

### Readings Assignment Notes

#### 1. Ch8: Forms

- ✓ **Web forms** are an essential topic, since submitting and processing a form is the only way a web client can make a request to a server-side resource. A form is a container of all kinds of parameters, which are sent via **GET** or **POST** to the server, and that is today the dynamics or the model that is used to be able to have functional web applications.
- ✓ A form is built on top of the **<form>** tag, containing child nodes with all the controls and possessing certain attributes and content. There are certain key attributes like **"action"**, **"method"**, and it is required to send the parameters with the **"name"** attribute and the proper **"type"**. There are some important events that are also handled, the most important and fundamental one is **"submit"**, and others less transcendental like **"reset"** to clean the input data from the form.

```
spalaciosgt.github.io > week04 > <> forms_example.html > ...
1  <!doctype html>
2  <html lang='en'>
3      <head>
4          <meta charset='utf-8'>
5          <title>Search</title>
6      </head>
7      <body>
8          <form name='search' action='/search'>
9              <input name='searchInput'>
10                 <button type='submit'>Search</button>
11                 <button type='reset'>Reset</button>
12             </form>
13             <script src='main.js'></script>
14         </body>
15     </html>
```

- ✓ A form is like any other node or control within the web page, so it responds to the model of the **event listeners**. The most important **attributes** and **methods** of a form are summarized below.

Attributes	
action	The server-side resource that will response the request with the form and its parameters. By default, it is the same page or location where the form is.
method	It specifies the HTTP method to send the data from the form to the server. It could be GET (for URL variables) or POST (transaction data). By default, is GET.
autocomplete	Allows us to specify to the browser whether the form inputs can be autocompleted from data recorded in the browser.
novalidate	It is boolean, it tells the browser if the inputs should be validated or not, that is, if this attribute is false, even if the inputs have validations, they will be ignored.

Events	
onchange	Fires when some form data starts to change.
onsearch	Fired when the form is submitted by an input action of type "search".
onsubmit	Fires when the form is submitted to the server, either by a GET or a POST.

- ✓ A form can contain one or more controls, typically inputs, selects, text areas, and buttons. With these **3 types** of controls, you can design and implement any type of form. The following example uses a few different types of input and renders it in a function, even using a JSON format function.

```

JS main.js U  forms_example.html U
spalaciosgt.github.io > week04 > forms_example.html > ...
1  <!doctype html>
2  <html lang='en'>
3    <head>
4      <meta charset='utf-8'>
5      <title>Hero Form</title>
6    </head>
7    <body>
8      <form id='hero'>
9        <label for='heroName'>Name:
10       <input type='text' id='heroName' name='heroName' autofocus placeholder='Name'>
11      </label>
12      <button type='submit'>Submit</button>
13    </form>
14    <script src='main.js'></script>
15  </body>
16 </html>

```

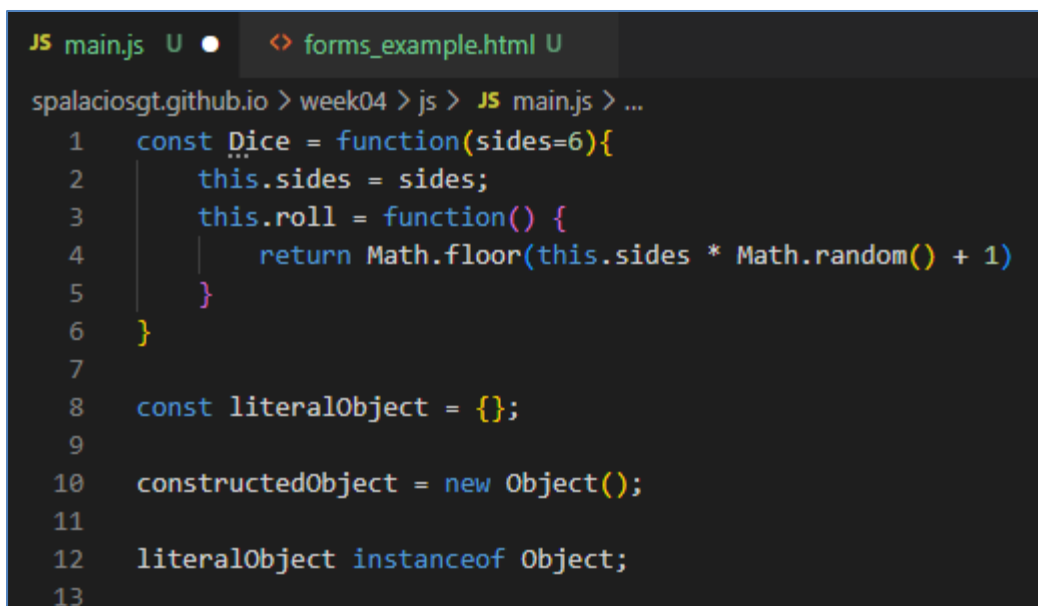
```
JS main.js U • <> forms_example.html U
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  function makeHero(event) {
2      event.preventDefault(); // prevent the form from bei
3      const hero = {}; // create an empty object
4      hero.name = form.heroName.value; // create a name pr
5      alert(JSON.stringify(hero)); // convert object to JS
6      return hero;
7  }
```

