Objectes definits pel programador

Xavier Garcia Rodríguez

Índex

Introducció **5**

Resultats d'aprenentatge **7**

1 Objectes definits pel programador **9**

1.1 Introducció a la programació orientada a objectes . . . 11
1.2 Creació d'objectes amb el constructor . . . 12
1.3 Declaració literal d'objectes . . . 13
1.3.1 Assignar i accedir a propietats . . . 14
1.3.2 Assignar mètodes . . . 19
1.3.3 Actualitzar propietats . . . 23
1.3.4 Augmentar objectes . . . 28
1.3.5 Eliminar propietats i mètodes . . . 29
1.4 Definir un espai de noms . . . 31
1.5 Prototipus . . . 33
1.5.1 Augmentar objectes predefinits . . . 34
1.6 Classes en JavaScript . . . 35
1.6.1 Mètodes d'accés i actualitzadors (getters i setters) . . . 35
1.6.2 Mètodes estàtics . . . 36
1.7 Herència . . . 38
1.7.1 Implementar l'herència . . . 38
1.7.2 Herència múltiple: mix-ins . . . 42
1.7.3 Patró: factoria . . . 45
1.7.4 Composició i delegació . . . 45

Introducció

Tot i que JavaScript és un llenguatge orientat a objectes, en el qual pràcticament tots els seus elements són objectes, es tracta d'un sistema basat en prototips i no en classes. Aquesta diferència, juntament amb algunes decisions poc afortunades en la implementació del llenguatge, fan que pocs desenvolupadors utilitzin els objectes de JavaScript correctament.

Per aquesta raó, la correcta assimilació d'aquesta unitat és fonamental pel desenvolupament d'aplicacions en l'entorn del client, ja que en tot moment es treballa amb objectes.

En aquesta unitat es tractarà la creació, actualització, augment i modificació dels objectes a més a més dels usos alternatius que se'ls pot donar, com pot ser crear un diccionari de dades o un espai de noms.

Seguidament, es descriurà el sistema de prototipus, les seves característiques, com utilitzar-lo i com augmentar els objectes predefinits del sistema per afegir noves característiques globalment.

A continuació, veureu com implementar diferents patrons de creació d'objectes, juntament amb els seus avantatges i inconvenients. Concretament es tractaran els patrons prototípic, funcional i constructor.

Atesa la importància de l'herència en els llenguatges orientats a objectes, s'explicarà quines alternatives es troben a la vostra disposició per implementar-la a JavaScript, tant en forma de jerarquia d'objectes i constructors, com a través de la composició i la delegació per construir objectes complexos.

Finalment, trobareu un resum de les característiques relacionades amb la creació d'objectes afegides a ES6. En concret el sistema de classes, molt similar al que es pot trobar en altres llenguatges clàssics com Java o C++.

Cal recordar que, com en tots els llenguatges de programació, la millor manera d'assimilar els conceptes d'aquesta unitat és practicant, modificant els exemples que trobareu al llarg de la unitat, escrivint els vostres propis programes, consultant l'extensa documentació en línia que podeu trobar a Internet i preguntant al fòrum de l'assignatura.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

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

- Reconeix les característiques d'orientació a objectes del llenguatge.
- Crea codi per definir l'estructura d'objectes.
- Crea mètodes i propietats.
- Crea codi que faci ús d'objectes definits per l'usuari.
- Depura i documenta el codi.

1. Objectes definits pel programador

El terme *herència clàssica* es refereix a un tipus d'herència basada en classes.

A excepció dels nombres, les cadenes de caràcters, els booleans (`true` i `false`), `null` i `undefined`, a JavaScript tot són objectes. Fins i tot les funcions són tractades com objectes i, per tant, comparteixen les seves característiques. Encara que es pot pensar que els tipus primitius també són objectes (inclouen mètodes i propietats) aquests són immutables i per tant no en són, per exemple: el número 2 sempre serà 2, no pot ser un altre número i, en conseqüència, és immutable.

Ara bé, si tenim una variable a la qual s'ha assignat un valor numèric és possible cridar mètodes a partir d'aquesta variable com per exemple:

```
1 const PI = 3.141592653589793;
2 console.log(num.toFixed(2); // mostra "3.14"
```

Això és possible perquè internament es fa la conversió del valor assignat a un objecte de tipus `Number` i llavors es crida al mètode `toFixed`.

Actualment JavaScript admet la creació d'objectes fent servir l'*herència clàssica*, mitjançant la definició de classes i la seva instanciació mitjançant l'operador `new`, de forma similar a com es fa en altres llenguatges com Java o C++:

```
1  class Persona {
2    constructor(nom, edat) {
3      this.nom = nom;
4      this.edat = edat;
5    }
6  }
7
8  let ricard = new Persona('ricard', 40);
9  let patrici = new Persona('patrici', 35);
10
11 console.log(ricard.nom, ricard.edat);
12 console.log(patrici.nom, patrici.edat);
```

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

Fixeu-vos que a la definició de la classe s'ha posat el primer caràcter en majúscules: `Persona`, en canvi el nom de les variables comença amb minúscules(`ricard`). Això es fa per convenció: els noms de les classes sempre comencen per majúscules, d'aquesta manera és fàcil diferenciar quan es fa referència a una classe o a un altre element.

> Una **classe** és una plantilla a partir de la qual es creen **objectes** i es diu que aquests objectes són **instàncies** de la classe.

Un altre sistema per crear objectes a JavaScript és la declaració literal d'objectes:

```
let ricard = {nom: 'ricard', edat: 40};
let patrici = {nom: 'patrici', edat: 35};

console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat);
```

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

Com es pot apreciar, és molt més concís, ja que no requereix la definició de les classes, però requereix definir totes les propietats de cada objecte.

En versions anteriors a ES6 no existien les classes i per simular l'herència clàssica calia utilitzar funcions constructores. Aquest sistema continua funcionant, però no es recomana utilitzar-lo perquè complica la creació d'objectes complexos i és més limitada, ja que per implementar l'herència cal modificar el prototipus de l'objecte i la manera de simular els mètodes estàtics no és clara. A continuació podeu veure un exemple on s'utilitzen aquestes funcions constructores, típiques del codi anterior a ES6:

```
const Persona = function (nom, edat) {
  this.nom = nom;
  this.edat = edat;
};

let ricard = new Persona('Ricard', 40);
let patrici = new Persona('Patrici', 35);
console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat);
```

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

La instanciació dels objectes `Persona` es fa de la mateixa manera que quan es defineix una classe, però s'utilitza una funció juntament amb l'operador `new`. Cal tenir compte amb això, perquè si ens oblidem d'afegir l'operador, l'aplicació continuarà funcionant però no s'hauria creat l'objecte, ja que s'hauria invocat la funció i assignat el retorn d'aquesta que serà `undefined`:

```
let ricard = new Persona('Ricard', 40);
let patrici = Persona('Patrici', 35);
console.log(ricard.nom, ricard.edat);
console.log(patrici.nom, patrici.edat); // error, patrici és undefined
```

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

Fixeu-vos que `patrici` és `undefined`, ja que en lloc d'assignar-se l'objecte creat mitjançant la funció constructora `persona` s'ha assignat el retorn d'aquesta, que en aquest cas és `undefined`.

Un dels problemes d'utilitzar funcions constructores és que per crear objectes complexos cal recórrer a la modificació del prototipus de l'objecte o afegir tot el codi dintre del cos de la funció constructora (per exemple, múltiples funcions niuades).

Originalment JavaScript només permetia l'*herència prototípica*. Tots els objectes es creen a partir d'un prototipus del qual hereten les seves propietats i mètodes.

En aquests materials ens centrarem en la creació d'objectes mitjançant la definició de classes i la declaració literal d'objectes.

1.1 Introducció a la programació orientada a objectes

Les característiques que ha d'acomplir un llenguatge per considerar-se orientat a objectes són les següents:

- **Abstracció**: s'han de poder crear objectes que ens serviran per modelar la realitat i el problema a resoldre (per exemple en un joc sobre un apocalipsi zombi s'han de poder modelar els personatges, les armes, els zombis, els vehicles que es troben, etc.)
- **Encapsulament**: s'ha de poder encapsular informació dins d'un objecte que no sigui accessible externament. En llenguatges com Java seria equivalent a mètodes i propietats amb accés privat, en canvi en JavaScript es fa a través de les **clausures**.
- **Herència**: una classe ha de poder heretar d'un altre de manera que la nova classe tingui totes les característiques (accessibles) del que hereta a més de les pròpies.
- **Polimorfisme**: una classe que hereti d'un altre ha de poder modificar el comportament de les accions, per exemple sobreescrivint els mètodes.

Una classe conté propietats i mètodes. Les **propietats** són similars a les variables, ja que permeten emmagatzemar valors (nombres, cadenes de text, objectes, *arrays*, etc.), però lligades a l'objecte. Per altra banda, els **mètodes** són funcions lligades a l'objecte i el seu funcionament és idèntic, amb excepció del context `this` que passa a ser el mateix objecte.

Cal destacar que a JavaScript el concepte de modificador d'accés (públic o privat) no forma part del llenguatge. **Totes les propietats i mètodes són considerats públics**, encara que és possible encapsular-los mitjançant *clausures*.

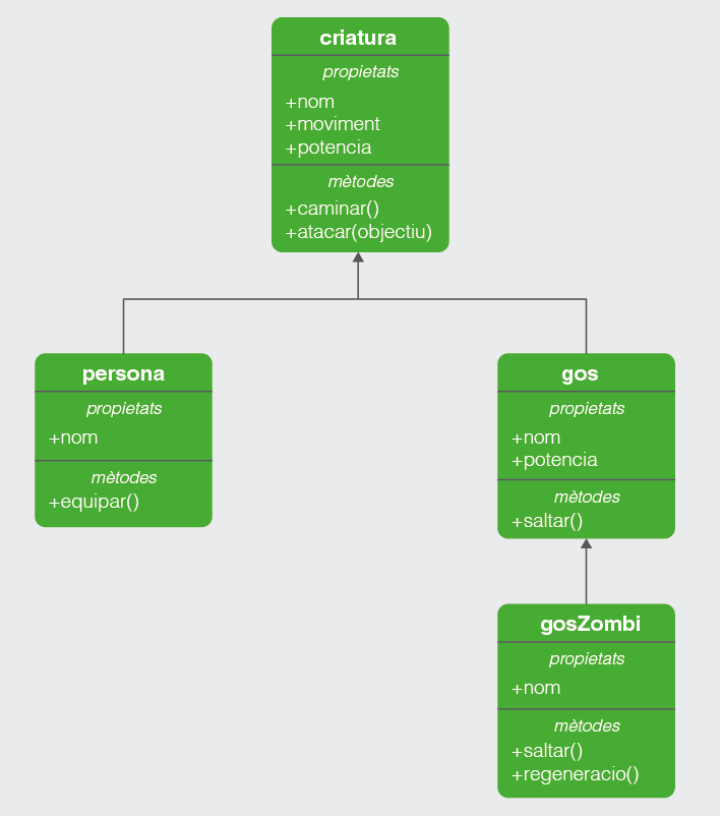
Actualment JavaScript permet utilitzar explicitament totes aquestes característiques a excepció de l'encapsulament, que requereix la utilització de clausures.

