ES6

COEN 161

What is ECMAScript?

- ECMAScript is the scripting language that forms the basis of JavaScript
- ECMAScript standardized by the ECMA International standards organization in the ECMA-262 specification

What is ECMAScript 5?

- ECMAScript 5 is also known as ES5 and ECMAScript 2009
- These were some new features released in 2009:
  - The "use strict" Directive
  - String.trim()
  - Array.isArray()
  - Array.forEach()
  - Array.map()
  - Array.indexOf()
  - Array.lastIndexOf()
  - JSON.parse()
  - JSON.stringify()
  - Date.now()

Browser Support for ES5 (ECMAScript 5)

- Chrome 23, IE 10, and Safari 6 were the first browsers to fully support ECMAScript 5

				
Chrome 23	IE10 / Edge	Firefox 21	Safari 6	Opera 15
Sep 2012	Sep 2012	Apr 2013	Jul 2012	Jul 2013

What is ECMAScript 6?

- ECMAScript 6 is also known as ES6 and ECMAScript 2015
- Some of the new features in ES6 include:
  - JavaScript let
  - JavaScript const
  - Exponentiation (**)
  - Default parameter values
  - Array.find()
  - Array.findIndex()

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

ES6 let and const

- ES6 introduced two important new JavaScript keywords: `let` and `const`
- These two keywords provide **Block Scope** variables (and constants) in JavaScript
- Before ES6, JavaScript had only two types of scope: **Global Scope** and **Function Scope**

Global Scope

- Variables declared **Globally** (outside any function) have **Global Scope**

```
var carName = "Volvo";

// code here can use carName

function myFunction() {
  // code here can also use carName
}
```

Function Scope

- Variables declared **Locally** (inside a function) have **Function Scope**

```
// code here can NOT use carName

function myFunction() {
  var carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

JavaScript Block Scope

- Variables declared with the **`var`** keyword can **not** have **Block Scope**

```
{
  var x = 2;
}
// x CAN be used here
```

JavaScript Block Scope

- Before ES6 JavaScript did not have **Block Scope**
- Variables declared with the `let` keyword can have Block Scope

```
{
  let x = 2;
}
// x can NOT be used here
```

Redeclaring Variables

- Redeclaring a variable using the var keyword can impose problems
- Example

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

Redeclaring Variables

- Redeclaring a variable using the let keyword can solve this problem
- Example

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

Loop Scope

- Using `var` in a loop, the variable declared in the loop redeclares the variable outside the loop

```
var i = 5;
for (var i = 0; i < 10; i++) {
  // some statements
}
// Here i is 10
```

Loop Scope

- Using `let` in a loop, the variable declared in the loop does not redeclare the variable outside the loop

```
let i = 5;
for (let i = 0; i < 10; i++) {
  // some statements
}
// Here i is 5
```

Function Scope

- Variables declared with var and let are quite similar when declared inside a function, they will both have **Function Scope**

```
function myFunction() {
  var carName = "Volvo";   // Function Scope
}

function myFunction() {
  let carName = "Volvo";   // Function Scope
}
```

Global Scope

- Variables declared with var and let are quite similar when declared outside a block, they will both have **Global Scope**

```
var x = 2;        // Global scope

let x = 2;        // Global scope
```

Global Variables in HTML

- With JavaScript, the global scope is the JavaScript environment
- In HTML, the global scope is the window object
- Global variables defined with the **`var`** keyword belong to the window object

```
var carName = "Volvo";
// code here can use window.carName
```

- Global variables defined with the **`let`** keyword do not belong to the window object

```
let carName = "Volvo";
// code here can not use window.carName
```

Redeclaring

- Redeclaring a JavaScript variable with **`var`** is allowed anywhere in a program

```
var x = 2;

// Now x is 2

var x = 3;

// Now x is 3
```

Redeclaring

- Redeclaring a **`var`** variable with **`let`**, in the same scope, or in the same block, is not allowed

```
var x = 2;         // Allowed
let x = 3;         // Not allowed

{
  var x = 4;    // Allowed
  let x = 5    // Not allowed
}
```

Redeclaring

- Redeclaring a `let` variable with `let`, in the same scope, or in the same block, is not allowed

