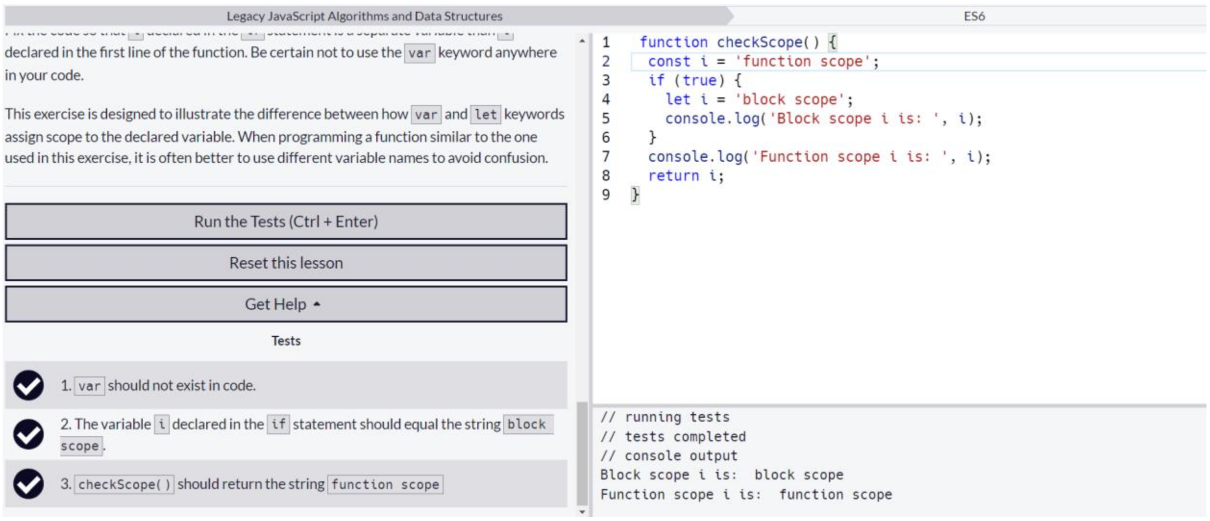


declared in the first line of the function. Be certain not to use the `var` keyword anywhere in your code.

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.

Tests

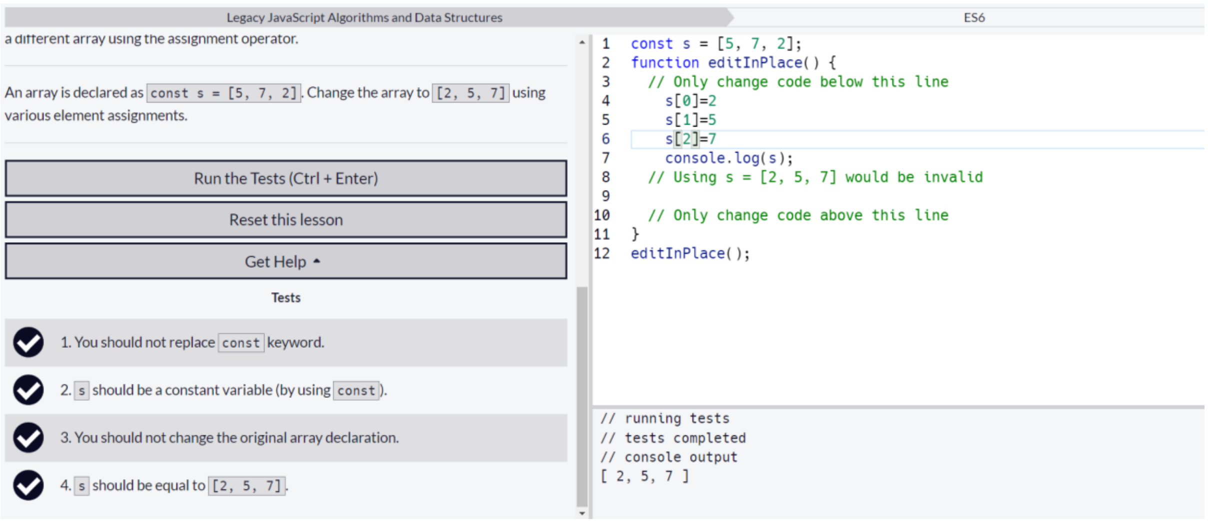
✓ 1. `var` should not exist in code.

✓ 2. The variable `i` declared in the `if` statement should equal the string `block scope`.

✓ 3. `checkScope()` should return the string `function scope`

```
function checkScope() {
  const 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;
}
```

```
// running tests
// tests completed
// console output
Block scope i is:  block scope
Function scope i is:  function scope
```


Legacy JavaScript Algorithms and Data Structures
ES6

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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help ▲

Tests

✓ 1. You should not replace `const` keyword.

✓ 2. `s` should be a constant variable (by using `const`).

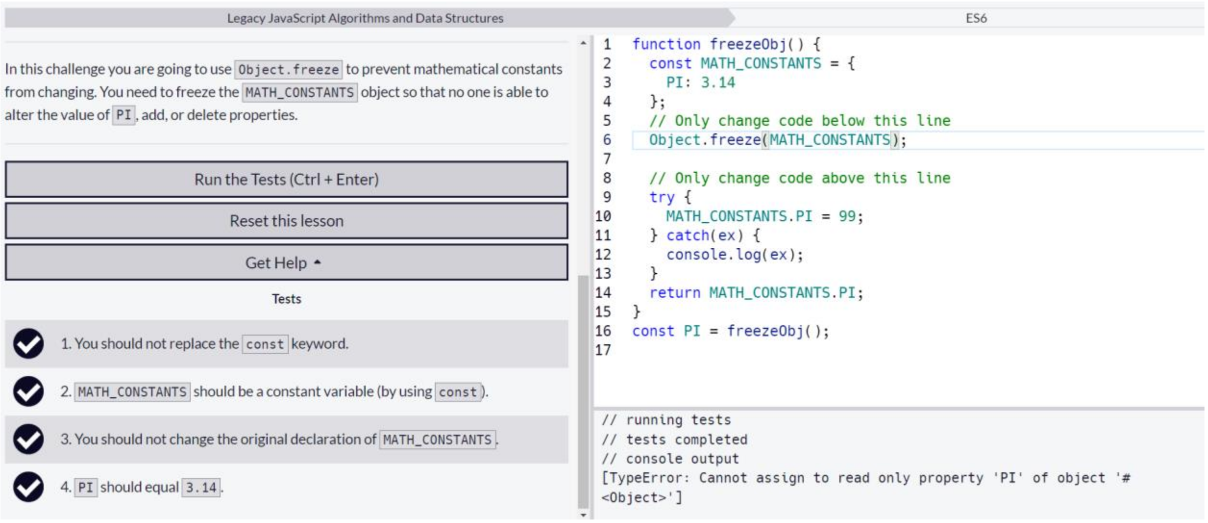
✓ 3. You should not change the original array declaration.

✓ 4. `s` should be equal to `[2, 5, 7]`.

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

  // Only change code above this line
}
editInPlace();
```

```
// running tests
// tests completed
// console output
[ 2, 5, 7 ]
```


Legacy JavaScript Algorithms and Data Structures
ES6

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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help ▲

Tests

✓ 1. You should not replace the `const` keyword.

✓ 2. `MATH_CONSTANTS` should be a constant variable (by using `const`).

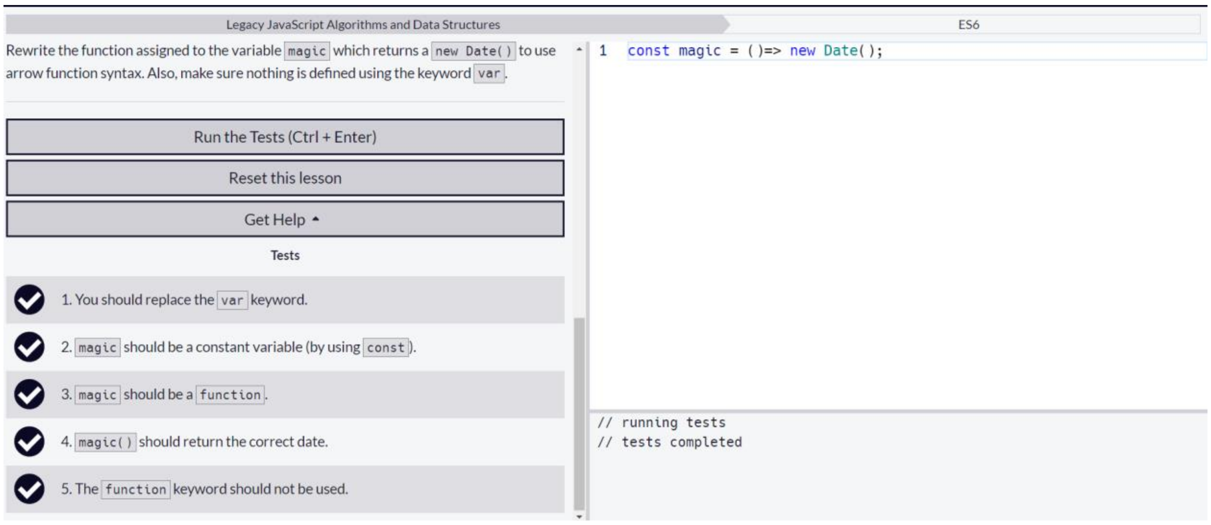
✓ 3. You should not change the original declaration of `MATH_CONSTANTS`.

✓ 4. `PI` should equal `3.14`.