Quant a l'herència cal tenir clars dos conceptes basats en l'herència clàssica:

- **Generalització**: consisteix en la creació d'una classe que implementi les propietats i mètodes comuns d'altres classes que heretaran d'aquesta.
- **Especialització**: consisteix en la creació d'una classe amb un comportament especialitzat que afegeix o modifica el comportament de la classe de la qual hereta.

Fixeu-vos en la figura 1.1 per veure un exemple de jerarquía d'herència:

FIGURA 1.1. Generalització i especialització



Quan es tracta d'objectes el seu nom comença amb minúscula.

Com es pot apreciar, `Criatura` és la generalització de `Persona` i `Gos`, ja que conté els mètodes comuns. Per altra banda, `Persona` i `Gos` són especialitzacions de `Criatura`, ja que actualitzen algunes propietats i afegeixen nous mètodes. Al mateix temps, `GosZombi` és una especialització de `Gos` perquè hereta d'aquest i hi afegeix les seves propietats i mètodes.

En els **diagrames de classes**, com el de la figura 1.1, es posa a la capçalera del requadre el nom de la classe, a continuació les propietats i en darrer lloc els mètodes (fàcilment distingibles perquè inclouen parèntesis). Per indicar l'herència es fan servir fletxes amb la punta buida, que assenyalen a la super-classe (o objecte/constructor pare).

Es pot trobar més informació sobre els *diagrames de classes* a l'enllaç següent: ca.wikipedia.org/wiki/Diagrama_de_classes.

1.2 Creació d'objectes amb el constructor

En els llenguatges clàssics es denomina constructor a la funció que es crida automàticament quan es crea un objecte d'una classe concreta. Aquesta funció pot acceptar paràmetres que habitualment s'utilitzen per inicialitzar la classe.

A JavaScript existeix un concepte de constructor que és una mica diferent, ja que no fa referència al constructor que es pot trobar a una classe, sinó que es tracta

d'una funció per inicialitzar un objecte i que s'utilitza juntament amb l'operador `new`:

Tots els objectes hereten d'`Object`, que es pot fer servir com a funció constructora per crear un objecte buit.

```
let persona = new Object();
```

El resultat d'executar aquest codi és equivalent a: `let persona = {}`. En cas de voler crear un objecte nou amb propietats fent servir aquest format es pot fer de dues maneres:

- Afegint un objecte declarat literalment com a paràmetre de `Object`, per exemple: `new Object({nom: "Ricard"})`.
- Augmentar-lo afegint les propietats que siguin necessaries.

De la mateixa manera, si volem crear una instància d'una classe, només cal indicar el nom de la classe, i, si escau, els paràmetres que passaran al constructor:

```
let ricard = new Persona('ricard', 40);
```

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

1.3 Declaració literal d'objectes

La manera més simple de crear un objecte en JavaScript és a través de la seva declaració literal. Aquesta declaració consisteix a fer servir un parell de claus `{}` dins de les quals es defineixen els valors desitjats en forma de parells clau-valor, que corresponen al nom i al valor de la propietat respectivament, separats per comes.

Per exemple, es pot crear un objecte referenciat per la variable `persona` que guardarà les dades d'aquesta persona:

```
let persona = {
  nom: 'Ricard',
  ocupacio: 'Policia',
  edat: 40
};
```

És important tenir en compte que el format és diferent de l'habitual que trobem a JavaScript, ja que en lloc de fer servir l'operador d'assignació `=` es fan servir els dos punts `:`.

Aquesta forma de crear objectes és la base del **format d'intercanvi de dades JSON** (*JavaScript Object Notation* en anglès), un format molt utilitzat tant en JavaScript com en altres llenguatges.

Podeu trobar més informació sobre el format JSON en la secció "Annexos" del web del mòdul.

1.3.1 Assignar i accedir a propietats

Les **propietats** d'un objecte són similars a les variables, però els seus valors són accessibles només a través de l'objecte. En aquest cas per mostrar el valor del nom de `persona` es faria de la manera següent:

```
1 console.log(persona.nom);
```

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

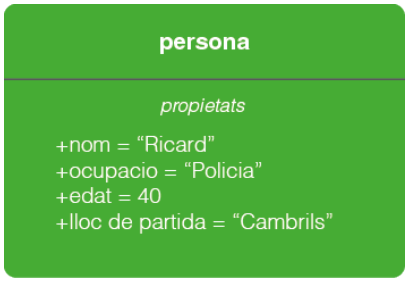
El nom de les propietats es pot afegir amb cometes o sense, excepte si ha d'incloure espais (cosa poc habitual), llavors és obligatori:

```
1 let persona = {
2   'nom': 'Ricard',
3   'ocupacio': 'Policia',
4   'edat': 40,
5   'lloc de partida': 'Cambrils'
6 };
7
8 console.log(persona.nom);
```

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

i el diagrama corresponent a la figura 1.2.

Figura 1.2. Diagrama d'objecte amb quatre propietats



El símbol + indica que es tracta de propietats públiques

En aquest cas s'han fet servir les cometes per totes les propietats encara que només són necessàries en el cas de `lloc de partida`, ja que pel fet de contenir espais es produiria un error.

Cal tenir en compte que es poden fer servir indistintament les cometes simples o les cometes dobles per definir el nom de les propietats, de la mateixa manera que si es tractessin de cadenes de text, com es pot apreciar en el següent exemple:

```
1 let persona = {
2   "nom": "Ricard",
3   "ocupacio": "Policia",
4   "edat": 40,
5   "lloc de partida": "Cambrils"
6 };
```

Com podeu apreciar, s'hi han substituït les cometes simples per cometes dobles, tant al nom de les propietats com al de les cadenes de text. Utilitzar les primeres és un punt més òptim que fer servir les dobles, però allò realment important és mantenir un mateix criteri al llarg del codi.

A més de la notació de punt, és possible accedir a les propietats fent servir la notació de claudàtors, com si es tractés d'un *array* en lloc d'un objecte. Per exemple, podem afegir a l'exemple anterior:

```
console.log(persona.['edat']);
console.log(persona.['ocupacio']);
```

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

Això permet tractar aquests objectes de la mateixa manera que en altres llenguatges es tracten els *arrays* associatius o els diccionaris. A més a més, al contrari que en altres llenguatges, els *arrays* de JavaScript no són tan eficients, raó per la qual es poden utilitzar objectes o *arrays* indistintament segons les vostres necessitats.

Encara que s'ha de tenir en compte que els objectes no ofereixen les mateixes funcionalitats que els *arrays*, ja que l'ordre és irrellevant.

A més a més, actualment JavaScript inclou l'objecte predefinit `Map` que permet crear col·leccions de dades en parells clau-valor i funcionalitats més apropiades per treballar amb dades com la iteració, la comprovació dels valors, l'addició i l'eliminació de les dades.

En el següent exemple, en el cas de les ocupacions només interessa conèixer el nom de l'ocupació i per tant és més pràctic fer servir un *array*, en canvi, en el cas dels supervivents és més útil poder accedir a la informació directament fent servir el continent com a clau:

```
let ocupacions = ['Sense ocupació', 'Policia', 'Militar', 'Metge', 'Tècnic', '
    Mecànic', 'Delinqüent'];

let supervivents = {
    'Barcelona': 23140,
    'Girona': 6789,
    'Lleida': 11298,
    'Tarragona': 19830
};

console.log('Array: Primera ocupació?', ocupacions[0]);
console.log('Objecte: Supervivents a Girona?', supervivents['Girona']);
```

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

Però no només es poden emmagatzemar valors primitius, les propietats dels objectes ens permeten emmagatzemar tot tipus de dades com poden ser *arrays* i altres objectes. Al mateix temps, aquests objectes poden ser emmagatzemats en *arrays*, el que permet crear estructures de dades complexes:

```
let MOTOR_DEL_JOC = {
```

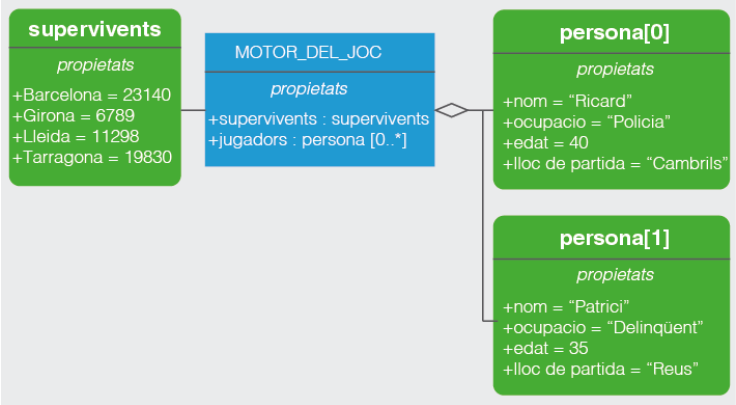
Ús de caràcters no estàndard com a nom de propietats

Quan es tracten els objectes com a diccionaris és normal utilitzar caràcters que habitualment no són recomanables pels noms de les propietats i variables, per exemple: caràcters amb titlla o dièresi.

```
  ocupacions: ['Sense ocupació', 'Policia', 'Militar', 'Metge', 'Tècnic', 'Mecà
      nic', 'Delinqüent'],

  supervivents: {
    'Barcelona': 23140,
    'Girona': 6789,
    'Lleida': 11298,
    'Tarragona': 19830,
  },

  jugadors: [{
    'nom': 'Ricard',
    'ocupacio': 'Policia',
    'edat': 40,
    'lloc de partida': 'Cambril'
  }, {
    'nom': 'Patrici',
    'ocupacio': 'Delinqüent',
    'edat': 35,
    'lloc de partida': 'Reus'
  }]
}

console.log('Tercera ocupació:', MOTOR_DEL_JOC.ocupacions[3]);
console.log('Supervivents a Girona:', MOTOR_DEL_JOC.supervivents['Girona']);
console.log('Nom del primer jugador:', MOTOR_DEL_JOC.jugadors[0]['nom']);
console.log('Ocupació del primer jugador:', MOTOR_DEL_JOC.jugadors[0].ocupacio)
    ;
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/ExKNvro?editors=0012 i el diagrama corresponent a la figura 1.3.

Figura 1.3. Estructura de dades que conté arrays i altres objectes



Als diagrames de classe es fa servir un rombe a l'extrem de la classe que conté la **composició** (els múltiples elements que formen l'altre objecte, per exemple els elements d'un *array*), en canvi quan es tracta d'un sol element es defineix com una associació i es fa servir una línia simple que uneix les dues classes.

Com es pot apreciar, per accedir als elements d'*arrays* i d'altres objectes niuats només cal fer servir la notació de punt o de claudàtors segons correspongui:

```
console.log('Nom del primer jugador:', motorDelJoc.jugadors[0]['nom']);
```

Per enumerar les propietats d'un objecte o els elements d'un diccionari de dades s'utilitza una variant de la sentència `for`, coneguda com `for...in`. Aquesta sentència recorre tots els elements d'un objecte o *array* sense haver d'especificar el valor inicial ni final, ja que fa un recorregut complet, com es pot apreciar en el

següent exemple:

```
let ocupacions = ['Sense ocupació', 'Policia', 'Militar', 'Metge', 'Tècnic', '
    Mecànic', 'Delinqüent'];

let supervivents = {
  'Barcelona': 23140,
  'Girona': 6789,
  'Lleida': 11298,
  'Tarragona': 19830
};

console.log('Llistat de professions');
for (let clau in ocupacions) {
  console.log(ocupacions[clau]);
}

console.log('Llistat de supervivents per províncies')
for (let provincia in supervivents) {
  console.log(provincia, ':', supervivents[provincia]);
}
```

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

> La sentència `for...in` itera sobre totes les propietats de l'objecte, diccionari o *array* **sense oferir cap garantia de respectar l'ordre**.

Aquesta variant de la sentència `for` emmagatzema a la variable la clau de la propietat actual de l'objecte especificat a continuació del `in`, iterant sobre totes les propietats de l'objecte o els elements d'un *array*.

Un altre ús molt habitual d'un objecte declarat literalment és utilitzar-lo com a paràmetre de funcions i mètodes. D'aquesta manera només cal passar l'objecte com a argument i no cal recordar l'ordre dels paràmetres, ja que s'accedeix a aquell que sigui necessari fent servir el nom de la propietat (o clau):

```
function mostrarDades(dadesPersona) {
  console.log(`Nom: ${dadesPersona.nom}`);
  console.log(`Ocupació: ${dadesPersona.ocupacio}`);
  console.log(`Edat: ${dadesPersona.edat}`)
};

let dadesPersona = {
  nom: 'Ricard',
  ocupacio: 'Policia',
  edat: 40
};

mostrarDades(dadesPersona);
```

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

A més a més, és possible desestructurar l'objecte per convertir les propietats en paràmetres:

```
function mostrarDades({nom, ocupacio, edat}) {
  console.log(`Nom: ${nom}`);
```

Desestructuració d'objectes

Per desestructurar un objecte cal escriure els noms de les propietats que volem extreure entre claus. Els valors d'aquestes propietats s'assignen automàticament a variables amb el mateix nom.

```
  console.log(‘Ocupació: ${ocupacio}‘);
  console.log(‘Edat: ${edat}‘)
};

let dadesPersona = {
  nom: ’Ricard’,
  ocupacio: ’Policia’,
  edat: 40
};

mostrarDades(dadesPersona);
```

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

Utilitzant la desestructuració es poden definir també valors per defectes per alguns o per a tots els paràmetres:

```
function mostrarDades({nom = "Lluis", ocupacio = "Comerciant", edat}) {
  console.log(‘Nom: ${nom}‘);
  console.log(‘Ocupació: ${ocupacio}‘);
  console.log(‘Edat: ${edat}‘)
};

let dadesPersona = {
  nom: ’Ricard’,
  edat: 40
};

mostrarDades(dadesPersona);
```

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

Com es pot apreciar, la funció `mostrarDades` utilitza el `nom` i l´`edat` que hem passat com argument i l´`ocupacio` per defecte.

És recomanable aplicar aquesta tècnica quan hi ha molts paràmetres optatius, ja que s'evita haver de passar valors nuls com a paràmetres a la funció. L'inconvenient és que, si no es documenta la funció o mètode que rep l'objecte, s'ha d'analitzar el seu codi per saber quines són les propietats que ha de contenir l'objecte.

Quan es treballa ma classes es possible de definir valors per defecte assignant-los directamente a la classe:

```
class Persona {

  constructor (nom) {
    this.nom = nom;
  }

  edat = 0
  ocupacio = ’desconeguda’
}

let persona = new Persona(’Ricard’);

console.log(persona.nom);
console.log(persona[’edat’]);
console.log(persona[’ocupacio’]);

console.log("=".repeat(15));
```

```
persona.edat = 40;
persona.ocupacio = 'Policia';

console.log(persona.nom);
console.log(persona['edat']);
console.log(persona['ocupacio']);
```

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

Fixeu-vos que el cas del `nom` hem optat per assignar-lo al constructor. Tot i que no ho hem definit com a propietat, un cop es crea una instància de l'objecte es crida automàticament al constructor i s'assigna el valor passat com a paràmetre a la propietat `nom`.

Per altra banda, hem assignat a `edat` i `ocupacio` uns valors per defectes, per tant es poden consultar a partir de la instància de `Persona` i el seu valor serà el definit a la classe.

> **Definició de propietats dins de classes**
>
> Tot i que és possible definir propietats fora del constructor, no s'acostuma a fer així. Es recomana definir totes les propietats amb els seus valors per defecte dins del constructor de la classe.

Com es pot apreciar a l'hora de consultar-los i modificar-los ho podem fer de la mateixa manera que es fa amb la declaració literal.

1.3.2 Assignar mètodes

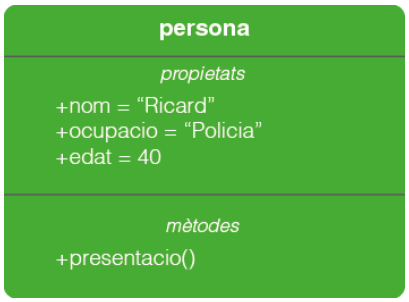
Mètode és el nom que rep una funció quan forma part d'un objecte o classe.

Atès que, en JavaScript, les funcions poden ser emmagatzemades com a variables, el fet d'assignar una funció a una propietat converteix aquesta en un mètode. Aquests funcionen exactament igual que una funció, però **el seu context d'execució** (`this`) **és l'objecte**.

```
let persona = {
  nom: 'Ricard',
  ocupacio: 'Policia',
  edat: 40,
  presentacio: function () {
    console.log(`Hola, em dic ${this.nom}, tinc ${this.edat} anys i sóc ${this.
        ocupacio}`);
  }
};

persona.presentacio();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/NWNbaxr?editors=0012 i el diagrama que el representa a la figura 1.4.

Figura 1.4. Objecte amb mètodes



Els mètodes s'afegeixen a la capsa inferior del requadre de la classe o objecte.

Com es pot apreciar, per invocar un mètode cal escriure primer el nom de la variable que referencia a l'objecte i a continuació fent servir la notació de punt (o de claudàtors) el nom del mètode a invocar seguit dels parèntesis.

Totes les classes inclouen un mètode per defecte anomenat `constructor`, que és cridat automàticament quan es crea una nova instància de la classe:

```
1  class Persona {
2
3    constructor(nom, edat) {
4      this.nom = nom;
5      this.edat = edat;
6    }
7  }
```

Fixeu-vos que per definir mètodes dins d'una classe la sintaxi és diferent: `nomFuncio(arguments) {/* Cos de la funció */}`. Com es pot apreciar, no cal utilitzar `function` ni assignar la funció a cap variable ni propietat.

Per definir altres mètodes ho fem de la mateixa manera:

```
1  class Persona {
2    constructor(nom, edat, ocupacio) {
3      this.nom = nom;
4      this.edat = edat;
5      this.ocupacio = ocupacio;
6    }
7
8    saludar() {
9       console.log('Hola, em dic ${this.nom}, tinc ${this.edat} anys i sóc ${this
            .ocupacio}');
10   }
11 }
12
13 let ricard = new Persona('Ricard', 40, 'Policia');
14 ricard.saludar();
```

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

Cal recordar que les funcions a JavaScript es poden declarar amb nom o com a funcions anònimes. En qualsevol dels dos casos es poden afegir com a mètodes d'un objecte, només cal assignar com a valor de la propietat el nom de la funció o el nom de la variable que fa referència a la funció sense incloure els parèntesis.

Fixeu-vos que, tot i que no és idèntic, el format en què es declaren els mètodes

és molt similar a com es declaren funcions anònimes que es guarden en variables: `let nomVariable = function(arguments) {/* Cos de la funció */}`.

> Si a una variable o propietat s'assigna una funció afegint parèntesis aquesta **és invocada i el resultat es guarda a la funció**. En canvi, si **no** es posen els parèntesis el que es guarda és la referència a la funció, el que permet invocar-la.

Veieu en el següent exemple les dues formes en què es pot declarar una funció i com s'assignen a un objecte:

```
function funcioPresentacio() {
  console.log(‘Hola, em dic ${this.nom}, tinc ${this.edat} anys i sóc ${this.
      ocupacio}‘);
}

const funcioComiat = function() {
  console.log(‘${this.nom} abandona la sala‘);
};

let persona = {
  nom: ’Ricard’,
  ocupacio: ’Policia’,
  edat: 40,
  presentacio: funcioPresentacio,
  comiat: funcioComiat
};

// El context d’aquestes invocacions és l’objecte persona
persona.presentacio();
persona.comiat();

// El context d’aquestes invocacions es l’espai global (window)
funcioPresentacio();
funcioComiat();
```

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

Com es pot apreciar, l'assignació com a mètode és igual en tots dos casos i, encara que les funcions s'han declarat fora de l'objecte, el context del mètode és correcte. En canvi, si s'invoquen les funcions de forma independent, el context d'aquestes és l'espai global i per tant no es troben definides les propietats `nom`, `ocupacio`, ni `edat` i mostrarà `undefined` en el seu lloc.

De la mateixa manera que les funcions, els mètodes també poden acceptar paràmetres que són aplicats exactament igual. És a dir, es crea un objecte `arguments` que pot ser manipulat, poden passar-se múltiples arguments i en cas de no passar-ne suficients, el valor d'aquests serà `undefined` però no es produirà cap error:

```
let persona = {
  nom: ’Ricard’,
  parlar: function (missatge, buit) {
    console.log(‘${this.nom} diu: ${missatge}‘);
    console.log(‘Contingut de buit: ${buit}‘);
    console.log(’Contingut d\’arguments: ’, arguments);
  }
};
```

```
9
10 persona.parlar('Bon dia!');
```

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

Conflicte de contextos

Possiblement el punt més conflictiu en treballar amb objectes és el canvi de context, ja que pot resultar molt confós saber a quin context fa referència `this`. Especialment en el moment que es comença a treballar de forma asíncrona, per exemple amb temporitzadors:

```
1  let persona = {
2    nom: 'Ricard',
3    sortir: function(temps) {
4      setTimeout(function() {
5        console.log(`${this.nom} surt de la sala ${temps}s després`);
6      }, temps * 1000);
7    }
8  };
9
10 persona.sortir(3);
```

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

Com podeu veure el resultat és confós, per una banda `this.nom` ha estat avaluat com `undefined`, en canvi el valor del paràmetre `temps` és correcte.

En el primer cas el valor és `undefined` perquè quan la funció és invocada per `setTimeout` es fa amb el seu context (l'espai global, `window`) i, per tant, `this.nom` no està definit.

Les **clausures** es tracten a la unitat "Estructures definides pel programador".

El segon cas (`temps`) és un paràmetre i, per tant, es tracta com una variable. Dins del mètode es crea una **clausura**, de manera que la funció que es passa com a argument a `setTimeout` té accés a les variables declarades dins del mètode `sortir` i per aquesta raó el valor és correcte.

