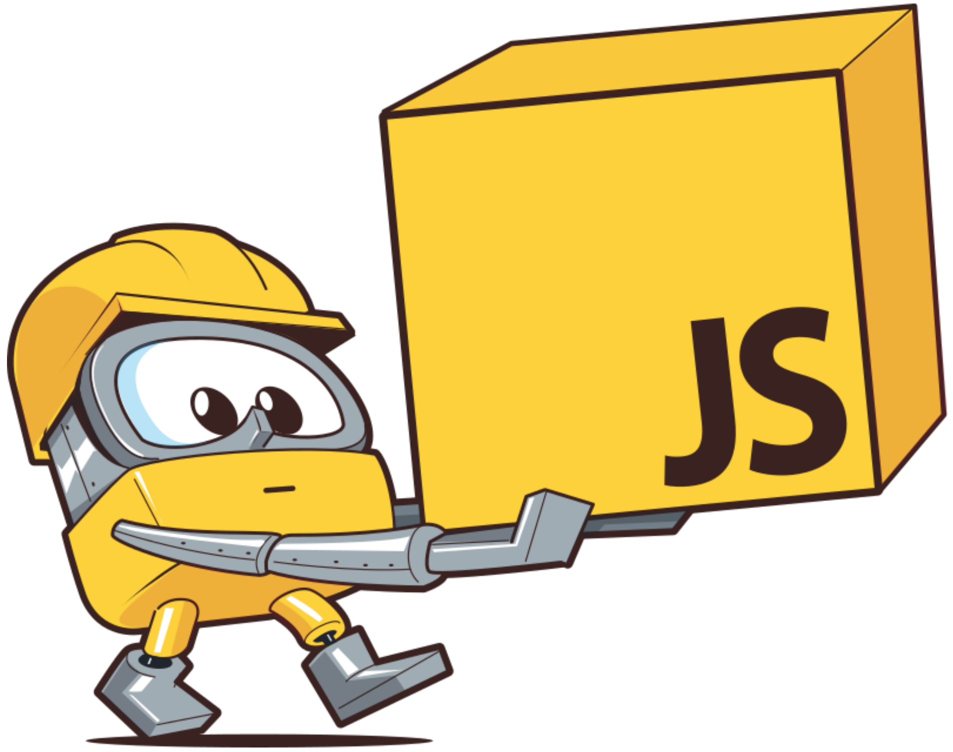
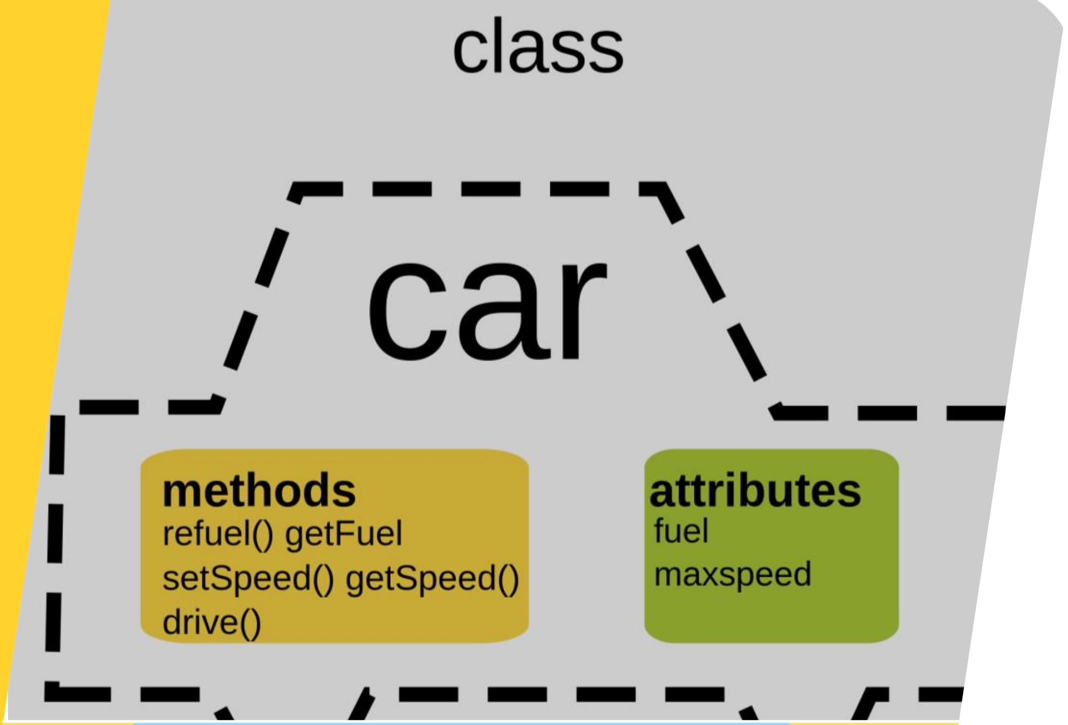
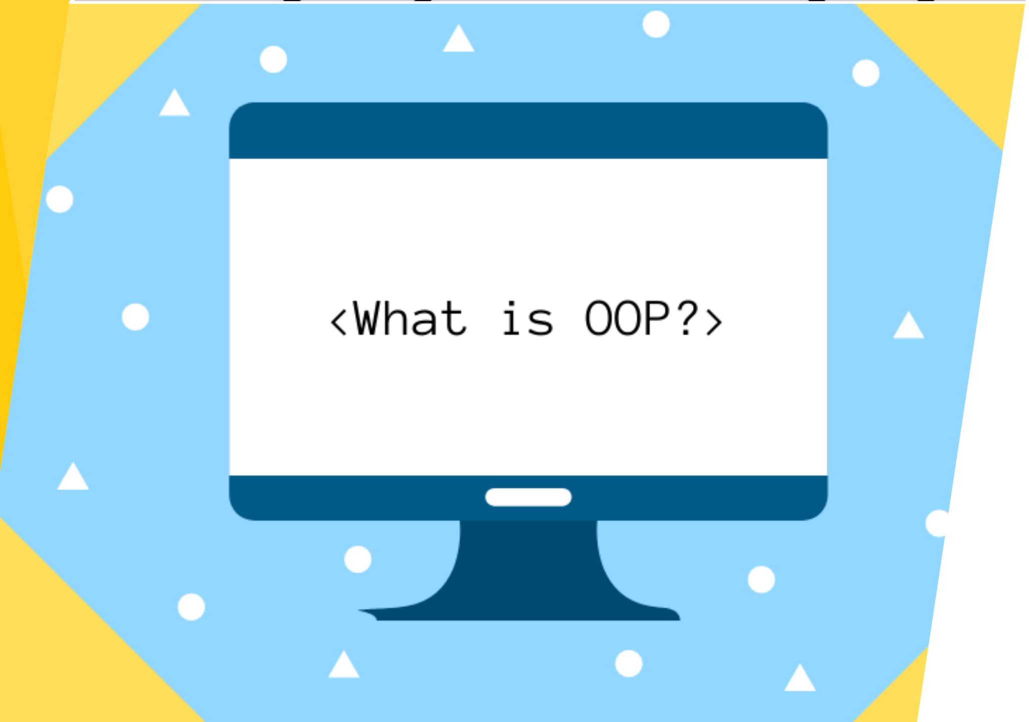


Advanced JS





Object Oriented Programing

Focus points:

- What is OOP?
- Objects And Object Constructors
- Factory Functions
- Classes
- Inheritance
- Encapsulation
- NPM & Installation
- WebPack
- ES6 Modules

What is OOP?

