ES6 Part 2

IMY 220 ● Lecture 9

ES6 – Promises

Promises give us a way to deal with asynchronous behaviour, such as timeouts, AJAX calls, etc.

“A Promise is an object representing the eventual completion or failure of an asynchronous operation”

“Essentially, a Promise is a returned object to which you attach callbacks, instead of passing callbacks into a function”

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

ES6 – Promises

The Promise object constructor is as follows

```
const examplePromise = new Promise(function(resolve, reject){
      // asynchronous operation goes here

      // if everything goes right:
      resolve(optionalReturnData);

      //if something goes wrong:
      reject(otherReturnData);
});
```

ES6 – Promises

How it works (broadly): instead of passing callback functions for a function to call when it finishes an asynchronous function (or fails)…

```
// example: loadUserDetails fetches user data with an AJAX call
loadUserDetails(userid, successCallback, failCallback);
```

…the asynchronous function returns a *Promise*, which is dealt with when it finishes (or fails)

```
// in this example, we’re assuming loadUserDetails returns a
// promise (we’ll look at returning promises next)
loadUserDetails(userid).then(successCallback, failCallback);
```

ES6 – Promises

The resolve parameter in the Promise corresponds to the first parameter (i.e. the “success-callback”) in the then function

```
const promise1 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 1000);
});

promise1.then(() => {alert("Success")}, () => {alert("Fail")});
```

This code will alert “Success” after waiting one second

(since we don’t call reject in the Promise, it will never alert “Fail”)

ES6 – Promises

We can also send back data when calling resolve

```
const promise1 = new Promise(function(resolve, reject) {
  setTimeout(resolve("Success"), 1000);
});

promise1.then(data => {alert(data)}, () => {alert("Fail")});
```

ES6 – Promises

And if we call reject, it corresponds to the second parameter (i.e. the “error-callback”) in the then function

```
const promise1 = new Promise(function(resolve, reject) {
  setTimeout(reject, 1000);
});

promise1.then(() => {alert("Success")}, () => {alert("Fail")});
```

This code will alert “Fail” after waiting one second

ES6 – Promises

One of the useful features of Promises is how it deals with chaining of asynchronous events.

Previously, if we wanted to call asynchronous events after one another, we would have to pass them as callbacks to each other

```
firstAsyncFunction(function(result){
      secondAsyncFunction(result, function(secondResult){
            thirdAsyncFunction(secondResult, function(thirdResult){
                  console.log(`This is not ideal: ${thirdResult}`);
            });
      });
});
```

