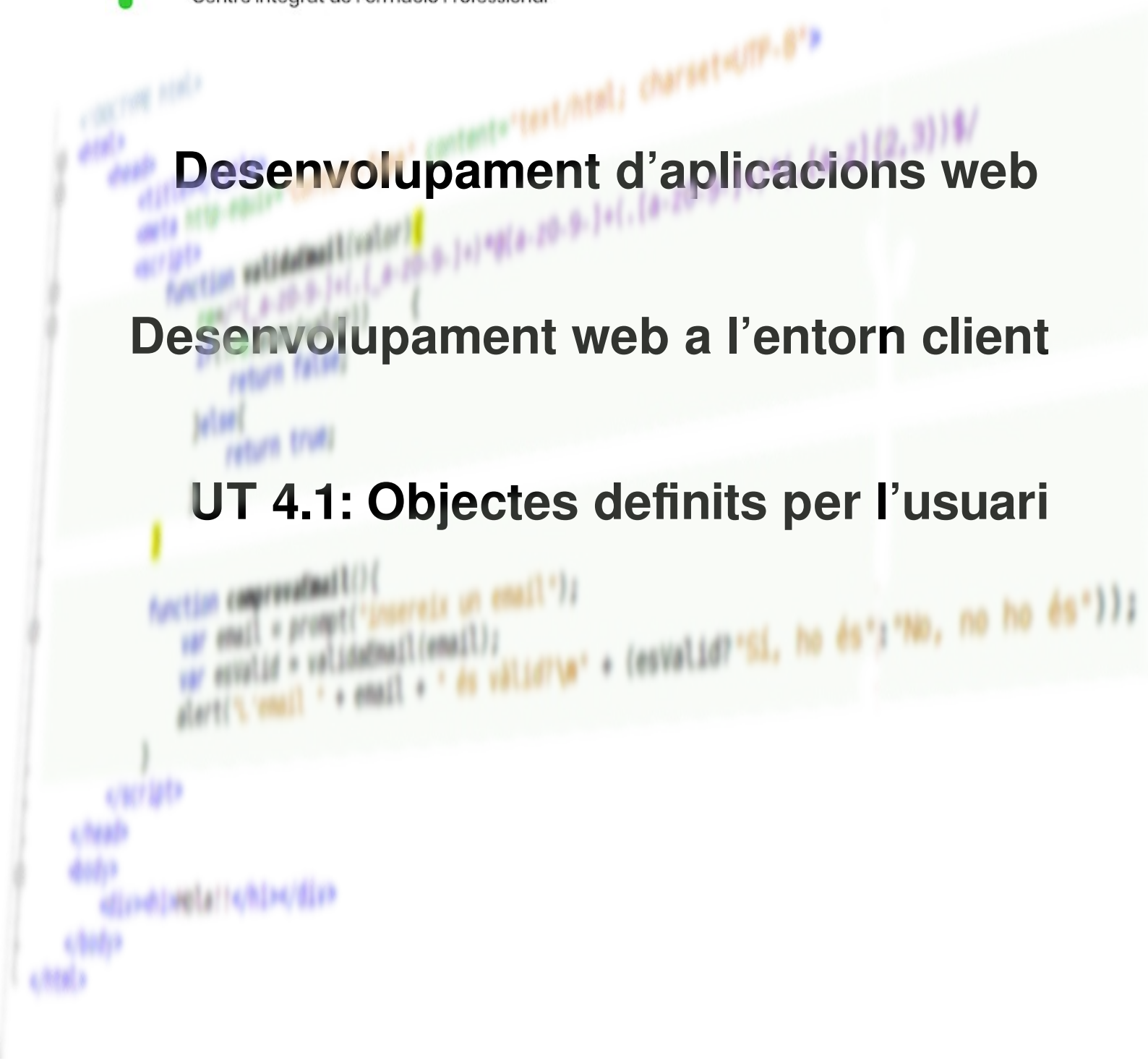
PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions web

Desenvolupament web a l'entorn client

UT 4.1: Objectes definits per l'usuari



Joan Pons Tugores

Índex de continguts

Llenguatge basat en objectes / orientat a objectes

Com ja hem dit en alguna ocasió, Javascript va començar com un llenguatge basat en objectes. Això vol dir que tenia el concepte d'objecte però no el de classe ni per descomptat el d'herència.

