

# Advanced JavaScript Concepts



with examples and  
hands-on exercises

---

**WEBUCATOR**

Copyright © 2023 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

**Version:** 1.1.0

## **The Authors**

### ***Nat Dunn***

Nat Dunn is the founder of Webucator ([www.webucator.com](http://www.webucator.com)), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

### ***Chris Minnick***

Chris Minnick, the co-founder of WatzThis?, has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies. A prolific writer, Chris has authored and co-authored books and articles on a wide range of Internet-related topics including HTML, CSS, mobile apps, e-commerce, e-business, Web design, XML, and application servers. His published books include Adventures in Coding, JavaScript For Kids For Dummies, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW E-Commerce Designer Certification Bible, and XHTML.

## **Class Files**

Download the class files used in this manual at

<https://static.webucator.com/media/public/materials/classfiles/JSC151-1.1.0-advanced-javascript-concepts.zip>.

## **Errata**

Corrections to errors in the manual can be found at <https://www.webucator.com/books/errata/>.

# Table of Contents

LESSON 1. Advanced JavaScript Concepts.....	1
Node.js.....	2
Scope, var, let, and const.....	4
Arrow Functions.....	7
Rest Parameters.....	11
Spread Operator.....	12
Array Destructuring.....	14
Template Literals.....	16
Objects.....	17
The this Object.....	18
Array map() Method.....	20
Array filter() Method.....	24
Array find() Method.....	26
JavaScript Modules.....	28
npm.....	34



# LESSON 1

## Advanced JavaScript Concepts

---

EVALUATION COPY: Not to be used in class.

### Topics Covered

- ☒ Block-scoped variables.
- ☒ Constants.
- ☒ Arrow functions.
- ☒ Rest parameters.
- ☒ The spread operator.
- ☒ Array destructuring.
- ☒ Template literals.
- ☒ Objects, Context, and the `this` object.
- ☒ The `map()`, `find()`, and `filter()` methods of arrays.
- ☒ JavaScript modules.
- ☒ Node.js and package managers.

Evaluation  
Copy

### Introduction

In this lesson, you'll dive a little deeper into JavaScript and learn some of its more advanced features. These features are particularly useful to know when working with JavaScript frameworks such as React, Vue, and Angular.

EVALUATION COPY: Not to be used in class.



## 1.1. Node.js

Node.js, pronounced to rhyme with “Toads say yes” and often referred to as just *Node*, is a JavaScript runtime that makes it easy to run JavaScript anywhere, not just in a web browser. Its original purpose and most common use is to create server-side web applications. Many JavaScript frameworks, including React, Vue, and Angular, make use of Node to create web applications.

### Installing Node

See <https://www.webucator.com/article/nodejs-and-node-package-manager-npm/> for instructions on checking to see if you have Node installed, and to install it if you don't. The instructions also show how to install npm, which we will cover later in this lesson. You should make sure you have that installed as well.

Node can be used for other purposes as well. For example, Node makes it possible to run JavaScript files at the command line, which is great for testing your JavaScript code. To run a JavaScript file at the command line, simply navigate to the folder that file is in and run `node file-name.js`:

```
PS ...\\AdvancedJSConcepts\\Demos> node file-name.js
```

Let's give it a try. In `AdvancedJSConcepts/Demos`, you will see a file called `hello-world.js` that simply outputs “Hello, world!” to the console:

### Demo 1.1: AdvancedJSConcepts/Demos/hello-world.js

```
1. console.log('Hello, world!');
```

### Code Explanation

To run this file:

1. Navigate to `AdvancedJSConcepts/Demos` in the terminal.
2. Run `node hello-world.js`
3. You should see something like this:

```
PS ...\\AdvancedJSConcepts\\Demos> node hello-world.js
Hello, world!
```

## ❖ 1.1.1. Node Shell

In addition to running JavaScript files, Node also allows you to run JavaScript at the command line using the Node shell. Type `node` at the command prompt to activate the Node shell:

```
PS ..\AdvancedJSConcepts\Demos> node
Welcome to Node.js v13.7.0.
Type ".help" for more information.
>
```

Then you can start writing JavaScript:

```
> const fruit = ['banana', 'apple', 'peach'];
undefined
> for (frr of fruit) {
...   console.log('I love a good ' + frr + '!');
... }
I love a good banana!
I love a good apple!
I love a good peach!
undefined
```

Evaluation  
Copy

The `undefined` output simply means that the statement didn't return anything.

To exit the Node shell, run `.exit` or press **Ctrl+D**.

### Node Version Manager (nvm)

You can run multiple versions of Node on the same computer using `nvm` for Mac and Linux (<https://github.com/nvm-sh/nvm>) or `nvm-windows` for Windows (<https://github.com/coreybutler/nvm-windows>).

**EVALUATION COPY: Not to be used in class.**



## 1.2. Scope, var, let, and const

Scope refers to the context in which an object can be referenced. JavaScript used to allow for only two scopes: global and function. All variables were declared with the `var` keyword. Variables declared in a function were scoped to that function, meaning that they are not visible outside of that function. All other variables were globally scoped, meaning that they are visible everywhere.

JavaScript now allows for block scoping. This is done with the `let` keyword for variables and the `const` keyword for constants. Blocks are denoted with curly braces. Constants declared with `const` and variables declared with `let` are visible within the block in which they are declared and all subblocks. If they are declared globally, they are visible everywhere.

Consider the following demo:



## Demo 1.2: AdvancedJSConcepts/Demos/scope.js

---

```
1.  if (true) {
2.    let aLet = 'aLet'; // block scoped
3.    var aVar = 'aVar'; // global
4.    const A_CONST = 'A_CONST'; // block scoped
5.    console.log('Found ' + aLet + ' in block.');
```

6. console.log('Found ' + aVar + ' in block.');

7. console.log('Found ' + A\_CONST + ' in block.');

8. }

9. console.log('-----');

10.

