una tipificació estricta pot donar lloc, en determinades situacions, a suportar aquest tipus de polimorfisme

```
function log (arg) {
   console.log(arg);
}
```

  encara que no ho fa de manera completa, al contrari que TypeScript, un superconjunt de JavaScript creat per Microsoft que permet ser compilat a JavaScript pur i que avança en molts aspectes les capacitats i manera de treballar amb POO del futur estàndard ECMAScript 6[3] amb el qual intenta estar molt alineat.
  TypeScript afegeix funcionalitats a JavaScript, entre elles l'ús de tipus i objectes basats en classes. Està pensat per a ser usat en projectes grans en què l'excessiva "flexibilitat" de JavaScript pot ser un problema.

[(3)]Es pot veure la taula de les compatibilitats actuals per navegadors d'aquest estàndard a http://kangax.github.io/compat-table/es6/.

- **Subtipificació o polimorfisme clàssic**
  És la forma més habitual de polimorfisme i es basa que el fet que dos objectes comparteixin un mateix pare per mitjà de l'herència ens permet treballar amb els mètodes disponibles en el pare sense haver de preocupar-nos del tipus d'objecte que en realitat estem manipulant, i aquests poden tenir

diferent comportament gràcies a la sobreescriptura (*overwriting*) de mètodes.

1.4. Implementació de POO en JavaScript

En els punts següents farem un recorregut per les implementacions dels aspectes més importants de POO en JavaScript, posant l'accent principalment en bones pràctiques i en els avantatges i inconvenients d'algunes implementacions alternatives existents.

Nota

Encara que avui dia hi ha múltiples enfocaments de treball amb POO en JavaScript, cadascun amb els seus avantatges i inconvenients, per qüestions didàctiques s'ha triat l'aproximació més clàssica, a pesar que JavaScript encara no disposa de tots els elements desitjables per a simplificar el seu ús tals com paraules reservades per a les estructures convencionals, mètodes especials, modificadors d'accés, etc.

1.4.1. El concepte de *classe* i *constructor*

Atès que JavaScript no disposa de la paraula reservada `class` per a crear aquesta estructura bàsica de la POO, s'empren funcions (que no deixen de ser objectes) per a obtenir el mateix resultat quant a funcionalitat.

Aquestes funcions es denominen *funcions constructores* i lògicament poden ser natives (com, per exemple, la classe *Date*) o definides per l'usuari.

Nota

Per comoditat, d'ara endavant s'emprarà el terme **classe** (encara que aquesta estructura de dades en JavaScript no existeixi com tal) per a referir-se a aquestes funcions constructores, natives o definides per l'usuari.

Per tant, la manera de definir una classe en JavaScript és creant una funció que la representi, que per convenció s'anomena amb la primera lletra en majúscula, la qual cosa ja estableix una manera de diferenciar-les de funcions normals del llenguatge o de la nostra aplicació.

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

Aquest fragment de codi posaria a la nostra disposició la classe Animal, que en aquest punt del nostre exemple, no tindria cap propietat ni mètode.

A més, JavaScript tampoc no té una paraula reservada per a indicar el mètode constructor de manera explícita com ho fan altres llenguatges, per la qual cosa la mateixa funció que defineix la classe es converteix en constructor de la mateixa. És a dir, Animal és a més la funció constructora de la classe Animal.

1.4.2. El concepte d'*objecte*. Instanciació

Recordem que una classe és un model que presenta propietats (característiques) i mètodes (funcionalitats), a partir de com es generen objectes que per tant són representacions particulars d'aquesta classe. Aquest procés de creació d'un objecte a partir d'una classe es denomina *instanciació*.

Per a instanciar un objecte en JavaScript a partir d'una classe s'empra l'operador `new`, que s'acompanya amb la classe o funció constructora a partir de la qual es vol crear l'objecte. El procés intern que du a terme aquest operador internament consta de quatre passos, essent d'especial interès el segon:

1) Crea un nou objecte, del tipus natiu *Object*.

2) Vincula el nou objecte a la classe mitjançant la seva propietat especial `__proto__` (o `[[prototype]]`), a la qual assigna el prototip de la seva classe. D'aquesta manera, l'objecte instanciat podrà accedir a la classe de la qual prové mitjançant la seva propietat `prototype`.

3) Fa una crida a la funció constructora a partir de l'objecte creat i, per tant, executa totes les instruccions que es troben dins d'aquesta funció constructora.

4) Retorna el nou objecte creat.

A causa que JavaScript és un llenguatge basat en prototips, els mecanismes de creació d'objectes i l'herència depenen àmpliament d'aquestes dues propietats: `__proto__` i `prototype`. En l'apartat següent s'aprofundirà una mica més en aquests termes.

Per tant, si volem crear una instància de la nostra classe Animal, ampliarem el nostre exemple amb el codi següent:

```
function Animal () {
}


var elmeuAnimal = new Animal();
```

Analitzant el que implica en el punt 2 esmentat més amunt, el resultat de la seva interpretació seria que l'objecte `elmeuAnimal` disposa d'una propietat `__proto__` no accessible directament, que referencia a `Animal.prototype`, i que és la que permet que des de l'objecte es pugui accedir a les propietats i mètodes de la classe, que són compartides per mitjà de la propietat `prototype` d'aquesta classe.

A més, en aquest exemple, i atès que el constructor està buit, el punt 3 indicat anteriorment no du a terme cap acció, ja que no hi ha sentències a l'interior de la funció constructora, però l'habitual és incloure en aquesta funció constructora la inicialització de les propietats de l'objecte, directament o mitjançant la crida a un mètode que faci aquesta acció.

1.4.3. Prototype i __proto__

Web recomanat

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto

Encara que aquests aspectes seran notablement més importants quan tractem l'herència, és interessant tenir clar des del primer moment què són i com interactuen.

Simplificant, podríem dir que `__proto__` és la referència que té cada instància d'una classe al `prototype` d'aquesta, i és la propietat que empra per a poder accedir a les propietats i mètodes definits en aquesta classe mitjançant `prototype`.

Actualment, es desaconsella l'accés directe a aquesta propietat en benefici d'Object.create i `Object.getPrototypeOf`.

1.4.4. Les propietats

Les propietats permeten definir les característiques d'un objecte, i personalitzar-lo per a diferenciar-se d'altres, per la qual cosa no han de pertànyer al prototip sinó a l'objecte en si mateix.

Depenent del criteri que es prengui en POO amb JavaScript, podem dividir les propietats en diferents tipus. Si com a criteri marquem la pertinença, tindrem la classificació següent:

- Propietats estàtiques o de classe
- Propietats d'objecte (les més habituals i comunament referides com a *propietats a seques*)

D'altra banda, dins de les propietats d'objecte, si el criteri és l'accessibilitat, podem dividir-les en:

- Propietats públiques (accessibles externament)
- Propietats privades (accessibles solament dins de la pròpia classe)

Propietats estàtiques o de classe

En JavaScript, gràcies al fet que les funcions no deixen de ser objectes, i igual que en altres llenguatges, les classes poden tenir vinculades directament propietats, que se solen denominar *estàtiques*, i que es comparteixen per mitjà de la classe entre totes les seves instàncies, i són emmagatzemades solament en un lloc, la classe, i són accessibles per mitjà del nom d'aquesta.

La manera de definir aquestes propietats és fer-ho directament sobre la classe i no sobre el seu prototip. A continuació es mostra un exemple típic de l'ús d'aquest tipus de propietats per a una hipotètica classe que albergui utilitats matemàtiques:

```
function MathUtils() {
}

MathUtils.factorRadiants = Math.PI/180;
MathUtils.factorGraus = 180/Math.PI;

var graus = 38;
console.log('Radiants: ' + graus * MathUtils.factorRadiants );
// output: 0.6632
```

Propietats públiques

Aquest tipus de propietats d'objecte es defineixen en el mateix objecte, i per tant per a això es fa ús de la paraula reservada `this`, que ens permet tenir accés a un objecte dins de qualsevol dels seus mètodes.

El lloc habitual per a declarar aquestes propietats és el constructor de la classe, és a dir, la funció constructora si parlem de JavaScript. Òbviament i com qualsevol altra funció, el nostre constructor podria rebre paràmetres, per exemple amb l'objectiu d'inicialitzar algunes propietats:

```
function Animal (nom) {
   this.nom = nom;
   this.pes = 0;
   this.altura = 0;
}

var elmeuAnimal = new Animal('Vostè');
```

Això faria que `elmeuAnimal` tingués automàticament en el seu interior aquestes tres propietats, que inicialitzarà la propietat nom amb el paràmetre que rep, i que a més per ser públiques serien accessibles des de la pròpia instància:

```
function Animal (nom) {
```

```
    this.nom = nom;
    this.pes = 0;
    this.altura = 0;
}

var elmeuAnimal = new Animal('Vostè');
elmeuAnimal.peso = 30;
elmeuAnimal.altura = 50;
```

Com podem veure, les propietats públiques són accessibles per la instància simplement accedint-hi per mitjà del seu nom mitjançant la sintaxi de punt, encara que també serien accessibles de la manera següent:

```
var elmeuAnimal = new Animal('Vostè');
elmeuAnimal['pes'] = 30;
elmeuAnimal['altura'] = 50;
```

Aquesta notació ens permet accedir a una propietat d'un objecte de manera dinàmica, ja que el contingut exposat entre els claudàtors pot ser una expressió avaluada en temps d'execució.

```
var elmeuAnimal = new Animal("Vostè");
elmeuAnimal["pe"+"so"] = 30;
```

Fixeu-vos que en l'exemple s'usen cometes dobles per a fer notar que en aquest cas no hi ha diferència en la seva elecció per a indicar un literal de cadena de caràcters.

> Tret que s'hagi d'accedir a una propietat o mètode de manera dinàmica, sempre és aconsellable emprar la notació del punt abans que la notació dinàmica, ja que és més ràpida i més llegible.

És important comprendre que les propietats nom, pes i altura s'assignen i es creen per a cada instància en el seu constructor (per a usar la paraula reservada `this`) i, per tant, no són compartides per mitjà del prototip entre les diferents instàncies de la classe Animal. Això permet que cada instància o objecte pugui donar-los un valor, sense afectar la resta, ja que si es fes sobre el `prototype` de la classe les modificacions s'aplicarien automàticament a totes les instàncies creades a partir d'aquesta classe.

La conseqüència d'aquest comportament és que el consum de memòria per a les propietats d'objecte és individual per a cada objecte instanciat, mentre que el consum de memòria de les propietats i mètodes del prototip és compartit per a totes les instàncies.

Propietats privades

L'objectiu de les propietats privades és mantenir el seu accés restringit a la funcionalitat interna de la pròpia classe, i que no sigui accessible des de l'exterior.

En JavaScript no hi ha propietats privades com a tals, però es pot aconseguir aquesta característica utilitzant les denominades *closures* per a mitjançant l'àmbit de les variables obtenir resultats similars. Per a això per tant es defineixen variables locals a la funció constructora amb la declaració mitjançant `var`, de manera que no siguin accessibles excepte des de l'àmbit d'aquesta funció.

En el nostre exemple, podríem disposar d'una propietat privada que ens permetés controlar els anys de vida del nostre animal:

```
function Animal () {
   var edat = 0;

   this.nom = '';
   this.pes = 0;
   this.altura = 0;

}

var elmeuAnimal = new Animal();
console.log(elmeuAnimal.pes); // mostraria "0" per consola
console.log(elmeuAnimal.edat); // error, undefined no és accessible
```

Aquest exemple mostra com la propietat `edat` no és accessible des de la instància, ja que el seu ús està fora de l'àmbit intern de la mateixa classe. És molt important tenir en compte que l'ús de propietats privades no és una cosa excessivament natural en JavaScript per la naturalesa que tenen els mateixos objectes en el llenguatge, i que hi ha multitud d'aproximacions per a aconseguir aquest efecte, cadascuna amb els seus pros i contres.

Web recomanat

http://www.2ality.com/2012/03/private-data.html

Entre les diferents opcions que trobem per a gestionar aquesta característica en JavaScript, apareix com a opció molt vàlida l'ús d'una convenció en la nomenclatura per mitjà del prefix "_" per a identificar les variables, encara que realment siguin públiques. Més endavant, quan analitzem els diferents tipus de mètodes, aprofundirem en les formes d'accés intern a una propietat privada.

Reforçant aquest enfocament, cal destacar que TypeScript omet l'ús de propietats privades una vegada que el codi és compilat a JavaScript.

Exemple

Exemple de classe Animal amb propietat privada edat en TypeScript:

http://www.typescriptlang.org/Playground#src=class%20Animal%20%7B%0A%20%20%20%20private%20edad%3A%20number%20%3D%200%3B%0A

%09public%20nombre%3A%20string%20%3D%20''%3B%0A
%09public%20peso%3Anumber%20%3D%200%3B%0A
%09public%20altura%3Anumber%20%3D%200%3B%0A%7D%0A%0Avar
%20miAnimal%20%3D%20new%20Animal()%3B

1.4.5. Els mètodes

Els mètodes permeten definir la funcionalitat de la classe, i interactuar amb els objectes instanciats a partir d'aquesta, per a poder donar vida al model real que vol representar el nostre programa. Com passava amb les propietats disposarem de:

- Mètodes estàtics
- Mètodes d'objecte, que al seu torn es divideixen en:
  - públics
  - privats

A més, veurem un tercer tipus de mètode denominat *privilegiat*, que sorgeix en JavaScript com a conseqüència de les limitacions en l'ús de propietats privades en la implementació de la POO clàssica.

Mètodes estàtics

Igual que teníem propietats estàtiques vinculades a la classe, és possible definir mètodes estàtics que poden ser cridats directament pel nom d'aquesta classe. Continuant amb el nostre exemple d'utilitats matemàtiques, tindríem:

```
function MathUtils() {
}

MathUtils.factorRadiants = Math.PI/180;
MathUtils.factorGraus = 180/Math.PI;

MathUtils.arrodonirDecimals = function(num, digits)
{
      return Math.round(num * Math.pow(10,digits)) / Math.pow(10,digits);
}

console.log( MathUtils.arrodonirDecimals(1.123456789, 3) );
// output: 1.123
```

Mètodes públics

Els mètodes públics definits en una classe tenen com a finalitat dotar l'objecte d'una interfície externa sobre la qual podran interactuar altres parts del nostre programa.

