Explore Differences Between the var and let Keywords

One of the biggest problems with declaring variables with the `var` keyword is that you can overwrite variable declarations without an error.

```
var camper = 'James';
var camper = 'David';
console.log(camper);
// logs 'David'
```

As you can see in the code above, the `camper` variable is originally declared as `James` and then overridden to be `David`. In a small application, you might not run into this type of problem, but when your code becomes larger, you might accidentally overwrite a variable that you did not intend to overwrite. Because this behavior does not throw an error, searching and fixing bugs becomes more difficult.
A new keyword called `let` was introduced in ES6 to solve this potential issue with the `var` keyword. If you were to replace `var` with `let` in the variable declarations of the code above, the result would be an error.

```
let camper = 'James';
let camper = 'David'; // throws an error
```

This error can be seen in the console of your browser. So unlike `var`, when using `let`, a variable with the same name can only be declared once. Note the `"use strict"`. This enables Strict Mode, which catches common coding mistakes and "unsafe" actions. For instance:

```
"use strict";
x = 3.14; // throws an error because x is not declared
```

Compare Scopes of the var and let Keywords

When you declare a variable with the `var` keyword, it is declared globally, or locally if declared inside a function.

The `let` keyword behaves similarly, but with some extra features. When you declare a variable with the `let` keyword inside a block, statement, or expression, its scope is limited to that block, statement, or expression.

For example:

```
var numArray = [];
for (var i = 0; i < 3; i++) {
  numArray.push(i);
```

```
}
console.log(numArray);
// returns [0, 1, 2]
console.log(i);
// returns 3
```

With the `var` keyword, `i` is declared globally. So when `i++` is executed, it updates the global variable. This code is similar to the following:

```
var numArray = [];
var i;
for (i = 0; i < 3; i++) {
  numArray.push(i);
}
console.log(numArray);
// returns [0, 1, 2]
console.log(i);
// returns 3
```

This behavior will cause problems if you were to create a function and store it for later use inside a for loop that uses the `i` variable. This is because the stored function will always refer to the value of the updated global `i` variable.

```
var printNumTwo;
for (var i = 0; i < 3; i++) {
  if (i === 2) {
    printNumTwo = function() {
      return i;
    };
  }
}
console.log(printNumTwo());
// returns 3
```

As you can see, `printNumTwo()` prints 3 and not 2. This is because the value assigned to `i` was updated and the `printNumTwo()` returns the global `i` and not the value `i` had when the function was created in the for loop. The `let` keyword does not follow this behavior:

```
let printNumTwo;
for (let i = 0; i < 3; i++) {
```

```
  if (i === 2) {
    printNumTwo = function() {
      return i;
    };
  }
}
console.log(printNumTwo());
// returns 2
console.log(i);
// returns "i is not defined"
```

`i` is not defined because it was not declared in the global scope. It is only declared within the for loop statement. `printNumTwo()` returned the correct value because three different `i` variables with unique values (0, 1, and 2) were created by the `let` keyword within the loop statement.

Declare a Read-Only Variable with the const Keyword

The keyword `let` is not the only new way to declare variables. In ES6, you can also declare variables using the `const` keyword.

`const` has all the awesome features that `let` has, with the added bonus that variables declared using `const` are read-only. They are a constant value, which means that once a variable is assigned with `const`, it cannot be reassigned.

```
const FAV_PET = "Cats";
FAV_PET = "Dogs"; // returns error
```

As you can see, trying to reassign a variable declared with `const` will throw an error. You should always name variables you don't want to reassign using the `const` keyword. This helps when you accidentally attempt to reassign a variable that is meant to stay constant. A common practice when naming constants is to use all uppercase letters, with words separated by an underscore.

Note: It is common for developers to use uppercase variable identifiers for immutable values and lowercase or camelCase for mutable values (objects and arrays). In a later challenge you will see an example of a lowercase variable identifier being used for an array.

