Advanced JavaScript / ES6

ForEach, Map and Reduce

`forEach` and `map` are JavaScript array methods that are used to iterate and execute functions on elements of an array. For example, let's say we have an array.

```
var arr = ['Apple', 'Banana', 'Orange'];
```

And we would like to alert all 3 elements of that array. In "classic" JavaScript, you would use a `for` loop:

```
for (i = 0; i < arr.length; i++) {
  alert(arr[i]);
}
```

Or you could use a `forEach`:

```
arr.forEach(function(e) {
  alert(e);
});
```

This works out nicely, because, as we will see later, you can tidy this up with *arrow functions*:

```
arr.forEach(e => alert(e));
```

The JavaScript `map` method is similar — it too can iterate and perform functions on elements of an array. The key difference is that `map` is used to create a new array based on an operation of another array's values, whereas `forEach` simply just executes a function on each of the array's elements without returning any new values. For example, if you wanted to instead generate a new array that contains the upper case values of the elements from an old array, you would write the following using `map`:

```
var arr = ['Apple', 'Banana', 'Orange'];

var arrUpperCase = arr.map(function(e) {
  return e.toUpperCase();
});

console.log(arrUpperCase);
// ["APPLE", "BANANA", "ORANGE"]
```

As we'll see later, we can also tidy this up using *arrow functions*:

```
var arrUpperCase = arr.map(e => e.toUpperCase());
```

The `this` *keyword*

The `this` keyword refers to the context of a function. The value of `this` depends largely upon the scope that it's called in. While the `this` keyword often represents a source of confusion within JavaScript, it's role is relatively simple — the `this` keyword represents the value of the object that invokes the function.

For a simple function on the global that's not called within an object or outer function, `this` will return the `window` (for Web browsers) or `global` (for NodeJS).

```
function f1() {
  return this;
}

console.log(f1()); // window
```

However, if strict mode is enabled, `this` will return undefined if the function is defined globally and not called within the scope of an object or outer function.

```
'use strict';

function f1() {
  return this;
}

console.log(f1()); // undefined
```

The `this` keyword is commonly used to refer to the scope of the calling object. In this way, functions can be written as methods of an object that use the `this` keyword to access the members of that object.

```
var dog = {
  name: 'Fido',
  favouriteFood: 'cheesecake',
  description: function() {
    return 'My doggie ' + this.name + ' absolutely loves ' +
this.favouriteFood + '.';
  }
};

dog.description(); // My doggie Fido absolutely loves cheesecake.
```

There are some cases where the `this` keyword might not work as expected. The most common scenario is that you may pass a function or object method as a callback that relies on `this` to access data (such as members or fields for that object). Let's say we refactor the above code so that we add a `print` method to the `dog` object, and assign a click handler to a button so that when the user clicks on a button, it invokes the `dog.print()` function. We do this by passing `dog.print` as a callback to the button's `addEventListener` method.

```
var dog = {
  name: 'Fido',
  favouriteFood: 'cheesecake',
  description: function() {
    return 'My doggie ' + this.name + ' absolutely loves ' +
this.favouriteFood + '.';
  },
  print: function() {
    console.log('Hello, my name is ' + this.name);
  }
};

var btn = document.getElementById('btn');
btn.addEventListener('click', dog.print);
```

In this case, the function doesn't work as expected: when the print method as invoked by the callback, the `this.name` property is not accessible. This is because the function was called from the global scope, rather than the scope of the object.

To resolve this situation, we would need to explicitly tell the function that `this` refers to the object of `dog`, rather than the global scope. We do this by using the bind method:

```
btn.addEventListener('click', dog.print.bind(dog));
```

In this case, `this` within the context of the `print()` method refers to dog, because we made that explicit using the `bind()` method, and the function works as expected.

Arrow Functions

Arrow functions are a less verbose way of writing functions in JavaScript, particularly if they perform a simple task. For example, these:

```
var greeter = function(message, name) {
  return message + ' ' + name;
}

var squared = function(num) {
  return num * num;
}
```

Could be refactored as these:

```
var greeter = (message, name) => message + ' ' + name;
var squared = num => num * num;
```

For the `squared` function, the parentheses around the parameter list are optional due to the fact that there is only the single `num` parameter.