Continuant amb el nostre exemple d'Animal, podrem definir en la classe dos mètodes públics néixer i créixer de la manera següent:

```
function Animal (nom) {
   var edat = 0;

   this.nom = nom;
   this.pes = 0;
   this.altura = 0;
}

Animal.prototype.néixer = function (pes, altura) {
   this.pes = pes;
   this.altura = altura;
}

Animal.prototype.créixer = function(pes, altura) {
   this.pes += pes;
   this.altura += altura;
}

var elmeuAnimal = new Animal('Vostè');
elmeuAnimal.néixer(2, 10); // neix amb 2 kg i 10 cm
elmeuAnimal.créixer(5, 10); // creix 5 kg i 10 cm
```

És important entendre que els definim en la classe per la seva propietat *prototype* perquè aquests mètodes estiguin presents en totes les instàncies de la classe, ja que, com hem dit, aquesta propietat *prototype* de la classe és la que s'assigna a la propietat *__proto__* de l'objecte instanciat i, per tant, dóna accés a aquests mètodes.

Per cridar els mètodes, i igual que passava amb les propietats, simplement hem d'emprar el seu nom amb la notació del punt, i encara que és una cosa molt poc freqüent, també seria viable fer-ho amb la notació dinàmica amb claudàtors:

```
var elmeuAnimal = new Animal("Vostè");
elmeuGos['néixer'](2, 10);
```

Mètodes privats

Els mètodes privats d'una classe són mètodes que estan dissenyats per a ser emprats per la pròpia classe, i no de manera externa. Per tant, són mètodes que no formen part de la interfície que volem exposar a l'exterior, sinó de suport a altres mètodes.

Igual que per a aconseguir disposar de propietats privades s'empraven variables locals a la funció constructora, per a aconseguir mètodes privats s'empren funcions locals en aquest constructor.

```
function Animal (nom) {
   var edat = 0;
   var controlEdat = -1;

   this.nom = nom;
   this.pes = 0;
   this.altura = 0;

   function envellir() {
      edat += 1;
   }

   this.néixer(2, 10);
   controlEdat = setInterval(envellir, 10000);
}

Animal.prototype.néixer = function (pes, altura) {
   this.pes = pes;
   this.altura = altura;
}

Animal.prototype.créixer = function(pes, altura) {
   this.pes += pes;
   this.altura += altura;
}

var elmeuAnimal = new Animal('Vostè');
elmeuAnimal.créixer(5, 10); // creix 5 kg i 10 cm
```

En aquest exemple hem introduït diversos canvis, que són importants per a reforçar els problemes que es deriven de l'ús de mètodes i propietats privades, en aconseguir aquesta funcionalitat per mitjà de *closures*, jugant amb l'àmbit de les variables i funcions.

Primer de tot hem declarat una propietat privada (variable local al constructor) denominada `controlEdat` per disposar d'un interval que ens permeti automatitzar l'envelliment de l'animal. D'altra banda, hem creat un mètode privat (funció local al constructor) denominat `envellir` que el que fa és sumar 1 a la propietat privada `edat`.

Per a declarar un mètode privat no s'usa la paraula reservada ***this***, ja que això faria que el mètode fos accessible per a les instàncies de la classe. Aquesta és la causa que s'empri una funció local.

Finalment, veiem que en el mateix constructor hem cridat la funció `néixer`, i just immediatament hem llançat i desat a `controlEdat` l'envelliment mitjançant la funció `setInterval` de JavaScript, que ens permet executar una funció de manera periòdica (l'interval el marca el temps en mil·lisegons passats com a segon paràmetre).

El lògic hagués estat haver llançat l'interval a `envellir` dins de la funció `néixer`, però a causa dels àmbits, el mètode privat `envellir` no seria accessible des del mètode públic `néixer`, cosa que no té res a veure amb els criteris de POO que trobem en altres llenguatges.

No obstant això, dins del constructor sí que tenim accés a aquest mètode, ja que està dins del seu àmbit o *scope*. De fet, la propietat `controlEdat` també és perfectament accessible en estar exactament en el mateix àmbit o *scope* en el qual ens trobem.

A més, atès que `envellir` està en el mateix àmbit que la propietat `edat` (tots dos locals al constructor), des de dins del mètode `envellir` podem tenir accés a la propietat `edat` sense problemes, ja que encara que no la troba dins de l'àmbit envellir, pujant un nivell cap amunt en la jerarquia d'àmbits arriba al constructor en què efectivament està definida.

No obstant això, i com es pot intuir, des dels mètodes públics declarats en `el prototype` tampoc no tindríem accés a les propietats privades, per la qual cosa la propietat `edat` no seria accessible més que des del mètode `envellir` i cap altre mètode de la classe o fins i tot la resta del programa podria tenir accés a aquest valor.

Un altre gran problema d'aquesta implementació és que no tindríem accés al propi objecte (propietats públiques i mètodes públics) des dels mètodes privats, ja que a l'interior d'aquests la paraula reservada `this` fa al·lusió al propi mètode (per ser una funció i, per tant, un objecte) i no a l'objecte en el qual es troba inclosa la funció.

La manera de salvar aquest problema seria afegir una closure en el constructor com es veu en l'exemple següent simplificat:

```
function LamevaClasse()
{
   var privada = 'privada';
   var _this = this;

   this.publica = 'publica';

   function metodePrivat() {
      privada = nouValor;
```

```
        _this.publica = nouValor;
        _this.metodePublic();
    }
}

LamevaClasse.prototype.metodePublic = function() {
}
```

D'aquesta manera, des del mètode privat usem `_this` per a accedir als mètodes i propietats públiques de la classe.

Mètodes privilegiats

Els mètodes privats d'una classe són mètodes que estan dissenyats per a ser emprats conjuntament amb propietats privades, de manera que solucionin alguns dels problemes comentats anteriorment en l'accés a aquestes propietats.

Nota

A causa dels complexos problemes que comporta l'ús de mètodes privats com a funcions locals en el constructor, els mètodes privilegiats són l'aproximació recomanada per a aquesta assignatura quan calgui treballar amb propietats privades.

Una altra aproximació acceptada serà l'ús de la nomenclatura amb prefix de guió baix per a especificar propietats i mètodes privats, encara que en realitat s'apliquin en el prototip i per tant siguin públiques.

Es basen a declarar els mètodes com a propis de l'objecte, dins del constructor mitjançant la paraula reservada `this`, de manera que mitjançant l'àmbit local puguin tenir accés a les propietats privades i, al seu torn, puguin ser usats en mètodes declarats en el prototip, fora del constructor o directament per mitjà de la instància (ja que aquests mètodes no deixen de ser públics).

```
function Animal (nom) {
    var edat = 0;
    var controlEdat = -1;

    this.nom = nom;
    this.pes = 0;
    this.altura = 0;

    this.getEdat = function() {
        return edat;
    }

    this.setEdat = function(novaEdat) {
        edat = novaEdat;
    }

    this.getControlEdat = function() {
```

```
        return controlEdat;
    }

    this.initControlEdat = function(pes, altura) {
        var _this = this;
        controlEdat = setInterval( function() {
            _this.envellir();
        }, 10000);
    }
}

Animal.prototype.envellir = function() {
    this.setEdat( this.getEdat()+1 );
}

Animal.prototype.néixer = function (pes, altura) {
    this.pes = pes;
    this.altura = altura;
    this.initControlEdat();
}

Animal.prototype.créixer = function(pes, altura) {
    this.pes += pes;
    this.altura += altura;
}

var elmeuAnimal = new Animal('Vostè');
elmeuAnimal.créixer(5, 10); // creix 5 kg i 10 cm
```

La contrapartida de l'ús d'aquests mètodes és que, en ser creats en la instància i no en el prototip (perquè es declaren en `this`), consumeixen més memòria en funció de la quantitat d'instàncies creades malgrat solucionar en bona part els greus problemes derivats de l'ús de mètodes privats com a funcions locals del constructor.

Pèrdues de l'àmbit de la classe

En l'exemple anterior es dóna la necessitat d'usar una *closure* per a no perdre l'àmbit de la classe en aplicar una funció com `setInterval` sobre un mètode de la classe (passaria el mateix si assignéssim una crida a un mètode d'una classe com a conseqüència d'un esdeveniment, per exemple, al clic d'un botó).

Per a això es declara una variable local `_this` a la qual s'assigna l'objecte mateix mitjançant `this`, i d'aquesta manera sí que podrem tenir accés a aquesta variable local des de la funció anònima i, per tant, no perdre l'àmbit de la classe.

1.4.6. Herència

L'herència persegueix l'objectiu d'especialització, és a dir, que una o diverses classes es beneficiïn de les propietats i mètodes d'una classe existent per a, en comptes de partir de zero, poder simplement especialitzar-se en determinats aspectes.

JavaScript suporta l'herència com a mecanisme per a l'aplicació de POO als nostres programes, i per a això fa ús dels prototips que ja hem anat entenent en els punts anteriors.

La manera d'implementar l'herència és assignant al prototip d'una classe (classe filla) una instància de la classe de la qual es vol heretar (classe pare). Òbviament, aquesta aproximació reflecteix la incapacitat de disposar d'herència múltiple en JavaScript, en no poder assignar a un mateix prototip dos elements diferents. No obstant això, hi ha diferents aproximacions més complexes per a aconseguir altres tipus d'herència.

Tipus d'herència

Tipus d'herència en JavaScript: http://javascript.crockford.com/inheritance.html

Si volguéssim preveure en el nostre exemple una classe Gos, que òbviament comparteix les circumstàncies generals per a tots els animals, podríem estendre mitjançant herència aquesta classe Animal, en comptes de declarar-ne de nou totes les propietats i mètodes.

A continuació, es mostra un exemple d'aquest procés que se centra exclusivament en les parts de codi que intervenen en el procés d'herència, és a dir, la definició de la classe filla i l'aplicació d'aquesta herència sobre la classe Animal:

```
function Gos (nom) {
   this.parent.constructor.call(this,nom);
   }

Gos.prototype = new Animal();
Gos.prototype.constructor = Gos;
Gos.prototype.parent = Animal.prototype;

Gos.prototype.lladrar = function() {
   console.log('guau');
}

var elmeuGos = new Gos('Vostè');
elmeuGos.créixer(5, 10); // creix 5 kg i 10 cm
miPerro.lladrar();
```

Analitzem cadascun dels apartats clau del procés d'herència representat en el codi anterior:

1) Com es pot veure, la declaració de la classe filla no té cap particularitat pel que fa a la classe pare; no obstant això, atès que es vol aprofitar la funcionalitat del constructor d'Animal, es fa una crida a aquest constructor mitjançant:

```
this.parent.constructor.call(this, nom);
```

2) A continuació s'aplica el mecanisme per a aconseguir l'herència, que com veiem simplement assigna al `prototype` de la classe filla una instància de la classe pare. És fonamental fer això en segon lloc, perquè la resta d'operacions sobre el `prototype` de la nova classe siguin "afegits" o "sobreescriptures" sobre les propietats i mètodes de la classe pare.

Si féssim aquesta operació en últim lloc, en realitat podríem sobreescriure mètodes de la classe filla amb els de la classe pare, que és justament el contrari del que es persegueix.

En aquest sentit, l'herència funciona de manera similar als àmbits o *scopes*, ja que busca una propietat o mètode en la classe actual, i si no la troba puja jeràrquicament per mitjà del seu *prototype* a la classe pare per buscar-la allà, i així successivament.

3) Posteriorment es corregeix el constructor de la classe filla, ja que en aplicar al `prototype` d'una classe l'herència, se sobreescriu aquest mètode i per tant és necessari fer l'ajust corresponent si volem evitar que el constructor de la classe filla sigui el de la classe pare (en aquest cas, **sense aquesta línia** el constructor de la classe Gos seria la funció Animal).

4) Desem una referència a la classe pare perquè aquesta sigui accessible des del constructor i mètodes de la classe filla, per poder aconseguir, per exemple, la crida al constructor d'Animal des del constructor de Gos que s'ha esmentat en el pas 1 (aquest pas és per a disposar d'una cosa similar a *super* existent en altres llenguatges OO).

5) Finalment, declarem tots els mètodes públics de la nova classe mitjançant `el seu prototype`. Aquests mètodes poden ser nous i, per tant, estendre la funcionalitat de la classe pare, o poden ser mètodes existents en la classe pare i, per tant, els estaríem sobreescribint per tenir una funcionalitat diferent per mitjà dels mateixos mètodes (base del concepte de *polimorfisme*).

Com podem veure en la part del codi en què s'instancia un objecte de la classe Gos, aquest objecte no solament té els seus propis mètodes (lladrar) sinó que disposa dels mètodes de la classe pare (créixer).

Els beneficis de l'ús d'herència en els casos aplicables són molts, però entre tots destaquem l'agilitat per a dotar de funcionalitat diverses classes del nostre codi quan l'ampliació que es requereix afecta les classes superiors en la jerarquia d'herència, ja que, en aplicar el codi comú en aquestes, es trasllada automàticament a totes les filles (tants nivells de profunditat com tingui l'herència).

Aquesta mateixa característica és la que fa que l'herència gairebé sempre permeti una reducció de la quantitat del nostre codi (gràcies a la reutilització dels mètodes heretats) i, per tant, augmenta la llegibilitat i maneig de les nostres aplicacions.

1.4.7. Polimorfisme

Com vèiem en parlar d'herència, una de les possibilitats que ens dóna la seva implementació en JavaScript és disposar de l'anomenat *polimorfisme subtipificat o clàssic*.

Aquest tipus de polimorfisme ens permet treballar amb un objecte a un determinat nivell sense haver de preocupar-nos del seu tipus exacte, gràcies a treballar amb ell a un nivell superior (classe pare), que garanteix que gràcies a l'herència sempre disposarem dels mètodes que cridem, tant si són els originals de la classe pare com les versions sobreescrites en les classes filles.