```
let x = 2;        // Allowed
let x = 3;        // Not allowed

{
  let x = 4;    // Allowed
  let x = 5;    // Not allowed
}
```

Redeclaring

- Redeclaring a **`let`** variable with **`var`**, in the same scope, or in the same block, is not allowed

```
let x = 2;        // Allowed
var x = 3;        // Not allowed

{
  let x = 4;    // Allowed
  var x = 5;    // Not allowed
}
```

Redeclaring

- Redeclaring a variable with **`let`**, in another scope, or in another block, is allowed

```
let x = 2;        // Allowed

{
  let x = 3;   // Allowed
}

{
  let x = 4;   // Allowed
}
```

Hoisting

- Variables defined with **`var`** are hoisted to the top
- You can use a variable before it is declared

```
// you CAN use carName here

var carName;
```

Hoisting

- Variables defined with **`let`** are **not** hoisted to the top
- Using a **`let`** variable before it is declared will result in a **`ReferenceError`**

```
// you can NOT use carName here

let carName;
```

ES6 const

- Variables defined with **`const`** behave like **`let`** variables, except they cannot be reassigned

```
const PI = 3.141592653589793;

PI = 3.14;      // This will give an error

PI = PI + 10;   // This will also give an error
```

Block Scope

- Declaring a variable with `const` is similar to `let` when it comes to **Block Scope**

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

Assigned when Declared

- JavaScript `const` variables must be assigned a value when they are declared
- Incorrect

```
const PI;

PI = 3.14159265359;
```

- Correct

```
const PI = 3.14159265359;
```

Not Real Constants

- The keyword `const` is a little misleading
- It does NOT define a constant value. It defines a constant reference to a value
- Because of this, we cannot change constant primitive values, but we can change the properties of constant objects

Primitive Values

- If we assign a primitive value to a constant, we cannot change the primitive value

```
const PI = 3.141592653589793;

PI = 3.14;      // This will give an error

PI = PI + 10;   // This will also give an error
```

Constant Objects can Change

- You can change the properties of a constant object

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
```

Constant Objects can Change

- But you can NOT reassign a constant object

```
const car = {type:"Fiat", model:"500", color:"white"};

car = {type:"Volvo", model:"EX60", color:"red"}; // ERROR
```

Constant Arrays can Change

- You can change the elements of a constant array

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
```

Constant Arrays can Change

- But you can NOT reassign a constant array

```
const cars = ["Saab", "Volvo", "BMW"];

cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

Redeclaring

- Redeclaring a JavaScript **`var`** variable is allowed anywhere in a program

```
var x = 2;     //  Allowed

var x = 3;     //  Allowed

x = 4;         //  Allowed
```

Redeclaring

- Redeclaring or reassigning an existing **`var`** or **`let`** variable to **`const`**, in the same scope, or in the same block, is not allowed

```
var x = 2;           // Allowed
const x = 2;         // Not allowed
{
  let x = 2;       // Allowed
  const x = 2;     // Not allowed
}
```

Redeclaring

- Redeclaring or reassigning an existing `const` variable, in the same scope, or in the same block, is not allowed

```
const x = 2;        // Allowed
const x = 3;        // Not allowed
x = 3;              // Not allowed
var x = 3;          // Not allowed
let x = 3;          // Not allowed
{
  const x = 2;    // Allowed
  let x = 3;      // Not allowed
}
```

Redeclaring

- Redeclaring a variable with **`const`**, in another scope, or in another block, is allowed

```
const x = 2;        // Allowed

{
  const x = 3;   // Allowed
}

{
  const x = 4;   // Allowed
}
```

Hoisting

- Variables defined with **`var`** are hoisted to the top
- You can use a **`var`** variable before it is declared

```
carName = "Volvo";     // You CAN use carName here
var carName;
```

- Variables defined with **`const`** are not hoisted to the top
- A **`const`** variable cannot be used before it is declared

```
carName = "Volvo";     // You can NOT use carName here
const carName = "Volvo";
```

