El llenguatge javascript

IES Jaume II El Just
Tavernes de la Valldigna

Aplicacions multiplataforma amb tecnologies web

Curs 20-21

Continguts

1. Aspectes generals 3
2. Elements del llenguatge 3
    2.1. Variables 3
    2.2. Operadors 7
3. Funcions 9
    3.1. Àmbit de les variables (scope) 10
4. Programació estructurada en Javascript 12
    4.1. Estructures condicionals 12
    4.2. Estructures repetitives 15
5. Programació Orientada a Objectes en JS 18
    5.1. Orientació a Objectes basada en prototipus 18
    5.2. Orientació a Objectes a partir de ES6 23

1. Aspectes generals

Com hem comentat, nodeJS es basa en el motor de Javascript V8 de Google Chrome, pel que començarem amb una introducció a aquest llenguatge per començar amb nodeJS.

Veiem algunes generalitats del llenguatge:

- Un programa o script en Javascript està compost de sentències, que poden o no acabar en `;`. Tot i que no és obligatori, és recomanable el seu ús.
- Els espais en blanc sobrants i les noves línies, no es tenen en compte, pel que admet qualsevol tipus de tabulació.
- Els bloc de codi es defineixen seguit l'estil de llenguatges com C o Java, amb claus `{...}`.
- Es tracta d'un llenguatge *case sensisive*, pel que disgingeix majúscules i minúscules.
- Els comentaris també segueixen una sintaxi similar a C o Java:
  - `//` Dues barres per als comentaris d'una línia
  - `/*` Barra i asterisc per a comentaris de més d'una línia `*/`
- **Paraules reservades**: Aquelles que s'utilitzen per construir sentències, i que no es poden utilitzar lliurement. Alguens d'aquestes són: **`break`**, **`case`**, **`catch`**, **`class`**, **`continue`**, **`default`**, `delete`, **`do`**, **`else`**, **`finally`**, **`for`**, `function`, **`if`**, `in`, **`instanceof`**, **`new`**, **`return`**, **`switch`**, **`this`**, **`throw`**, **`try`**, `typeof`, `var`, **`void`**, **`while`**, `with`.
- Tot i que Javascript suporta programació orientada a objectes, no estem obligats a utilitzar-los. Per tant, no és necessari definir un mètode Main dins una classe, sinò que directament l'execució comença amb la primera línia de codi executable.

2. Elements del llenguatge

2.1. Variables

No és necessari declarar variables en Javascript, tot i que és altament recomanable. Per tal de declarar una variable fem:

```
1 let variable=valor;
```

També es pot fer ús de `var` en lloc de `let`, però des de la versió ES2015, la recomanació és utilitzar `let`.

El mode strict Javascript ens permet treballar en mode *strict*, que ens obliga a declarar les variables abans d'utilitzar-les.

Per exemple, el següent script (saluda.js):

```
nom="Jose";
console.log("Hola "+nom);
```

Funcionaría correctament:

```
$ node saluda.js
Hola Jose
```

Però si habilitem el mode estricte:

```
'use strict';

nom="Jose";
console.log("Hola "+nom);
```

Ens donaría el següent error:

```
/tmp/prova1.js:3
nom="Jose"
   ^

ReferenceError: nom is not defined
    at Object.<anonymous> (/tmp/saluda.js:3:4)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:663:10)
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
    at Function.Module._load (module.js:497:3)
    at Function.Module.runMain (module.js:693:10)
    at startup (bootstrap_node.js:188:16)
    at bootstrap_node.js:609:3
```

Com veiem, el treballar amb el mode strict, ens obliga a definir variables, de manera que detectem fàcilment quan cometem alguna errada escrivint el nom d'una variable.

El nom de les variables

- El nom de les variables pot contindre qualsevol caràcter alfanumèric, més el símbol $ i _.
- Poden començar per qualsevol caràcter, més el $ i el _,però no per un número.
- El nom El nom de les variables sol indicar-se amb format **CamelCase** (la primera lletra de cada paraula en majúscules i la resta en minúscules).

Tipus de variables i assignació dinàmica Una variable també pot ser inicialitzada en la declaració, i pot ser qualsevol valor, o el resultat d'una expressió.

```
let variable=[ valor | expressió ];
```

Una variable en javascript emmagatzema dades d'un tipus concret en un moment donat. Mitjançant l'**assignació dinàmica de tipus**, el tipus de dada que emmagatzema aquesta variable pot canviar en temps d'execució. És a dir, el tipus d'una variable es defineix de forma dinàmica en el moment de la seua assignació, no de la declaració.