11. try {

12. console.log('Found ' + aLet + ' in global scope.');

13. } catch(e) {

14. console.log('Did not find aLet in global scope.');

15. }

16.

17. try {

18. console.log('Found ' + aVar + ' in global scope.');

19. } catch(e) {

20. console.log('Did not find aVar in global scope.');

21. }

22.

23. try {

24. console.log('Found ' + A\_CONST + ' in global scope.');

25. } catch(e) {

26. console.log('Did not find A\_CONST in global scope.');

27. }

28. console.log('-----');

29.

30. function outputLet() {

31. try {

32. console.log('Found ' + aLet + ' in function.');

33. } catch(e) {

34. console.log('Did not find aLet in function.');

35. }

36. }

37.

38. function outputVar() {

39. try {

40. console.log('Found ' + aVar + ' in function.');

41. } catch(e) {

42. console.log('Did not find aVar in function.');

43. }

44. }

```
45.  
46. function outputConst() {  
47.   try {  
48.     console.log('Found ' + A_CONST + ' in function.');49.   } catch(e) {  
50.     console.log('Did not find A_CONST in function.');51.   }  
52. }  
53.  
54. outputLet();  
55. outputVar();  
56. outputConst();
```

---

## Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node scope.js
```

```
Found aLet in block.
```

```
Found aVar in block.
```

```
Found A_CONST in block.
```

```
-----
```

```
Did not find aLet in global scope.
```

```
Found aVar in global scope.
```

```
Did not find A_CONST in global scope.
```

```
-----
```

```
Did not find aLet in function.
```

```
Found aVar in function.
```

```
Did not find A_CONST in function.
```

Note that **aLet**, **aVar**, and **A\_CONST** were all declared within a block, so it makes sense that all three can be found within that same block. But notice that **aVar** can also be found in the global scope and in the function scope; whereas **aLet** and **aConst** cannot. That is because variables declared with the **let** keyword and constants are block scoped. **As a rule**, you should avoid declaring variables with **var**.

---

### ❖ 1.2.1. When Constants aren't Constant

When you declare a constant, you create a pointer to a specific object. You **may not** change that pointer (i.e., you **cannot** assign a new value to a constant), but you can change the object that is assigned to the constant. Consider the following demo:

## Demo 1.3: AdvancedJSConcepts/Demos/constants.js

---

```
1.  const a = ['banana', 'apple'];
2.
3.  console.log(a);
4.
5.  // OK to change object constant points to.
6.  a.push('peach');
7.
8.  console.log(a);
9.
10. try {
11.    // Not OK to assign new value to constant.
12.    a = ['banana', 'apple', 'peach', 'cherry'];
13. } catch(e) {
14.    console.log('Error: ' + e.message);
15. }
```

---

### Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ...\\AdvancedJSConcepts\\Demos> node constants.js
['banana', 'apple']
['banana', 'apple', 'peach']
Error: Assignment to constant variable.
```

Note that appending to the array assigned to **a** is fine, but when we try to assign a new value to the constant, we get an error.

---

**EVALUATION COPY: Not to be used in class.**



## 1.3. Arrow Functions

Arrow functions (also called fat-arrow functions because they use the `=>` operator) are a more compact way of writing JavaScript functions. The syntax is shown below:

```
(parameters) => {  
  // function body  
}
```

To illustrate, consider this simple function, which squares *x* and returns the result:

```
function square(x) {  
  return x * x;  
}
```

The fat-arrow version of this function could be written like this:

```
(x) => {  
  return x * x;  
}
```

Notice that this function doesn't have a name, but you can give it one by assigning it to a constant:

```
const square = (x) => {  
  return x * x;  
}
```

Evaluation  
Copy

You call this function in the same way that you call a function created in the standard way:

```
square(2);
```

### ❖ 1.3.1. Shortening Arrow Functions

Fat-arrow functions that only have a `return` statement can be made shorter by removing the curly braces. When a single statement follows the fat-arrow operator, that statement is evaluated and returned. This is the equivalent of the function above:

```
const square = (x) => x * x;
```

And when a fat-arrow function takes only one parameter, you do not need to wrap that parameter in parentheses:

```
const square = x => x * x;
```

Try this at the Node shell:

```
> const square = x => x * x;  
undefined  
> square(5);  
25
```

Evaluation  
Copy


While these last methods are brief, they are more difficult to understand. In practice, you may prefer explicitly returning a value. However, it is not unlikely that you will come across code like we have shown above (Vue uses this!), so you should understand how it works.

The following demo includes all of these fat-arrow examples:

## Demo 1.4: AdvancedJSConcepts/Demos/fat-arrow.js

---

```
1.  // Standard function
2.  function square(x) {
3.      return x * x;
4.  }
5.
6.  let result = square(2);
7.  console.log(result);
8.
9.  // fat-arrow function
10. const square2 = (x) => {
11.     return x * x;
12. };
13.
14. result = square2(3);
15. console.log(result);
16.
17. // briefer fat-arrow function
18. // when curly braces aren't included, the expression
19. // to the right of the fat arrow is evaluated and returned.
20. const square3 = (x) => x * x;
21.
22. result = square3(4);
23. console.log(result);
24.
25. // briefest fat-arrow function
26. // when only one parameter is passed, parentheses
27. // are not required.
28. const square4 = x => x * x;
29.
30. result = square4(5);
31. console.log(result);
```



---

### Code Explanation

Run the file with Node. You will get the following output:

```
PS ...\\AdvancedJSConcepts\\Demos> node fat-arrow.js
4
9
16
25
```

---



## 1.4. Rest Parameters

A rest parameter is denoted with three periods preceding a variable name (e.g., ...myVar) and takes all the rest of the arguments passed to the function and packs them into an array. It must be the last parameter in a parameter list. Consider the following code:

### Demo 1.5: AdvancedJSConcepts/Demos/rest-param-1.js

```
1.  function addNums(...nums) {  
2.      let total = 0;  
3.      for (let num of nums) {  
4.          total += num;  
5.      }  
6.      const equation = nums.join(' + ') + ' = ' + String(total);  
7.      console.log(equation);  
8.  }  
9.  
10. addNums(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

### Code Explanation

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node rest-param-1.js  
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55
```

Here is another example, which shows a function that takes a rest parameter in addition to another parameter. Remember that the the rest parameter must be the last parameter in the function:

## Demo 1.6: AdvancedJSConcepts/Demos/rest-param-2.js

---

```
1. function countMarbles(personName, ...marbles) {  
2.     console.log(personName + ' has ' + marbles.length + ' marbles.');
```

---

```
3. }  
4.  
5. countMarbles('Joe', 'blue', 'green', 'yellow');
```

---

### Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node rest-param-2.js  
Joe has 3 marbles.
```

---

EVALUATION COPY: Not to be used in class.

Evaluation  
Copy

## 1.5. Spread Operator

The spread operator is also created using three periods, but it does the opposite of the rest parameter. The spread operator starts with an array or object and “spreads” it out into individual variables, as you can see in this demo:

## Demo 1.7: AdvancedJSConcepts/Demos/spread-operator-1.js

---

```
1. const fruits = ['banana', 'apple', 'peach', 'cherry'];  
2.  
3. console.log(fruits); // Outputs array  
4. console.log(...fruits); // Outputs each string in array individually
```

---

### Code Explanation

---

Run the file with Node. You will see that the spread operator separates the array into parts:



```
PS ...\\AdvancedJSConcepts\\Demos> node spread-operator-1.js
['banana', 'apple', 'peach', 'cherry']
banana apple peach cherry
```

---

The following example shows how to use the spread operator to append one array onto another:

### Demo 1.8: AdvancedJSConcepts/Demos/spread-operator-2.js

---

```
1.  const fruits = ['banana', 'apple', 'peach', 'cherry'];
2.  const veggies = ['squash', 'spinach', 'asparagus', 'peas'];
3.
4.  const foods = [];
5.
6.  // Attempt 1
7.  foods.push(fruits);
8.  foods.push(veggies);
9.
10. console.log('ATTEMPT 1:');
11. console.log(foods); // foods will contain an array of arrays
12.
13. // Clear foods
14. foods.length = 0;
15.
16. // Attempt 2
17. foods.push(...fruits);
18. foods.push(...veggies);
19.
20. console.log('-----');
21. console.log('ATTEMPT 2:');
22. console.log(foods); // foods will now contain an array of strings
```

---

### Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node spread-operator-2.js
```

```
ATTEMPT 1:
```

```
[[ 'banana', 'apple', 'peach', 'cherry'],  
  ['squash', 'spinach', 'asparagus', 'peas']]
```

```
-----
```

```
ATTEMPT 2:
```

```
['banana',  
 'apple',  
 'peach',  
 'cherry',  
 'squash',  
 'spinach',  
 'asparagus',  
 'peas']
```

Notice that when we don't use the spread operator, the full `fruits` and `veggies` arrays are pushed onto the `foods` array, creating an array of arrays. Using the spread operator breaks the `fruits` and `veggies` arrays into parts and pushes those parts onto the `foods` array.

**EVALUATION COPY: Not to be used in class.**



## 1.6. Array Destructuring

It is possible to assign the elements of an array to individual constants or variables using the following syntax:

```
const fruit = ['apple', 'banana', 'cherry'];  
const [a, b, c] = fruit;
```

After running this, `a` will contain 'apple', `b` will contain 'banana', and `c` will contain 'cherry'.

Array destructuring is often used with functions that return arrays, as shown in the following demo:

## Demo 1.9: AdvancedJSConcepts/Demos/array-destructuring.js

---

```
1.  // Return a random integer
2.  function randInt(low, high) {
3.      const rndDec = Math.random();
4.      const rndInt = Math.floor(rndDec * (high - low + 1) + low);
5.      return rndInt;
6.  }
7.
8.  // Return a random president as an array: [firstName, lastName]
9.  function randPresident() {
10.     const presidents = [
11.         'George Washington',
12.         'Thomas Jefferson',
13.         'Abraham Lincoln',
14.         'Teddy Roosevelt',
15.         'Richard Nixon',
16.         'Ronald Reagan',
17.         'Barack Obama'
18.     ];
19.
20.     const i = randInt(0, presidents.length-1);
21.     return presidents[i].split(' ');
22. }
23.
24. const [firstName, lastName] = randPresident();
25. console.log('First name:', firstName);
26. console.log('Last name:', lastName);
```

---

### Code Explanation

---

Run the file with Node. You will get the following output (presidents will vary):

```
PS ...\\AdvancedJSConcepts\\Demos> node array-destructuring.js
```

```
First name: Thomas
```

```
Last name: Jefferson
```

---

**EVALUATION COPY: Not to be used in class.**



## 1.7. Template Literals

When you use the concatenation operator (+) to combine literal text and variables, the code can get a little ugly:

```
const myGreeting = 'Hello, ' + personName + '. The date and time is ' + datetime + '.'
```

This code can be written in a cleaner way using template literals, which are enclosed with back-ticks (`). Template literals can contain placeholders, which are denoted with dollar signs and curly braces (e.g., `\${expression}`). The concatenated string above can be rewritten like this using a template literal:

```
const myGreeting = `Hello, ${personName}. The date and time is ${datetime}.`;
```

Try this in the Node shell:

```
> const datetime = new Date().toLocaleString();
undefined
> const personName = 'Nat Dunn';
undefined
> const myGreeting = `Hello, ${personName}. The current date and time is ${datetime}.`;
undefined
> myGreeting;
'Hello, Nat Dunn. The current date and time is 5/13/2022, 7:51:01 AM.'
```

And here is an example that spans multiple lines:

### Demo 1.10: AdvancedJSConcepts/Demos/template-literals.js

```
1. function madLib(name, characteristic, verb, adjective) {
2.     const haiku = `${name} is ${characteristic}
3.     I ${verb} with ${name} often
4.     How ${adjective} I am`;
5.     console.log(haiku);
6. }
7.
8. madLib('Mary', 'merry', 'sing', 'lucky');
```

## Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node template-literals.js
Mary is merry
I sing with Mary often
How lucky I am
```

As demonstrated, template literals can include multiple lines. If you want to add line breaks to format your code, but don't want those line breaks to be part of the literal, you must escape the newline with a backslash (\) like this:

```
const sentence = `This sentence should not include any newline characters \
but I am breaking it across lines in my code so that I don't \
have to scroll horizontally.`;
```

**EVALUATION COPY: Not to be used in class.**

*Eva's Cop*

## 1.8. Objects

You have worked a lot with objects already. Arrays are objects. Elements on an HTML page are objects. String, Number, Date and Math are built-in JavaScript objects. Essentially, everything except primitive types, like simple strings and numbers, is an object. And even simple strings and numbers can be treated like objects.

So, what is an object? An object is something that has properties and/or methods. Some examples:

1. An Array object has a `length` property and a `find()` method.
2. An HTML Element object has an `innerHTML` property and an `addEventListener()` method.
3. A Date object has a `getFullYear()` method.

You can create your own objects in two ways:

1. With a constructor function and the new keyword:

```
function Building(name, location) {  
  this.name = name;  
  this.location = location;  
  this.logLocation = function() {  
    document.getElementById('build').innerHTML = this.location;  
  }  
}  
  
const whiteHouse = new Building('White House',  
                                '1600 Pennsylvania Ave.');
```

Note that constructor functions are just regular functions, but they are meant to be called using the new keyword, in which case, they *construct* a new object. By convention, the first letter of a constructor function should be capitalized, making it clear that it is a constructor function.

2. With literal notation:

```
const whiteHouse = {  
  name: 'White House',  
  location: '1600 Pennsylvania Ave.',  
  logLocation: function() {  
    document.getElementById('build').innerHTML = this.location;  
  }  
};
```

As a general rule, you would use the constructor approach if you plan to create many objects of the same type and use the literal syntax approach if you're just creating a one-off object.

## 1.8. The this Object

JavaScript's `this` object is complex, but there are a few essential things that you need to understand:

1. Code within an object is said to be in that object's *context*.
2. The *global* context refers to the context of the top-level object. In browsers, the global context is the context within the window object.
3. Within an object, `this` refers to the object it is within. So, in browsers, `this` refers to the global context (the window object) unless it is used within another object.

The easiest way to understand this is to consider an example:

## Demo 1.11: AdvancedJSConcepts/Demos/this.html

---

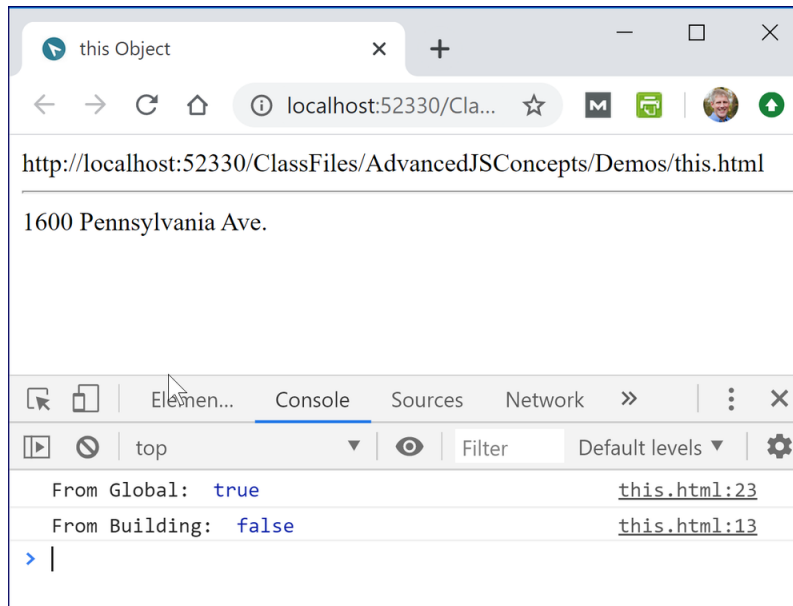
```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width, initial-scale=1">
6. <script>
7.   function Building(name, location) {
8.     this.name = name;
9.     this.location = location;
10.    this.logLocation = function() {
11.      document.getElementById('build').innerHTML = this.location;
12.    }
13.    console.log('From Building: ', window.location === this.location);
14.  }
15.
16.  window.addEventListener('load', (e) => {
17.    document.getElementById('win').innerHTML = this.location;
18.    const whiteHouse = new Building('White House',
19.                                     '1600 Pennsylvania Ave. ');
20.    whiteHouse.logLocation();
21.  });
22.
23.  console.log('From Global: ', window.location === this.location);
24. </script>
25. <title>this Object</title>
26. </head>
27. <body>
28.   <div id="win"></div>
29.   <hr>
30.   <div id="build"></div>
31. </body>
32. </html>
```

---

## Code Explanation

---

Open this file in the browser:



Notice that `this.location` outputs the `location` property of `window` when called from the global context, but it outputs the `location` property of the `Building` object when called from that object's method.

Also, notice the output in the console, which shows that `this.location` and `window.location` are the same when referenced in the global context, but not when referenced within the `Building` object.

**EVALUATION COPY: Not to be used in class.**



## 1.9. Array `map()` Method

The `map()` method of JavaScript arrays passes each value of the array to a callback function to create a new array based on the values of the original array. Consider the following code:



## Demo 1.12: AdvancedJSConcepts/Demos/without-map.js

---

