WDD 330 Web Frontend Development II
Samuel Palacios
Week 03
May 11th, 2022

Readings Assignment Notes

1. Object Methods: this

- ✓ Object-oriented programming has revolutionized the world of programming, and it is not the exception in languages that were previously rather structured only by modules or functions such as JavaScript. JavaScript also allows you to implement classes and objects. A class is a definition of an **abstract data** type made up of certain characteristics (properties) and certain behavior (methods).

- ✓ The "this" keyword has been implemented in most object-oriented programming languages. For example, in Java, it is used to refer to the object itself, and it is very common to see it in constructors to differentiate the parameters sent from the object's own attributes, but in JavaScript it goes further, it can have different functions depending on how it is invoked. The most general use is a method that allows **access to the members** (properties or methods) of the object that is being referenced.

- ✓ In the following example, if the "this" method did not exist, we would have to put an absolute reference to the object it is referring to, we would have to define this method for each particular object where it is to be implemented, but ***this*** **generalizes** this part:

```
let user = { name: "John" };
let admin = { name: "Admin" };

function sayHi() {
  alert( this.name );
}

// use the same function in two objects
user.f = sayHi;
admin.f = sayHi;

// these calls have different this
// "this" inside the function is the object "before the dot"
user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)

admin['f'](); // Admin (dot or square brackets access the method – doesn't matter)
```

- ✓ There is another method that can replace "this" and that exists in all JavaScript functions, when "this" is used to include members of the object that were not previously defined as arguments...***bind*** **can be used to load such arguments or members** in the same call.

- ✓ In the following example, "this" or "bind" could be used to **load these arguments** so that this function can be added to the boundFunc object.

```
function sayParams (...args) {
    console.log(...args)
}
const boundFunc = sayParams.bind(null, 1, 2, 3, 4, 5)
boundFunc()
```

2. Ch5: Objects

- ✓ As we saw earlier, everything in JavaScript is primitive data or **objects**, this final part being the largest number of data structures that are handled in a real-world application program. Primitive data have direct reference to a space on the "stack", while objects are pointers or an address to a memory space on the **"heap"**.

- ✓ I hadn't seen the example of **"Object Literals"** before. This concept is new to me, it is practically the definition of an object (properties, data, or attributes + behavior, methods, or functions) without the need to define it in a previously created template or class. In the following example you see a literal object, which can then be used in code within JavaScript, and was created without a previously established class, it is **"literal"**.

```
const superman = {
  name: 'Superman',
  'real name': 'Clark Kent',
  height: 75,
  weight: 235,
  hero: true,
  villain: false,
  allies: ['Batman','Supergirl','Superboy'],
  fly() {
      return 'Up, up and away!';
  }
  };
```

- ✓ Objects in JavaScript can be created from scratch, just by declaring them and putting curly braces, or they can be completely defined in a structured way. Is there any flexibility to define or restructure an object, which I haven't seen in another language I guess it's because **JavaScript isn't strongly typed**, in JavaScript you can add, modify, remove attributes or methods, and you can do almost anything with them even mix them.

- ✓ This topic is vital for what is to come, **nested objects**. Objects can be inside other objects at the attribute level, that is, nested objects. This definition will allow representing important things like JSON later. In the following example we see how an object contains others, in this case "jla" to several superheroes.

```
const jla = {
    superman: { realName: 'Clark Kent' },
    batman: { realName: 'Bruce Wayne' },
    wonderWoman: { realName: 'Diana Prince' },
    flash: { realName: 'Barry Allen' },
    aquaman: { realName: 'Arthur Curry' },
}
```

- ✓ Objects can be passed as parameters too, in fact this principle allows working with **lambda expressions** when a function can be passed as a parameter when executing another function, it's kind of fascinating.

```
function greet({greeting='Hello',name,age=18}) {
    return `${greeting}! My name is ${name} and I am ${age} years old.`;
}
```

- ✓ In programming with JavaScript then we can create different constants, variables, functions, objects, etc. From one programmer to another, certain objects or elements could match the name and have conflicts or collisions, and this is a problem for the Interpreter. N**amespaces** are used to differentiate them; this allows there to be no conflicts in the names of variables or methods that can be defined in the different JavaScript elements.