Hi ha tres maneres de solucionar aquest conflicte, una consisteix a aprofitar el funcionament de les clausures:

- Creant una variable que guardi el context.
- Utilitzar el mètode `bind` (propi de totes les funcions) que estableix el context en què s'executarà la funció
- Utilitzar una funció de fletxa en lloc d'una funció anònima.

Vegeu un exemple en el qual s'apliquen les tècniques de la clausura i la utilització del mètode `bind`:

Quan es guarda el valor del context `this` en una variable s'acostuma a anomenar-la `that` o `self`.

```
1 let persona = {
```

```
  nom: 'Ricard',
  entrar: function(temps) {
    setTimeout(function() {
      console.log(`${this.nom} entra a la sala al cap de ${temps}s`);
    }.bind(this), temps * 1000);
  },
  sortir: function(temps) {
    let that = this;

    setTimeout(function() {
      console.log(`${that.nom} surt de la sala ${temps}s després`);
    }, temps * 1000);
  }
};

persona.entrar(1);
persona.sortir(3);
```

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

Com es pot apreciar, en aquest cas concret se soluciona el conflicte aprofitant la clausura i fent servir el mètode `bind`. Fer servir una tècnica o altra dependrà de les circumstàncies. Per exemple, si la funció que s'ha de cridar no es declara dins de la clausura, no serà possible passar-li el context fent servir una variable ni tampoc si es tracta d'un mètode d'un altre objecte.

bind, call i apply

Els mètodes `call` i `apply` es poden utilitzar quan la invocació es realitza immediatament (síncrona), en canvi `bind` s'utilitza quan l'execució es produirà en el futur (asíncrona).

Cal destacar que actualment és possible evitar el conflicte de canvis de context fent servir funcions de fletxa. Si en lloc de fer servir una funció anònima fem servir una funció de fletxa, el context d'execució de la funció continua sent l'objecte:

```
let persona = {
  nom: 'Ricard',
  sortir: function(temps) {
    setTimeout(() => {console.log(`${this.nom} surt de la sala ${temps}s despré
        s`)}, temps * 1000);
  }
};

persona.sortir(3);
```

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

Atés que les funciones de fletxa conserven el context original, el codi és més clar. A més a més, un dels casos d'ús més importants per les funcions de fletxes és fer-les servir com a paràmetres de funcions.

Els mètodes `call` i `apply` es tracten a la unitat "Estructures definides pel programador".

Les funcions de fletxa es tracten a la unitat "Estructures definides pel programador".

1.3.3 Actualitzar propietats

A JavaScript no existeix el concepte d'àmbit privat o públic d'altres llenguatges, així doncs, totes les propietats dels objectes són accessibles tant per llegir-les com per canviar-les (o actualitzar-les).

Canviar el valor d'una propietat és tan simple com assignar-li un nou valor tal com

es pot comprovar en l'exemple següent:

```
let persona = {
  nom: 'Ricard',
  edat: 40,
  saludar: function () {
    console.log(`Hola, em dic ${this.nom} i tinc ${this.edat} anys`);
  }
};

console.log('Salutació original');
persona.saludar();

persona.nom = 'Pere';
persona.edat++;

console.log('Salutació després de modificar l\'objecte');
persona.saludar();
```

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

Primer se li ha modificat el nom, assignant-n'hi un de nou, i seguidament s'ha incrementat l'edat fent servir l'operador ++. Com es pot apreciar, modificar el valor d'una propietat és igual que canviar el valor d'una variable.

Modificar una funció és igual de fàcil, fixeu-vos en l'exemple següent com s'actualitza el mètode `atacar` per modificar el comportament de l'objecte:

```
let persona = {
  nom: 'Ricard',
  atacar: function(objectiu) {
    console.log(`— ${this.nom} dona un cop de puny a ${objectiu}`);
  }
};

console.log('Personatge ataca desarmat:');
persona.atacar('Zombi');

console.log('Es troba una pistola i l\'equipa:');
persona.atacar = function(objectiu) {
  console.log(`— ${this.nom} dispara dos trets a ${objectiu}`);
};

console.log('Utilitza la nova arma:');
persona.atacar('Zombi');
```

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

Inicialment l'objecte `persona` quan invoca `atacar` fa servir la funció predefinida (per donar cops de puny). Però, una vegada actualitzem el mètode amb la nova funció, dispara trets a l<nocode>'</nocode>`objectiu`.

Una manera més neta de fer aquests canvis interns és afegir un mètode que s'encarregui de fer la substitució com a l'exemple següent, on s'ha afegit el mètode `equiparArma`, de manera que fer servir l'objecte és més clar:

```
let persona = {
  nom: 'Ricard',
  atacar: function(objectiu) {
    console.log(`— ${this.nom} dóna un cop de puny a ${objectiu}`);
```

```
  },
  equiparArma: function(arma) {
    this.atacar = arma;
  }
};

let pistola = function(objectiu) {
  console.log(`— ${this.nom} dispara dos trets a ${objectiu}`);
}

console.log('Personatge ataca desarmat:');
persona.atacar('Zombi');

console.log('Es troba una pistola i l\'equipa:');
persona.equiparArma(pistola);

console.log('Utilitza la nova arma:');
persona.atacar('Zombi');
```

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

Aquesta solució també és aplicable quan es treballa amb classes, ja que un cop instanciat un objecte les seves propietats i mètodes poden ser reassignats:

```
class Persona {

  constructor(nom) {
    this.nom = nom;
  }

  atacar(objectiu) {
    console.log(`— ${this.nom} dona un cop de puny a ${objectiu}`);
  }

  equiparArma(arma) {
    this.atacar = arma;
  }
}

let persona = new Persona('Ricard');
let pistola = function(objectiu) {
  console.log(`— ${this.nom} dispara dos trets a ${objectiu}`);
}

console.log('Personatge ataca desarmat:');
persona.atacar('Zombi');

console.log('Es troba una pistola i l\'equipa:');
persona.equiparArma(pistola);

console.log('Utilitza la nova arma:');
persona.atacar('Zombi');
```

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

En altres llenguatges, aplicar un comportament similar no és trivial, ja que els mètodes no es poden actualitzar directament sobre els objectes i s'ha de recórrer a l'herència o la *delegació*.

La delegació és una tècnica que es basa a encarregar les operacions d'un objecte a un altre en lloc de realitzar-les ell mateix.

L'actualització de mètodes és una clara demostració de com s'aplica el **polimorfisme** a JavaScript.

Per altra banda, com que JavaScript és un llenguatge dèbilment *tipat*, s'ha de tenir en compte que en actualitzar una propietat és possible assignar un tipus de dada incorrecte. Per exemple, si proveu el següent codi, no es produirà cap error, però el resultat no és el desitjat:

Un llenguatge és dèbilment tipat (*weak typing*) si les variables poden tenir valors de tipus diferents al llarg de l'execució del programa . En un llenguatge fortament tipat (*strong typing*), les variables sempre tenen valors del mateix tipus. En aquests llenguatges sol haver també mecanismes de conversió de tipus. A vegades als llenguatges fortament tipats se'ls anomena, simplement, tipats.

```
let persona = {
  nom: 'Ricard',
  edat: 40,
  saludar: function () {
    console.log(`Hola, em dic ${this.nom} i tinc ${this.edat} anys`);
  }
};

console.log('Salutació original');
persona.saludar();

persona.edat = 'quaranta';
persona.edat++;

console.log('Salutació després de modificar l\'objecte');
persona.saludar();
```

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

Com que s'ha canviat un nombre per una cadena de text, l'operació d'increment ja no és possible i el resultat és `NaN` (*Not a number*).

Però no només afecta les propietats. Atès que els mètodes no són més que propietats que referencien una funció, és possible actualitzar una propietat assignant com a valor una funció i a la inversa, canviar un mètode assignant-li qualsevol altra cosa:

```
let persona = {
  nom: 'Ricard',
  edat: 40,
  saludar: function () {
    console.log(`Hola, em dic ${this.nom} i tinc ${this.edat} anys`);
  }
}

console.log('Salutació original');
persona.saludar();

persona.edat = function() {
  console.log('Aquesta funció sobreescriu el valor de l\'edat');
};

persona.saludar = 42;

console.log('Contingut de la propietat edat: ', persona.edat)

console.log('Salutació després de modificar l\'objecte');
persona.saludar();
```

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

En el cas de la propietat `edat`, en lloc de mostrar el seu valor, ens mostra el codi de la funció i en el cas del mètode `saludar` es produeix un error (es pot veure a la consola de les eines de desenvolupador), ja que s'intenta invocar a `persona.saludar` i com que es tracta d'un nombre això no és possible.

A ES6 s'han afegit mètodes d'accés i actualització juntament amb la definició de classe.

Una possible solució és fer servir **mètodes d'accés** (*getters* o *accessors* en anglès) i **mètodes d'actualització** (*setters* o *mutators* en anglès). Consisteix a agregar dos mètodes per cada propietat: un per accedir i un altre per actualitzar-la, de manera que es pot fer el control del tipus dins del mètode.

Aquesta solució pot facilitar que els tipus de cada propietat i mètode siguin correctes, però a JavaScript presenta inconvenients que la fan inviable en molts casos:

- Si no es fan servir clausures tant els mètodes com les propietats continuen sent accessibles i actualitzables directament, per tant, **no es garanteix que es conservi la integritat de l'objecte**.
- **El codi es complica**, ja que per cada propietat s'han d'afegir 2 mètodes.

Així doncs, s'ha de valorar cas per cas si és raonable afegir aquesta complexitat extra. Per aquesta raó JavaScript requereix una major disciplina per part dels desenvolupadors perquè mentre en altres llenguatges aquests tipus d'error són detectats pel compilador, a JavaScript moltes vegades produeixen errors silenciosos difícilment detectables.

Un altre punt important a tenir en compte a l'hora de treballar amb objectes és que, igual que els *arrays*, aquests es passen a les funcions/mètodes per referència. És a dir, quan s'invoca una funció a la qual es passa un objecte com argument, qualsevol canvi que es produeixi afectarà l'objecte original, ja que es tracta del mateix:

```
let pistola = {
  municio: 19
};

let recarregar = function (arma) {
  arma.municio = 42;
};

console.log('Munició actual de l\'arma: ', pistola.municio);
recarregar (pistola);
console.log('Munició actual de la pistola: ', pistola.municio)
```

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

Com es pot apreciar, la propietat `municio` de l'objecte s'ha actualitzat tot i que l'operació s'ha realitzat en una funció aliena a l'objecte.

1.3.4 Augmentar objectes

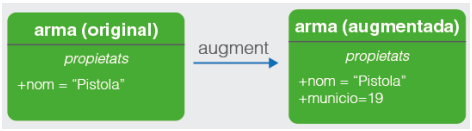
Una altra peculiaritat de JavaScript, que no es troba en gaires llenguatges, és que els objectes poden ser augmentats. És a dir, **es poden afegir noves propietats i mètodes a qualsevol objecte ja existent**.

