Advanced Javascript Features

ES6, ES7 and Beyond

Some Javascript History (Pt 1)

- When Javascript began to see browser adoption, a standard was created to make sure browsers implemented the same features in the same way
- This standard was known as **ECMAScript**, established in 1997
- By 1999, ECMAScript version 3 (ES3) had been established
- It added Regular Expressions, try / catch, and a ton of handy functionality
- It would stay the gold standard for another 10 years…

Some Javascript History (Pt 2)

- In the year 2009, ES5 becomes the new standard for browsers
  - We skipped ES4 due to a bunch of silly politics and arguing
- It added better JSON support, and a ton of handy functions for better object oriented programming and arrays / objects
- Most Javascript code you find examples of is ES5, as this standard has been around the longest in the modern web
- And it would remain the standard for another 6 years…

Some Javascript History (Pt 3)

- However, in the last 3 years, Javascript development has exploded
- This has resulted in a new ES version *every year*
- So much so, that many have stopped referring to versions by number, and instead by year
    - ES8 is often referred to as ES2017
- In addition, many of the proposed features for future versions get created and used before they're even a part of the standard

Feature Overview

- Today we're going to explore a bunch of the new features in Javascript
- We won't cover them all today as there are just too many, but we'll look at some of the most handy and commonly used ones
    - Template strings
    - Arrow functions
    - Default parameters
    - Object shorthand
    - Array methods
    - Spread operators
    - Destructuring assignments

Template Strings

- Template strings allow you to write *multi-line* strings and use string interpolation
- You do so by using the "backtick" character (Left of the 1 key) instead of a single or double quote
- String interpolation allows you to place variables or small javascript snippets directly inside of strings, instead of having to add them
- Simple wrap it in ${} and it'll put itself in the right place

```
// Regular string
const oldStr = "Hello " + student + ", I'm an old string.";

// Template string
const newStr = `And I'm a new string. Much better, huh ${student}?`;
```

Template Strings - Example 1

- String interpolation makes for smaller code, and makes it easier to deal with spaces in sentences

```
function shoutGreeting(name) {
    // Regular string
    return "HELLO " + name.toUpperCase() + "!";

    // Template string
    return `HELLO ${name.toUpperCase()}!`;
}
```

Template Strings - Example 2

- Multi line strings are a pain in regular strings
  - We have to use the \r\n character to insert them
  - We have to split the code into multiple concatenations

```
// Multi-line strings
function getHaiku(text, color) {
    // Regular string
    let haiku = "Another error.\r\n";
    haiku += "Hours of coding go by.\r\n";
    haiku += "Run code and repeat.";
    return haiku;

    // Template string
    return `Another task down.
        Hours of coding go by.
        Run code and code more.`;
}
```

Template Strings - Challenge

- Convert this to a template string
- Break it into multiple lines if you need to!

```
function madlib(adjective, noun1, noun2) {
  let sentence = "The very <em>" + adjective + "<em>";
  sentence += "<em>" + noun1 + "</em> went to the store";
  sentence += "and bought a <em>" + noun2 + "</em>. ";
  sentence += "I didn't know stores sold those."
  return sentence;
}
```

Try this out at https://codepen.io/wbobeirne/pen/jwRERL

Arrow Functions

- Arrow functions are a shorthand for defining anonymous functions
- Just leave off the word `function`, and put an arrow (`=>`) between the parens and curly brace

```
// Old function
app.get(function(req, res) {
    /* My route */
});

// Arrow function
app.get((req, res) => {
    /* My route */
});
```

Arrow Functions - This Keyword

- Arrow functions come with a slight change to regular anonymus functions, **they automatically** `.bind(this)`
- Most often they're used because they're shorter, but it may be intentional, or you may be making a mistake!

```
const UserForm = {
    init: () => {
        this.form = $("#user-form");

        // Old function
        this.form.on("submit", function() {
            console.log(this.form); // undefined
        });

        // Arrow function
        this.form.on("submit", () => {
            console.log(this.form); // <form>
        });
    },
}
```