```
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();
```

```
// running tests
// tests completed
// console output
[TypeError: Cannot assign to read only property 'PI' of object '#
<Object>']
```


Legacy JavaScript Algorithms and Data Structures
ES6

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`.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. You should replace the `var` keyword.

✓ 2. `magic` should be a constant variable (by using `const`).

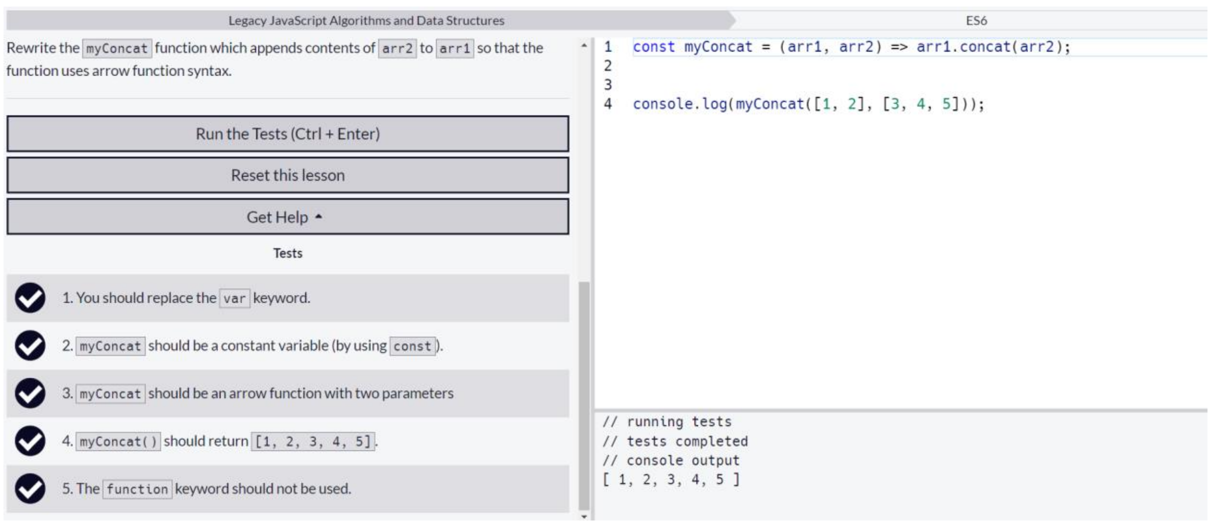
✓ 3. `magic` should be a `function`.

✓ 4. `magic()` should return the correct date.

✓ 5. The `function` keyword should not be used.

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

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures
ES6

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

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. You should replace the `var` keyword.

✓ 2. `myConcat` should be a constant variable (by using `const`).

✓ 3. `myConcat` should be an arrow function with two parameters

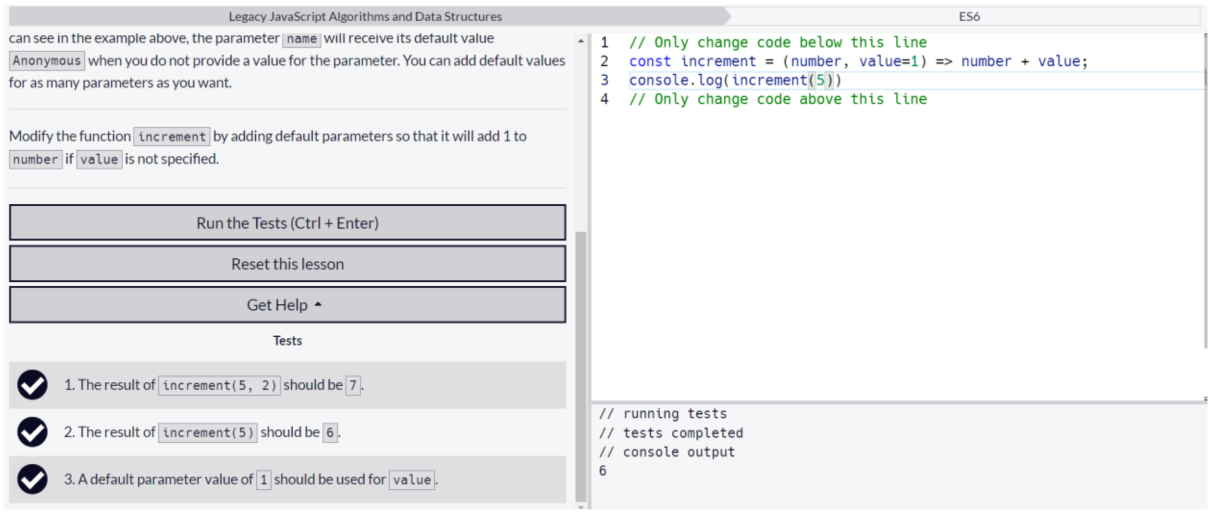
✓ 4. `myConcat()` should return `[1, 2, 3, 4, 5]`.

✓ 5. The `function` keyword should not be used.

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


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

```
// running tests
// tests completed
// console output
[ 1, 2, 3, 4, 5 ]
```


Legacy JavaScript Algorithms and Data Structures
ES6

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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

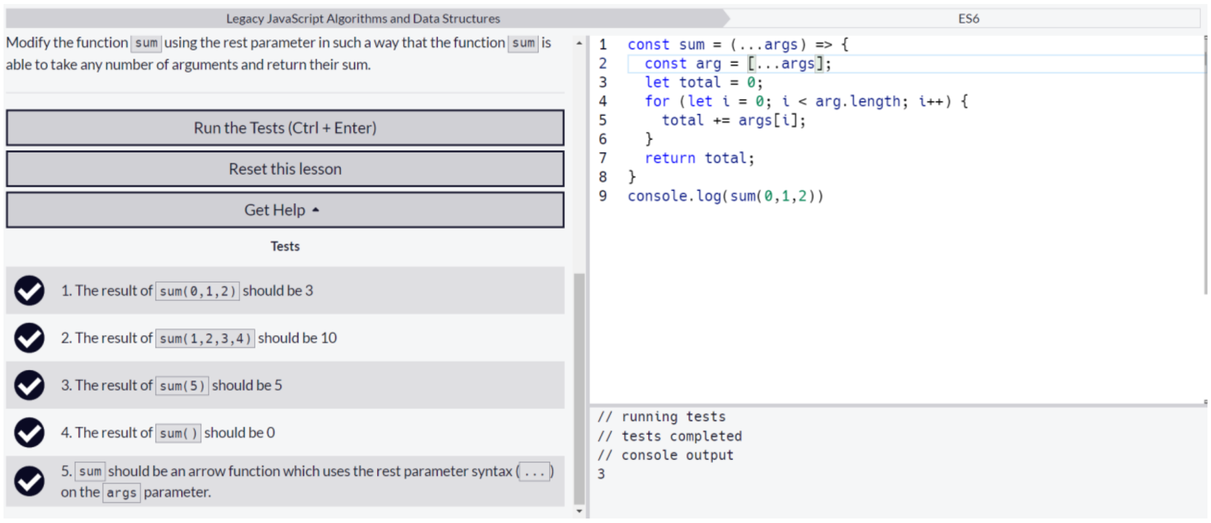
✓ 1. The result of `increment(5, 2)` should be `7`.

✓ 2. The result of `increment(5)` should be `6`.

✓ 3. A default parameter value of `1` should be used for `value`.