La manera d'augmentar-los és molt simple, només cal establir el nom de la propietat i assignar-li el valor desitjat. A partir d'aquest moment la nova propietat formarà part de l'objecte com es pot veure a la figura 1.5 i a l'exemple següent:

```
let arma = {
  nom: 'Pistola'
};

arma.municio = 19;

console.log(`Munició de ${arma.nom}: ${arma.municio}`);
```

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

FIGURA 1.5. Objecte augmentat amb una propietat

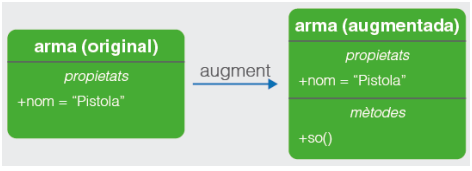


Com es pot apreciar, la declaració de l'objecte `arma` no inclou cap propietat `municio`, però una vegada es fa l'assignació aquesta s'afegeix i es pot tractar com qualsevol altra propietat. En cas de tractar-se d'una funció, l'augment es realitza exactament igual com es pot comprovar en la figura 1.6 i en l'exemple següent:

```
let arma = {
  nom: 'Pistola'
};

arma.so = function() {
  console.log('Bang!');
};

arma.so();
```

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

FIGURA 1.6. Objecte augmentat amb un mètode



Fixeu-vos que aquesta flexibilitat permet afegir noves propietats i mètodes directament o copiant-les d'altres objectes o *arrays* (iterant sobre ells fent servir la sentència `for...in`).

La possibilitat d'augmentar els objectes fa factible utilitzar-los com a diccionaris o estructures de dades.

Cal remarcar que fer ús d'aquesta característica pot portar fàcilment a errors de difícil detecció, ja que un error tipogràfic en una propietat pot causar que en lloc d'actualitzar un valor s'agregui una nova propietat amb el nom mal escrit i malauradament no es produirà cap error que us permeti detectar-lo:

```
let arma = {
  municio: 19,
  disparar: function() {
    if (this.municio > 0) {
      this.mumicio = this.municio – 1;
      console.log(‘Bang! resten ${this.municio} bales‘);
    } else {
      console.log(’Click! munició esgotada’);
    }

  }
};

arma.disparar();
arma.disparar();
arma.disparar();

console.log(arma);
```

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

Com es pot apreciar, no es produeix cap error, però el resultat és incorrecte. Dins del mètode `disparar` s'ha comès un error tipogràfic i en lloc de `municio` s'ha escrit `mumicio`, cosa que provoca un augment de l'objecte afegint una propietat amb aquest mateix nom i valor 18 (obtingut a partir de `this.municio - 1`) en lloc de reduir la propietat `this.municio`.

S'ha de tenir en compte que l'augmentació es produeix sobre els objectes, independentment de com s'hagin creat, i no té cap efecte sobre les classes.

1.3.5 Eliminar propietats i mètodes

De la mateixa manera que es pot augmentar un objecte, JavaScript ens ofereix l'operador `delete` per eliminar mètodes o propietats:

```
let arma = {
  nom: ’pistola’,
  municio: 19
};

console.log(‘Munició de ${arma.nom}: ${arma.municio}‘);
delete arma.municio;
console.log(‘Munició de ${arma.nom}: ${arma.municio}‘);
```

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

m06/pen/JjXbMJr?editors=0012.

En el mateix moment en què es crida l'operador `delete` seguit de la propietat a eliminar, aquesta deixa d'estar vinculada a l'objecte. En cas de voler eliminar propietats niuades només s'ha d'aplicar la notació de punt o de claudàtors per especificar-la, de la mateixa manera que per accedir, augmentar o actualitzar:

```
let motxilla = {
  armes: [{
    nom: 'pistola',
    atacs: 1,
    abast: 60
  }, {
    nom: 'ganivet',
    atacs: 2
  }]
};

function mostrarInventari(inventari) {
  for (let i = 0; i < inventari.armes.length; i++) {
    for (let propietat in inventari.armes[i]) {
      console.log(`${propietat}: ${inventari.armes[i][propietat]}`);
    }
    console.log('——————————');
  }
};

console.log('Mostrant inventari:');
mostrarInventari(motxilla);

delete motxilla.armes[0]['abast'];

console.log('Mostrant inventari després d\'eliminar l\'abast:')
mostrarInventari(motxilla);
```

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

Fixeu-vos que per eliminar la propietat s'ha fet servir la notació de claudàtors per accedir al primer element de l'*array* i a continuació el nom de la propietat. En aquest cas s'ha optat per fer servir la notació de claudàtors en tots dos casos, però fent servir la notació de punt el resultat hauria estat el mateix: `delete motxilla.armes[0].abast`.

En el cas d'intentar accedir a una propietat esborrada el valor retornat serà `undefined`. En canvi, en invocar un mètode esborrat es produirà un error i s'aturarà l'execució, com es pot comprovar en l'exemple següent:

```
let arma = {
  nom: 'Pistola',
  so: function() {
    console.log('Bang!');
  }
};

delete arma.nom;
delete arma.so;

console.log('Nom de l\'arma: ', arma.nom);
arma.so(); // Produeix un error i s'atura l'execució
console.log('Aquesta línia mai s\'executa');
```

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

Com es pot apreciar l'última línia no s'executa mai perquè en invocar a `arma.so` es dispara una excepció de tipus `TypeError` i s'atura l'execució del codi.

Generalment l'operador `delete` s'utilitza amb diccionaris i estructures de dades per poder eliminar els elements, ja que la necessitat d'eliminar una propietat o mètode és molt menys habitual.

1.4 Definir un espai de noms

Un dels problemes de treballar amb JavaScript és la facilitat amb què es pot **contaminar l'espai global del navegador**, ja que si dins d'una funció no es declara una variable aquesta es defineix a l'espai global i els resultats poden ser imprevisibles.

L'espai global al navegador es correspon amb l'objecte `window` i amb `global` a **Node.js**.

Per altra banda, s'ha de tenir en compte que en una mateixa pàgina és habitual carregar múltiples fonts de codi JavaScript que no són controlades per nosaltres com poden ser llibreries (jQuery, Google Maps, Google Analytics, etc.) o codi afegit pels gestors de continguts.

En el cas de fer servir variables globals, és molt fàcil que alguna d'aquestes aplicacions sobreescrigui alguns dels vostres valors (o al contrari), fet que genera uns errors molt difícils de depurar, ja que tant el codi de tercers com el propi funcionarà correctament de forma individual.

També és més difícil depurar i gestionar els objectes de l'aplicació si aquests es troben barrejats amb els objectes del navegador i el codi de tercers, fixeu-vos en el contingut de l'objecte `window` amb les eines de desenvolupador del navegador:

```
1 console.log(window);
```

Com es pot apreciar, inclou tots els objectes predefinits, variables i funcions de JavaScript, a més de les que afegeixen els navegadors i això sense comptar amb el vostre codi ni carregar llibreries externes.

Hi ha dues solucions a aquest problema:

- Fer servir un **espai de noms** (*namespace* en anglès), ficant tots els components de l'aplicació es troben al seu interior. D'aquesta manera és més fàcil inspeccionar els objectes que formen part de l'aplicació al mateix temps que s'eviten possibles conflictes amb altres aplicacions.
- Fer servir **mòduls** per encapsular l'aplicació. D'aquesta manera tota l'aplicació es troba dintre de l'àmbit del mòdul i no a l'espai global.

Podeu trobar informació sobre els mòduls al següent enllaç: mzl.la/3ibdoOl i a la unitat "Objectes predefinits del llenguatge".

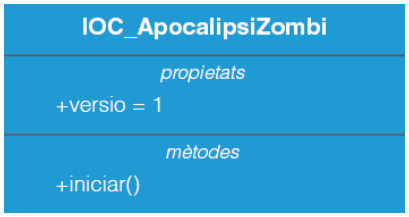
Actualment, el mètode preferit per aïllar l'aplicació és la utilització de **mòduls**.

Crear un espai de noms consisteix a crear un únic objecte que es trobarà a l'espai global del navegador, dins del qual s'afegeix la resta de l'aplicació:

```
1  // Aquest és l'espai de noms per l'aplicació
2  let IOC_ApocalipsiZombi = {
3    // Codi de l'aplicació
4    versio: 1,
5    iniciar: function() { /*Inicialització de l'aplicació */}
6  };
7
8  // Inici de l'aplicació
9  IOC_ApocalipsiZombi.iniciar();
10
11 // Contingut de l'aplicació
12 console.log(window.IOC_ApocalipsiZombi);
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/mdPOpLa?editors=0012 i el diagrama corresponent a la figura 1.7.

Figura 1.7. Espai de noms que agrupa diferents components de l'aplicació



S'ha de tenir en compte que tant la propietat com el mètode s'han afegit amb finalitat demostrativa i no són obligatoris. El contingut serà el que necessiteu per la vostra aplicació.

Si us hi fixeu, tot el contingut de la vostra aplicació es troba ara dins de `window.IOC_ApocalipsiZombi`, de manera que s'eviten possibles conflictes i és més fàcil comprovar el seu contingut.

Al contrari del que passa en altres llenguatges, a JavaScript no existeix cap convenció per establir el nom de l'objecte que es fa servir com a espai de noms, per aquesta raó se'n pot fer servir qualsevol que considereu prou únic per evitar conflictes.

Podeu trobar més informació sobre el *patró mòdul* en la secció "Annexos" del web del mòdul.

El punt feble d'aquesta tècnica és que no s'encapsula la informació i, conseqüentment, es pot accedir a qualsevol de les propietats o mètodes de l'espai de noms directament. Per solucionar aquest problema es pot aplicar el **patró mòdul**, una tècnica basada en l'ús de clausures i funcions autoexecutables.

En conclusió, es recomana fer servir sempre un mòdul o un espai de noms per encapsular la vostra aplicació, encara que això no s'aplica al codi d'exemples o demostracions perquè causaria una complicació innecessària del codi.

1.5 Prototipus

JavaScript és un llenguatge basat en prototipus, és a dir, els objectes poden construir-se a partir d'uns altres objectes (el seu prototipus) i seguidament augmentar-lo amb nous mètodes i propietats.

> A causa de deficiències del propi llenguatge el funcionament d'aquest mecanisme porta fàcilment a confusió. Per aquest motiu a partir de la versio ECMAScript 2015 **es recomana treballar sempre amb classes**, ja que la seva finalitat és molt semblant.

Un prototipus és un objecte del qual altres objectes hereten propietats i mètodes. Aquest objecte només es troba definit a les funcions, com a propietat `prototype` i és possible actualitzar-lo o augmentar-lo.

Quan s'utilitza una funció com a **constructor**, el prototipus de l'objecte creat referencia al de la funció constructora, de manera que aquest hereta totes les propietats i mètodes establerts al prototipus. Conseqüentment, només els objectes generats a partir d'aquest patró tenen accés al prototipus automàticament.

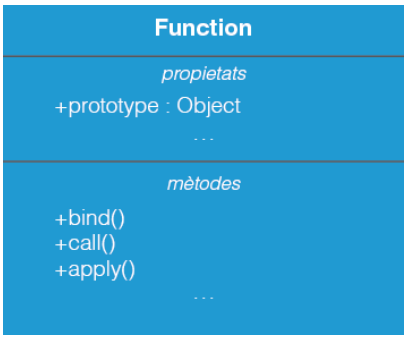
> Un **constructor** a JavaScript és una funció que crea un objecte. Per convenció, el nom d'aquesta funció sempre comença amb majúscules i la creació de l'objecte es realitza amb l'operador `new`.