- ✓ The most used controls in a form are the **<input>**, and there are different types. There are the normal input inputs, normal information, passwords, check boxes, radio button options, hidden fields, among others.
- ✓ To other complementary controls such as **<label>** tags and **<buttons>** buttons, but also to more specialized ones such as **<selects>**, such as drop-down option lists and text combo boxes.
- ✓ When there are large memo-type information inputs, such as descriptions, a **<textarea>** tag or control should be used. These controls are better suited to that need than a conventional **<input>**. The most used **<input>** types are detailed below:
  - Button
  - Checkbox
  - Color
  - Date
  - Email
  - File
  - Hidden
  - Image
  - Month
  - Number
  - Password
  - Radio
  - Range
  - Reset
  - Search
  - Submit
  - Telephone
  - Text
  - URL
  - Week

- ✓ There are 2 types of **form validations**: if an input is mandatory or optional, and by data type. For example, if an input is of type "email"...the expected input is an email pattern, if it is of type "number" a number is expected, and so on. It is important to know that with the "NoValidate" attribute these validations can be disabled, so it is important to always validate on the server side, if it can even be **sanitized**.

## 2. Ch12: Object Oriented Programming.

- ✓ **Object-oriented programming** is the foundation of modern programming, it's almost impossible to find a programming language or development platform that hasn't acquired it, and JavaScript is no exception. This paradigm is based on 4 pillars: abstraction (the power to go from the general to the specific, specification), inheritance (a parent class inherits from the child classes), polymorphism (multiple forms in attributes and methods), and encapsulation (declarative programming)., the behavior is shown to the outside world but not how it is done, only what is done).
- ✓ JavaScript was not born supporting objects, so some features of this paradigm are not supported or implemented differently than programming languages that are 100% object-oriented, for example in JavaScript inheritance is handled in the **prototype-based** and **literal objects** by what an object will do whatever the prototype it inherited from can do.
- ✓ When an object is instantiated, that is, an object is created, a **constructor** is called. JavaScript, because of the way it handles inheritance and not being strongly typed, is flexible in this regard. The constructor is any function, even something that is not defined. The following are examples of constructor functions.



```
JS main.js U • <> forms_example.html U
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  const Dice = function(sides=6){
2      this.sides = sides;
3      this.roll = function() {
4          return Math.floor(this.sides * Math.random() + 1)
5      }
6  }
7
8  const literalObject = {};
9
10 constructedObject = new Object();
11
12 literalObject instanceof Object;
13
```

- ✓ In most object-oriented programming languages, classes are predefined in separate files, like in Java. In the case of JavaScript, until before a certain version, **function-**

**based constructors** were the only way to instantiate objects. From a certain more modern version, it is now possible to define classes, it may be in different files, but it can be like programming languages that were previously 100% object-oriented.

```
JS main.js U ● <> forms_example.html U
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  class Dice {
2      constructor(sides=6) {
3          this.sides = sides;
4      }
5
6      roll() {
7          return Math.floor(this.sides * Math.random() + 1)
8      }
9  }
```