En qualsevol cas, els beneficis d'aquest mecanisme són que és el mateix llenguatge el que resol aquesta problemàtica, sense que al nivell que és emprat en el nostre codi ens hàgim de preocupar per aquestes comprovacions de tipus. Vegem un exemple per comprendre-ho millor:

```
function LamevaApp() {
   this.pantallaActual = null;
}
LamevaApp.PANTALLA_MENU = 'MENU';
LamevaApp.PANTALLA_DETALL = 'DETALL';

LamevaApp.prototype.inicialitzar = function() {
   this.actualitzarPantalla( LamevaApp.PANTALLA_MENU );
}

LamevaApp.prototype.actualitzarPantalla = function(novaPantalla) {
   if (this.pantallaActual === null || novaPantalla !== this.pantallaActual.getNombre() )
   {
      if (this.pantallaActual !== null) {
         this.pantallaActual.ocultar();
         this.pantallaActual.destruir();
      }

      switch (novaPantalla)
```

```
      {
         case LamevaApp.PANTALLA_MENU:
         this.pantallaActual = new PantallaMenu();
         break;

         case LamevaApp.PANTALLA_DETALL:
         this.pantallaActual = new PantallaDetall();
         break;
      }

      this.pantallaActual.setNom(novaPantalla);
      this.pantallaActual.mostrar();
   }
}

function Pantalla () {
   this._nom = '';
}

Pantalla.prototype.setNom = function(nom) {
   this._nom = nom;
}

Pantalla.prototype.getNom = function() {
   return this._nom;
}

Pantalla.prototype.mostrar = function() {
      // Codi per a mostrar la pantalla
}

Pantalla.prototype.ocultar = function() {
   // Codi per a ocultar la pantalla
}

Pantalla.prototype.destruir = function() {
   // Codi per a eliminar la pantalla
}

function PantallaMenu() {
}
PantallaMenu.prototype = new Pantalla();
PantallaMenu.prototype.constructor = PantallaMenu;
PantallaMenu.prototype.parent = Pantalla.prototype;

function PantallaDetall() {
}
```

```
PantallaDetall.prototype = new Pantalla();
PantallaDetall.prototype.constructor = PantallaDetall;
PantallaMenu.prototype.parent = Pantalla.prototype;

var app = new LamevaApp();
app.inicialitzar();
```

El benefici del polimorfisme en aquest exemple se centra en el mètode `actualitzarPantalla` de la classe `LamevaApp`, en què podem veure com diverses vegades es fa ús de crides a mètodes sobre un objecte del qual no tenim la certesa de quin tipus és, però que es resolen correctament perquè n'unifiquem l'ús mitjançant els mètodes de la classe pare, i per tant garanteix que sigui quin sigui el tipus de la instància (`PantallaMenu` o `PantallaDetall`) tindrem a la nostra disposició els mètodes emprats (tots de la classe pare `Pantalla`).

Per a controlar manualment aquestes situacions, JavaScript disposa de la funció `instanceof`, que ens permet saber si un objecte és instància d'una classe, complint-se sempre que una instància d'una classe filla sempre es considera instància també de la classe pare.

```
instanciaPare instanceof ClassePare; // true
instanciaPare instanceof ClasseFilla; // false
instanciaFilla instanceof ClasseFilla; // true
instanciaFilla instanceof ClassePare; // true
```

En aquest exemple, i per simplificar-ne el codi, les classes `PantallaMenu` i `PantallaDetall` no personalitzen els mètodes `mostrar` i `ocultar`, però podrien fer-ho sense afectar el funcionament del programa, de manera que cadascuna canviï la forma en què es mostra o oculta, simplement sobreescribint aquests mètodes.

En altres llenguatges, aquest mecanisme també es pot aconseguir mitjançant l'ús d'interfícies (mecanisme de POO del qual JavaScript com tal no té suport, no així per exemple TypeScript).

1.4.8. Composició

La composició és un mecanisme que té lloc en POO quan una classe conté com a propietats objectes d'altres classes. Aquest concepte es pot observar en l'exemple anterior, en què la classe `LamevaApp` disposa d'una propietat `pantallaActual` que és de tipus `Pantalla`, per tant entre `LamevaApp` i `Pantalla` s'està donant una relació de composició.

Depenent de les relacions i condicions del problema a representar mitjançant POO, la composició pot ser un mecanisme vàlid com a alternativa a l'herència i, altres vegades, un mecanisme complementari a aquesta.

1.5. Alternativa a l'ús de constructors en POO

A part de l'aproximació a la POO que hem vist durant tot aquest apartat del mòdul, i que s'associa a la POO clàssica, hi ha una alternativa que se sol conèixer com a *prototypal object oriented programming* o *aproximació per prototips* que es basa en l'ús exclusiu de prototips en lloc de funcions constructores.

Hi ha multitud d'articles entre la comunitat de desenvolupadors JavaScript professionals en què s'avaluen aquestes dues alternatives, analitzant-ne els pros i els contres, sense que ara per ara hi hagi un vencedor clar.

1.5.1. 1 POO basada en prototips i Object.create

L'aproximació a POO per prototips en JavaScript es basa en el mètode `create` de la classe `Object` que està disponible des d'ECMAScript 5, i encara que no disposa de suport en IE8 es pot esmenar aquest problema de manera manual, estenent mitjançant una funció pròpia que faci la mateixa tasca, de manera que garantim que disposem d'aquest mètode independentment del navegador que usem:

```
if (typeof Object.create !== "function") {
   Object.create = function (o) {
      function F() {}
      F.prototype = o;
      return new F();
   };
}
```

Object.create

Object.create:

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/create

Suport d'Object.create (ECMAScript 5) en navegadors:

http://kangax.github.io/compat-table/es5/

Aquest mètode rep un objecte com a paràmetre i retorna un nou objecte que té l'objecte original passat com el seu `[[Prototype]]` (`__proto__`).

A partir d'aquest mètode es pot veure la POO bàsica com reflecteix l'exemple següent:

```
var persona = {
   dni: '',
   nom: '',
   amics: null,

   init: function(dni, nom) {
```

```
        this.dni = dni;
        this.nom = nom;
        this.amics = new Array();

        return this;
    },

    saludar: function() {
        console.log('Hola em dic '+this.nom);
    }
    };

var lameva = Object.create(persona).init('1', 'Marc');
mi.saludar();
```

En l'exemple, l'objecte `Persona` (definit mitjançant una notació literal amb format JSON) s'usa com a prototip de la instància `lameva`, de manera que `lameva` tindrà les propietats i mètodes de `Persona`.

Durant aquest mòdul, s'ha treballat amb l'enfocament de POO clàssica, perquè es considera més didàctica, requerir una corba d'aprenentatge més petita per als que vénen d'altres llenguatges no basats en prototips i perquè està alineada amb el futur estàndard ECMAScript 6.

Ampliant aquesta aproximació a l'herència, tindríem un escenari com el següent:

```
var persona = {
    dni: '',
    nom: '',
    amics: null,

    init: function(dni, nom) {
        this.dni = dni;
        this.nom = nom;
        this.amics = new Array();

        return this;
    },

    saludar: function() {
        console.log('Hola em dic '+this.nom);
    }
    };

var estudiant = Object.create(persona);
estudiant.estudiar = function() {
    console.log(this.nom + ' està estudiant.');
```

```
};

var jo = Object.create(estudiant).init('1', 'Marc');
jo.saludar();
jo.estudiar();
```

Com podem veure, tota la cadena jeràrquica es basa en l'ús d'Object.create i els prototips. Com és lògic, entorn d'aquesta manera d'implementar la POO hi ha mètodes per a disposar de propietats i mètodes privats, i fins i tot patrons de disseny que permeten disposar d'herència múltiple.

Un altre detall intencionat, és que els noms de les classes ara van en minúscules, perquè aquest acostament defensa l'ús d'objectes com a tals i, per tant, deixa de tenir sentit la necessitat d'aplicar aquesta convenció habitual en altres llenguatges en parlar de classes.

Com a complement a la forma revisada en l'exemple anterior, és important destacar que el mètode `Object.create` admet un segon paràmetre que seria novament un objecte en notació JSON, que permet estendre precisament directament en la crida a `Object.create` l'objecte passat com a primer paràmetre amb propietats i mètodes addicionals, i fins i tot controlar alguns aspectes de la capacitat de visualització i accés a aquestes propietats i mètodes. L'exemple anterior podria ser més complet i representatiu de la manera següent:

```
var persona = {
   dni: '',
   nom: '',
   amics: null,

   init: function(dni, nom) {
      this.dni = dni;
      this.nom = nom;
      this.amics = new Array();

   return this;
   },

   saludar: function() {
      console.log('Hola em dic '+this.nom);
   }
   };

   var estudiant = Object.create(persona, {
   numeroMatricula: {
      get: function() {
         return this.value;
```

```
        },
        set: function(newValue) {
            this.value = newValue;
        }
    },

    init: {
        value: function(dni, nom, numeroMatricula) {
            persona.init.call(this, dni, nom);
            this.numeroMatricula = numeroMatricula;
            return this;
        }
    },

    estudiar: {
        value: function() {
            console.log('estudiant > '+this.numeroMatricula);
        }
    }
});

    var jo = Object.create(estudiant).init('1','Marc','12');
jo.saludar();
jo.numeroMatricula = '333';
jo.estudiar();
```

En què si ens fixem hem fet les modificacions següents:

1) Tant les propietats com els mètodes passats en l'objecte que actua com a segon paràmetre s'han de declarar al seu torn com a objectes literals. En el cas dels mètodes, contenen una propietat `value` que és la que alberga la funció que representa al mètode.

2) Ara estudiant afegeix una propietat pública, que requereix de l'ús d'un *getter* i un *setter* que amb aquest format es declaren com dos mètodes `get` i `set` en l'objecte que defineix aquesta nova propietat.

3) Se sobreescriu la funció `init`, que actua com a constructora, de manera que s'amplia un paràmetre, i en el seu interior es crida la funció `init` de l'objecte estès mitjançant l'ús de la funció `call` en la línia:

```
persona.init.call(this, dni, nom);
```

4) S'estén amb un nou mètode `estudiar` l'objecte original `persona`, i des del mètode es demostra que es pot accedir a la nova propietat `numeroMatricula`.

Encara que no s'ha representat en l'exemple, aquest mecanisme o enfocament disposa del que es denomina **descriptors de propietat** com ara enumerable, *writeable*, configurable, value, *get* i *set*, i que són els mateixos que s'apliquen mitjançant Object.defineProperty, que també és emprat en aquest enfocament.

Descriptors de propietat en MDN

https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/defineProperty

Seguint aquest exemple, es poden veure molts paral·lelismes amb l'aproximació clàssica, essent les diferències més importants quant al format de declaració, i és que, en realitat, l'única diferència entre usar l'operador `new` i `Object.create` és que el segon NO fa la crida a la funció constructora que anava implícita en el cas del primer.

2. Objectes predefinits

Tal com s'ha dit abans, en JavaScript hi ha un conjunt d'objectes incorporats que permet accedir a moltes de les funcions que hi ha en qualsevol altre llenguatge de programació. Ens referim als objectes Object, Array, Boolean, Date, Function, Math, Number, String i RegExp.

En aquest mòdul es presentaran les propietats que es fan servir més i que suporten comunament els navegadors principals.

Web recomanat

Es pot consultar en línia l'especificació completa dels objectes predefinits en l'estàndard ECMA-262.

2.1. Els objectes Object i Function

2.1.1. L'objecte Object

Es tracta de l'objecte pare o avi a partir de qual hereten tots els objectes que hi ha en el llenguatge o que es crearan en el llenguatge. Així, aquest objecte defineix les propietats i els mètodes que són comuns a tots els objectes, de manera que cada objecte particular podrà reescriure mètodes o propietats si ho necessita per adequar-lo a l'objectiu que té.

Un ús de l'objecte, encara que no és gaire comú o recomanable, és com a mètode de creació alternatiu d'objectes. Per exemple:

```
var cotxe = new Object();
cotxe.marca = "Ford";
cotxe.model = "Focus";
cotxe.aireAcon = true;
cotxe.mostra = function()
               {
               alert(this.marca +" "+this.model);
               }
```

En l'exemple anterior s'ha creat un objecte cotxe a partir d'Object.

Una altra de les estructures que es generen amb l'ús d'Object són les arrays associatives. A diferència de les bàsiques, en les associatives cada element de l'array es referencia pel nom que s'hi ha assignat i no amb l'índex que n'indica la posició.

Vegeu també

Les arrays associatives s'han tractat en l'apartat 2 d'aquest mòdul.

Tot seguit es presenta un exemple de l'ús d'aquestes arrays:

```
var adreces = new Object();
adreces["Víctor"] = "Santiago de Cuba";
```

```
adreces["Pablo"] = "Madrid";
adreces["Miquel"] = "València";
```

D'aquesta manera, es pot recuperar l'adreça de Miquel utilitzant la sintaxi següent:

```
var adrMiq = adreces["Miquel"];
```

Per tant, com es pot veure, simplement se simula una array amb l'assignació dinàmica de propietats a un objecte del tipus Object. Malgrat que sigui un array associatiu i en l'exemple es vegi el seu accés "dinàmic", com objectes que són és més aconsellable usar la notació de punt.

Taula 1. Propietats de l'objecte Object

Nom	Descripció
constructor	Referència a la funció que s'ha cridat per a crear l'objecte genèric.
prototype	Representa el prototip per a la classe.

Taula 2. Mètodes de l'objecte Object

Nom	Descripció
toString()	Torna a l'objecte una cadena, de manera predeterminada "[objecte Object]".
valueOf()	Torna el valor primitiu associat amb l'objecte, de manera predeterminada "[objecte Object]".

2.1.2. L'objecte Function

És l'objecte de què deriven les funcions de JavaScript; proporciona propietats que transmeten informació útil durant l'execució de la funció. Un exemple d'aquestes propietats és l'array arguments[ ].

El constructor per a l'objecte Function és aquest:

```
new Function (arg1, arg2..., argN, funció)
```

en què:

- arg1, arg2,..., argN: paràmetres opcionals, de la funció.
- funció: és una cadena que conté les sentències que componen la funció.

La propietat arguments [ ] permet saber el nombre d'arguments que s'han passat en la crida a la funció.

En l'exemple següent, es crea una funció, a dins de la qual hi ha un bucle for l'extrem superior del qual el defineixen el nombre d'arguments de la funció mateixa, ja que l'objectiu del bucle és manipular els arguments que s'han passat en la crida a la funció, però dels quals *a priori* no se sap el nombre.

```
function llista(tipus) {
        document.write("<" + tipus + "l>");
        for (var i=1; i<llista.arguments.length; i++) {
                document.write("<li>" + llista.arguments[i]);
  }
  document.write("</" + tipus + "l>");
}
```

La crida a la funció amb els arguments següents:

```
llista("u", "U", "Dos", "Tres");
```

té com a resultat el següent:

```
<ul>
<li>u
<li>Dos
<li>Tres
</ul>
```

Taula 3. Propietats de l'objecte function

Nom	Descripció
arguments	Array amb els paràmetres que s'han passat a la funció.
caller	Nom de la funció que ha cridat la que s'executa.
constructor	Referència a la funció que s'ha cridat per a crear l'objecte.
length	Nombre de paràmetres que s'han passat a la funció.
prototype	Valor a partir del qual es creen les instàncies d'una classe.

