ES2015 JavaScript

ES2015 or ES6?

What should we call it?

- You will see both ES2015 and ES6 used to describe the current widely-released version of JavaScript
- The official specification is called ECMAScript 2015 Language, but that is for the language specification and NOT the implemented coding language
- There is not an officially preferred name; you may use either ES2015 or ES6 to refer to the current JS release

ES2015 JavaScript

Template Literals

Life Before Template Literals

```
function sayHello(name) {
    console.log('Hello ' + name + '!');
}

sayHello('John');
```

- Traditionally, when you want to use a variable's value as part of a string, you have to use string concatenation to combine the pieces of the string together
- You have to be careful with spacing and punctuation, otherwise you might get something like: HelloJohn!

Template Literals to the Rescue

```
function sayHello(name) {
    console.log(`Hello ${name}!`);
}

sayHello('John');
```

- A **template literal** is a special type of string that is denoted by backticks (`) instead of single or double quotes
- Inside a template literal, everything is assumed to be a character unless you interpolate it
- **Interpolation** is the process of evaluating a JavaScript expression or variable and outputting its value
- To interpolate, surround the variable or expression with the interpolation syntax: `${someVar}`

- When using template literals, our statements flow more naturally and we aren't prone to spacing or punctuation errors

ES2015 JavaScript

Default Parameter Values

Life Before Default Parameter Values

```
function greet(name) {
    alert(`Hello ${name}!`);
}

greet('James'); // Hello James!
greet(); // Hello undefined!
```

- If an argument is not passed into a function when it is called, the parameter will be undefined

Life Before Default Parameter Values

```
function greet(name) {
    if (name === undefined) {
        name = 'World';
    }
    alert(`Hello ${name}!`);
}

greet('James'); // Hello James!
greet(); // Hello World!
```

- To prevent this, you could check and see whether the parameter is undefined, and if so, assign a default value to it

Using Default Parameter Values

```
function greet(name = 'World') {
    alert(`Hello ${name}!`);
}

greet('James'); // Hello James!
greet(); // Hello World!
```

- A default parameter value looks like an assignment statement in the parameter list
- The assignment only occurs if an argument is not passed into the function to fill that parameter

Mixing Defaults and Non-Defaults

```
function product(a = 1, b) {
    console.log(a * b);
}

product(2, 3); // 6
product(2);    // NaN
```

- You can have some parameters with default values and some without
- Although having your default parameters first is not technically an error, it is a poor practice
- In this example, the function works as expected when sent two arguments
- When a single argument is sent, that value is set into the `a` parameter, and `b` is `undefined`
  - `2 * undefined = NaN` (Not a Number)

Mixing Defaults and Non-Defaults

```
function product(a, b = 1) {
    console.log(a * b);
}

product(2, 3); // 6
product(2);    // 2
```

- By making sure your default parameter values are at the end of the parameter list, you will have the option of leaving arguments out when you call the function
- In this example, this function behaves as an identity function if you pass just a single value in

ES2015 JavaScript

Arrow Function Expressions

Arrow Function Expression

```
let btn = document.getElementById('my-btn');

// Using standard function expression
btn.addEventListener('click', function() {
    alert('Hello!');
});

// Using arrow function
btn.addEventListener('click', () => {
    alert('Hello!');
});
```

- An **arrow function**, or arrow function expression, is a more concise representation of a function expression
- This example compares using a standard function expression in a click listener to using an arrow function
- The primary difference in syntax is that you DO NOT use the `function` keyword when writing an arrow function
- You also need to add an arrow (=>) between the parameter parenthesis and the function body curly braces

Another Example

```
let add = (a, b) => {
    alert(a + b);
};

add(5,4);
```

- Here is another example of an arrow function
- Once you get used to the missing function keyword and the extra arrow, you can see they are very similar syntactically
- However, there are many popular shorthands that you will see on the web
    - Many of these make it hard to read and understand the arrow expression

Shorthand: Single Parameter

```
// shorthand
let greet = name => {
    alert(name);
};

// equivalent to:
let greet = (name) => {
    alert(name);
};

greet('Jane');
```

- When an arrow function has a single parameter, you may discard the parenthesis around the parameter list
- Note that when you have no parameters, or more than one parameter, you must have parenthesis

Shorthand: Concise Body

```
// shorthand
let add = (a, b) => a + b;

// equivalent to:
let add = (a, b) => {
    return a + b;
};

let sum = add(5,4);
console.log(sum);
```

- If an arrow function simply needs to return a value that can be easily accessed or computed, you can use the concise body shorthand
- With a concise body, you DO NOT use curly braces to specify the body of the arrow function
- You simply place a variable or expression right after the arrow, and that value will be *returned* by the arrow function

Shorthand: Concise Body

```
let getStats = (a, b) => ({
    sum: a + b,
    difference: a - b,
    product: a * b,
    quotient: a / b
});

let result = getStats(4, 2);
console.log(result);
```

