HTML5, CSS3 & JS

Chapter 6

ES6 Features

Chapter Objectives

In this chapter, we will discuss:

- ES5 Revisit
- New ES6 Syntax
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

ES5 Revisit

- var - has function scope
- this keyword - points to the owner of the function in which it is used
- 4 patterns of invoking functions
- Binding with .bind()
- Closures
- JS Array functions

Function invocation patterns

- Function invocation Pattern

```
var func = function() {
    // ...
  };

func()
```

Function invocation patterns

- Method invocation Pattern

```
var frog = {
    RUN_SOUND: "POP!!",
    run: function() {
      return this.RUN_SOUND;
    }
  };

  frog.run();
  // returns "POP!!"
```

Function invocation patterns

- Constructor invocation Pattern

```
function Wizard() {
  this.castSpell = function() { return "KABOOM"; };
}

var merlin = new Wizard();
merlin.castSpell() // returns "KABOOM";
```

Function invocation patterns

- Call & Apply invocation Pattern

```
function addAndSetX(a, b) {
  this.x += a + b;
}

var obj1 = { x: 1, y: 2 };

addAndSetX.call(obj1, 1, 1);
// this = obj1, obj1 after call = { x: 3, y : 2 }
// It is the same as:
// addAndSetX.apply(obj1, [1, 1]);
```

bind()

- Using bind()

```
let user = {
  firstName: "John"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
```

Array functions

- map() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- find() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find
- findIndex() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex
- filter() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- concat() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b
- slice() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice
- splice() => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

Chapter Concepts

- **const & let**
- Arrow functions
- Default function parameters
- Rest parameter
- Destructuring assignment
- Template Literals
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

const & let

- JavaScript ES6 comes with two more ways to declare variables: `const` and `let`. They have block scope unlike `var` which has function scope.
- A variable declared with `const` cannot be re-assigned or re-declared, and cannot be changed or modified. Once the variable is assigned, you cannot change it

```
// not allowed
const helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';


// allowed
let helloWorld = 'Welcome to the Road to learn React';
helloWorld = 'Bye Bye React';
```

const & let

- A variable declared directly with const cannot be modified. However, when the variable is an array or object, the values it holds can get updated through indirect means:

```
// allowed
const helloWorld = {
  text: 'Welcome to the Road to learn React'
};
helloWorld.text = 'Bye Bye React';
```

Chapter Concepts

- const & let
- **Arrow functions**
- Default function parameters
- Rest parameter
- Destructuring assignment
- Template Literals
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

Arrow functions

- JavaScript ES6 introduced arrow functions expressions, which are shorter than a function expres- sions.

```
// function declaration
function () { ... }

// arrow function declaration
() => { ... }
```

Arrow functions

- You can remove the parentheses in an arrow function expression if it only has one argument, but you have to keep the parentheses if it gets multiple arguments:

```
// allowed
item => { ... }

// allowed
(item) => { ... }

// not allowed
item, key => { ... }

// allowed
(item, key) => { ... }
```

Arrow functions

- You can remove the block body, the curly braces, with the ES6 arrow function.

```
{list.map(item =>
  <div key={item.objectID}>
    <span>
      <a href={item.url}>{item.title}</a>
    </span>
    <span>{item.author}</span>
    <span>{item.num_comments}</span>
    <span>{item.points}</span>
  </div>
)}
```

Chapter Concepts

- const & let
- Arrow functions
- **Default function parameters**
- Rest parameter
- Destructuring assignment
- Template Literals
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

Default Params

- A function that automatically provides default values for undeclared parameters can be a beneficial safeguard for your programs

```
function link(height = 50, color = 'red', callbackFn = () => {}) {
  // function content...
}
```

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- **Rest parameter**
- Destructuring assignment
- Template Literals
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

Rest Parameters

- Rest arguments **COLLAPSE** all remaining arguments of a function into a single array.

```
function stringStuffTogether(...whateverTheArgsAre) {
  return whateverTheArgsAre.join(' ');
}

stringStuffTogether('this', 'is', 'so', 'much', 'cleaner', 'WOOT!!!')
```

Spread operator

- ES6 array spread operator

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']

const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

Spread operator

- Object spread operator

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }

const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const userAge = { age: 28 };
const user = { ...userNames, ...userAge };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

Summary

- When we see "..." in the code, it is either rest parameters or the spread operator.
- There's an easy way to distinguish between them:
  - When ... is at the end of function parameters, it's "rest parameters" and gathers the rest of the list of arguments into an array.
  - When ... occurs in a function call or alike, it's called a "spread operator" and expands an array into a list.

