JESUÏTES educació

DAWM06UF2

ESTRUCTURES DEFINIDES PEL PROGRAMADOR

OBJECTES

UF2.3 ESTRUCTURES DEFINIDES PEL PROGRAMADOR. OBJECTES

CFGS Desenvolupament d'Aplicacions Web

M06. Desenvolupament web en entorn client

Fundació Jesuïtes Educació - Escola del Clot

Sergi Grau sergi.grau@fje.edu

OBJECTIUS

- Aprendre el funcionament de la POO amb JS, i les seves característiques basades en prototipus

- Aplicar les millores que aporten les darreres versions d'ES en el treball amb POO

INTRODUCCIÓ

- El tipus fonamental de javascript es l'objecte. es un tipus agregat i ens permet encapsular dades en forma de propietats (matriu associativa o diccionari, son sinonims), cadascuna de les quals te un nom i un valor.

- Utilitza un mapatge cadena-valor, de la mateixa manera que ho fan les taules de dispersió.

- JavaScript també pot heretar propietats d'un altre objecte, anomenat prototipus. Els mètodes d'un objecte, habitualment són propietats heretades.

- JavaScript és un llenguatge dinàmic, i això ens permet afegir i esborrar propietats en temps d'execució, encara que podem simular el compartament de llenguatges estàtics com ara Java o C++

- Qualsevol valor en JavaScript que no sigui string, nombre, true/false, null o undefined és un objecte, i per tant són mutables i manipulats per referència

DIFERÈNCIES EN OO BASADA EN CLASSES I BASADA EN PROTOTIPUS

- hi ha una diferencia fonamental entre els sistemes d'objectes classics i els basats en prototips. els objectes classics es defineixen abstractament com a part d'un grup conceptual i hereten caracteristiques d'altres classes, o grups d'objectes.

- per contra, els objectes prototipics es defineixen concretament com objectes especifics i hereten el comportament d'altres objectes especifics.

+ Per tant, **un llenguatge basat en classes OO té un caràcter dual que requereix almenys dos construccions fonamentals: classes i objectes.** Com a resultat d'aquesta dualitat, es creen complexes jerarquies de classes. En general és impossible predir totes les les classes que es necessiten per a ser utilitzades en el futur, de manera que la jerarquia de classes necessita ser constantment refactoritzada per facilitar els canvis.

+ **En el llenguatges basats en prototips s'ha eliminat la necessitat de la dualitat esmentada** anteriorment i faciliten la creació directa i la manipulació d'objectes. Sense objectes obligats per una classe, els sistemes s'ajuda a mantenir la modularitat i reduir la necessitat de refactorització.

PROPIETATS

- **Una propietat té nom i valor.** El nom pot ser qualsevol cadena, incloent la cadena buida, però no pot tenir dos propietats amb el mateix nom.

- El valor pot ser qualsevol valor de JavaScript incloent funcions accessores (getters i setters).

- Cada propietats té associats atributs de propietat (EC5):

- a) l'atribut d'escriptura, que diu quan es pot escriure una propietat.

- b) l'atribut d'enumeració que ens indica quan una propietat es retornada en un bucle for/in

- c) l'atribut de configuració, que ens diu quan una propietat pot ser esborrada, o modificada.

ATRIBUTS D'OBJECTE

- **A més de les propietats els objectes disposen d'un conjunt d'atributs d'objecte.**

- a) **un objecte prototipus** és una referència a un altre objecte les propietats del qual s'hereten.

- b) **un objecte classe és una cadena que categoritza el tipus d'un objecte**.

- c) unes **flags** que indiquen quines noves propietats es poden afegir a un objecte.

- Cal distingir entre objectes natius (Array, funcions, Date, etc..), objectes hostes (com ara el client-side) , objectes definits per l'usuari, propietats propies i propietats heretades..