La propietat caller

Aquesta propietat només és accessible des del cos de la funció. Si es fa servir fora de la funció mateixa, el valor és null.

Taula 4. Mètodes de l'objecte function

Nom	Descripció	Sintaxi	Paràmetres
apply	Crida la funció.	apply(obj[,arg1,arg2...argN])	obj: nom de la funció arg1,..., argN: llista d'arguments de la funció
call	Crida la funció.	call(obj[,args])	obj: nom de la funció args: array d'arguments de la funció
toString	Retorna una string que representa l'objecte especificat.	toString()	
valueOf	Retorna el valor primitiu associat a l'objecte.	valueOf()	

2.2. Els objectes Array, Boolean i Date

2.2.1. L'objecte Array

Les arrays emmagatzemen llistes ordenades de dades heterogènies. Les dades s'emmagatzemen en índexs enumerats començant des de zero, als quals s'accedeix utilitzant l'operand d'accés a arrays ([ ]).

Nota

En el mòdul "Programació en el costat del client: llenguatges script en els navegadors" vam veure una introducció a l'objecte Array, en aquest apartat repassarem i ampliarem els conceptes estudiats anteriorment.

Crear arrays

En el mòdul "Programació en el costat del client: llenguatges script en els navegadors" vam veure que el constructor de l'objecte admet les sintaxis següents:

```
var vector1 = new Array();
var vector2 = new Array(longitud);
var vector3 = new Array(element0, element1,..., elementN);
```

La primera sintaxi del constructor crea una array buida sense una dimensió definida; en la segona sintaxi, s'hi passa la grandària de l'array com a paràmetre, i en el tercer constructor, s'hi passen els valors que s'emmagatzemen en l'array.

Per exemple, es pot definir la longitud de l'array i després emplenar cadascuna de les posicions que té:

```
colors = new Array(16);
colors[0] = "Blau";
colors[1] = "Groc";
colors[2] = "Verd";
```

o bé començar l'array en el moment mateix de crear-la:

```
colors = new Array("Blau", "Vermell", "Verd");
```

També hi ha la possibilitat de crear arrays utilitzant literals d'array. Per exemple:

```
var vector1 = [ ];
var vector2 = [,,,,,,,,,,,,];
var vector3 = ["element0", "element1",..., "elementN"];
```

Accedir als elements

L'accés als elements d'una array es fa utilitzant el nom de l'array seguit de l'índex de l'element que s'ha de consultar tancat entre claudàtors.

Tot seguit es mostra un exemple d'accés als elements d'una array:

```
var vector = new Array("Plàtan", "Poma", "Pera");
var fruita1 = vector[0];
```

Afegir i modificar elements a una array

Recordem que en JavaScript la mida de l'array és gestionat directament pel llenguatge i podem afegir elements indicant un índex sense necessitat de realitzar-lo de manera consecutiva. Podem realitzar-lo de la manera següent:

```
var vector1 = [2, 54, 16];
vector1[33] = 25
```

En modificar o afegir elements a una array, s'ha de tenir en compte que les arrays són objectes o tipus per referència, de manera que si s'ha assignat una array a dues variables, la modificació d'un element de l'array afecta les dues variables i no solament aquella des de la qual s'ha modificat:

```
var vector2 = [2, 4, 6, 8];
var vector3 = vector2;
vector3[0] =1;
```

L'assignació del valor 1 a l'array vector3 no solament modifica el contingut d'aquesta array sinó també el de l'array vector2.

Eliminar elements d'una array

Els elements d'una array es poden eliminar utilitzant la sentència delete:

```
delete vector1[3];
```

La sentència anterior elimina el quart element de l'array vector1, en el sentit que hi assigna el valor undefined, però no modifica la grandària de l'array.

Nota

"Delete" en realitat no esborra sinó que, en aquesta posició, deixa un valor "undefined". Això té connotacions amb el que es pot esperar de la propietat "length" després d'usar "delete". Per esborrar i eliminar de veritat aquest element s'usa "splice". "Splice" s'explica en detall més endavant, ja que té altres usos a més d'eliminar.

La propietat length

Aquesta propietat ens retorna la següent posició disponible al final de l'array, sense tenir en compte els índexs buits que hi hagi al mig. La sintaxi és la següent:

```
longitud = vector1.length;
```

El mètode sort

El mètode sort requereix que s'indiqui la funció de comparació per a ordenar l'array. Si no s'especifica aquesta funció, els elements de l'array es converteixen en strings i s'ordenen alfabèticament. Per exemple, Barcelona aniria abans que Saragossa, i 80 abans que 9.

En el cas de no especificar la funció de comparació, els elements s'ordenen segons el retorn d'aquesta funció:

- Si compara(a,b) és més petit que 0, b té un índex en l'array més petit que a.
- Si compara(a,b) és 0, no modifica les posicions.
- Si compara(a,b) és més gran que 0, b té un índex en l'array més gran que a.

La funció de comparació té la forma següent:

```
function compara(a, b) {
       if (a < b per un criteri d'ordenació)
                return -1;
       if (a > b per un criteri d'ordenació)
                return 1;
       return 0; //són iguals
}
```

El criteri d'ordenació definirà si l'ordenació serà numèrica, alfanumèrica o qualsevol que el programador pugui definir.

Per a ordenar números, la funció es defineix de la manera següent:

```
function compara(a, b){
       if (a < b)
                return -1;
       else if (a === b)
                return 0;
       else
                return 1;
}
```

En una array amb els valors 1, 2 i 11, si s'aplica el mètode sort() sense modificar la funció de comparació, el resultat de l'ordenació és 1, 11, 2, és a dir, s'ordenen els valors alfabèticament.

En l'exemple següent es mostra com s'ha de fer per a modificar la funció de comparació:

```
numeros = new Array(1, 2, 11);
function compara(a, b){
       if (a < b)
                return -1;
       else if (a === b)
                return 0;
       else
                return 1;
}

function ordena(){
       numeros.sort(compara);
       document.write(numeros);
}
```

El resultat d'aquest exemple és el següent: 1, 2, 11.

Simulació de piles LIFO

Una pila LIFO és una estructura que s'utilitza per a emmagatzemar dades seguint l'ordre de darrer d'entrar, primer de sortir. L'analogia és una pila de documents sobre una taula de manera que el primer que retirem per estudiar coincideix amb el darrer que posem a la taula.

Per a simular l'ús de piles amb arrays, s'utilitzen els mètodes push() i pop(); quan es crida el mètode push(), s'afegeixen els arguments donats al final de l'array i s'incrementa la grandària d'aquesta array, que es reflecteix en la propietat length.

El mètode pop() elimina el darrer element de l'array, el torna i disminueix la propietat length en una unitat.

En l'exemple següent es veuen aquests mètodes en acció:

```
var pila = [ ];          //[ ]
pila.push("primer");    //["primer"]
pila.push(15, 30);      //["primer",15, 30]
pila.pop();             //["primer",15] i torna el valor 30
```

El fet que aquests dos mètodes es facin servir per a simular piles no implica que no es puguin utilitzar els mètodes per a inserir i eliminar valors situats al final de l'array.

Simular cues FIFO

Una cua FIFO es basa en l'ordre de primer d'entrar, primer de sortir. L'analogia és, per exemple, qualsevol cua per a pagar en un comerç o a l'entrada del cinema. JavaScript disposa dels mètodes unshift() i shift(), que, a diferència dels dos anteriors, afegeixen i eliminen dades del començament d'array.

D'aquesta manera, unshift() introdueix els arguments al principi de l'array i fa que els elements que hi ha canviïn a índexs més alts, de manera que, com és d'esperar, augmenta el valor de la propietat length.

El mètode shift(), per la seva banda, elimina el primer element de l'array, el torna i redueix l'índex de la resta d'elements de l'array, i acaba amb la disminució obligatòria de la propietat length.

En l'exemple següent se'n veu el comportament:

```
var cua = ["Joan", "Pere", "Andreu", "Vicenç"];
cua.unshift("Clàudia", "Raquel");    //cua contindrà ["Clàudia", "Raquel", "Joan",
                                     //"Pere", "Andreu", "Vicenç"]
var primer = cua.shift();            //primer contindrà el valor "Clàudia"
```

La simulació d'una cua es fa combinant els mètodes push(), que afegeix elements al final de la cua, i shift(), que extreu el primer element d'aquesta cua:

```
var cua = ["Joan", "Pere", "Andreu", "Vicenç"];
cua.push("Clàudia");            //cua contindrà ["Joan","Pere", "Andreu", "Vicenç","Clàudia"]
var seguent = cua.shift();    //seguent contindrà el valor "Joan"
```

Concatenar arrays

El mètode concat() torna l'array resultat d'afegir els arguments a l'array sobre la qual s'ha fet la crida. Per exemple:

```
var color = ["Vermell", "Verd", "Blau"];
var colorAmp = color.concat("Blanc", "Negre");
```

En l'exemple anterior s'obté una array colorAmp nova amb el contingut següent: ["Vermell", "Verd", "Blau", "Blanc", "Negre"]. L'array color, en canvi, no s'ha modificat.

Es poden concatenar dues arrays simplement si es compleix la condició que l'argument del mètode concat() sigui una array.

Conversió en una cadena

Per a convertir una array en una cadena, s'utilitza el mètode join(), que a més de la conversió permet especificar per mitjà del paràmetre que té com se separaren els elements de la cadena. Es fa servir join() per a mostrar els elements d'una array amb un separador específic:

```
var color = ["Vermell", "Verd", "Blau"];
var cadena = color.join("-");
```

En l'exemple anterior, la variable *cadena* adquireix el contingut següent: Vermell-Verd-Blau.

Invertir l'ordre dels elements

El mètode reverse() inverteix l'ordre dels elements d'una array i, a diferència dels dos mètodes anteriors, la mateixa array emmagatzema els elements, amb l'ordre invertit:

```
var color = ["Vermell", "Verd", "Blau"];
color.reverse();
```

En l'exemple anterior l'array color, després d'executar el mètode, té el contingut següent: ["Blau", "Verd", "Vermell"].

Extreure fragments d'una array

El mètode slice() torna un fragment de l'array sobre la qual es crida; no actua realment sobre l'array (tal com fa reverse()), i queda intacta. El mètode agafa dos arguments, l'índex inicial i el final, i torna una array que conté els elements que hi ha entre l'índex inicial i el final (exclòs el final).

En cas que només rebi un argument, el mètode torna l'array que componen tots els elements des de l'índex indicat fins al final de l'array.

Una característica interessant és el fet que admet valors negatius per als índexs i, quan aquests índexs són negatius, es compten des del final de l'array. En els exemples següents es veu l'ús d'aquest mètode:

```
var carrera = [21, 25, 12, 23];
carrera.slice(2);      //torna [12, 23]
carrera.slice(1,3);    //torna [25, 12]
carrera.slice(-2, -1)  //torna [12]
```

Afegir, eliminar i modificar elements d'una array

El mètode splice() es pot utilitzar per a afegir, reemplaçar o eliminar elements en una array i torna els elements que s'han eliminat. Dels arguments que pot agafar, l'únic que és obligatori és el primer, la sintaxi del qual és la següent:

```
splice(inici, elementsEsborrar, elementsAfegir);
```

El significat dels arguments és el següent:

- inici: indica l'índex a partir del qual es fa l'operació.
- elementsEsborrar: nombre d'elements que s'eliminen, començant pel que marca el primer paràmetre. Si s'omet aquest paràmetre, s'eliminen tots els elements des del començament fins al final de l'array i, tal com s'ha dit, són tornats (es poden emmagatzemar en una variable).
- elementsAfegir: llista d'elements separats per comes, no obligatòria, que substitueixen els eliminats.

Tot seguit, se'n mostra un exemple d'ús:

```
var carrera = [21, 25, 12, 23];
carrera.splice(2, 2);  //torna els elements eliminats [12, 23] i l'array
                       //contindrà els valors [21, 25]
carrera.splice(2, 0, 31, 33);  //no elimina cap valor i per tant torna [ ]
                               //i afegeix els valors [31, 33] a la cadena
```

Arrays multidimensionals

Una array multidimensional és una array en què cada element és, al seu torn, una array. En l'exemple següent es defineix una array bidimensional:

```
var matriu = [[1,2,3],[4,5,6],[7,8,9]];
```

L'accés a les arrays multidimensionals es fa a partir de la concatenació de claudàtors que indiquen els elements a què s'accedeix:

```
matriu[0][1];   //torna el valor 2, ja que accedim al segon
                //element de la primera array
```

Ús de prototips

Tal com s'ha explicat, es poden afegir mètodes i propietats nous a qualsevol objecte utilitzant els prototips. Tot seguit es defineix un mètode mostra() nou, que ensenya en una finestra el contingut d'una array:

```
function mMostra(){
       if (this.length != 0)
```

```
                alert(this.join());
        else
                alert("L'array és buida");
}
Array.prototype.mostra = mMostra;
```

Per acabar l'objecte Array, es mostra en una taula les propietats i mètodes principals de l'objecte.

Taula 5. Propietats de l'objecte Array

Nom	Descripció
constructor	Referència a la funció que s'ha cridat per a crear l'array actual.
length	És el nombre d'elements de l'array.
prototype	Representa el prototip per a la classe.

