ES6

Compare Scopes of the var and let Keywords

If you are unfamiliar with `let`, check out this challenge about the difference between `let` and `var`.

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);
console.log(i);
```

Here the console will display the values `[0, 1, 2]` and `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);
console.log(i);
```

Here the console will display the values `[0, 1, 2]` and `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());
```

Here the console will display the value `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());
console.log(i);
```

Here the console will display the value `2`, and an error that `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.

Fix the code so that `i` declared in the `if` statement is a separate variable than `i` declared in the first line of the function. **Be certain not to use the `var` keyword anywhere in your code.**

This exercise is designed to illustrate the difference between how `var` and `let` keywords assign scope to the declared variable. When programming a function similar to the one used in this exercise, it is **often better to use different variable names to avoid confusion**.

Solution:

```
function checkScope() {
  let i = 'function scope';
  if (true) {
    let i = 'block scope';
    console.log('Block scope i is: ', i);
  }
  console.log('Function scope i is: ', i);
  return i;
}
```

Mutate an Array Declared with const

If you are unfamiliar with `const`, check out this challenge about the `const` keyword.

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];
s[2] = 45;
console.log(s);
```

`s = [1, 2, 3]` will result in an error. After commenting out that line, the `console.log` will display the value `[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.**

An array is declared as `const s = [5, 7, 2]`. Change the array to `[2, 5, 7]` using various element assignments.

Solution:

```
const s = [5, 7, 2];
function editInPlace() {
  // Only change code below this line
s[0] = 2;
s[1] = 5;
s[2] = 7;
  // Using s = [2, 5, 7] would be invalid
  // Only change code above this line
}
editInPlace();
```

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.

Any attempt at changing the object will be rejected, with an error thrown if the script is running in strict mode.

```
let obj = {
  name:"FreeCodeCamp",
  review:"Awesome"
};
Object.freeze(obj);
obj.review = "bad";
obj.newProp = "Test";
console.log(obj);
```

The `obj.review` and `obj.newProp` assignments will result in errors, because our editor runs in strict mode by default, and the console will display the value `{ name: "FreeCodeCamp", review: "Awesome" }`.

In this challenge you are going to use `Object.freeze` to prevent mathematical constants from changing. You need to freeze the `MATH_CONSTANTS` object so that no one is able to alter the value of `PI`, add, or delete properties.

Solution:

```
function freezeObj() {
  const MATH_CONSTANTS = {
    PI: 3.14
  };
  // Only change code below this line
Object.freeze(MATH_CONSTANTS);

  // Only change code above this line
  try {
    MATH_CONSTANTS.PI = 99;
  } catch(ex) {
    console.log(ex);
  }
  return MATH_CONSTANTS.PI;
}
const PI = freezeObj();
```

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.

Rewrite the function assigned to the variable `magic` which returns a `new Date()` to use arrow function syntax. Also, make sure nothing is defined using the keyword `var`.

Solution:

```
const magic = () => new Date();
```

Write Arrow Functions with Parameters

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

```
const doubler = (item) => item * 2;
doubler(4);
```

`doubler(4)` would return the value `8`.

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

```
const doubler = item => item * 2;
```

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

```
const multiplier = (item, multi) => item * multi;
multiplier(4, 2);
```

`multiplier(4, 2)` would return the value `8`.

Rewrite the `myConcat` function which appends contents of `arr2` to `arr1` so that the function uses arrow function syntax.

Solution:

```
const myConcat = (arr1, arr2) => arr1.concat(arr2);

console.log(myConcat([1, 2], [3, 4, 5]));
```

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"));
console.log(greeting());
```

The console will display the strings `Hello John` and `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.

Modify the function `increment` by adding default parameters so that it will add 1 to `number` if `value` is not specified.

Solution:

```
const increment = (number, value = 1) => number + value;
```

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));
console.log(howMany("string", null, [1, 2, 3], { }));
```

The console would display the strings `You have passed 3 arguments.` and `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.

Modify the function `sum` using the rest parameter in such a way that the function `sum` is able to take any number of arguments and return their sum.

Solution:

```
const sum = (...args) => {
  return args.reduce((a, b) => a + b, 0);
}
```

```
console.log(sum(5, 3, 10));
```

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);
```

`maximus` would have a value of `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);
```

`maximus` would have a value of `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;
```

Copy all contents of `arr1` into another array `arr2` using the spread operator.