Fa uns anys s'afegiren aquests conceptes al llenguatge, encara que no s'imposà la necessitat que un objecte pertanyi a una classe.

D'aquesta manera conviuen les dues formes de programar:

- Podem crear directament objectes, especificant per a cada un les seves propietats i els seus mètodes.

  Útil si sempre cream els objectes al mateix punt, per exemple, per recollir les dades d'un formulari per enviar-les al servidor en XML o JSON.

- Podem crear classes per facilitar la tasca de crear objectes amb les mateixes característiques.

  Útils quan hem de crear objectes des de diversos llocs del codi, els objectes són molt complexos, ...

Creació literal d'objectes

Javascript no ens obliga a crear una classe per instanciar objectes a partir d'ella, podem crear els objectes directament.

Aquesta tècnica és sol utilitzar per enviar dades entre el client i el servidor, normalment transformant-les a un format anomenat JSON que veurem més endavant.

Els objectes es declaren amb les { } //Claus, no claudàtors.

Podem aprofitar per inicialitzar-los al moment:

```
let jo={
  nom: "Jo",
  llinatges: "Mateix",
  mostraNom: function(){
          return this.nom+" "+this.llinatges;
  }
}
```

El nom de les propietats va sense cometes i separada del valor per : Les propietats es separen amb comes ,

this fa referència a l'objecte.

Per a cada objecte hem de repetir tot el codi amb els valors de les propietats adequats.

Les *arrow functions* no s'han d'utilitzar com a mètodes, ja que no tenen l'enllaç, el *binding*, a *this*. El següent codi torna *undefined undefined*.

```
let jo = {      nom: "Jo",      llinatges: "Mateix",
    mostraNom: () => this.nom + " " + this.llinatges
}
```

També podem crear l'objecte buit i afegir propietats i mètodes amb la notació per punts o amb la d'array.

```
let jo = {};
 jo.nom = "Jo";
 jo["llinatges"] = "Mateix";
 jo.mostraNom = function () {
     return this.nom + " " + this.llinatges;
 };
 console.log(jo.mostraNom());
```

Cada propietat té un nom i un valor. Només van entre cometes els valors alfanumèrics.

Ara *jo* té dues "propietats" i un mètode.

Així podem afegir a *jo* totes les propietats i mètodes que ens facin falta. La única pega és que només les estam afegint a aquest objecte, no a cap classe. Per tant si volguéssim un altre objecte igual, li hauríem d'afegir una altra vegada totes aquestes propietats i mètodes.

Com ja hem dit, és molt habitual crear objectes d'aquesta manera per intercanviar informació amb el servidor. En aquest cas, no tendran mètodes, només propietats.

```
let jo={};
jo.nom="Jo";
jo.llinatges="Mateix";
```

Funció constructor

Fa temps que Javascript incorpora la paraula reservada ***new***. Té dues funcions:

- Crea un objecte nou
- Fa que l'objecte acabat de crear executi la funció que segueixi la paraula *new*.

Aquesta funció "constructor" no té res especial que la defineixi com a constructor, senzillament que l'executam en fer el new, més ben dit, **que l'objecte que acaba de crear el new l'executa**.

La paraula reservada ***this*** és pot utilitzar dins una funció **per fer referència al propietari de la funció, a qui l'executa**. Per això la podem utilitzar dins la funció constructor per afegir propietats a l'objecte acabat de crear:

```
function Persona(nom, llinatges){
        this.nom=nom;
        this.llinatges=llinatges;
}
var jo=new Persona("Jo", "Mateix");
```

Cada objecte que creem tendrà la mateixa estructura inicialitzada amb les dades que passem a la funció.