Object-oriented programming (OOP) - is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

OOP is about modeling a system as a collection of objects, where each object represents some particular aspect of the system. Objects contain both functions (or methods) and data. An object provides a public interface to other code that wants to use it but maintains its own private, internal state; other parts of the system don't have to care about what is going on inside the object.

There are a lot of OOP principles, but we are going to focus on these 4:

- **Objects**
- **Classes**
- **Inheritance**
- **Encapsulation**

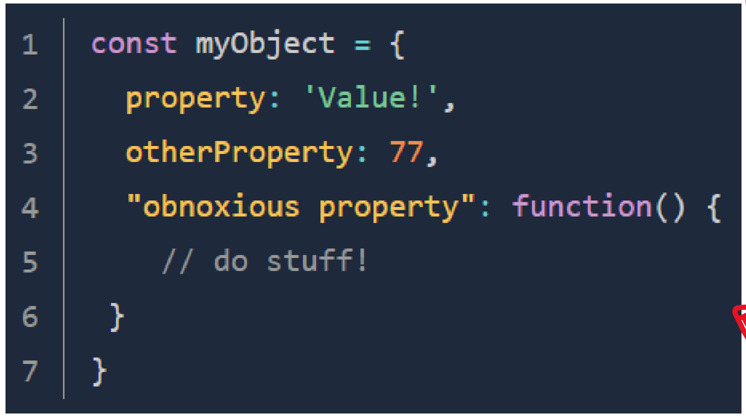
Objects And Object Constructors

Let's remember objects - Objects are a very important part of the JavaScript language, and while for the most part you can accomplish simple and even intermediate tasks without worrying about them, any real project that you're going to attempt is going to feature Objects.

As we know, there are eight data types in JavaScript. Seven of them are called "**primitive**", because their values contain only a single thing (be it a string or a number or whatever).

In contrast, objects are used to store keyed collections of various data and more complex entities. In JavaScript, objects penetrate almost every aspect of the language.

An object can be created with figure brackets **{…}** with an optional list of properties. A property is a **"key: value" pair**, where key is a string (also called a "property name"), and value can be anything.



```
const myObject = {
  property: 'Value!',
  otherProperty: 77,
  "obnoxious property": function() {
    // do stuff!
  }
}
```

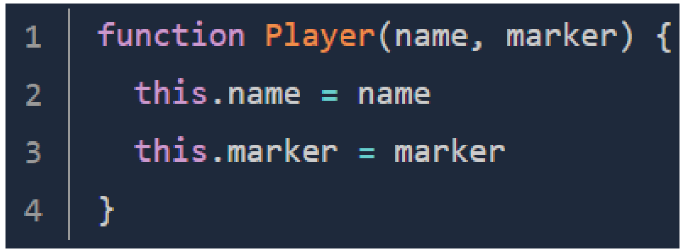
There are multiple ways to define objects:

- using an **object literal.**
- with the **keyword new.**
- Define an **object constructor.**
- Create an object using **Object.create().**

Ex. # 1.0

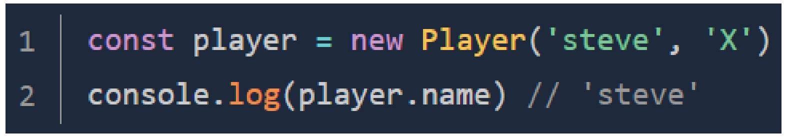
Objects And Object Constructors

Object Constructors - A constructor is a special function that creates and initializes an object instance of a class. In JavaScript, a constructor gets called when an object is created using the **new** keyword.



```
function Player(name, marker) {
  this.name = name
  this.marker = marker
}
```

The purpose of a constructor is to create a new object and set values for any existing object properties.



```
const player = new Player('steve', 'X')
console.log(player.name) // 'steve'
```

When a constructor gets invoked in JavaScript, the following sequence of operations take place:

- A new empty object gets created.
- The **this** keyword begins to refer to the new object and it becomes the current instance object.
- The new object is then returned as the return value of the constructor.

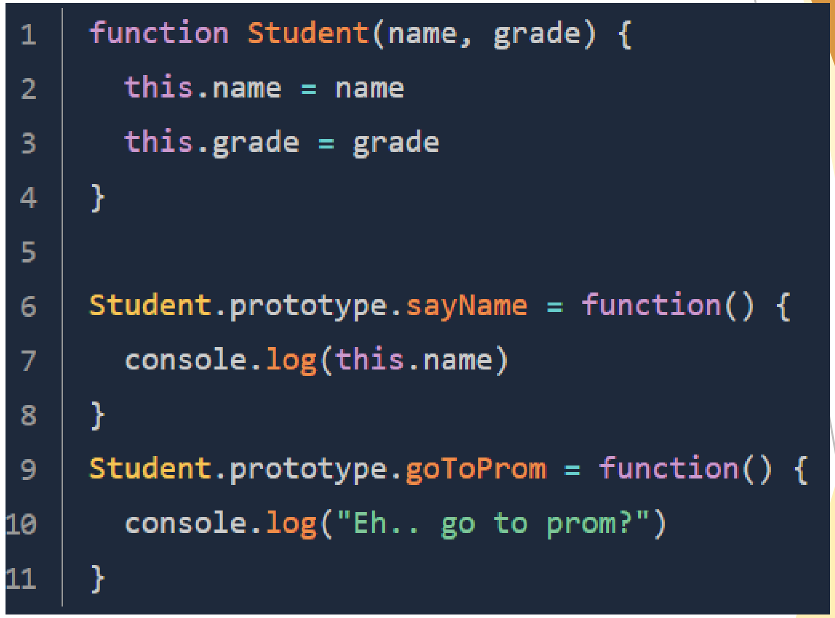
Ex. #
1.1-2

Objects And Object Constructors

The Prototype - All objects in JavaScript have a **prototype**. Stated simply, the **prototype** is another object that the **original object inherits from**, which is to say, **the original object has access to all of its prototype's methods and properties.**

