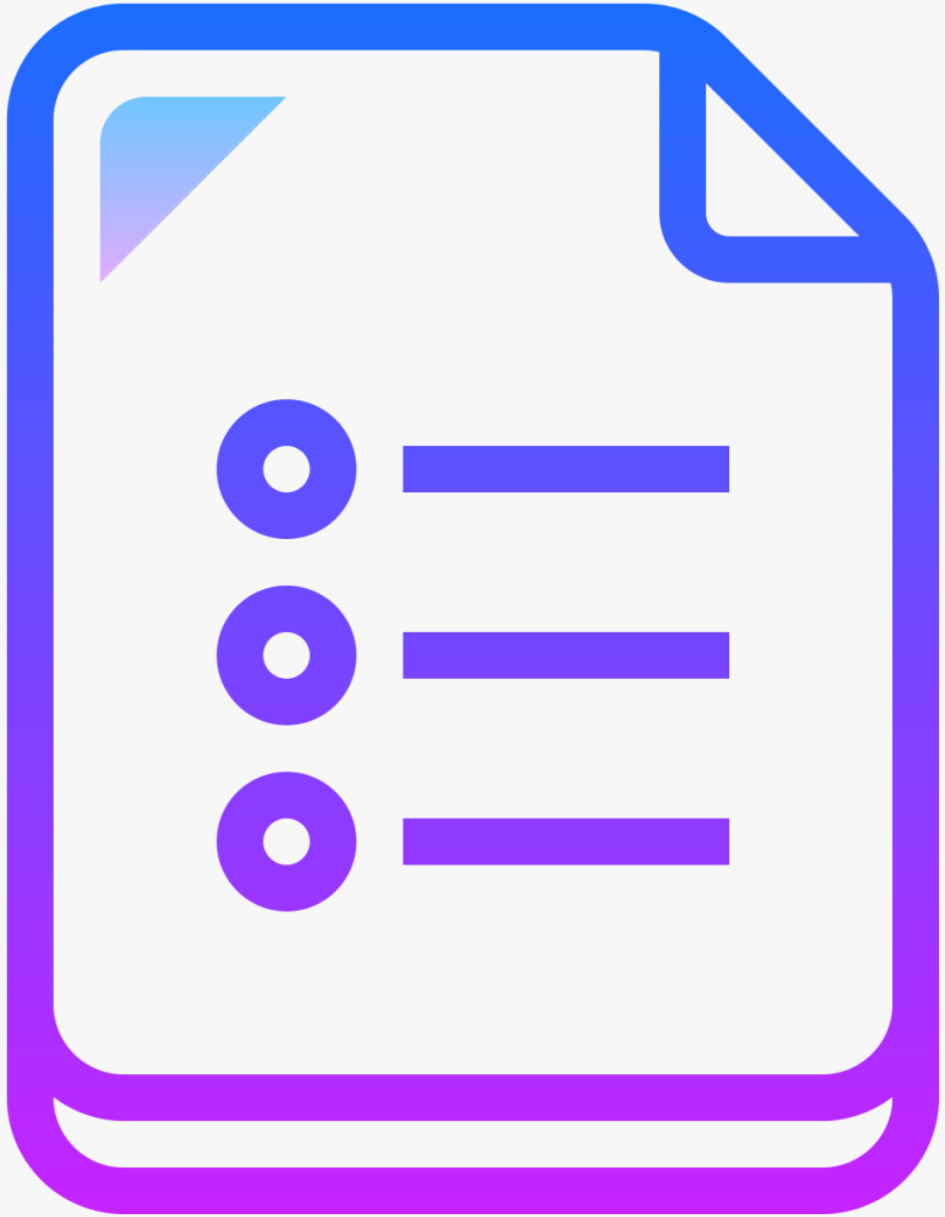
BOOTCAMP ’20

SESIONI **#18**

JS

What will be covered?

