Introduction to ECMAScript6

Objectives

- What is ECMAScript6?
- Transpiling with Babel
- Setting up Babel
- ESCMAScript6 features

What is ECMAScript6?

- ECMAScript6 is the latest JavaScript spec
- JavaScript was created in 1995 by Netscape
- Adopted by Microsoft as JScript
- Different browsers using different versions of the language resulted in the **need for a standard**

What is ECMAScript6?

Standards

- ECMA - Stands for European Computer Manufacturer Association
- ECMA - Provides ECMAScript specifications for JavaScript browsers
- ECMAScript - A **standard** for scripting languages

What is ECMAScript6?

Standards

The ECMA puts out specifications in an effort to put forth a "standard" for all browsers to follow. Browsers do not have to follow the standard and browsers rarely support all of the same features.

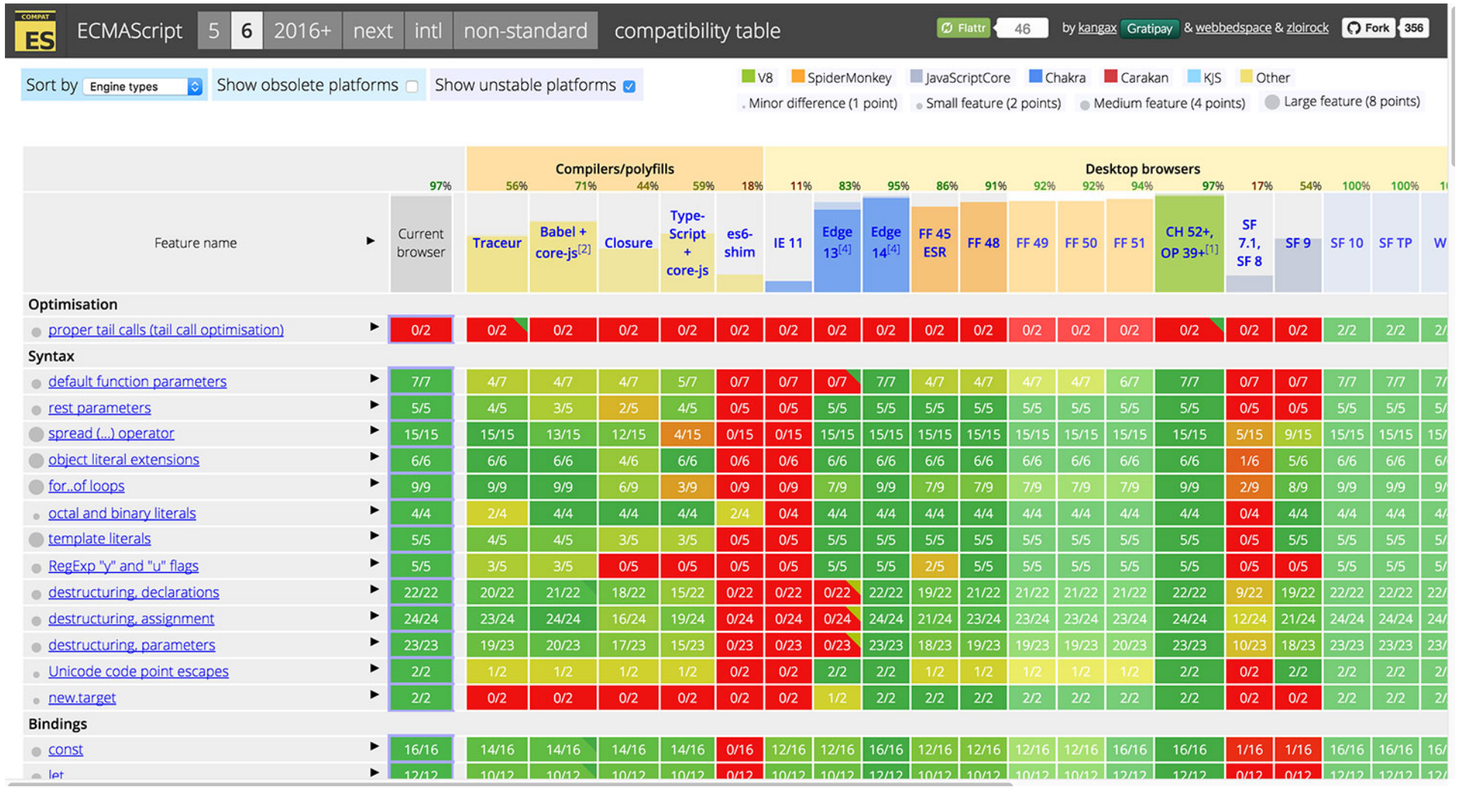
What is ECMAScript6?