Solution:

```
const arr1 = ['JAN', 'FEB', 'MAR', 'APR', 'MAY'];
let arr2;

arr2 = [...arr1];  // Change this line

console.log(arr2);
```

Use Destructuring Assignment to Extract Values from Objects

Destructuring assignment is special syntax introduced in ES6, for neatly assigning values taken directly from an object.

Consider the following ES5 code:

```
const user = { name: 'John Doe', age: 34 };


const name = user.name;
const age = user.age;
```

`name` would have a value of the string `John Doe`, and `age` would have the number `34`.

Here's an equivalent assignment statement using the ES6 destructuring syntax:

```
const { name, age } = user;
```

Again, `name` would have a value of the string `John Doe`, and `age` would have the number `34`.

Here, the `name` and `age` variables will be created and assigned the values of their respective values from the `user` object. You can see how much cleaner this is.

You can extract as many or few values from the object as you want.

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables `today` and `tomorrow` the values of `today` and `tomorrow` from the `HIGH_TEMPERATURES` object.

Solution:

```
const HIGH_TEMPERATURES = {
  yesterday: 75,
  today: 77,
  tomorrow: 80
};

// Only change code below this line

const { today, tomorrow } = HIGH_TEMPERATURES;
```

Use Destructuring Assignment to Assign Variables from Objects

Destructuring allows you to assign a new variable name when extracting values. You can do this by putting the new name after a colon when assigning the value.

Using the same object from the last example:

```
const user = { name: 'John Doe', age: 34 };
```

Here's how you can give new variable names in the assignment:

```
const { name: userName, age: userAge } = user;
```

You may read it as "get the value of `user.name` and assign it to a new variable named `userName`" and so on. The value of `userName` would be the string `John Doe`, and the value of `userAge` would be the number `34`.

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables `highToday` and `highTomorrow` the values of `today` and `tomorrow` from the `HIGH_TEMPERATURES` object.

Solution:

```
const HIGH_TEMPERATURES = {
  yesterday: 75,
  today: 77,
  tomorrow: 80
};

// Only change code below this line

const { today: highToday, tomorrow: highTomorrow } =HIGH_
TEMPERATURES;
```

Use Destructuring Assignment to Assign Variables from Nested Objects

You can use the same principles from the previous two lessons to destructure values from nested objects.

Using an object similar to previous examples:

```
const user = {
  johnDoe: {
    age: 34,
    email: 'johnDoe@freeCodeCamp.com'
  }
};
```

Here's how to extract the values of object properties and assign them to variables with the same name:

```
const { johnDoe: { age, email }} = user;
```

And here's how you can assign an object properties' values to variables with different names:

```
const { johnDoe: { age: userAge, email: userEmail }} = user;
```

Replace the two assignments with an equivalent destructuring assignment. It should still assign the variables `lowToday` and `highToday` the values of `today.low` and `today.high` from the `LOCAL_FORECAST` object.

Solution:

```
const LOCAL_FORECAST = {
  yesterday: { low: 61, high: 75 },
  today: { low: 64, high: 77 },
  tomorrow: { low: 68, high: 80 }
};

// Only change code below this line

const { today: {low: lowToday, high: highToday}} = LOCAL_
FORECAST;
```

Use Destructuring Assignment to Assign Variables from Arrays

ES6 makes destructuring arrays as easy as destructuring objects.

One key difference between the spread operator and array destructuring is that the spread operator unpacks all contents of an array into a comma-separated list. Consequently, you cannot pick or choose which elements you want to assign to variables.

Destructuring an array lets us do exactly that:

```
const [a, b] = [1, 2, 3, 4, 5, 6];
```

```
console.log(a, b);
```

The console will display the values of `a` and `b` as `1, 2`.

The variable `a` is assigned the first value of the array, and `b` is assigned the second value of the array. We can also access the value at any index in an array with destructuring by using commas to reach the desired index:

```
const [a, b,,, c] = [1, 2, 3, 4, 5, 6];
console.log(a, b, c);
```

The console will display the values of `a`, `b`, and `c` as `1, 2, 5`.

Use destructuring assignment to swap the values of `a` and `b` so that `a` receives the value stored in `b`, and `b` receives the value stored in `a`.