CREACIÓ D'UN OBJECTE

- Es poden crear de 3 formes:
- a) **com a literals**, la forma mes senzilla amb una llista separada per comes amb parelles nom:valor, el nom és un identificador o cadena (o cadena buida), i el valor una expressió valida.
- b) **amb l'operador new,** es crea i incialitza un objecte cridant a un mètode construtor (una funció en JavaScript)
- c) amb la funció **Object.create**()

CREACIÓ D'OBJECTES COM A LITERALS

JESUÏTES educació
DAWM06UF2 sergi.grau@fje.edu

JS

```
//els literals creen i inicialitzen objectes

var buida = {};
var punt = { x:0, y:0 }; // dos propietats
var punt2 = {x:punt.x, y:punt.y };
var assignatures = {“entorn client”:”UF 1”, “programació II”: {UF1: “Introducció
a POO”}
};

//operador new
var o = new Object(); //objecte buit
var a = new Array(); // objecte array buit
var d = new Date(); objecte Date amb la data actual
var e = new RegExp(“js”); //crea una expressió regular
```

- **Cada objecte de JavaScript té un segon objecte (o null però rarament) associat amb ell. S'anomena prototipus** i el primer objecte hereta totes les propietats del prototipus.

- Tots els objectes creats com a literals tenen el mateix prototipus, i podem accedir mitjançant Object.prototype.

- Els objectes creats amb l'operador new, utilitzen el valor de la propietat prototype de la funció constructor, per tant new Date(), utilitza Date.prototype, i new Array(), Array.prototype.

- Com aquest hereten el prototipus d'Object, de retruc s'hereten també les propietats d'Object.prototype, formant una cadena de prototipus.

OBJECT.CREATE

- Object.create() és un mètode estàtic, i no cal cap objecte per a invocar-la.

- El primer paràmetre passat defineix el prototipus.

```
var o1 = Object.create({x:1, y:2}); // o1 hereta les
propietats x i y
var o2 = Object.create(null); // si no volem heretar de
ningú
var o3 = Object.create(Object.prototype); // o3 com {} o
new Object()
```

OBJECT.CREATE

JS

```
var punt = {
    x : 0,
    y : 0
};

punt.x // => 0
punt.y // => 0

// crea un nou objecte on punt és el prototipus i hereta tot el seu
comportament.

punt3D = Object.create(punt);
punt3D.z = 0;

punt3D.x // => 0
punt3D.y // => 0
punt3D.z // => 0
```

OBJECT.CREATE

JS

```
//heretem d'un objecte arbitrari. Retorna un nou objecte.
// funciona amb  EC5 i EC3

function hereta(p) {
    if (p == null) throw TypeError(); // p no pot ser null
    if (Object.create) // Si Object.create() es definit...
      return Object.create(p);

    var t = typeof p; // en cas contrari (no suportem EC5
    if (t !== "object" && t !== "function") throw TypeError();
    function f() {}; // constructor fictici
    f.prototype = p; // assignem prototipus a p
    return new f();
}
```

CONSTRUCTORS

JS

```
function Punt(x, y) {
    this.x = x;
    this.y = y;
}

var p = new Punt(3, 4);
p.x // => 3
p.y // => 4

Punt.prototype.r = function() {
    return Math.sqrt((this.x * this.x) + (this.y * this.y));
};
```

PATRÓ D'HERÈNCIA PSEUDOCLÀSSIC

JESUÏTES educació
DAWM06UF2 sergi.grau@fje.edu

JS

```
//equivalent literal

var punt = {
    x: 1,
    y: 2,
    r: function () {
        return Math.sqrt((this.x * this.x) + (this.y * this.y));
    }
};


var punt3D = Object.create(punt);
punt3D.z = 3;
punt3D.r = function() {
    return Math.sqrt((this.x * this.x) + (this.y * this.y) + (this.z *
this.z));
};
```

CONSULTANT I ASSIGNANT PROPIETATS

- Per obtenir o escriure el valor d'una propietat podem utilitzar la notació .
  O []

```
var autor = llibre.autor;
var nom = autor.nom;
var titol = llibre["titol principal"];
```

```
Llibre.edicio=6;
llibre[“autor principal”] = “Sergi Grau”;
```

- **Recordeu que les propietats dels objectes en JavaScript es comporten com a matrius associatives, i es poden crear en temps d'execució.**

- **Els objectes de JavaScript hereten un conjunt de propietats de l'objecte prototipus que prenen com a base**.

- En el cas de sol·licitar una propietat d'un objecte, si aquest no la disposa es busca en el seu prototipus, i així fins a trobar algun objecte de la jerarquia que no té prototipus.