- ✓ This is a very simple example of **Namespaces:**

```
varsampleNamespace = {
    function_one: function()
    {
    // body of code
    },
    function_two: function()
    {
    // body of code
    }
    };
    sampleNamespace. function_one ();
```

- ✓ **JSON** is a core issue in software development today, maximum in everything that is web and backend services. This notation is based on the idea of nested objects, it is based on key-value notation. A JSON object is practically a dictionary in other languages like Java and Python. In the following example you can see a constant in the form of an object and a JSON, there are quite a few similarities.

```
const block01 = [
    {
        label: "Week01",
        url: "week01/"
    },
    {
        label: "Week02",
        url: "week02/"
    },
    {
        label: "Week03",
        url: "week03/"
    }
]

{
    block01: [
        {
            label: "Week01",
            url: "week01/"
        },
        {
            label: "Week02",
            url: "week02/"
        },
        {
            label: "Week03",
            url: "week03/"
        }
    ]
}
```

3. Ch6: Document Object Model

- ✓ The **DOM** is a very important concept. When we work with web pages, the elements displayed to the end user are built on a document based on tags which have the shape of a tree. A tree is a basic data structure in computing, and it works through nodes that can have children (zero or more), and these in turn can also have other children. Also, a node can have attributes and a content. In the following example the <html> tag is the parent or root element of the entire document, then there are the 2 children <head> and <body>, and subsequently these in turn have other children and so on. We see how several nodes or tags like "li" have attributes and content (between the opening and closing tags).

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Justice League</title>
</head>
<body>
    <header>
    <h1 id='title'>Justice League</h1>
    </header>
    <ul id='roster'>
    <li class='hero'>Superman</li>
    <li class='vigilante hero' id='bats'>Batman</li>
    <li class='hero'>Wonder Woman</li>
    </ul>
</body>
</html>
```

- ✓ From JavaScript, having the content of a document ready, what is involved is preparing or **manipulating** the elements that are presented to the user through the DOM. For this it is necessary to obtain the reference to the desired node. The following methods allows us this:

    - getElementById
    - getElementsByClassName
    - getElementsByTagName
    - querySelector
    - querySelectorAll

  Something important about all the elements or nodes in an HTML document is that they must have a unique ID which will allow them to be unequivocally identified. The previous methods allow to obtain one or more elements or nodes of the HTML

document based on the Id, the Class (several elements tagged with a class), by Tag Name, or other criteria based on attributes or content.

- ✓ The following examples locate HTML elements by Id, Tag Name, Class Name, or by a **query** that can locate tag names, or any combination of the above options.

```
const h1 = document.getElementById('title');
const listItems = document.getElementsByTagName('li');
const heroes = document.getElementsByClassName('hero');
document.querySelector('#bats');
const ul = document.querySelector('ul#roster');
```

- ✓ All nodes or HTML elements are arranged for a single purpose: to present information to the end user. This information can be obtained knowing in advance the ID or the Class of the object that contains it, but certain **navigation expressions** can also be performed to obtain this content.

```
const textNode = wonderWoman.firstChild;
textNode.nodeValue;
```

- ✓ But how do we manipulate the DOM of a document? It's simple, **updating attribute values or content**. If the end user performs an action, we can present the result of that action by manipulating a tag's content and its appearance... for them we can manipulate the DOM to do it.

```
wonderWoman.setAttribute('class', 'got_result');
wonderWoman.getAttribute('got_result');
```

- ✓ Something really about the DOM is that it can also be manipulated in the sense that it is possible to create or modify elements programmatically, that is, handle **dynamic HTML** to cover the frontend requirements that are needed in an application. It is possible to create, modify and remove any HTML element or node, as in the previous example.