There are two interrelated concepts with prototype in JavaScript:

- First, every JavaScript function has a **prototype property,** and you attach properties and methods on this prototype property when you want to implement inheritance. **(Example #1.1)**
- Second, **prototype attribute.** An object's prototype **attribute points** to the object's "parent"— the object it inherited its properties from. **(Example #1.2)**



```
function Student(name, grade) {
  this.name = name
  this.grade = grade
}

Student.prototype.sayName = function() {
  console.log(this.name)
}
Student.prototype.goToProm = function() {
  console.log("Eh.. go to prom?")
}
```

***If you're using constructors to make your objects it is best to define functions on the prototype of that object. Doing so means that a single instance of each function will be shared between all of the Student objects. If we declare the function directly in the constructor, like we did in the **Example #1.0-1**, that function would be duplicated every time a new Student is created. In this example, that wouldn't really matter much, but in a project that is creating thousands of objects, it really can make a difference.

Ex. # 1.3-4

Objects And Object Constructors

Object.create() - method creates a new object, using an existing object as the prototype of the newly created object.

By now you should have a good idea of what **prototype** is and how it is used. **Object.create()** is an improvement on the whole concept of object creation in JS, it is designed to be more efficient and faster.

Object.create() can take up to **two parameters:**

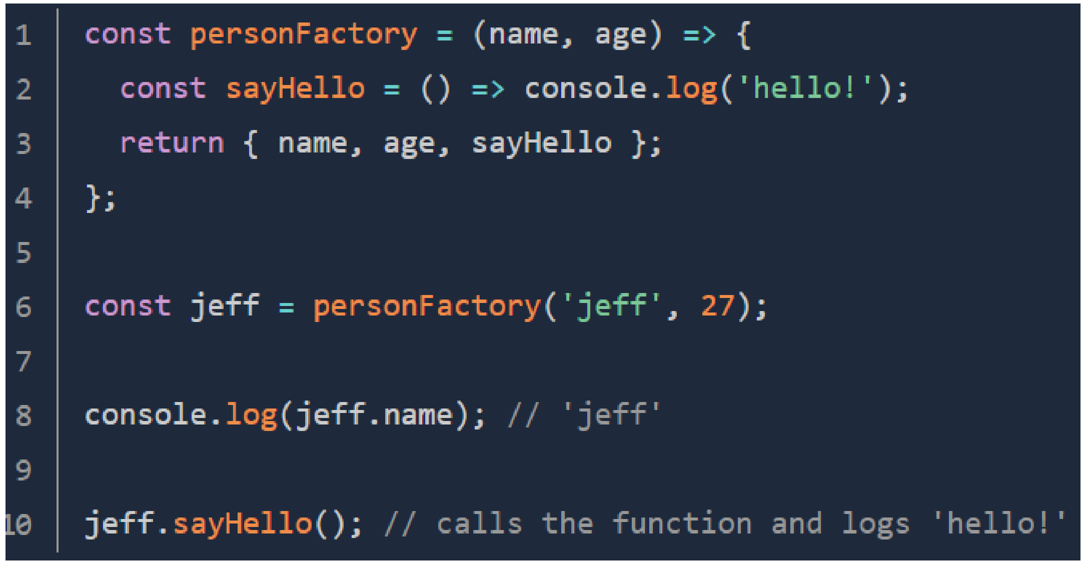
- Object.create(proto) – **Example #1.3**
- Object.create(proto, propertiesObject) - **Example #1.4**

*****Object.prototype Properties Inherited by all Objects -** All objects in JavaScript inherit properties and methods from Object.prototype. These inherited properties and methods are constructor, hasOwnProperty (), isPrototypeOf (), propertyIsEnumerable (), toLocaleString (), toString (), and valueOf (). ECMAScript 5 also adds 4 accessor methods to Object.prototype.

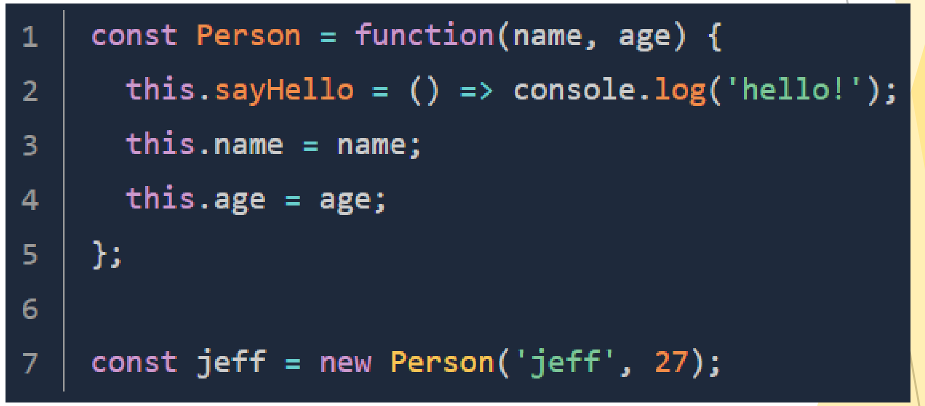
Ex. # 1.5

Factory Functions And The Module Pattern

Factory function - The factory function pattern is similar to constructors, but instead of using **new** to create an object, factory functions simply set up and return the new object when you call the function. Take a look at these example, the first one is of the Factory function while the second one is a **constructor**:

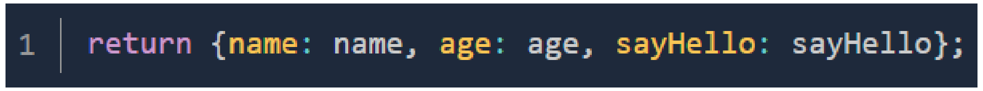


```
1  const personFactory = (name, age) => {
2    const sayHello = () => console.log('hello!');
3    return { name, age, sayHello };
4  };
5
6  const jeff = personFactory('jeff', 27);
7
8  console.log(jeff.name); // 'jeff'
9
10 jeff.sayHello(); // calls the function and logs 'hello!'
```



```
1  const Person = function(name, age) {
2    this.sayHello = () => console.log('hello!');
3    this.name = name;
4    this.age = age;
5  };
6
7  const jeff = new Person('jeff', 27);
```

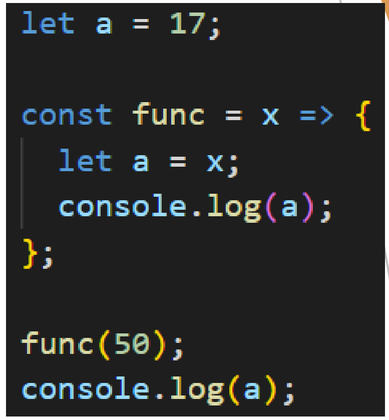
Object Shorthand - A quick note about line 3 from the factory function example. In 2015, a handy new shorthand for creating objects was added into JavaScript**(Example #1.5).** Without the shorthand, line 3 would have looked something like this:



```
1  return {name: name, age: age, sayHello: sayHello};
```

Ex. #
1.6 - 7

Factory Functions And The Module Pattern



```
let a = 17;

const func = x => {
  let a = x;
  console.log(a);
};

func(50);
console.log(a);
```

Scope and Closure - Scope is the term that refers to where things like variables and functions can be used in your code.

