ES6

IMY 220 ● Lecture 2.2

(Brief) history of JavaScript

Released in 1995

Was initially used to add some (relatively simple) interactivity to web pages

Became more robust with technologies like DHTML and AJAX

Now: JS is *everywhere*.

You can build full-stack applications using mostly JavaScript node js

React

ANGULARJS
by Google

BACKBONE.JS

Vue.js

ember

ECMAScript

Committee in charge of making changes to JS: European Computer Manufacturer’s Association (ECMA)

Anyone can propose changes to JS by writing a proposal to the ECMA committee

ECMAScript

Big update in ES specification is ECMAScript 6

aka ES6, ES2015, ES6Harmony

(Newest update is actually ES8, but since ES6 brought about a large amount of new features and syntactical specifications, we will be focusing on those)

ECMAScript

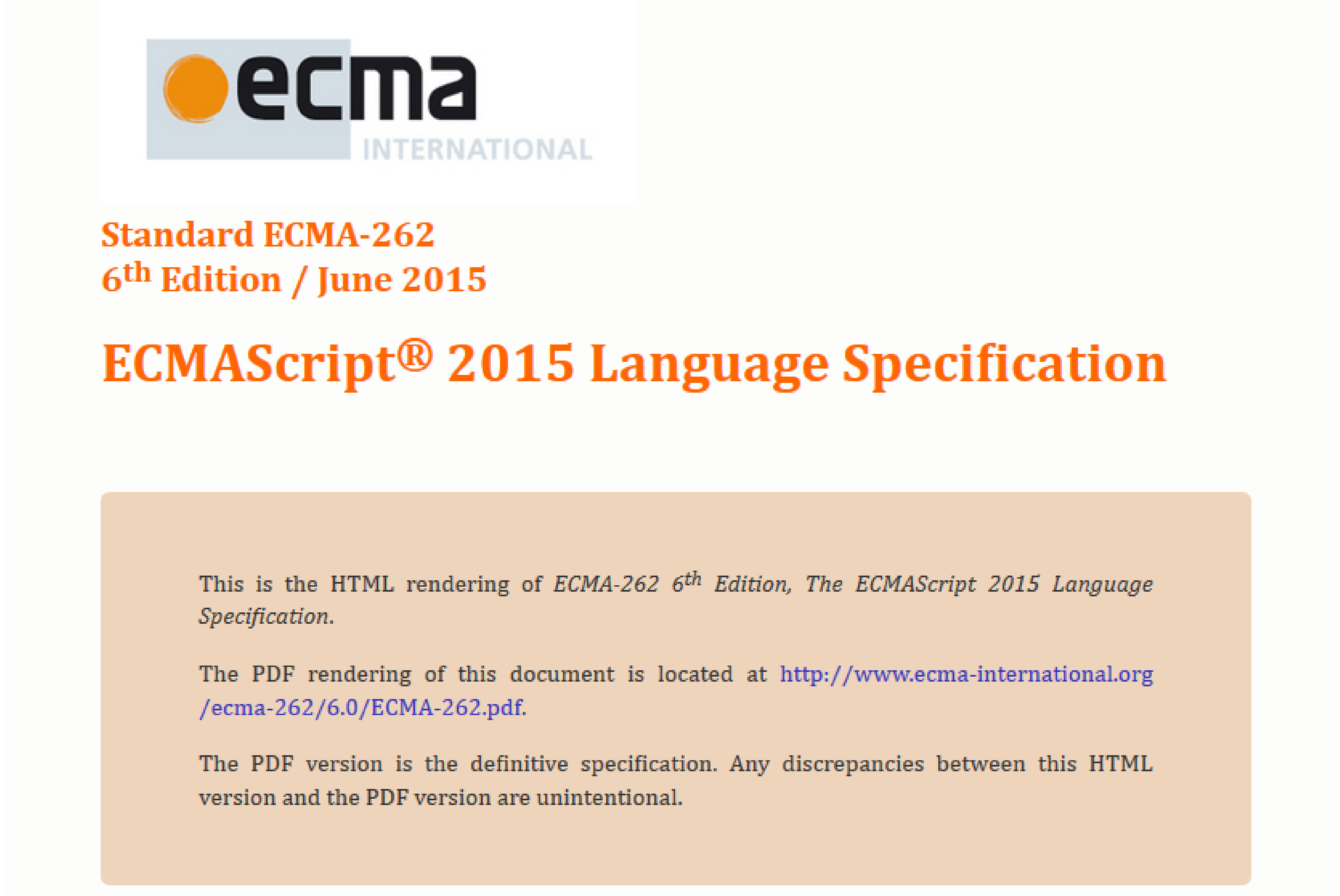
ECMA: Committee/organisation that publishes standards