Les variables poden ser:

- **Variables de tipus numèriques (number)**: Emmagatzema valors enters o decimals (amb el caràcter . per separar la part entera de decimal).
- **Tipus text (string)**: Emmagatzema una llista de caracters, entre cometes, bé amb cometes simples o dobles. En cas que el text continga cometes, cal anar amb compte amb la construcció, no podent utilitzar internament les cometes que hem utilitzat per definir la cadena de text o bé escapant estes:

```
var text="Aqueta frase utilita 'cometes simples'"
var text="Aquesta frase utilitza \"cometes dobles escapades\""
```

```
* Dins les cadenes de text, també podem incloure:
    * Salts de línia: \n
    * Tabuladors: \t
    * Barra inclinada: \\
    * Cometes escapades: \" o \'
```

Conversió de tipus

La conversió de tipus es realitza de forma automàtica. En cas d'ambigüetat, utilitza prioritats:

Exemple: `'10'+3` : Suma un string i un number. En este cas, té prioritat la cadena de caràcters, pel que el number 3 es converteix al string «3». En canvi, quan es tracta del + unari, només s'aplica als number, pel que +«20», representaria el número 20.

Vegem alguns exemples més:

```
13 + 7 => 20
"13" + "7" => "137"
"13" + 7 => "137"
+"13" + 7 => 20
```

Tipus lògic: Poden contenir els valores true i false.

```
let itemSelected=false;
let display=true;
```

- **Valors null i undefined**: Quan una variable no s'ha definit, el tipus d'aquesta és *undefined*. En canvi, quan volem fer referència al valor nul, fem ús de *null*, sense cometes. Com veurem, el valor null serà un tipus especial d'objecte.

Typeof

Per tal de saber el tipus de dada que conté una variable en un moment donat, podem fer ús de l'operador `typeof`:

```
> typeof(10)
'number'
> typeof("hola")
'string'
> typeof(true)
'boolean'
> typeof(a)
'undefined'
> typeof(null)
'object'
> typeof({"nom":"jose"})
'object'
```

Vectors j JSON Els vectors són col·leccions de valors/elements, bé del mateix tipus o de tipus diferent (a diferència d'altres llenguatges, que els tipus han de ser homogenis).

Per definir el vector utilitzem els caracters [i] al principi i al final, i utilitzem les comes (,) per separar elements.

```
let nom_vector = [valor1, valor2, ..., valorN];
```

Exemple:

```
let dies_setmana=["dilluns", "dimarts", "dimecres", "dijous", "
    divendres", "dissabte", "diumenge"];
let ex_vector=["cadena1", num1, "cadena2", ["cadena1_vector1"]];
```

Per tal d'accedir als elements, indiquem la posició en el vector (la posició inicial és la 0):

- **Consulta**: `var hui=dia[1];`
- **Assignació**: `ex_vector[1]="cadena3";`
- Un vector buit, es pot definir inicialitzant-lo a buit, o creant un nou objecte de la classe Array:

```
ex_vector2=[];
ex_vector3=new Array();
```

Per la seua banda, els elements *JSON* (JavaScript Object Notation) són una forma més compacta d'emmagatzemar i transmetre informació que el format XML, a través de parells clau-valor.

Si recordem de la primera unitat sobre fitxers, els tipus de dades que podem representar en JSON són:

- **Números**, tant enters com decimals,
- **Cadenes**, expressades entre cometes i amb la possibilitat d'incloure seqüències d'escapament,
- **Booleans**, per representar els valors *true* i *false*,
- **Null**, per representar el valor nul,
- **Array**, per representar llistes de zero o més valors, de qualsevol tipus, entre corxets i separats per comes,
- **Objectes**, com a col·leccions de parells `<clau>:<valor>`, separats per comes i entre claus, i de qualsevol tipus de valor.

Recordem un xicotet exemple d'aquesta notació:

```
{
    "curs": [
            {
                "nom": "Accés a Dades",
                "hores": 6,
                "qualificacio":  8.45
            },
            {
                "nom": "Programació de serveis i processos",
                "hores": 3,
                "qualificacio": 9.0
            }
        ]
}
```

A diferència de Java, Javacript suporta la manipulació d'elements JSON de forma nativa, de manera que no necessitem cap llibreria per tal de poder utilitzar aquest format.

Declaració de constants Quan no desitgem que canvie el valor d'una variable o si volem definir un valor global immutable, es recomana fer ús de la paraula reservada **`const`** en lloc de `let`. Per a les constants globals es recomama utilitzar majúscules.

```
const PI=3.1416;
```

2.2. Operadors

Les variables es combinen amb elles, formant expressions mitjançant els operadors. Aquestes expressions, podran avauar-se, donant lloc a un nou valor, que pot utilitzar-se per prendre decisions en un script, o bé per assignar-lo a noves variables.

Els operadors poden ser d'assignació, aritmètics, lògics o relacionals.