```
let Persona = function (nom) {
  this.nom = nom;
};

Persona.prototype.saludar = function () {
  console.log('Hola, em dic ${this.nom}');
};

let jugador = new Persona('Ricard');
jugador.saludar();
```

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

Cal recordar que les funcions a JavaScript són objectes i, per consegüent, tenen propietats i mètodes. Podeu veure'n alguns a la figura 1.8.

Figura 1.8. Representació d'una funció



S'ha de tenir en compte que la propietat `prototype` dels objectes generats a partir d'un constructor és una referència al prototipus de la funció constructora i, conseqüentment, si s'augmenta o actualitza en un dels objectes aquest canvi afectarà el prototipus del constructor i de totes les instàncies generades.

Per aquesta raó **es desaconsella modificar el prototipus d'un objecte generat a partir d'un constructor** fent servir la seva propietat `prototype`. Si realment es vol modificar el seu prototipus, s'han d'aplicar els canvis a la propietat `prototype` de la funció constructora.

1.5.1 Augmentar objectes predefinits

Gràcies als prototipus és possible augmentar també els objectes predefinits del sistema, per exemple els objectes `Array`, `String` o `Number`; ja que una vegada augmentat un prototip s'aplica a tots els objectes dels quals forma part immediatament.

No és possible invocar un mètode a partir d'un número literal (per exemple `5`), però és posible invocar-lo a partir d'una variable amb el valor `5` assignat com a literal.

Per exemple, es pot augmentar el prototip de `Number` per afegir un mètode anomenat `duplicar` a tots els nombres:

```
1  let x = 9.5;
2
3  Number.prototype.duplicar = function () {
4    return this * 2;
5  };
6
7  let y = 21;
8
9  console.log(x.duplicar());
10 console.log(y.duplicar());
```

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

Com es pot apreciar, una vegada modificat el prototip de `Number`, tots els nombres de JavaScript tenen accés a aquest mètode, tant els que existien abans de l'augment com els nous.

Fixeu-vos també que s'ha fet servir `this` dins del nou mètode i el context s'ha

aplicat correctament, ja que en invocar al mètode `duplicar` primer s'ha cercat entre els mètodes de l'objecte i a continuació entre els mètodes del seu prototip, no es tracta d'una funció externa.

Cal remarcar que d'aquesta manera es poden augmentar objectes per accedir a una nova funcionalitat globalment, fins i tot entre diferents espais de nom, sense contaminar l'espai global. Veieu a continuació un altre exemple, aquest cop amb `String`:

```
String.prototype.revertir = function() {
  let revertit = '';
  for (let i = this.length - 1; i >= 0; i--) {
    revertit += this[i];
  }
  return revertit;
};

console.log('Bang!'.revertir());
```

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

En aquest cas, al contrari que amb els nombres, és possible cridar directament al mètode a partir de la cadena de caràcters.

1.6 Classes en JavaScript

Actualment, tots els navegadors admeten l'ús de classes de JavaScript, això ens permet treballar amb un sistema d'herència clàssica en lloc de l'herència prototípica pròpia de JavaScript, afegir mètodes d'accés, mètodes actualitzadors i mètodes estàtics.

1.6.1 Mètodes d'accés i actualitzadors (getters i setters)

Les classes permeten afegir mètodes d'accés i actualitzadors directament a les classes (cosa que també es pot fer amb ES5, però resulta més farragós):

```
class Persona {
  constructor(nom, cognom) {
    this._nom = nom;
    this._cognom = cognom;
  }

  set nom(nom) {
    this._nom = nom;
  }

  get nom() {
    return this._nom;
  }

```

```
  set cognom(cognom) {
    this._cognom = cognom;
  }

  get cognom() {
    return this._cognom;
  }

  get nomComplet() {
    return `${this._nom} ${this._cognom}`;
  }
}

let ricard = new Persona('Ricard', 'Ensutjs');

console.log('Nom (amb accessor):', ricard.nom);
console.log('Cognom (amb accessor):', ricard.cognom);
console.log('Nom complet:', ricard.nomComplet);
console.log('Nom (propietat):', ricard._nom);
console.log('Cognom (propietat):', ricard._cognom);
```

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

Fixeu-vos en l'*accessor* a `nomComplet` i com s'utilitza. No s'invoca, sinó que s'hi accedeix com si es tractés d'una propietat, però realment no existeix, el que es fa és generar-la dinàmicament a partir de les propietats `nom` i `cognom`. De la mateixa manera s'accedeix a les propietats `nom` i `cognom` que són generades dinàmicament (però sense variació) a partir de `_nom` i `_cognom` respectivament.

Malauradament, com es pot apreciar, les propietats `_nom` i `_cognom` continuen sent accessibles (i actualitzables) i per tant no s'està aplicant l'encapsulació de la informació. Per aquesta raó, la utilització d'*accessors* i actualitzadors només és interessant en casos molt marginals (per exemple, per generar propietats dinàmiques com el cas de `nomComplet`).

L'encapsulament de la informació encara no forma part de cap especificació ni esborrany, per tant cal recórrer a mètodes alternatius que escapen a l'abast d'aquests materials. Així doncs, la recomanació és fer servir el prefix '_' per distingir-les i en cas de requerir mètodes o propietats privades utilitzar la implementació del patró constructor o funcional per aquests objectes en lloc de les classes d'ES6.

Podeu trobar mètodes d'implementar l'encapsulació a JavaScript en l'enllaç següent: goo.gl/2t6jfU.

1.6.2 Mètodes estàtics

Les classes a JavaScript també permeten crear mètodes estàtics, és a dir, mètodes que poden utilitzar-se sense instanciar cap classe, per exemple:

```
class FactoriaArmes {
  static crearArma({tipus}) {
    let arma;

    switch (tipus) {
      case 'simple':
        arma = new Arma(arguments[0]);
```

```
        break;
      case 'arma de foc':
        arma = new ArmaAmbMunicio(arguments[0]);
        break;
      default:
        console.error(`No existeix aquest tipus d´\arma: ${params.tipus}`);
    }

    return arma;
  }
}
```

Tal com es pot apreciar només consta d'un mètode, però va precedit per la paraula clau `static`, això indica que es tracta d'un mètode estàtic i s'hi pot accedir sense instanciar la classe `FactoriaArmes`.

En aquest cas, la factoria crea instàncies de les classes `Arma` i `ArmaAmbMunicio` que han d'implementar un constructor que accepti un objecte com a paràmetre, per exemple:

```
class Arma {
  constructor: ({nom, potencia}) {
    // cos del constructor
  }
}

class ArmaAmbMunicio extends Arma {
  constructor: ({nom, potencia, maxMunicio}) {
    // cos del constructor
  }
}
```

Per instanciar els nous objectes a partir de la factoria i utilitzar els objectes es faria de la següent manera:

```
let ganivet = FactoriaArmes.crearArma({
  tipus: 'simple',
  nom: 'Ganivet',
  potencia: 1
});

let pistola = FactoriaArmes.crearArma({
  tipus: 'arma de foc',
  nom: 'Pistola',
  maxMunicio: 2,
  potencia: 2
});

ganivet.atacar('Zombi');
pistola.atacar('Gos Zombi');
pistola.atacar('Gos Zombi');
pistola.atacar('Gos Zombi');
```

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

Fixeu-vos que no s'instancia cap objecte fent servir l'operador `new`, es generen cridant al mètode estàtic `crearArma` que rep un objecte amb l'estructura de dades que conté la informació necessària per instanciar cada objecte.

Els mètodes estàtics resulten útils per la implementació de diferents patrons de

disseny com ara les *factories* o la creació de biblioteques, però s'ha de tenir en compte la complexitat afegida per encapsular la informació utilitzant el sistema de classes si realment és necessari utilitzar propietats o mètodes privats.

1.7 Herència

El paradigma de la programació orientada a objectes, a banda de permetre crear abstraccions de la realitat, facilita la reutilització del codi gràcies a l'herència.

En els llenguatges clàssics, l'herència permet crear noves classes que hereten d'unes altres. Això permet modificar-ne el comportament (polimorfisme) o augmentar-les.

Cal tenir en compte que a JavaScript no existeix el concepte d'**interfície** (*interface* en anglès) d'altres llenguatges clàssics, ja que no es pot forçar el compliment del "contracte" que representa la interfície en tractar-se d'un llenguatge dèbilment tipat.

> **Interficies**
>
> Una interfície és el conjunt de mètodes que un objecte ha de tenir per permetre la comunicació amb uns altres. Així doncs, tots els objectes amb aquests mètodes podran utilitzar-se indistintament com a component. Atès que l'especificació de JavaScript no inclou les interfícies i, consegüentment, no es pot forçar el comportament, el desenvolupador ha de parar atenció per respectar aquest "contracte".

Per altra banda, encara que JavaScript no suporta l'herència múltiple (un objecte que hereti de múltiples objectes), és possible implementar-la gràcies a la gran flexibilitat del llenguatge. Per exemple, utilitzant una funció per crear els *mix-ins* (mescles d'objectes).

Es pot trobar més informació sobre els *mix-ins* al següent enllaç: es.wikipedia.org/wiki/Mixin.

En aquests materials ens centrarem en l'herència clàssica fent servir classes en lloc de prototips.

1.7.1 Implementar l'herència

JavaScript ofereix diverses tècniques per reaprofitar el codi, ja que es poden crear jerarquies d'objectes utilitzant el sistema de classes o aplicant diferents patrons de generació (fins i tot combinant-los) amb alguna petita modificació, és a dir, no està restringit només al sistema de classes.

Podeu trobar més informació i exemples sobre les tècniques alternatives a l'ús de constructors per implementar l'herència prototípica en la secció "Annexos" d'aquesta unitat.

Cal tenir en compte que independentment del sistema empleat per implementar l'herència, els objectes generats funcionen exactament de la mateixa manera.

Herència a partir d'altres objectes

La implementació de l'herència fent servir aquesta tècnica és la més simple de totes. Consisteix a crear un primer objecte que serà el pare dels altres objectes (contindrà les propietats i mètodes comuns) i a continuació invocar `Object.create` per crear noves instàncies que podran ser actualitzar-se i augmentar-se.

Herència a través de la generació funcional

Les modificacions que s'han de fer per afegir un objecte pare a aquest patró són mínimes. Només cal substituir la creació de l'objecte buit per la generació de l'objecte pare, que seguidament serà actualitzat i augmentat segons sigui necessari.

Herència a partir de constructors

La creació d'objectes a partir de constructors és la tècnica que més s'assembla a l'herència clàssica sense utilitzar classes, ja que s'afegeix la propietat `prototype` que permet distingir quins objectes tenen un mateix pare, aplicar la cadena de prototipus i cridar mètodes del constructor pare.

La creació de constructors especialitzats requereix aplicar els canvis següents:

- Dins del constructor s'ha de cridar al constructor pare passant com a context el nou objecte i els arguments: `ConstructorPare.apply(this, arguments);`.
- La propietat `prototype` del constructor s'ha de substituir per una còpia de la del constructor pare: `ConstructorFill.prototype = Object.create(ConstructorPare.prototype);`.
- Una vegada substituït el prototipus s'ha de corregir el constructor perquè apunti al nou constructor: `ConstructorFill.prototype.constructor = ConstructorFill`.

Com podeu imaginar, la implementació de l'herència fent servir constructors es força complicada, per aquest motiu es recomana fer servir el sistema d'herència mitjançant classes que es va afegir a ES2015.

Herència a mitjançant el sistema de classes

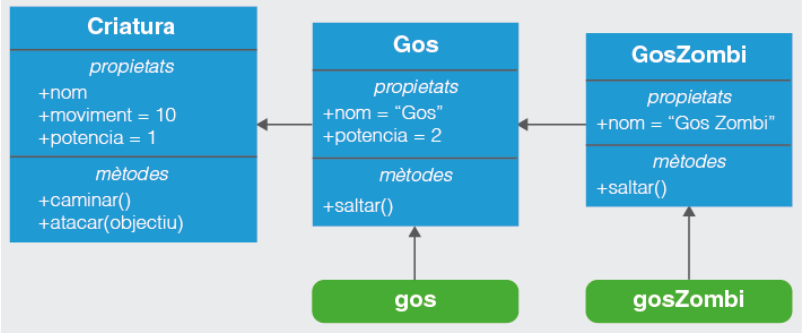
Actualment JavaScript permet utilitzar el sistema de classes molt similar al que trobem en altres llenguatges com Java o C++, això ens permet crear subclasses fàcilment, només cal utilitzar la paraula clau `extends`:

```
1 // Superclasse
2 class Criatura {
3   constructor() {
```

```
    this.nom = 'Criatura de tipus desconegut';
    this.moviment = 10;
    this.potencia = 1;
  }

  caminar() {
    console.log(`${this.nom} camina ${this.moviment} metres`);
  }

  atacar(objectiu) {
    console.log(`${this.nom} ataca a ${objectiu.nom} i li causa ${this.potencia
        } punts de dany`);
  }
}

class Gos extends Criatura {
  constructor() {
    super();
    this.nom = "Gos";
    this.potencia = 2;
  }

  saltar() {
    console.log(`${this.nom} salta ${this.moviment / 5} metres`);
  }
}

class GosZombi extends Gos {
  constructor() {
    super();
    this.nom = "Gos Zombi";
  }

  saltar() {
    console.log(`${this.nom} intenta saltar... però no pot`);
  }
}

// Es generen les instancies del objectes a partir de les classes
let gos = new Gos(),
    gosZombi = new GosZombi();

gos.caminar();
gos.atacar(gosZombi);
gos.saltar();
gosZombi.atacar(gos);
gosZombi.saltar();
gosZombi.caminar();
```

Podeu veure aquest exemple en l'enllaç següent: https://codepen.io/ioc-daw-m06/pen/poyNKPr?editors=0012 i el diagrama corresponent a la figura 1.9.

FIGURA 1.9. Herència fent servir classes



Els noms de les classes comencen amb majúscula mentre que els objectes instanciats ho fan amb minúscula.

Tal com es pot apreciar, encara que els mètodes `caminar` i `atacar` no s'han definit a les classes `Gos` ni `GosZombi`, es poden invocar correctament perquè totes dues tenen com a superclasse `Criatura`, que és on es troben definits aquests mètodes.

Per altra banda, podem veure que totes dues subclasses reassignen el valor de la propietat `nom` dins del constructor, per consegüent encara que no s'hagi indicat cap nom en crear les instàncies s'assigna automàticament.

La classe `Gos` afegeix el mètode `saltar`, augmentant la classe `Criatura` i la classe `GosZombi` la sobreescriu, modificant el comportament (polimorfisme). És a dir, quan s'invoca aquest mètode des d'una instància de `Gos` es crida al mètode de `Gos`, però quan s'invoca des d'una instància de `GosZombie` el codi executat és el corresponent al mètode definit en `GosZombie`.

Quant a jerarquia de classes, es pot comprovar que es creen les referències correctament i es pot determinar si un objecte és instància d'una classe determinada. Proveu d'afegir les següents línies a l'exemple anterior:

```
1 console.log('GosZombie és una instància d\'Object?', gosZombi instanceof Object
      );
2 console.log('GosZombie és una instància d\'Criatura?', gosZombi instanceof
      Criatura);
3 console.log('GosZombie és una instància d\'Gos?', gosZombi instanceof Gos);
4 console.log('GosZombie és una instància d\'GosZombie?', gosZombi instanceof
      GosZombi);
```

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

Fixeu-vos que dins dels constructors de les subclasses s'ha utilitzat la paraula clau `super`. Aquesta paraula clau es pot fer servir de dues maneres diferents:

```
1 super([arguments]); // crida al constructor de la superclasse amb els arguments
2 super.funcioEnSuperclasse([arguments]); // crida a la funció concreta de la
      superclasse
```

Cal tenir en compte que encara que els arguments són opcionals, s'han de passar els paràmetres adequats en invocar `super` (o a la funció de la superclasse), en cas contrari el resultat no serà l'esperat. Per exemple, si el constructor de la superclasse espera el paràmetre `nom` i des de la subclasse no es passa el seu valor serà `undefined`.

> Tingueu en compte que dins dels constructors cal invocar a `super` **abans de poder utilitzar** `this`, en cas contrari es produirà un error.

Veiem un exemple d'utilització de `super` en les dues situacions:

```
1 class Arma {
2   constructor(nom) {
3     this.nom = nom;
4     this.so = 'Zas!'
5     this.potencia = 1;
6   }
7
8   atacar(objectiu) {
```

```
9       console.log(`${this.so} S´\han causat ${this.potencia} punts de dany a ${
            objectiu} amb ${this.nom}`);
10    }
11  }
12
13  class ArmaAmbMunicio extends Arma {
14    constructor(nom, maxMunicio) {
15      super(nom);
16      this.municio = maxMunicio;
17      this.so = 'Bang!';
18      this.potencia = 2;
19    }
20
21    atacar(objectiu) {
22      if (this.municio > 0) {
23        this.municio--;
24        super.atacar(objectiu);
25      } else {
26        console.log("Click! no queda munició!");
27      }
28    }
29  }
30
31  let ganivet = new Arma('Ganivet');
32  let pistola = new ArmaAmbMunicio('Pistola', 2);
33
34  ganivet.atacar('Zombi');
35  pistola.atacar('Gos Zombi');
36  pistola.atacar('Gos Zombi');
37  pistola.atacar('Gos Zombi');
```

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

Com es pot apreciar, quan s'invoca `atacar` des de la pistola només es comprova que hi hagi munició, si hi ha munició es crida al mètode `atacar` de la superclasse i, en cas contrari, mostra un missatge.

Fixeu-vos que quan s'invoca a `super` en el constructor de `ArmaAmbMunicio` només es passa el paràmetre `nom` i no `maxMunicio`, ja que la superclasse només accepta un paràmetre.

1.7.2 Herència múltiple: mix-ins

L'herència múltiple consisteix en el fet que una classe hereti de múltiples classes. Per exemple es poden definir dues classes diferents que no estiguin relacionades però que tinguin subclasses que comparteixin algun comportament.

Utilitzant l'herència simple hauríem de duplicar el codi del comportament compartit a cada subclasse, en canvi utilitzant l'herència múltiple es podria definir el nou comportament en una classe externa i fer que les subclasses heretin de la seva superclasse i d'aquesta nova classe.

Mix-ins

És una associació de mètodes i propietats que, després, poden ser heretats per qualsevol classe.

Tot i que JavaScript no permet l'herència múltiple directament, es pot implementar utilitzant *mix-ins*. Per crear un mix-in, en lloc de definir una classe, el que fem és definir una funció que:

- rep com a paràmetre la classe a la qual s'ha d'afegir el mix-in.
- retorna una nova classe que té com a superclasse la classe passada com a paràmetre.

Per exemple, per crear un nou mix-in que afegeixi un nou comportament definirí-em la funció així:

```
let nouMixin = Base => class extends Base {
  nouComportament() { }
};
```

Fixeu-vos que la funció s'ha definit com una funció de fletxa. Això no és necessari però fa que el codi sigui més net.

Tingueu en compte que en aquest exemple només hem afegit un nou comportament, però un mix-in pot contenir qualsevol quantitat de mètodes i propietats, com una classe normal.

Per crear una subclasse que utilitzi aquest mix-in es faria de la següent manera:

```
class ClasseBase {
  // mètodes i propietats de ClasseBase
};

class NovaClasse extends nouMixin(ClasseBase) {
  // mètodes i propietats de NovaClasse
}
```

Com es pot apreciar, es crida a la funció `nouMixin` passant com a paràmetre la classe que volem fer servir com a classe base. Si volguéssim afegir més mix-ins només caldria niuar les invocacions a les funcions de manera que el resultat d'una passa com argument de la següent:

```
class NovaClasse extends nouMixin1(nouMixin2(nouMixin3(ClasseBase))) {
  // mètodes de NovaClasse
}
```

Vegeu a continuació un exemple complet:

```
class Arma {
  constructor({nom}) {
    this.nom = nom;
    this.so = "Zas!";
    this.potencia = 1;
  }

  atacar(objectiu) {
    console.log(
      `${this.so} S\'han causat ${this.potencia} punts de dany a ${objectiu}
          amb ${this.nom}`
    );
  }
}

let carguesMixin = (Base) =>
  class extends Base {
    constructor({cargues}) {
      super(arguments[0]);
      this.cargues = cargues;
```