- Si assignem un valor a un objecte d'una propietat que no disposa, es crea, i en cas d'existir en el seu prototipus aquest queda ocult.

QÜESTIONS SOBRE L'ACCÉS A PROPIETATS

- La consulta de propietats que no existeixen no és un error. Es cerca a la cadena de prototipus i si no existeix retorna un undefined.

- Tanmateix sí és un error consultar un objecte que no existeix, o si la propietat pròpia o heretada és només de lectura

- Var long = llibre.subtitol.length ; // llença TypeError, doncs la propietat no existeix, o llibre no existeix

- Cal fer

```
var long = undefined;
if (llibre) {
 if (llibre.subtitol) long = llibre.subtitol.length;
}

0

var long = llibre && llibre.subtitol &&
llibre.subtitol.length;
```

ESBORRANT PROPIETATS

+ Utilitzem l'operador **delete** per esborrar una propietat, no el contingut. Esborra només les propietats pròpies no les heretades (caldria utilitzar prototype). S'avalua a true si s'ha pogut esborrar correctament. Les propietats que tenen l'atribut configurable com a false no es poden esborrar.

+ Cal fer

```
delete llibre.autor;
delete llibre[“titol principal”];

o={x:1};
delete o.x; // retorna true

delete Object.prototype; //no es pot esborrar, no és
configurable
var x=1;
delete this.x; //idem
```

VERIFICANT PROPIETATS

- Per a verificar si un objecte té una propietat utilitzem **hasOwnProperty**() i **propertyIsEnumerable**() (pròpia i enumerable), i l'operador **in**

```
var o={x:1};
“x” in o; //retorna true
“y” in o; //retorna false
o.x !== undefined; // true
```

```
o.hasOwnProperty(“x”); //retorna true
o.hasOwnProperty(“y”); //retorna false
o.hasOwnProperty(“toString”); //retorna false

var o = hereta({ y: 2 });
o.x = 1;
o.propertyIsEnumerable("x"); // true
o.propertyIsEnumerable("y"); // false
Object.prototype.propertyIsEnumerable("toString"); //
false
```

ENUMERANT PROPIETATS

- Per iterar per totes les propietats d'un objecte utilitzem el bucle for/in

- EC5 disposa de **Object.keys**() que retorna els noms de totes les propietats enumerables i de **Object.getOwnProperty**() que retorna les propietats pròpies (enumerables o no)

```
var o = {x:1, y:2, z:3};
o.propertyIsEnumerable("toString"); //false
for(p in o) console.log(p);

/* funció d'utilitat per heretar (copiar totes les
propietats)
function extend(o, p) {
 for(prop in p) {
  o[prop] = p[prop];
 }
    return o;
}
```

GETTERS I SETTERS

- Una propietat és un nom, un valor i un conjunt d'atributs.

- En EC5 un atribut pot ser modificat per dos mètodes , getter i setter. Són els coneguts com a accessors de propietats.

- Si una propietat té getter i setter és llegible i es pot escriure.

```
var o = {data_prop: valor,
get accessor_prop() { ... },
set accessor_prop(value) {... }
};
```

GETTERS I SETTERS

JS

```
//coordenades polars d'un punt

var p = {
    x: 1.0,
    Y:1.0,
    get r() { return Math.sqrt(this.x*this.x + this.y*this.y); },
    set r(nou) {
         var vell = Math.sqrt(this.x*this.x + this.y*this.y);
         var ratio = nou/vell;
         this.x *= ratio;
         this.y *= ratio;
        },
        get theta() { return Math.atan2(this.y, this.x); }
};

//creem un objecte que hereta els getters i setters, amb les pròpies propiuetats

var q= hereta(p);
q.x=2; q.y=3; console.log(q.r);
```

ATRIBUTS DE LES PROPIETATS

JS

```
var o = {}; // objecte sense propietats
     // afegim una propietat no enumerable amb valor 1
     Object.defineProperty(o, "x", { value : 1,
         writable: true,
         enumerable: false,
         configurable: true});

// verfiquem
o.x; // => 1
Object.keys(o) // => []

// Modifiquem la propietat per a que sigui només de lectura
Object.defineProperty(o, "x", { writable: false });
```