Taula 6. Mètodes de l'objecte Array

Nom	Descripció	Sintaxi	Paràmetres	Retorn
concat	Concatena dues arrays.	concat(array2)	array2: nom de l'array que s'ha de concatenar a la que ha cridat el mètode.	Array nova, unió de totes dues.
join	Uneix tots els elements de l'array en una string, separats pel símbol indicat.	join(separador)	separador: signe que separa els elements de l'string.	String.
pop	Esborra el darrer element de l'array.	pop()		L'element esborrat.
push	Afegeix un element o més d'un al final de l'array.	push(elt1,..., eltN)	elt1,..., eltN: elements que s'han d'afegir.	El darrer element afegit.
reverse	Trasllada els elements de l'array.	reverse()		
shift	Elimina el primer element de l'array.	shift()		L'element eliminat.
slice	Extreu una part de l'array.	slice(inici,final)	inici: índex inicial de l'array que s'ha d'extreure. final: índex final de l'array que s'ha d'extreure.	Una array nova amb els elements extrets.
splice	Canvia el contingut d'una array, de manera que hi afegeix elements nous i, alhora, elimina els que ja hi havia.	splice(índex, quants, nouEl1,..., nouElN)	índex: índex inicial a partir del qual es comença a canviar. quants: nombre d'elements que s'han d'eliminar. nouEl1,..., nouElN: elements que s'han d'afegir.	Array d'elements eliminats.
sort	Ordena els elements de l'array.	sort(compara)	compara: funció que defineix l'ordre dels elements.	
toString	Converteix els elements de l'array a text.	toString()		String que conté els elements de l'array passats a text.

Nom	Descripció	Sintaxi	Paràmetres	Retorn
unshift	Afegeix un element o més d'un al començament de l'array.	unshift(elt1,..., eltN)	elt1,..., eltN: elements que s'han d'afegir.	La longitud nova de l'array.

2.2.2. L'objecte Boolean

És l'objecte incorporat en el llenguatge que representa les dades de tipus lògic. Es tracta d'un objecte molt simple, ja que no disposa de propietats ni mètodes exceptuant els que hereta de l'objecte Object. El constructor de l'objecte és el següent:

```
new Boolean(valor)
```

Si el paràmetre s'omet, o té els valors 0, null o false, l'objecte agafa el valor inicial com a false. En qualsevol altre cas, agafa el valor true. Aquestes característiques fan que aquest objecte es pugui usar per a convertir un valor no booleà a booleà.

Taula 7. Propietats de l'objecte Boolean

Nom	Descripció
constructor	Referència a la funció que s'ha cridat per a crear l'objecte.
prototype	Representa el prototip per a la classe.

Taula 8. Mètodes de l'objecte Boolean

Nom	Descripció	Sintaxi	Retorn
toString	Representa un objecte mitjançant una string.	toString()	"true" o "false" segons el valor de l'objecte.
valueOf	Obtenir el valor que té l'objecte.	valueOf()	string "true" o false" segons el valor que té.

2.2.3. L'objecte Date

L'objecte Date proporciona una varietat extensa de mètodes que permeten manipular dates i hores. L'objecte, però, no conté un rellotge en funcionament, sinó un valor de data estàtic que, a més, internament s'emmagatzema com el nombre de mil·lisegons des de les dotze de la nit de l'1 de gener de 1970.

El constructor de l'objecte pot rebre un ventall interessant de paràmetres. Tot seguit es presenten els que es fan servir més:

```
new Date()
new Date(any_num, mes_num, dia_num)
new Date(any_num, mes_num, dia_num, hora_num, min_num, seg_num)
```

El significat dels paràmetres és el següent:

any_num, mes_num, dia_num, hora_num, min_num, seg_num: són enters que formen part de la data. En aquest cas, s'ha de tenir en compte que el mes 0 correspon al gener, i el mes 11, al desembre.

Treballar amb dates

En l'exemple següent, es pot veure l'ús de les funcions que permeten mostrar la data actual en la pàgina del navegador:

```
mesos = new Array("Gener", "Febrer", "Març", "Abril", "Maig", "Juny", "Juliol",
"Agost", "Setembre", "Octubre", "Novembre", "Desembre");
var data = new Date();
var mes = data.getMonth();
var any = data.getFullYear();
document.write("Avui és" + data.getDate() + "de" + mesos[mes] + "de" + any);
```

Tal com es veurà tot seguit, els objectes Date disposen d'un conjunt molt ampli de mètodes que permeten establir o llegir una propietat directament de l'objecte fent internament les conversions que facin falta, ja que, tal com s'ha indicat abans, l'objecte emmagatzema els valors en mil·lisegons.

Taula 9. Propietats de l'objecte Date

Nom	Descripció
constructor	Referència a la funció que s'ha cridat per a crear l'objecte.
prototype	Representa el prototip per a la classe. És de lectura i prou.

Taula 10. Mètodes de l'objecte Date

Nom	Descripció	Sintaxi	Paràmetres	Retorn
getDate	Retorna el dia del mes per a una data.	getDate()		Enter entre 1 i 31.
getDay	Retorna el dia de la setmana per a una data.	getDay()		Enter entre 0 i 6. El 0 és diumenge, l'1 és dilluns, etc.
getHours	Retorna l'hora per a una data.	getHours()		Enter entre 0 i 23.
getMinutes	Retorna els minuts per a una data.	getMinutes()		Enter entre 0 i 59.
getMonth	Retorna el mes per a una data.	getMonth()		Enter entre 0 i 11.
getSeconds	Retorna els segons per a una data.	getSeconds()		Enter entre 0 i 59.

Nom	Descripció	Sintaxi	Paràmetres	Retorn
getTime	Retorna un nombre que correspon al temps transcorregut per a una data.	getTime()		Nombre de mil·lisegons transcorreguts des de l'1 de gener de 1970 a les 00.00.00 hores.
getTimezoneOffset	Retorna la diferència horària, entre l'hora local i l'hora GMT (*Greenwich mean time*).	getTimezoneOffset()		Nombre de minuts que marca la diferència horària.
getFullYear	Retorna l'any per a una data.	getFullYear()		Retorna tots quatre dígits.
parse	String que conté els mil·lisegons transcorreguts per a la data.	Date.parse(dataString)	dataString: data en format string.	Mil·lisegons transcorreguts des de l'1 de gener de 1970 a les 00.00.00 hores.
setDate	Posa el dia a una data.	setDate(valordia)	valordia: enter entre 1 i 31 que representa el dia.	
setDay	Posa el dia a una data.	setDate(valordia)	valordia: enter entre 1 i 31 que representa el dia.	
setHours	Posa l'hora a una data.	setHours(valorhora)	valorhora: enter entre 0 i 23 que representa l'hora.	
setMinutes	Posa els minuts a una data.	setMinutes(valorminuts)	valorminuts: enter entre 0 i 59 que representa els minuts.	
setMonth	Posa el mes a una data.	setMonth(valormes)	valormes: enter entre 0 i 11 que representa el mes.	
setSeconds	Posa els segons a una data.	setSeconds(valorsegons)	valorsegons: enter entre 0 i 59 que representa els segons.	
setTime	Posa el valor a una data.	setTime(valorhorari)	valorhorari: mil·lisegons transcorreguts des de l'1 de gener de 1970 a les 00.00.00 hores.	
setYear	Posa l'any a una data.	setYear(valorany)	valorany: enter que representa l'any.	
toGMTString	Converteix una data a string, fent servir les convencions GMT d'Internet.	toGMTString()		
toLocaleString	Converteix una data a string, fent servir les convencions locals. Vegeu retorn.	toLocaleString()		Exemple: dissabte, 07 de desembre de 2002 21.22.59.
toLocaleDateString	Converteix una data a string, fent servir les convencions locals. Vegeu retorn.	toLocaleDateString()		Exemple: Sat Dec 7 21:22:59 UTC +0100 2002.
toLocaleTimeString	Converteix una data a string, fent servir les convencions locals.			
UTC	Mil·lisegons transcorreguts per a la data indicada.	Date.UTC(any, mes, dia, hores, min, s)	Enters. Els tres darrers són opcionals.	Mil·lisegons transcorreguts des de l'1 de gener de 1970 a les 00.00.00 hores.

2.3. Els objectes Math, Number i String

2.3.1. L'objecte Math

L'objecte Math és format per un conjunt de constants i mètodes que permeten fer operacions matemàtiques de certa complexitat. Aquest objecte és estàtic, de manera que no es pot crear una instància i l'accés a les constants i els mètodes que té es fa directament utilitzant l'objecte mateix.

L'exemple següent mostra un conjunt de càlculs que s'han fet amb l'objecte Math:

```
function calcula(nombre){
    cosValue = Math.cos(nombre); //Emmagatzema a cosValue el valor del cosinus
    sinValue = Math.sin(nombre); //Emmagatzema a sinValue el valor del sinus
    tanValue = Math.tan(nombre); //Emmagatzema a cosValue el valor de la tangent
    sqrtValue = Math.sqrt(nombre); //Emmagatzema a sqrtValue el valor de l'arrel
    powValue = Math.pow(nombre,3); //Emmagatzema a powValue el valor d'elevar a 3 el valor
    expValue = Math.exp(nombre); //Emmagatzema a expValue el valor de e elevat al valor del nombre
}
```

Taula 11. Propietats de l'objecte Math

Nom	Descripció	Exemple

E	Constant d'Euler. El valor aproximat que té és 2,718. És només de lectura.	function valorE() {
		return **Math.E**
		}

LN10	Logaritme neperià de 10. El valor aproximat que té és 2,302. És només de lectura.	function valorLN10() {
		return **Math.LN10**
		}

LN2	Logaritme neperià de 2. El valor aproximat que té és 0,693. És només de lectura.	function valorLN2() {
		return **Math.LN2**
		}

LOG10E	Logaritme en base 10 del nombre E. El valor aproximat que té és 0,434. És només de lectura.	function valorLog10e() {
		return **Math.LOG10E**
		}

LOG2E	Logaritme en base 2 del nombre E. El valor aproximat que té és 1,442. És només de lectura.	function valorLog2e() {
		return **Math.LOG2E**
		}

PI	Nombre pi. El valor aproximat que té és 3,1415. És només de lectura.	function valorPi() {
		return **Math.PI**
		}

SQRT1_2	1 dividit per l'arrel quadrada de 2. El valor aproximat que té és 0,707. És només de lectura.	function valorSQRT1_2() {
		return **Math.SQRT1_2**
		}

SQRT2	Arrel quadrada de 2. El valor aproximat que té és 1,414. És només de lectura.	function valorSQRT2() {
		return **Math.SQRT2**
		}

Taula 12. Mètodes de l'objecte Math

Nom	Descripció	Sintaxi	Paràmetres	Retorn
abs	Calcula el valor absolut d'un nombre.	abs(x)	x: un nombre.	Valor absolut de *x*.
acos	Calcula l'arc cosinus d'un nombre.	acos(x)	x: un nombre.	Valor de l'arc cosinus de *x* en radiants. Valor entre [0, PI].
asin	Calcula l'arc sinus d'un nombre.	asin(x)	x: un nombre.	Valor de l'arc sinus de *x* en radiants. Valor entre [-PI/2,PI/2].
atan	Calcula l'arc tangent d'un nombre.	atan(x)	x: un nombre.	Valor de l'arc tangent de *x* en radiants. Valor entre [-PI/2,PI/2].
atan2	Calcula l'arc tangent del quocient de dos nombres.	atan2(y,x)	y: un nombre (coordenada y). x: un nombre (coordenada x).	
ceil	Retorna, més gran o igual que un nombre, l'enter més petit.	ceil(x)	x: un nombre.	El primer enter més gran o igual que *x* (per a 9,8 en retorna 10).
cos	Calcula el cosinus d'un nombre.	cos(x)	x: un nombre.	Valor del cosinus de *x*. Valor entre [-1,1].
exp	Calcula el valor del nombre E elevat a un nombre.	exp(x)	x: un nombre.	Valor de E^x.
floor	Retorna, més petit o igual que un nombre, l'enter més gran.	floor(x)	x: un nombre.	El primer enter més petit o igual que *x* (per a 9,8 en retorna 9).
log	Calcula el logaritme en base E d'un nombre.	log(x)	x: un nombre.	El logaritme en base E de *x*.
max	Retorna el més gran de dos nombres.	max(x,y)	x: un nombre. y: un nombre.	Si $x > y$, retorna *x*. Si $y > x$, retorna *y*.
min	Retorna el més petit de dos nombres.	min(x,y)	x: un nombre. y: un nombre.	Si $x > y$, retorna y. Si $y > x$, retorna *x*.
pow	Calcula la potència entre dos nombres.	pow(x,y)	x: un nombre que representa la base. y: un nombre que representa l'exponent.	Valor de x^y.
random	Retorna un nombre aleatori entre 0 i 1.	random()		Nombre aleatori entre [0,1].
round	Arrodoneix un nombre a l'enter més proper.	round(x)	x: un nombre.	El valor arrodonit de *x*.
sin	Calcula el sinus d'un nombre.	sin(x)	x: un nombre.	Valor del sinus de *x*. Valor entre [-1,1].
sqrt	Calcula l'arrel quadrada d'un nombre.	sqrt(x)	x: un nombre.	Arrel quadrada de *x*.
tan	Calcula la tangent d'un nombre.	tan(x)	x: un nombre.	Valor de la tangent de *x*.

2.3.2. L'objecte Number

Aquest objecte es fa servir sobretot per a accedir a mètodes de formatació de nombres, a més de proporcionar propietats estàtiques que defineixen constants numèriques. El constructor de l'objecte Number és el següent:

```
new Number (valor)
```

en què valor és el valor numèric que hi haurà en l'objecte creat. Si el paràmetre és omès, la instància s'inicialitza amb el valor 0.

Taula 13. Propietats de l'objecte Number

Nom	Descripció	Exemple d'ús
Constructor	Referència a la funció que s'ha cridat per a crear l'objecte.	

MAX_VALUE	És el valor numèric més gran que es pot representar en JavaScript. És només de lectura.	`if (nombre < = Number.MAX_VALUE)`
		`lamevaFuncio(nombre)`
		`else`
		`alert("nombre massa gran")`

MIN_VALUE	És el valor numèric més petit que es pot representar en JavaScript. És només de lectura.	`if (nombre >= Number.MIN_VALUE)`
		`lamevaFuncio(nombre)`
		`else`
		`lamevaFuncio(0)`

NaN	Indica que no es tracta d'un valor numèric (Not-A-Number). És només de lectura.	
NEGATIVE_INFINITY	Representa el valor negatiu infinit. És només de lectura.	
POSITIVE_INFINITY	Representa el valor positiu infinit. És només de lectura.	
Prototype	Representa el prototip per a la classe.	

