Programming in JavaScript

Section 3: ES6 and JavaScript

3.1 Const and Let Variables

Const and Let variables are a new feature that is introduced into ES6. The syntax is exactly the same as the var variable keyword; however, the behaviour are slightly different and will throw an error depending on the variable type used.

A const variable must be initialised when being declared i.e. we must assign it a value otherwise we would get an error.

A const or constant variable can store a value just like a variable. However, when declaring a const variable we must assign it a value (unlike var which does not require it to be initialised when declared).

```
var name;
name = "John";
```

```
const name; SyntaxError: Missing initializer in const declaration
```

```
const name = "John";
```

The var value can change its value anytime. The const variable on the other hand does not allow you to change the value of the constant variable and will result an error.

```
var name = "John";
name = "Mary";
```

```
const name = "John";
name = "Mary";       TypeError: Assignment to constant variable.
```

Therefore, when declaring a const variable we must assign it a value when it is declared and we cannot re-assign it a new value at any point in the code hence, as the name indicates, it has a constant value.

The let variable is almost the same as var but similar to const variables there is a difference between var and let variables. A var variable has two scopes a global and a function scope. The let variable has three scopes a global scope, function scope and a block scope.

When we declare a variable outside a function the variable is said to be declared in the global scope. This means any child functions of the code can access this global variable.

```
var global = "I am a global variable";

function sayMyName() {
    console.log(global);
};

sayMyName();                    I am a global variable
console.log(global);            I am a global variable
```

A local scope (or a function scope) is a variable that is only available in the function and is not available outside in the global scope i.e. it is only local to the function it was declared in. The below code will throw a ReferenceError on console.log(local) because the global scope does not have access to the function variable outside the local scope. Therefore, the variable can only be manipulated inside of the function where it was declared.

```
function sayMyName() {
    var local = "I am a local variable";
    console.log(local);
};

sayMyName();
console.log(local);
```

```
ReferenceError: local is not defined
```

The let variable functions exactly the same as the var variable i.e. it can be initialised at a later point in time and has both the global and function scope.

```
let global;
global = "I am a global variable";

function sayMyName() {
    let local = "I am a local variable";
    console.log(global);
    console.log(local);
};

sayMyName();
console.log(global);
// console.log(local);
```

The console.log(local) code will throw a ReferenceError as seen above example for var. As we can see the let and var variables work exactly the same.

```
I am a global variable
I am a local variable
I am a global variable
```

The only difference as mentioned above is that the let variable has a third scope which is called the block scope. In JavaScript a block is anything that starts and closes with curly brackets ({ }). The code contained in the curly braces is part of that block of code.

The let variable inside of a block can only be accessed in the block scope and cannot be accessed outside of the block i.e. in the global scope (as seen below this will return a ReferenceError).

```
{
    let block = "I am a block scope variable";
    console.log(block);
};

console.log(block);
```

```
ReferenceError: block is not defined
```

When the block code completes its execution the let variable no longer exists which is why it gives the ReferenceError. This behaviour does not apply to var variables as demonstrated in the below example:

```
{
    var block = "I am a block scope variable";
    console.log(block);
};

console.log(block);
```

```
I am a block scope variable
I am a block scope variable
```

Finally, it is important to note that a let and var variable cannot have the same name (no variables can be declared with the same name) and this will throw a SyntaxError to inform you that the variable name has already been declared. In the below example the variable name is called ‘variable’ and this will change to whatever the name of the variable is that has already been declared.

```
SyntaxError: Identifier 'variable' has already been declared
```

3.2 Block Scope Functions

When a function is created outside in the global scope and another function with the same name is created in a block scope, the new block scoped function will override the global scoped function.

Remember a block scope is anything that starts and ends with the curly brackets e.g. if statements, while, do while and for each loops.

In this example the foo( ) function in the block scope has overwritten the foot( ) function in the global scope and will print 'bar' in the console twice.

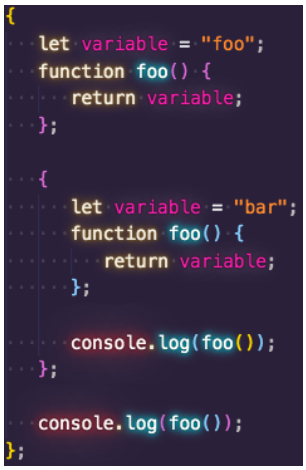
Note the same does not apply in the vice versa i.e. if we declare a function inside of the block scope first and then in the global scope declare a function with the same name, this will not override the block scope.

```
bar
bar
```

When we have a block inside another block (nested blocks) the same behaviour occurs i.e. whereby the new function with the same name overrides the previous function as see in the example below:

The console.log( ) in the inner block returns 'bar' to the terminal/ console because the inner foo function overrides the outer block foo function.

However, the console.log( ) in the outer main block returns 'foo' in the terminal/console and is not overwritten by the inner block function.



The reason for this behaviour is because the inner block code i.e. functions will only exist in the inner block execution and will override the outer block function with the same name. Once the inner block execution completes the inner function (code block) no longer exists. Therefore, the function in the outer block is now the only function that exists in the block execution.

Therefore, when working with inner block functions that have functions with the same name as the outer block functions we need to pay special attention to this behaviour of JavaScript. This behaviour also applies to variables within block scope.

```
bar
foo
```

Important Note: If the inner block 'variable' did not have the 'let' keyword then this will overwrite the outer blocks let variable and would end up printing 'bar' twice.

3.3 Optional (Default Value) Parameters

Before ES6 when a function had parameters it was required to pass in all the function parameters within the round brackets when calling the function.

```
function peopleToString(name, age) {
    return "Your name is: " + name + " and your are " + age + " years old.";
};

console.log(peopleToString("John", 35));
```

```
Your name is: John and your are 35 years old.
```

Now in ES6 if a parameter is not passed into a function the parameters will be replaced by the undefined data types as seen below:

```
console.log(peopleToString());
```

```
Your name is: undefined and your are undefined years old.
```

ES6 also allows us to create optional parameters. To do this we simply assign a default value to the optional parameter(s) as demonstrated below. This will now use the default value unless the parameter is passed in which will override the default value.

```
function peopleToString(name, age = 30) {
    return "Your name is: " + name + " and your are " + age + " years old.";
};

console.log(peopleToString("John"));
console.log(peopleToString("John", 35));
```

```
Your name is: John and your are 30 years old.
Your name is: John and your are 35 years old.
```

3.4 Spread Operator

The spread operator (…) within the function parameters allows the function to receive an unlimited parameters. To have an unlimited parameters we would use the spread operator followed by the name of the parameter. This will act as an array where we can pass in any number of parameters.

```
function people(name, age = 45, ...family) {
    console.log(family);
};

people("John Doe", 35, "Julie Doe", "Barry Doe");
```

In this example, every other parameter values passed in after the age parameter will be added into the family array.

```
[ 'Julie Doe', 'Barry Doe' ]
```

This is a very powerful operator to make function parameters dynamic and have an unlimited amount of parameter that can be passed in based on the scenario. The spread operator in JavaScript is used with arrays. We can loop through the array to do some interesting things with the data.

```
function people(name, age = 45, ...family) {
    console.log("Your name is " + name + " and you are " + age + " years old. Your family members are:");
    for(var i = 0; i < family.length; i++ ) {
        console.log(family[i]);
    };
};

people("John Doe", 35, "Julie Doe", "Barry Doe");
```

```
Your name is John Doe and you are 35 years old. Your family members are:
Julie Doe
Barry Doe
```