ECMA-262: Language specification for scripting language

http://www.ecma-international.org/ecma-262/6.0/#sec-scope

ES6 (ECMAScript 6): 6th edition of specification for scripting languages that conform to ECMA-262 standards

In other words, we'll still be using **JavaScript**, but JS that conforms to ES6 standards



Standard ECMA-262
6th Edition / June 2015

ECMAScript® 2015 Language Specification

This is the HTML rendering of *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification*.

The PDF rendering of this document is located at http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

The PDF version is the definitive specification. Any discrepancies between this HTML version and the PDF version are unintentional.

http://www.ecma-international.org/ecma-262/6.0/#sec-scope

ECMAScript

You can find the full specification of ES6 here: http://www.ecma-international.org/ecma-262/6.0/#sec-scope

As well a quick reference of new features here: https://github.com/lukehoban/es6features/blob/master/README.md

And (as always) detailed explanations on: https://developer.mozilla.org/

We'll only be looking at some of the new ES6 features

ES6

Not all ES6 features are compatible with older browsers

Only way to know for sure that it will work on older browsers is to convert it to ES5

Process of converting code from one language to another = *transpiling*

One of the most popular ES6 -> ES5 transpilers = Babel (http://babeljs.io/)

ES6

Including Babel is very simple: just include the js-file and use type="text/babel" in your script tag when writing JS

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.26.0/babel.js"></script>
<script type="text/babel">
    // Everything you write here will be transpiled at runtime
</script>
```

Generally speaking, however, it is not a good idea to use Babel this way, since it makes your web application very slow. For production purposes, you should use a module bundler like Webpack (*https://webpack.js.org/*)

ES6

You can check browser support for new ES features here

https://kangax.github.io/compat-table/es6/

And here

https://caniuse.com/

ES6

However…

For this module, we will assume you’re using a modern browser that can run at least ES6

(If your browser can’t, you *really* should use a more up-to-date one)

ES6 – Declaring variables

Prior to ES6, we had limited options for declaring variables

ES6 contains two new ones: const and let

const value, once set, cannot be changed

```
const speedOfLight = 299792458;
// If we try and change this value,
// we'll get an error
```

ES6 – Declaring variables

It is good practice to declare variables you don't intend to change as constants

It is good practice to declare **all function expressions** as constants (which you will be required to do for this module)

```
const square = function(n){return n * n;}
```

ES6 – Declaring variables

Note that since arrays are essentially pointers, you can still add/change array items if the array is declared const

```
const arr = [];
arr.push('Cool');
arr[0] = 'Cool cool cool';
console.log(arr[0]);
// Output: Cool cool cool
```

ES6 – Declaring variables

JavaScript functions block variable scope, in other words, variables created inside a function are scoped to that function.

```
var currentScope = "global";

function doTheStuff(){
     var currentScope = "local";
     console.log(currentScope);
}

doTheStuff();
console.log(currentScope);
// Output: local
// Output: global
```

ES6 – Declaring variables

However, the same does not apply for other cases, such as if-statements.

```
var currentScope = "global";

if(currentScope){
     var currentScope = "local";
     console.log(currentScope);
}


console.log(currentScope);
// Output: local
// Output: local
```

ES6 – Declaring variables

The let keyword in ES6 allows us to scope a variable to any code block and thus protect the value of the global variable.