```
1.  function cube(num) {
2.      return num * num * num;
3.  }
4.
5.  const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9];
6.
7.  const cubes = [];
8.  for (let num of nums) {
9.      cubes.push(cube(num));
10. }
11.
12. console.log(cubes);
```

---

### Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node without-map.js
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```

This code loops through the `nums` array, passes each element of the array to the `cube()` function, and pushes the result onto the `cubes` array.

---

The `for` loop in the preceding code can be replaced with this:

```
const cubes = nums.map(cube);
```

`nums.map(cube)` creates a new array by passing each element in `nums` to `cube()`.

As the following demo shows, this can be done with an anonymous function as well:

## Demo 1.13: AdvancedJSConcepts/Demos/map-1.js

---

```
1.  function cube(num) {
2.      return num * num * num;
3.  }
4.
5.  const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9];
6.
7.  // with named function
8.  const cubes = nums.map(cube);
9.
10. console.log(cubes);
11.
12. // with anonymous fat-arrow function
13. const squares = nums.map((num) => num * num );
14.
15. console.log(squares);
16.
17. // Note that nums is unchanged by the map() calls
18. console.log(nums); // unchanged
```

---

### Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node map-1.js
[1, 8, 27, 64, 125, 216, 343, 512, 729]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The preceding code uses a named function to create the `cubes` array and an anonymous function to create the `squares` array. Notice that `map()` does not alter the original array.

---

The following demos show how to use `map()` to create an HTML `select` list from an array of values. The first demo uses a named function and the second demo uses an anonymous function:

## Demo 1.14: AdvancedJSConcepts/Demos/map-2.js

---

```
1. function createOption(value) {
2.     return `<option value='${value}'>${value}</option>`;
3. }
4.
5. function createSelect(values) {
6.     let select = '<select>\n';
7.     let options = values.map(createOption);
8.     for (option of options) {
9.         select += '\t' + option + '\n';
10.    }
11.    select += '</select>';
12.    return select;
13. }
14.
15. const fruits = ['banana', 'apple', 'peach', 'cherry'];
16. let select = createSelect(fruits);
17. console.log(select);
```

---

## Demo 1.15: AdvancedJSConcepts/Demos/map-3.js

---

```
1. function createSelect(values) {
2.     let select = '<select>\n';
3.     const options = values.map((value) =>
4.         `<option value='${value}'>${value}</option>`
5.     );
6.     for (option of options) {
7.         select += '\t' + option + '\n';
8.     }
9.     select += '</select>';
10.    return select;
11. }
12.
13. const fruits = ['banana', 'apple', 'peach', 'cherry'];
14. let select = createSelect(fruits);
15. console.log(select);
```

---

Both of the files above will output a `select` list, which you could then inject into `innerHTML` of an element on the page:

```
PS ...\\AdvancedJSConcepts\\Demos> node map-2.js
<select>
  <option value='banana'>banana</option>
  <option value='apple'>apple</option>
  <option value='peach'>peach</option>
  <option value='cherry'>cherry</option>
</select>
```

EVALUATION COPY: Not to be used in class.



## 1.10. Array filter() Method

The `filter()` method of JavaScript arrays passes each value of the array to a callback testing function to create a new array containing only those values that passed the test. Consider the following code:

### Demo 1.16: AdvancedJSConcepts/Demos/without-filter.js

```
1.  function isOdd(num) {
2.    return num % 2 === 1;
3.  }
4.
5.  const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9];
6.
7.  const oddNumbers = [];
8.  for (let num of nums) {
9.    if (isOdd(num)) {
10.     oddNumbers.push(num);
11.    }
12.  }
13.
14.  console.log(oddNumbers);
```

#### Code Explanation

Run the file with Node. You will get the following output:

```
PS ...\\AdvancedJSConcepts\\Demos> node without-filter.js
[ 1, 3, 5, 7, 9 ]
```

This code loops through the `nums` array, passes each element of the array to the `isOdd()` function, and, if `isOdd()` returns `true`, pushes the element onto the `oddNumbers` array.

---

The `for` loop in the preceding example can be replaced with the following code:

```
const oddNumbers = nums.filter(isOdd);
```

`nums.filter(isOdd)` creates a new array from `nums` that only includes those values that pass the `isOdd()` test.

As the following demo shows, this can be done with an anonymous function as well:

### Demo 1.17: AdvancedJSConcepts/Demos/filter.js

---

```
1.  function isOdd(num) {  
2.      return num % 2 === 1;  
3.  }  
4.  
5.  const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
6.  
7.  // with named function  
8.  const oddNumbers = nums.filter(isOdd);  
9.  
10. console.log(oddNumbers);  
11.  
12. // with anonymous fat-arrow function  
13. const evenNumbers = nums.filter((num) => num % 2 === 0);  
14.  
15. console.log(evenNumbers);  
16.  
17. // Note that nums is unchanged by the filter() calls  
18. console.log(nums);
```

---

### Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ...\\AdvancedJSConcepts\\Demos> node filter.js  
[ 1, 3, 5, 7, 9 ]  
[ 2, 4, 6, 8 ]  
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This uses a named function to create the `oddNumbers` array and an anonymous function to create the `evenNumbers` array. Notice that `filter()` does not alter the original array.

---

EVALUATION COPY: Not to be used in class.

---



## 1.11. Array `find()` Method

The `find()` method of JavaScript arrays iterates through an array passing each element to a callback testing function until one of those elements passes the test, at which point, it returns that element. If no element passes the test, it returns `undefined`. Consider the following code:

### Demo 1.18: `AdvancedJSConcepts/Demos/without-find.js`

---