Standards

- 1995 - JavaScript is born
- 1997 - ECMAScript 1
- 2009 - ECMAScript 5
- 2015 - ECMAScript 6

What is ECMAScript6?

ES6 Compatibility Table



Transpilers

Transpiling - Taking source code written in one language and transforming into another language or a lower version of the same language

The lack of complete ES6 support in browsers results in the need for a **transpiler**.

A transpiler will allow you to use most of the ECMAScript6 features right now!

Transpilers

Transpilers are used for various different languages

- CoffeeScript => JavaScript
- SASS => CSS
- ES6 => ES5

Transpilers

Babel is a very popular transpiler for ES6.

Babel converts ES6 code into ES5 code.

```
[1,2,3].map(n => n + 1);
```

```
[1,2,3].map(function(n) {
  return n + 1;
});
```

Transpilers

If you take a look at the compatibility table we saw before, you will notice that Chrome supports most of ES6.

You still need to use Babel because you never know which browser your visitors will be using!

Babel

There are two ways to transpile ES6 with Babel:

- In browser transpiler: Transforms ES6 to ES5 at **runtime**
- Babel CLI: Transforms ES6 to ES5 over the CLI. The JS served to the browser is already transpiled.

The in browser transpiler is not good for production sites. It is much slower than pre-transpiling with the Babel CLI.

Babel

The in browser transformer is useful for tinkering with ES6 and trying out the new features. It saves you from having to transpile every time you make a code change. Let's try out the in browser transformer.

Babel in Browser Transformer

```
<!DOCTYPE html>
<html>
<head>
    <title>Babel Demo</title>
</head>
<body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.38/browser.js"></script>
    <script type="text/babel">
        // ES5 does not support default arguments for functions
        var sayHello = function(name="World") {
            return "Hello " + name;
        };

        console.log(sayHello());        // Hello World
        console.log(sayHello('Greg')); // Hello Greg
    </script>
</body>
</html>
```

Babel CLI

The in browser transformer is great for tinkering but we need to be able to transpile our code before runtime for our production applications.

Let's install Babel CLI

Babel CLI

Babel recommends you do not install it globally for two reasons:

1. Different projects on the same machine can depend on different versions of Babel allowing you to update one at a time.
2. It means you do not have an implicit dependency on the environment you are working in. Making your project far more portable and easier to setup.

Let's set up Babel in an example project!

Babel Setup

```
cd ~/Desktop
mkdir babel-demo
cd babel-demo
mkdir src lib
npm init -f
npm install --save-dev babel-cli
```

Babel Setup

Instead of running Babel directly from the command line we're going to put our commands in npm scripts which will use our local version of babel

Add a "scripts" field to your package.json and put the Babel command inside there as "build"

Babel Setup

```
{
  "name": "babel-demo",
  "version": "1.0.0",
  "scripts": {
    "build": "babel src/js -d lib/js"
  },
  "devDependencies": {
    "babel-cli": "^6.14.0"
  }
}
```

Babel Setup

Before we can run our scripts we need to tell babel what ES6 features should be converted to ES5. This is new in Babel 6.

Babel allows you to specify which transformations should be run.

In other words you can have it convert some ES6 features to ES5 and leave others as straight ES6.