```
var currentScope = "global";

if(currentScope){
     let currentScope = "local";
     console.log(currentScope);
}



console.log(currentScope);
// Output: local
// Output: global
```

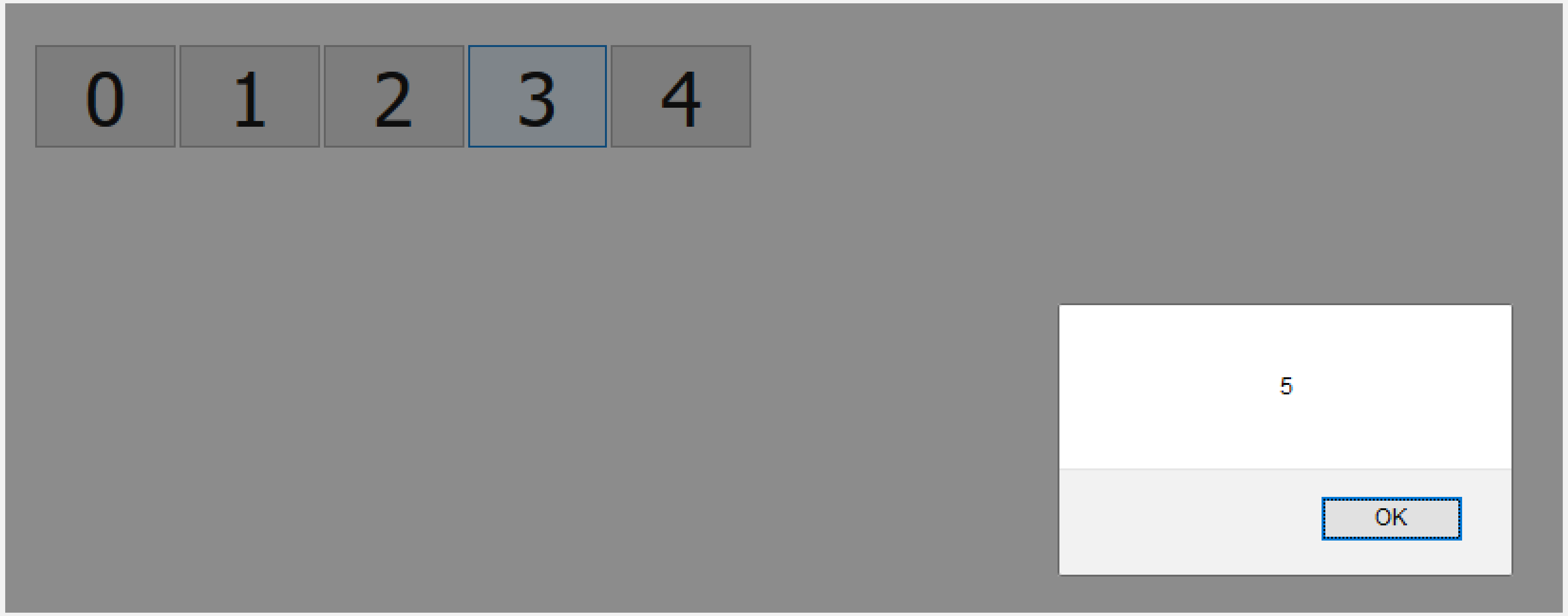
ES6 – Declaring variables

The same applies to for-loops. In the code below, the variable i is scoped globally

```
var container = document.getElementById("container");

