**ES6: Explore Differences Between the var and let KeywordsPassed**

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

Update the code so it only uses the let keyword.

**ES6: Compare Scopes of the var and let KeywordsPassed**

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:

'use strict';

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.

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.

**ES6: Declare a Read-Only Variable with the const KeywordPassed**

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.

"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.

**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.

Change the code so that all variables are declared using let or const. Use let when you want the variable to change, and const when you want the variable to remain constant. Also, rename variables declared with const to conform to common practices, meaning constants should be in all caps.

**ES6: Mutate an Array Declared with constPassed**

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.

"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.

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

**ES6: Prevent Object MutationPassed**

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"}

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.

**ES6: Use Arrow Functions to Write Concise Anonymous FunctionsPassed**

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 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.

**ES6: Write Arrow Functions with ParametersPassed**

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

// doubles input value and returns it

const doubler = (item) => item \* 2;

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

// the same function, without the argument 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;

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

**ES6: Set Default Parameters for Your FunctionsPassed**

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.

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

**ES6: Use the Rest Parameter with Function ParametersPassed**

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.

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.

**ES6: Use the Spread Operator to Evaluate Arrays In-PlacePassed**

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

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

**ES6: Use Destructuring Assignment to Extract Values from ObjectsPassed**

*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; // name = 'John Doe'

const age = user.age; // age = 34

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

const { name, age } = user;

// name = 'John Doe', age = 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

**ES6: Use Destructuring Assignment to Assign Variables from ObjectsPassed**

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;

// userName = 'John Doe', userAge = 34

You may read it as "get the value of user.name and assign it to a new variable named userName" and so on.

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.

Run the TestsReset All Code

Get Help

**ES6: Use Destructuring Assignment to Assign Variables from Nested ObjectsPassed**

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.

**ES6: Use Destructuring Assignment to Assign Variables from ArraysPassed**

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); // 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); // 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.

**ES6: Use Destructuring Assignment with the Rest Parameter to Reassign Array ElementsPassed**

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); // 1, 2

console.log(arr); // [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 destructuring assignment with the rest parameter to perform an effective Array.prototype.slice() so that arr is a sub-array of the original array source with the first two elements omitted.

**ES6: Use Destructuring Assignment to Pass an Object as a Function's ParametersPassed**

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;

// do something with these variables

}

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

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

/\* do something with these fields \*/

}

This removes some extra lines and makes our code look neat. This has the added benefit of not having to manipulate an entire object in a function — only the fields that are needed are copied inside the function.

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

**ES6: Create Strings using Template LiteralsPassed**

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

};

// Template literal with multi-line and string interpolation

const greeting = `Hello, my name is ${person.name}!

I am ${person.age} years old.`;

console.log(greeting); // prints

// Hello, my name is Zodiac Hasbro!

// 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.

Use template literal syntax with backticks to display each entry of the result object's failure array. Each entry should be wrapped inside an li element with the class attribute text-warning, and listed within the resultDisplayArray.

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>'

]

**ES6: Write Concise Object Literal Declarations Using Object Property ShorthandPassed**

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 tox: 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.

**ES6: Write Concise Declarative Functions with ES6Passed**

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

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.

## ES6: Use class Syntax to Define a Constructor FunctionPassed

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, we usually define a constructor function and use the new keyword to instantiate an object.

var SpaceShuttle = function(targetPlanet){

this.targetPlanet = targetPlanet;

}

var zeus = new SpaceShuttle('Jupiter');

The class syntax simply replaces the constructor function creation:

class SpaceShuttle {

constructor(targetPlanet) {

this.targetPlanet = targetPlanet;

}

}

const zeus = new SpaceShuttle('Jupiter');

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.  
**Notes:**

* 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.

## ES6: Use getters and setters to Control Access to an ObjectPassed

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 lol = new Book('anonymous');

console.log(lol.writer); // anonymous

lol.writer = 'wut';

console.log(lol.writer); // wut

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.

Now create a getter and a setter in the class, to obtain 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.

## ES6: Create a Module ScriptPassed

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

## ES6: Use export to Share a Code BlockPassed

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.

## ES6: Reuse JavaScript Code Using importPassed

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.

## ES6: Use \* to Import Everything from a FilePassed

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.

## ES6: Create an Export Fallback with export defaultPassed

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:

// named function

export default function add(x, y) {

return x + y;

}

// anonymous function

export default function(x, y) {

return x + y;

}

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.

Run the TestsReset All Code

Get Help

## ES6: Import a Default ExportPassed

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.

## ES6: Create a JavaScript PromisePassed

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.

## ES6: Complete a Promise with resolve and rejectPassed

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.

## ES6: Handle a Fulfilled Promise with thenPassed

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 => {

// do something with the 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.

## ES6: Handle a Rejected Promise with catchPassed

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 => {

// do something with the error.

});

error is the argument passed in to the reject method.

**Note:** the then and catch methods can be chained to the promise declaration if you choose.

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