Babel Setup

Pre-6.x, Babel enabled certain transformations by default. However, Babel 6.x does not ship with any transformations enabled. You need to explicitly tell it what transformations to run.

— *babel docs*

Babel Setup

To avoid manually specifying a long list of transformations, babel provides presets.

https://babeljs.io/docs/plugins/

Babel Setup

Each yearly preset only compiles what was ratified in that year

- latest
- es2017
- es2016
- es2015
- react

Babel Setup

The preset "latest" will transform all yearly presets so let's use that one.

Install the preset:

```
npm install babel-preset-latest --save-dev
```

Babel Setup

Now that we have the preset installed we have to actually tell Babel to use it.

Babel reads configuration from a .babelrc file so let's make one in the root of your project.

```
touch .babelrc
```

Babel Setup

/path/to/babel-demo/.babelrc

```
{
  "presets": ["latest"]
}
```

Babel Setup

So far:

- Installed babel in our local project
- Added a script to packages.json to run babel "npm run build"
- Installed a babel preset "latest"
- Told babel to use the "latest" preset with the .babelrc file

Babel Setup

We are almost ready to actually use babel!

First we need to create a js file with some ES6 code in it.

Create a file named example.js in src/js

```
touch /path/to/babel-demo/src/js/example.js
```

Babel Setup

```
// src/js/example.js
function sayHello(name="World") {
    return "Hello " + name;
}
```

Babel Setup

Once you have created the file and added the code to it run your build script.

```
npm run build
```

Babel Setup

Take a look at "lib/js/example.js". You can see that the code required to allow for a default argument in ES5 is much more verbose than ES6.

```
function sayHello() {
    var name = arguments.length <= 0 || arguments[0] === undefined ? "World" : arguments[0];

    return "Hello " + name;
}
```

Babel Setup

You could also directly access babel from the node_modules directory.

Babel pushes the output to **stdout** (standard output) by default. Meaning it doesn't write anything, it just shows the result in your terminal.

```
./node_modules/.bin/babel src
```

```
// Output
function sayHello() {
    var name = arguments.length <= 0 || arguments[0] === undefined ? "World" : arguments[0];

    return "Hello " + name;
}
```

Babel Setup

Anytime you install a new CLI tool, it's a good idea to run --help to see the usage information.

```
./node_modules/.bin/babel --help
```

Babel Setup

One of the most useful babel features is the "watch" command.

You can have it watch a file or directory for file changes and automatically write the transformed file on save.

```
./node_modules/.bin/babel src -d lib --watch
```

Babel

Most applications will need to be cross browser. Your employer won't be happy when their application doesn't work because the code you wrote doesn't work in IE. A transpiler is pretty much mandatory if you want cross browser support and Babel is the most popular transpiler for ES6.

Babel

Now that you know how to set up Babel and have your ES6 code work cross browser, we can start digging into the actual features of ES6.

ECMAScript6 Features

Overview

- Default function parameters
- let and const
- Template strings
- Spread operator
- Arrow functions
- Enhanced Object Literals
- Classes
- Modules

Default Params

You've already seen an example of default params.

You simply provide a default value for a parameter in the function definition.

```
function sayHello(name="World") {
  return "Hello " + name;
}

sayHello(); // Hello World
sayHello('Jane'); // Hello Jane
```

let and const

ECMAScript6 introduces two new identifiers to be used as an alternative to var, **let** and **const**

- const: Indicates that the variable should not be reassigned
- let: Indicates that the variable may be reassigned

let and const

Good code does a great job of **describing its intent**.

let and **const** are all about describing how a program should work and enforcing rules. These new indentifiers give programmers the ability to be more descriptive when writing code and their code is more readable and less bug prone as a result.

let and const

Again, const should be used when you don't want the value to ever change.