for(var i = 0; i < 5; i++){
     var btn = document.createElement("button");
     btn.innerHTML = i;
     btn.onclick = function(){
          alert(i);
     }
     container.appendChild(btn);
}
```

ES6 – Declaring variables



5

When we click on any button, the value 5 is alerted, because once the loop has finished, the (global) variable i is set to 5

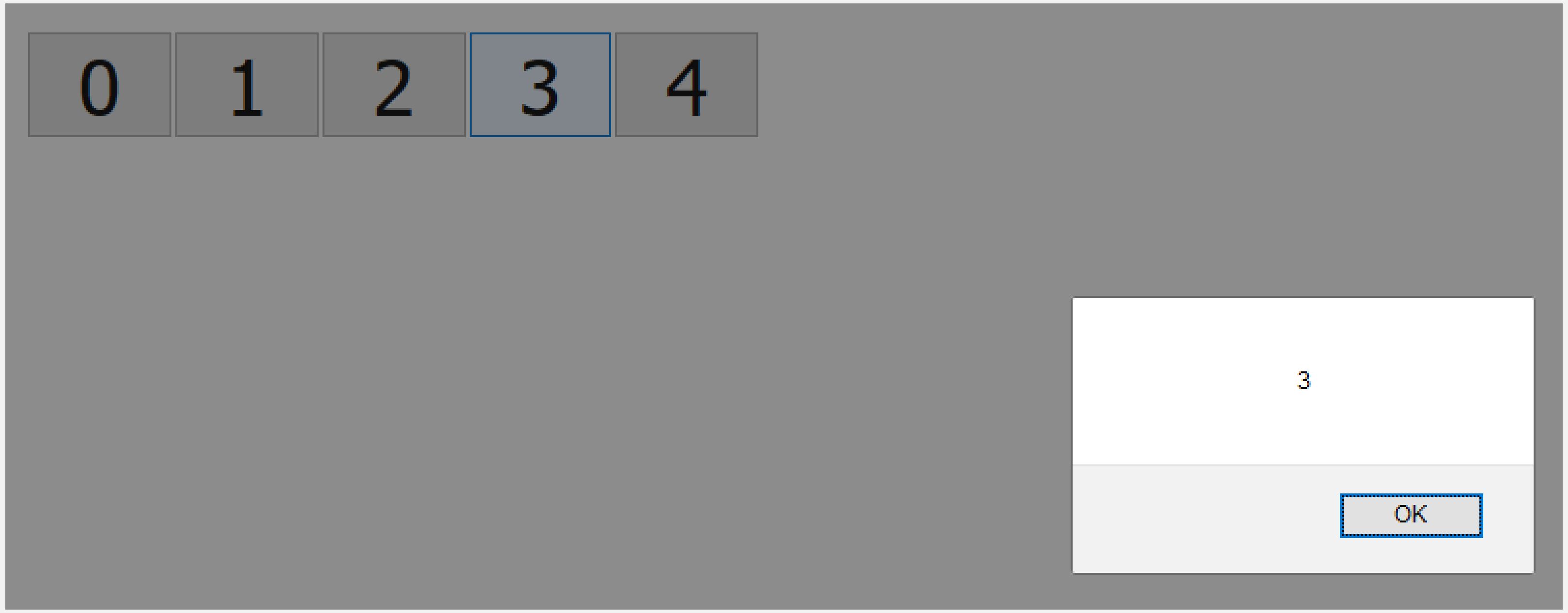
ES6 – Declaring variables

By using let instead of var, we scope the value of i to the loop iteration

```
var container = document.getElementById("container");

for(let i = 0; i < 5; i++){
      var btn = document.createElement("button");
      btn.innerHTML = i;
      btn.onclick = function(){
            alert(i);
      }
      container.appendChild(btn);
}
```

ES6 – Declaring variables



3

Now, when we click on any button, the corresponding value is alerted, since a scoped variable is used instead of the global one

ES6 – Template Strings

Creating a string with lots of variables in JS can get messy

```
let message = "Hello " + name + " " + surname + ".";
```

Template strings offer a way to do this in a clean, easily-readable way

```
let message = `Hello ${name} ${surname}.`;
```

NB: Note that tick marks (`) are used instead of single quotes (‘) or double quotes (“). (Tick marks are usually left of the “1” key)

ES6 – Template Strings

Template strings also honour whitespace, so creating multi-line strings is straightforward

```
let message = `

Hello ${name} ${surname},

We've updated our privacy policy

Regards`;
```

Note that you can also use single and double quotes inside a template string

ES6 – Template Strings

You can also use this to create multi-line HTML strings

```
document.body.innerHTML = `
<div>
     <h1>${pageTitle}</h1>
     <p>${pageContent}</p>
</div>
`;
```

ES6 – Default Parameters

ES6 also allows the use of default values for function parameters