Solution:

```
let a = 8, b = 6;
// Only change code below this line
[a, b] = [b, a];
```

Use Destructuring Assignment with the Rest Parameter to Reassign Array Elements

In some situations involving array destructuring, we might want to collect the rest of the elements into a separate array.

The result is similar to `Array.prototype.slice()`, as shown below:

```
const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];
console.log(a, b);
console.log(arr);
```

The console would display the values `1, 2` and `[3, 4, 5, 7]`.

Variables `a` and `b` take the first and second values from the array. After that, because of the rest parameter's presence, `arr` gets the rest of the values in the form of an array. The rest element only works correctly as the last variable in the list. As in, you cannot use the rest parameter to catch a subarray that leaves out the last element of the original array.

Use a destructuring assignment with the rest parameter to emulate the behavior of `Array.prototype.slice()`. `removeFirstTwo()` should return a sub-array of the original array `list` with the first two elements omitted.

Solution:

```
function removeFirstTwo(list) {
  // Only change code below this line
  const [a,b, ...shorterList] = list; // Change this line
  // Only change code above this line
  return shorterList;
}

const source = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const sourceWithoutFirstTwo = removeFirstTwo(source);
```

Use Destructuring Assignment to Pass an Object as a Function's Parameters

In some cases, you can destructure the object in a function argument itself.

Consider the code below:

```
const profileUpdate = (profileData) => {
  const { name, age, nationality, location } = profileData;
}
```

This effectively destructures the object sent into the function. This can also be done in-place:

```
const profileUpdate = ({ name, age, nationality, location }) =>
{


}
```

When `profileData` is passed to the above function, the values are destructured from the function parameter for use within the function.

Use destructuring assignment within the argument to the function `half` to send only `max` and `min` inside the function.

Solution:

```
const stats = {
  max: 56.78,
  standard_deviation: 4.34,
  median: 34.54,
  mode: 23.87,
  min: -0.75,
  average: 35.85
};

// Only change code below this line
const half = ({ max, min }) => (max + min) / 2.0;
// Only change code above this line
```

Create Strings using Template Literals

A new feature of ES6 is the *template literal*. This is a special type of string that makes creating complex strings easier.

Template literals allow you to create multi-line strings and to use string interpolation features to create strings.

Consider the code below:

```
const person = {
  name: "Zodiac Hasbro",
  age: 56
};

const greeting = `Hello, my name is ${person.name}!
I am ${person.age} years old.`;

console.log(greeting);
```

The console will display the strings `Hello, my name is Zodiac Hasbro!` and `I am 56 years old.`.

A lot of things happened there. Firstly, the example uses backticks (`` ` ``), not quotes (`'` or `"`), to wrap the string. Secondly, notice that the string is multi-line, both in the code and the output. This saves inserting `\n` within strings. The `${variable}` syntax used above is a placeholder. Basically, you won't have to use concatenation with the `+` operator anymore. To add variables to strings, you just drop the variable in a template string and wrap it with `${` and `}`. Similarly, you can include other expressions in your string literal, for example `${a + b}`. This new way of creating strings gives you more flexibility to create robust strings.

Interpolation

Template literals provide an easy way to interpolate variables and expressions into strings.

The method is called string interpolation.

The syntax is: ${variable}

Use template literal syntax with backticks to create an array of list element (`li`) strings. Each list element's text should be one of the array elements from the `failure` property on the `result` object and have a `class` attribute with the value `text-warning`. The `makeList` function should return the array of list item strings.

Use an iterator method (any kind of loop) to get the desired output (shown below).

```
[
  '<li class="text-warning">no-var</li>',
  '<li class="text-warning">var-on-top</li>',
  '<li class="text-warning">linebreak</li>'
]
```

Solution:

```
onst result = {
  success: ["max-length", "no-amd", "prefer-arrow-
functions"],
  failure: ["no-var", "var-on-top", "linebreak"],
  skipped: ["no-extra-semi", "no-dup-keys"]
};
function makeList(arr) {
  // Only change code below this line
  const failureItems = [];
  for (let i = 0; i < arr.length; i++) {
    failureItems.push(`<li class="text-
warning">${arr[i]}</li>`)
  }
  // Only change code above this line

  return failureItems;
}

const failuresList = makeList(result.failure);
```

Write Concise Object Literal Declarations Using Object Property Shorthand

ES6 adds some nice support for easily defining object literals.