Operador d'assignació (=) S'utilitza per assignar a una variable un valor o expressió.

```
let variable1 = Expressió;
```

Operadors aritmètics Els operadors artimètics són aquells que treballen amb valors de tipus numèrics per tal d'obtindre altres valors de tipus numèric. A banda dels habituals de *suma(+)*, *resta (-)*, *multiplicació ()*, *divisió (/)*, i *resta de la divisió (%)*, tenims els següents operadors:

- **Operadors d'increment (++) i decrement (–)**: per incrementar o decrementar el valor d'una variable, i no té el mateix significat posar-lo davant que darrere de la variable.

```
let a=2
let b=2
let c=++a + b;
>> c = 5
>> a = 3
```

No és el mateix que:

```
a2=2
b2=2
c2=a2++ + b2
>> c2 = 4
>> a2 = 3
```

- **Suma Combinada (+=)**: A l'igual que en C i Java, podem fer ús d'expressions tipus `a+=3` per simplificar l'expressió `a=a+3`.

Operadors lògics Són aquells que s'apliquen sobre valors lògics i retornen un valor també lògic.

- **Negació**: !

```
> let  a=true;
> !a;
false
```

Quan s'aplica sobre valors no lògics el significat és el següent: El valor numèric de 0 i la cadena de text buida es converteixen al valor lògic *false*, mentre qu eun número diferent de 0 i una cadena no buida es consideren *true*.

- **Operador and (&&)**: El resultat és cert si els dos operands ho són. **Operador or (||)**: El resultat és cert si un dels dos operands ho és.

Operadors relacionals Són aquells que operen amb qualsevol tipus de dada, comparant-los i ens proporcionen un valor de tipus lògic.

Operador	Semàntica
>	Major que
<	Menor que
>=	Major o igual
<=	Menor o igual
== (Compte amb =)	Igual que `5=="5"==> true`
===	Igual que, en valor i tipus. `5==="5"==> false`
!=	Distitnt de
!==	Distint, o de tipus distint

Quan treballem amb cadenes, els operadors relacionals el que fan és comparar lletra a lletra, els caràcters ASCII de dos cadenes, d'esquerra a dreta.

Altres operadors A banda dels operadors tradicionals, javascript aporta alguns operadors més:

- **Operador condicional ternari**: L'operador condicional ternari, permet assignar un valor a una variable, basant-se en certa condició:

```
variablename = (condition) ? value1:value2
```

Per exemple:

```
let  pot_votar = (edat < 18) ? "Massa jove":"Adult";
```

- **Operador Typeof**: Com ja hem comentat abans, aquest operador obté el tipus de valor d'una variable.

3. Funcions

La delcaració d'una funció en Javascript es declara amb la paraula reservada `function`. A diferència de Java, les funcions poden declarar-se fora d'un objecte.

```
function nom_funcio(llista_arguments) {
  // Cos de la funció
```

```
3   }
```

El nom de la funció s'utilitza per fer referència a ella des de qualsevol part del codi. Si volem que la funció ens torne algun valor, i assignar aquest a una variable, farem ús de la paraula reservada **`return`**. L'ordre dels arguments és important (pas de paràmetres posicional). A més, el nom de les variables utilitzades quan s'invoca la funció no té per què coincidir amb el nom que li donem als arguments en la declaració.

Exemple:

```
function suma(numero1, numero2) {
  resultat = numero1 + numero2;
  return resultat;
}

s=suma(a, b);
```

També cal recordar que una funció torna únicament un valor, pel que en cas que necessitem tornar informació variada, podem fer ús de JSON.

Exemple:

```
function calculaPreu(preu) {
    preu_iva=(preu*1.21).toFixed(2);  // Arrodonim a dos decimals
    preu_iva_reduit=(preu*1.1).toFixed(2);
  return {"preu_amb_iva": preu_iva, "preu_iva_red":preu_iva_reduit}
}

let preus = calculaPreu(50);
console.log(preus);
>> Object { preu_amb_iva: "60.50", preu_iva_red: "55.00" }
```

3.1. Àmbit de les variables (scope)

L'àmbit d'una variable és la part del programa on aquesta està definida, i on pren sentit. L'àmbit pot ser:

- **global**, visible des de tot l'script, o
- **local**, visible només en la funció on es troba declarada (amb var o let).

Per entendre bé l'àmbit de les variables, cal tindre en compte alguns aspectes:

- Javascript és un llenguatge amb *àmbit global* com a àmbit predeterminat, i on tot es passa també de forma predeterminada per referència: quan declarem una variable fora d'una funció, aquesta és global, i es passa per referència als àmbits descendents o heratats (per tant, qualsevol modificació que fem a la variable en qualsevol àmbit tindrà efecte sobre ella).

- Quan declarem una variable amb `var` dins una funció, es crea un variable local, amb visibilitat restringida a eixa funció.
- Quan definim una variable amb `let` o una constant amb const, el seu àmbit es redueix als **blocs** on està definida.