Valors de l'objecte Number

- El nombre més gran que es pot representar en JavaScript és 1.79E+308. Els valors més grans que aquest adquireixen el valor Infinity.
- El nombre més petit que es pot representar en JavaScript és 5E-324. La propietat MIN_VALUE no representa el nombre negatiu més petit, sinó el positiu, més pròxim al 0, que pot representar el llenguatge. Els valors més petits que aquest es converteixen en 0.
- Les propietats MAX_VALUE, MIN_VALUE, POSITIVE_INFINITY i NEGATIVE_INFINITY són estàtiques. Per tant, s'han d'utilitzar amb l'objecte mateix, per exemple: Number.MAX_VALUE.
- Per a veure el valor de la propietat NaN, s'ha de fer servir la funció isNaN.
- Un valor més petit que NEGATIVE_INFINITY es visualitza com a -Infinity.
- Un valor més gran que POSITIVE_INFINITY es visualitza com a Infinity.

Taula 14. Mètodes de l'objecte Number

Nom	Descripció	Sintaxi	Paràmetres	Retorn
toExponential	Expressa un nombre en notació exponencial.	toExponential(n)	n: nombre de decimals	String
toFixed	Per a arrodonir a partir del nombre de decimals indicat.	toFixed(n)	n: nombre de decimals	String
toString	String que representa l'objecte especificat.	toString([base])	base: nombre entre 2 i 16 que indica la base en què es representa el nombre.	String
toPrecision	La dada expressada segons la precisió indicada.	toPrecision(n)	n: nombre de dígits de precisió.	String
valueOf	Obté el valor que té l'objecte.	valueOf()		String

2.3.3. L'objecte String

L'objecte String és el contenidor de tipus primitius de tipus cadena de caràcters. Proporciona un conjunt molt ampli de mètodes que permeten manipular i extreure cadenes i convertir-les a text HTML.

La sintaxi del constructor de l'objecte String és la següent:

```
new String(text)
```

en què text és una cadena de caràcters opcional en el constructor.

Els mètodes de l'objecte es poden cridar en cadenes primitives, és a dir, sense haver creat l'objecte amb el constructor. Aquesta característica fa que la creació de cadenes amb la sintaxi anterior sigui poc comuna:

```
var cadena = "Hola, Món";
longitud = cadena.length;
```

La propietat length de l'objecte String, a diferència de l'objecte Array, no la pot establir el programador; es tracta d'una propietat de lectura i prou i canvia quan la cadena modifica la grandària que té en caràcters.

En general, tots els mètodes de l'objecte String no modifiquen el contingut de l'objecte, sinó que tornen el resultat de l'acció, encara que sense modificar el contingut inicial de la cadena; per a modificar una cadena, s'hi ha d'assignar un valor nou:

```
var cadena = "Hola, Món";
cadena.toUpperCase();
```

La variable *cadena* continua emmagatzemant el valor "Hola, Món"; la funció ha actuat i ha tornat la cadena canviant els caràcters a majúscules, però no ha modificat la variable que conté la cadena original:

```
var cadena = "Hola, Món";
cadena = cadena.toUpperCase();
```

En aquest cas, s'ha modificat la variable *cadena* i ara té el contingut "HOLA, MÓN"

Treballar amb cadenes

El mètode charAt(enter) torna el caràcter que hi ha en la posició que indica el nombre que s'ha passat com a paràmetre; s'ha de tenir en compte que, igual que en les arrays, el primer caràcter d'una cadena té l'índex 0:

```
var cadena = "Hola, Món";
caracter = cadena.charAt(2); //La variable caracter recupera el valor "l".
```

El mètode indexOf(cadena) torna l'índex de la primera aparició de l'argument a la cadena. Seguint l'exemple anterior:

```
var cadena = "Hola, Món";
posicio = cadena.indexOf("Món"); //La variable posicio recupera el valor 6.
```

En cas que l'argument no sigui a la cadena, el mètode torna el valor -1. El mètode accepta un segon paràmetre opcional que especifica l'índex des del qual s'ha de començar la cerca:

```
var cadena = "Hola, Món";
posicio = cadena.indexOf("ó",3); //La variable posicio recupera el valor 7.
```

El mètode lastIndexOf(cadena) torna l'índex de la darrera aparició de la cadena passada com a argument. Igual que l'anterior, disposa d'un argument opcional que indica l'índex en què s'ha d'acabar la cerca:

```
var cadena = "Hola, Món";
posicio = cadena.lastIndexOf("o",3); //La variable posicio recupera el valor 1.
```

Igual que l'anterior, torna el valor -1 quan no troba la cadena que busca.

El mètode substring(inici,final) fa una extracció de la cadena, de manera que el primer argument especifica l'índex en què comença la cadena que es vol extreure i el segon argument (opcional) assenyala el final de la subcadena.

En cas que no s'hi passi el segon argument, s'extreu la subcadena fins al final de la cadena original:

```
var cadena = "Hola, Món";
var subCadena = cadena.substring(5); //subCadena adquireix el valor " Món"
var subCadena2 = cadena.substring(5,7); //subCadena2 adquireix el valor " M"
```

El mètode concat() fa la concatenació de totes les cadenes que s'hi passen com a paràmetres (accepta qualsevol quantitat d'arguments) i torna la cadena resultat de concatenar la cadena original amb les que s'hi han passat com a paràmetres:

```
var cadena = "Hola, Món";
var cadena2 = cadena.concat(" lliure ", "i feliç"); //cadena2 adquireix el valor
"Hola, Món lliure i feliç"
```

La concatenació, però, és més comuna de fer-la amb l'operador +:

```
var cadena2 = cadena + " lliure " + "i feliç";
```

El mètode split() divideix la cadena en cadenes separades segons un delimitador passat com a argument del mètode; el resultat s'emmagatzema en una array:

```
var vector = "Hola, Món lliure".split(" ");
```

L'exemple anterior assigna a la variable *vector* una array amb tres elements: "Hola,", "Món", "lliure".

Taula 15. Propietats de l'objecte String

Nom	Descripció	Exemple
constructor	Referència a la funció que s'ha cridat per a crear l'objecte.	
length	Longitud de l'string	`var tema = "Programació"` `alert("La longitud de la paraula Programació és " + tema.length)`

Taula 16. Mètodes de l'objecte String

Nom	Descripció	Sintaxi	Paràmetres	Retorn
anchor	Crea una àncora HTML.	anchor(nom_ancora)	nom_ancora: text per a l'atribut NAME de l'etiqueta < A NAME=.	
big	Mostra un text en lletres grans.	big()		
blink	Mostra el text parpellejant.	blink()		
bold	Mostra el text en negreta.	bold()		
charAt	Retorna un caràcter del text, el que és en la posició indicada.	charAt(índex)	índex: nombre entre 0 i la longitud-1 del text.	Caràcter del text que és en la posició indicada pel paràmetre.
charCodeAt	Retorna el codi ISO-Latin-1 del caràcter que és en la posició indicada.	charCodeAt(índex)	índex: nombre entre 0 i la longitud-1 del text.	Codi del caràcter que és en la posició indicada pel paràmetre.
concat	Uneix dues cadenes de text.	concat(text2)	text2: cadena de text que s'ha d'unir a la que crida el mètode.	String resultat de la unió de les altres dues.
fixed	Mostra el tipus amb font de lletra teletip.	fixed()		
fontcolor	Mostra el text en el color especificat.	fontcolor(color)	color: color per al text.	
fontsize	Mostra el text en la grandària especificada.	fontsize(size)	size: grandària per al text.	
fromCharCode	Retorna el text creat a partir de caràcters en codi ISO-Latin-1.	fromCharCode(num1, ..., numN)	numN: codis ISO-Latin-1	String

Nom	Descripció	Sintaxi	Paràmetres	Retorn
indexOf	Retorna l'índex en què hi ha per primera vegada la seqüència de caràcters especificada.	indexOf(valor, inici)	valor: caràcter o caràcters que s'han de buscar. inici: índex a partir del qual comença a buscar.	Nombre que indica l'índex. Retorna -1 si no es troba el valor especificat.
italics	Mostra el text en cursiva.	italics()		
lastIndexOf	Retorna l'índex en què hi ha per darrera vegada la seqüència de caràcters especificada.	lastIndexOf(valor, inici)	valor: caràcter o caràcters que s'han de buscar. inici: índex a partir del qual comença a buscar.	Nombre que indica l'índex. Retorna -1 si no es troba el valor especificat.
link	Crea un enllaç HTML.	link(href)	href: string que especifica la destinació de l'enllaç.	
match	Retorna les parts del text que coincideixen amb l'expressió regular indicada.	match(exp)	exp: expressió. Pot incloure els indicadors /f (global) i / i (ignorar majúscules i minúscules).	Array que conté els textos que s'han trobat.
replace	Substitueix una part del text pel text nou indicat.	replace(exp, text2)	exp: expressió regular per a fer la cerca del text que s'ha de substituir. text2: text que substitueix el que s'ha trobat.	String nova.
search	Retorna l'índex del text que s'ha indicat en l'expressió regular.	search(exp)	exp: expressió per a fer la cerca.	
slice	Extreu una porció de l'string.	slice(inici,[final])	inici: índex del primer caràcter que s'ha d'extreure. final: índex del darrer caràcter que s'ha d'extreure. Pot ser negatiu, i llavors indica quants en cal restar des del final. Si no s'hi indica, extreu fins i tot el final.	String que conté els caràcters que hi havia entre inici i final.
small	Mostra el text en font petita.	small()		
split	Crea una array, i separa el text segons el separador indicat.	split([separador], [límit])	separador: caràcter que indica per on s'ha de separar. Si s'omet, l'array només contindrà un element, que és l'string completa. límit: indica el nombre màxim de parts per a posar en l'array.	Array.
strike	Mostra el text ratllat.	strike()		
sub	Mostra el text com a subíndex.	sub()		
substr	Retorna una porció del text.	substr(inici, [longitud])	inici: índex del primer caràcter per extreure. longitud: nombre de caràcters per extreure. Si s'omet, s'extreu fins al final de l'string.	String.
substring	Retorna una porció del text.	substring(inici, final)	inici: índex del primer caràcter que s'ha d'extreure. final: índex+1 del darrer caràcter que s'ha d'extreure.	

Nom	Descripció	Sintaxi	Paràmetres	Retorn
sup	Mostra el text com a superíndex.	sup()		
toLocaleLowerCase	Converteix el text a minúscules, tenint en compte el llenguatge de l'usuari.	toLocaleLowerCase()		String nova en minúscules.
toLocaleUpperCase	Converteix el text a majúscules, tenint en compte el llenguatge de l'usuari.	toLocaleUpperCase()		String nova en majúscules.
toLowerCase	Converteix el text a minúscules.	toLowerCase()		String nova en minúscules.
toUpperCase	Converteix el text a majúscules.	toUpperCase()		String nova en majúscules.
toString	Obté l'string que representa l'objecte.	toString()		String.
valueOf	Obté l'string que representa el valor de l'objecte.	valueOf()		String.

Codis ISO-Latin-1

El conjunt de codis ISO-Latin-1 agafa valors de 0 a 255. Els 128 primers es corresponen amb el codi ASCII.

3. Expressions regulars i ús de galetes

3.1. Les expressions regulars

3.1.1. Introducció a la sintaxi

Les expressions regulars són un mecanisme que permet dur a terme cerques, comparacions i certs reemplaçaments complexos.

Per exemple si s'escriu en la línia d'ordres de Microsoft Windows l'ordre:

```
dir *.exe,
```

S'està utilitzant una expressió regular que defineix totes les cadenes de caràcters que comencin amb qualsevol valor seguit de ".exe", és a dir, tots els arxius executables independentment del seu nom.

L'acció anterior en la qual es compara la cadena de text amb el patró (expressió regular) es denomina *reconeixement de patrons* (pattern matching).

Les expressions regulars ajuden a la cerca, comparació i manipulació de cadenes de text.

En JavaScript, les expressions regulars es basen en les del llenguatge Perl, de manera que són molt semblants a les d'aquest i es representen per l'objecte RegExp (de *regular expresion*).

Es pot crear una expressió regular amb el constructor de l'objecte RegExp o bé utilitzant una sintaxi especialment creada per a això.

En l'exemple següent s'observa això últim:

```
var patro = /Cubo/;
```

L'expressió regular anterior és molt simple: en una comparació amb una cadena retornaria *true* si la cadena amb la qual es comparés fos "Cubo" (totes les expressions regulars s'escriuen entre barres invertides).

De la mateixa manera, és possible crear l'expressió regular anterior utilitzant l'objecte RegExp:

```
var patro = new RegExp("Cubo");
```

Però, en aquest cas, el que se li passa al constructor és una cadena, per tant, en lloc d'usar / es tanca entre cometes dobles.

Les expressions regulars es poden crear utilitzant-ne la sintaxi específica o amb l'objecte RegExp.

Per a complicar una mica més l'exemple anterior, se suposa que es vol comprovar si la cadena és "Cubo" o "Cuba". Per a això s'usen els claudàtors, que indiquen opció, és a dir, en comparar amb /[ao]/ retornaria *cert* en cas que la cadena fos la lletra *a* o la lletra *o*.

```
var patro = /Cub[ao]/;
```

I si es volgués comprovar si la cadena és Cub0, Cub1, Cub2, ..., Cub9? En lloc d'haver de tancar els deu dígits dins dels claudàtors s'utilitzaria el guió, que serveix per a indicar rangs de valors. Per exemple, 0-9 indica tots els nombres de 0 a 9 inclusivament.

```
var patro = /Cub[0-9]/;
```

Si a més es busca que l'últim caràcter sigui un dígit (0-9) o una lletra minúscula (*a-z*), se soluciona fàcilment escrivint dins dels claudàtors un criteri darrere de l'altre.

```
var patro = /Cub[0-9a-z]/;
```

I, què passaria si en lloc de tenir solament un nombre o una lletra minúscula es volgués comprovar que n'hi pot haver més, però sempre minúscules o nombres?

En el cas anterior s'hauria de recórrer als marcadors següents:

- +: indica que el que té a l'esquerra pot estar 1 o més vegades.
- *: indica que pot estar 0 o més vegades (en el cas de +, el nombre o la minúscula hauria d'aparèixer almenys 1 vegada, amb * Cub també s'acceptaria).
- ?: indica opcionalitat, és a dir, el que es té a l'esquerra pot aparèixer o no (pot aparèixer 0 o 1 vegades).