Consider the following code:

```
const getMousePosition = (x, y) => ({
  x: x,
  y: y
});
```

`getMousePosition` is a simple function that returns an object containing two properties. ES6 provides the syntactic sugar to eliminate the redundancy of having to write `x: x`. You can simply write `x` once, and it will be converted to`x: x` (or something equivalent) under the hood. Here is the same function from above rewritten to use this new syntax:

```
const getMousePosition = (x, y) => ({ x, y });
```

Use object property shorthand with object literals to create and return an object with `name`, `age` and `gender` properties.

Solution:

```
const createPerson = (name, age, gender) => {
  // Only change code below this line
    return { name, age, gender };
  // Only change code above this line
};
```

Write Concise Declarative Functions with ES6

When defining functions within objects in ES5, we have to use the keyword `function` as follows:

A method is a function stored as a property.

```
const person = {
  name: "Taylor",
  sayHello: function() {
    return `Hello! My name is ${this.name}.`;
  }
};
```

With ES6, you can remove the `function` keyword and colon altogether when defining functions in objects. Here's an example of this syntax:

```
const person = {
  name: "Taylor",
  sayHello() {
    return `Hello! My name is ${this.name}.`;
  }
};
```

Refactor the function `setGear` inside the object `bicycle` to use the shorthand syntax described above.

```
// Only change code below this line
const bicycle = {
  gear: 2,
  setGear(newGear) {
    this.gear = newGear;
  }
};
// Only change code above this line
bicycle.setGear(3);
console.log(bicycle.gear);
```

Use class Syntax to Define a Constructor Function

JavaScript Class Syntax

Use the keyword `class` to create a class.

Always add a method named `constructor()`:

```
class ClassName {
  constructor() { ... }
}
```

Example

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
```

The example above creates a class named "Car".

The class has two initial properties: "name" and "year".

A JavaScript class is **not** an object.

It is a **template** for JavaScript objects.

Using a Class

When you have a class, you can use the class to create objects:

Example

```
let myCar1 = new Car("Ford", 2014);
let myCar2 = new Car("Audi", 2019);
```

Try it Yourself »

The example above uses the **Car class** to create two **Car objects**.

The constructor method is called automatically when a new object is created.

The Constructor Method

The constructor method is a special method:

- It has to have the exact name "constructor"
- It is executed automatically when a new object is created
- It is used to initialize object properties

If you do not define a constructor method, JavaScript will add an empty constructor method.

Class Methods

Class methods are created with the same syntax as object methods.

Use the keyword `class` to create a class.

Always add a `constructor()` method.

Then add any number of methods.

Syntax

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

Create a Class method named "age", that returns the Car age:

ES6 provides a new syntax to create objects, using the *class* keyword.

It should be noted that the `class` syntax is just syntax, and not a full-fledged class-based implementation of an object-oriented paradigm, unlike in languages such as Java, Python, Ruby, etc.

In ES5, an object can be created by defining a `constructor` function and using the `new` keyword to instantiate the object.

In ES6, a `class` declaration has a `constructor` method that is invoked with the `new` keyword. If the `constructor` method is not explicitly defined, then it is implicitly defined with no arguments.

```javascript
// Explicit constructor
class SpaceShuttle {
  constructor(targetPlanet) {
    this.targetPlanet = targetPlanet;
  }
  takeOff() {
    console.log("To " + this.targetPlanet + "!");
  }
}

// Implicit constructor
class Rocket {
  launch() {
    console.log("To the moon!");
  }
}

const zeus = new SpaceShuttle('Jupiter');
// prints To Jupiter! in console
zeus.takeOff();

const atlas = new Rocket();
// prints To the moon! in console
```

```
atlas.launch();
```

It should be noted that the `class` keyword declares a new function, to which a constructor is added. This constructor is invoked when `new` is called to create a new object.

Note: UpperCamelCase should be used by convention for ES6 class names, as in `SpaceShuttle` used above.

The `constructor` method is a special method for creating and initializing an object created with a class. You will learn more about it in the Object Oriented Programming section of the JavaScript Algorithms And Data Structures Certification.

Use the `class` keyword and write a `constructor` to create the `Vegetable` class.

The `Vegetable` class allows you to create a vegetable object with a property `name` that gets passed to the `constructor`.