Veiem-ho amb un exemple. El següent codi defineix algunes variables, amb let i var en diferents àmbits:

```
var var1=1;
var var2=2;

function fun(){
    var1=3;
    var var2=4;
    var var3=5;
    let var4=6;
    {
        var var5=7;
        let var6=8;
    }

    console.log("Àmbit: Dins la funció:");
    if (typeof(var1)!="undefined") console.log("var1="+var1);
        else console.log("var1 no està definida");
    if (typeof(var2)!="undefined") console.log("var2="+var2);
        else console.log("var2 no està definida");
    if (typeof(var3)!="undefined") console.log("var3="+var3);
        else console.log("var3 no està definida");
    if (typeof(var4)!="undefined") console.log("var4="+var4);
        else console.log("var4 no està definida");
    if (typeof(var5)!="undefined") console.log("var5="+var5);
        else console.log("var5 no està definida");
    if (typeof(var6)!="undefined") console.log("var6="+var6);
        else console.log("var6 no està definida");

}

fun();

console.log("Àmbit: Fora la funció:");
if (typeof(var1)!="undefined") console.log("var1="+var1);
    else console.log("var1 no està definida");
if (typeof(var2)!="undefined") console.log("var2="+var2);
    else console.log("var2 no està definida");
if (typeof(var3)!="undefined") console.log("var3="+var3);
    else console.log("var3 no està definida");
if (typeof(var4)!="undefined") console.log("var4="+var4);
    else console.log("var4 no està definida");
if (typeof(var5)!="undefined") console.log("var5="+var5);
    else console.log("var5 no està definida");
```

```
43 if (typeof(var6)!="undefined") console.log("var6="+var6);
44     else console.log("var6 no està definida");
```

Pareu-se un moment a pensar l'eixida que tindría aquest script, i una vegada ho hajau pensat, veieu-ne el resultat per contrastar:

```
1  $ nodejs ambits.js
2  Dins la funció:
3  var1=3
4  var2=4
5  var3=5
6  var4=6
7  var5=7
8  var6 no està definida
9  Dins la funció:
10 var1=3
11 var2=2
12 var3 no està definida
13 var4 no està definida
14 var5 no està definida
15 var6 no està definida
```

Algunes observacions:

- Dins la funció, tenim accés a totes les variables definides, tant fora, com dins, a excepció de la variable var6, que s'ha definit dins un bloc de codi amb la paraula reservada `let`.
- Dins la funció, var1 i var2 han modificat el seu valor, però fora de la funció, la modificació s'ha mantés només per a var1, però var2 no s'ha vist modificat. Açò es deu a que la modificació realitzada sobre var1 s'ha fet sobre la referència a la variable global que s'ha passat a la funció, mentre que var2 s'ha definit de nou amb var dins la funció, sent, per tant, una referència diferent a la var2 definida globalment.
- I finalment, veiem com totes les variables que s'han definit dins la funció, independentment de si s'han fet amb var o let, no tenen visibilitat fora de la funció.

La recomanació és sempre fer ús de la paraula reservada `let`, i definir les variables sempre de forma local a aquells àmbits on es necessiten. Quan necessitem passar variables a una funció, ho farem sempre pel pas d'arguments.

4. Programació estructurada en Javascript

4.1. Estructures condicionals

4.1.1. if

Es tracta de la condició més senzilla:

```
if (condició) {
    // bloc de codi a executar si la condició s'avalua a cert
}
```

Exemple 1

```
if (hora < 18) {
    salutacio = "Bon dia";
}
```

4.1.2. if / else

```
if (condició) {
    // bloc de codi a executar si la condició s'avalua a cert
} else {
    // bloc de codi a executar si la condició s'avalua a fals
}
```

Exemple

```
if (hora < 18) {
    salutacio = "Bon dia";
} else {
    salutacio = "Bona vesprada";
}
```

4.1.3. else if Quan poden donar-se vàries condicions alternatives, podem fer ús de l'estructura else/if.

```
if (condició1) {
    // bloc de codi a executar si la condició 1 s'avalua a cert
} else if (condició2) {
    // bloc de codi a executar si la condició 2 s'avalua a cert
} else {
    // bloc de codi a executar si cap de les dos condicions s'avalúa a
       cert.
}
```

Exemple

```
if (hora < 12) {
    salutacio = "Bon dia";
} else if (hora < 22) {
    salutacio = "Bona vesprada";
} else {
    salutacio = "Bona nit";
}
```

4.1.4. Switch S'utilitza quan volem realitzar diferents accions en funció de múltiples condicions sobre una expressió. Segons aquesta s'avalúe, s'executarà un (o més, compte amb el break!) d'entre diversos blocs.

```
switch(expressió) {
    case n1:
        // Bloc de codi a executar si expressió s'avalúa a n1
        break;
    case n2:
        // Bloc de codi a executar si expressió s'avalúa a n2
        break;
    default:
        // Bloc de codi a executar si l'expressió s'avalúa
       // a un valor no contemplat
}
```