No es poden utilitzar arrow functions com a funció constructora. El següent codi és incorrecte:

```
const Persona = (nom, llinatges) => {
        this.nom=nom;
        this.llinatges=llinatges;
}
var jo=new Persona("Jo", "Mateix"); //Error
```

Dins la funció constructor també es poden afegir mètodes que tendran tots els objectes creats amb la funció:

```
function Persona(nom, llinatges){
        this.nom = nom;
        this.llinatges = llinatges;
        this.nomComplet = function(){
                return nom+ "  " + llinatges;
        }
}
var jo=new Persona("Jo", "Mateix");
console.log(jo.nomComplet()); // Jo Mateix
```

La pega de fer-ho així és l'eficiència. Per a cada objecte que es crea s'ha de crear una funció anònima i assignar-la a *nomComplet*.

Això pot millorar afegint els mètodes al prototipus de la funció constructor.

Prototipus

Tots els objectes creats tenen una propietat inaccessible anomenada *[[prototype]]*. Per altra banda, totes les funcions tenen una propietat accessible anomenada *prototype*.

La propietat *prototype* de les funcions es pot modificar com un objecte qualsevol, afegint-hi nous elements, propietats i funcions.

En crear un nou objecte amb *new* a la propietat *[[prototype]]* del nou objecte se li assigna un enllaç a la propietat *prototype* de la funció.

Que vol dir tot això? Que **qualsevol objecte creat amb *new* i la funció constructor tendrà totes les propietats i funcions que s'hagin afegit al *prototype* d'aquest constructor.**

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

Persona.prototype.getNomComplet = function () {
    return this.nom+' '+this.llinatges;
};
const jo = new Persona('Jo', 'Mateix');
console.log(jo.getNomComplet()); // Jo Mateix
```

Tots els objectes creats amb

```
new Persona()
```

tendran les propietats *nom* i *llinatges* i una funció *getNomComplet()*

La funció *getNomComplet()* només es crearà una sola vegada.

Class

Ecmascript 2015, també conegut com ES6, va introduir les classes com una manera mes clara de crear els objectes que el prototype.

Per defecte les classes utilitzen el *mode estricte*.

Declaració de classes

Utilitzam la paraula *class* seguida del nom de la classe i el codi entre claus.

```
class Rectangle{
  ...
}
```

O també

```
let Rectangle=class{
  ...
}
```

De qualsevol de les dues maneres tendrem una classe anomenada Rectangle. El codi entre les claus de la classe s'anomena cos de la classe i conté la definició de les propietats i dels mètodes i el constructor.

Constructor

Javascript només permet un constructor per classe, lògic ja que no suporta sobrecàrrega de funcions. El nom del constructor ha de ser literalment *constructor*. La seva tasca és inicialitzar els objectes.

Javascript l'executa automàticament en crear un objecte.

```
class Rectangle{
        constructor(alt, ample){
                ,,,
        }
}
```

Atributs o propietats d'instància

Són qualsevol variable que es declari utilitzant *this* dins qualsevol mètode de la classe, encara que **la norma és que es declarin dins el constructor**.

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

Mètodes

Per declarar un mètode no s'ha d'utilitzar la paraula *function*, hem utilitzar directament el nom del mètode:

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

	area(){
		return this.alçada * this.amplada;
	}

	perimetre(){
		return (this.alçada + this.amplada)*2;
	}
}
```

Per accedir als atributs dins dels mètodes hem d'utilitzar *this*.

Els mètodes s'han d'executar sempre a través d'un objecte.

```
let r=new Rectangle(10,20);
console.log(r.area());
```

No podem utilitzar *arrow functions* per definir els mètodes, a més de *this* tampoc reben cap referència a *super*.