The concept of **closure** is the idea that functions retain their scope even if they are passed around and called outside of that scope. See **Example #1.6**

In the context of factory functions, **closures** allow us to **create private variables and functions**. Private functions are functions that are used in the workings of our objects that are not intended to be used elsewhere in our program. In other words, even though our objects might only do one or two things, we are free to split our functions up as much as we want (allowing for a cleaner, easier way to read code) and only export the functions that the rest of the program is going to use. Using this terminology on the **Example #1.6**, **capitalizeString** is a private function and **printString** is public.

The concept of private functions is very useful and should be used as often as is possible! For every bit of functionality that you need for your program, there are likely to be several supporting functions that do NOT need to be used in your program as a whole. Tucking these away and making them inaccessible makes your code easier to refactor, easier to test, and easier to reason about for you and anyone else that wants to use your objects.

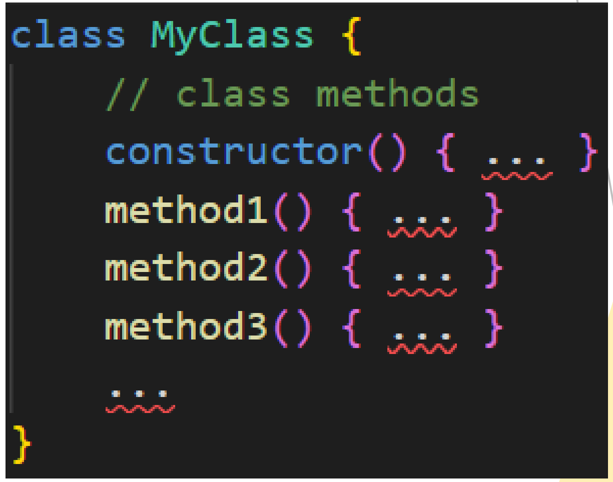
You can take a look at the **Example #1.7** and play a little game!

Classes

JavaScript does not have classes in the same sense as other Object Oriented languages like Java or Ruby. ES6, however, did introduce a syntax for object creation that uses the **class keyword**. It is basically a new syntax that does the exact same thing as the object constructors and prototypes.

There is a bit of controversy about using the **class** syntax, however. Opponents argue that class is basically just syntactic sugar over the existing prototype-based constructors and that it's dangerous and/or misleading to obscure what's really going on with these objects. Despite the controversy, classes are beginning to crop up in real code bases that you are almost certainly going to encounter such as frameworks like React.

In practice, we often need to create many objects of the same kind, like users, or goods or whatever. As we already know from the **Constructor**, operator "**new**", **new** function can help with that. But in the modern JavaScript, there's a more advanced "**class**" construct, that introduces great new features which are useful for object-oriented programming.



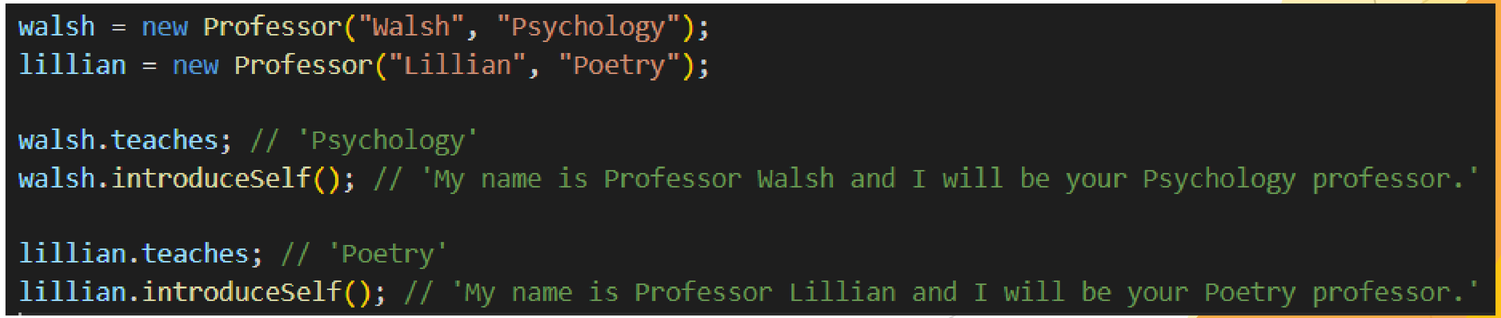
```
class MyClass {
    // class methods
    constructor() { ... }
    method1() { ... }
    method2() { ... }
    method3() { ... }
    ...
}
```

Classes

When we model a problem in terms of objects in OOP, we create abstract definitions representing the types of objects we want to have in our system. For example, if we were modeling a school, we might want to have objects representing professors. Every professor has some properties in common: they all have a name and a subject that they teach. Additionally, every professor can do certain things: they can all grade a paper, and they can introduce themselves to their students at the start of the year, for example.

```
class Professor {
    properties:
        name
        teaches

    methods:
        grade(assignment)
        introduceSelf()

    constructor:
        Professor(name, teaches)
}
```

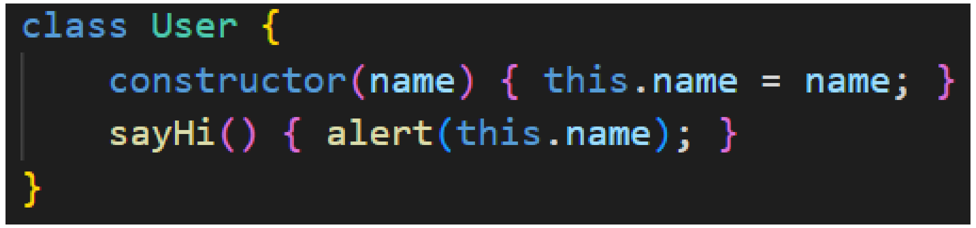
On its own, a class doesn't do anything: it's a kind of template for creating concrete objects of that type. Each concrete **professor** we create is called **an instance of the Professor class**. The process of creating an instance is performed by a special function called a **constructor**. We pass values to the constructor for any internal state that we want to initialize in the new instance.