Exemple: Fent ús de la funció `Date().getDay()` que torna el dia de la setmana:

```
switch (new Date().getDay()) {
    case 0:
        dia = "Diumenge";
        break;
    case 1:
        dia = "Dilluns";
        break;
    case 2:
        dia = "Dimarts";
        break;
    case 3:
        dia = "Dimecres";
        break;
    case 4:
        dia = "Dijous";
        break;
    case 5:
        dia = "Divendres";
        break;
    case 6:
        dia = "Dissabte";
        break;
    default:
       console.log("dia erroni");    }
```

El break s'utilitza per indicar que ja no cal seguir buscant més coincidències. Estalviem així comparacions innecessàries i errors. Cal tindre en compte que també es poden agrupar diversos case (sense el break pel mig), quan hi ha codi comú.

Exemple:

```
switch (new Date().getDay()) {
```

```
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        dia = "dia laboral";
        break;
    case 0: // Diumenge
    case 6: // Dissabte
        dia = "cap de setmana";
        break;
    default:
       console.log("dia erroni");
}
```

4.2. Estructures repetitives

4.2.1. Bucle for S'usa generalment quan sabem d'avantmà quantes repeticions necessitem:

```
for (inicialització; condició_de_repetició ; increment ) {
    // Codi a executar
}
```

Exemple:

```
for (i = 0; i < 10; i++) {
    console.log("Index: " + i);  // Fixeu-se que "sumem" un string i un
        enter!
}
```

Sintàcticament la inicialitació, condició de repetició i l'increment no són necessàries, i el bucle anterior podria expressar-se com a:

```
var i=0;
for(;;){
   console.log(i++);
    if (i>10) break;   // El break trenca el flux del bucle
}
```

Per altra banda, també podem realitzar inicialitzar diversos valors:

****Exemple****: Per mostrar els elements d'un vector:

```
vector=["pera", "poma", "plàtan"];
for (i = 0, len = vector.length, text = ""; i < len; i++) {
    console.log(vector[i]);
}
```

4.2.2. Bucle for/in - for/of S'utilitza per recórrer els elements d'un objecte:

```
for(item in object){
    ...
}
```

Exemple:

```
let peli={title:"Han Solo: Una historia de Star Wars", director: "Ron
   Howard"};

for (index in peli) {
   console.log(index+":"+peli[index]);
}
```

Així també podem indexar els elements d'un vector:

```
 v=[1,2,3,76,4,2,5]
[ 1,
  2,
  3,
  76,
  4,
  2,
  5 ]
> for (i in v) {
... console.log(v[i]);}
```

Compte: Un error habitual (per influència d'altres llenguatges d'script) és confondre l'índex amb el valor de l'element:

```
for (i in v) {
    console.log(i);
}  // Quan el que volem és accedir al contingut!
```

Per tal de fer açò, el que ens ofereix Javascript és l'operador *of*. L'únic requeriment és que l'objecte sobre el que l'apliquem siga ***iterable***.

Per exemple, el següent codi que imprimeix els elements del vetor fent ús de **for..in**:

```
> for (i in v) {console.log(v[i])};
```

Pot expressar-se més senzillament mitjançant **for..of**:

```
> for (i of v) {console.log(i)};
```

4.2.3. Bucles While i do/while Són equivalents al bucle for, però amb una sintaxi diferent, on hem de realitzar tant la inicialització com l'increment fora de la sintaxi de la instrucció.

While

```
inicialitzaió;
while(condició){
    // accions;
    //increment;
};
```

Do - While

```
// [inicialització;]
do{
    //accions;
    //[inicialitació]

} while (condició)
```

Exemples:

```
let i=1;
while(i<10){
     console.log(i);
     i++;
     }
```

```
let i=1;
do{
    console.log(i);
    i++;
    } while (i<10);
}
```

4.2.4. Break i continue El break, com hem vist, serveix per eixir d'un bucle (i també d'un case!). Per la seua banda, l'ordre **`continue`**, serveix per passar a la següent iteració del bucle.

Exemple

```
let i=0;
for(;;){
    i++;
    if(i%2) continue; // A qué s'avalua i%2 ?
    console.log(i);
    if (i>=10) break;
}
```

5. Programació Orientada a Objectes en JS

En el paradigma d'orientació a objectes, el programari es dissenya a través d'objectes que cooperen, intercanviant-se missatges i responent a events, a diferència del punt de vista tradicional on es considera com un conjunt de funcions.

