Introduction to the ES6 Challenges

ECMAScript is a standardized version of **JavaScript** with the goal of unifying the language's specifications and features. As all major browsers and JavaScript-runtimes follow this specification, the term ***ECMAScript*** is interchangeable with the term ***JavaScript***.

Most of the challenges on freeCodeCamp use the ECMAScript 5 (ES5) specification of the language, finalized in 2009. But JavaScript is an evolving programming language. As features are added and revisions are made, new versions of the language are released for use by developers.

The most recent standardized version is called ECMAScript 6 (ES6), released in 2015. This new version of the language adds some powerful features that will be covered in this section of challenges, including:

- Arrow functions
- Classes
- Modules
- Promises
- Generators
- `let` and `const`

Note
Not all browsers support ES6 features. If you use ES6 in your own projects, you may need to use a program (transpiler) to convert your ES6 code into ES5 until browsers support ES6.

ES6: 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
```

ES6: 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.

ES6: Declare a Read-Only Variable with the const Keyword

let is not the only new way to declare variables. In ES6, you can also declare variables using the const keyword.
consthas all the awesome features that lethas, with the added bonus that variables declared using constare read-only. They are a constant value, which means that once a variable is assigned with const, it cannot be reassigned.

```
"use strict"
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.

ES6: 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 re-assign 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 constare still mutable. Using the const declaration only prevents reassignment of the variable identifier.

```
"use strict";
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.

ES6: 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"}
```

ES6: 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 returnas well as the brackets surrounding the code. This helps simplify smaller functions into one-line statements:

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

This code will still return `value` by default.

ES6: Write Arrow Functions with Parameters

Just like a normal function, you can pass arguments into arrow functions.

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

You can pass more than one argument into arrow functions as well.

ES6: Write Higher Order Arrow Functions

It's time we see how powerful arrow functions are when processing data.
Arrow functions work really well with higher order functions, such as **map(), filter()**, and **reduce(),** that take other functions as arguments for processing collections of data.
Read the following code:

```
FBPosts.filter(function(post) {
  return post.thumbnail !== null && post.shares > 100 && post.likes > 500;
})
```

We have written this with filter() to at least make it somewhat readable. Now compare it to the following code which uses arrow function syntax instead:

```
FBPosts.filter((post) => post.thumbnail !== null && post.shares > 100 && post.likes > 500)
```

This code is more succinct and accomplishes the same task with fewer lines of code.

ES6: 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:

```
function greeting(name = "Anonymous") {
  return "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.

ES6: Use the Rest Operator with Function Parameters

In order to help us create more flexible functions, ES6 introduces the rest operator for function parameters. With the rest operator, 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 operator eliminates the need to check the `args` array and allows us to apply `map()`, `filter()` and `reduce()` on the parameters array.

ES6: Use the Spread Operator to Evaluate Arrays In-Place

ES6 introduces the spread operator, which allows us to expand arrays and other expressions in places where multiple parameters or elements are expected.
The ES5 code below uses `apply()` to compute the maximum value in an array:

```
var arr = [6, 89, 3, 45];
var maximus = Math.max.apply(null, arr); // returns 89
```

We had to use `Math.max.apply(null, arr)` because `Math.max(arr)` returns `NaN`. `Math.max()` expects comma-separated arguments, but not an array.
The spread operator makes this syntax much better to read and maintain.

```
const arr = [6, 89, 3, 45];
const maximus = Math.max(...arr); // returns 89
```

`...arr` returns an unpacked array. In other words, it *spreads* the array.
However, the spread operator only works in-place, like in an argument to a function or in an array literal. The following code will not work:

```
const spreaded = ...arr; // will throw a syntax error
```