Mètodes static

De vegades podem tenir mètode que relacionats amb una classe, però que no pertanyen a cap objecte. Es solen anomenar mètodes d'utilitat. Per exemple un mètode que compari les àrees de dos rectangles.

```
class Rectangle{
...
//Torna 0 si són iguals, positius si primer > segon i negatiu si primer < segon
static comparaArea(primer, segon){
        return primer.area() - segon.area();
  }
}
```

Per accedir als mètodes *static* ho hem de fer a través de la classe:

```
let a = new Rectangle(10, 20);
let b = new Rectangle(10, 30);

console.log(Rectangle.comparaArea(a, b));
```

Atributs de classe o propietats static

Podem definir propietats *static*, propietats que pertanyen a la classe i no a cap objecte concret.

En aquest cas, s'han d'afegir a la classe fora de la declaració:

```
class Rectangle{
...
}
Rectangle.staticWidth=3;
```

Per accedir a les propietats *static* ho hem de fer a través de la classe. Els objectes no la tenen definida:

```
console.log(Rectangle.staticWidth); //mostra 3
let b = new Rectangle(10, 30);
console.log(b.staticWidth); // mostra undefined
```

Getter i setter

Els atributs i els mètodes definits a les classes Javascript són públics. Per tant hi tenim accés directament i no ens fa falta definir mètodes específics per accedir o modificar el contingut d'un atribut.

Així i tot Javascript permet crear tant getters com setters. Es solen utilitzar per donar accés a propietats calculades. No totes les propietats tenen perquè ser un atribut de l'objecte, poden ser el resultat d'algun càlcul.

Per exemple, Rectangle pot tenir el getter d'una propietat, dimensions, que s'obté de concatenar l'amplada i l'alçada de l'objecte.

També pot tenir el setter d'aquesta propietat, que rep una cadena amb el format que torna el getter i la descompon per assignar els valors als atributs.

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

    area(){
        return this.alçada * this.amplada;
    }

    perimetre(){
        return (this.alçada + this.amplada)*2;
    }
```

```
    get dimensions(){
        return alçada+"x"+amplada;
    }

    set dimensions(dimensions){
        let valors=dimensions.split("x");
        this.alçada=parseInt(valors[0]);
        this.amplada=parseInt(valors[1]);
    }
}
```

Herència amb extends

Podem definir una classe a partir d'una altra amb *extends*. La classe filla hereta tot el de la classe mare, inclosos els membres de classe, i pot afegir i sobreescriure mètodes de la classe mare.

```
class Animal {
    constructor(nom) {
        this.nom = nom;
    }
    parla() {
        console.log(this.nom + ' fa renou.');
    }
}

class Dog extends Animal {
    constructor(nom, xip){
        super(nom);
        this.xip=xip;
    }
    mostraXip(){
        console.log(xip);
    }
    parla() {
        super.parla();
        console.log(this.nom + ' lladra.');
    }
}
```

Si la classe filla necessita un constructor, el primer que ha de fer és cridar al constructor de la classe mare amb *super*. Amb *super* també es pot cridar els mètodes sobreescrits de la classe mare.

Classes i *arrow functions*

Recordem que les *arrow functions* tenen unes quantes limitacions, algunes de les quals afecten a les classes.

- No reben la referència a *this*.
- No reben cap referència a *super*.

Això impedeix que siguin mètodes d'una classe i per tant, tampoc poden ser constructors.

Operador *instanceof*

Si volem saber si una variable conté un objecte creat amb una determinada funció constructor o amb una determinada classe o amb qualsevol subclasse d'una classe determinada. Com en Java.

```
let jo=new Persona('Jo Mateix');
...
if ( jo instanceof Persona ) { ... }
```

En aquest cas donarà true ja que Persona és la funció constructora o la classe amb la que s'ha creat l'objecte. El següent cas també tornarà true:

```
let ell=new Alumne('Un Altre'); //Alumne extends Persona
...
if ( ell instanceof Persona ) { ... }
```

En canvi aquest donaria false:

```
let ell=new Alumne('Un Altre'); //Alumne extends Persona
...
if ( ell instanceof String ){ ... }
```