```
// Only change code below this line
const increment = (number, value=1) => number + value;
console.log(increment(5))
// Only change code above this line
```

```
// running tests
// tests completed
// console output
6
```


Legacy JavaScript Algorithms and Data Structures
ES6

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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help

Tests

✓ 1. The result of `sum(0,1,2)` should be 3

✓ 2. The result of `sum(1,2,3,4)` should be 10

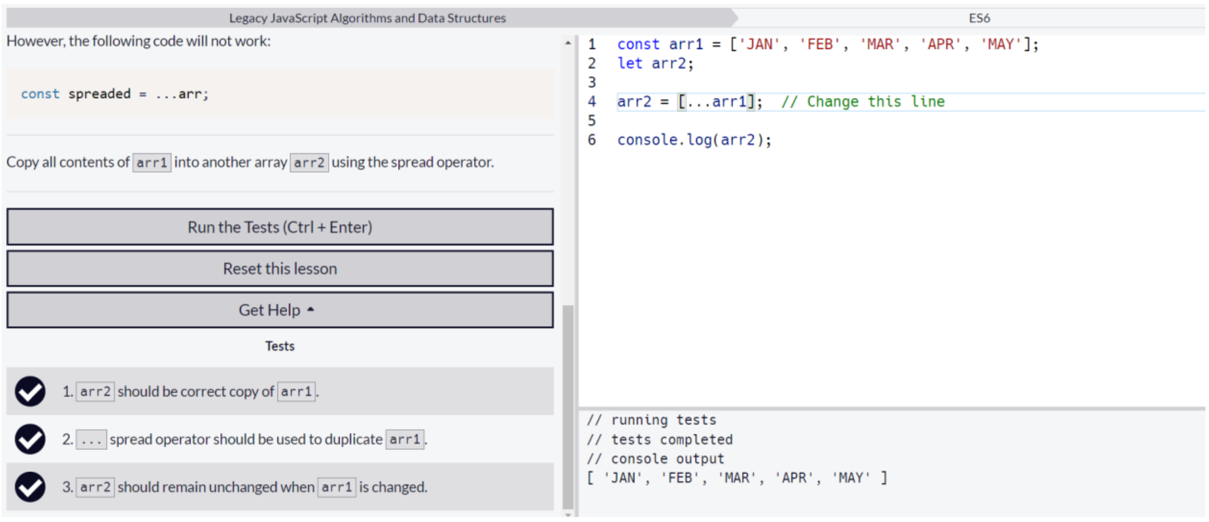
✓ 3. The result of `sum(5)` should be 5

✓ 4. The result of `sum()` should be 0

✓ 5. `sum` should be an arrow function which uses the rest parameter syntax (`...`) on the `args` parameter.

```
const sum = (...args) => {
  const arg = [...args];
  let total = 0;
  for (let i = 0; i < arg.length; i++) {
    total += args[i];
  }
  return total;
}
console.log(sum(0,1,2))
```

```
// running tests
// tests completed
// console output
3
```


Legacy JavaScript Algorithms and Data Structures
ES6

However, the following code will not work:

```
const spreaded = ...arr;
```

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

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help

Tests

✓ 1. `arr2` should be correct copy of `arr1`.

✓ 2. `...` spread operator should be used to duplicate `arr1`.

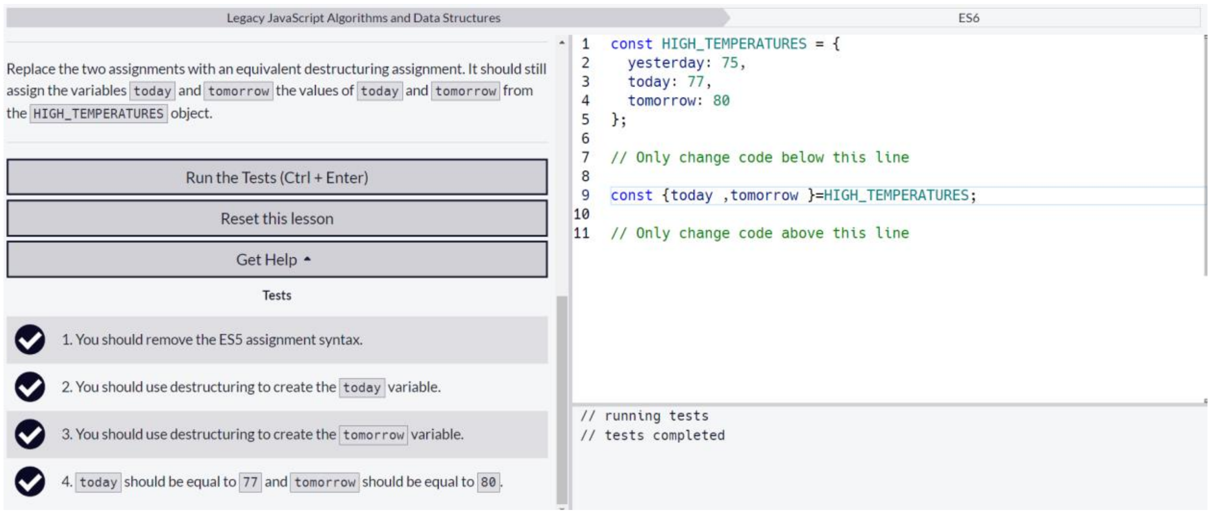
✓ 3. `arr2` should remain unchanged when `arr1` is changed.

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

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

console.log(arr2);
```

```
// running tests
// tests completed
// console output
[ 'JAN', 'FEB', 'MAR', 'APR', 'MAY' ]
```


Legacy JavaScript Algorithms and Data Structures
ES6

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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help

Tests

✓ 1. You should remove the ES5 assignment syntax.

✓ 2. You should use destructuring to create the `today` variable.

✓ 3. You should use destructuring to create the `tomorrow` variable.

✓ 4. `today` should be equal to `77` and `tomorrow` should be equal to `80`.

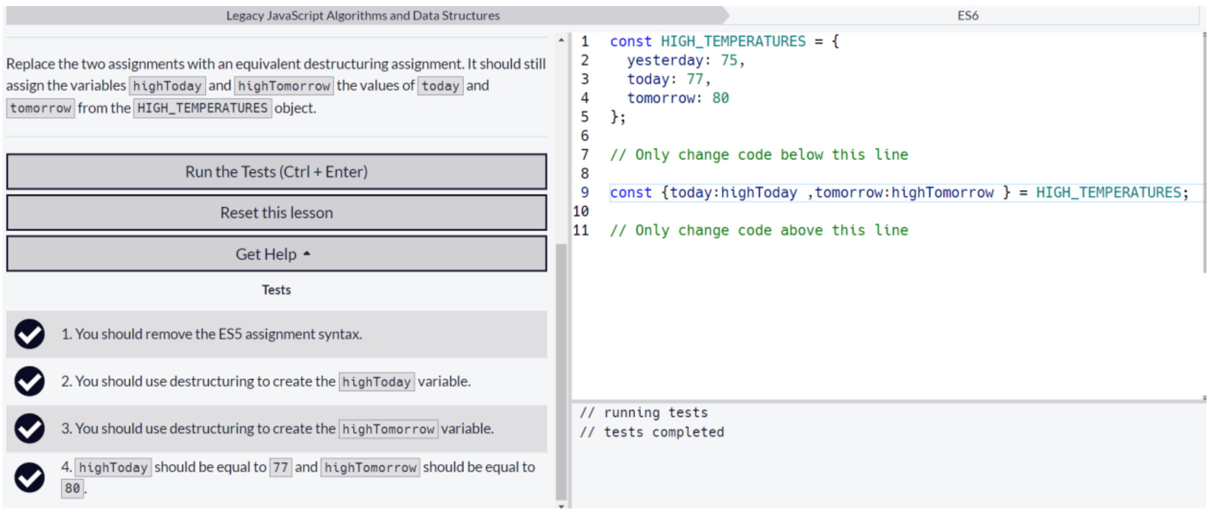
```
const HIGH_TEMPERATURES = {
  yesterday: 75,
  today: 77,
  tomorrow: 80
};

// Only change code below this line

const {today ,tomorrow }=HIGH_TEMPERATURES;

// Only change code above this line
```

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures — ES6

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 Tests (Ctrl + Enter)

Reset this lesson

Get Help

Tests

✓ 1. You should remove the ES5 assignment syntax.

✓ 2. You should use destructuring to create the `highToday` variable.

✓ 3. You should use destructuring to create the `highTomorrow` variable.

✓ 4. `highToday` should be equal to `77` and `highTomorrow` should be equal to `80`.

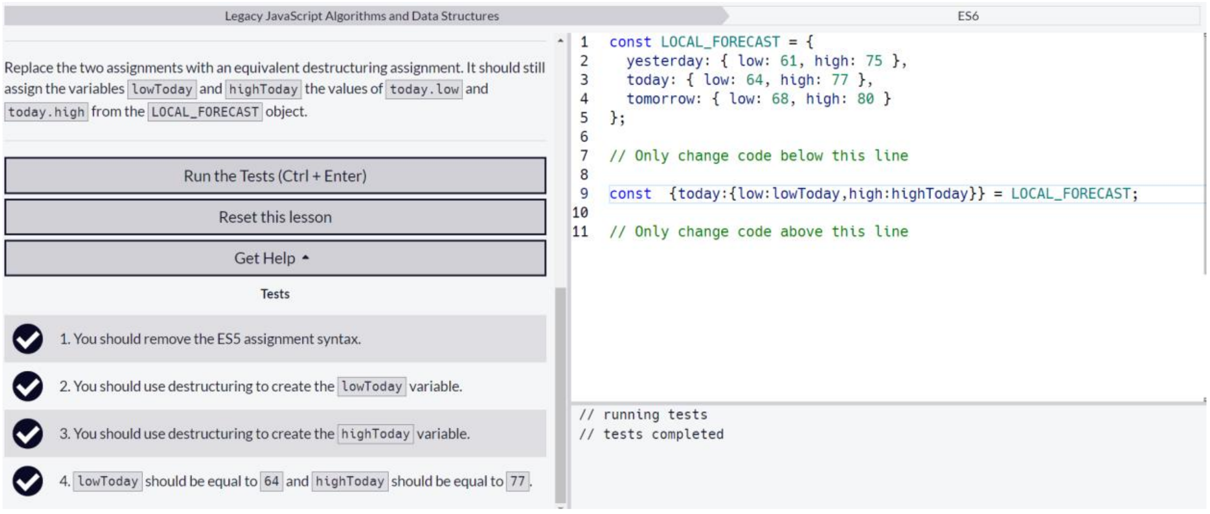
```
const HIGH_TEMPERATURES = {
  yesterday: 75,
  today: 77,
  tomorrow: 80
};

// Only change code below this line

const {today:highToday ,tomorrow:highTomorrow } = HIGH_TEMPERATURES;

// Only change code above this line
```

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures — ES6

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.