```
function sayHello(name="Generic Person"){
     alert(`Hello there ${name}!`);
}

let myName = "Diffie";
sayHello();
// Output: Hello there Generic Person!
sayHello(myName);
// Output: Hello there Diffie!
```

ES6 – Arrow Functions

Arrow functions are another way to define anonymous functions in JavaScript

Using arrow functions is similar to using the **function** keyword, except in the way it handles the **this** keyword

Arrow functions don’t create their own **this**. They also can’t be used as constructors

ES6 – Arrow Functions

“Normal” way

```
function doThing(thing){
      let foo = "Hello World";
      thing(foo);
}

doThing(function(bar){
      alert(bar);
});

// Alerts "Hello world"
```

ES6 – Arrow Functions

Using arrow functions

```
function doThing(thing){
      let foo = "Hello World";
      thing(foo);
}

doThing((bar) => {
      alert(bar);
});

// Still alerts "Hello World"
```

ES6 – Arrow Functions

If there is only one function parameter, the bracket is optional

```
function doThing(thing){
      let foo = "Hello World";
      thing(foo);
}

doThing(bar => {
      alert(bar);
});

// Still alerts "Hello World"
```

ES6 – Arrow Functions

Arrow functions can contain *block bodies* and *expression bodies*

First, "normal" function

```
function addYass(name){
      return `Yass ${name}!`;
}
```

Block body: works same as regular function (have to return function output with return-keyword)

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

ES6 – Arrow Functions

Expression body: leave out curly brackets and function returns the value (without needing the return-keyword)

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

Using expression bodies, you can only have one line of code in your function. However, it can be useful for simple processing

```
let numbers = [1, 2, 3, 4];
let doubleNumbers = numbers.map(n => n * 2);
// doubleNumbers is now [2, 4, 6, 8]
```

ES6 – Arrow Functions

You can also call arrow functions from within arrow functions

```
const add = x => y => x + y;
```

This is functionally similar to

```
const add = function(x){
    return function(y){
        return x + y;
    };
};
// this has to be called like this: add(1)(2)
```

ES6 – Arrow Functions

this in a normal function

```
function Person(){
      this.age = 0;

      setInterval(function growUp(){
            console.log(this.age++);
            // Doesn't work, because "this" refers to
            // growUp's this, instead of Person's this
      }, 1000);
}
const p = new Person();
```

ES6 – Arrow Functions

this in an arrow function

```
function Person(){
      this.age = 0;

      setInterval(() => {
            console.log(this.age++);
            // Now it works, because arrow functions
            // don't create their own "this"
      }, 1000);
}
const p = new Person();
```

ES6 – Destructuring Assignment

Destructuring assignment is a way to unpack values from objects and arrays and assign them to variables

We'll look at destructuring arrays first. In the code below, we are assigning x, y, and z to the first three values of the array

```
const [x, y, z] = [10, 20, 30, 40, 50];


console.log(x);
// Output: 10
console.log(z);
// Output: 30
```

ES6 – Destructuring Assignment

You can also skip over array values

```
const [,,x,y] = [10, 20, 30, 40, 50];


console.log(x);
// Output: 30
// This is because we skipped over the first two array-
// values when assigning the value of x
console.log(y);
// Output: 40
// Since we skipped to the 2nd index when assigning x,
// y is now at index 3
```

ES6 – Destructuring Assignment

Next, let’s look at destructuring objects. In the code below we are grabbing the values from the person object and saving them as variables

```
const person = {
     name: "Sterling",
     surname: "Archer",
     codeName: "Duchess",
     age: 38
}

const {name, age} = person;
console.log(`${name} is ${age} years old`);
```

ES6 – Destructuring Assignment

In other words, instead of writing this…

```
const prop = object.prop;
```

…we can write this…

```
const {prop} = object;
```

…for any number of an object’s properties

ES6 – Destructuring Assignment

We can also use destructuring in function arguments

```
// (using our "person" object from the previous slide)