```
// Only change code below this line
class Vegetable {
  constructor(name) {
    this.name = name;
  }
}
// Only change code above this line

const carrot = new Vegetable('carrot');
console.log(carrot.name); // Should display 'carrot'
```

Use getters and setters to Control Access to an Object

JavaScript Accessors (Getters and Setters)

JavaScript Getter (The get Keyword)

This example uses a `lang` property to `get` the value of the `language` property.

Example

```
// Create an object:
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {
    return this.language;
  }
};

// Display data from the object using a getter:
document.getElementById("demo").innerHTML = person.lang;
```

JavaScript Setter (The set Keyword)

This example uses a `lang` property to `set` the value of the `language` property.

Example

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "",
  set lang(lang) {
    this.language = lang;
  }
};

// Set an object property using a setter:
person.lang = "en";

// Display data from the object:
document.getElementById("demo").innerHTML = person.language;
```

Why Using Getters and Setters?

- It gives simpler syntax
- It allows equal syntax for properties and methods
- It can secure better data quality
- It is useful for doing things behind-the-scenes
- You can obtain values from an object and set the value of a property within an object.
- These are classically called *getters* and *setters*.
- Getter functions are meant to simply return (get) the value of an object's private variable to the user without the user directly accessing the private variable.
- Setter functions are meant to modify (set) the value of an object's private variable based on the value passed into the setter function. This change could involve calculations, or even overwriting the previous value completely.

```
class Book {
  constructor(author) {
    this._author = author;
  }
  // getter
  get writer() {
    return this._author;
  }
  // setter
  set writer(updatedAuthor) {
    this._author = updatedAuthor;
  }
}
const novel = new Book('anonymous');
console.log(novel.writer);
novel.writer = 'newAuthor';
```

- `console.log(novel.writer);`
- The console would display the strings `anonymous` and `newAuthor`.
- Notice the syntax used to invoke the getter and setter. They do not even look like functions. Getters and setters are important because they hide internal implementation details.
- **Note:** It is convention to precede the name of a private variable with an underscore (_). However, the practice itself does not make a variable private.

-

- Use the `class` keyword to create a `Thermostat` class. The `constructor` accepts a Fahrenheit temperature.
- In the class, create a `getter` to obtain the temperature in Celsius and a `setter` to set the temperature in Celsius.
- Remember that `C = 5/9 * (F - 32)` and `F = C * 9.0 / 5 + 32`, where `F` is the value of temperature in Fahrenheit, and `C` is the value of the same temperature in Celsius.
- **Note:** When you implement this, you will track the temperature inside the class in one scale, either Fahrenheit or Celsius.
- This is the power of a getter and a setter. You are creating an API for another user, who can get the correct result regardless of which one you track.
- In other words, you are abstracting implementation details from the user.

Solution:

```
// Only change code below this line
class Thermostat {
  constructor(fahrenheit) {
    this.fahrenheit = fahrenheit;
  }

  get temperature() {
    return (5 / 9) * (this.fahrenheit - 32);
  }

  set temperature(celsius) {
    this.fahrenheit = (celsius * 9.0) / 5 + 32;
  }
}
// Only change code above this line

const thermos = new Thermostat(76); // Setting in Fahrenh
eit scale
let temp = thermos.temperature; // 24.44 in Celsius
thermos.temperature = 26;
temp = thermos.temperature; // 26 in Celsius
```

Create a Module Script

Modules

JavaScript modules allow you to break up your code into separate files.

This makes it easier to maintain the code-base.

JavaScript modules rely on the `import` and `export` statements.

Export

You can export a function or variable from any file.

Let us create a file named `person.js`, and fill it with the things we want to export.

There are two types of exports: Named and Default.

Named Exports

You can create named exports two ways. In-line individually, or all at once at the bottom.

JavaScript started with a small role to play on an otherwise mostly HTML web. Today, it's huge, and some websites are built almost entirely with JavaScript. In order to make JavaScript more modular, clean, and maintainable; ES6 introduced a way to easily share code among JavaScript files. This involves exporting parts of a file for use in one or more other files, and importing the parts you need, where you need them. In order to take advantage of this functionality, you need to create a script in your HTML document with a `type` of `module`. Here's an example:

```
<script type="module" src="filename.js"></script>
```

A script that uses this `module` type can now use the `import` and `export` features you will learn about in the upcoming challenges.