ES6 – Promises

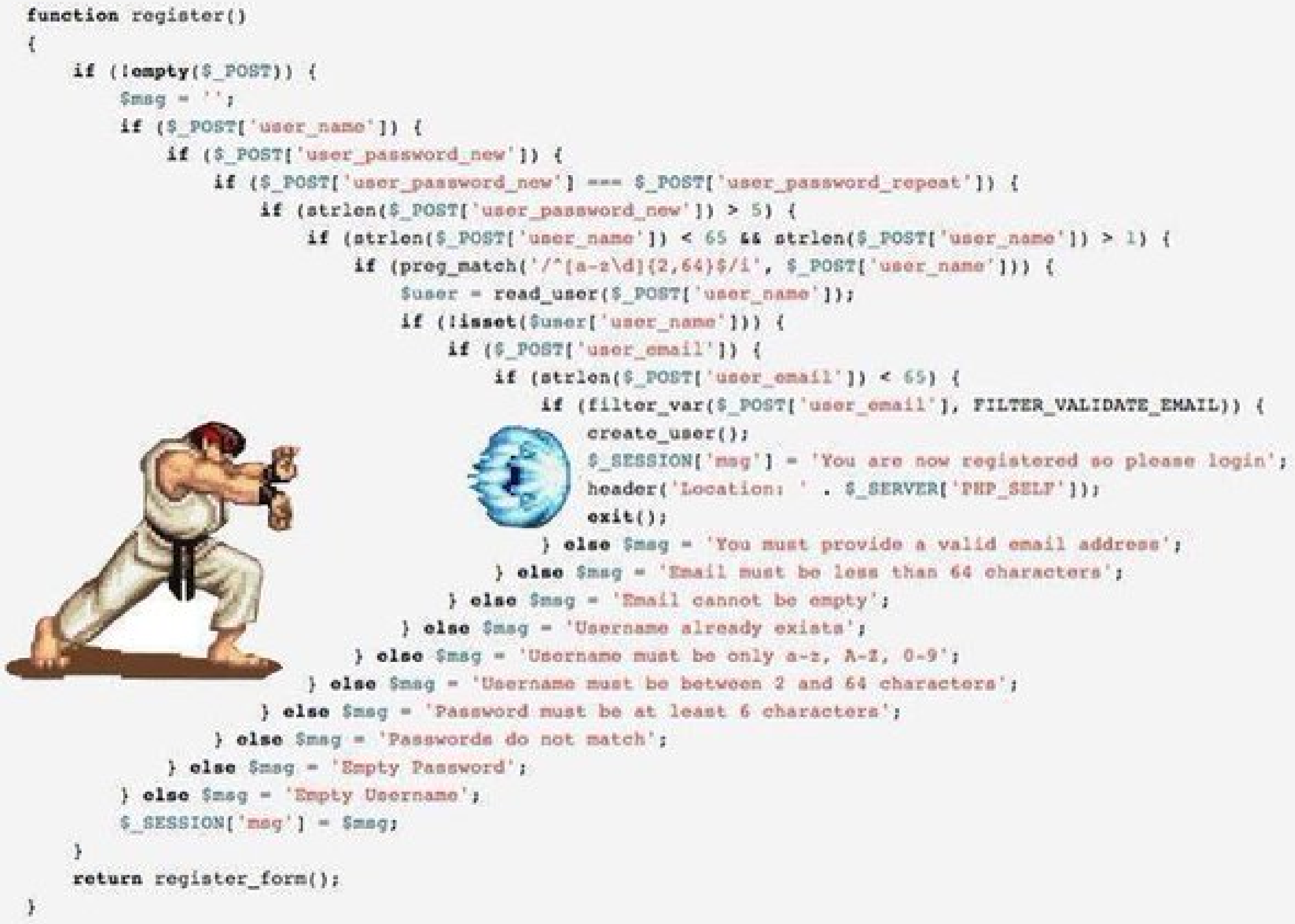
So, generally speaking, when function1 finishes, it calls function2, which calls function3, etc.

Due to the way JS used to deal with asynchronous requests, calling each function as the previous function’s callback was the only way to ensure that asynchronous functions finish in sequence

This is known as the “callback pyramid of doom” or “callback hell”

http://callbackhell.com/

Classic pyramid of doom



```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

ES6 – Promises

Promises allow us to do this in a neater, more intuitive way due to the way it deals with chaining

```
firstAsyncFunction().then(function(result){
      return secondAsyncFunction(result);
})
.then(function(secondResult){
      return thirdAsyncFunction(secondResult);
})
.then(function(thirdResult){
      console.log(`This is a lot better: ${thirdResult}`);
});
```

ES6 – Promises

Or even shorter with arrow functions

```
firstAsyncFunction()
.then(result => secondAsyncFunction(result))
.then(secondResult => thirdAsyncFunction(secondResult))
.then(thirdResult => {
      console.log(`This is a lot better: ${thirdResult}`);
});
```

Each .then function returns another promise, which allows us to chain them together

ES6 – Promises

If we’re referring to an existing function, we only need to pass the pointer to the function when it resolves or rejects

```
const p = new Promise((res, rej) => {
      setTimeout(res, 1000);
});
const sayYes = function(){
      alert('Done');
}
p.then(sayYes);
```

ES6 – Async/await

Technically part of ES7

Different way of dealing with asynchronous events

In other words, similar situations for what you’d want to use Promises for…

…BUT async/await is not the same as Promises

ES6 – Async

The async keyword is always used with a function like this