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

console.log(addYass(person));
// Output: Yass Sterling!
```

ES6 – Destructuring Assignment

Destructured variables can also have default values

```
const thirdValue = arr => {
     [,,x = -1] = arr;
     return x;
}
const numbers = [1, 2, 3];
console.log(thirdValue(numbers));
// Output: 3
numbers = [1, 2];
console.log(thirdValue(numbers));
// Output: -1
```

ES6 – Destructuring Assignment

Destructuring a value that does not exist gives us undefined

```
const [a] = [];

const [,b] = [1];

const {c} = {};

const {d} = {e: "foo"};

// a, b, c, and d are all undefined
```

ES6 – Object Literal Enhancement

You can think of object literal enhancement as the opposite of destructuring.

Instead of unpacking an object to save a variable, we are putting variables together and saving them as an object

```
const name = "Sterling";
const surname = "Archer";


const person = {name, surname};
// person is now an object that looks like this:
// {name: "Sterling", surname: "Archer"}
```

ES6 – Object Literal Enhancement

So basically, instead of this

```
const name = "Sterling";
const person = {name: name};
```

We can now write this

```
const name = "Sterling";
const person = {name};
```

Both of them give us this

```
{name: "Sterling"}
```

ES6 – Object Literal Enhancement

You can also include functions as part of your object

```
const name = "Slim";
const surname = "Shady";
const introduceYourself = function(){
      return `My name is ${this.name} ${this.surname}`;
}

const person = {name, surname, introduceYourself};
console.log(person.introduceYourself());
// Output: My name is Slim Shady
```

ES6 – Object Literal Enhancement

You also don’t have to include the **function** keyword when defining object methods

e.g. **Old Syntax**

```
var person = {
      name: name,
      surname: surname,
      introduceYourself: function(){
            return `My name is ${name} ${surname}`;
      }
}
```

ES6 – Object Literal Enhancement

You also don't have to include the **function** keyword when defining object methods

New syntax

```
const person = {
      name,
      surname,
      introduceYourself(){
            return `My name is ${name} ${surname}`;
      }
}
```

ES6 – Spread operator

The spread operator “expands” an array into elements (for arrays) or arguments (for functions)

We can use it to combine arrays

```
const list1 = ["Jeff", "Britta"];
const list2 = ["Abed", "Troy"];
const list3 = [...list1, ...list2];
// list3 is now [ "Jeff", "Britta", "Abed", "Troy"]
```

ES6 – Spread operator

We can also use it to expand an array into function parameters

```
function listNames(name1, name2, name3){
     return `${name1}, ${name2}, and ${name3}`;
}

const list1 = ["Jeff", "Britta", "Abed"];
console.log(listNames(...list1));
// Output: Jeff, Britta and Abed
```

ES6 – Spread operator

We can also use it while destructuring to only get some elements of an array, for example:

```
const list1 = ["Jeff", "Britta", "Abed", "Troy"];

const [firstName, ...rest] = list1;
// rest is now [ "Britta", "Abed", "Troy" ]

const [firstName, secondName, ...rest] = list1;
// rest is now [ "Abed", "Troy" ]
```

ES6 – Spread operator

It is also the recommended way to copy arrays

```
const list1 = ["Jeff", "Britta", "Abed", "Troy"];


const list2 = [...list1];
```

As opposed to looping through the values

ES6 – Spread operator

You can also use it to collect values from an object, similar to an array

```
const people = {
      name1: "Jeff",
      name2: "Britta",
      name3: "Abed",
      name4: "Troy"
};

const {name1, name2, ...otherNames} = people;
//otherNames is now { name3: "Abed", name4: "Troy" }
```

ES6 – Spread operator

Using the spread operator and object literal enhancement, we can easily combine objects to create new ones

```
const people = {
      name1: "Jeff",
      name2: "Britta",
      name3: "Abed"
};
const name4 = "Troy";
const combinedPeople = {...people, name4};
// combinedPeople is now { name1: "Jeff", name2:
// "Britta", name3: "Abed", name4: "Troy"}
```

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?
```

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://davidwalsh.name/spread-operator