Add a script to the HTML document of type `module` and give it the source file of `index.js`

```
Solution:
<html>
  <body>
    <!-- Only change code below this line -->
<script type="module" src="index.js"></script>
    <!-- Only change code above this line -->
  </body>
</html>
```

Use export to Share a Code Block

Imagine a file called `math_functions.js` that contains several functions related to mathematical operations. One of them is stored in a variable, `add`, that takes in two numbers and returns their sum. You want to use this function in several different JavaScript files. In order to share it with these other files, you first need to `export` it.

```
export const add = (x, y) => {
  return x + y;
}
```

The above is a common way to export a single function, but you can achieve the same thing like this:

```
const add = (x, y) => {
  return x + y;
}

export { add };
```

When you export a variable or function, you can import it in another file and use it without having to rewrite the code. You can export multiple things by repeating the first example for each thing you want to export, or by placing them all in the export statement of the second example, like this:

```
export { add, subtract };
```

There are two string-related functions in the editor. Export both of them using the method of your choice.

```
const uppercaseString = (string) => {
  return string.toUpperCase();
}

const lowercaseString = (string) => {
  return string.toLowerCase()
}

export { uppercaseString, lowercaseString};
```

Reuse JavaScript Code Using import

`import` allows you to choose which parts of a file or module to load. In the previous lesson, the examples exported `add` from the `math_functions.js` file. Here's how you can import it to use in another file:

```
import { add } from './math_functions.js';
```

Here, `import` will find `add` in `math_functions.js`, import just that function for you to use, and ignore the rest. The `./` tells the import to look for the `math_functions.js` file in the same folder as the current file. The relative file path (`./`) and file extension (`.js`) are required when using import in this way.

You can import more than one item from the file by adding them in the `import` statement like this:

```
import { add, subtract } from './math_functions.js';
```

Add the appropriate `import` statement that will allow the current file to use the `uppercaseString` and `lowercaseString` functions you exported in the previous lesson. These functions are in a file called `string_functions.js`, which is in the same directory as the current file.

Solution:

```
import { uppercaseString, lowercaseString } from './strin
g_functions.js'
// Only change code above this line

uppercaseString("hello");
```

```
lowercaseString("WORLD!");
```

Use * to Import Everything from a File

Suppose you have a file and you wish to import all of its contents into the current file. This can be done with the `import * as` syntax. Here's an example where the contents of a file named `math_functions.js` are imported into a file in the same directory:

```
import * as myMathModule from "./math_functions.js";
```

The above `import` statement will create an object called `myMathModule`. This is just a variable name, you can name it anything. The object will contain all of the exports from `math_functions.js` in it, so you can access the functions like you would any other object property. Here's how you can use the `add` and `subtract` functions that were imported:

```
myMathModule.add(2,3);
myMathModule.subtract(5,3);
```

The code in this file requires the contents of the file: `string_functions.js`, that is in the same directory as the current file. Use the `import * as` syntax to import everything from the file into an object called `stringFunctions`.

Solution:

```
import * as stringFunctions from "./string_functions.js";
// Only change code above this line

stringFunctions.uppercaseString("hello");
stringFunctions.lowercaseString("WORLD!");
```

Create an Export Fallback with export default

In the `export` lesson, you learned about the syntax referred to as a *named export*. This allowed you to make multiple functions and variables available for use in other files.

There is another `export` syntax you need to know, known as *export default*. Usually you will use this syntax if only one value is being exported from a file. It is also used to create a fallback value for a file or module.

Below are examples using `export default`:

```
export default function add(x, y) {
  return x + y;
}

export default function(x, y) {
  return x + y;
}
```

The first is a named function, and the second is an anonymous function.

Since `export default` is used to declare a fallback value for a module or file, you can only have one value be a default export in each module or file. Additionally, you cannot use `export default` with `var`, `let`, or `const`

The following function should be the fallback value for the module. Please add the necessary code to do so.

Solution:

```
export default function subtract(x, y) {
  return x - y;
}
```

Import a Default Export

In the last challenge, you learned about `export default` and its uses. To import a default export, you need to use a different `import` syntax. In the following example, `add` is the default export of the `math_functions.js` file. Here is how to import it:

```
import add from "./math_functions.js";
```

The syntax differs in one key place. The imported value, `add`, is not surrounded by curly braces (`{}`). `add` here is simply a variable name for whatever the default export of the `math_functions.js` file is. You can use any name here when importing a default.