- ✓ As in object-oriented languages, in a JavaScript class you can handle **static** properties or attributes, and methods, that is, those that do not need to build the object to invoke them, but only refer to the class itself.

```
JS main.js U ● <> forms_example.html U
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  class Turtle {
2      constructor(name,color) {
3          this.name = name;
4          let _color = color;
5          this.setColor = color => { return _color = color; }
6          this.getColor = () => _color;
7      }
8  }
9
```

- ✓ **Inheritance** is one of the pillars of object-oriented programming. In most object-oriented programming languages, this occurs almost naturally, there are more general abstract classes that everyone inherits from...the most general class that everyone inherits from is Object. The same happens in JavaScript, it works in the same way, although the use of prototypes and prototype chains is more used.

As far as inheritance is concerned, JavaScript only has one practical structure: objects. Each object has a private property (**prototype**) that holds a link to another object called its **prototype**. That prototype object has its own prototype, and so on until an object whose prototype is null is reached. By definition, **null** has no prototype, and acts as the final link in this chain of prototypes.

```

JS main.js U • <> forms_example.html U
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  class Turtle {
2      constructor(name) {
3          this.name = name;
4      }
5      sayHi() {
6          return `Hi dude, my name is ${this.name}`;
7      }
8
9      swim() {
10         return `${this.name} paddles in the water`;
11     }
12 }
13
14 class NinjaTurtle extends Turtle {
15     constructor(name) {
16         super(name);
17         this.weapon = 'hands';
18     }
19     attack() { return `Feel the power of my ${this.weapon}!` }
20 }
21


```

```

JS main.js U • <> forms_example.html U
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  function hacerAlgo(){}
2
3  console.log( hacerAlgo.prototype );
4  var hacerAlgo = function(){};
5  console.log( hacerAlgo.prototype );
6

```

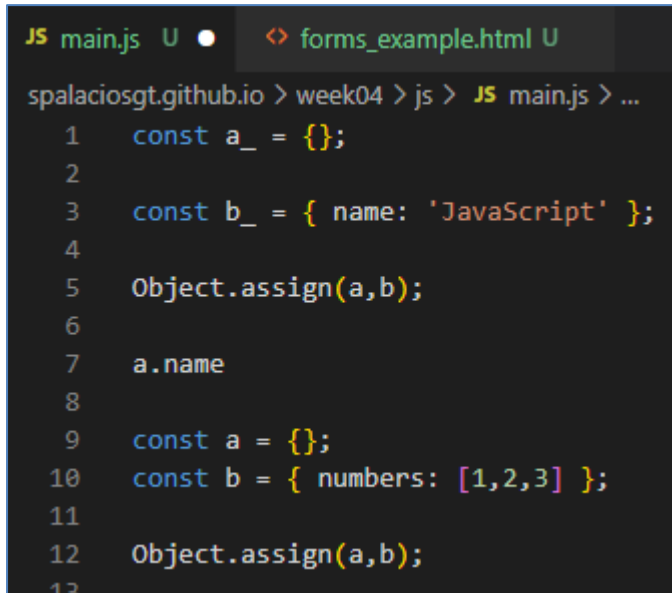
- ✓ **Polymorphism** is another pillar of object-oriented programming, it manifests itself in many ways, overloading attributes, and methods, and also applying interfaces, that is, using the same methods for different objects and obtaining different behaviors. The following example shows how the "showInfo" method works no matter what type of object is sent as a parameter.

JS main.js U  forms\_example.html U

- ✓ It is possible to create an object based on another by means of prototypes and another object already created this by means of **prototype chains**. The following example shows how this applied inheritance works through prototypes and prototype chains.

JS main.js U forms\_example.html U

- ✓ There is another very useful concept called "**Mixins**" that allows you to do the same thing as inheritance but through direct assignment of memory values to properties or methods. An example of this concept is shown below.