- When using the concise body shorthand, you must surround a returned object literal with parenthesis

How obscure can you go?

```
let p1 = {
    firstName: 'Jane',
    lastName: 'Doe',
    age: 29
};

// Most verbose
let getFullName = (person) => {
    return `${person.firstName} ${person.lastName}`;
};

let result = getFullName(p1);
console.log(result);
```

How obscure can you go?

```
let p1 = {
    firstName: 'Jane',
    lastName: 'Doe',
    age: 29
};

// single parameter shorthand
let getFullName = person => {
    return `${person.firstName} ${person.lastName}`;
}

let result = getFullName(p1);
console.log(result);
```

How obscure can you go?

```
let p1 = {
    firstName: 'Jane',
    lastName: 'Doe',
    age: 29
};

// single param shorthand, concise body
let getFullName = person => `${person.firstName} ${person.lastName}`;

let result = getFullName(p1);
console.log(result);
```

ES2015 JavaScript

Spread Syntax

Spreading Values in a Function Call

```
function add(a, b, c) {
    console.log(a + b + c);
}

let enteredValues = [5, 2, 7];

add(...enteredValues);

// same as:
add.apply(null, enteredValues);

// same as:
add(5, 2, 7);
```

- If you have an array of values, you can use the spread operator (`...`) to send the values into a function as arguments
- Basically, the **spread operator** is taking a list of values and breaking them out into a comma separated list

Spreading Values in an Array

```
let low = [0, 1, 2];
let high = [7, 8, 9];

let n = [...low, 3, 4, 5, 6, ...high];

console.log(n);
// [0,1,2,3,4,5,6,7,8,9]
```

- You can use the spread operator to unfurl the values of an array into another array

Spreading a String

```
let str = 'Code';

let chars = [...str];

console.log(chars);
// [ 'C', 'o', 'd', 'e' ]
```

- You can use the spread operator to spread the characters of a string into an array

ES2015 JavaScript

Object Literal Property Shorthand

Property Names in Object Literals

```
function getStats(a, b) {
    let resultAdd = a + b;
    let resultSub = a - b;

    return {
        sum: resultAdd,
        difference: resultSub
    };
}

let stats = getStats(4,2);
console.log(stats);
```

- This function has local variables `resultAdd` and `resultSub`, and it packages those values up in an object literal before returning it
- We picked the names of the local variables, as well as the names of the properties in the object (`sum`, `difference`)
- When we log `stats` to the console, we will see an object with properties `sum` and `difference`

Property Names in Object Literals

```
function getStats(a, b) {
    let sum = a + b;
    let difference = a - b;

    return {
        sum: sum,
        difference: difference
    };
}

let stats = getStats(4,2);
console.log(stats);
```

- There is nothing that says our property names cannot be the same as the variable names we are accessing
- This example functions the exact same as before
- When you are defining an object literal, the word to the left of the : is the name of the property you are choosing to put on the object
- The value after the : is going to be the value for that property

- So in this example, starting on the right, we are accessing the sum variable, getting its value, and setting that as the value of the sum property for the object literal we are creating

Property Name Shorthand

```
function getStats(a, b) {
    let sum = a + b;
    let difference = a - b;

    return {
        sum,
        difference
    };

    // or:
    return { sum, difference };
}

let stats = getStats(4,2);
console.log(stats);
```

- When a property name is the same as the variable that will fill its value, you can simply use the property/variable name once and omit the :
- This shorthand is often written on one line

ES2015 JavaScript

Destructuring Assignment

Destructuring Assignment

- **Destructuring assignment** syntax allows us to unpack values from an array or object and assign them into their own, separate variables
- In basic terms, destructuring is an opportunity to take a larger piece of structured data, and pick out just the pieces you need into conveniently named variables
- Destructuring can be performed on arrays and objects

Destructuring Assignment with Arrays

```
let [a, b, c] = ['Apple', 'Banana', 'Pear'];

console.log(a); // 'Apple'
console.log(b); // 'Banana'
console.log(c); // 'Pear'
```

- This is not a technique you will frequently use, but you may stumble across it online so it is important to see how it works
- The values are unpacked from the array and placed into variables in the corresponding order: `'Apple'` gets placed into created variable `a`, `'Banana'` gets placed into created variable `b`, and `'Pear'` gets placed into created variable `c`

Destructuring Assignment with Objects

```
let p1 = {
    firstName: 'Jane',
    lastName: 'Doe',
    age: 29
};

let { firstName } = p1;

// can do multiple
let { lastName, age } = p1;

console.log(firstName); // 'Jane'
console.log(lastName); // 'Doe'
console.log(age); // 29
```

- This technique is used often
- The right-hand-side of the equal sign is an object (object literal, variable containing an object, function that returns an object, etc)
- The left-hand-side is where you specify the property names you want to pull out as separate variables
- You don't have to grab all the properties