- Classes
- Modules
- Import and Export
- this identifier



Classes

- Klasat në JavaScript definohen përmes çelësfjalës **class**, pason emri i klasës, kllapat gjarpërore dhe brenda tyre trupi i klasës.

```
class Person {
    // class body
}
```

- Klasa ka një metodë speciale e cila njihet si konstruktor dhe definohet me constructor() – shkrepet sa herë që krijohet një objekt i ri nga ajo klasë.

Classes

```
class Person {
    constructor(name, surname, age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }
}
```

class expression

```
let Person = class {
    constructor(name, surname, age) {
        this.name = name;
        this.surname = surname;
        this.age = age;
    }
};
```

- Krijimi i një objekti të klasës bëhet përmes çelësfjalës new

```
const p = new Person("John", "Smith", 24);
```

```
const p = new Person("John", "Smith", 24);
```

Classes

Një klasë mund të ketë edhe funksione brenda saj – funksionet brenda klasave quhen metoda (prototype methods)

```
getAge() {
    return this.age;
}
```

Thirrja (ekzekutimi) i metodës:

```
console.log(o.getAge())
```

Classes

- Një klasë gjithashtu mund të ketë edhe metoda statike – që d.m.th se mund të ekzekutohen pa krijuar instancë të klasës ku është definuar

```
static address() {
    console.log("Po Box 100");
}
```

Thirrja (ekzekutimi) i metodës statike:

```
Person.address();
```

Static methods

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

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

        return Math.hypot(dx, dy);
    }
}

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

console.log(Point.distance(p1, p2));
```

Public field declarations

With the JavaScript field declaration syntax, the above example can be written as:

```
class Rectangle {
  height = 0;
  width;
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

By declaring fields up-front, class definitions become more self-documenting, and the fields are always present.

Private field declarations

Using private fields, the definition can be refined as below.

```
class Rectangle {
  #height = 0;
  #width;
  constructor(height, width) {
    this.#height = height;
    this.#width = width;
  }
}
```

It's an error to reference private fields from outside of the class; they can only be read or written within the class body. By defining things which are not visible outside of the class, you ensure that your classes' users can't depend on internals, which may change version to version.

Sub classing with `extends`

The `extends` keyword is used in *class declarations* or *class expressions* to create a class as a child of another class.

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

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and pass in the name parameter
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

Classes - Browser support

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
`classes`	49	13	45	No	36	9	49	49	45	36	9	5.0	6.0.0
`constructor`	49	13	45	No	36	9	49	49	45	36	9	5.0	6.0.0
`extends`	49	13	45	No	36	9	49	49	45	36	9	5.0	6.0.0
Private class fields	74	79	No	No	62	14	74	74	No	53	14	No	12.0.0
Public class fields	72	79	69	No	60	14	72	72	No	51	14	No	12.0.0
`static`	49	13	45	No	36	9	49	49	45	36	9	5.0	6.0.0
Static class fields	72	79	75	No	60	No	72	72	No	51	No	No	12.0.0

Modules

What is a module?

A module is just a file. One script is one module. As simple as that.

Modules can load each other and use special directives `export` and `import` to interchange functionality, call functions of one module from another one:

- `export` keyword labels variables and functions that should be accessible from outside the current module.
- `import` allows the import of functionality from other modules.

Modules – Import/Export

For instance, if we have a file `sayHi.js` exporting a function:

```
// 📁 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

...Then another file may import and use it:

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

The `import` directive loads the module by path `./sayHi.js` relative to the current file, and assigns exported function `sayHi` to the corresponding variable.

Modules – Import/Export (Example)

Moduli:

```
export function displayImage(path, width, elem) {
    const img = document.createElement('img');
    img.setAttribute('src', path);
    img.setAttribute('width', width);
    elem.appendChild(img);
}

// export { displayImage }
```

Modules – Import/Export (Example)

Përdorimi i modulit:

```
<button id="loadImage">Load image</button>

<div id="img"></div>

<script type="module">
    import { displayImage } from './modules/imageLoader.js';

    const parent = document.getElementById('img');
    const img_url = "https://d39w7f4ix9f5s9.cloudfront.net/dims4/default/54e9

    document.getElementById('loadImage').addEventListener('click', (e) =>
        displayImage(img_url, '600px', parent)
    );
</script>
```

Modules – this

In a module, “this” is undefined

That’s kind of a minor feature, but for completeness we should mention it.

In a module, top-level `this` is undefined.

Compare it to non-module scripts, where `this` is a global object:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

this identifier

What is **this**?

The JavaScript `this` keyword refers to the object it belongs to.

It has different values depending on where it is used:

- In a method, `this` refers to the **owner object**.
- Alone, `this` refers to the **global object**.
- In a function, `this` refers to the **global object**.
- In a function, in strict mode, `this` is `undefined`.
- In an event, `this` refers to the **element** that received the event.
- Methods like `call()`, and `apply()` can refer `this` to **any object**.

this identifier

this in a Method

In an object method, `this` refers to the "**owner**" of the method.

In the example on the top of this page, `this` refers to the **person** object.

The **person** object is the **owner** of the **fullName** method.

```
fullName : function() {
  return this.firstName + " " + this.lastName;
}
```

this identifier

this Alone

When used alone, the **owner** is the Global object, so `this` refers to the Global object.

In a browser window the Global object is `[object Window]`:

Example

```
var x = this;
```

this identifier

In **strict mode**, when used alone, `this` also refers to the Global object `[object Window]`:

Example

```
"use strict";
var x = this;
```

this identifier

this in a Function (Default)

In a JavaScript function, the owner of the function is the **default** binding for `this`.

So, in a function, `this` refers to the Global object `[object Window]`.

Example

```
function myFunction() {
  return this;
}
```

this identifier

this in a Function (Strict)

JavaScript **strict mode** does not allow default binding.

So, when used in a function, in strict mode, `this` is `undefined`.

Example

```
"use strict";
function myFunction() {
  return this;
}
```

this identifier

this in Event Handlers

In HTML event handlers, `this` refers to the HTML element that received the event:

Example

```
<button onclick="this.style.display='none'">
  Click to Remove Me!
</button>
```

this identifier

Explicit Function Binding

The `call()` and `apply()` methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.

You can read more about `call()` and `apply()` later in this tutorial.

this identifier

In the example below, when calling person1.fullName with person2 as argument, `this` will refer to person2, even if it is a method of person1:

Example

```
var person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
var person2 = {
  firstName:"John",
  lastName: "Doe",
}
person1.fullName.call(person2);  // Will return "John Doe"
```

this identifier

```
class Dog {
  constructor() {
    this.sound = 'woof'
  }
  talk() {
    console.log(this.sound)
  }
}
const sniffles = new Dog()
sniffles.talk() // Outputs: "woof"

$('button.myButton')
  .click(sniffles.talk)
```

this identifier

```
class Dog {
  constructor() {
    this.sound = 'woof'
  }
  talk() {
    console.log(this.sound)
  }
}
const sniffles = new Dog()
sniffles.talk() // Outputs: "woof"

$('button.myButton')
  .click(sniffles.talk.bind(sniffles))
```

this identifier

```
class Dog {
  constructor() {
    this.sound = 'woof'
  }
  talk() {
    console.log(this.sound)
  }
}
const sniffles = new Dog()
sniffles.talk() // Outputs: "woof"

$('button.myButton')
  .click( _ => sniffles.talk() )
```

this identifier

```
const dog = () => {
  const sound = 'woof'
  return {
    talk: () => console.log(sound)
  }
}
const sniffles = dog()
sniffles.talk() // Outputs: "woof"

$('button.myButton')
  .click(sniffles.talk)
```

Works!

QUESTIONS