Vegem alguns conceptes:

- **Classe**: Defineix les característiques de l'objecte.
- **Objecte**: Una instància de la classe.
- **Propietat/Atribut**: Una característica (variable) de l'objecte.
- **Mètode**: Una capacitat (funció) de l'objecte.
- **Constructor**: Mètode que s'invoca en crear l'objecte.
- **Herència**: Capacitat d'una classe d'heretar característiques d'altra classe.
- **Encapsulament**: Les propietats i els mètodes dels objectes es troben organitzats de manera estructurada, evitant l'accés des de qualsevol forma distinta a les especificades.
- **Polimorfisme**: Mecanisme pel qual diferents classes poden implementar de forma diferent un mateix mètode.

5.1. Orientació a Objectes basada en prototipus

Es tracta d'un estil de programació orientada a objectes on estos no es creen a través de la instanciació de classes, sinó mitjançant la clonació d'altres objectes que fan de prototipus. També es coneix com programació sense classes, orientada a prototipus o basada en exemples.

Espais de noms En aplicacions grans, solem utilitzar els espais de noms per tal d'evitar conflictes de noms amb entre els diferents scripts i llibreries que les componen. Aquests actúen com a contenidors que permeten associar la funcionalitat i les propietats d'un objecte amb un nom únic i aïllat de la resta. (Podría equipararse als packages de Java)

En javascript no existeix una sintaxi nativa per a eixa funcionalitat, sinò que s'utilitza un objecte per a esta finalitat. Per exemple:

```
1 var app={}
2 app.variable1 = 'variable1'
3 app.variable2='variable2'
4 app.funcio = function () { .... }
```

La definició d'un objecte prototip en javascript no és més que una funció, que fa al mateix temps de constructor:

```
function Objecte() {
    // Constructor i inicialització de valors
    }
```

I per crear un nou objecte basat en aquest prototip, fem ús de new:

```
var obj=new Objecte();
```

Quan es defineix així un objecte, aquest té accés a una propietat especial, anomenada prototipus (prototype), que permet accedir a la classe en sí. Per accedir a propietats dins una classe s'utilitza la paraula reservada **this**. Caldrà anar amb compte amb aquesta, ja que quan tenim objectes o propietats anidades, **this** sempre farà referència a l'objecte del nivell on es troba. Per accedir des de fora de la classe a un atribut o mètode fem ús de la notació punt (.): `Objecte.propietat` / `Objecte.mètode`. Per exemple, per definir la propietat *nom* d'una classe persona, podem fer-ho en la creació de la instància:

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

I per instanciar dos noves persones:

```
let persona1 = new Persona("Paco");
let persona2 = new Persona("Pepe");

console.log ('persona1 es ' + persona1.nom); // mostra "persona1 es
    Paco"
console.log ('persona2 es ' + persona2.nom); // mostra "persona2 es
    Pepe"
```

Per tal d'incloure mètodes, el que fem és definir funcions a dins la classe:

```
function Persona(nom) {
  this.nom = nom;
  this.saluda=function(){
        alert('Hola, em dic '+this.nom);
    }
}

var persona1=new Persona('Paco');
persona1.saluda();
```

Com veiem, els mètodes no són més que funcions que s'associen com una propietat a l'objecte, de manera que poden invocar-se fora del seu context.

Exemple

Vegem un exemple complet per a la definició de classes, i que ens aprofitarà per vore altres conceptes

com el de l'herència:

```
/* Exemple de definició de classes */

function punt(x, y){
    /*
         Funció constructor de la classe punt
       Rep dos paràmetres amb els que inicialitza
       els atributs x i y
     */

    this.x=x;  // Definim atributs amb la paraula
    this.y=y;  // reservada "this"

    this.get=function(){
        // Els métodes no són més que funcions
        // Associades a la classe, i que definim
        // amb this, igual que un atribut.
        return "("+this.x+","+this.y+")";
    }
}

function figura(color, position){
    /*
        Classe figura, té com atributs un color
        i una posició (objecte de tipus punt)
    */

    this.color=color;     // Inicialitzem el color
    this.position=position; // Inicialitzem la posició

    this.draw=function(){
        /*
              Métode que mostra un misstge donant
          informació de la figura
        */
        console.log("Dibuixant figura en posició "+this.position.get()+
                    " i color "+this.color);
    }
}

function rectangle(color, posicio, costat1, costat2){

    /*
    Classe rectangle, que és una especialització de
        la classe figura. De figura hereta:
        - l'atribut posició
        - l'attribut color
    A més:
        - Inclou dos nous atributs: costat1 i costat2
        - Redefineix el métode draw
```

```
50          - Defineix un nou mètode area
51      */
52
53      this.costat1=costat1;
54      this.costat2=costat2;
55      /*
56      Per heretar propietats de figura,
57      fem ús de "call", passant this com a
58      primer paràmetre, i els valors amb qué
59      volem invocar al constructor de la
60      superclasse
61      */
62
63      figura.call(this, color, posicio);
64
65      this.draw=function(){
66          console.log("Dibuixant Rectangle de "+
67                   this.costat1+"x"+this.costat2+
68                   " en posició "+this.position.get()+
69                   " i color "+this.color);
70          };
71
72      this.area=function(){
73          return this.costat1*this.costat2;
74          }
75      }
76
77
78  function cercle(color,posicio, radi){
79      /*
80      Classe cercle, hereta de figura: posició i color
81      Afig l'atribut radi
82      I redefineix el mètode draw i defineix area
83      */
84
85      this.radi=radi;
86      // Apply és pot utilitzar també per heretar de la superclasse
87      // La diferéncia és que requereix el pas de paràmetres a través
88      // d'un vector:
89      figura.apply(this, [color, posicio]);
90
91      this.draw=function(){
92          console.log("Dibuixant Cercle de radi "+this.radi+
93                         " en posició "+this.position.get()+
94                         " i color "+this.color);
95          };
96
97      this.area=function(){
98          return 2*this.radi*Math.PI;
99          }
100     }
```

```
101
102 p1=new punt(10,20);
103 p2=new punt(10,30);
104 p3=new punt(30,20);
105
106 f1=new figura("roig", p1);
107 f2=new rectangle("verd", p2, 40, 20);
108 f3=new cercle("blau", p3, 50)
109 f1.draw();
110 f2.draw();
111 f3.draw();
```