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help

Tests

✓ 1. You should remove the ES5 assignment syntax.

✓ 2. You should use destructuring to create the `lowToday` variable.

✓ 3. You should use destructuring to create the `highToday` variable.

✓ 4. `lowToday` should be equal to `64` and `highToday` should be equal to `77`.

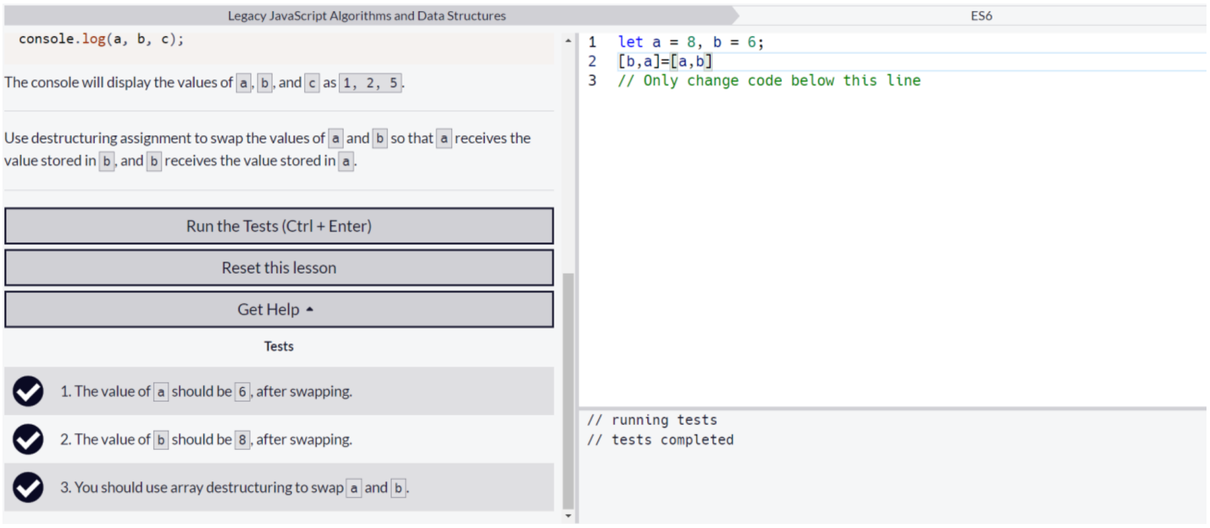
```
const LOCAL_FORECAST = {
  yesterday: { low: 61, high: 75 },
  today: { low: 64, high: 77 },
  tomorrow: { low: 68, high: 80 }
};

// Only change code below this line

const  {today:{low:lowToday,high:highToday}} = LOCAL_FORECAST;

// Only change code above this line
```

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures — ES6

```
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`.

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help

Tests

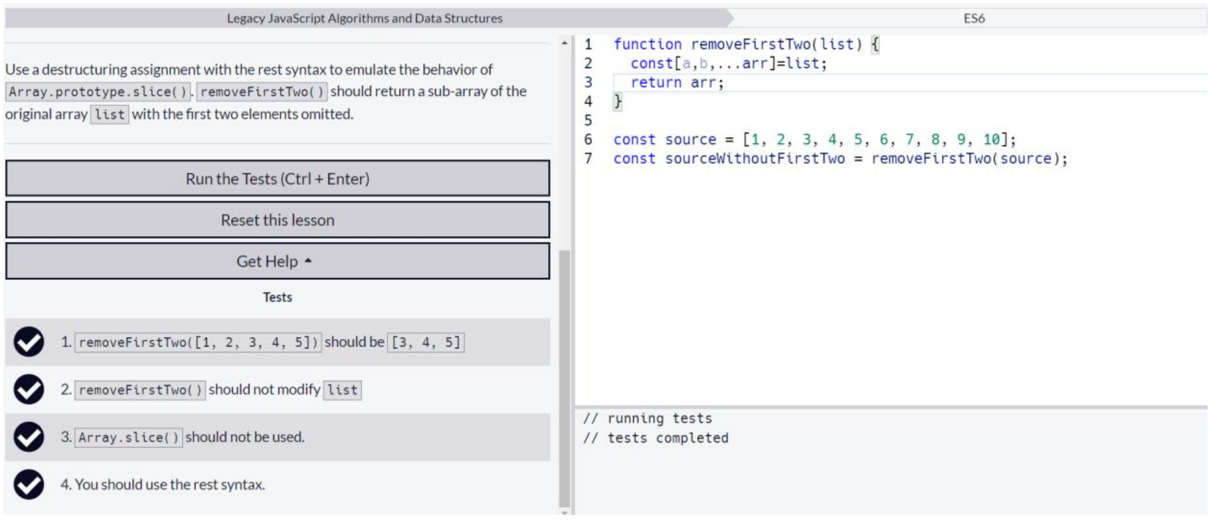
✓ 1. The value of `a` should be `6`, after swapping.

✓ 2. The value of `b` should be `8`, after swapping.

✓ 3. You should use array destructuring to swap `a` and `b`.

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

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures
ES6

Use a destructuring assignment with the rest syntax 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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. `removeFirstTwo([1, 2, 3, 4, 5])` should be `[3, 4, 5]`

✓ 2. `removeFirstTwo()` should not modify `list`

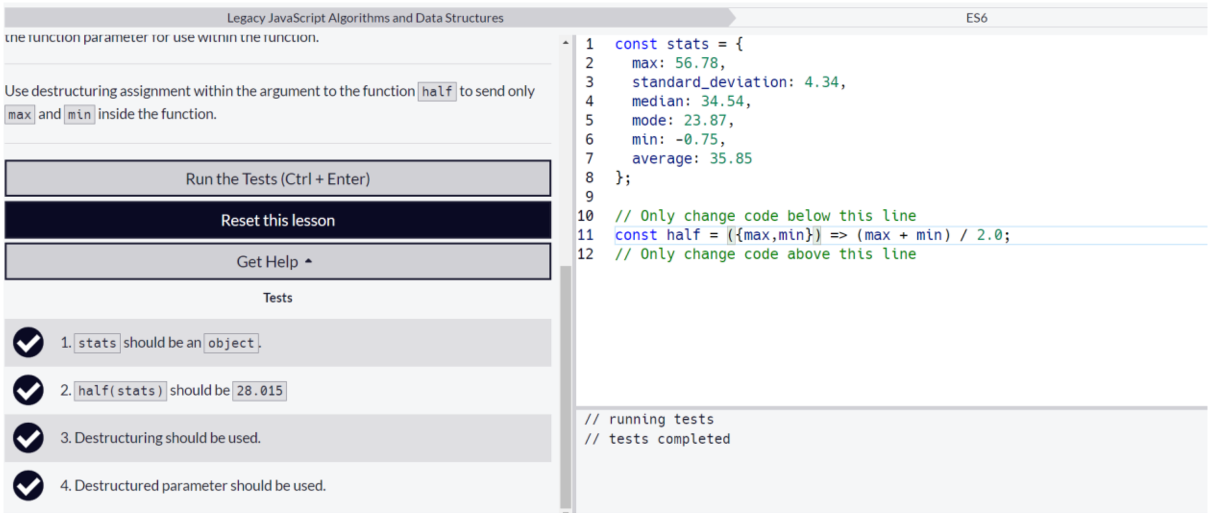
✓ 3. `Array.slice()` should not be used.

✓ 4. You should use the rest syntax.

```
function removeFirstTwo(list) {
  const[a,b,...arr]=list;
  return arr;
}

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

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures
ES6

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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. `stats` should be an `object`.