Use patterns

- Rest parameters are used to create functions that accept any number of arguments.
- The spread operator is used to pass an array to functions that normally require a list of many arguments.

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- Rest parameter
- **Destructuring assignment**
- Template Literals
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

Destructuring - Objects

- Destructuring in JavaScript ES6 provides easier access to properties in objects and arrays.

```
const user = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch
```

```
// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch
```

Destructuring - Objects

- For forming new Objects based on existing ones

```
const member = {
  name: 'Ben',
  title: 'software developer',
  skills: ['javascrip:t', 'react', 'redux'],
};

const memberWithMetadata = {
  ...member,
  previousProjects: ['BlueSky', 'RedOcean'];
};
```

Destructuring - Objects

- For passing objects in function calls

```
let jane = { firstName: 'Jane', lastName: 'Doe'};
let john = { firstName: 'John', lastName: 'Doe', middleName: 'Smith' }
function sayName({firstName, lastName, middleName = 'N/A'}) {
  console.log(`Hello ${firstName} ${middleName} ${lastName}`)
}

sayName(jane) // -> Hello Jane N/A Doe
sayName(john) // -> Helo John Smith Doe
```

Destructuring - Arrays

- The same concept applies to arrays. You can destructure them, too, again using multilines to keep your code scannable and readable.

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// output: Robin Andrew Dan
```

Destructuring - Arrays

- For forming new arrays based on existing ones

```
const fruits = ['apple', 'banana'];
const veggies = ['cucumber', 'potato'];

const food = ['grapes', ...fruits, ...veggies];
// -> ["grapes", "apple", "banana", "cucumber", "potato"]
```

Destructuring - Arrays

- For function calls

```
const food = ['grapes', 'apple', 'banana', 'cucumber', 'potato'];
function eat() {
  console.log(...arguments);
}

eat(...food)
// -> grapes apple banana cucumber potato
```

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- Rest parameter
- Destructuring assignment
- **Template Literals**
- ES6 Modules
- ES6 Class
- Promises
- ES6 Collections

Template Literals

- Traditionally text enclosed in double quotes/ single quotes was considered as string and also it all was to be in a single line with no provision to add dynamic values. This led to long concatenated strings.

```
var name = 'Sam';
var age = 42;

console.log('hello my name is ' + name + ' I am ' + age + ' years old');
//= 'hello my name is Sam I am 42 years old'
```

Template Literals

- ES6 introduces new tyle of string literal with back ticks (`)

```
var name = 'Sam';
var age = 42;

console.log(`hello my name is ${name}, and I am ${age} years old`);
//= 'hello my name is Sam, and I am 42 years old'
```

- It works with either member expressions or method calls

```
var name = 'Sam';
var age = 42;

console.log(`hello my name is ${name.toUpperCase()}, and I am ${age / 2} years old`);
//= 'hello my name is SAM, and I am 21 years old'
```

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- Rest parameter
- Destructuring assignment
- Template Literals
- **ES6 Modules**
- ES6 Class
- Promises
- ES6 Collections

Modules

- In JavaScript ES6, you can import and export functionalities from modules.
- These can be functions, classes, components, constants, essentially anything you can assign to a variable.
- Modules can be single files or whole folders with one index file as entry point.
- The import and export statements help you to share code across multiple files.
- These statements embrace code splitting, where we distribute code across multiple files to keep it reusable and maintainable.
- We also want to think about code encapsulation, since not every functionality needs to be exported from a file.

Modules

- The act of exporting one or multiple variables is called a named export

```
const firstname = 'Robin';
const lastname = 'Wieruch';

export { firstname, lastname };
```

- And import them in another file with a relative path to the first file.

```
import { firstname, lastname } from './file1.js';

console.log(firstname);
// output: Robin
```

Modules

- You can also import all exported variables from another file as one object.

```
import * as person from './file1.js';

console.log(person.firstname);
// output: Robin
```

- Imports can have aliases, which are necessary when we import functionalities from multiple files that have the same named export.

```
import { firstname as username } from './file1.js';

console.log(username);
// output: Robin
```

Modules

- There is also the default statement, which can be used for a few cases:
  - to export and import a single functionality
  - to highlight the main functionality of the exported API of a module
  - to have a fallback import functionality
- You have to leave out the curly braces to import the default export.