Unlike regular function expressions, arrow functions do not create a new function context `this`. Arrow functions instead inherit the function context `this` from where it was called. In the following example, the expression `var that = this` was necessary so that the correct context could be made available within the inner function.

```
var deliveryBoy = {

  name: 'John',

  handleMessage: function(message, handler) {
    handler(message);
  },

  receive: function() {
    var that = this; // required to access name property
    this.handleMessage('Hello, ', function(message) {
      console.log(message + that.name);
    });
  }

};
```

If we refactor the above code to use arrow functions, not only do we produce code that is more terse and readable, but it no longer means that we need to pass in the context via assignment. This is because, unlike regular function expressions, arrow functions do not create a new function context — `this` instead refers to the current enclosing lexical context.

```
var deliveryBoy = {

  name: 'John',

  handleMessage: function(message, handler) {
    handler(message);
  },

  receive: function() {
    this.handleMessage('Hello, ', message => console.log(message +
this.name));
  }

};
```

The `var` *and* `let` *keywords*

In JavaScript, variable assignments are done via `var`. Variables assigned using `var` are scoped and 'protected' within functions, as demonstrated via this example:

```
var name = 'Tim';
var greeting = 'hello';
var sayItInDanish = true;

function displayMessage() {
  var greeting = 'hej';
  console.log(greeting + ' ' + name);
}

console.log(greeting + ' ' + name); // hello Tim
displayMessage();                   // hej Tim
```

Variables assigned using `var`, however, not block scoped. Let's say we refactor the code to the following:

```
var name = 'Tim';
var greeting = 'hello';
var sayItInDanish = true;

if(sayItInDanish) {
  var greeting = 'hej';
  console.log(greeting + ' ' + name); // hej Tim
}

console.log(greeting + ' ' + name); // hej Tim (unexpected result)
```

In both cases, the output is 'hej Tim' because the variable assignment within the if block does not have its own scope — it instead changes the variable that was assigned prior to the block. We can fix this by using the `let` keyword.

```
let name = 'Tim';
let greeting = 'hello';
let sayItInDanish = true;

if(sayItInDanish) {
  let greeting = 'hej';
  console.log(greeting + ' ' + name); // hej Tim
}

console.log(greeting + ' ' + name); // hello Tim
```

In this example, the initial `greeting` variable and the `greeting` variable inside the `if` block are treated as seperate entities. This is because — unlike variables assigned with `var` — variables assigned with `let` are block scoped.

The const keyword

Previous versions of JavaScript do not natively support constants. The common convention was to write variable names in upper case that you would wish to remain constant. Although this is a useful convention, it doesn't actually prevent the variable from being mutable.

```
var FAVOURITE_ICECREAM_FLAVOUR = 'chocolate';
FAVOURITE_ICECREAM_FLAVOUR = 'strawberry'; // re-assignment
var message = 'I love ' + FAVOURITE_ICECREAM_FLAVOUR + ' ice cream';
console.log(message);
```

In the above example, FAVOURITE_ICECREAM_FLAVOUR does not behave like a true constant because it is re-assigned. We can overcome this using the const keyword:

```
const FAVOURITE_ICECREAM_FLAVOUR = 'chocolate';
FAVOURITE_ICECREAM_FLAVOUR = 'strawberry'; // error
var message = 'I love ' + FAVOURITE_ICECREAM_FLAVOUR + ' ice cream';
console.log(message);
```

The above code will produce an error because it is attempting to change the value of a constant. Note that const only enforces immutability by reference: while it is not possible to change a string, number or object reference, it is still possible to alter the member variables of an object assigned to a constant, given that the reference to that object remains the same.

```
const person = {
  name: 'Tim',
  location: 'Denmark'
};

person.name = 'John'; // OK - const's members are mutable
person = 'Alex';      // Error - changing the reference of the const
```