✓ 2. `half(stats)` should be `28.015`

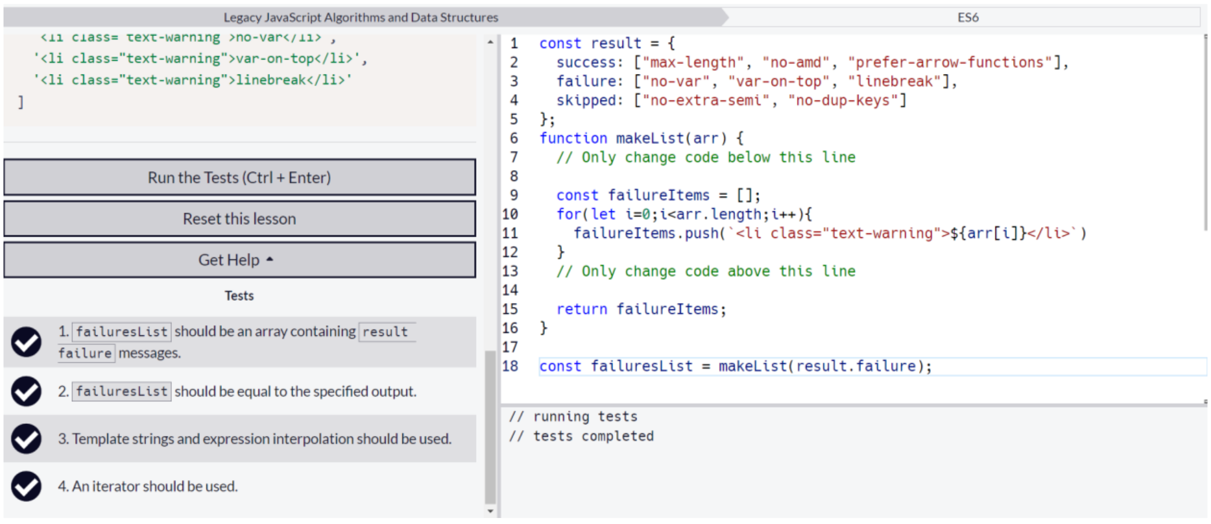
✓ 3. Destructuring should be used.

✓ 4. Destructured parameter should be used.

```
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
```

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures
ES6

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

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. `failuresList` should be an array containing `result failure` messages.

✓ 2. `failuresList` should be equal to the specified output.

✓ 3. Template strings and expression interpolation should be used.

✓ 4. An iterator should be used.

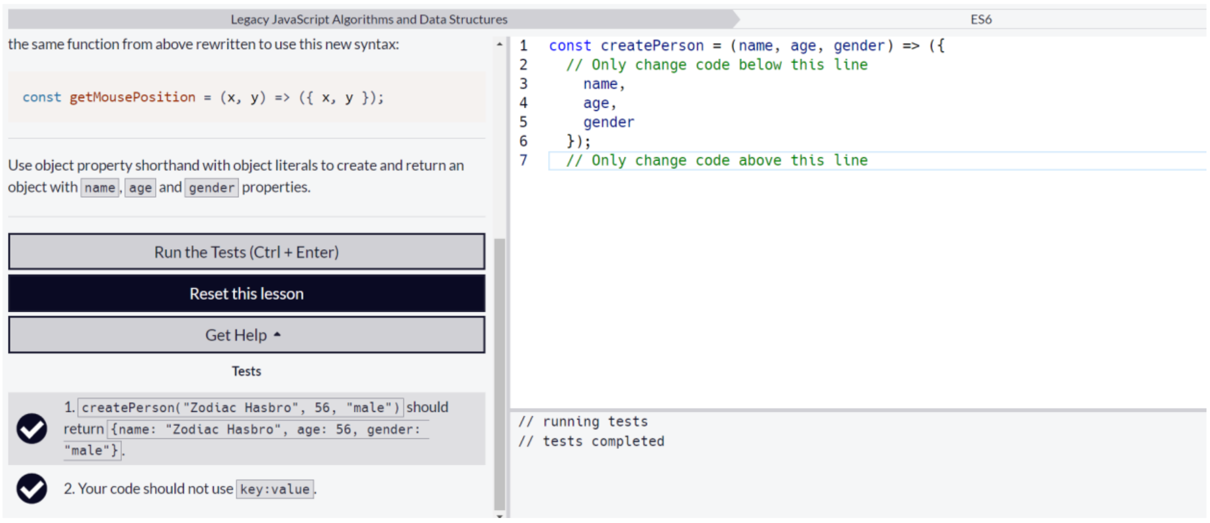
```
const result = {
  success: ["max-length", "no-amd", "prefer-arrow-functions"],
  failure: ["no-var", "var-on-top", "linebreak"],
  skipped: ["no-extra-semi", "no-dup-keys"]
};
function makeList(arr) {
  // Only change code below this line

  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);
```

```
// running tests
// tests completed
```



Legacy JavaScript Algorithms and Data Structures
ES6
Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help

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.

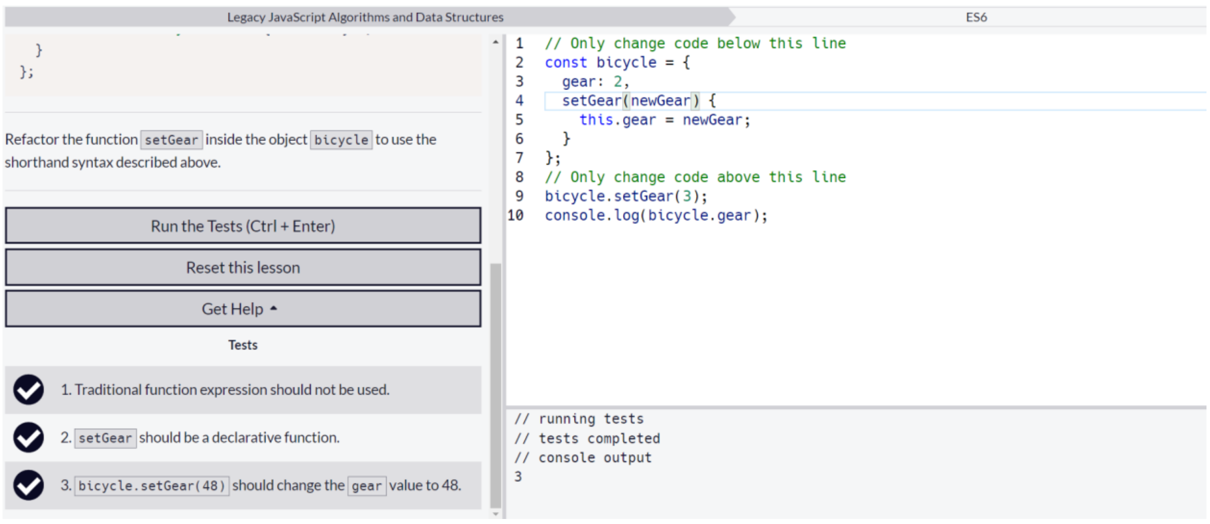
Tests

✓ 1. `createPerson("Zodiac Hasbro", 56, "male")` should return `{name: "Zodiac Hasbro", age: 56, gender: "male"}`.

✓ 2. Your code should not use `key:value`.

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

```
// running tests
// tests completed
```



Legacy JavaScript Algorithms and Data Structures
ES6
Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help

```
  }
};
```

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

Tests

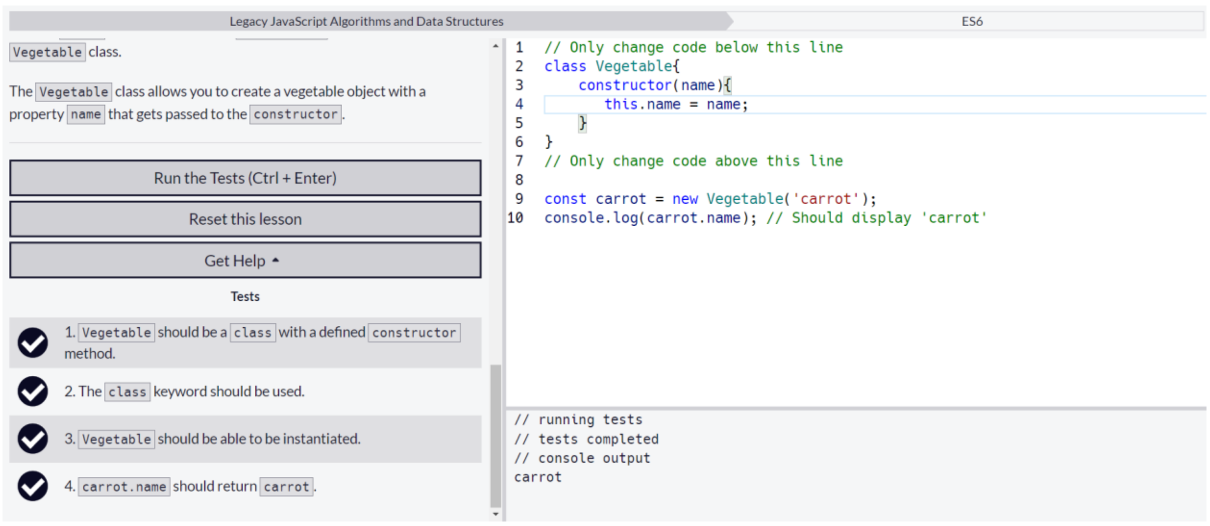
✓ 1. Traditional function expression should not be used.

✓ 2. `setGear` should be a declarative function.

✓ 3. `bicycle.setGear(48)` should change the `gear` value to 48.

```
// 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);
```

```
// running tests
// tests completed
// console output
3
```



Legacy JavaScript Algorithms and Data Structures
ES6
Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help

`Vegetable` class.

