Lesson 3.1 - Constants and Let

ES2015 introduced a new variable type: `const`. This new variable type should not be confused with constants in other languages as it doesn't mean the value is constant at all! It only means that the variable cannot be reassigned, therefore it has a constant reference. This is illegal:

```
const x = 1
x = 2 // Illegal
```

However, if the const is assigned an object such as an array, you can mutate the array:

```
const ary = [1, 2, 3]
ary.push(4) // This is allowed
```

You can mutate objects in the same way. This is allowed because you are adding a value to the array or object, and not reassigning. The `const` assignment only prevents you from reassigning the variable to a new value, but does not prevent you from mutating the value.

Another thing to note about the `const` type is that it cannot be used before assignment. Whereas this is perfectly acceptable:

```
typeof bar
var bar = 1
```

This is not allowed:

```
typeof bar
const bar = 1
```

The above will throw an error because you are attempting to access the variable before it has been assigned. In the first example, we are using the `var` keyword, which `hoists` the variable assignment to the top of the execution, whereas the `const` keyword does not do this.

Let

The `let` keyword defines a block scoped variable, just like `const`, however you *can* reassign to a variable declared with 'let'. It has the same properties as described above, and allows you to have finer grained control over the scoping of your variables as opposed to `var`.

The Temporal Dead Zone

The 'Temporal Dead Zone' is a scary sounding term that refers to a new set of ECMAScript semantics regarding scoping that has been introduced in ES2015. Essentially, it refers to a grey area in which undefined variables cannot yet be accessed. Take the following example.

```
foo = 2
let foo = 1
```

In this case, `foo` has not yet been defined, and therefore we can not re-assign it. However, if the variable is accessed inside of a function that has yet to be called, you *can* write code that relies on it before the declaration:

```
const readFoo = function() { return foo }
let foo = 1
```

This is perfectly valid, because the function we created has not yet been invoked. As long as we do not invoke the function before declaring the variable, this will not produce any errors.

However, if we tried to do this:

```
const readFoo = function() { return foo }
readFoo()
let foo = 1
```

This would throw an error because we invoked the `readFoo` function before the `foo` variable was defined.