Default Parameter Values

- 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
```

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, therefore, it might be a good habit to always keep them

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

Lexical this

- "Arrow functions do not have their own `this`…"
- Example

ES6 Classes

- JavaScript classes, are primarily syntactic sugar over JavaScript's existing prototype-based inheritance
- The class syntax does not introduce a new object-oriented inheritance model to JavaScript
- Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components:
  - class declarations
  - class expressions

Class Declarations

- To declare a class, you use the class keyword with the name of the class

```
class Rectangle {

  constructor(height, width) {

    this.height = height;

    this.width = width;

  }

}
```

Class Declaration Hoisting

- Unlike function declarations, class declarations are **not** hoisted
- You first need to declare your class and then access it, otherwise code like the following will throw a **`ReferenceError`**

```
const p = new Rectangle(); // ReferenceError

class Rectangle {}
```

Class Expressions

- A less common way to define a class is with a **class expression**
- Class expressions can be named or unnamed

```
// unnamed
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: "Rectangle"
```

Class Expressions

- The name given to a named class expression is local to the class's body
- It can be retrieved through the class's (not an instance's) name property

```
// named
let Rectangle = class Rectangle2 {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: "Rectangle2"
```

Class body and method definitions

- The body of a class is the part that is in curly brackets `{}`
- This is where you define class members, such as methods or constructor
- The `constructor` method is a special method for creating and initializing an object created with a `class`

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

Instance Properties

- Instance properties must be defined inside of class methods

```
class Rectangle {

  constructor(height, width) {

    this.height = height;

    this.width = width;

  }

}
```

Prototype Methods

- Prototype methods can be written straight into the class body

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
```

Sub Classes

- The **`extends`** keyword is used in class declarations or class expressions to create a class as a child of another class

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

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}
```

Sub Classes

- A constructor can use the **`super`** keyword to call the constructor of the super class

```
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor
  }

  speak() {
    console.log(this.name + ' barks.');
  }
}
```

ES5 Classes vs ES6 Classes

- Because the ES6 class syntax is just syntactic sugar over ES5 classes, they are both still the same classes
- Example

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

Animal.prototype.speak = function () {
  console.log(this.name + ' makes a noise.');
}
```

ES5 Classes vs ES6 Classes

- ES6 classes can even extend traditional, function-based "classes"

```
class Dog extends Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' barks.');
  }
}
```

Enhanced Complex Types

- ES6 introduced more flexible ways to work with complex types
- The **property shorthand** is a shorter syntax for setting the values of properties in objects
- ES5

```
var x = 0, y = 0;
obj = { x: x, y: y };
```

- ES6

```
var x = 0, y = 0
obj = { x, y }
```

Destructuring Assignment

- ES6 now allows us to create variables from complex types if we know the structure of that type
- We can assign specific indices in arrays to individual variables
- ES5

```
var list = [ 1, 2, 3 ];
var a = list[0], b = list[2];
```

- ES6

```
var list = [ 1, 2, 3 ]
var [ a, , b ] = list
```

Destructuring Assignment

- We can assign specific properties in objects to individual variables
- ES5

```
var tmp = { op: '+', lhs: 1, rhs: 2 };
var op  = tmp.op;
var lhs = tmp.lhs;
var rhs = tmp.rhs;
```

- ES6

```
var { op, lhs, rhs } = { op: '+', lhs: 1, rhs: 2 };
```

Modules - Import/Export

- ES6 added support for exporting/importing values from individual JS "modules" without having to add them to the **global namespace**

```
// lib/math.js
LibMath = {}; // global variable
LibMath.sum = function (x, y) { return x + y };
LibMath.pi = 3.141593;
//  someApp.js
var math = LibMath;
console.log("2π = " + math.sum(math.pi, math.pi));
//  otherApp.js
var sum = LibMath.sum, pi = LibMath.pi;
console.log("2π = " + sum(pi, pi));
```

Modules - Import/Export

- ES6 added support for exporting/importing values from individual JS "modules" without having to add them to the **global namespace**

```
//  lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593
//  someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))
//  otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

Resources

https://www.w3schools.com/js/js_es5.asp

https://www.w3schools.com/js/js_es6.asp

http://es6-features.org/

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