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

Tests

✓ 1. `Vegetable` should be a `class` with a defined `constructor` method.

✓ 2. The `class` keyword should be used.

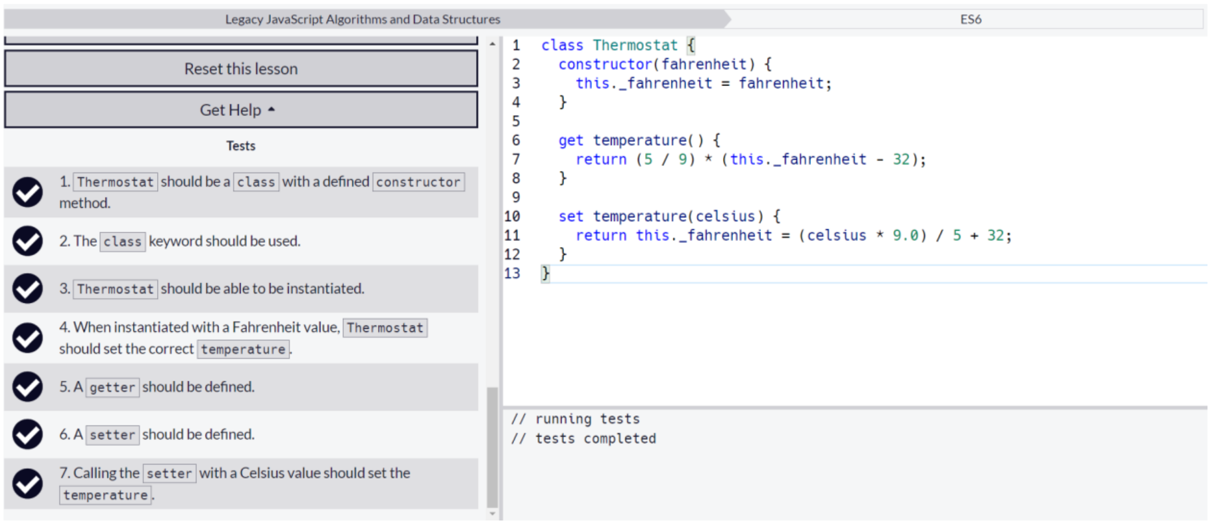
✓ 3. `Vegetable` should be able to be instantiated.

✓ 4. `carrot.name` should return `carrot`.

```
// 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'
```

```
// running tests
// tests completed
// console output
carrot
```


Legacy JavaScript Algorithms and Data Structures | ES6

Reset this lesson

Get Help ▲

Tests

✓ 1. `Thermostat` should be a `class` with a defined `constructor` method.

✓ 2. The `class` keyword should be used.

✓ 3. `Thermostat` should be able to be instantiated.

✓ 4. When instantiated with a Fahrenheit value, `Thermostat` should set the correct `temperature`.

✓ 5. A `getter` should be defined.

✓ 6. A `setter` should be defined.

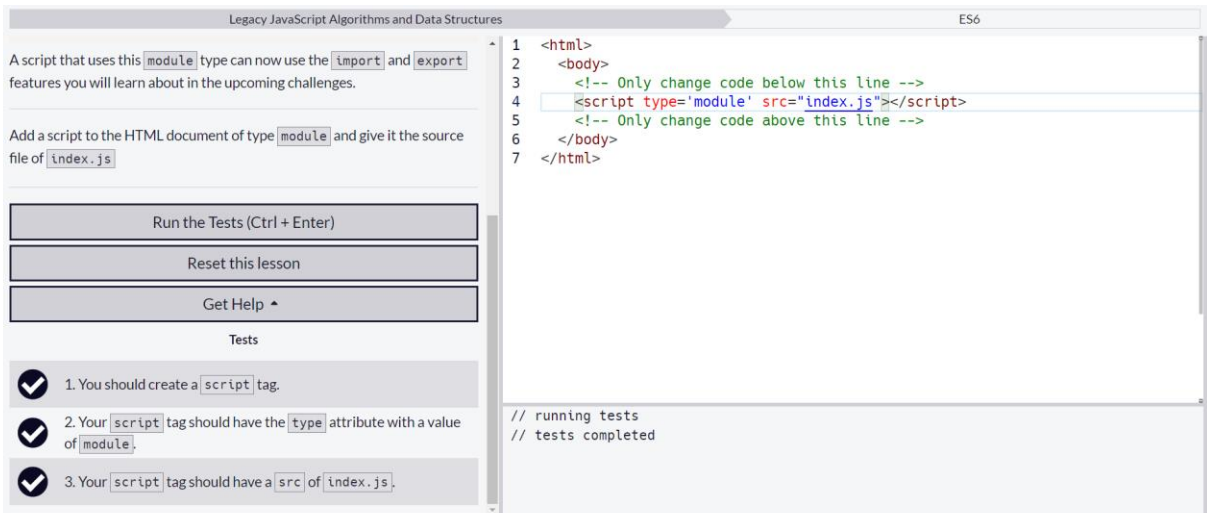
✓ 7. Calling the `setter` with a Celsius value should set the `temperature`.

```
class Thermostat {
  constructor(fahrenheit) {
    this._fahrenheit = fahrenheit;
  }

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

  set temperature(celsius) {
    return this._fahrenheit = (celsius * 9.0) / 5 + 32;
  }
}
```

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures | ES6

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`

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help ▲

Tests

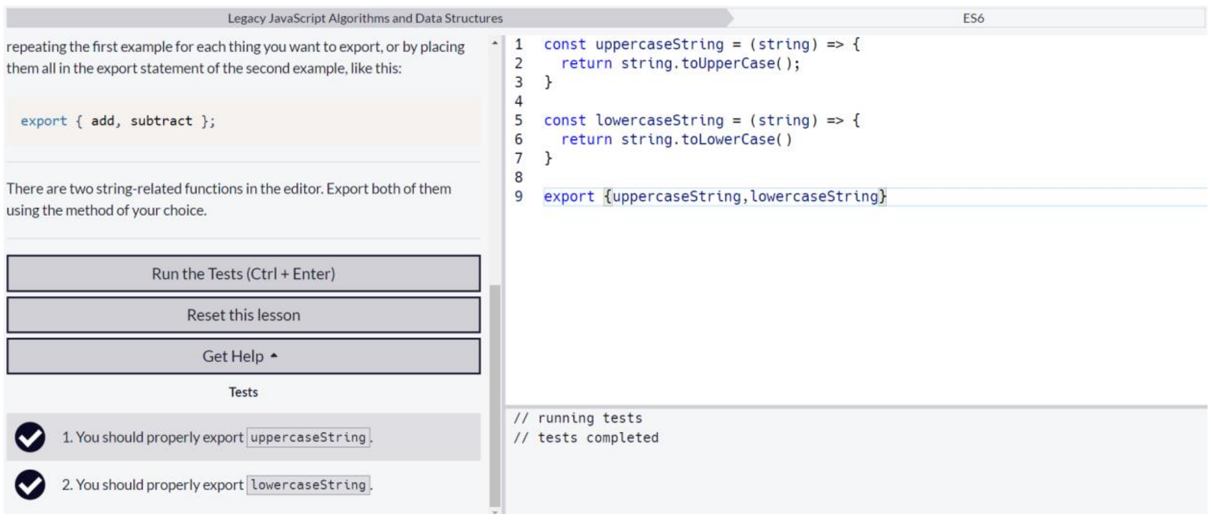
✓ 1. You should create a `script` tag.

✓ 2. Your `script` tag should have the `type` attribute with a value of `module`.

✓ 3. Your `script` tag should have a `src` of `index.js`.

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

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures | ES6

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.

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help ▲

Tests

✓ 1. You should properly export `uppercaseString`.

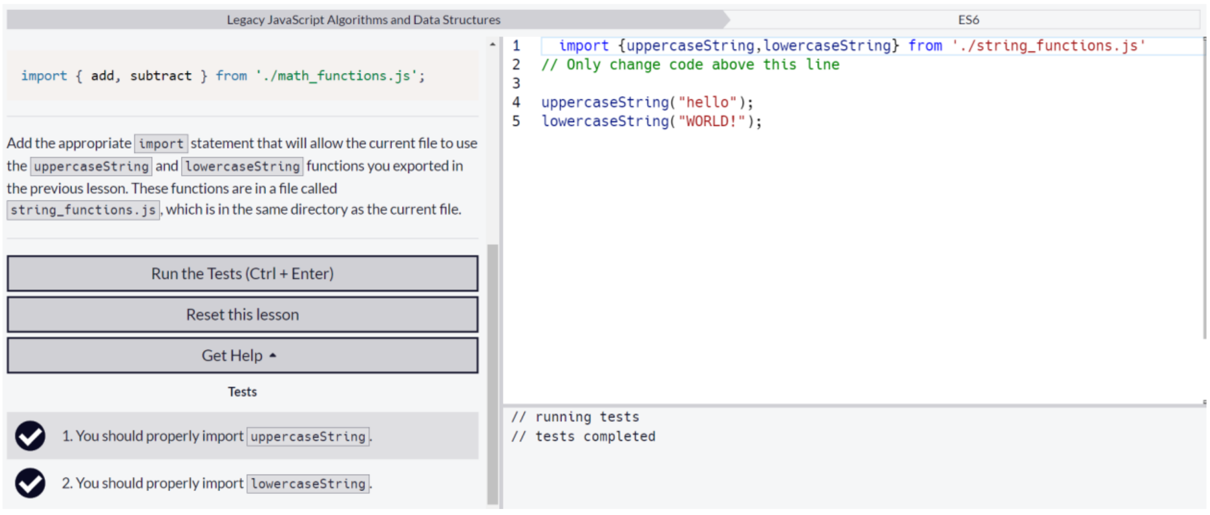
✓ 2. You should properly export `lowercaseString`.