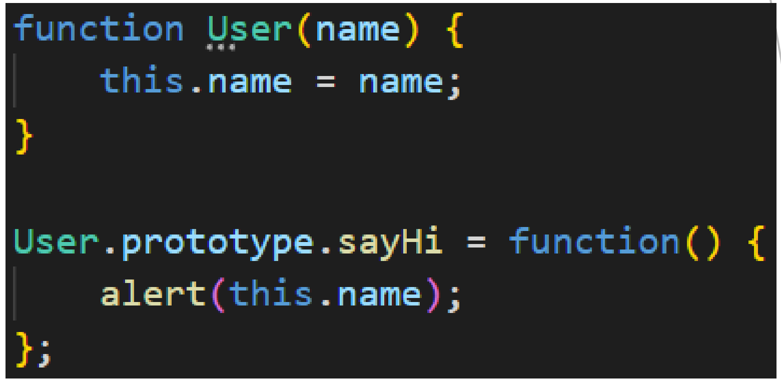
```
walsh = new Professor("Walsh", "Psychology");
lillian = new Professor("Lillian", "Poetry");

walsh.teaches; // 'Psychology'
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

lillian.teaches; // 'Poetry'
lillian.introduceSelf(); // 'My name is Professor Lillian and I will be your Poetry professor.'
```

Classes

Classes vs Constructors - Sometimes people say that **class** is a “**syntactic sugar**” (syntax that is designed to make things easier to read, but doesn’t introduce anything new), because we could actually do the same thing without using the class keyword at all:



```
class User {
    constructor(name) { this.name = name; }
    sayHi() { alert(this.name); }
}
```



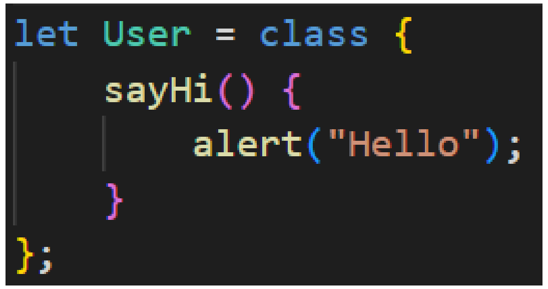
```
function User(name) {
    this.name = name;
}

User.prototype.sayHi = function() {
    alert(this.name);
};
```

The result of this definition is about the same. So, there are indeed reasons why class can be considered a syntactic sugar to define a constructor together with its prototype methods. Still, there are important differences.

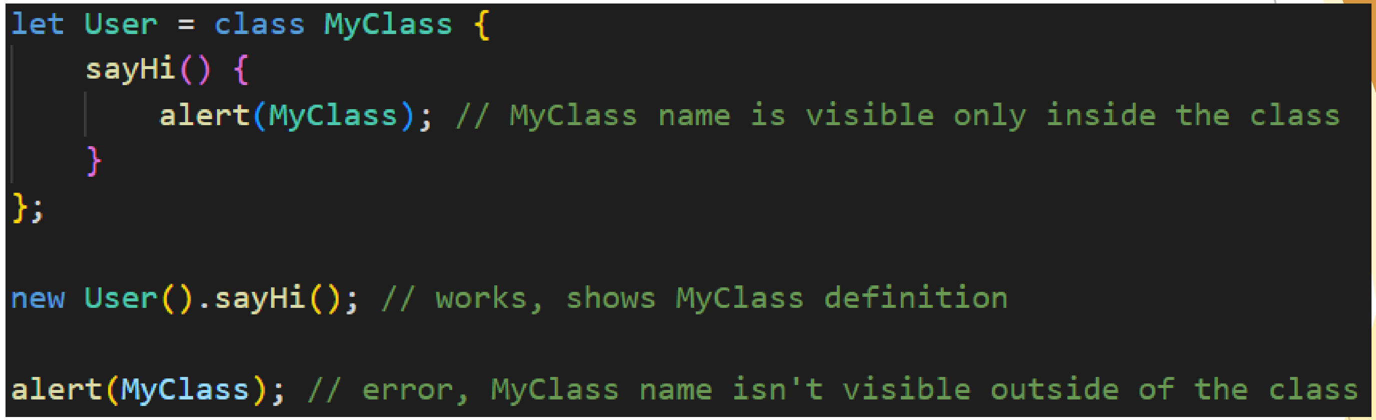
- **First**, a function created by class is labelled by a special internal property **[[IsClassConstructor]]: true**. So it’s not entirely the same as creating it manually.
- **Second**, Class methods are **non-enumerable**. A class definition sets enumerable flag to false for all methods in the "prototype". Meaning **Classes** know how to keep secrets.
- **Third**, **Classes** always use **strict**. All code inside the class construct is automatically in strict mode.

Classes

Class Expression - Just like functions, classes can be defined inside another expression, passed around, returned, assigned, etc. Here are a few examples of a class expression:

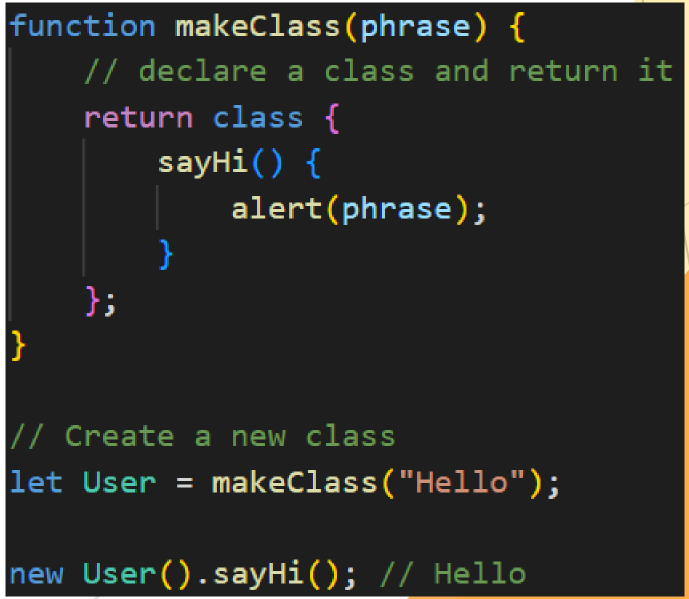


```
let User = class {
    sayHi() {
        alert("Hello");
    }
};
```



```
let User = class MyClass {
    sayHi() {
        alert(MyClass); // MyClass name is visible only inside the class
    }
};

new User().sayHi(); // works, shows MyClass definition

alert(MyClass); // error, MyClass name isn't visible outside of the class
```