Exemple 2

La forma anterior de definir els objectes prototipus pot resultar confusa quan es tracta d'objectes que van a requerir de moltes funcions. Una pràctica habitual és fer ús de l'element `prototype` per tal d'accedir al prototipus d'una funció i modificar-lo. Vegem-ho més clar amb el mateix exemple expressat d'una altra manera:

```
1  /* Exemple de definició de classes */
2
3  function punt(x, y){
4      /*
5      Classe(Objecte) punt, amb els atributs x i y.
6          */
7      this.x=x;
8      this.y=y;
9  }
10
11 punt.prototype.get=function(){
12     /*
13     Accedim al prototipus de l'objecte a través de prototype
14     I creem una nova propietat "get" que és una funció
15     */
16     return "("+this.x+","+this.y+")";
17 }
18
19 function figura(color, position){
20     this.color=color;
21     this.position=position;
22 }
23
24 figura.prototype.draw=function draw(){
25      // Podem especificar també el nom de la funció després del
            function
26     console.log("Dibuixant figura en posició "+this.position.get()+
27                 " i color "+this.color);
28 }
29
30 function rectangle(color, posicio, costat1, costat2){
```

```
31      this.costat1=costat1;
32      this.costat2=costat2;
33      figura.call(this, color, posicio);
34  }
35
36  rectangle.prototype.draw=function(){
37          console.log("Dibuixant Rectangle de "+this.costat1+
38                          "x"+this.costat2+" en posició "+
39                      this.position.get()+" i color "+this.color);
40          };
41
42  rectangle.prototype.area=function(){
43      return this.costat1*this.costat2;
44  }
45
46  function cercle(color,posicio, radi){
47      this.radi=radi;
48      figura.apply(this, [color, posicio]);
49  }
50
51  cercle.prototype.draw=function(){
52      console.log("Dibuixant Cercle de radi "+this.radi+
53                  " en posició "+this.position.get()+
54                  " i color "+this.color);
55          };
56
57  cercle.prototype.area=function(){
58      return 2*this.radi*Math.PI;
59  }
60
61  p1=new punt(10,20);
62  p2=new punt(10,30);
63  p3=new punt(30,20);
64
65
66  f1=new figura("roig", p1);
67  f2=new rectangle("verd", p2, 40, 20);
68  f3=new cercle("blau", p3, 50)
69  f1.draw();
70  f2.draw();
71  f3.draw();
```

5.2. Orientació a Objectes a partir de ES6

A partir de la versió EcmaScript 6, Javascript suporta una sintaxi per a l'orientació a objectes més propera a la de la resta dels llenguatges. Cal remarcar que no es tracta d'una reormulació del model d'objectes, sinò que simplement es dedica a proveïr de definicions més clares i simples per al treball amb objectes.

Declaració de classes Com en altres llenguatges, es pot declarar utilitzant la paraula class:

```
class Classe{

}

let objecte = new Classe;
```

Cal tindre en compte alguns detalls:

- La classe no necessita d'arguments per a la seua definició, pel que no s'acompanya dels parèntesis(`new Classe` en lloc de `new Classe()`).
- Com que s'ha definit com a classe, el sistema no permetrà que s'execute com a funció (a diferència dels prototipus), sinò que es reservarà com un constructor.
- El contingut de la classe s'executa en mode «strict» de manera automàtica.
- Les declaracions de classe no segueixen les regles del *hoisting* (la declaració pot ser posterior a l'ús). Açò significa que les classes només existeixen després de ser declarades.
- Una classe es comporta implícitament com una constant, pel que no es pot redeclarar més avant en un mateix àmbit.