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

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

export {uppercaseString,lowercaseString}
```

```
// running tests
// tests completed
```



Legacy JavaScript Algorithms and Data Structures | ES6

```
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.

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help ▴

Tests

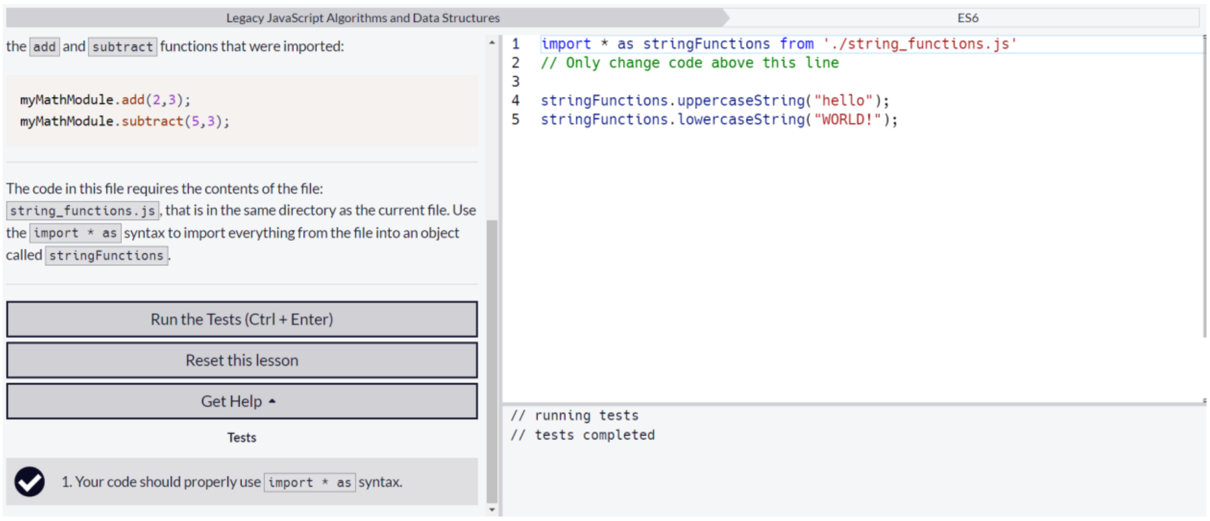
✓ 1. You should properly import `uppercaseString`.

✓ 2. You should properly import `lowercaseString`.

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

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

```
// running tests
// tests completed
```



Legacy JavaScript Algorithms and Data Structures | ES6

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`.

Run the Tests (Ctrl + Enter)

Reset this lesson

Get Help ▴

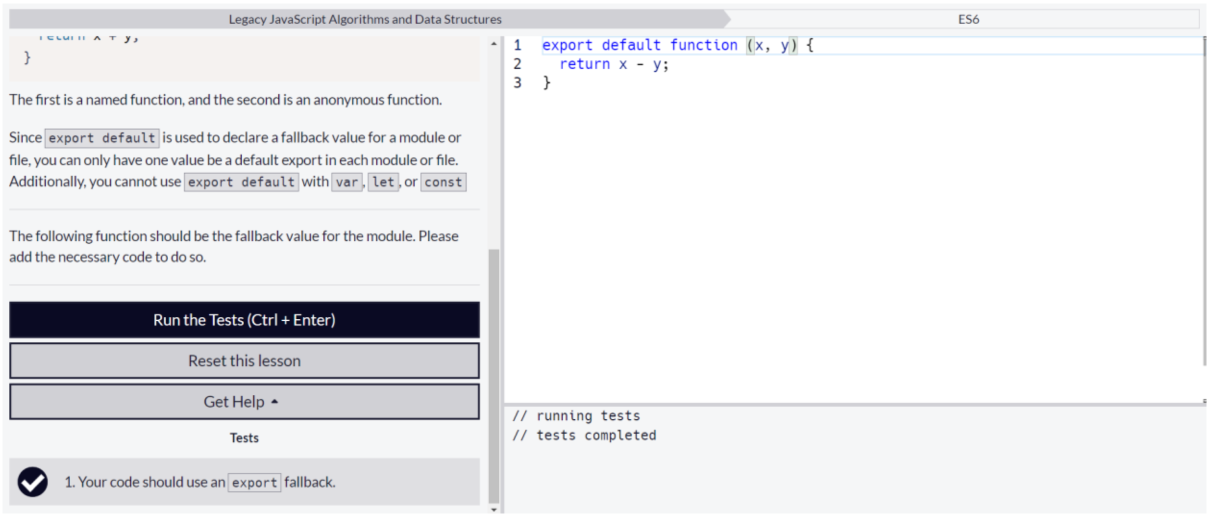
Tests

✓ 1. Your code should properly use `import * as` syntax.

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

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

```
// running tests
// tests completed
```



Legacy JavaScript Algorithms and Data Structures | ES6

```
}
```

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.

Run the Tests (Ctrl + Enter)

Reset this lesson

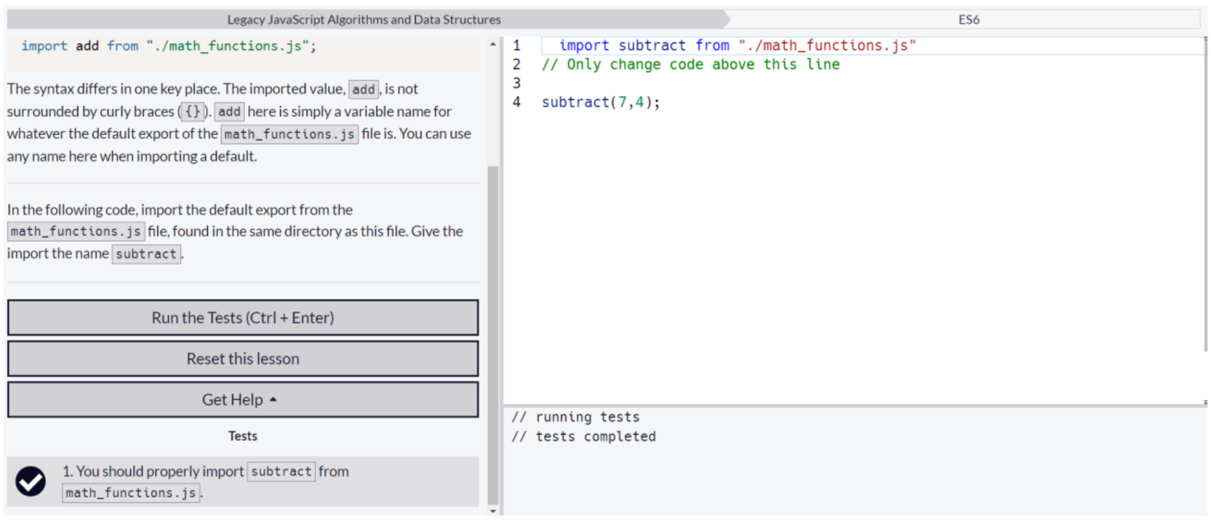
Get Help ▴

Tests

✓ 1. Your code should use an `export` fallback.

