Session 6. JavaScript Objects & JSON

Programació Multiplataforma i Distribuïda

Grau d'informàtica. EPSEVG

Octubre 2020

Jordi Esteve jesteve@cs.upc.edu

JavaScript Objects. Definition

The **objects** allow the aggregation of **properties** (variables) and **methods** (functions). The objects are classified in **classes**.

e.g. An object of the circle class could have the center, radius and color properties and the area and perimeter methods.

Predefined object classes: **Object**, **Array**, **Date**, **null**

null is a special object to represent the empty object.

We can define literal objects with curly brackets {}:

```
let circle = {
  xCenter: 2.5,
  "yCenter": -4.1,
  radius: 2,
  color: "red",
  perimeter: function() {return 2 * Math.PI * this.radius;} // this is a reference to self object
  "area": function() {return Math.PI * this.radius * this.radius;}
};
```

JavaScript Objects. Properties and methods

The properties and methods names of an object:

- must all be different.
- must follow the same rules as variable names (they can contain letters, digits, $ or _ characters except they can not start with a digit).

We can access to properties and methods using:

- **point operator**: object.property, object.method()
- **array operator**: object["property"], object["method"]()

```
// Access to properties
circle.radius;      // Gets 2
circle["radius"];  // Gets 2
var name = "radius";
circle[name];      // Gets 2
// Access to methods
circle.perimeter();      // Gets 12.566370614359172
circle["perimeter"](); // Gets 12.566370614359172
```

JavaScript Objects. Add, delete and check properties and methods

Any string for property/method names can be used if we access with the array operator. Anyway, it is better to use variable name rules to be able to use point operator.

Properties/methods can be **added** and **deleted** dynamically and can be **checked** if they are defined. If we try to access to a not defined property, the undefined value is returned.

```
let student = {"First name": "Joan", "Last-name": "Bertran"};
student["First name"];     // Gets 'Joan'
student["Last-name"];      // Gets 'Bertran'
student.birth_date;        // Gets undefined
student.birth_date = new Date("1992-04-24"); // Adds new property or changes its value
student.birth_date;        // Gets 1992-04-24T00:00:00.000Z
student.age = function() {return (new Date() - this.birth_date) / (365*24*60*60*1000);}
student.age()              // Gets the student has 27.4737… years
"birth_date" in student;   // Gets true
delete student.birth_date; // Deletes a property/method if exists
"birth_date" in student;   // Gets false
```

JavaScript Objects. ES6 shorthands

A string expressions can be written inside array operator to define a property/method.

When we create an object from variables and the name of the properties and the name of the variables containing the initial values are the same, there is no need to repeat the name twice. For example, {a: a, b: b} can be simplified as {a, b}.

The **function** reserved word can be omitted when a method is defined.

```
let concept = "name";
let student = {
  ["first_"+concept]: "Joan",
  ["last_"+concept]: "Bertran",
  full_name() {return this["first_"+concept] + " " + this["last_"+concept];}
};
student.full_name();       // Gets 'Joan Bertran'
let first_name = "Esther", last_name = "Molins";
let student2 = {first_name, last_name} // {first_name: first_name, last_name: last_name}
student2.first_name;      // Gets 'Esther'
```

JavaScript Objects. Iteration throw properties and methods

Several tools allow to iterate throw all the properties and methods of an object:

- for (let k in obj) {…} // k contains the name of each property/method
- Object.keys(obj) // Returns an array with the name of each property/method
- Object.values(obj) // Returns an array with the values of each property/method
- Object.entries(obj) // Returns an array with the [name, value] of each property/meth.

```
for (let k in student) {console.log(k, ":",  student[k]);}
Object.keys(student).forEach(k => console.log(k, ":",  student[k]));
Object.entries(student).forEach(([k,v]) => console.log(k, ":",  v));

/* The 3 commands return the same result:
first_name : Joan
last_name : Bertran
full_name : full_name() {return this["first_"+concept]+" "+this["last_"+concept];} */
```

JavaScript Objects. Multiple assignment. Rest/Spread operator …

Like in arrays, multiple or destructuring assignment can be done with objects:

```
var a, b, rest; // Notice the use of parenthesis in multi assignment if objects are not declared

// Multi assignment
[a, b] = [10, 20];                  // a gets 10, b gets 20
({ a, b, c} = { b: 20, a: 10 }); // a gets 10, b gets 20, c is undefined

// Multi assignment with rest operator (the ... operator is on the left)
[a, b, ...rest] = [10, 20, 30, 40, 50];              //  a gets 10, b gets 20, rest gets [30, 40, 50]
({a, b, ...rest} = {b: 20, c: 30, d: 40, a: 10}); // a gets 10, b gets 20, rest gets {c: 30, d: 40}

// Default values
[a = 5, b = 7] = [1];                // a gets 1, b gets 7
({a = 5, b = 7, c} = {a: 1}); // a gets 1, b gets 7, c is undefined

// Default values and assign to new variable names
({a:x = 5, b:y = 7, c:z} = {a: 1}); // x gets 1, y gets 7, z is undefined

// Spread operator (the ... operator is on the right)
let obj = {a: 2, b: 4};
let obj2 = {c: 3, d: 5, ...obj} // obj2 contains { c: 3, d: 5, a: 2, b: 4 }
```

JavaScript Objects. Task exercise (I)

We want to manage a list of tasks to know which ones are done and which ones are not. A task has a title describing it and a mark to annotate if it is done or not.

An array can be used to store the list of tasks. And a object with the properties title (string) and done (boolean) to store each of the tasks.

```
[ {
   title: "PMUD HTML exercise",
   done: true
 },
 {
   title: "PMUD CSS exercise",
   done: false
 },
 {
   title: "PMUD JavaScript exercise",
   done: false
 } ]
```

JavaScript Objects. Task exercise (II)

CRUD (Create, Read, Update, Delete) functions must be created to manage the task list: Read tasks, create a new task, update or delete a specific task, …

Download from Atenea the task_model.js file that contains CRUD functions to manage a task lists stored in an array of objects. Complete the deletes(id) function (it is similar to update). Load the file in Node.js and test all the CRUD methods:

```
> .load task_model.js
> count(); // Returns 3
> getAll();
[ { title: 'PMUD HTML exercise', done: true, id: 0 },
  { title: 'PMUD CSS exercise', done: false, id: 1 },
  { title: 'PMUD JavaScript exercise', done: false, id: 2 } ]
> get(0); // Returns { title: 'PMUD HTML exercise', done: true, id: 0 }
> update(1, 'PMUD CSS3', true);
> create('PMUD jQuery', false);
> deletes(2);
> getAll();
[ { title: 'PMUD HTML exercise', done: true, id: 0 },
  { title: 'PMUD CSS3', done: true, id: 1 },
  { title: 'PMUD jQuery', done: false, id: 2 } ]
> reset(); // Does not work properly
```

JavaScript. JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

JSON allows data serialization: Reversible conversion from data values to string

JSON makes storage and sending data easy: After converting the data values to a string it can be saved in a file or it can be sent to other computer.

JSON is not a specific JavaScript tool, it is becoming the most used data serialization format. There are JSON libraries for the must used programming languages.

In JavaScript:

- JSON.stringify(object); // Transforms the JavaScript object into a string
- JSON.parse(string); // Transforms the string into a JavaScript object

JavaScript. JSON

JSON can serialize: **string**, **number**, **boolean**, **array**, **object**, **null**

(NaN, Infinity and -Infinity special numbers are converted to null)

JSON can not serialize: function, undefined, date (dates are converted to a string)

```
> JSON.stringify(1);
'1'
> JSON.stringify("Hi");
'"Hi"'
> JSON.stringify([1, "Hi", true]);
'[1,"Hi",true]'
> let s = JSON.stringify({title: "PMUD", done: false, date: new Date()});
> s;
'{"title":"PMUD","done":false,"date":"2019-10-09T15:21:56.301Z"}'
> JSON.parse(s);
{ title: 'PMUD', done: false, date: '2019-10-09T15:21:56.301Z' }
```

JavaScript. Creating a copy of arrays/objects through JSON

The JavaScript arrays and objects are assigned by reference, so changes in the values of the new variable also changes the values of the original.

We can create real copies by: copy = JSON.parse(JSON.stringify(original));

```
> let e=[1,2,3];
> let f = e;
> f[0] = 4;
> e
[ 4, 2, 3 ] // Original array has been changed, because e and f are the same array

> e=[1,2,3];
> f = JSON.parse(JSON.stringify(e));
> f[0] = 4;
> e
[ 1, 2, 3 ] // Original array is not changed, now e and f are different arrays
```

Improve the code in task_model.js to get reset() function fixed. Make real copies between initial_tasks and tasks arrays. Also make a real copy of task returned by get().

JavaScript. OOP without classes: Constructor and new operator

JavaScript supports OOP (Object Oriented Programming) without the class concept (a set of objects with common properties and methods).

JavaScript simulates the class concept using functions and prototypes.

The **constructor** is a function that creates new objects when is called with **new**.

```
function circle(r, x=0, y=0, c="white") {
  this.xCenter = x,
  this.yCenter = y,
  this.radius = r,
  this.color = c,
  this.perimeter = function() {return 2 * Math.PI * this.radius;}
  this.area = function() {return Math.PI * this.radius * this.radius;}
};
let c1 = new circle(2) , c2 = new circle(3, 1, 1, "red");
c1.perimeter();  // Returns 12.566370614359172
c2.perimeter();  // Returns 18.84955592153876
c2.border ="blue" ; // We can define new properties/methods outside the constructor: UGLY
```

JavaScript. Common properties and methods = prototype

In previous example each circle has its own copy of xCenter, yCenter, radius and color (Ok! They are specific for each circle). But also each circle has its own copy of perimeter and area functions (Wrong! These functions never change).

We can define **class level** properties and methods (they are common for all the objects) with **prototype**:

```
function circle(r, x=0, y=0, c="white") {
  this.xCenter = x,
  this.yCenter = y,
  this.radius = r,
  this.color = c,
  circle.prototype.perimeter = function() {return 2 * Math.PI * this.radius;}
  circle.prototype.area = function() {return Math.PI * this.radius * this.radius;}
};
let c1 = new circle(2) , c2 = new circle(3, 1, 1, "red");
c1.perimeter();  // Returns 12.566370614359172
c2.perimeter();  // Returns 18.84955592153876
```

JavaScript. Predefined objects

Some JavaScript predefined objects:

- **Object**: Basic object to store properties and methods. Everyone inherits from it.
- **Array**: A list of values that can be accessed by an index.
- **Number**, **String**, **Boolean**: Objects to store a number, a string or a boolean.
- **Date**: Object that stores day, month, year, hour, minute, second, …
- **Error**: Stores error information that can be thrown by JavaScript interpreter.
- **Function**: Defines a block of code with parameters.

```
> new Object();
{}
> new Array(3);
[ <3 empty items> ]
> new Date();
2019-10-09T16:31:07.382Z
> new Function();
[Function: anonymous]
```

Exercise: Constructor of task model

Move all previous code for manage tasks (task_model.js) inside a constructor named TaskModel(). So we will be able to manage different task lists.

initial_tasks and **tasks** will be properties. **count**, **getAll**, **get**, **create**, **update**, **deletes** and **reset** will be common methods for all the objects.

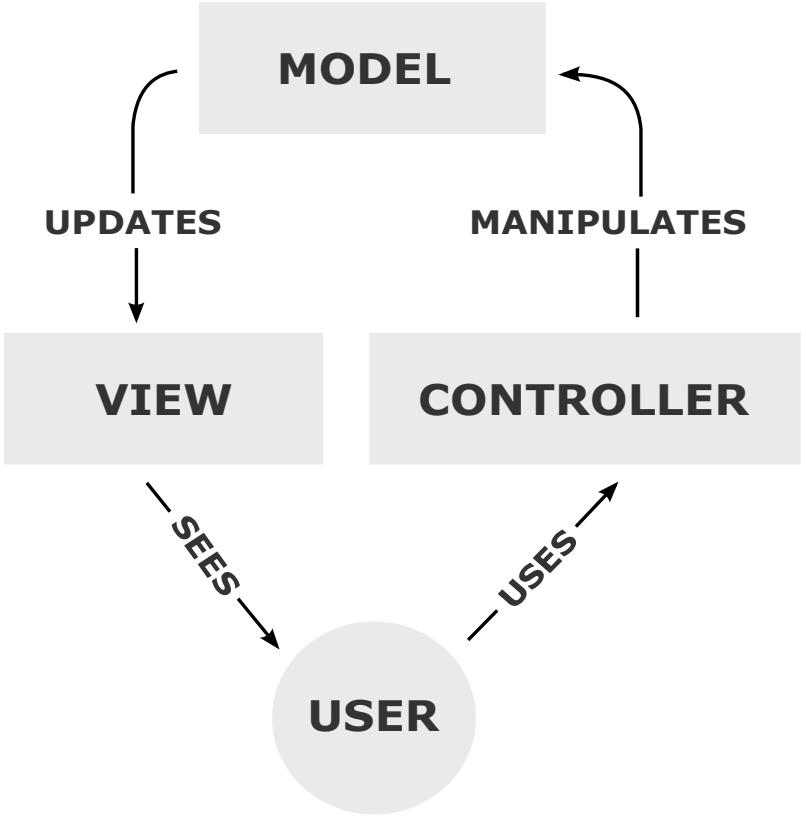
Test in Node.js the creation of a pair of task models, for example one for store work tasks and other for store home tasks. Add, update, delete and get tasks to the 2 models:

```
let tw = new TaskModel(); // Tasks for work
let th = new TaskModel(); // Tasks for home
tw.create("Meeting");
th.create("Cook dinner", true);
th.deletes(0);
th.deletes(0);
th.deletes(0);
tw.getAll();
// [ { title: 'PMUD HTML exercise', done: true, id: 0 },
//   { title: 'PMUD CSS exercise', done: false, id: 1 },
//   { title: 'PMUD JavaScript exercise', done: false, id: 2 },
//   { title: 'Meeting', done: false, id: 3 } ]
th.getAll();
// [ { title: 'Cook dinner', done: true, id: 0 } ]
```

MVC pattern (Model-View-Controller)

Model–View–Controller (usually known as MVC) is a software architectural pattern commonly used for developing user interfaces which divides the related program logic into three interconnected elements:

- **Model**: It is the application's dynamic data structure, independent of the user interface. It directly manages the data and logic of the application.
- **View**: Any representation of information such as a chart, diagram or table.
- **Controller**: Accepts input and converts it to commands for the model or view.



MVC exercise: Task list

Download the *task_VC.zip* file from Atena that contains a basic HTML web page and a script (*task_view_controller.js*) with the **views** and **controllers** for CRUD operations to manage tasks. The **model** is defined inside the class `TaskModel` in the previous *task_model.js* file.

Unzip the file and add the previous *task_model.js* file in the same folder. Load *task.html* in a web browser. Test the creation (New task) and the edition of tasks.

Fill the missing code in the `switchController`, `deleteController` and `resetController` functions. Then test the deletion, reset and switch work fine.

Analyze how the code is separated in views and controllers, obtaining a clean and tidy up code easy to understand and to maintain. Observe there is a special controller function at the end that routes each event towards the right controller.

Add a button "All tasks"/"Active tasks" that lists all tasks or only the not done ones.

Add a "Search" text field that allows filter the tasks containing this text.