ATRIBUTS DE LES PROPIETATS

JS

```
var p = Object.defineProperties({}, {
    x: { value: 1, writable: true, enumerable:true, configurable:true },
    y: { value: 1, writable: true, enumerable:true, configurable:true },
    r: {
        get: function() { return Math.sqrt(this.x*this.x +
        this.y*this.y) },enumerable:true, configurable:true
    }
});
```

ATRIBUTS DELS OBJECTES
ATRIBUT PROTOTIPUS

- Cada objecte té associats els atributs prototipus, classe i extensible
- L'atribut **prototype** especifica l'objecte des del qual heretem propietats.
- S'assigna quan un objecte es creat.
- Podem consultar el prototipus de qualsevol objecte amb Object.getPrototypeOf()
- Per saber si un objecte és prototipus d'un altre disposem del mètode isPrototypeOf()

```
var p = {x:1};
var o = Object.create(p);
p.isPrototypeOf(o) ; //true
Object.prototype.isPrototypeOf(o); //true
```

- L'atribut classe és una cadena que subministra informació sobre el tipus d'objecte. El mètode **toString**() heretat d'Object.prototype retorna [object class]

```
function classeDe(o) {
     if (o === null) return "Null";
     if (o === undefined) return "Undefined";
     return Object.prototype.toString.call(o).slice(8,-1);
}
```

ATRIBUTS DELS OBJECTES
ATRIBUT PROTOTIPUS

- L'atribut **extensible** especifica quan noves propietats poden ser afegides a un objecte i quan no. Per a fer-ho utilitzen Object.isExtensible() i Object.preventExtensions(objecte). No es possible modificar aquest atribut una vegada s'ha definit.

- La seva finalitat és evitar que objectes que el prenguin com a prototipus puguun afegir noves propietats.

- Object.seal(objecte), a més de fer que un objecte no sigui extensible, fa que les seves propietats siguin no configurables, és a dir les propietats no es podran esborrar ni configurar.

- Object.freeze(objecte), a més del que fa seal(), fa que les propietats són només de lectura.

```
var o= Object.create(Object.freeze({x:1}),{y: {value: 2,
writable: true}});
```

SERIALITZACIÓ D'OBJECTES

- La serialització és el procés de salvar un objecte a un medi d'emmagatzematge (com pot ser un fitxer, o un buffer de memòria) amb la finalitat de transmetre'l a través d'una connexió en xarxa com una sèrie de bytes o en un format humanament més llegible com XML. La sèrie de bytes o el format es poden utilitzar per recrear un objecte que és idèntic en el seu estat intern a l'objecte original (de fet un clon). Aquest tipus de serialització s'utilitza principalment per transportar un objecte a través d'una xarxa, per persistir objectes a un fitxer o base de dades, o per a distribuir objectes idèntics a unes quantes aplicacions o localitzacions.

- Aquest procés de serialitzar un objecte també s'anomena desinflar (deflating en anglès) un objecte o ordenar (marshalling en anglès) un objecte.

- L'operació oposada, que extreu una estructura de dades d'una sèrie de bytes, és la deserialització (que s'anomena també inflating o unmarshalling).

- http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats

SERIALITZACIÓ D'OBJECTES

- La serialització en JavaScript és un procés que converteix l'estat d'un objecte a una cadena, a partir de la qual després pot ser restaurat. EC5 proveeix de funcions natives **JSON.stringify**() i **JSON.parse**() per a serialitzar i deserialitzar objectes. Aquestes funcions utilitzen la notació d'intercanvi de dades JSON, la qual s'assembla molt a les matrius literals de JavaScript.

```
o = {x:1, y:{z:[false,null,""]}}; // creem un objecte
s = JSON.stringify(o); // s es '{"x":1,"y":{"z":
[false,null,""]}}' p = JSON.parse(s); // p és una deep
copy de s
```

MÈTODES D'OBJECTES, TOSTRING()

- Com hem vist Object té definits els següents mètodes:
  - `HasOwnProperty()`
  - `PropertyIsEnumerable(),`
  - `IsPrototypeOf()`
  - `Create()`
  - `GetPrototypeOf()`

- Si ho desitgem podem sobreescriure els mètodes toString(), per defecte no ofereix molta informació.