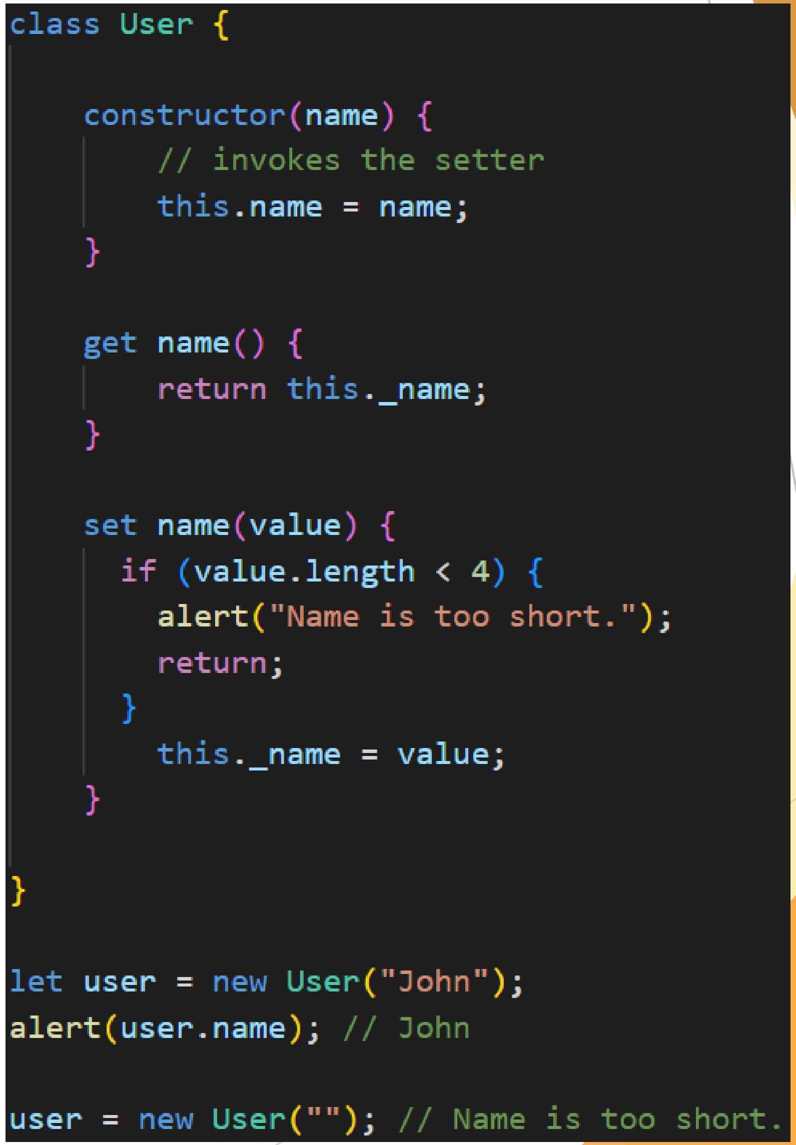
Similar to Named Function Expressions, class expressions may have a name. If a class expression has a name, it's visible inside the class only. We can even make classes dynamically "on-demand", like this:



```
function makeClass(phrase) {
    // declare a class and return it
    return class {
        sayHi() {
            alert(phrase);
        }
    };
}

// Create a new class
let User = makeClass("Hello");

new User().sayHi(); // Hello
```

Classes

Getters/Setters - Just like literal objects, classes may include getters/setters, computed properties etc. Here’s an example for **user.name** implemented using get/set:



```
class User {

    constructor(name) {
        // invokes the setter
        this.name = name;
    }

    get name() {
        return this._name;
    }

    set name(value) {
      if (value.length < 4) {
        alert("Name is too short.");
        return;
      }
        this._name = value;
    }

}

let user = new User("John");
alert(user.name); // John

user = new User(""); // Name is too short.
```

Inheritance

Suppose in our school we also want to represent **students**. Unlike **professors**, **students** can't grade papers, don't teach a particular subject, and belong to a particular year. However, students do have a name and may also want to introduce themselves, so we might write out the definition of a student class like this:

```
class Student
    properties:
        name
        year
    constructor:
        Student(name, year)
    methods:
        introduceSelf()
```

It would be helpful if we could represent the fact that **students** and **professors** share some properties, or more accurately, the fact that on some level, they are the same kind of thing.

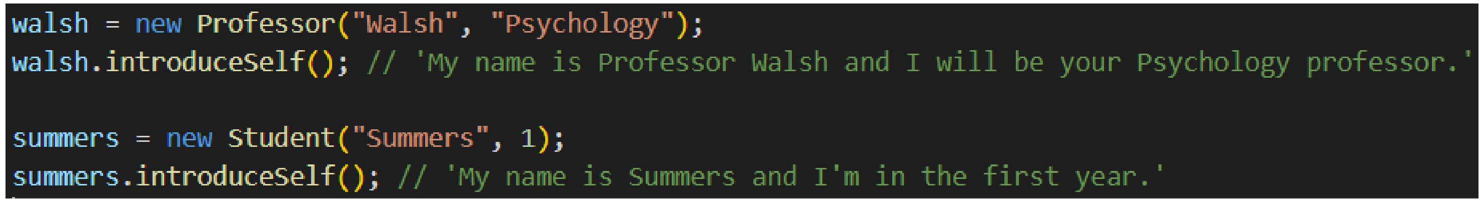
We start by observing that **students** and **professors** are both people, and people have names and want to introduce themselves. We can model this by defining a **new class Person**, where we define all the common properties of people. Then, Professor and Student can both derive from Person, adding their extra properties:

```
class Person
    properties:
        name
    constructor:
        Person(name)
    methods:
        introduceSelf()

class Student : extends Person
    properties:
        year
    constructor:
        Student(name, year)
    methods:
        introduceSelf()

class Professor : extends Person
    properties:
        teaches
    constructor:
        Professor(name, teaches)
    methods:
        grade(assignment)
        introduceSelf()
```

Inheritance

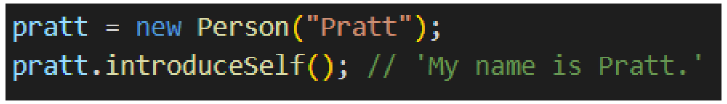
In this case, we would say that Person is the **superclass** or **parent class** of both **Professor** and **Student**. Conversely, **Professor** and **Student** are **subclasses** or **child classes** of **Person**.