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

```
// running tests
// tests completed
```



```
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.

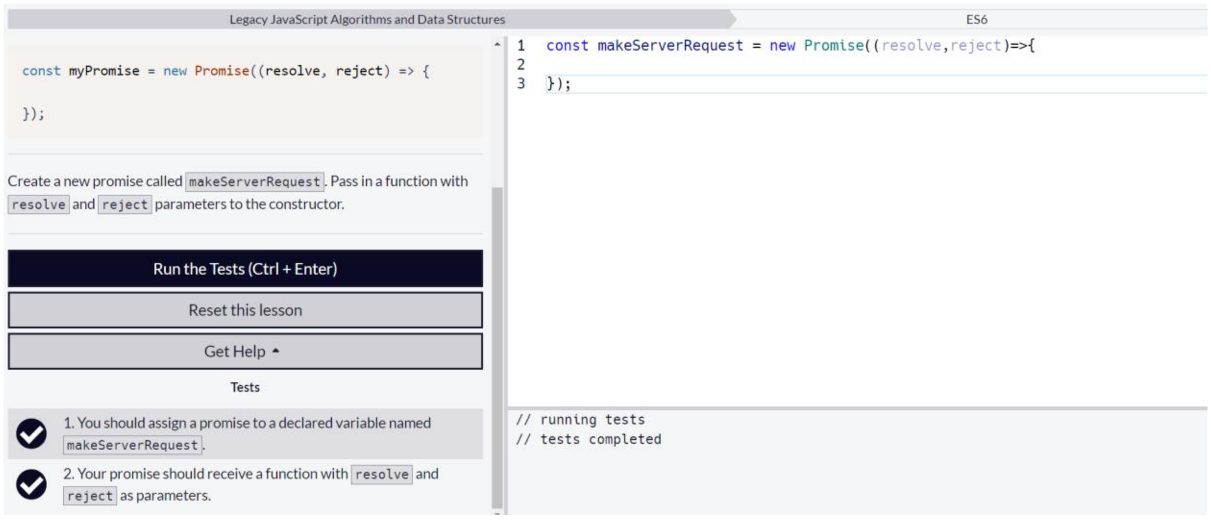
In the following code, import the default export from the `math_functions.js` file, found in the same directory as this file. Give the import the name `subtract`.

✓ 1. You should properly import `subtract` from `math_functions.js`.

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

subtract(7,4);
```

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures
ES6

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

});
```

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

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

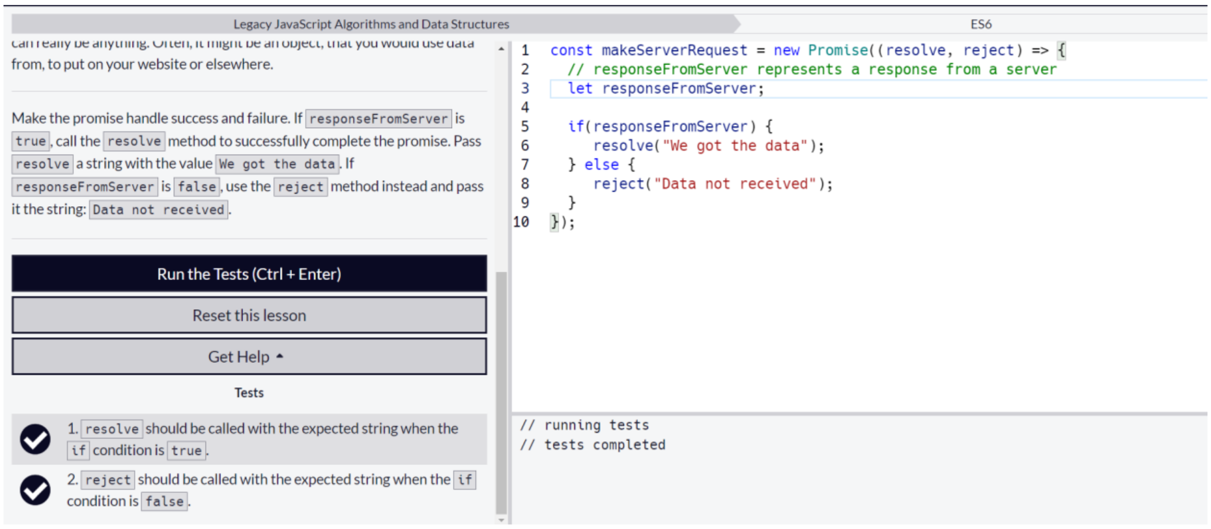
✓ 1. You should assign a promise to a declared variable named `makeServerRequest`.

✓ 2. Your promise should receive a function with `resolve` and `reject` as parameters.

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

});
```

```
// running tests
// tests completed
```



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`.

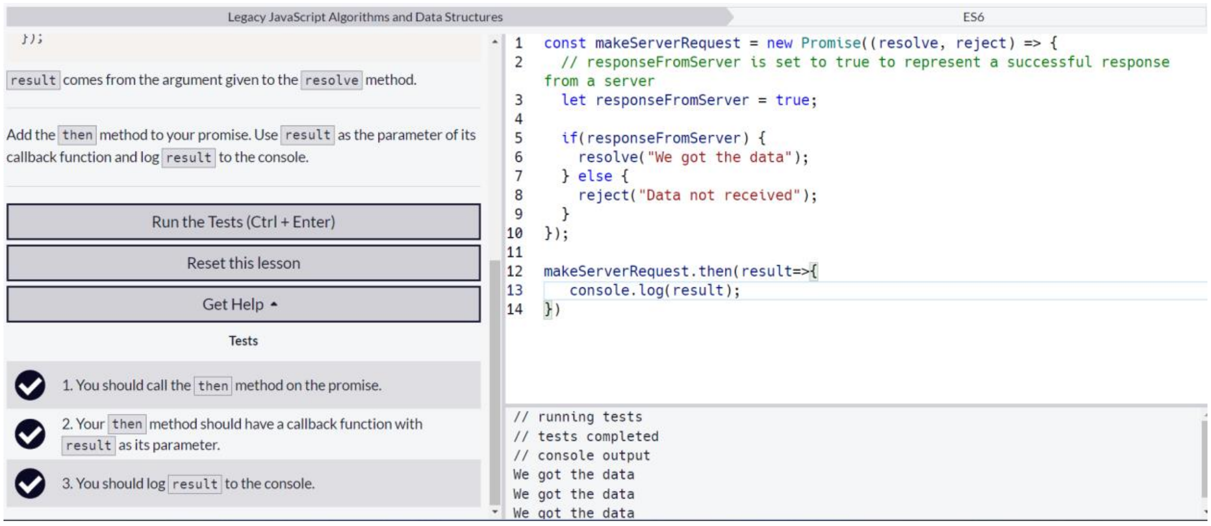
✓ 1. `resolve` should be called with the expected string when the `if` condition is `true`.

✓ 2. `reject` should be called with the expected string when the `if` condition is `false`.

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

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

```
// running tests
// tests completed
```


Legacy JavaScript Algorithms and Data Structures
ES6

```
});
```

`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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. You should call the `then` method on the promise.

✓ 2. Your `then` method should have a callback function with `result` as its parameter.

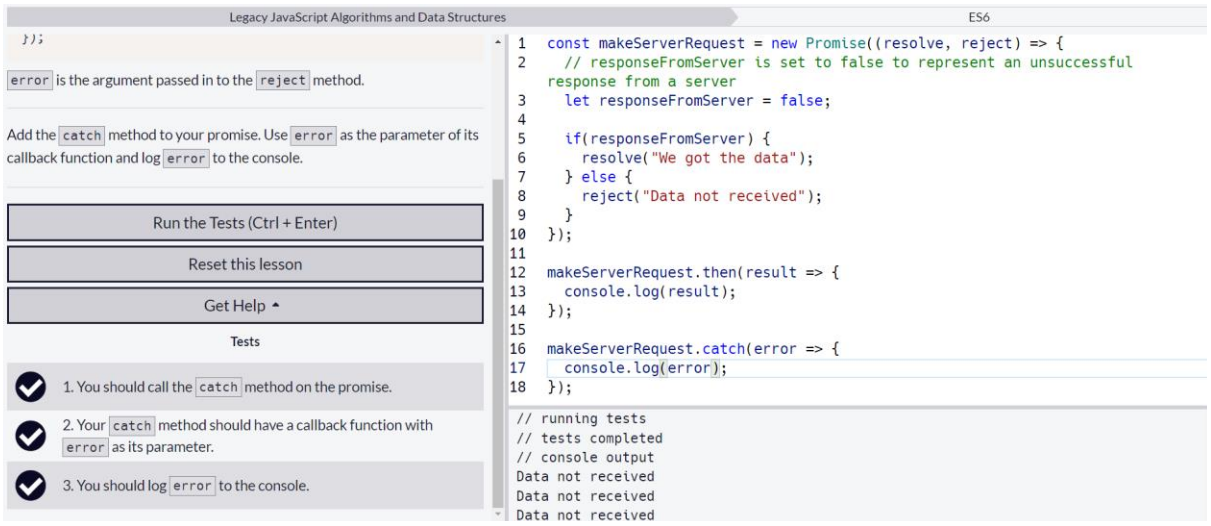
✓ 3. You should log `result` to the console.

```
const makeServerRequest = new Promise((resolve, reject) => {
  // responseFromServer is set to true to represent a successful 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);
})
```

```
// running tests
// tests completed
// console output
We got the data
We got the data
We got the data
```


Legacy JavaScript Algorithms and Data Structures
ES6

```
});
```

`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.

Run the Tests (Ctrl + Enter)
Reset this lesson
Get Help
Tests

✓ 1. You should call the `catch` method on the promise.

✓ 2. Your `catch` method should have a callback function with `error` as its parameter.

✓ 3. You should log `error` to the console.

```
const makeServerRequest = new Promise((resolve, reject) => {
  // responseFromServer is set to false to represent an unsuccessful 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);
});
```

```
// running tests
// tests completed
// console output
Data not received
Data not received
Data not received
```