```
1.  function isAdult(person) {
2.      return person.age >= 18;
3.  }
4.
5.  const people = [
6.      {name: 'Cindy', age: 6},
7.      {name: 'Bobby', age: 7},
8.      {name: 'Jan', age: 10},
9.      {name: 'Peter', age: 10},
10.     {name: 'Marsha', age: 12},
11.     {name: 'Greg', age: 13},
12.     {name: 'Alice', age: 44}
13. ]
14.
15. let firstAdult;
16. for (let person of people) {
17.     if (isAdult(person)) {
18.         firstAdult = person;
19.         break;
20.     }
21. }
22.
23. console.log(firstAdult);
```

---

## Code Explanation

---

Run the file with Node. You will get the following output:

```
PS ..\AdvancedJSConcepts\Demos> node without-find.js
{ name: 'Alice', age: 44 }
```

This code loops through the `people` array, passing each element of the array to the `isAdult()` function until it finds an element that passes the `isAdult()` test, at which point it breaks out of the `for` loop.

---

In the following code, we do the same thing using the `find()` method:

### Demo 1.19: AdvancedJSConcepts/Demos/find.js

---

```
1.  function isAdult(person) {
2.      return person.age >= 18;
3.  }
4.
5.  const people = [
6.      {name: 'Cindy', age: 6},
7.      {name: 'Bobby', age: 7},
8.      {name: 'Jan', age: 10},
9.      {name: 'Peter', age: 10},
10.     {name: 'Marsha', age: 12},
11.     {name: 'Greg', age: 13},
12.     {name: 'Alice', age: 44}
13. ]
14.
15. const firstAdult = people.find(isAdult);
16.
17. console.log(firstAdult);
```

---

## Code Explanation

---

Run the file with Node and you will see that it also returns Alice:

```
PS ..\AdvancedJSConcepts\Demos> node find.js
{ name: 'Alice', age: 44 }
```

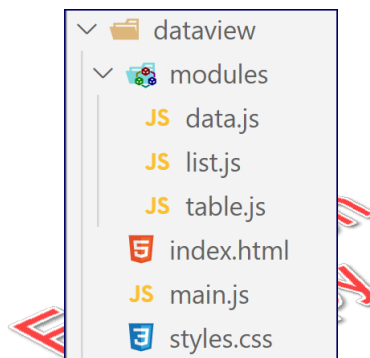
---



## 1.12. JavaScript Modules

JavaScript modules are files that either import or are imported into other JavaScript modules. JavaScript frameworks, like React and Vue, use modules to make it easier to separate applications into components.

Consider the following directory structure:



The `index.html` file includes `main.js` using a `<script>` tag. Here's the code:

### Demo 1.20: AdvancedJSConcepts/Demos/dataview/index.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4. <meta charset="UTF-8">
5. <meta name="viewport" content="width=device-width, initial-scale=1">
6. <link href="styles.css" rel="stylesheet">
7. <script type="module" src="main.js"></script>
8. <title>My Data</title>
9. </head>
10. <body>
11.   <div id="my-data"></div>
12. </body>
13. </html>
```



## Code Explanation

---

### Things to notice:

1. The `script` element has a `type` attribute set to `"module"`. This indicates that `main.js` may import other modules.
  2. There is no content in the body except for an empty `div` element with the `id` of `"my-data"`. We will inject HTML into that element.
- 

Now, take a look at `main.js`:

### Demo 1.21: AdvancedJSConcepts/Demos/dataview/main.js

---

```
1. // import modules
2. import {init, addItem, removeItem} from './modules/data.js';
3. import create from './modules/list.js';
4.
5. // create data object
6. const data = init();
7.
8. // add items to data object
9. addItem(data, 'NY', 'New York');
10. addItem(data, 'CA', 'California');
11. addItem(data, 'TX', 'Texas');
12. addItem(data, 'FL', 'Florida');
13.
14. // remove an item from the data object
15. removeItem(data, 'CA');
16.
17. // call create, passing the data, the id of the target, and a heading
18. create(data, 'my-data', 'State Abbreviations');
```

---

## Code Explanation

---

### Things to notice:

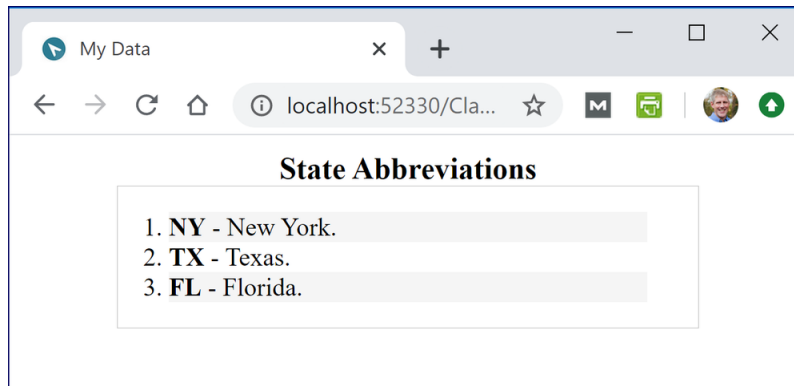
1. The code begins by importing functions from `./modules/data.js` and `./modules/list.js`. The `"./"` at the beginning of the path is required, indicating that this is a relative path.
  2. The two imports are slightly different from each other:
    - A. The first imports three specific named functions. This is done by listing the functions within curly braces.
-

- B. The second import doesn't use curly braces. This indicates that it is importing the default function exported by `./modules/list.js`.

We will look at both of these modules in a moment.

3. The rest of the code in this file uses the functions that were imported to create an HTML list and inject that list into the “my-data” div.

The resulting page, which uses some styles from `styles.css`, will look like this:



## Local Development Server

When using modules, your HTML page must be delivered from a server. You can configure the Visual Studio Code Open in Browser<sup>1</sup> extension to deliver pages from a local development server:

1. In Visual Studio Code, select **File > Preferences > Settings**.
2. Search **Settings** for “Open in Default Browser”.
3. Check the **Open with local http server** checkbox.
4. Then you can right-click your HTML page and select **Open in Default Browser** to open a web page in the browser. The URL will begin with something like “http://localhost:52330”.