```
async function load(){
      return 1;
}

// Or with an arrow function expression

const load = async () => {
      return 1;
}
```

ES6 – Async

The use of the async keyword does two things:

- The function always returns a Promise, which can be called with then
- The function allows the use of the await functionality

ES6 – Async

```
const f = async () => {
   return 1;
}

// can be used the same as
const f = () => {
   return new Promise((res, rej) => {
      res(1);
   })
}

// Both can be used as follows
// (note that you have to execute the function in both cases)
f().then(x => alert(x));
```

ES6 – Await

The await keyword is used to get the value from a Promise

Using await actually pauses execution of the script until the Promise returns a value

You can only use await inside an async function

ES6 – Await

This example alerts “Done” after 1 second

```
const myPromise = new Promise((resolve, reject) => {
        setTimeout(() => { resolve("Done") }, 1000);
});

const callAsync = async () => {
        let value = await myPromise;
        // execution pauses here until myPromise resolves
        alert(value);
}

callAsync();
```

ES6 – Await

Alternatively, if we want to use await without having to create and call a named function, we can wrap it in an anonymous self-invoking function

```
// assuming myPromise is declared here

( async () => {
       let value = await myPromise;
       // execution pauses here until myPromise resolves
       alert(value);
})();
```

ES6 – Await

Let’s look at the difference between await and then

Given the following Promise declaration…

```
const setValue = () => {
        return new Promise((res, rej) => {
                setTimeout(()=>{res("bar")}, 500);
        });
}
```

…we’re going to call it using await and then

ES6 – Await

First, using await

```
(async () => {
        let val = "foo";
        val = await setValue();

        alert(val);
})();
```

Result: ???

ES6 – Await

First, using await

```
(async () => {
      let val = "foo";
      val = await setValue();

      alert(val);
})();
```

Result: bar

Execution is paused until setValue resolves

ES6 – Await

Using then

```
(async () => {
        let val = "foo";
        setValue().then(x => {val = x});

        alert(val);
})();
```

Result: ???

ES6 – Await

Using then

```
(async () => {
        let val = "foo";
        setValue().then(x => {val = x});

        alert(val);
})();
```

Result: foo

While setValue was completing in the background, execution continues and val is alerted before it is changed

ES6 – Promises + async/await

We’ll revisit promises and async/await when we deal with AJAX

ES6 – Classes

Previously, OOP in JS was done with the use of functions

(Note that class names are always capitalised, as per convention)

```
function Car(make, model, year) {
       this.make = make;
       this.model = model;
       this.year = year;
}

Car.prototype.noise = function(){
       alert("Toot!");
}

var newCar = new Car("Toyota", "Corolla", 2014);

newCar.noise();
```

ES6 – Classes

In ES6, the class keyword was introduced.

```
class Car{
        constructor(make, model, year){
                this.make = make;
                this.model = model;
                this.year = year;
        }

        // no need to use this.noise = … when defining member functions
        noise(){
                 alert("Toot!");
        }
}

const newCar = new Car("Toyota", "Corolla", 2014);

newCar.noise();
```

ES6 – Classes

However, it still works the same way. Car is still a function and the instance of Car, newCar, is still an object

```
const newCar = new Car("Toyota", "Corolla", 2014);

console.log(newCar);
// Object { make: "Toyota", model: "Corolla", year: 2014 }

console.log(Car);
// Car()
// length: 3
// name: "Car"/
// prototype: Object { … }
// <prototype>: function ()
```

The syntax from the previous slide does not allow you to do anything new, but it makes more sense from a classical OOP perspective

ES6 – Classes

The new syntax also comes with some new keywords, which allow you to create classes in a classical OOP manner

```
class Rectangle {
        constructor(height, width) {
                this._height = height;
                this._width = width;
        }
        // Getter
        get area() {
                return this._height * this._width;
        }
        // Setter
        set height(height){
                this._height = height;
                console.log(`Height has been changed to: ${height}`);
        }
}
```

ES6 – Classes

The above class definition can be used to create and change an instance of a class like this