If someone tries to reassign a variable you created using `const`, an error will be thrown.

```
const animal = "dog";
animal = "cat";
// TypeError: Assignment to constant variable.
```

let and const

It is important to note that while primitive types such as strings, integers, and booleans cannot be reassigned on a variable declared with const, object properties can be.

```
const animals = {cow: 'moo', cat: 'meow'};
animals.cow = 'quack';
console.log(animals.cow); // quack
```

This is something you should avoid doing, just be aware that it is possible.

let and const

The let keyword gives us **block level scope** in JavaScript

Block level scope refers to any code surrounded in curly braces {}

In programming, it is common for a variable to only be relevant within a certain block of code

A great example of this is within a for loop

let and const

```
// The numbers 1 - 5 will be logged to the console
for (var i = 1; i < 6; i++) {
  console.log(i);
}

// 6 will be logged to the console
// in a block scoped language, i would be undefined
console.log(i);
```

let and const

Swap var with let to enfore block scope

```
// The numbers 1 - 5 will be logged to the console
for (let i = 1; i < 6; i++) {
  console.log(i);
}

console.log(i); // ReferenceError: i is not defined
```

let and const

```
if (true) {
  var a = 1; // we are inside of a block
}
console.log(a); // 1

if (true) {
  let b = 2; // we are inside of a block
}
console.log(b); // ReferenceError: b is not defined
```

let and const

If a variable is relevant only within a certain scope, it is best practice not to let it leak outside of that scope.

Some other code might use the same variable name and the author might not realize that variable was already defined resulting in unexpected output or behavior.

let and const

Variables leaking outside of their relevant scope can lead to bugs that are hard to find.

You should now **use let instead of var** and use const when the value should not be reassigned.

Template Strings

Template strings allow you to write multi-line strings and use string interpolation.

String interpolation allows you to place variables within a string using a special syntax.

Let's look at an example.

Template Strings

```
function createEmailContent(to, from, message) {
  // template strings open and close with backticks
  return `Hello ${to},

${message},

Regards,
${from}`
}

console.log(createEmailContent('Jane', 'Acme Inc', 'Thank you for contacting us!'));
```

Template Strings

Note that template strings are white space sensitive. In the function body, we did not indent anything after the return. If we did, then the lines following the return would be indented in the output.

```
Hello Jane,

Thank you for contacting us!

Regards,
Acme Inc
```

Exercise

Create a function that takes 2 or more parameters and output them inside of a template string.

Spread Operator

The spread operator has two great features:

1. Turn elements of an array into arguments of a function call

2. Place elements of an array into another array

Spread Operator

Turn elements of an array into arguments of a function call

```
let numbers = [5, 5];
function add(a, b) {
  return a + b;
}

foo(...numbers); // 10
```

Spread Operator

Place elements of an array into another array

```
let newMembers = ['Jane', 'Bob'];
let members = ['Tom', 'Ashley', ...newMembers];
console.log(members); // ['Tom', 'Ashley', 'Jane', 'Bob'];
```

Exercise

1. Define a subtract function that has two parameters. Pass two numbers to the function using the spread operator.

2. Create an array of animals and add more animals to it using the spread operator.

Arrow Functions

ES6 provides a shorthand syntax for defining an anonymous function. It is similar to CoffeeScript.

```
// ES5
var squares = [2, 4, 6].map(function(x) {
  return x * x;
}); // 4, 16, 36

// ES6
let squares = [2, 4, 6].map(x => x * x); // [4, 16, 36]
```

Arrow Functions

Arrow functions are useful anywhere you might need an anonymous function as an argument.

You will use this a lot when writing jquery.

```
$(document).keypress(event => {
  if (event.which === 13) {
    console.log('You pressed enter');
  }
});
```

Arrow Functions

Arrow function need curly braces when they have a statement body

```
// expression bodies don't need curly braces
var squares = [2,3,4].map(x => x * x);

// statement bodies need curly braces
[1,2,3,4].forEach(num => {
  if (num % 2 === 0) {
    console.log(num + ' is an even number');
  }
});
```

Arrow Functions

Wrap arguments in parenthesis when there are multiple arguments.

```
// assume we loaded the jQuery library

// no args, just use empty parenthesis
$(document).keypress(() => {
  console.log("You pressed a key. I don't care which one!");
});

// one argument, no parenthesis
$(document).keypress(e => {
  if (e.which === 13) {
    console.log('You pressed enter!');
  }
});

// multiple arguments, use parenthesis
var add = (a, b) => a + b;
```

Arrow Functions

In typical anonymous functions, "this" refers to the global object. Let's look at an example.