- {}: serveix per a indicar exactament el nombre de vegades que pot aparèixer o un rang de valors. Per exemple, {3} indica que ha d'aparèixer exactament 3 vegades, {3,8} indica que ha d'aparèixer entre 3 i 8 vegades i {3,} indica que ha d'aparèixer almenys 3 vegades.

S'ha d'anar amb compte amb les {}, ja que exigeixen que es repeteixi l'últim, de manera que quan no s'estigui segur del que es farà s'utilitzaran els (). Per a il·lustrar l'anterior, es planteja l'exemple següent d'ús d'expressions regulars:

```
var patro = /Cub[ao]{2}/;
document.write("Cubocuba".search(patro));
document.write("Cuboa".search(patro));
```

La funció *search* de String comprova si la cadena representada pel patró que se li passa com a argument es troba dins de la cadena sobre la qual es crida. Si és així, retorna la posició (per exemple, per a la cadena Cubo, amb el patró /C/ retornaria 0, 1 si el patró és *u*, 2 si és *b*, etc.), i #1 si no es troba.

Un altre ús interessant és l'ús del mètode *replace* de l'objecte String, la sintaxi del qual cadena.replace(patro, substitut) indica que se substitueix la cadena sobre la qual es criden les ocurrències del patró per la cadena especificada en la crida.

Un dels elements més interessants de les expressions regulars és l'especificació de la posició en què s'ha de trobar la cadena. Per a això s'utilitzen els caràcters ^ i $, que indiquen que l'element sobre el qual actua ha d'anar al principi de la cadena o al final d'aquesta respectivament.

En l'exemple següent se'n pot veure l'ús:

```
var patro = /^aa/; //Es busca la cadena aa a l'inici de la cadena
var patro = /uu$/; //Es busca la cadena uu al final de la cadena
```

Altres expressions interessants són les següents:

- \d: un dígit, equival a [0-9].
- \D: qualsevol caràcter que no sigui un dígit.
- \w: qualsevol caràcter alfanumèric, equival a [a-zA-Z0-9_].
- \W: qualsevol caràcter no alfanumèric.
- \s: espai.
- \t: tabulador.

L'enllaç següent apunta a un portal web dedicat a les expressions regulars, en què s'ha de tenir en compte que aquestes són compatibles en la majoria dels llenguatges de programació que les implementen:

```
http://www.regular-expressions.info/
```

3.1.2. L'objecte RegExp

L'objecte RegExp conté el patró d'una expressió. El constructor per a l'objecte RegExp és el següent:

```
new RegExp("patró", "indicador")
```

en què:

- patro és text de l'expressió regular.
- indicador és opcional i pot prendre tres valors:
  - *g*: s'han de tenir en compte totes les vegades que l'expressió apareix en la cadena.
  - *i*: ignora majúscules i minúscules.
  - *gi*: tenen efecte les dues opcions, *g* i *i*.

En el patró de l'expressió regular, es poden usar caràcters especials. Els caràcters especials substitueixen una part del text. A continuació hi ha una llista dels caràcters especials que es poden usar:

Taula 17

Caràcters especials en expressions regulars	
\	Per als caràcters que normalment s'interpreten com a literals, indica que el caràcter que el segueix no s'ha d'interpretar com un literal. Per exemple: /b/ s'interpretaria com a "b", però /\b/ s'interpretaria com a indicador de límit de paraula. Per als caràcters que normalment no s'interpreten com a literals, indica que en aquest cas sí ha d'interpretar-se com un literal i no com un caràcter especial. Per exemple, * és un caràcter especial que s'utilitza com a comodí, /a*/ pot significar cap o diverses *a*. Si l'expressió conté /a*/ s'interpreta com el literal "a*"
^	Indica inici de línia. Per exemple, /^A/ no trobarà l'A en la cadena "HOLA", però sí la trobarà a "Alarma".
$	Indica final de línia. Per exemple, /$A/ no trobarà l'A en la cadena "ADEU", però sí la trobarà a "HOLA".
*	Indica que el caràcter que el precedeix pot aparèixer cap o diverses vegades. Per exemple, /ho*/ es trobarà a "hola", "hoooola", i també a "heura", però no a "camell".
+	Indica que el caràcter que el precedeix pot aparèixer una o diverses vegades. Per exemple, /ho+/ es trobarà a "hola" i "hoooola", però no a "heura" ni a "camell".
?	Indica que el caràcter que el precedeix pot aparèixer cap o una vegada. Per exemple, /ho?/ es trobarà a "hola" i "heura", però no a "hoooola" ni a "camell".
.	Indica un únic caràcter a excepció del salt de línia. Per exemple, /.o/ es trobarà a "hola" però no a "camell".
(x)	Indica que a més de buscar el valor *x*, es repetirà la cerca entre el resultat de la primera cerca. Per exemple, en la frase "hola, t'espero a l'hotel d'holanda", /(holan*)/ trobaria "hola", "holan", i "hola" (l'últim "hola" és part de "holan").
x|y	Indica el valor de *x* o el de *y*. Per exemple, /sol|vent/ trobaria "sol" en la frase "Fa sol a Sevilla".
{n}	Indica quantes vegades exactes ha d'aparèixer el valor que el precedeix (*n* és un enter positiu). Per exemple, /o{4}/ es trobaria a "hoooola" però no a "hola".

Caràcters especials en expressions regulars	
{n,}	Indica quantes vegades com a mínim ha d'aparèixer el caràcter que el precedeix (*n* és un enter positiu). Per exemple, /o{2,}/ es trobaria a "hoooola" i "hoola" però no a "hola".
{n,m}	Indica el nombre mínim i màxim de vegades que pot aparèixer el caràcter que el precedeix (*n* i *m* són enters positius). Per exemple, /o(1,2)/ es trobaria a "hola" i "hoola", però no a "hoooola".
[xyz]	Indica qualsevol dels valors entre claudàtors. Els elements continguts expressen un rang de valors, per exemple [abcd] podria expressar-se també com [a-d].
[^xyz]	Busca qualsevol valor que no aparegui entre claudàtors. Els elements continguts expressen un rang de valors, per exemple [^abcd] podria expressar-se també com a [^a-d].
[\b]	Indica un *backspace*.
\b	Indica un delimitador de paraula, com un espai. Per exemple, /\bn/ es troba a "llúdria" però no en "mico", i /n \b/ en "camaleó" però no a "mico".
\B	Indica que no pot haver-hi delimitador de paraula, com un espai. Per exemple, /\Bn/ es troba a "mico" però no a "llúdria" i a "camaleó".
\cX	Indica un caràcter de control (*X* és el caràcter de control). Per exemple, /cM/ indica Ctrl+M.
\d	Indica que el caràcter és un dígit. També pot expressar-se com a /[0-9]/.Per exemple, /\d/ a "carrer peix, núm. 9" trobaria 9.
\D	Indica que el caràcter no és un dígit. /\D/ també pot expressar-se com a /[^0-9]/.
\f	Indica salt de pàgina (*form-feed*).
\n	Indica salt de línia (*linefeed*).
\r	Indica retorn de carro.
\s	Indica un espai en blanc que pot ser l'espai, el tabulador, el salt de pàgina i el salt de línia. Per tant, és equivalent a posar [\f\n\r\t\v].
\S	Indica un únic caràcter diferent de l'espai, del tabulador, del salt de pàgina i del salt de línia. Per tant, és equivalent a posar [^\f\n\r\t\v].
\t	Indica el tabulador.
\v	Indica un tabulador vertical.
\w	Indica qualsevol caràcter alfanumèric incloent el _ És equivalent a posar [A-Za-z0-9_].
\n	*n* és un valor que fa referència al parèntesi anterior (explica els parèntesis oberts). Per exemple, a "poma, taronja, pera, préssec", l'expressió /poma(,)\staronja \1/ trobaria "poma, taronja".
\ooctal \xhex	Permet incloure codis ASCII en expressions regulars. Per a valors octals i hexadecimals. *o* i *x* prendrien els valors, per exemple, \2Fhex.

Taula 18

Propietats de l'objecte RegExp	
Nom	**Descripció**
$1, ..., $9	Contenen les parts de l'expressió contingudes entre parèntesis. És només de lectura.
$_	Vegeu la propietat *input*.
$*	Vegeu la propietat *multiline*.

Propietats de l'objecte RegExp	
Nom	**Descripció**
$&	Vegeu la propietat *lastMatch*.
$+	Vegeu la propietat *lastParen*.
$`	Vegeu la propietat *leftContext*.
$'	Vegeu la propietat *rightContext*.
global	Indica si s'usa l'indicador "g" en l'expressió regular. Podem tenir els valors *true* (si s'usa l'indicador "g") i *false* en cas contrari. És de solament lectura.
ignoreCase	Indica si s'usa l'indicador "i" en l'expressió regular. Puc tenir els valors *true* (si s'usa l'indicador "i") i *false* en cas contrari. És només de lectura.
input	Representa l'*string* sobre el qual s'aplica l'expressió regular. També es diu $_. És estàtica, per la qual cosa s'usa de la forma: RegExp.input
lastIndex	Especifica l'índex a partir del qual es pot aplicar l'expressió regular.
lastMatch	Representa l'últim ítem oposat. També es diu $&. És estàtica, per la qual cosa s'usa de la forma: RegExp. lastMatch. És només de lectura.
lastParen	Representa l'últim ítem oposat per una expressió de parèntesi. També es diu $+. És estàtica, per la qual cosa s'usa de la forma: RegExp. lastParen. És només de lectura.
leftContext	Representa el *substring* que precedeix a l'últim ítem oposat. També es diu $`. És estàtica, per la qual cosa s'usa de la forma: RegExp. leftContext És només de lectura.
multicine	Indica si s'aplicarà la cerca en diverses línies. Els seus possibles valors són *true* i *false*. També es diu $*. És estàtica, per la qual cosa s'usa de la forma: RegExp. multiline.
prototype	Representa el prototip per a la classe.
rightContext	Representa el *substring* que segueix a l'últim ítem oposat. També es diu $'. És estàtica, per la qual cosa s'usa de la forma: RegExp. rightContext És només de lectura.
source	Representa l'*string* que conté el patró per a l'expressió regular, excloent les \ i els indicadors "i" i "g". És només de lectura.

Taula 19

Mètodes de l'objecte RegExp				
Nom	**Descripció**	**Sintaxi**	**Paràmetres**	**Retorn**
compile	Compila l'expressió regular durant l'execució d'un *script*.	regexp.compile(patro, [indicadors])	patro: text de l'expressió regular. indicadors: "g" i/o "i".	
Exec	Executa la cerca.	regexp.exec(str) regexp(str)	str: *string* sobre el qual s'aplica la cerca.	Un *array* amb els ítems oposats.
test	Executa la cerca.	regexp.test(str)	str: *string* sobre el qual s'aplica la cerca.	*true* si troba algun ítem, *false* en cas contrari.

3.2. Les galetes

Les galetes o *cookies* neixen amb l'objectiu de solucionar una limitació del protocol HTTP 1.0. Aquesta procedeix del fet que l'HTTP és un protocol sense estat. Això fa que no hi hagi manera de mantenir comunicació o informació de l'usuari al llarg de les diferents peticions que es fan al servidor en una mateixa connexió o visita a una espai o pàgina web.

Les galetes aporten un mecanisme que permet emmagatzemar, en l'equip del client, un conjunt petit de dades de tipus text que són establertes pel servidor web. D'aquesta manera, en cada connexió, el client retorna la galeta amb el valor emmagatzemat al servidor que processa el valor i actua d'una manera que fa les accions pertinents.

> Les galetes aporten un mecanisme que permet emmagatzemar un conjunt petit de dades en l'equip client.

L'assignació d'una galeta segueix la sintaxi següent:

```
nom=valor [;expires=dataGMT] [;domain=domini] [;path=ruta] [;secure]
```

en què:

- nom=valor: defineix el nom de la galeta i el valor que emmagatzemarà.

- expires=dataGMT: estableix la data de caducitat de la galeta, aquesta data ha d'establir-se en format GMT, per la qual cosa serà útil el mètode toGMTString() de l'objecte *Date*. Aquest paràmetre és opcional, de manera que, quan una galeta no té establerta una data de caducitat, aquesta es destrueix quan l'usuari tanca el navegador. En aquest cas es diu que la galeta és de sessió. Les galetes que tenen establerta una data de caducitat es coneixen com a galetes persistents.

- domain=domini: estableix el domini que ha assignat la galeta, de manera que aquesta solament es retornarà davant una petició d'aquest, per exemple, domain=www.uoc.edu implica que la galeta solament es retorna al servidor www.uoc.edu quan s'estableix una connexió amb aquest. Si no s'estableix cap valor, és el mateix navegador el que estableix el valor del domini que ha creat la galeta.

- path=ruta: estableix una ruta específica del domini sobre la qual es retornarà la galeta. Si no s'estableix, la ruta per defecte assignada pel navegador és la ruta des d'on s'ha creat la galeta.

- secure: si s'indica aquest valor, la galeta solament es retorna quan s'ha establert una comunicació segura mitjançant HTTPS. Si no s'assigna aquest valor, el navegador retorna la galeta en connexions no segures HTTP.

Per tant, el funcionament bàsic és el següent: un usuari es connecta a un lloc web, el navegador a partir de l'URL d'aquest revisa el seu conjunt de galetes buscant-ne una que coincideixi amb el domini i la ruta. Si hi ha una o més galetes amb aquestes característiques, el navegador envia al servidor la galeta o, si n'hi ha més, les galetes separades pel caràcter *punt i coma*:

```
nom=Victor; email=vriospazos@uoc.edu
```

Perquè les galetes funcionin és necessari tenir-les habilitades en el navegador. Alguns usuaris les consideren un mecanisme d'invasió de la intimitat, ja que es poden establir galetes persistents que portin un seguiment de certes accions en el procés de navegació pels usuaris.

Nota

És possible comprovar si les galetes estan habilitades. Es pot trobar més informació a:
https://developer.mozilla.org/es/docs/Web/API/Navigator.cookieEnabled
i a
http://www.w3schools.com/jsref/prop_nav_cookieenabled.asp

3.2.1. Maneig de galetes

En JavaScript és possible treballar amb galetes a partir de la propietat galeta de l'objecte *Document*. El funcionament és molt simple, solament és necessari assignar a aquesta propietat una cadena que representi la galeta, de manera que el navegador analitzi la cadena com a galeta i l'afegeixi a la seva llista de galetes.

```
document.cookie="nom=Victor; expires=Sun, 14-Dec-2010 08:00:00 GMT; path=/fitxers";
```

En l'assignació anterior es crea una galeta persistent amb data de caducitat el 14/12/10. En aquest cas, s'assigna una ruta que s'utilitzarà conjuntament amb el domini per defecte de la pàgina que ha creat la galeta.