Review the two files being imported.

1. <https://marketplace.visualstudio.com/items?itemName=peakchen90.open-html-in-browser>

## Demo 1.22: AdvancedJSConcepts/Demos/dataview/modules/data.js

---

```
1.  function init() {
2.      return {}
3.  }
4.
5.  function addItem(dict, key, value) {
6.      dict[key] = value;
7.  }
8.
9.  function removeItem(dict, key) {
10.     delete dict[key];
11.  }
12.
13.  export {init, addItem, removeItem};
```

---

### Code Explanation

---

This is the data module. It has a few simple functions for creating an object and adding and removing items from it.

The interesting piece is the last line, which exports these three functions, thereby making it possible for other modules to import them as `main.js` does.

## Demo 1.23: AdvancedJSConcepts/Demos/dataview/modules/list.js

---

```
1.  export default function create(data, target, title) {
2.      const list = document.createElement('ol');
3.
4.      for (let key in data) {
5.          const item = document.createElement('li');
6.          const value = data[key];
7.          item.innerHTML = `<strong>${key}</strong> - ${value}.`;
8.          list.appendChild(item);
9.      }
10.
11.     const header = document.createElement('h3');
12.     header.innerHTML = title;
13.
14.     document.getElementById(target).appendChild(header);
15.     document.getElementById(target).appendChild(list);
16. }
```

---

## Code Explanation

---

This is the `list` module. Its `create()` function uses the passed-in `data` object to create the HTML list and inject it into the element with the `id` equal to the passed-in `target`.

The most interesting piece of this file is the `export default` at the beginning of the first line. This indicates that this function will be exported automatically when the `list.js` module is imported. Typically, the importing file will use the same name for that function, but it doesn't have to, and, as you'll soon see, there are cases when it cannot.

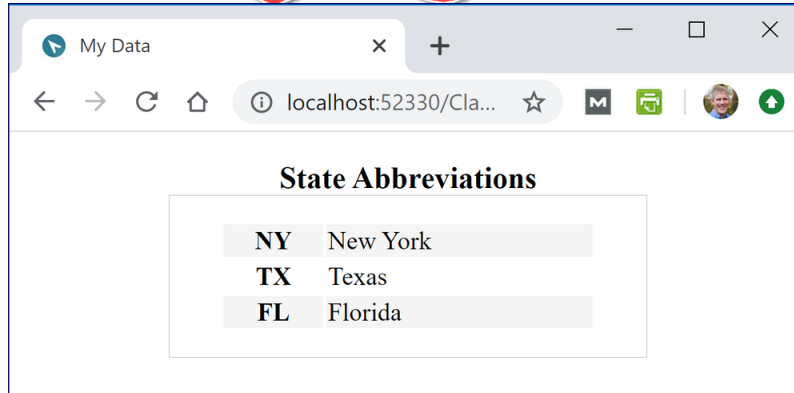
---

### ❖ 1.12.1. Try This

1. Open `AdvancedJSConcepts/Demos/dataview/index.html` in your browser.
2. Open `AdvancedJSConcepts/Demos/dataview/main.js` in your editor.
3. Change the second `import` statement to import from `table.js` instead of `list.js`:

```
import create from './modules/table.js';
```

4. Refresh the browser page. Notice it now shows a table:



5. Open `AdvancedJSConcepts/Demos/dataview/modules/table.js` in your editor and examine the code. Like `list.js`, it exports a `create()` function by default. That function uses the passed-in `data` object to create the HTML table and inject it into the element with the `id` equal to the passed-in `target`.

Here is the file:

## Demo 1.24: AdvancedJSConcepts/Demos/dataview/modules/table.js

---

```
1. export default function create(data, target, title) {
2.     const table = document.createElement('table');
3.
4.     const caption = document.createElement('caption');
5.     caption.innerHTML = title;
6.     table.appendChild(caption);
7.
8.     const tbody = document.createElement('tbody');
9.     table.appendChild(tbody);
10.
11.     for (let key in data) {
12.         const row = document.createElement('tr');
13.         const th = document.createElement('th');
14.         const td = document.createElement('td');
15.         const value = data[key];
16.         th.innerHTML = key;
17.         td.innerHTML = value;
18.         row.appendChild(th);
19.         row.appendChild(td);
20.         tbody.appendChild(row);
21.     }
22.
23.     document.getElementById(target).appendChild(table);
24. }
```

---

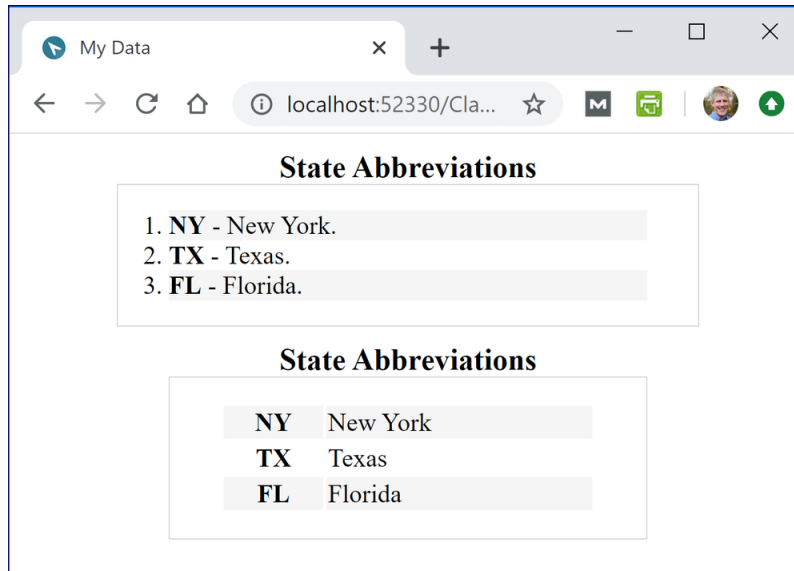
Now, think about this: what would happen if we wanted to include both the table and the list in the target div? Both modules export a default function named `create()`, but we cannot call them both `create()` in `main.js`. Because the two modules use default `export`, the function is imported as an *unnamed* function. In the importing file, we can call it whatever we want, so to avoid a conflict, we can do this:

```
import createList from './modules/list.js';
import createTable from './modules/table.js';
```