```
const robin = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

export default robin;
```

```
import developer from './file1.js';

console.log(developer);
// output: { firstname: 'Robin', lastname: 'Wieruch' }
```

Modules

- The import name can differ from the exported default name, and it can be used with the named export and import statements:

```
const firstname = 'Robin';
const lastname = 'Wieruch';

const person = {
  firstname,
  lastname,
};

export {
  firstname,
  lastname,
};

export default person;
```

Modules

- Import the default or the named exports in another file:

```
import developer, { firstname, lastname } from './file1.js';

console.log(developer);
// output: { firstname: 'Robin', lastname: 'Wieruch' }
console.log(firstname, lastname);
// output: Robin Wieruch
```

Modules

- You can spare the extra lines, and export the variables directly for named exports.

```
export const firstname = 'Robin';
export const lastname = 'Wieruch';
```

Folder modules

- The folder acting as a module should have index.js file.
- The index.js file naming is convention was introduced in the Node.js world, where the index file is the entry point to a module.
- It describes the public API to the module.
- External modules are only allowed to use the index.js file to import shared code from the module.

Folder modules

- Consider the following module structure to demonstrate it:

```
src/
  index.js
  App/
    index.js
  Buttons/
    index.js
    SubmitButton.js
    SaveButton.js
    CancelButton.js
```

Folder modules

- The Buttons/ folder has multiple button components defined in its distinct files. Each file can export default the specific component, making it available to Buttons/index.js. The Buttons/index.js file imports all different button representations and exports them as public module API. (src/Buttons/index.js)
-)

```
import SubmitButton from './SubmitButton';
import SaveButton from './SaveButton';
import CancelButton from './CancelButton';

export {
  SubmitButton,
  SaveButton,
  CancelButton,
};
```

Folder modules

- Now the src/App/index.js can import the buttons from the public module API located in the index.js file. (src/App/index.js)

```
import {
  SubmitButton,
  SaveButton,
  CancelButton
} from '../Buttons';
```

- However, it can be seen as bad practice to reach into other files than the index.js in the module, as it breaks the rules of encapsulation.

```
// bad practice, don't do it
import SubmitButton from '../Buttons/SubmitButton';
```

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- Rest parameter
- Destructuring assignment
- Template Literals
- ES6 Modules
- **ES6 Class**
- Promises
- ES6 Collections

ES6 Classes

- While React embraces functional programming, e.g. immutable data structures and function compositions, classes are used to declare ES6 class components. React mixes the good parts of both programming paradigms.

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}
```

ES6 Classes

- A class has a constructor to make it instantiable. The constructor takes arguments and assigns them to the class instance.
- A class can also define functions. Because the function is associated with a class, it is called a method, or a class method.
- Class can be instantiated.

```
const robin = new Developer('Robin', 'Wieruch');
console.log(robin.getName());
// output: Robin Wieruch
```

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- Rest parameter
- Destructuring assignment
- Template Literals
- ES6 Modules
- ES6 Class
- **Promises**
- ES6 Collections

Promises

- Promises are inbuilt in ES6

```
const wait (ms) => {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
  });
}

wait(1000).then(() => console.log('tick'));
```

Promises vs. callbacks

- For HTTP calls our existing solution was to use callbacks

```
request(url, (error, response) => {
  // handle success or error.
});
doSomethingElse();
```

- A few problems exists with callbacks. One of them being callback hell.

Promises vs. callbacks

- A callback pattern requires us to specify the task and the callback at the same time. In contrast promises allow us to specify and dispatch the request in one place.

```
promise = fetch(url); //fetch is a replacement for XMLHttpRequest
```

- And then add a callback later in a different place

```
promise.then(response => {
  // handle the response.
});
```

Promises vs. callbacks

- This also allows us to attach multiple handlers to the same task

```
promise.then(response => {
  // handle the response.
});
promise.then(response => {
  // do something else with the response.
});
```

- .then() always returns a promise

Catching Rejections

- Second argument function to .then() is the error handler function. Note that one catch at the end is often enough

```
fetch('http://ngcourse.herokuapp.com/api/v1/tasks')
  .then(response => response.data)
  .then(tasks => filterTasksAsynchronously(tasks))
  .then(tasks => {
    $log.info(tasks);
    vm.tasks = tasks;
  })
  .then(
    null,
    error => log.error(error)
  );
```

Chapter Concepts

- const & let
- Arrow functions
- Default function parameters
- Rest parameter
- Destructuring assignment
- Template Literals
- ES6 Modules
- ES6 Class
- Promises
- **ES6 Collections**

Sets & Maps

- Map
  - A map is a collection of key and value pairs

- Set:
  - A Set is a collection of unique elements

```
const arr = [5, 1, 5, 7, 7, 5];
const unique = [...new Set(arr)]; // [ 5, 1, 7 ]
```