```
      this.maxCargues = cargues;
    }

    recarregar() {
      this.cargues = this.maxCargues;
      console.log(`${this.nom} recarregada`);
    }
  };

class ArmaDeFoc extends carguesMixin(Arma) {
  constructor({nom}) {
    super(arguments[0]);
    this.so = "Bang!";
    this.potencia = 2;
  }

  atacar(objectiu) {
    if (this.cargues > 0) {
      this.cargues--;
      super.atacar(objectiu);
    } else {
      console.log("Click! no queda munició!");
    }
  }
}

let ganivet = new Arma({nom: "Ganivet"});
let pistola = new ArmaDeFoc({nom: "Pistola", cargues: 2});

ganivet.atacar("Zombi");
pistola.atacar("Gos Zombi");
pistola.atacar("Gos Zombi");
pistola.atacar("Gos Zombi");
pistola.recarregar();
pistola.atacar("Gos Zombi");


class Dispositiu {
  constructor ({nom}) {
    this.nom = nom;
  }

  utilitzar() {
    console.log(`Utilitzant ${this.nom}`);
  }
}

class DispositiuAmbPiles extends carguesMixin(Dispositiu) {

  utilitzar() {
    if (this.cargues > 0) {
      this.cargues--;
      super.utilitzar();
    } else {
      console.log("Piles esgotades!");
    }
  }
}

let llinterna = new DispositiuAmbPiles({nom: 'llinterna', cargues: 3});
llinterna.utilitzar();
llinterna.utilitzar();
llinterna.utilitzar();
llinterna.utilitzar();
```

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

Fixeu-vos que s'ha fet servir un objecte com a paràmetre dels constructors (on són desestructurats). D'aquesta manera es poden passar els paràmetres per nom sense importar l'ordre. Això permet que el mix-in només agafi el paràmetre que necessita `cargues`, mentre que les superclasses agafen el paràmetre `nom`.

Atès que el constructor rep un objecte desestructurat, no podem passar a `super` només el nom del paràmetre, ja que llavors es perdria la resta d'informació passada originalment com objecte. Afortunadament podem accedir a l'objecte original mitjançant l'objecte `arguments`, i com només es passa un paràmetre podem assegurar que l'índex 0 conté aquest objecte. Per consegüent, amb `super(arguments[0])` es passa l'objecte original als constructors de les classes pares.

objecte arguments

És un objecte especial similar a un array que rep cada funció i conté tots els paràmetres indexats per ordre. És a dir, l'índex 0 conté el primer paràmetre, l'índex 1 el segon, l'índex 2 el tercer, etc.

Com es pot apreciar, s'han fet servir dues classes base diferents:, `Arma` i `Dispositiu`, i s'ha afegit el mix-in `carguesMixin` a les subclasses `ArmaDeFoc` i `DispositiuAmbPiles`, de manera que totes dues compareixen les propietats `cargues` i `maxCargues` i el mètode `recargar`.

En conclusió, la utilització dels mix-ins permet afegir a subclasses, que pertanyen a diferents jerarquies de classes, un comportament compartit sense duplicar el codi.

1.7.3 Patró: factoria

A causa de la complexitat que pot presentar la generació d'objectes i les jerarquies d'herència a JavaScript, de vegades pot ser recomanable utilitzar alguna tècnica que simplifiqui aquesta tasca.

Una opció és utilitzar el **patró factoria**, un patró de disseny que amaga la implementació de la creació d'objectes i exposa un mètode a través del qual es poden obtenir instàncies dels objectes sense necessitat de conèixer la seva implementació.

Podeu trobar més informació sobre el *patró factoria* en l'apartat "Annexos" del web del mòdul.

És a dir, la implementació d'aquest patró ens ajuda a crear un objecte a través del qual invocant un mètode es generen completament altres objectes, per exemple una *factoria* d'armes ens permet invocar els mètodes: `factoria.crearArma('ganivet')`, `factoria.crearArma('subfusil')`, etc.

Patrons de disseny

Els patrons de disseny són solucions comunes a problemes de disseny de programari. Van ser introduïts pel *Gang of Four* el 1994. Aquestes solucions són aplicables a tots els llenguatges de programació. Podeu trobar-ne més informació en l'enllaç següent: bit.ly/3bznlEV.

1.7.4 Composició i delegació

Una tècnica molt potent que afegeix una gran flexibilitat a la utilització d'objectes és la **composició**. Consisteix a afegir objectes com a propietats de l'objecte compost, de manera que es pot dividir la complexitat entre diferents components.

> Un dels principis de la programació orientada a objectes recomana **afavorir la composició per sobre de l'herència** (*composition over inheritance* en anglès), ja que la composició és molt més flexible i l'herència pot portar fàcilment a la duplicació de codi i la generació de jerarquies d'objectes molt més complexes.

Per determinar si cal utilitzar una composició o una relació d'herència podeu comprovar si la relació entre els dos objectes és de *ser* o de *tenir*:

- Un gos zombi **és** un zombi, per tant, és una relació d'**herència**.
- Ricard **té** una arma, així doncs, es tracta d'una **composició**.

Simplificació dels diagrames

Per simplificar els diagrames s'ha optat per fer servir la representació d'associació (una línia contínua) quan la composició és amb un sol objecte i la d'agregació (rombe contigu a l'objecte compost) quan són múltiples (per exemple, un *array* d'objectes).

Per altra banda, per fer més entenedors els diagrames es mostra tant la propietat a l'objecte com l'associació, encara que allò correcte és que només s'utilitzi una representació o l'altra (associació/composició o propietat).

A diferència dels llenguatges clàssics, a JavaScript qualsevol objecte pot formar part d'una composició (no hi ha restriccions per classe o interfície). Això permet una gran flexibilitat però, per altra banda, és possible establir com a propietats objectes que no implementin els mètodes necessaris.

En el següent exemple podeu veure com les instàncies de `Persona` estan compostes per una propietat, `arma`, on s'espera que s'estableixi un objecte amb el mètode `atacar`. Una vegada es detecta que s'ha establert la propietat, en invocar al mètode `atacar` es delega la invocació a l'objecte emmagatzemat a `arma`.

D'aquesta manera només actualitzant la propietat `arma` es pot canviar el comportament de l'objecte, per exemple: quan es fa servir el ganivet es pot atacar un nombre il·limitat de vegades, però en utilitzar la `pistola` s'afegeix una complexitat extra ja que s'ha afegit el control de munició.

Fixeu-vos que sense haver de tocar la implementació de `Persona` es podrien crear nous tipus d'armes amb funcionaments molts diferents, sempre que complissin amb els següents requisits:

- El nou tipus d'arma té assignada a la propietat `atacar` una funció, és a dir, és tracta d'un mètode.
- Aquest mètode accepta un paràmetre, que serà l'objectiu de l'atac.
- Opcionalment, el constructor podrà rebre un o dos paràmetres, corresponents a les propietats `nom` i la `potencia`.

```
1 class Arma {
2   constructor(nom, potencia) {
```

```
    this.nom = nom;
    this.potencia = potencia;
  }

  atacar(objectiu) {
    console.log(`Zas! ${this.personatge.nom} ataca amb ${this.nom} i causa ${
        this.potencia} punts de dany a ${objectiu}`);
  }
}

class ArmaAmbMunicio extends Arma {
  constructor (nom, potencia, municio) {
    super(nom, potencia);
    this.municio = municio;
  }

  atacar (objectiu) {
    if (this.municio > 0) {
      console.log(`Bang! ${this.personatge.nom} ataca amb ${this.nom} i causa $
          {this.potencia} punts de dany a ${objectiu}`);
      this.municio--;
    } else {
      console.log('Click! no hi ha munició!');
    }
  }
}

class Persona {
  constructor (nom) {
    this.nom = nom;
  }

  atacar(objectiu) {
    if (this.arma) {
      this.arma.atacar(objectiu)
    } else {
      console.log('No es pot atacar perquè no hi ha cap arma equipada!');
    }
  }

  equipar(arma) {
    this.arma = arma;
    arma.personatge = this;
    console.log(this.nom + ' ha equipat ' + arma.nom);
  }
}


let ricard = new Persona('Ricard');
let ganivet = new Arma('Ganivet', 2);
let pistola = new ArmaAmbMunicio('Pistola', 3, 2);

ricard.atacar('Zombi');
ricard.equipar(ganivet);
ricard.atacar('Zombi');
ricard.equipar(pistola);
ricard.atacar('Zombi');
ricard.atacar('Zombi');
ricard.atacar('Zombi');
```

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

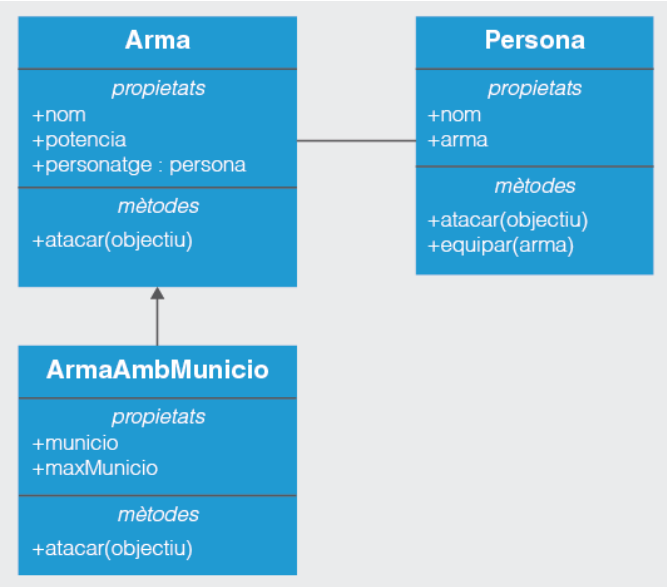
i el diagrama corresponent a la figura 1.10.

Tal com es pot apreciar, la clau de la composició és assignar un objecte a una propietat (en aquest cas a través del mètode `equipar`) i delegar de-

terminades accions a aquests objectes (com el mètode `atacar` de l'exemple: `this.arma.atacar(objectiu);`).

Com que `ArmaAmbMunicio` hereta d' `arma` i aquesta té el mètode `atacar` que és el requisit per poder equipar-lo (encara que no se'n força la comprovació), es pot assegurar que, encara que no s'actualitzés el valor d' `atacar`, tots els objectes instanciats per aquest constructor es podran equipar correctament.

FIGURA 1.10. Constructor d'objecte compost Persona amb Arma



S'ha afegit la propietat 'personatge' amb valor 'undefined' perquè és afegida per la Persona quan s'invoca el mètode equipar.

La composició es pot realitzar de diverses maneres. En aquest cas s'ha fet a través d'un mètode, però es podria haver generat l'objecte al constructor. Proveu de substituir la declaració del constructor de `Persona` per la següent:

```
constructor (nom) {
  this.nom = nom;
  let puny = new Arma('Cop de puny', 1);
  this.equipar(puny);
}
```

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

En aquest cas s'ha creat un nou objecte `puny` i s'ha equipat. Fixeu-vos que no s'ha assignat directament a la propietat `arma`, sinó que s'ha cridat al mètode `equipar`. Pareu atenció a la declaració del mètode:

```
equipar(arma) {
  this.arma = arma;
  arma.personatge = this;
  console.log(this.nom + ' ha equipat ' + arma.nom);
}
```

Primerament es mostra un missatge i seguidament s'actualitza la propietat `arma` amb l'objecte passat com a paràmetre. Però, a continuació, s'augmenta l'objecte `arma` amb la propietat `personatge` i el valor `this`. El valor de `this` en aquest

cas és la instància de `Persona`. Així doncs, si la instància de `Persona` és l'objecte referenciat `ricard`, en equipar un ganivet es pot entendre que `arma.personatge = ricard;`.

D'aquesta manera, `ganivet` té la informació de qui l'ha equipat i pot fer servir `this.personatge.nom` per accedir al `nom` de la `Persona`. És a dir, l' `Arma` té accés a tota la informació de la `Persona` que l'equipa i per tant en un cas real es podria fer servir per determinar la probabilitat d'encertar un atac, sumar la potència a la del personatge, etc.

Un altre detall a tenir en compte és que s'ha aplicat l'augment al paràmetre `arma` i no a la propietat `this.arma` (que faria referència a la instància de `Persona`). Es podria haver fet de les dues maneres:

```
1 arma.personatge = this; // Paràmetre
2 this.arma.personatge = this; // Propietat d'instància de Persona
```

La raó per la qual el resultat és el mateix d'una forma o l'altra és que `arma` és un objecte i, per tant, s'està treballant amb una referència a aquest i qualsevol canvi al paràmetre o a la propietat afecten el mateix objecte.