Then, we can inject the list and table into the div like this:

```
createList(data, 'my-data', 'State Abbreviations');
createTable(data, 'my-data', 'State Abbreviations');
```

Give it a try. When you're done, the page should look like this:



Note it is also possible to get conflicts when importing named functions. You can avoid that by using the `as` keyword and an alias:

```
import {init as initData} from './modules/data.js';
```

**EVALUATION COPY: Not to be used in class.**



## 1.13. npm

Node.js is free and thousands of Node programs (called *packages*) have been written by JavaScript developers. Because so many packages exist, it is necessary to *manage* them. Enter *package managers*. A package manager is software that simplifies the process of installing, uninstalling, and updating packages. When you install Node, it automatically installs a Node package manager named *npm*. The result is that JavaScript programmers today have a vast library of free tools, which can be found easily at <https://www.npmjs.com> and quickly installed with npm. Let's look at a simple one.

### ❖ 1.13.1. A Little Riddle

The following items can pass through the green glass door:

1. puddles
2. mommies
3. aardvarks
4. balloons

The following items cannot pass through the green glass door:

1. ponds
2. moms
3. anteaters
4. kites

Knowing that, which of the following can pass through the green glass door?

1. bananas
2. apples
3. pears
4. grapes
5. cherries

Evaluation  
Copy

Did you figure it out? The two that can pass are **app**les and cher**rr**ies. Any word with a double letter can pass through the **green glass door**.

Here is a function for checking whether a word can pass through the green glass door:

```
function greenGlassDoor(word) {  
  let letter = '';  
  for (let i = 0; i < word.length; i++) {  
    const newLetter = word.charAt(i).toLowerCase();  
    if (newLetter === letter) {  
      return i;  
    }  
    letter = newLetter;  
  }  
  return 0;  
}
```

Imagine you want to test that function by passing a bunch of randomly generated strings. You could write your own function for creating a random string, but you don't have to...

## The randomstring Package


As its name implies, the `randomstring` package creates random strings. Visit <https://www.npmjs.com/package/randomstring> and briefly review the **Usage** documentation.

Now review the following demo:

### Demo 1.25: AdvancedJSConcepts/Demos/random-string.js

---

```
1. // First run npm install randomstring
2. // Documentation at https://www.npmjs.com/package/randomstring
3.
4. var randomstring = require('randomstring');
5.
6. const a = randomstring.generate();
7. console.log(a);
8.
9. const b = randomstring.generate(7);
10. console.log(b);
11.
12. const c = randomstring.generate({
13.   length: 12,
14.   charset: 'alphanumeric'
15. });
16. console.log(c);
17.
18. const d = randomstring.generate({
19.   charset: 'abc'
20. });
21. console.log(d);
```



### Code Explanation

---

This creates random strings by passing different parameters to `randomstring.generate()`. It then outputs the created strings to the console.

1. Open `AdvancedJSConcepts/Demos` in the terminal.
2. Run `random-string.js` with Node (`node random-string.js`). You will get an error like this one:

```
Error: Cannot find module 'randomstring'
```

That's because you haven't installed the package yet.



3. Now, install randomstring:

```
npm install randomstring
```

4. Run random-string.js with Node again. This time you should get output similar to the following, though your random strings will be different:


```
PS ..\AdvancedJSConcepts\Demos> node random-string.js  
mqr5iEYYI8wUoIkTYuqN8U2rSX0HVM4I  
dDCah1C  
pRdXngoDHANd  
baaabcaabaccaaccabaaabbccacacaba
```

Now that we can easily create random strings, we can use this to test our `greenGlassDoor()` function:

## Demo 1.26: AdvancedJSConcepts/Demos/test-green-glass-door.js

---

```
1.  function greenGlassDoor(word) {
2.    let letter = '';
3.    for (let i = 0; i < word.length; i++) {
4.      const newLetter = word.charAt(i).toLowerCase();
5.      if (newLetter === letter) {
6.        return i;
7.      }
8.      letter = newLetter;
9.    }
10.   return 0;
11. }
12.
13. const randomstring = require('randomstring');
14. const testWords = [];
15. for (let i=0; i<10; i++) {
16.   const word = randomstring.generate({
17.     length: 10,
18.     charset: 'alphanumeric'
19.   });
20.   testWords.push(word);
21. }
22.
23. for (let w of testWords) {
24.   const i = greenGlassDoor(w);
25.   if (i) {
26.     console.log(`${w} passes: ${w[i-1]}${w[i]} at position ${i}.`);
27.   } else {
28.     console.log(`${w} cannot pass.`);
29.   }
30. }
```



---


### Code Explanation

Notice that we include `randomstring` using Node's `require()` function, which is similar to `import`, but can be done outside of a module:

```
const randomstring = require('randomstring');
```

Run the file with Node. You will get output similar to the following:

```
PS ...\\AdvancedJSConcepts\\Demos> node test-green-glass-door.js
EMVKPuBXzs cannot pass.
vKdLQckMxM cannot pass.
vB0HsFGSzs cannot pass.
qVVsaGiRlb can pass: VV at position 2.
Zdd0SJCywr can pass: dd at position 2.
DjahnFqxf cannot pass.
aPcqysTiuK cannot pass.
bPsZQvJrRX can pass: rR at position 8.
qiCWIlGZjd cannot pass.
vYGDwKyXJi cannot pass.
```



---

## Conclusion

In this lesson, you have learned about some of the more advanced features of JavaScript. You should now be ready to learn a JavaScript framework.