Aquest mecanisme de domini/ruta provoca que una galeta solament pugui ser recuperada pel domini/ruta que s'indica. D'aquesta manera s'impedeix l'accés a la informació emmagatzemada des d'altres dominis.

En JavaScript es treballa amb galetes a partir de la propietat galeta de l'objecte *Document*.

L'anàlisi que fa el navegador sobre la cadena assignada a document.cookie comprova que el nom i el valor no continguin caràcters com espais en blanc, comes, enes palatals, accents i altres caràcters.

Per a solucionar aquest problema s'utilitzen les funcions escape() i unescape(), que fan la conversió de cadenes en cadenes URL que el validador del navegador dóna per bones.

D'aquesta manera s'utilitzarà el mètode escape() per a convertir una cadena en format URL abans d'emmagatzemar-la en la galeta, i quan la galeta es recuperi s'utilitzarà la funció unescape() sobre el valor de la galeta.

3.2.2. Escriptura, lectura i eliminació de galetes

L'escriptura de galetes és molt senzilla, solament és necessari assignar a la propietat galeta una cadena en què s'especifiqui el nom, el valor i els atributs de caducitat, domini, ruta i seguretat que es vulguin aplicar.

En l'exemple següent es mostra una funció que rep com a paràmetres el nom, el valor i el nombre de dies durant els quals la galeta estarà activa com a paràmetre opcional, de manera que, si aquest no es passa, la galeta serà de sessió i per tant s'eliminarà quan l'usuari tanqui el navegador.

```
function assignaCookie(nom,valor,dies){
   if (typeof(dies) == "undefined"){
      //Si no es passa del paràmetre dies, la cookie és de sessió
      document.cookie = nom + "=" + valor;
   } else {
      //Es crea un objecte Date al qual s'assigna la data actual
      //i s'hi afegeixen els dies de caducitat transformats en
      //mil·lisegons
      var caduca = new Date;
      caduca.setTime(dataCaduca.getTime() + dies*24*3600000);
      document.cookie = nom + "=" + valor +";expires=" + caduca.toGMTString();
   }
}
```

La lectura de galetes es fa examinant la cadena emmagatzemada en la propietat galeta de l'objecte *Document*, però s'ha de tenir en compte que aquesta cadena està formada per tants parells nom=valor com galetes amb diferent valor s'han establert en el document actual. Per exemple, si es fa l'assignació:

```
document.cookie= "nom=Victor";
document.cookie="email=vriospazos@uoc.edu"
```

Cada una de les cadenes anteriors s'afegeix a la galeta, de manera que el valor de document.cookie serà el següent:

```
"nombre=Victor; email=vriospazos@uoc.edu"
```

Normalment, solament és necessari recuperar o treballar amb alguna galeta en concret, per exemple es necessita la cadena que indica el correu electrònic, però la galeta conté totes les cadenes emmagatzemades.

Una galeta emmagatzema el conjunt de cadenes que es van assignant a la propietat galeta de l'objecte document.

Això fa necessari implementar un mecanisme que recuperi cada cadena per separat a partir d'una anàlisi de la cadena retornada per document.cookie.

L'exemple següent utilitza un *array* associatiu per a emmagatzemar els noms i els valors de cadascun dels components de la galeta:

```
//SEs crea l'objecte que contindrà l'array associativa cookies[nom] = valor
var cookies = new Object();

//Es defineix la funció que analitza la cadena i crea l'array a partir de la cadena
//document.cookie, i es comproven certes situacions especials
function extreuCookies(){

   //Variables que emmagatzemaran les cadenes nom i valor
   var nom, valor;

   //Variables que controlaran els límits que marquen la posició de les
   //diverses cookies a la cadena
   var inici, mig, final;

   //El bucle següent comprova si hi ha alguna entrada en l'array
   //associativa, de manera que si es així, es crea una instància nova de
   //l'objecte cookies per a eliminar-les
   for (name in cookies){
      cookies = new Object();
      break;
   }
inici = 0;

   //Es fa un bucle que captura a cada pas el nom i valor de
   //cada cookie de la cadena i l'assigna a l'array associativa
   while (inici< document.cookie.length){
      //la variable medio almacena la posición del próximo carácter "="
      medio =  document.cookie.indexOf('=', inicio);

      //lla variable mig emmagatzema la posició del pròxim caràcter ";"
      final = document.cookie.indexOf(';',inici);
```

```
        //El següent if comprova si final adquireix el valor -1 que indica
        //que no s'ha trobat cap caràcter ";", cosa que indica que
        //s'ha arribat a la darrera cookie i, per tant, s'assigna a la
        //variable final la longitud de la cadena
        if (final == -1) {
           final = document.cookie.length;
        }

        //El següent if fa dues comprovacions; en primer lloc, si
        //mig és més gran que final o mig és -1 (que indica que no
        //s'ha trobat cap caràcter "="), la cookie té nom però
        //no valor assignat
        //En l'altre cas el nom de la cookie és entre els
        //caràcters que hi ha entre inici i mig i el valor de la cookie
        //entre els caràcters que hi ha entre mig+1 i final
        if ( (mig > final) || (mig == -1) ) {
           nom = document.cookie.substring(inici, final);
           valor = "";
        } else {
           nom = document.cookie.substring(inici,mig);
           valor= document.cookie.substring(mig+1,final);
        }

        //Una vegada recuperat el nom i el valor, s'assigna a l'array
        //associativa aplicant la funció de conversió unescape()
           cookies[nom] = unescape(valor);

           //En el pas següent del bucle while, la variable inici adquireix
           //el valor final +2, i d'aquesta manera salten el punt i coma i
           //l'espai que separa les diverses cookies en la cadena
           inici = final + 2;
     }
  }
```

L'eliminació d'una galeta es fa assignant una data de caducitat del passat, per exemple es pot utilitzar la data “01-Jan-1970 00:00:01 GMT”.

L'únic problema apareix quan la galeta no té un valor assignat. Aquests casos s’han de tractar d’una manera especial, com es veu en l’exemple següent:

```
function eliminaCookie(nombre){
   //S'actualitza la cookie modificant la data de caducitat i assignant
   //un valor qualsevol, en l'exemple esborrada
   document.cookie = nom + "=esborrada; expires=Thu, 01-Jan-1970 00:00:01 GMT";
   //S'actualitza la cookie sense valor indicant una data de caducitat
   //anterior
   document.cookie = nom + "; expires=Thu, 01-Jan-1970 00:00:01 GMT";
```

```
}
```

Sobre la funció anterior, s'ha de comentar que en cas que la galeta s'hagi creat amb informació sobre el domini i la ruta, és necessari introduir aquesta informació en la cadena que actualitza o esborra la galeta.

Les funcions anteriors proporcionen mecanismes suficients per al control de les galetes, encara que aquestes poden ser modificades depenent de les necessitats del programador.

A continuació es crearan dues galetes, amb el nom "nom" i "email", i amb els valors "Victor" i "vriospazos@uoc.edu" respectivament:

```
assignaCookie("nom","Victor");
assignaCookie("email","vriospazos@uoc.edu");
```

S'obté l'*array* associatiu amb les galetes emmagatzemades cridant la funció extraeCookies(). Es poden utilitzar aquestes:

```
extreuCookies();
var nom = cookies["nom"];
var corr = cookies["email"];
```

S'eliminen les dues galetes utilitzant la funció eliminaCookie():

```
eliminaCookie("nom");
eliminaCookie("email");
```

3.2.3. Usos principals de les galetes

Actualment, s'utilitzen les galetes en els casos següents:

- S'utilitzen galetes per a emmagatzemar l'estat de l'usuari, adaptant la presentació o el contingut de la pàgina basant-se en les preferències de l'usuari.

- Redireccionar l'accés a una pàgina diferent quan l'usuari compleix certes condicions; per exemple, un usuari registrat passarà directament a les pàgines de contingut, mentre que si és la primera vegada que l'usuari hi accedeix se l'adreça a la pàgina de registre.

- Igual que en el cas anterior, a un usuari que accedeix per primera vegada se li pot obrir una finestra d'informació inicial, de manera que aquesta solament s'obri la primera vegada que accedeix a la pàgina.

3.2.4. Limitacions

Cada navegador imposa un conjunt de limitacions sobre la grandària i el nombre de galetes que es poden establir. Encara que la recomanació RFC 2109 estableix unes limitacions mínimes:

- almenys tres-centes galetes
- mínim de 4.096 bytes per galeta (d'acord amb la grandària dels caràcters que componen la galeta no terminal en la descripció de la sintaxi de l'encapçalament Set-Cookie)
- almenys vint galetes per nom de *host* o domini únic

Evidentment es tracta de limitacions aproximades, ja que cada navegador aplica les seves pròpies limitacions.

3.3. Web Storage

Web Storage apareix amb HTML5 i es presenta com a alternativa a les galetes (*cookies*) per a emmagatzemar informació de les pàgines web en el costat del client. A diferència de les galetes que només permeten emmagatzemar fins a 4 kB d'informació, amb Web Storage podem emmagatzemar fins a 5 MB. Aquesta capacitat la defineix el navegador, de manera que depenent del navegador que utilitzem aquesta capacitat pot ser més gran.

A diferència de les galetes, les dades emmagatzemades amb Web Storage no s'envien automàticament al servidor. Això redueix el pes de cada petició al servidor; per aquest mateix motiu, només es pot accedir a la informació emmagatzemada des de la part del client, és a dir, no hi ha un mecanisme per a accedir-hi directament des d'un llenguatge de programació del costat del servidor com podria ser PHP.

Hi ha dos tipus d'emmagatzematge: localStorage i sessionStorage. La diferència entre tots dos és que el primer conserva la informació permanentment fins que es netegi la memòria cau (*cache*) del navegador, mentre que en el segon la informació es manté fins que es tanca la sessió o, el que és el mateix, fins que es tanca la pestanya que conté el nostre web.

Taula 20. Suport de Web Storage en els diferents navegadors d'Internet

IE	Firefox	Chrome	Safari	Opera	Safari iOS	Android
8+	3.5+	5+	4+	10.5+	2+	2+

La manera d'arxivar i accedir a les dades és igual en tots dos casos, així que ens centrarem en localStorage.

Podem saber si el navegador de l'usuari és compatible amb Web Storage utilitzant el codi següent:

```
<script>
   //Podem utilitzar l'objete localStorage o el sessionStorage
   if(localStorage){
      //Existeix localStorage i podem fer emmagatzematge
      alert("El navegador és compatible amb localStorage ");
   } else {
      //No existeix localStorage
      alert("El navegador no és compatible amb localStorage ");
   }
<script>
```

3.3.1. Utilitzar Web Storage

Emmagatzemar un valor

Per a emmagatzemar dades utilitzant Web Storage, hem de tenir en compte un parell de coses: les dades només poden ser cadenes de text, i cada dada que emmagatzemem està relacionada amb la clau que hi hàgim assignat.

Hi ha tres maneres diferents d'emmagatzemar un valor:

- Inserint el parell clau-valor:

```
localStorage.setItem('clau','valor');
```

- Inserint un nou element com si localStorage fos una taula associativa:

```
localStorage['clau'] = 'valor';
```

- Tractant localStorage com un objecte:

```
localStorage.clau = 'valor';
```

Recuperar un valor

De la mateixa manera que teníem tres maneres diferents d'assignar valors, també tenim tres maneres de recuperar un valor:

- Utilitzant el mètode getItem():

```
valor = localStorage.getItem('clau');
```

- Tractant d'accedir al valor utilitzant la clau com si localStorage fos una taula associativa:

```
valor = localStorage['clau'];
```

- Tractant localStorage com un objecte i utilitzant la clau com si en fos una propietat:

```
valor = localStorage.clau';
```

Eliminar valors

Podem eliminar un determinat valor emmagatzemat utilitzant el mètode removeItem de la manera següent:

```
localStorage.removeItem('clau');
```

Si el que volem és eliminar tots els valors emmagatzemats, podem fer-ho amb el mètode clear:

```
localStorage.clear();
```

Altres operacions

Atès que localStorage es comporta com una taula associativa, podem consultar tots els elements que tinguem emmagatzemats de la manera següent:

```
var total = localStorage.length;
```

També podem obtenir el nom d'una clau determinada usant *localStorage.key(i)*, en què *i* és la posició de la clau.

Amb aquest mètode, si ens interessa, podem recórrer tots els elements emmagatzemats a localStorage. Per exemple, amb aquest codi escrivim totes les claus i els valors corresponents:

```
<script>
   for (var i=0; i < localStorage.length; i++) {
      document.write("La clau "+localStorage.key(i)+" emmagatzem el valor "
      +localStorage[localStorage.key(i)]);
   }
</script>
```

Si volem fer proves en local amb Web Storage, és possible que en alguns navegadors, com Internet Explorer, sigui necessari executar el document HTML utilitzant un servidor web local.

Web recomanat

En la revista *Mosaic* podeu veure alguns exemples funcionant: http://mosaic.uoc.edu/2014/02/11/web-storage/

Activitats

1. Descriviu tres exemples d'objectes del món real:

- Per a cadascun dels objectes, definiu la classe a la qual pertanyen.
- Assigneu a cada classe un identificador descriptiu adequat.
- Enumereu diversos atributs i operacions per a cadascuna de les classes.

2. Creeu una classe per a cadascun dels objectes plantejats en l'exercici anterior utilitzant el llenguatge Javascript.

3. Creeu una pàgina web en la qual es creen almenys 2 objectes de les classes anteriors i s'utilitzen els seus mètodes i propietats.

4. Creeu un rellotge que l'usuari del web actualitzi manualment fent un clic en un botó.

5. Utilitzant l'objecte String creeu dues funcions:

- encripta(text): que substitueix cadascun dels caràcters de la cadena per d'altres, de manera que el text resultant quedarà inintel·ligible.
- desencripta(text): que realitza la substitució inversa a la de la funció anterior.

Utilitzeu les dues funcions anteriors en una pàgina web on perquè un text introduït per l'usuari, s'encripti o desencripti.