```
const square = new Rectangle(10, 10);

console.log(square.area);
// Output: 100

square.height = 5;
// Output: Height has been changed to: 5
// (Note that the const keyword does not prevent changing object values)

console.log(square.area);
// Output: 50
```

ES6 – Classes

“The constructor method is a special method for creating and initializing an object created with a class.

There can only be one special method with the name "constructor" in a class.

A SyntaxError will be thrown if the class contains more than one occurrence of a constructor method.”

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

ES6 – Classes

Getters and setters (get and set) allow you to get and set variables and functions directly, without invoking functions on class instances. They are called whenever a property of an instance is accessed

In our example above, we called get area() when we logged the value of square.area

Similarly, we called set height() when we set the value of

square.height = 5;

ES6 – Classes

Note that setting a value inside the class definition also invokes a setter, which is why you can’t do the following:

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

        //Setter
        set height(height){
                this.height = height;
                // setting height here calls the setter, which calls the
                // setter, i.e. infinite recursion
        }
}

const square = new Rectangle(10, 10);
```

ES6 – Classes

Getter and setters are good places to parse values into usable outputs and validate incoming input. This example checks that height is a positive number

```
set height(height){
        try {
                if(isNaN(height - parseFloat(height)))
                        throw 'Non-numeric height input';
                else if (height < 0)
                        throw 'Negative height input';
                else
                        this._height = height;
        }

        catch(error){
                console.log(`Error while setting height: ${error}`);
        }
}
```

ES6 – Classes

ES6 classes also support inheritance. Child classes inherit all functions and properties from parents

```
class Rectangle{
   constructor(length, height, name){
      this._length = length;
      this._height = height;
      this._name = name;
   }

   calcArea(){
      return this._length * this._height;
   }

   getArea(){
      return `${this._name}: ${this.calcArea()}`;
   }
}
```

```
class Square extends Rectangle{
   constructor(length, name){
      super(length, length, name);
   }

   calcArea(){
      return Math.pow(this._length, 2);
   }

   getArea(){
      return super.getArea();
   }
}

const square = new Square(10, "square1");
```

The super keyword is used to call functions on an object’s parent, in this case, the Rectangle class’ constructor and two member functions: calcArea() and getArea()

ES6 – Classes

ES6 also provides support for static methods, which “*aren't called on instances of the class. Instead, they're called on the class itself. These are often utility functions, such as functions to create or clone objects.*”

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static

```
class Person{
     static sayHello(){
          console.log(`Hello there!`);
     }
}

Person.sayHello();
```

ES6 – Classes

Static methods don’t have access to the properties and functions of a class instance through the this keyword

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

        makeGreeting(){
                return `Hello ${this._name}`;
        }

        static sayHello(){
                console.log(this._name);           // Neither of these
                console.log(this.makeGreeting());  // will work
        }
}
```

ES6 – Classes

To call a static method from inside a class, you need to call them using the class name or by calling the method as a property of the constructor

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

        static makeGreeting(name){
                return `Hello ${name}`;
        }

        sayHello(){
                console.log(Person.makeGreeting(this._name));
        }
}
const person = new Person("Diffie");
person.sayHello();  // Output: Hello Diffie
```

ES6 – Classes

Static methods also support inheritance

```
class FriendlyPerson extends Person{
        constructor(name){
                super(name);
        }

        static makeGreeting(name){
                return `${super.makeGreeting(name)}, how are you?`;
        }

        sayHello(){
                super.sayHello();
        }
}
const person = new FriendlyPerson("Diffie");
person.sayHello();  // Output: ???
```

ES6 – Classes

Static methods also support inheritance

```
class FriendlyPerson extends Person{
        constructor(name){
                super(name);
        }

        static makeGreeting(name){
                return `${super.makeGreeting(name)}, how are you?`;
        }

        sayHello(){
                super.sayHello();
        }
}
const person = new FriendlyPerson("Diffie");
person.sayHello();  // Output: Hello Diffie
                    // (Because sayHello calls Person.makeGreeting)
```