Vegem la declaració de classe de forma més ampliada:

```
class  Classe{
    constructor (arguments){
        // super (arguments);
        // Declaració d'atributs amb this
    }

    get getAtribut(){
        // getter; compte que el nom no siga el
        // mateix que l'atribut

        return (this.atribut);
    }
    set setAtribut(valor){
        // setter; compte que el nom no siga el
        // mateix que l'atribut
        this.atribut=valor;
    }

    static staticMethod(){
        // Aquest és un métode estàtic,
        // S'exectarà des de la classe,
        // no des de les instàncies d'aquesta.
    }

    Metode(){
        // Métode públic
```

```
27          // No existeixen per tant els privats o protegits
28      }
29  }
```

Veiem un altre exemple més complet:

```
1  class Classe{
2  constructor(valor=1){
3      // Atributs de la classe (_a)
4      // Si no s'especifica, el valor per defecte és 1
5      this._a=valor;
6  }
7
8  get a(){
9    // getter (no s'ha de dir igual que l'atribut!)
10   return this._a;
11  }
12
13 }
14 // Instanciació sense especificar valor
15 b=new Classe;
16 console.log(b.a);
17
18 // Instanciació amb valor
19 c=new Classe(4);
20 console.log(c.a);
```

El resultat de l'script serà:

```
1  1
2  4
```

Herència L'herència s'aconsegueix mitjançant la parala clau `extends`:

```
1  class Sublclasse extends Superclasse{
2      constructor(params={}){
3          // Invocació al constructor de la classe pare
4          super(valors);
5      }
6
7      metode(){
8          // Invocació a métodes de la classe pare
9          super.metode();
10     }
11 }
```

Com podem veure, amb la paraula reservada `super` podem accedir a les propietats i mètodes del pare. Si la utilitzem en forma de funció (`super(valors)`) invoquem al constructor de la classe pare,

amb els valors per defecte que li passem. Per la seua banda, quan la utilitzem en forma d'objecte (`super.metode()`), el que fem és invocar a un mètode de la classe pare.

Alguns aspectes a destacar sobre l'herència en Javascript:

- El constructor d'una classe filla, necessàriament ha d'invocar a **`super`** abans de fer ús de **`this`**.
- Els constructors de les subclasses, han d'invocar a **`super`**, per inicialitzar la classe, o bé retornar un objecte que reemplace l'objecte que no ha estat inicialitzat.

Veiem finalment com quedaría l'exemple anterior amb aquesta nova sintaxi:

```
1  /* Exemple de definició de classes */
2
3  class punt{
4      constructor(x, y){
5          // Cconstructor de la classe punt
6         // Rep dos paràmetres amb els que inicialitza
7         // els atributs x i y
8          this.x=x;
9          this.y=y;
10     }
11
12     get Posicio(){
13         return "("+this.x+","+this.y+")";
14     }
15 }
16
17 class figura{
18     constructor(color, position){
19         this.color=color;     // Inicialitzem el color
20         this.position=position; // Inicialitzem la posició
21     }
22
23     Dibuixa(){
24         console.log("Dibuixant figura en posició "+this.position.
               Posicio+
25                     " i color "+this.color);
26     }
27 }
28
29 class rectangle extends figura {
30     constructor (color, posicio, costat1, costat2){
31         // Hereta de figura la posició i el color
32         // Inclou nous atributs: costat1 i costat2
33
34         super(color, posicio);
35         this.costat1=costat1;
36         this.costat2=costat2;
37     }
38
```

```
39      Dibuixa(){
40      console.log("Dibuixant Rectangle de "+
41                  this.costat1+"x"+this.costat2+
42                  " en posició "+this.position.Posicio+
43                  " i color "+this.color);
44      }
45
46      area(){
47          return this.costat1*this.costat2;
48          }
49  }
50
51  class cercle extends figura{
52      constructor (color,posicio, radi){
53      super(color, posicio);
54      this.radi=radi;
55      }
56
57
58      Dibuixa(){
59          console.log("Dibuixant Cercle de radi "+this.radi+
60                          " en posició "+this.position.Posicio+
61                          " i color "+this.color);
62          }
63
64      area(area){
65          return 2*this.radi*Math.PI;
66
67      }
68  }
69
70  let p1=new punt(10,20);
71  let p2=new punt(10,30);
72  let p3=new punt(30,20);
73
74  let f1=new figura("roig", p1);
75  let f2=new rectangle("verd", p2, 40, 20);
76  let f3=new cercle("blau", p3, 50)
77  f1.Dibuixa();
78  f2.Dibuixa();
79  f3.Dibuixa();
```