```
const h01 = document.getElementById('title');
const oldText = h01.firstChild;
const newText = document.createTextNode('Justice League of Latin America');
h1.replaceChild(newText,oldText);
```

When we see web pages with input data forms, if you put one option or another sometimes elements disappear or appear instantly on the screen. A very common way to do it is by manipulating the name of the class, for example if it should appear the class "Appear" should be assigned, otherwise "Desapper".

- ✓ We can also modify the appearance by modifying the **CSS** through the "style" attribute as shown in the following example. Although this can be done, it is best to have the most suitable classes in the CSS definition and then only assign or unassign classes.

```
const heroes_ = document.getElementById('biz');
const superman_ = heroes.children[0];
superman.style.border = "red 2px solid";
```

- ✓ Quiz **Ninja** worked:

```
<!doctype html>
<html lang='en'>
<head>
<meta charset='utf-8'>
<meta name='description' content='A JavaScript Quiz Game'>
<title>Quiz Ninja</title>
<link rel='stylesheet' href='styles.css'>
</head>
<body>
<section class='dojo'>
    <div class='quiz-body'>
    <header>
            <div id='score'>Score: <strong>0</strong></div>
        <h1>Quiz Ninja!</h1>
    </header>
    <div id='question'></div>
    <div id='result'></div>
    <div id='info'></div>
    </div>
</section>
<script src='main.js'></script>
</body>
```

```
// View Object
const view = {
    score: document.querySelector('#score strong'),
    question: document.getElementById('question'),
    result: document.getElementById('result'),
    info: document.getElementById('info'),
    render(target,content,attributes) {
        for(const key in attributes) {
            target.setAttribute(key, attributes[key]);
        }
        target.innerHTML = content;
    }
    };

    ask(){      You, hace 1 segundo • Uncommitted changes
        const question = `What is ${this.question.name}'s real name?`;
        view.render(view.question,question);
        const response =  prompt(question);
        this.check(response);
    },
    check(response){
        const answer = this.question.realName;
        if(response === answer){
        view.render(view.result,'Correct!',{'class':'correct'});
        alert('Correct!');
        this.score++;
        view.render(view.score,this.score);
        } else {
        view.render(view.result,`Wrong! The correct answer was ${answer}`,{'class':'wrong'});
        alert(`Wrong! The correct answer was ${answer}`);
```

4. Ch7: Events

- ✓ The **events** are the way to interact with the user. When an end user does anything with the keyboard or mouse while viewing our web page, it is possible to capture any of those events to perform a specific reaction such as manipulating DOM or CSS elements.

- ✓ There are several ways to define an action as a stimulus to an event that is triggered by something the end user does, but the most standard and recommended way is to implement **"Event Listeners"**.

```
function doSomething() {
    alert('You Clicked!');
    }

    addEventListener('click',doSomething);
```

As in the above example, an event listener is the definition of an event (already defined according to the version of JavaScript and supported by the browser), and a function that contains the code that will be fired when an action that fires that event occurs, in this example is “click” and doSomething”.

- ✓ The **"Event Target",** when an event listener is defined, it is necessary to have access to the node where the event was executed, then this "event" parameter allows access to the control or controls where the event occurred. This is great for programming; you can access the control's data and do whatever the trigger completes. In the following example you can see how data can be obtained from the "Event" parameter, and this event allows access to data entered by the user, displays, attributes, etc.

```
function doSomething(event){
    console.log(event.target);
    }
```

- ✓ Before the smartphone era there were only 2 types of events, **"keyboard"** and **"mouse"**, now we add a new **"touch"** section. The latter has certain differences, and the keyboard and mouse ones are the classic ones, putting the mouse in front of a control, clicking, holding the click, right clicking, etc. The event listeners can be modified, you can assign a function or remove it if necessary, also remove the event listener, etc.