APPENDIX B

■ ■ ■

JavaScript 2015

JavaScript 2015 (Also named ECMAScript 6, or simply ES6) is the newer version of the JavaScript Language. It adds significant new syntax for writing complex applications, including classes, modules, new variable declaration keywords and promises. It also includes new helpers and syntactic sugar functionalities for more expressive code, such as: arrow functions, template strings and destructuring.

Classes

JavaScript classes are introduced in ECMAScript 6 and are syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is not introducing a new object-oriented inheritance model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance.

Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

```
class Point {

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

    toString() {
        return `(${this.x}, ${this.y})`;
    }

}

class Pixel extends Point {

    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }

    toString() {
        return super.toString() + ' in ' + this.color;
    }
```

```
}

const p = new Pixel(25, 8, 'green');
p.toString(); // (25, 8) in green
```

Under the hood, ES6 classes are not something that is radically new: They mainly provide more convenient syntax to create old-school constructor functions. You can see that if you use typeof:

typeof Point // 'function'

Modules

Modules are one of the most important features of any programming language. Modules are first class citizens in ES6. An ES6 module is a file containing JS code. There's no special module keyword; a module mostly reads just like a script, with the addition of the export keyword.

Export

Let's talk about export first. Everything declared inside a module is local to the module, by default. If you want something declared in a module to be public, so that other modules can use it, you must export that feature.

```
function generateRandom() {
    return Math.random();
}

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

export { generateRandom, sum }
```

You can export any top-level function, class, var, let, or const.

Import

The import statement is used to import functions, objects or primitives that have been exported from an external module, another script, etc.

```
import { generateRandom, sum } from 'utility';

console.log(generateRandom()); //logs a random number
console.log(sum(1, 2)); //3
```

Default Exports

A default export can be used to export a single value from the module:

```
var utils = {
  generateRandom: function() {
    return Math.random();
  },
  sum: function(a, b) {
    return a + b;
  }
};

export default utils;
```

Importing is simply a matter of using the name of the exported default:

```
import utils from 'utility';
console.log(utils.generateRandom()); //logs a random number
console.log(utils.sum(1, 2)); //3
```

Let & Const

The "var" declaration has a characteristic that can be potentially dangerous: It only has local scope inside functions, as illustrated in the following sample code:

```
function foo(){
  var myVariable="10";
}

foo()
console.log(myVariable)
// As expected, it throws a "Not defined" error - the variable is scoped to the function.
```

When declared inside other block constructs, the variable gets global scope:

```
if(true){
  var myVariable="10";
}

console.log(myVariable)
// Logs "10": myVariable is not scoped to the "if" block - it's global.
```

Let

ES6 introduces two new keywords for declaring variables: let and const. Both constructs are block-scoped binding constructs. let is the new var.

```
if(true){
  let myVariable="10";
}

console.log(myVariable) // throws a "Not defined" error - the variable is scoped to the if
block.
```

It's generally safe to use let anywhere you would use var.

Const

The const declaration is for single-assignment values: it creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned.

```
const MY_FAV = 7; // define MY_FAV as a constant and give it the value 7
MY_FAV = 20; // Throws a "Assignment to constant variable" error.
```

Promises

Promises are a library for asynchronous programming. Promises are a first class representation of a value that may be made available in the future. Promises are used in many existing JavaScript libraries.

```
function timeout(duration = 0) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, duration);
    })
}

var p = timeout(1000).then(() => {
    return timeout(2000);
}).then(() => {
    throw new Error("hmm");
}).catch(err => {
    return Promise.all([timeout(100), timeout(200)]);
})
```

Arrow Functions

Arrows are a function shorthand using the => syntax. It serves two main purposes: more concise syntax and sharing lexical this with the parent scope.

Concise syntax

Classical JavaScript function syntax doesn't provide for any flexibility, be that a 1 statement function or an unfortunate multi-page function. Every time you need a function you have to type out the dreaded function () {}. More concise function syntax is one of the main benefits of the arrow function syntax:

```
// Simple example
setInterval(() => console.log("Time is passing"),1000);

// One-liner (Implicit return)
let square = (num) => num * num;
console.log(square(5)) // Returns 25


// Multiple lines
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

// Multiple lines with implicit return
let actors = ['Adam West', 'Michael Keaton', 'Val Kilmer', 'George Clooney', 'Christian Bale',
'Ben Affleck']
actors.map((actor)=>(
  actor + ' was Batman!\n'
));
```

Lexical binding

Each function in JavaScript defines its own this context, which is as easy to get around as it is annoying. Arrow functions lexically binds the value of this to the parent scope where it was declared:

```
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

Template Strings

Strings in JavaScript have been historically limited, lacking the capabilities one might expect coming from languages like Python or Ruby.

ES6 Template Strings fundamentally change that. They introduce a syntactic sugar for constructing strings.

Syntax

Template Strings use back-ticks (``) rather than the single or double quotes we're used to with regular strings. A template string could thus be written as follows:

```
var greeting = `Yo World!`;
```

So far, Template Strings haven't given us anything more than normal strings do. Let's change that.

String Substitution

One of their first real benefits is string substitution. Substitution allows us to take any valid JavaScript expression (including say, the addition of variables) and inside a Template Literal, the result will be output as part of the same string.

Template Strings can contain placeholders for string substitution using the ${} syntax, as demonstrated below:

```
// Simple string substitution
var name = "Brendan";
console.log(`Yo, ${name}!`);

// => "Yo, Brendan!"
```

Multiline Strings

Multiline strings in JavaScript have required hacky workarounds for some time.

Template Strings significantly simplify multiline strings. Simply include newlines where they are needed. Here's an example:

```
console.log(`string text line 1
string text line 2`);
```

Destructuring assignment

The destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects into distinct variables.

```
// Array matching
var a, b, rest;
[a, b] = [1, 2]
console.log(a) // 1
console.log(b) // 2

//Object matching
var robotA = { name: "Bender" };
var robotB = { name: "Flexo" };

var { name: nameA } = robotA;
var { name: nameB } = robotB;

console.log(nameA);
// "Bender"
console.log(nameB);
// "Flexo"

// Can be used in parameter position
function g({name: x}) {
  console.log(x);
}
g({name: 5})
```