You might notice that **introduceSelf()** is defined in all three classes. The reason for this is that while all people want to introduce themselves, the way they do so is different:



```
walsh = new Professor("Walsh", "Psychology");
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

summers = new Student("Summers", 1);
summers.introduceSelf(); // 'My name is Summers and I'm in the first year.'
```

We might have a default implementation of **introduceSelf()** for people who aren't students or professors:



```
pratt = new Person("Pratt");
pratt.introduceSelf(); // 'My name is Pratt.'
```

This feature - when a method has the same name but a different implementation in different classes - is called **polymorphism**. When a method in a subclass replaces the superclass's implementation, we say that the subclass overrides the version in the superclass.

Encapsulation

Objects provide an interface to other code that wants to use them but maintain their own internal state. The object's internal state is kept **private**, meaning that it can only be accessed by the object's own methods, not from other objects. Keeping an object's internal state private, and generally making a clear division between its public interface and its private internal state, is called **encapsulation**.

```
class Student : extends Person
        properties:
                year
        constructor:
                Student(name, year)
        methods:
                introduceSelf()
```

For example, suppose students are allowed to study archery if they are in the second year or above. We could implement this just by exposing the student's year property, and other code could examine that to decide whether the student can take the course.

The problem is, if we decide to change the criteria for allowing students to study archery - for example by also requiring the **parent** or **guardian** to give their permission - we'd need to update every place in our system that performs this test. It would be better to have a method on **Student** objects, that implements the logic in one place.

```
class Student : extends Person
        properties:
                private year
        constructor:
                Student(name, year)
        methods:
                introduceSelf()
                canStudyArchery() {
                        return this.year >1
                }

student = new Student('Weber', 1)
student.year      // error: 'year' is private
property of Student
```

NPM

npm - is the world's largest software registry. Open source developers from every continent use **npm** to share and borrow packages, and many organizations use **npm** to manage private development as well.

npm consists of three distinct components:

- the website
- the Command Line Interface (CLI)
- the registry

Use the **website** to discover packages, set up profiles, and manage other aspects of your **npm** experience. For example, you can set up organizations to manage access to public or private packages.

The **CLI** runs from a terminal, and is how most developers interact with **npm**.

The **registry** is a large public database of JavaScript software and the meta-information surrounding it.

NPM - Installation

Windows **-** Download the Windows Installer from the official website. Make sure you have downloaded the latest version of **NodeJs**. It includes the **NPM** package manager.

Linux - You may need to update Ubuntu to the latest version for the Curl installation to complete. If so, run the command: **sudo apt update && sudo apt upgrade**

To install **nvm** properly, you'll need **curl**. Simply run the bolded command: **sudo apt install curl**

Now run this command which installs **nvm:**
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash

Now restart your terminal, you can verify **nvm** is installed by running the command: **command -v nvm**

if this returns **nvm: command not found**, close the terminal and re-open it.

NPM - Installation

macOS - On macOS 10.15 and above, the default shell is now **zsh**. During installation, **nvm** will look for a **.zshrc** file in your user home directory. By default, this file does not exist so we need to create it.

To create the **.zshrc** file and start the **nvm** installation, run the following commands:

1. **touch ~/.zshrc**

2. **curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash**

Restart your terminal and test your **nvm** installation by running: **nvm --version**

For more information, view NVM’s github documentation.

NPM - Installation

Installing Node - Now that we have **nvm** installed, we can install Node.

Step 1: Installation Run: **nvm install --lts**

This will install the most recent stable version of Node in ‘long-term support’ (LTS), and you’ll see a lot of output in the terminal. If everything worked, you should see something similar to this somewhere in the lines of output (with the X’s replaced with actual numbers): *Downloading and installing Node vXX.xx.x...*

If not, close the terminal, re-open it and run **nvm install --lts** again.

Step 2: Setting the Node Version

We need to tell nvm which version of Node to use when we run the node command. It’s easy; just run the following command: **nvm use --lts**

We have told nvm to use the most recent LTS version of Node installed on our computer. The LTS version of Node is simply a version that is guaranteed support for thirty months after its initial release. It is more stable and compatible with a variety of packages than a non-LTS version of Node.

Now when you run **node -v**, you should see vXX.xx.x or something similar (with the X’s replaced with actual numbers).

Webpack

Webpack is simply a tool for bundling modules. Webpack is a very powerful tool, and with that power comes a decent amount of complexity. Don't let it scare you off! The basic configuration is not difficult and proficiency with webpack looks amazing on resumes.

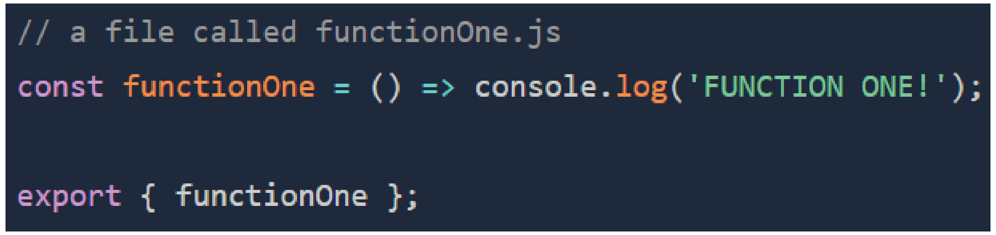
In the email you will receive a **Tutorial link**, code along with all of the steps of this tutorial ("Basic Setup" through "Conclusion").

After installing webpack using **npm**, we set up a simple project that required an external library (**lodash** - check it out here if it's new to you) using a simple script tag. The site lists a few reasons why this is probably not ideal and then steps through using webpack to accomplish the same thing.

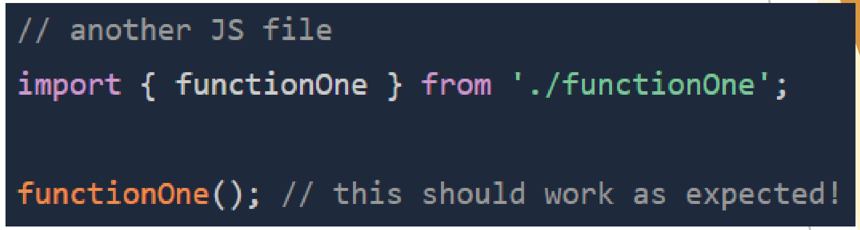
There are a couple of key concepts to understanding how **webpack** works - **entry** and **output**. In the tutorial, we rearranged the files into a **src** and **dist** folder. Technically we could have called those folders anything, but those names are typical. **src is our source directory**. In other words, **src** is where we write all of the code that webpack is going to bundle up for us. When webpack runs, it goes through all of our files looking for any import statements and then compiles all of the code we need to run our site into a single file inside of the **dist** folder (**short for distribution**). Our entry file, then is the main application file that links (either directly or indirectly) to all of the other modules in our project. In the tutorial example, it is **/src/index.js**. The output file is the compiled version - **dist/main.js**.