Object Destructuring: Default Values

```
let p1 = {
    firstName: 'Jane',
    lastName: 'Doe',
    age: 29
};

let { firstName, gender = 'F' } = p1;

console.log(firstName);  // 'Jane'
console.log(gender); // 'F'
```

- You can specify a default value for a destructured property
- The default value will be used if the object does not contain the property you are trying to unpack

Object Destructuring: Renaming Properties

```
let p1 = {
    firstName: 'Jane',
    lastName: 'Doe',
    age: 29
};

let { firstName: fName } = p1;

console.log(fName); // 'Jane'
```

- This is a non-obvious syntax
- First you specify the property you want to unpack from the object
- You then specify a new name for it after a `:`
- The variable that is created with the unpacked value will use the new name

Array Destructuring in Function Parameters

```
function myFunc([op1, op2]) {
    console.log(op1);
    console.log(op2);
}

let myArray = [12, 7];

myFunc(myArray);
```

Object Destructuring in Function Parameters

```
function myOtherFunc({ lat, lng }) {
    console.log(lat);
    console.log(lng);
}

let address = {
    street: '123 Main St',
    city: 'Birmingham',
    state: 'AL',
    lat: '33.514961',
    lng: '86.807853'
};

myOtherFunc(address);
```

ES2015 JavaScript

JavaScript Modules

JavaScript Modules

- JavaScript module syntax allows code to be exported from and imported into other files
- Relies on the `import`, `export`, and `default` keywords

Exporting at Declaration

```
// myStuff.js

export function sayHello(name) {
    alert(`Hello ${name}!`);
}

export let cityCode = 'BHM';

export const MAX_SEATS = 20;
```

- Exporting is a passive act; you are simply marking something as being available for import elsewhere
- To make a function or variable available for import in another module (file), simply place the **export** keyword in front of the declaration

Exporting Later

```
// myStuff.js

function sayHello(name) {
    alert(`Hello ${name}!`);
}

let cityCode = 'BHM';

const MAX_SEATS = 20;

export { sayHello, cityCode };
```

- This syntax allows you to later specify which functions/variables will be available outside this module
- In this example, the `MAX_SEATS` constant was not included in the export, so it will NOT be available for import elsewhere

Default Export - Function

```
// myStuff.js

export default function sayHello(name) {
    alert(`Hello ${name}!`);
}

export let cityCode = 'BHM';

export const MAX_SEATS = 20;
```

- So far, we've seen examples of **named exports**
- If you have one, main piece of code you want to be exported from this module, you can use a **default export**
- To do that, use the `default` keyword
- This example contains a default export of function `sayHello` and named exports of `cityCode` and `MAX_SEATS`

Default Export - Variable/Constant

```
// myStuff.js

export function sayHello(name) {
    alert(`Hello ${name}!`);
}

// Bad
export default let cityCode = 'BHM';

// Good
let cityCode = 'BHM';
export default cityCode;

export const MAX_SEATS = 20;
```

- While the `default` keyword can be used in a function declaration, it cannot be used inline with a variable/constant declaration
- Instead, you must first declare the variable/constant, and then specify it as the default export

import

- To import resources that have been exported from other modules, we use the `import` keyword
- The import syntax varies depending on what type of export you are importing

Importing Named Exports

```
// myStuff.js
export function sayHello(name) {
    alert(`Hello ${name}!`);
}

export let cityCode = 'BHM';

export const MAX_SEATS = 20;
```

```
// app.js
import { sayHello } from './myStuff.js';

sayHello('Jane');
```

- To import named exports into a JS file, specify the name/path to the module or other file after the `from` keyword
- Reads as "import this stuff from this module here"
- Can specify one or more individually exported items to bring into this module

Importing All Named Exports

```
// myStuff.js
export function sayHello(name) {
    alert(`Hello ${name}!`);
}

export let cityCode = 'BHM';

export const MAX_SEATS = 20;
```

```
// app.js
import * as helpers from './myStuff.js';

helpers.sayHello('Jane');
```

- You can also use the wildcard syntax to grab all named exports from a module
- Use the as keyword to specify a name (that you came up with) in which to store the imported code
- In this case, helpers will be an object with properties `sayHello`, `cityCode`, and `MAX_SEATS`

Import a Default Export

```
// myStuff.js
export default function sayHello(name) {
    alert(`Hello ${name}!`);
}

export let cityCode = 'BHM';

export const MAX_SEATS = 20;
```

```
// app.js
import greet from './myStuff.js';

greet('Jane');
```

- To import a default export, you simply specify a name in which to store that default export, without the use of curly braces
  - `import { greet }` means "import the named export greet into this module"
  - `import greet` means "import the default export, whatever it is called, into this module and refer to it as `greet`
- To clarify, the name in the import statement is something we came up with; it doesn't have to match the name of the default export (although it can if you'd like)

Browser Support

- At this moment in time, most browsers do not support module import syntax
- To be able to use import/export, you need to use a packager
    - There are many packagers available, and many development frameworks include packager scripts for you