Lesson - 30

ES6

SkillUp by Dobrea Vladislav

Lesson Plan

- HW Review
- Webpack dev server
- ES6

Webpack dev server

webpack-dev-server can be used to quickly develop an application. See the development guide to get started.

webpack.config.js

```
var path = require('path');

module.exports = {
  //...
  devServer: {
    contentBase: path.join(__dirname, 'dist'),
    compress: true,
    port: 9000
  }
};
```

When the server is started, there will be a message prior to the list of resolved modules:

```
http://localhost:9000/
webpack output is served from /build/
Content not from webpack is served from /path/to/dist/
```

Clean dist/build folder

```
npm install --save-dev clean-webpack-plugin
```

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
+ const { CleanWebpackPlugin } = require('clean-webpack-plugin');

  module.exports = {
    entry: {
      app: './src/index.js',
      print: './src/print.js',
    },
    plugins: [
+     new CleanWebpackPlugin(),
      new HtmlWebpackPlugin({
        title: 'Output Management',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
  };
```

EcmaScript

ECMAScript 6 is also known as ES6 and ECMAScript 2015.

Some people call it JavaScript 6.

This chapter will introduce some of the new features in ES6.

- JavaScript let
- JavaScript const
- JavaScript Arrow Functions
- JavaScript Classes
- Default parameter values
- Array.find()
- Array.findIndex()
- Exponentiation (**) (EcmaScript 2016)
- …etc

Javascript let

The let statement allows you to declare a variable with block scope.

```
var x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
```

It's not possible with var

```
var x = 10;
// Here x is 10
{
  var x = 2;
  // Here x is 2
}
// Here x is 2
```

Javascript const

The const statement allows you to declare a constant (a JavaScript variable with a constant value).

Constants are similar to let variables, except that the value cannot be changed.

```
var x = 10;
// Here x is 10
{
  const x = 2;
  // Here x is 2
}
// Here x is 10
```

Arrow functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

```
// ES5
var x = function(x, y) {
   return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Arrow functions

Arrow functions do not have their own this. They are not well suited for defining **object methods**.

Arrow functions are not hoisted. They must be defined **before** they are used.

Using const is safer than using var, because a function expression is always constant value.

You can only omit the return keyword and the curly brackets if the function is a single statement. Because of this, it might be a good habit to always keep them:

```
const x = (x, y) => { return x * y };
```

Classes

ES6 introduced classes.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

Use the keyword class to create a class, and always add a constructor method.

The constructor method is called each time the class object is initialized.

```
class APOD {
    constructor(id) {
      this.id = id;
      this.date = undefined;
      let body = document.body;
      this.container = document.createElement('div');
      this.container.setAttribute('id', this.id);
      this.dateFormatter = dateFormatter;
      this.renderApp = renderApp;
      this.addElement = addElement;
      this.addElement();
      body.append(this.container);
    }
}

const apod = new APOD('apod-container')
```

Array find()

The find() method returns the value of the first array element that passes a test function.

This example finds (returns the value of) the first element that is larger than 18:

```
var numbers = [4, 9, 16, 25, 29];
var first = numbers.find(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Array findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

```
var numbers = [4, 9, 16, 25, 29];
var first = numbers.findIndex(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

Default parameters

ES6 allows function parameters to have default values.

```
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

Rest parameters

A function can be called with any number of arguments, no matter how it is defined.

Like here:

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted.

Rest parameters

The rest of the parameters can be included in the function definition by using three dots `...` followed by the name of the array that will contain them. The dots literally mean "gather the remaining parameters into an array".

For instance, to gather all arguments into array `args`:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Rest parameters

⚠️ **The rest parameters must be at the end**

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
  // error
}
```

The `...rest` must always be last.

Spread syntax

We've just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

For instance, there's a built-in function Math.max that returns the greatest number from a list:

```
alert( Math.max(3, 5, 1) ); // 5
```

Now let's say we have an array `[3, 5, 1]`. How do we call `Math.max` with it?

Passing it "as is" won't work, because `Math.max` expects a list of numeric arguments, not a single array:

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

Spread syntax

And surely we can't manually list items in the code Math.max(arr[0], arr[1], arr[2]), because we may be unsure how many there are. As our script executes, there could be a lot, or there could be none. And that would get ugly.

Spread syntax to the rescue! It looks similar to rest parameters, also using ..., but does quite the opposite.

When ...arr is used in the function call, it "expands" an iterable object arr into the list of arguments.

For Math.max:

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

Get a new copy of object/array

Remember when we talked about `Object.assign()` in the past?

It is possible to do the same thing with the spread syntax.

```
let arr = [1, 2, 3];
let arrCopy = [...arr]; // spread the array into a list of parameters
                        // then put the result into a new array

// do the arrays have the same contents?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true

// are the arrays equal?
alert(arr === arrCopy); // false (not same reference)

// modifying our initial array does not modify the copy:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Template literals

Intuitive expression interpolation for single-line and multi-line strings. (Notice: don't be confused, Template Literals were originally named "Template Strings" in the drafts of the ECMAScript 6 language specification)

```
var customer = { name: "Foo" }
var card = { amount: 7, product: "Bar", unitprice: 42 }

var message = `Hello ${customer.name},
want to buy ${card.amount} ${card.product} for
a total of ${card.amount * card.unitprice} bucks?`
```

```
var customer = { name: "Foo" };
var card = { amount: 7, product: "Bar", unitprice: 42 };
var message = "Hello " + customer.name + ",\n" +
"want to buy " + card.amount + " " + card.product + " for\n" +
"a total of " + (card.amount * card.unitprice) + " bucks?";
```

Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
const foo = ['one', 'two', 'three'];

const [red, yellow, green] = foo;
console.log(red); // "one"
console.log(yellow); // "two"
console.log(green); // "three"
```

Destructuring assignment

Object destructuring:

```
function renderApp(data) {
    const {picture, copyright, title, description} = data;
    console.log(picture);
    console.log(title);
    console.log(description);
}
```

Browser Support

Browser Support for ES6 (ECMAScript 2015)

Safari 10 and Edge 14 were the first browsers to fully support ES6:

Chrome 58	Edge 14	Firefox 54	Safari 10	Opera 55
Jan 2017	Aug 2016	Mar 2017	Jul 2016	Aug 2018

Home work

Visit this link to find lesson homework:

https://docs.google.com/document/d/1DjHA57xV3AiSd-IycKfBWCg5V9YR69e6hfdj07MuMo8/edit?usp=sharing

Learning Resources

1. **EcmaScript Features:**
    1. http://es6-features.org/
    2. https://www.w3schools.com/js/js_es6.asp
    3. https://javascript.info/rest-parameters-spread