Arrow Functions - Implicit Return

- Arrow functions have an even shorter syntax, no curly braces
- This is limited to one line functions, and returns the value "implicitly"
    - This means, without the `return` keyword being needed

```
// Sort into descending order
const numbers = [10, 8, 200, 9, 47];

// Old function
numbers.sort(function(a, b) {
    return b - a;
});

// Arrow function
numbers.sort((a, b) => b - a);
```

Arrow Functions - Challenge

- Figure out what's wrong with these two events and fix them

```
// See codepen for the full code snippet...
const WindowReporter = {
    bind: function() {
        this.report = $("#report");

        // Event one
        $(window).on("resize", function() {
            this.reportSize();
        });

        // Event two
        this.report.on("click", () => {
            this.html("Hey, no clicking!");
        });
    },

    reportSize: function() { /* report the size */ }
}
```

Available at https://codepen.io/wbobeirne/pen/RgOPPr

Object Shorthand

- A minor change, but confusing if you don't know about it, is object shorthand
- When defining an object, keys are often the same name as local variables
- And functions as keys are common as well (Think singletons)
- Now both can be shortened

```
const key = "I'm a key!";

// Old object
const oldObject = {
    key: key,
    printKey: function() {
        console.log(this.key);
    },
};

// New object shorthand
const newObject = {
    key,
    printKey() {
        console.log(this.key);
    },
}
```

Object Shorthand - Keys

- Leaving out the `: value` part of defining a key makes the key look for a variable of the same name
- Picking a key name of which there is no variable will give you a `ReferenceError`, just like using an undefined variable

```
/**
 * Given a user ID, returns a promise that resolves with
 * {
 *   firstName: "Barry",
 *   lastName: "White",
 *   fullName: "Barry White",
 *   lastInitial: "W.",
 * };
 */
function getNameById(id) {
    User.findById(id).then((user) => {
        const firstName = user.get("firstName");
        const lastName = user.get("lastName");
        const fullName = `${firstName} ${lastName}`;
        const lastInitial = `${lastName.substring(0, 1)}.`;

        return { firstName, lastName, fullName, lastInitial };
    });
}
```

Object Shorthand - Methods

- We can shorten "`key: function() {}`" to simply "`key() {}`"
- This behaves completely the same, so `this` still refers to the singleton

```
const MySingleton = {
    methodOne() {
        // Do some stuff...
    },
    methodTwo(arg1, arg2) {
        // Do some stuff...
    },
};
```

Array Methods

- A lot of data that we deal with comes in arrays
- So far, we've always been using `for` loops to deal with them
- However, a *ton* of handy array methods were added to make this much nicer
- We'll be looking at `forEach`, `map`, `reduce`, and `filter`

Array Methods - `array.forEach`

- If we want to do something to every element in an array, `forEach` is a quicker way to do so
- It calls a function on every element in the array, passing the array (and index) as arguments

```
// Old arrays
for (let i = 0; i < array.length; i++) {
    console.log(array[i]);
}

// New forEach
array.forEach((element, idx) => {
    console.log(element); // Whatever array[idx] would be
    console.log(idx);     // 0, 1, 2 etc.
});
```

Array Methods - `array.map`

- Sometimes the purpose of looping over an array is to convert it to an equal length array, just with different values
- Using `array.map` is a good way to do that quickly
- It functions the same as `forEach`, except each function returns the new value for that index
- It returns the newly formed array, without affecting the old one

```
const people = [{ name: "John", age: 23 }, /* ... */ ];
const names = people.map((person) => person.name);

console.log(names); // ["John", "George", ...]
```

Array Methods - `array.reduce`

- Another common use case of loops is converting all of the items in an array to a single value
- Reduce will do so by passing a "previous" value, in addition to the element
- The previous value will keep changing every loop, and be returned at the end
- In addition to a function, `reduce` takes an initial value as its second argument

```
const numbers = [1, 2, 3];
const sum = numbers.reduce((total, number) => {
    console.log(`Total is ${total} so far...`);
    return total + number
}, 0);
console.log(`Sum is ${sum}!`);

// Total is 0 so far...
// Total is 1 so far...
// Total is 3 so far...
// Sum is 6!
```