ES6 – Classes

Static methods also support inheritance

```
class FriendlyPerson extends Person{
        constructor(name){
                super(name);
        }

        static makeGreeting(name){
                return `${super.makeGreeting(name)}, how are you?`;
        }

        sayHello(){
                console.log(FriendlyPerson.makeGreeting(this._name));
        }
}
const person = new FriendlyPerson("Diffie");
person.sayHello();  // Output: Hello Diffie, how are you?
```

ES6 – Modules

“A JavaScript module is a piece of reusable code that can easily be incorporated into other JavaScript files”

Learning React

Until ES6, the only way to do this was to use libraries that could import and export modules, like the module.exports functionality found in node.js

ES6 – Modules

When we talk about a single module, we are referring to a single file that exports some *type*

A single file can export one or more objects that contain any JavaScript type, such as objects, functions (which are objects anyway), classes (which are also actually functions), primitives, and arrays

```
// myModule.js
export const print = message => console.log(message);
```

```
<!-- index.html -->
<script type="module">
      import {print} from './myModule.js';
      print("Cool beans");
      // Output: Cool beans
</script>
```

ES6 – Modules

You need to set the type attribute in the script tag to “module” for the module to be imported successfully

The import keyword only supports absolute URLs, so you need to prepend ./ to the file name for files that are in the same directory

There are two ways to export modules: *named* and *default*

ES6 – Modules

With named exports, all exported types are given a unique name which *have* to correspond with the name(s) used to import it again

```
// myModule.js
export const print = message => console.log(message);

export const addYass = name => `Yass ${name}!`;
```

```
// index.html (assuming you have the correct script tags)
import {print, addYass} from './myModule.js';

print(addYass("Diffie"));
// Output: Yass Diffie!
```

ES6 – Modules

In this example, attempting to import print and addYass using different names will give an error, for example:

```
import {log, sayYass} from './myModule.js’;           // won't work
```

However, you can scope module variables under different variable names

```
// index.html (assuming you have the correct script tags)
import {print as log, addYass as sayYass} from './myModule.js';

// Now you can use the different names to access the module

log(sayYass("Diffie"));
// Output: Yass Diffie!
```

ES6 – Modules

Another way to import is to save everything from a file to an object:

```
import * as myStuff from './myModule.js';

myStuff.print(myStuff.addYass("Diffie"));
```

Exporting classes works exactly the same way, including inherited classes

```
// Square.js
export class Square extends Rectangle{
// Class definition goes here
}
```

```
// index.html (assuming you have the correct script tags)
import {Square} from './Square.js';
// Do something with Square
```

ES6 – Modules

Default exports are used to export one object as a (default) variable/function/etc., for example:

```
// names.js
const names = ["Jake", "Amy", "Charles", "Rosa", "Raymond", "Terry"];
export default names;
```

When working with default variables, we don’t have to use the same names as in the module file, since the exported type doesn’t have a name

```
// index.html (assuming you have the correct script tags)
import people from './names.js';
// in this example, "people" can be anything

console.log(people[4]);
// Output: Raymond
```

ES6 – Modules

Note that the example from the previous slide presents two bad practices

The first is creating global variables which all files can access

The second is that using default exports does not require consistent naming while importing modules, which can lead to problems with refactoring and tree-shaking (i.e. removal of useless code)

Some advocate strongly against the use of default exports:

https://blog.neufund.org/why-we-have-banned-default-exports-and-you-should-do-the-same-d51fdc2cf2ad

ES6 – Modules

We can also include modules inside other modules

(This example assumes the files are all in the same directory)

```
// names.js
export const names = ["Jake", "Amy", "Charles", "Rosa", "Raymond"];
```

```
// myModule.js
import {names} from './names.js';
export const addYass = num => `Yass ${names[num]}!`;
export const print = message => console.log(message);
```

```
// index.html (assuming you have the correct script tags)
import {print, addYass} from './myModule.js';
print(addYass(4));
// Output: Yass Raymond!
```

References

Banks, A. & Porcello, E. 2017. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, Inc.

https://developer.mozilla.org/

https://github.com/lukehoban/es6features/blob/master/README.md

https://javascript.info/async-await

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await