Default Values for Function Parameters

In ES6, you can assign default values for functions. For example, let's consider a function that allows you to calculate the nth power of a number, but squares the number by default. We can write a function that takes two parameters: a mandatory number (the base), and an exponent (which is 2 for squared (default), 3 for cubed, etc.). In essence, the function is a simple wrapper for the native Math.pow function.

```
function pow(base, exponent = 2) {
  return Math.pow(base, exponent);
}

pow(4);    // 16 - squares the number by default
pow(4, 3); // 64 - second parameter indicates that the number is cubed
```

We can simplify this further using arrow function syntax:

```
let pow = (base, exponent = 2) => Math.pow(base, exponent);
```

In addition to assigning strings and numbers as default values, you can also assign functions. The following example demonstrates a simple logging function. The first call to the function logs the output to the console. The second call passes in a custom logging method that instead prints it to the current web page.

```
function log(string, logFn = function(string) {
  console.log(string);
}) {
  logFn(string);
}

// Log to the console
log("hello, I'm in your console");

// Log, but with a custom logging function
log("hello, I'm on your web page", function(string) {
  document.write(string);
});
```

We can also refactor the above code to use arrow functions:

```
let log = (string, logFn = (string) => console.log(string)) =>
logFn(string);

// Log to the console
log("hello, I'm in your console");

// Log, but with a custom logging function
log("hello, I'm on your web page", string => document.write(string));
```

Template Literals

Template literals provide a nice alternative to building strings using the + operator.

Consider the following code, which uses string concatenation to construct the introduction:

```
var name = 'Anders';
var dateOfBirth = new Date("21 Jan 1996");

var introduction = "Hello, my name is " + name + " and I was born in " +
dateOfBirth.getFullYear() + ".";

console.log(introduction);
// "Hello, my name is Anders and I was born in 1996"
```

We can alter the introduction variable to clean this up using template literals:

```
var introduction = `Hello, my name is ${name} and I was born in $
{dateOfBirth.getFullYear()}.`
```

It's also worthwhile to note that you can use expressions inside template literals:

```
var name = 'Anders';
var dateOfBirth = new Date("21 Jan 1996");

var introduction = `Hello, my name is ${name} and I am ${new
Date().getFullYear() - dateOfBirth.getFullYear()} years old.`

console.log(introduction);
// "Hello, my name is Anders and I am 22 years old."
```

The spread operator and the rest parameter

The spread operator is a very useful tool that allows us to easily work with the members of an array. The spread operator is used to extract the members of an array without the need to use loops, such as `for`, `forEach` and `map`.

```
var arr = [1, 2, 3];

console.log(arr);    // [1, 2, 3]
console.log(...arr); // 1 2 3
```

The first console.log() call references the array itself, whereas the second console.log() call uses the spread operator to return the members of an array. This feature is useful if we wish to work directly with the members of an array, such as having the ability to concatenate two arrays together. In the following example, it seems that we are concatenating arr1 with arr2, but in fact we actually nest arr2 as a final element of arr1:

```
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

arr1.push(arr2);

console.log(arr1); // [1, 2, 3, [4, 5, 6]]
```

We can use a loop to individually add the elements of arr1 to arr2:

```
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

arr2.forEach(e => arr1.push(e));

console.log(arr1); // [1, 2, 3, 4, 5, 6]
```

Or more simply, we can use the spread operator to destructure the array and add the elements individually:

```
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

arr1.push(...arr2);

console.log(arr1); // [1, 2, 3, 4, 5, 6]
```

The spread operator can also be used to pass an array as parameters of a function:

```
var arr = [1, 2, 3];

let sum = (num1, num2, num3) => num1 + num2 + num3;

sum(arr[0], arr[1], arr[2]); // Sum by calling members of arr
sum(...arr);                 // Sum by spreading members of arr
```

Our sum function currently adds a maximum of three numbers. But what if we wanted to modify our `sum` function so that we can add a set of numbers of any size? Using the same syntax as the spread operator, we can use the rest parameter to specify that our function takes N number of arguments, and then reference that parameter as an array within the function.

```
var arr = [4, 5, 6, 7, 8, 10];

// Use the rest operator to access N parameters,
// then reduce them to retrieve their sum
let sum = (...args) => args.reduce((total, val) => total += val);

sum(...arr); // Sum by spreading members of arr
```