The screenshot shows a code editor with two tabs: 'JS main.js' and 'forms\_example.html'. The active tab is 'JS main.js'. The code in the editor is as follows:

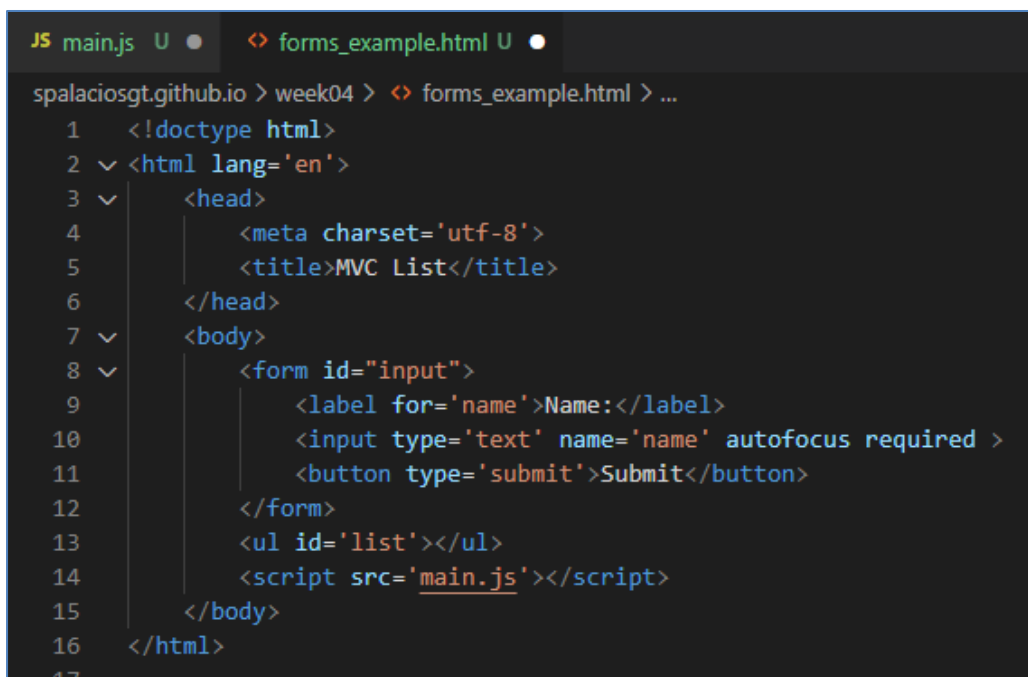
```
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  const a_ = {};
2
3  const b_ = { name: 'JavaScript' };
4
5  Object.assign(a,b);
6
7  a.name
8
9  const a = {};
10 const b = { numbers: [1,2,3] };
11
12 Object.assign(a,b);
13
```

### 3. Ch15: Modern Javascript

- ✓ JavaScript is a programming language that started on the web to make the static content of hyperlinked tags dynamic, but over time it became the only way to enrich the user interface and make it even more dynamic. But in the last 15 years the appearance of frameworks based on JavaScript has increased exponentially, almost every year we see a new one, and they have become standard in the industry... and a **modern JavaScript** application implements at least one of them.
- ✓ **Libraries** allow grouping object definitions (attributes and methods) that allow all kinds of tasks to be carried out and can contain a significant number of namespaces or source files for JavaScript. These libraries allow to reduce the complexity of many tasks, for example the manipulation of the DOM. The idea of having a library is that what would take 5 lines of code can be done in just 1 line using a library. The library fights with the complexity that even the browser can give, and the programmer only uses it, very similar to using an API.
- ✓ **Advantages** of using a library, simplify complexity, have the support of a company or community that maintains it, comply with standards, be more efficient. **Disadvantages**, having to adapt a definition or convention defined by the manufacturer of the library or API, consuming more resources by often only using a couple of functions of the library and loading all the possible ones, having a constant maintenance of its versions , etc.