Array Methods - Challenge

- Convert all of the examples in this Codepen from for loops to array methods
- Make sure the console output is EXACTLY the same!

https://codepen.io/wbobeirne/pen/yXWyZX

Sidenote - Array Method Chaining

- Much like jQuery or `res` methods, array methods can be chained
- This is not a requirement, but you may often see code that does so

```
const sentence = array.map((item) => {
        /* Convert "word" objects into strings */
    })
    .filter((prev, item) => {
        /* Remove curse words */
    })
    .reduce((prev, item) => {
        /* Re-combine strings into one sentence */
    }, "");

console.log(sentence); // Some string
```

Array Methods - `array.filter`

- The final method we'll talk about is `filter`
- This behaves the same as `map`, only the new array only has values that were returned `true`
- That way, you can remove unwanted elements from an array

```
const numbers = [1, 2, 3, 4, 5];
const oddNumbers = numbers.filter((number) => number % 2 === 1);

console.log(oddNumbers); // [1, 3, 5]
```

Default Parameters

- Many functions we make should have a "default" if nothing is provided
- Normally we handle that in the function, but now we have a shorthand
- Defaults will be overridden if anything is passed, unless you pass `undefined`
  - However, `null` will still be `null` if it's passed

```
function ghostRideThe(noun = "whip") {
    console.log(`Ghost ride the ${noun}`);
}

ghostRideThe();          // "Ghost ride the whip"
ghostRideThe(null);      // "Ghost ride the null"
ghostRideThe(undefined); // "Ghost ride the whip"
ghostRideThe("taco");    // "Ghost ride the taco"
```

Spread Operators

- Often times, we want to break arrays and objects into all of their pieces
- To do that now, we'd have to loop over them to do what we want
- Using the spread operator by putting . . . before one gets us all of the pieces
- This has a lot of uses that we'll look at in the future, but the most useful one for now is copying arrays and objects

```
const arrayOne = [1, 2, 3];
const arrayBefore = [...arrayOne, 4];
const arrayAfter = [0, ...arrayOne];

console.log(arrayOne);    // [1, 2, 3]
console.log(arrayBefore); // [1, 2, 3, 4]
console.log(arrayAfter);  // [0, 1, 2, 3]
```

Spread Operators - Objects

- This copying ability also applies to objects
- However, objects can choose which key to take by where they but the spread operator

```
const objOne = {
    a: 1,
    b: 2,
};
const objTwo = {
    ...objOne,
    a: "banana",
};
const objThree = {
    a: "salmon",
    ...objOne,
};

console.log(objOne);   // { a: 1, b: 2 }
console.log(objTwo);   // { a: "banana", b: 2 }
console.log(objThree); // { a: 1, b: 2 }
```

Destructuring Assignments

- We often want to refer to attributes of objects, or particular indexes in arrays
- However, listing out `object.attribute1.attribute2...` can get pretty long
- We can quickly make variables out of them by **destructuring** them

```
const myObject = { propertyOne: 1, propertyTwo: 2 };
const myArray = [1, 2, 3];

// Object Destructuring
const { propertyOne, propertyTwo } = myObject;

// Array Destructuring
const [first, second] = myArray;
```

Destructuring Assignments - Example

```
// Old way
if (!this.elements.title.value || !this.elements.description.value) {
    alert("Missing input!");
    return false;
}
else {
    submit(this.elements.title.value, this.elements.description.value);
    return true;
}
```

Destructuring Assignments - Example

```
// Destructuring way
const { title, description } = this.elements;

if (!title.value || !description.value) {
    alert("Missing input");
    return false;
}
else {
    submit(title.value, description.value);
    return true;
}
```

Final Challenge - Convert This Project

- Take this small project here and **clone** the repository
- Create a branch for your conversion (Put your name in the branch)
- Convert the following old-style javascript using the new features we learned
- When you're finished, and it still works as it did before, push your branch and make a new **pull request**
- If I leave comments, fix them! Otherwise, I'll mark it as all good.

https://github.com/wbobeirne/nycda-advanced-js-challenge