Mutate an Array Declared with const

The `const` declaration has many use cases in modern JavaScript.

Some developers prefer to assign all their variables using `const` by default, unless they know they will need to reassign the value. Only in that case, they use `let`.

However, it is important to understand that objects (including arrays and functions) assigned to a variable using `const` are still mutable. Using the `const` declaration only prevents reassignment of the variable identifier.

```
const s = [5, 6, 7];
s = [1, 2, 3]; // throws error, trying to assign a const
s[2] = 45; // works just as it would with an array declared with var or let
console.log(s); // returns [5, 6, 45]
```

As you can see, you can mutate the object `[5, 6, 7]` itself and the variable `s` will still point to the altered array `[5, 6, 45]`. Like all arrays, the array elements in `s` are mutable, but because `const` was used, you cannot use the variable identifier `s` to point to a different array using the assignment operator.

Prevent Object Mutation

As seen in the previous challenge, `const` declaration alone doesn't really protect your data from mutation. To ensure your data doesn't change, JavaScript provides a function `Object.freeze` to prevent data mutation.

Once the object is frozen, you can no longer add, update, or delete properties from it. Any attempt at changing the object will be rejected without an error.

```
let obj = {
  name:"FreeCodeCamp",
  review:"Awesome"
};
Object.freeze(obj);
obj.review = "bad"; // will be ignored. Mutation not allowed
obj.newProp = "Test"; // will be ignored. Mutation not allowed
console.log(obj);
// { name: "FreeCodeCamp", review:"Awesome"}
```

Use Arrow Functions to Write Concise Anonymous Functions

In JavaScript, we often don't need to name our functions, especially when passing a function as an argument to another function. Instead, we create inline functions. We don't need to name these functions because we do not reuse them anywhere else.

To achieve this, we often use the following syntax:

```
const myFunc = function() {
  const myVar = "value";
  return myVar;
}
```

ES6 provides us with the syntactic sugar to not have to write anonymous functions this way. Instead, you can use **arrow function syntax**:

```
const myFunc = () => {
  const myVar = "value";
  return myVar;
}
```

When there is no function body, and only a return value, arrow function syntax allows you to omit the keyword `return` as well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

```
const myFunc = () => "value";
```

This code will still return the string `value` by default.

Write Arrow Functions with Parameters

Just like a regular function, you can pass arguments into an arrow function.

```
// doubles input value and returns it
const doubler = (item) => item * 2;
doubler(4); // returns 8
```

If an arrow function has a single parameter, the parentheses enclosing the parameter may be omitted.

```
// the same function, without the parameter parentheses
const doubler = item => item * 2;
```

It is possible to pass more than one argument into an arrow function.

```
// multiplies the first input value by the second and returns it
const multiplier = (item, multi) => item * multi;
multiplier(4, 2); // returns 8
```

Set Default Parameters for Your Functions

In order to help us create more flexible functions, ES6 introduces *default parameters* for functions.

Check out this code:

```
const greeting = (name = "Anonymous") => "Hello " + name;

console.log(greeting("John")); // Hello John
console.log(greeting()); // Hello Anonymous
```

The default parameter kicks in when the argument is not specified (it is undefined). As you can see in the example above, the parameter `name` will receive its default value `"Anonymous"` when you do not provide a value for the parameter. You can add default values for as many parameters as you want.

Use the Rest Parameter with Function Parameters

In order to help us create more flexible functions, ES6 introduces the *rest parameter* for function parameters. With the rest parameter, you can create functions that take a variable number of arguments. These arguments are stored in an array that can be accessed later from inside the function.

Check out this code:

```
function howMany(...args) {
  return "You have passed " + args.length + " arguments.";
}
console.log(howMany(0, 1, 2)); // You have passed 3 arguments.
console.log(howMany("string", null, [1, 2, 3], { })); // You have
passed 4 arguments.
```

The rest parameter eliminates the need to check the `args` array and allows us to apply `map()`, `filter()` and `reduce()` on the parameters array.