  - `var s = { x:1, y:1 }.toString(); // [object Object]`

- Moltes classes redefineixen el seu comportament, per exemple Array retorna una llista dels elements.

- LocaleString(), és mètode la finalitat del qual és mostrar una cadena localitzada, per defecte crida a toString()

MÈTODES D'OBJECTES, TOJSON(), VALUEOF()

- El mètode toJSON(), es cercat pel mètode estàtic JSON.stringify(), i s'utilitza per definir com es fa la serialització.

- El mètode **valueOf**(), s'assembla a toString(), però es crida quan JavaScript necessita convertir un tipus primitiu, normalment un nombre, a cadena

COMPARACIÓ JAVA VS JAVASCRIPT

JS

```
class FootballPlayer {
    private string name;
    private string team;

    static void FootballPlayer() { }

    string getName() {
        return this.name;
    }

    string getTeam() {
        return this.team;
    }

    void setName(string val) {
        this.name = val;
    }

    void setTeam(string val) {
        this.team = val;
    }
}
```

JS

```
class RunningBack extends FootballPlayer {
    private bool offensiveTeam = true;

    bool isOffesiveTeam() {
        return this.offensiveTeam;
    }
}

RunningBack emmitt = new RunningBack();
RunningBack lt   = new RunningBack();
```

COMPARACIÓ JAVA VS JAVASCRIPT

JS

```
var footballPlayer = {
    name : "";
    team : "";
};

var runningBack = Object.create(footballPlayer);
runningBack.offensiveTeam = true;

var lineBacker = Object.create(footballPlayer);
lineBacker.defensiveTeam = true;

footballPlayer.run = function () { this.running = true };
lineBacker.run();
lineBacker.running; // => true
```

CLASSES

- Les classes de javascript, introduïdes en ECMAScript 2015, són una millora sintàctica sobre l'herència basada en prototips de JavaScript.

- La sintaxi de les classes no introdueix un nou model d'herència orientada a objectes en JavaScript. Les classes de JavaScript proveeixen una sintaxi molt més clara i simple per crear objectes i bregar amb l'herència.

DECLARACIÓ DE CLASSES

+ Una manera de definir una classe és mitjançant una declaració de classe. Per declarar una classe, s'utilitza la paraula reservada class i un nom per a la classe “Rectangle”.

+ En primer lloc necessites declarar la teva classe i després accedir-hi, d'una altra manera donarà un ReferenceError. No té Hoisting (allotjament)

```
class Rectangle {
  constructor(alt, ample) {
    this.alt = alt;
    this.ample = ample;
  }
}
```

EXPRESSIONS DE CLASSE

JS

```
// Anonima
let Rectangle = class {
    constructor(alt, ample) {
      this.alt = alt;
      this.ample = ample;
    }
  };

console.log(Rectangle.name);
// output: "Rectangle"

// Nombrada
let Rectangle = class Rectangle {
  constructor(alt, ample) {
      this.alt = alt;
      this.ample = ample;
  }
};
console.log(Rectangle.name);
// output: "Rectangulo2"
```

CONSTRUCTOR

- El mètode constructor és un mètode especial per crear i inicialitzar un objecte creat amb una classe. Només hi pot haver un mètode especial amb el nom "constructor" en una classe. Si aquesta conté més d'una ocurrència de l'mètode constructor, es donarà un Error SyntaxError

- Un constructor pot fer servir la paraula reservada super per cridar a constructor d'una superclasse

```
class Rectangle {
  constructor(alt, ample) {
    this.alt = alt;
    this.ample = ample;
  }
}
```

JS

```
class Rectangle {
    constructor(alt, ample) {
        this.alt = alt;
        this.ample = ample;
      }

    // Getter
    get area() {
       return this.calcArea();
     }
    // Mètode
    calcArea () {
      return this.alto * this.ancho;
    }
}

const quadrat = new Rectangle(10, 10);

console.log(quadrat.area); // 100
```