ES6 Modules

There are only 2 components to it - import and export. The import statement is the same thing that you used during the webpack tutorial!

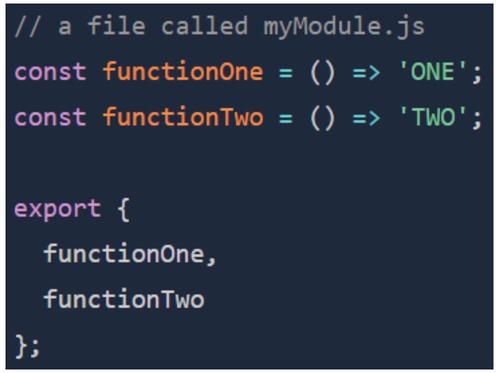


```
// a file called functionOne.js
const functionOne = () => console.log('FUNCTION ONE!');

export { functionOne };
```

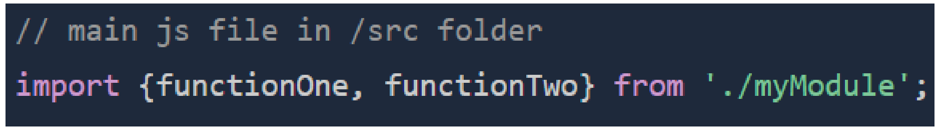


```
// another JS file
import { functionOne } from './functionOne';

functionOne(); // this should work as expected!
```

There are many benefits to writing your code in modules. One of the most compelling is code reuse. There are also the same benefits as when using factory functions or the module pattern (the module pattern and ES6 modules are not the same things; this naming convention is frustrating). With the introduction of ES6 Modules, the module pattern (IIFEs) is not needed anymore, though you might still encounter them. By using ES6 modules, you can keep different parts of your code cleanly separated, which makes writing and maintaining your code much easier and less error-prone. Keep in mind that you can **definitely export constructors, classes** and **factory functions** from your modules.



```
// a file called myModule.js
const functionOne = () => 'ONE';
const functionTwo = () => 'TWO';

export {
  functionOne,
  functionTwo
};
```



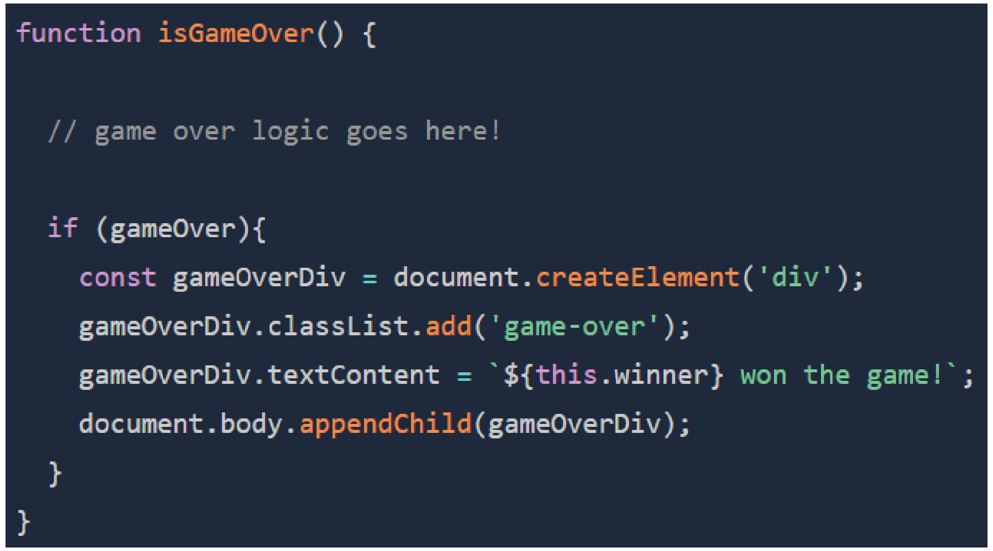
```
// main js file in /src folder
import {functionOne, functionTwo} from './myModule';
```

***Using this pattern gives you the freedom to only import the functions you need in the various files of your program. So it's perfectly fine to only import functionOne if that's the only one you need.

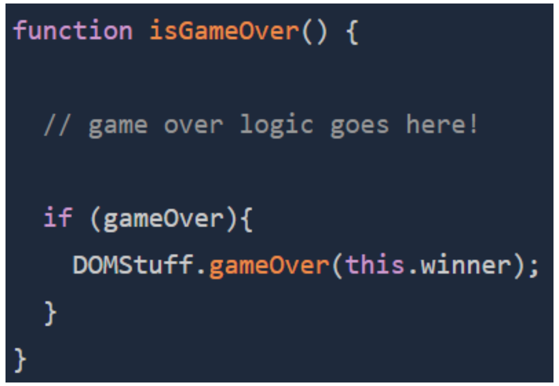
OOP Principles

Single Responsibility - As you craft your objects, one of the most important things to remember is the **Single Responsibility** Principle which states that a **class (or object or module… you get the point) should only have one responsibility.** This doesn't mean that an object can only do one thing, but it does mean that everything an object does should be part of one responsibility.

Most of our code has functions to update and write things to the DOM in addition to our application logic. It's a really good idea to separate your DOM stuff from the application logic. Imagine it from a point of a game:



```
function isGameOver() {

  // game over logic goes here!

  if (gameOver){
    const gameOverDiv = document.createElement('div');
    gameOverDiv.classList.add('game-over');
    gameOverDiv.textContent = `${this.winner} won the game!`;
    document.body.appendChild(gameOverDiv);
  }
}
```



```
function isGameOver() {

  // game over logic goes here!

  if (gameOver){
    DOMStuff.gameOver(this.winner);
  }
}
```

***In fact - the function **isGameOver** shouldn't be calling the DOM function anyway. That should go elsewhere (directly in the game-loop). **Single Responsibility** Principle is that a given method/class/component should have a single reason to change. Otherwise, if an object is trying to have multiple responsibilities, changing one aspect might affect another.

Assignment

branko.fd.br@gmail.com

Thank you for your attention!