- ✓ When to use a **library**? If the application is very simple, if a lot of efficiency is needed, for example a web service that will be used by millions of people and does something very simple, in those cases it is not recommended. But if it is a business application, something extensive, it is recommended to use it. Currently there are several of them that are industry standards and are supported by companies or communities as large as Google, so their use is more than recommended for the latter cases.
- ✓ One of the first frameworks or libraries was **jQuery**. This library marked a milestone, then many more came out, until reaching **NodeJS**, which allows executing JS on the server side to be displayed on the client and marked another generational leap.
- ✓ **NodeJS** has become a standard for JavaScript, and it is possible to use several of the functions or modules that it has in any JS application, it is something that is interoperable.
- ✓ **MVC** is the most important design pattern of recent times. This design pattern states that any problem or design can be solved by dividing the problem into 3 layers: the model, the view, and the controller. These 3 layers will solve the design problem by displaying the view to the user, the controller collecting user requests and requesting data from the model, and vice versa. This dynamic of going from the view to the controller to the model and back allows you to create robust modular applications that allow you to have this independence of layers. Without something going on with the view, this change shouldn't affect both the controller and the model; if the controller changes, it shouldn't affect the view or the model, and the same with the model, the idea is independence. This pattern is the basis for many software architectures models. The following example implements all 3 layers.



```
JS main.js U • forms_example.html U •
spalaciosgt.github.io > week04 > forms_example.html > ...
1 <!doctype html>
2 <html lang='en'>
3 <head>
4   <meta charset='utf-8'>
5   <title>MVC List</title>
6 </head>
7 <body>
8   <form id="input">
9     <label for='name'>Name:</label>
10    <input type='text' name='name' autofocus required >
11    <button type='submit'>Submit</button>
12  </form>
13  <ul id='list'></ul>
14  <script src='main.js'></script>
15 </body>
16 </html>
17
```

```

JS main.js U ● <> forms_example.html U ●
spalaciosgt.github.io > week04 > js > JS main.js > ...
1  'use strict'
2  const form = document.forms[0];
3  class Item {
4  constructor(name) {
5      this.name = name;
6  }
7  }
8  const controller = {
9      watch(form) {
10         form.addEventListener('submit', (event) => {
11             event.preventDefault(); // prevent the form from being submitted
12             this.add(form.name.value);
13         }, false);
14     },
15
16     add(name) {
17         const item = new Item(name);
18         view.render(item);
19     }
20 };
21 const view = {
22     render(item) {
23         const list = document.getElementById('list');
24         const li = document.createElement('li');
25         li.innerHTML = item.name;
26         list.appendChild(li);
27         // reset the input field
28         form.name.value = '';
29     }
30 };
31 controller.watch(form);

```

- ✓ There are several **MVC** JS-based libraries or frameworks such as NodeJS, AngularJS, ReactJS, Aurelia, Amber, etc. All of these are supported by companies like Google and communities and are made up of a large amount of code and dependencies. Therefore, it is necessary to use **package managers** to install the appropriate dependencies at the appropriate level.

```
JS main.js U • {} package.json U • <> forms_example.html U •
spalaciosgt.github.io > week04 > js > {} package.json > ...
1  {
2      "name": "project-x",
3      "version": "1.0.0",
4      "description": "",
5      "main": "index.js",
6      "scripts": {
7          "test": "echo \"Error: no test specified\" && exit 1"
8      },
9      "author": "",
10     "license": "ISC"
11 }
```

- ✓ The most important or frequently used use cases with "**npm**" are summarized below.

Command Option	Description
npm init	Start a new project.
npm install -save	Install a library/package inside the project source directories.
npm install -global	Install a library/package globally available to any project in the machine.
npm list	Lists the installed packages or libraries.
npm outdated	Check last update libraries/packages.
npm update	Update to last versions libraries/packages.
npm uninstall	Uninstall specific libraries/packages.

- ✓ **Deploying** an application is publishing it for use in a development, pre-production, or production environment, already on an application server. As we all know, JavaScript is an interpreted language so it is necessary to publish the source; but by having the source available, anyone can see it and copy it, so you need to obfuscate it or make it unreadable so it can't be copied so easily. Apart from obfuscating, this process also considers the **compression** of the code through references that are duplicated or can be optimized.
- ✓ There is a **structure of directories** that allows you to manage and organize all the files of an application, highlighting the source directories (src), lib (libraries), node\_modules (NodeJS modules), app (application files, which will be executed with the end user), dist (the deliverables or executables that are deployed to get the application running). Packaging all this is the concept of "web pack" which finally brings together all the project files and can also be managed with "npm".