MÈTODES ESTÀTICS

JS

```
/*
La paraula clau static defineix un mètode estàtic per a una classe.
Els mètodes estàtics són anomenats sense instanciar la seva classe i no poden ser cridats
mitjançant una instància de classe.
Els mètodes estàtics són sovint usats per crear funcions d'utilitat per a una aplicació.
*/

class Punt {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distancia(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx * dx + dy * dy);
    }
}

const p1 = new Punt(5, 5);
const p2 = new Punt(10, 10);

console.log(Punt.distancia(p1, p2)); // 7.0710678118654755
```

JS

```
class Animal {
    constructor(nom) {
      this.nom = nom;
    }

    parlar() {
      console.log(this.nom + ' arrrr.');
    }
  }

  class Gos extends Animal {
    parlar() {
      console.log(this.nom + ' bub bub.');
    }
  }
```

HERÈNCIA DE PROTOTIPUS

JS

```
function Animal (nom) {
    this.nom = nom;
  }
  Animal.prototype.parkar = function () {
    console.log(this.nom + 'arrr.');
  }

  class Gos extends Animal {
    hablar() {
      super.hablar();
      console.log(this.nom + ' buf buf.');
    }
  }

  var p = new Gos('Mitzie');
  p.hablar();
```

MIX-INS

+ Subclasses abstractes or mix-ins són plantilles de classes. Una classe ECMAScript només pot tenir una classe pare, amb la qual cosa l'herència múltiple no és possible. La funcionalitat ha de ser proporcionada per la classe pare.

+ Una funció amb una classe pare com a entrada i una subclasse extenent la classe pare com a sortida pot ser usat per implementar mix-ins en EMCAScript:

```
var calculatorMixin = Base => class extends Base {
  calc() { }
};

var randomizerMixin = Base => class extends Base {
  randomize() { }
};

class Foo { }
class Bar extends calculatorMixin(randomizerMixin(Foo))
{ }
```

MÒDULS EN ES6

- Els programes JavaScript començar sent bastant petits - la major part del seu ús en els primers dies era per realitzar tasques de scripting aïllades, proporcionant una mica d'interactivitat a les teves pàgines web on fos necessari, pel que generalment no es necessitaven grans scripts. Avancem uns anys i ara tenim aplicacions completes que s'executen en navegadors amb molt JavaScript, estigui habilitat ara s'usa en altres contextos (NODE.JS, per exemple).

- NODE.JS ha tingut aquesta capacitat durant molt de temps, i hi ha una sèrie de biblioteques i marcs de JavaScript que permeten l'ús de mòduls amb CommonJS (require)

- La bona notícia és que els navegadors moderns han començat a admetre la funcionalitat dels mòduls de forma nativa.

- https://caniuse.com/?search=import

MJS VS JS

- Es poden fer servir extensions .js per als nostres arxius de mòdul, però en altres recursos, pots veure que en el seu lloc s'usa l'extensió .mjs. La documentació de V8 recomana això. Les raons donades són:

- És bo per claredat, és a dir, deixa clar quins arxius són mòduls i quines JavaScript.

- Assegura que els teus arxius de mòdul siguin analitzats com un mòdul pels entorns d'execució com NODE.JS i eines de compilació com Babel.

- Has d'assegurar que el teu servidor els estigui servint amb una capçalera Content-Type que contingui un tipus MIME de JavaScript com a text / javascript, utilitzar mjs por donar problemes segons el servidor web

EXPORTAR CARACTERÍSTIQUES DEL MÒDUL

index.html

```
<!DOCTYPE html>
<html lang="ca">
  <head>
    <meta charset="utf-8">
    <title>Exemple de mòduls</title>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
    <script type="module" src="main.js"></script>
  </head>
  <body>

  </body>
</html>
```

EXPORTAR CARACTERÍSTIQUES DEL MÒDUL

moduls/canvas.js

```
function crear(id, parent, width, height) {
    let divWrapper = document.createElement('div');
    let canvasElem = document.createElement('canvas');
    parent.appendChild(divWrapper);
    divWrapper.appendChild(canvasElem);

    divWrapper.id = id;
    canvasElem.width = width;
    canvasElem.height = height;

    let ctx = canvasElem.getContext('2d');

    return {
      ctx: ctx,
      id: id
    };
}

function crearInforme(wrapperId) {
    let list = document.createElement('ul');
    list.id = wrapperId + '-reporter';

    let canvasWrapper = document.getElementById(wrapperId);
    canvasWrapper.appendChild(list);

    return list.id;
}

export { crear as crear, crearInforme as crearInforme };
```