In the following code, import the default export from the `math_functions.js` file, found in the same directory as this file. Give the import the name `subtract`.

Solution:

```
import subtract from "./math_functions.js";
// Only change code above this line

subtract(7,4);
```

Create a JavaScript Promise

A promise in JavaScript is exactly what it sounds like - you use it to make a promise to do something, usually asynchronously. When the task completes, you either fulfill your promise or fail to do so. `Promise` is a constructor function, so you need to use the `new` keyword to create one. It takes a function, as its argument, with two parameters - `resolve` and `reject`. These are methods used to determine the outcome of the promise. The syntax looks like this:

```
const myPromise = new Promise((resolve, reject) => {
});
```

Create a new promise called `makeServerRequest`. Pass in a function with `resolve` and `reject` parameters to the constructor.

Solution:

```
const makeServerRequest = new Promise((resolve, reject) =
> {
});
```

When the producing code obtains the result, it should call one of the two callbacks:

Result	**Call**
Success	myResolve(result value)
Error	myReject(error object)

Promise Object Properties

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

The Promise object supports two properties: **state** and **result**.

While a Promise object is "pending" (working), the result is undefined.

When a Promise object is "fulfilled", the result is a value.

When a Promise object is "rejected", the result is an error object.

myPromise.state	**myPromise.result**
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

You cannot access the Promise properties **state** and **result**.

You must use a Promise method to handle promises.

Complete a Promise with resolve and reject

A promise has three states: `pending`, `fulfilled`, and `rejected`. The promise you created in the last challenge is forever stuck in the `pending` state because you did not add a way to complete the promise. The `resolve` and `reject` parameters given to the promise argument are used to do this. `resolve` is used when you want your promise to succeed, and `reject` is used when you want it to fail. These are methods that take an argument, as seen below.

```
const myPromise = new Promise((resolve, reject) => {
  if(condition here) {
    resolve("Promise was fulfilled");
  } else {
    reject("Promise was rejected");
  }
});
```

The example above uses strings for the argument of these functions, but it can really be anything. Often, it might be an object, that you would use data from, to put on your website or elsewhere.

Make the promise handle success and failure. If `responseFromServer` is `true`, call the `resolve` method to successfully complete the promise. Pass `resolve` a string with the value `We got the data`. If `responseFromServer` is `false`, use the `reject` method instead and pass it the string: `Data not received`.

Solution:

```
const makeServerRequest = new Promise((resolve, reject) =
> {
  // responseFromServer represents a response from a serv
er
  let responseFromServer;

  if(responseFromServer) {
    resolve("We got the data");
  } else {
    reject("Data not received");
  }
});
```

Handle a Fulfilled Promise with then

Promises are most useful when you have a process that takes an unknown amount of time in your code (i.e. something asynchronous), often a server request. When you make a server request it takes some amount of time, and after it completes you usually want to do something with the response from the server. This can be achieved by using the `then` method. The `then` method is executed immediately after your promise is fulfilled with `resolve`. Here's an example:

```
myPromise.then(result => {


});
```

`result` comes from the argument given to the `resolve` method.

Add the `then` method to your promise. Use `result` as the parameter of its callback function and log `result` to the console.

Solution:

```
const makeServerRequest = new Promise((resolve, reject) =
> {
  // responseFromServer is set to true to represent a suc
cessful response from a server
  let responseFromServer = true;

  if(responseFromServer) {
    resolve("We got the data");
  } else {
    reject("Data not received");
  }
});

makeServerRequest.then(result => {
  console.log(result);
})
```

Handle a Rejected Promise with catch

`catch` is the method used when your promise has been rejected. It is executed immediately after a promise's `reject` method is called. Here's the syntax:

```
myPromise.catch(error => {


});
```

`error` is the argument passed in to the `reject` method.

Add the `catch` method to your promise. Use `error` as the parameter of its callback function and log `error` to the console.

Solution:

```
const makeServerRequest = new Promise((resolve, reject) =
> {
  // responseFromServer is set to false to represent an u
nsuccessful response from a server
  let responseFromServer = false;

  if(responseFromServer) {
    resolve("We got the data");
  } else {
    reject("Data not received");
  }
});

makeServerRequest.then(result => {
  console.log(result);
});

makeServerRequest.catch(error => {
  console.log(error);
})
```