Arrow Functions

```
var person = {
  name: 'Jane',
  attributes: ["5'5", 'female'],

  printDescription() {
    this.attributes.forEach(function(attribute) {
      // Logs the global object if not in strict mode (Window in a browser)
      // Logs 'undefined' if in strict mode
      // Either way, 'this' is not person
      console.log(this);

      // this.name will be undefined
      console.log(this.name + ' is ' + attribute);
    });
  }
}
```

Arrow Functions

Making the previous code work properly in ES5 would look like the following:

```
var person = {
  name: 'Jane',
  attributes: ["5'5", 'female'],

  printDescription() {
    var self = this; // declare self to current value of this
    this.attributes.forEach(function(attribute) {
      // use self to access property "name"
      console.log(self.name + ' is ' + attribute);
    });
  }
}
```

Arrow Functions

Arrows share the same lexical "this" as their surrounding code. We don't need to declare a variable just to access "this" on person.

```
var person = {
  name: 'Jane',
  attributes: ["5'5", 'female'],

  printDescription() {
    this.attributes.forEach(attribute => {
      console.log(this.name + ' is ' + attribute);
    });
  }
}

person.printDescription();
// Jane is 5'5
// Jane is female
```

Enhanced Object Literals

ES6 makes some enhancements to object literals.

Some enhancements include:

- shorthand property assignments
- shorthand for defining methods

Enhanced Object Literals

```
// ES5
var person = {
  _name: '',
  setName: function(name) {
    this._name = name;
  },
  getName: function() {
    return this._name;
  }
}

// ES6
var person = {
  _name,
  setName(name) {
    this._name = name;
  },
  getName() {
    return this._name;
  }
}
```

Classes

Classes are a very popular concept in object oriented programming languages. They allow for encapsulation and reusability.

ES6 finally brings classes to JavaScript!

Classes

```
class Prescription {
  constructor(drug, quantity) {
    this._drug = drug;
    this._quantity = quantity;
  }

  getDrug() {
    return this._drug;
  }

  getQuantity() {
    return this._quantity;
  }
}

let rx = new Prescription('amoxicillin', 30);
rx.getDrug(); // amoxicillin
```

Classes

Classes can extend other classes, inheriting their properties and methods.

Classes

```
class Controller {
  constructor(urlPrefix = '') {
    this._urlPrefix = urlPrefix;
  }

  getUrlPrefix() {
    return this._urlPrefix;
  }
}

class UsersController extends Controller {
  constructor(urlPrefix = 'users') {
    super(urlPrefix); // call parent constructor or urlPrefix won't be set!
  }
}

let usersController = new UsersController;
usersController.getUrlPrefix(); // users
```

Modules

You've already seen modules they just weren't native to JavaScript!
You've probably seen them in the CommonJS format within Node.js

```
modules.exports = function() {
  // return some value
}
```

ES6 makes modules native to JavaScript

Modules

Why do we use modules?

Before modules, we basically stuffed everything in global scope in JavaScript. Globals are generally bad. The more globals you have the greater the risk of bugs.

Let's look an HTML file.

```
<!DOCTYPE html>
<html>
<head>
    <title>Modules</title>
</head>
<body>

  <script src="/src/js/main.js"></script>
  <script src="/src/js/NavBar.js"></script>
  <script src="/src/js/ListWriter.js"></script>
  <script src="/src/js/TodoList.js"></script>
  <!-- and so on... -->
</body>
</html>
```

As your project grows, you will add more and more scripts. Some scripts will depend on other scripts and you have to keep track of which order they come in. It might seem trivial at a glance but it actually gets complicated and causes a lot of headaches. This is the problem that modules solve.

```
<script src="/src/js/main.js"></script>
<script src="/src/js/NavBar.js"></script>
<script src="/src/js/TodoList.js"></script>
<script src="/src/js/ListWriter.js"></script>
```

Modules

Modules allow us to keep code related to one thing in it's own file and then require that file in another file if we need its functionality.

Modules

Example:

ListWriter.js relies on TodoList.js. This is because TodoList is a model object. TodoList holds data and ListWriter manipulates data on the TodoList based on buttons being clicked on the screen.

```
<script src="/src/js/main.js"></script>
<script src="/src/js/NavBar.js"></script>
<script src="/src/js/TodoList.js"></script>
<script src="/src/js/ListWriter.js"></script>
```

Modules

Modules allow you to write code in any scope you want inside of a file and then only export one value, ensuring you do not expose anything else to the script that imports it.

```
// Person.js
let foo = 'bar';
export default class Person {
  // class code here...
}

// main.js
import Person from './Person';
console.log(foo); // undefined
console.log(Person) // class Person {}
```

Modules

Before imports and exports, we could accomplish including a file into global scope without exposing anything but one object by way of the module pattern.

```
var SomeModule = (function () {
  // all vars and functions are in this scope only
  // still maintains access to all globals
}());
```

We would then include this script in an HTML file and only "SomeModule" would be in global scope.

```
<script src="SomeModule.js"></script>
```

With imports and exports you don't have to worry about wrapping your code in anonymous function.

Modules

Before ES6 modules came into play, the JS community came up with their own solutions for modules.

- CommonJS Modules:
  - Used in Node.js
  - Primarily used for the server
- Asynchronous Module Definition (AMD):
  - Used in Require.js
  - Mainly used in browsers

You won't have to worry about these unless a library or framework you are using includes them.

Modules

ES6 modules have **named exports** and **default exports**

When you export something, you expose it to the script that imports it. You've probably seen a script imported via **require** in Node.js

ES6 uses the keyword **import** instead of **require**

Modules

Named exports are used when you want to expose several values or objects from a script

Default exports are used when you want to export only one object or value from a script

It is considered best practice to export only one value so you will primarily be using default exports

Modules

Named Exports

To export more than one value, simply add the export keyword in front of the declaration.

```
// FILE: player.js
export const maxPlays = 5;
export function run() {
  // make player run
}
export function jump() {
  // player jump
}

// FILE: main.js
import { maxPlays, jump } from './player';
// import only maxPlays and jump

import * from './player'
// import everything that player exposes (maxPlays, run, jump)
```

Modules

You can also import a module as an object and access its exports as properties.

```
// FILE: player.js
export default class Player {
  // code
}

// FILE: main.js
import * as player from './player';
player.jump();
player.maxPlays;
```

Modules

Default Export

Concerning the default export, there is only a single default export per module. A default export can be a function, a class, an object or anything else. This value is to be considered as the "main" exported value since it will be the simplest to import.

— Mozilla Docs

Modules

Use default export when you want to export a single value from a module.

```
// FILE player.js
export default class Player {
  // code
}
```

Modules

Specifying a default export makes it easier to import the module.

```
// no default export defined in player
import { player } from 'player';

// with default export defined in player
import player from 'player';
```

Wrapping Up

We learned how to set up babel so we can start using ES6 features cross browser and we covered a lot of ES6 features.

There are even more ES6 features than we covered and features continue to be added.

Two great resources for diving further into ES6:

- https://github.com/lukehoban/es6features
- http://www.benmvp.com/learning-es6-series

Exercise

Complete the following using the babel demo project you created:

- Create two modules (one file per module, both files export a single class)
- Kudos if one of your classes takes the other class as a dependency (constructor param)
- Import the modules into a main.js
- Create some instances of your object(s) and log them to the console
- Use babel --watch to compile files as you write
- Add some ES6 features you learned to your classes if you have time

Resources

- babeljs.io
- CDNJS
- Compatibility Table
- Template Literals
- Spread Operator
- Learning ES6
- Learn ES2015
- ES6 Modules