EXPORTAR CARACTERÍSTIQUES DEL MÒDUL

moduls/quadrat.js

```
  listItem.textContent = `${nom} perimetre es ${length * 4}px.`

  let list = document.getElementById(listId);
  list.appendChild(listItem);
}

function dibuixarQuadrat(ctx) {
  let color1 = aleatori(0, 255);
  let color2 = aleatori(0, 255);
  let color3 = aleatori(0, 255);
  let color = `rgb(${color1},${color2},${color3})`
  ctx.fillStyle = color;

  let x = aleatori(0, 480);
  let y = aleatori(0, 320);
  let length = aleatori(10, 100);
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color
  };
}

export { nom as name, dibuixar as dibuixar, obtenirArea as obtenirArea, obtenirPerimetre as obtenirPerimetre };
export default dibuixarQuadrat;
```

EXPORTAR CARACTERÍSTIQUES DEL MÒDUL

moduls/quadrat.js

```
const nom = 'quadrat';

function dibuixar(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color
  };
}

function aleatori(min, max) {
   let num = Math.floor(Math.random() * (max - min)) + min;
   return num;
}

function obtenirArea(length, listId) {
  let listItem = document.createElement('li');
  listItem.textContent = `${nom} area es ${length * length}px quadrades.`

  let list = document.getElementById(listId);
  list.appendChild(listItem);
}

function obtenirPerimetre(length, listId) {
  let listItem = document.createElement('li');
```

EXPORTACIONS PREDETERMINADES

- La funcionalitat que hem exportat fins ara es compon d'exportacions amb nom - cada element (ja sigui una funció, const, etc.) s'ha denominat pel seu nom en export, i aquest nom també s'ha utilitzat per referir-se a ell en import.

- També hi ha un tipus d'exportació anomenat exportació per defecte - està dissenyat per facilitar que un mòdul proporcioni una funció predeterminada, i també ajuda als mòduls JavaScript a interoperar amb els sistemes de mòduls CommonJS

- Tingues en compte la falta de claus. Això es deu al fet que només es permet una exportació per defecte per mòdul

- export default quadrat o export default function(ctx) { ...}

- import quadrat from './modules/square.js';

EVITAR CONFLICTE DE NOMS

- Dins de les claus del teu instruccions import i export, pots fer servir la paraula clau es juntament amb un nou nom de funció, per canviar el nom d'identificació que utilitzarà una funció dins de la lliçó de nivell superior.

```
// module.js
export {
  function1 as newFunctionName,
  function2 as anotherNewFunctionName
};

// main.js
import {newFunctionName, anotherNewFunctionName} from
‘./modules/module.js'

// ALTERNATIVA module.js
export {function1, function2};

// ALTERNATIVA de main.js
import {function1 as newFunctionName,
        function2 as anotherNewFunctionName } from './
modules/module.js';
```

CREAR UN MODUL

- El mètode anterior funciona bé, però és una mica complicat i llarg. Una solució encara millor és importar les característiques de cada mòdul dins d'un objecte Modul

```
import * as Modul from './modules/module.js';

Modul.function1()
Modul.function2()
etc.
```

CARREGA ESTÀTICA VS DINÀMICA

+ La càrrega estàtica és la que hem estudiat fins ara.

+ La càrrega dinàmica et permet carregar mòduls dinàmicament només quan són necessaris, en lloc d'haver de carregar tot per avançat. Això té alguns òbvies avantatges de rendiment; segueix llegint i vegem com funciona.

```
import('./modules/myModule.js')
  .then((module) => {
    // fer quelcom
  });

b.addEventListener('click', () => {
  import('./moduls/quadrat.js').then((Modul) => {
    let q1 = new Modul.Quadrat(canvas.ctx,
canvas.listId, 50, 50, 100, 'blue');
    q1.draw();
    q1.reportArea();
    q1.reportPerimeter();
  })});
```

BIBLIOGRAFIA

+ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

+ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#objects_and_properties

+ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes