# **Lesson 6: Syntax**

## **1. Video: Harmony, ES6, ES2015**

Welcome to this course on the future of JavaScript.

James Parkes and Richard Kalehoff, instructors for this course.

Technically, Harmony ES6 and ES2015, are all different names for virtually the same thing.

The important thing is that these names represent the biggest update to the JavaScript programming language today. Fast forward from 1995 to today and JavaScript has undergone a renaissance of sorts by making some much needed improvements to the language. And with these improvements come a whole new batch of keywords, ways of writing functions, asynchronous goodies and so much more.

In this course, we will explore the additions to the JavaScript programming language so you can write faster, cleaner, and more efficient code. In this first lesson, we will start by outlining changes and additions to the **JavaScript Syntax**.

In lesson two, we will investigate updates to **JavaScript Functions**.

In lesson three, we will cover new **ES6 Built-ins**.

And in lesson four, we will wrap things up by showing you how you can incorporate these latest updates into your next JavaScript project with **JavaScript Developer-fu**.

What shall we start with first? Well, a bunch of keywords have been added to the language. So, let's start with one of those. Let us take a look at let in context.

## **2. Let and Const**

There are now two new ways to **declare variables** in JavaScript: **let** and **const**. The only way to declare a variable in JavaScript before was to use the keyword var. Take a look at the following code:

|  |
| --- |
| **function getClothing(isCold) {  if (isCold) {  var freezing = 'Grab a jacket!';  } else {  var hot = 'It’s a shorts kind of day.';  console.log(freezing);  } }** |

What do you expect to be the output from running getClothing(false)?

### **Hoisting**

Hoisting is a result of how JavaScript is interpreted by your browser. Essentially, before any JavaScript code is executed, all variables are "hoisted", which means they're raised to the top of the function scope. So at run-time, the getClothing() function actually looks more like this…

|  |
| --- |
| function getClothing(isCold) {  var freezing, hot;  if (isCold) {  freezing = 'Grab a jacket!';  } else {  hot = 'It’s a shorts kind of day.';  console.log(freezing);  } } |

### **let and const**

Variables declared with let and const eliminate this specific issue of hoisting because they’re scoped **to the block**, not to the function. Previously, when you used var, variables were either scoped *globally* or *locally* to an entire function scope.

If a variable is declared using let or const inside a block of code (denoted by curly braces { }), then the variable is stuck in what is known as the **temporal dead zone** until the variable’s declaration is processed. This behavior prevents variables from being accessed only until after they’ve been declared.

What do you expect to be the output from running getClothing(false)?

|  |
| --- |
| function getClothing(isCold) {  if (isCold) {  const freezing = 'Grab a jacket!';  } else {  const hot = 'It’s a shorts kind of day.';  console.log(freezing);  } } |

NOTE: Variables declared with let and const are only available within the block they’re declared

### **Rules for using let and const**

let and const also have some other interesting properties.

* Variables declared with let can be reassigned, but can’t be redeclared in the same scope.
* Variables declared with const must be assigned an initial value, but can’t be redeclared in the same scope, and can’t be reassigned.

What do you expect to be output from running the following code?

|  |
| --- |
| let instructor = 'James'; instructor = 'Richard'; console.log(instructor); |

### **Use cases**

The big question is when should you use let and const? The general rule of thumb is as follows:

* use let when you plan to reassign new values to a variable.
* use const when you don’t plan on reassigning new values to a variable.

Since const is the strictest way to declare a variable, we suggest that you always declare variables with const because it'll make your code easier to reason about since you know the identifiers won't change throughout the lifetime of your program. If you find that you need to update a variable or change it, then go back and switch it from const to let.

That’s pretty straightforward, right? But what about var?

### **What about var?**

Is there any reason to use var anymore? *Not really*.

There are some arguments that can be made for using var in situations where you want to globally define variables, but this is often considered bad practice and should be avoided. From now on, we suggest ditching var in place of using let and const.

## **3. Quiz: Using Let and Const**

### **Directions:**

Replace the variable declarations using let or const.

### **Your Code:**

|  |
| --- |
| var CHARACTER\_LIMIT = 255; var posts = [  "#DeepLearning transforms everything from self-driving cars to language translations. AND it's our new Nanodegree!",  "Within your first week of the VR Developer Nanodegree Program, you'll make your own virtual reality app",  "I just finished @udacity's Front-End Web Developer Nanodegree. Check it out!" ];  *// prints posts to the console* function displayPosts() {  for (var i = 0; i < posts.length; i++) {  console.log(posts[i].slice(0, CHARACTER\_LIMIT));  } }  displayPosts(); |

## **4. Template Literals**

Prior to ES6, the old way to concatenate strings together was by using the string concatenation operator ( +).

|  |
| --- |
| const student = {  name: 'Richard Kalehoff',  guardian: 'Mr. Kalehoff' };  const teacher = {  name: 'Mrs. Wilson',  room: 'N231' }  let message = student.name + ' please see ' + teacher.name + ' in ' + teacher.room + ' to pick up your report card.'; |

Returns: Richard Kalehoff please see Mrs. Wilson in N231 to pick up your report card.

This works alright, but it gets more complicated when you need to build multi-line strings.

|  |
| --- |
| let note = teacher.name + ',\n\n' +  'Please excuse ' + student.name + '.\n' +  'He is recovering from the flu.\n\n' +  'Thank you,\n' +  Student.guardian; |

**Returns:**

Mrs. Wilson,

Please excuse Richard Kalehoff.

He is recovering from the flu.

Thank you,

Mr. Kalehoff

However, that’s changed with the introduction of *template literals* (previously referred to as "template strings" in development releases of ES6).

**NOTE:** As an alternative to using the string concatenation operator ( + ), you can use the String's concat() method, but both options are rather clunky for simulating true [string interpolation](https://en.wikipedia.org/wiki/String_interpolation).

### **Template Literals**

**Template literals** are essentially string literals that include embedded expressions.

Denoted with backticks ( `` ) instead of single quotes ( '' ) or double quotes ( "" ), template literals can contain placeholders which are represented using ${expression}. This makes it *much easier* to build strings.

Here's the previous examples using template literals:

|  |
| --- |
| let message = `${student.name} please see ${teacher.name} in ${teacher.room} to pick up your report card.`; |

Returns: Richard Kalehoff please see Mrs. Wilson in N231 to pick up your report card.

By using template literals, you can drop the quotes along with the string concatenation operator. Also, you can reference the object's properties inside expressions.

...but what about the multi-line example from before?

|  |
| --- |
| var note = `${teacher.name},   Please excuse ${student.name}.  He is recovering from the flu.   Thank you,  ${student.guardian}`; |

Template literals make multi-line strings easier to read and more concise.

Here’s where template literals really shine. In the code above, the quotes and string concatenation operator have been dropped, as well as the newline characters ( \n ). That's because template literals also preserve newlines as part of the string!

TIP: Embedded expressions inside template literals can do more than just reference variables. You can perform operations, call functions and use loops inside embedded expressions!

## **5. Quiz: Build an HTML Fragment**

### **Directions:**

Modify the createAnimalTradingCardHTML() function to use a template literal for cardHTML.

### **Your Code:**

|  |
| --- |
| const cheetah = {  name: 'Cheetah',  scientificName: 'Acinonyx jubatus',  lifespan: '10-12 years',  speed: '68-75 mph',  diet: 'carnivore',  summary: 'Fastest mammal on land, the cheetah can reach speeds of 60 or perhaps even 70 miles (97 or 113 kilometers) an hour over short distances. It usually chases its prey at only about half that speed, however. After a chase, a cheetah needs half an hour to catch its breath before it can eat.',  fact: 'Cheetahs have "tear marks" that run from the inside corners of their eyes down to the outside edges of their mouth.' };  *// creates an animal trading card* function createAnimalTradingCardHTML(animal) {  const cardHTML = '<div class="card">' +  '<h3 class="name">' + animal.name + '</h3>' +  '<img src="' + animal.name + '.jpg" alt="' + animal.name +'" class="picture">' +  '<div class="description">' +  '<p class="fact">' + animal.fact + '</p>' +  '<ul class="details">' +  '<li><span class="bold">Scientific Name</span>: ' + animal.scientificName + '</li>' +  '<li><span class="bold">Average Lifespan</span>: ' + animal.lifespan + '</li>' +  '<li><span class="bold">Average Speed</span>: ' + animal.speed + '</li>' +  '<li><span class="bold">Diet</span>: ' + animal.diet + '</li>' +  '</ul>' +  '<p class="brief">' + animal.summary + '</p>' +  '</div>' +  '</div>';   return cardHTML; }  console.log(createAnimalTradingCardHTML(cheetah)); |

## **6. Destructuring**

In ES6, you can extract data from arrays and objects into distinct variables using *destructuring*.

This probably sounds like something you’ve done before, for example, look at the two code snippets below that extract data using pre-ES6 techniques:

|  |
| --- |
| const point = [10, 25, -34];  const x = point[0]; const y = point[1]; const z = point[2];  console.log(x, y, z); |

Prints: 10 25 -34

The example above shows extracting values from an array.

#### **const** gemstone = { type: 'quartz', color: 'rose', karat: 21.29 }; **const** type = gemstone.type; **const** color = gemstone.color; **const** karat = gemstone.karat; console.log(type, color, karat);

Prints: quartz rose 21.29

And this example shows extracting values from an object.

Both are pretty straightforward, however, neither of these examples are actually using destructuring.

So what exactly is *destructuring*?

### **Destructuring**

**Destructuring** borrows inspiration from languages like [Perl](https://en.wikipedia.org/wiki/Perl) and [Python](https://en.wikipedia.org/wiki/Python_%28programming_language%29) by allowing you to specify the elements you want to extract from an array or object *on the left side of an assignment*. It sounds a little weird, but you can actually achieve the same result as before, but with much less code; and it's still easy to understand.

Let’s take a look at both examples rewritten using destructuring.

### **Destructuring values from an array**

#### **const** point = [10, 25, -34]; **const** [x, y, z] = point; console.log(x, y, z);

Prints: 10 25 -34

In this example, the brackets [ ] represent the array being destructured and x, y, and z represent the variables where you want to store the values from the array. Notice how you don’t have to specify the indexes for where to extract the values from because the indexes are implied.

**TIP:** You can also ignore values when destructuring arrays. For example, const [x, , z] = point; ignores the y coordinate and discards it.

### **QUESTION 1 OF 2**

What do you expect to be the value of second after running the following code?

#### **let** positions = ['Gabrielle', 'Jarrod', 'Kate', 'Fernando', 'Mike', 'Walter']; **let** [first, second, third] = positions;

* Kate
* Gabrielle
* Jarrod
* Walter

### 

### **Destructuring values from an object**

#### **const** gemstone = { type: 'quartz', color: 'rose', karat: 21.29 }; **const** {type, color, karat} = gemstone; console.log(type, color, karat);

**Prints:** quartz rose 21.29

In this example, the curly braces { } represent the object being destructured and type, color, and karat represent the variables where you want to store the properties from the object. Notice how you don’t have to specify the property from where to extract the values. Because gemstone has a property named type, the value is automatically stored in the typevariable. Similarly, gemstone has a color property, so the value of color automatically gets stored in the color variable. And it's the same with karat.

**TIP:** You can also specify the values you want to select when destructuring an object. For example, let {color} = gemstone; will only select the color property from the gemstone object.

### **QUESTION 2 OF 2**

What do you expect to be returned from calling getArea()?

#### **const** circle = { radius: 10, color: 'orange', getArea: **function**() { **return** Math.PI \* **this**.radius \* **this**.radius; }, getCircumference: **function**() { **return** 2 \* Math.PI \* **this**.radius; } }; **let** {radius, getArea, getCircumference} = circle;

* 314.1592653589793
* NaN
* 62.83185307179586
* pie

## **7. Quiz: Destructuring Arrays**

### **Directions:**

Use array destructuring to pull out the three **colors** from the array of things and store them into the variables one, two, and three.

### **Your Code:**

/\*

\* Programming Quiz: Destructuring Arrays (1-3)

\* Use destructuring to initialize the variables `one`, `two`, and `three`

\* with the colors from the `things` array.

\*/

#### const things = ['red', 'basketball', 'paperclip', 'green', 'computer', 'earth', 'udacity', 'blue', 'dogs'];

#### const one = things;

#### const two = '';

#### const three = '';

#### const colors = `List of Colors

#### 1. ${one}

#### 2. ${two}

#### 3. ${three}`;

#### console.log(colors);

### 

### **Your Answer:**

#### const things = ['red', 'basketball', 'paperclip', 'green', 'computer', 'earth', 'udacity', 'blue', 'dogs'];

#### const [one] = things;

#### const [ , , , two] = things;

#### const [ , , , , , , , three] = things;

#### const colors = `List of Colors

#### 1. ${one}

#### 2. ${two}

#### 3. ${three}`;

#### console.log(colors);

## **8. Object literal shorthand**

A recurring trend in ES6 is to remove unnecessary repetition in your code. By removing unnecessary repetition, your code becomes easier to read and more concise. This trend continues with the introduction of new *shorthand* ways for initializing objects and adding methods to objects.

Let’s see what those look like.

### **Object literal shorthand**

You’ve probably written code where an object is being initialized using the same property names as the variable names being assigned to them.

But just in case you haven’t, here’s an example.

#### **let** type = 'quartz'; **let** color = 'rose'; **let** carat = 21.29; **const** gemstone = { type: type, color: color, carat: carat }; console.log(gemstone);

Prints: Object {type: "quartz", color: "rose", carat: 21.29}

Do you see the repetition? Doesn't type: type, color: color, and carat:carat seem redundant?

The good news is that you can remove those duplicate variables names from object properties *if* the properties have the same name as the variables being assigned to them.

Check it out!

NOTE: If object properties have the same name as the variables being assigned to them, then you can drop the duplicate variable names.

Speaking of shorthand, there’s also a shorthand way to add methods to objects.

To see how that looks, let’s start by adding a calculateWorth() method to our gemstone object. The calculateWorth() method will tell us how much our gemstone costs based on its type, color, and carat.

#### **let** type = 'quartz'; **let** color = 'rose'; **let** carat = 21.29; **const** gemstone = { type, color, carat, calculateWorth: **function**() { *// will calculate worth of gemstone based on type, color, and carat* } };

In this example, an anonymous function is being assigned to the property calculateWorth, but is the **function** keyword *really* needed? In ES6, it’s not!

### **Shorthand method names**

Since you only need to reference the gemstone’s calculateWorth property in order to call the function, having the function keyword is redundant, so it can be dropped.

#### **let** gemstone = { type, color, carat, calculateWorth() { ... } };

## **9. Video: Lesson 1 Checkup**

## **10. Video: Iteration**

## **11. Family of For Loops**

The **for...of loop** is the most recent addition to the family of for loops in JavaScript.

It combines the strengths of its siblings, the **for loop** and the **for...in loop**, to loop over any type of data that is **iterable**(meaning it follows the [iterable protocol](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) which we'll look at in lesson 3). By default, this includes the data types String, Array, Map, and Set—notably absent from this list is the Object data type (i.e. {}). Objects are not iterable, by default.

Before we look at the for...of loop, let’s first take a quick look at the other for loops to see where they have weaknesses.

### **The for loop**

The for loop is obviously the most common type of loop there is, so this should be a quick refresher.

#### **const** digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; **for** (**let** i = 0; i < digits.length; i++) { console.log(digits[i]); }

Prints:

0

1

2

3

4

5

6

7

8

9

Really the biggest downside of a for loop is having to keep track of **the counter** and **exit condition**.

In this example, we’re using the variable i as a counter to keep track of the loop and to access values in the array. We’re also using digits.length to determine the exit condition for the loop. If you just glance at this code, it can sometimes be confusing exactly what’s happening; especially for beginners.

While for loops certainly have an advantage when looping through arrays, some data is not structured like an array, so a for loop isn’t always an option.

### **The for...in loop**

The for...in loop improves upon the weaknesses of the for loop by eliminating the counting logic and exit condition.

#### **const** digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; **for** (**const** index **in** digits) { console.log(digits[index]); }

Prints:

0

1

2

3

4

5

6

7

8

9

But, you still have to deal with the issue of using an **index** to access the values of the array, and that stinks; it almost makes it more confusing than before.

Also, the for...in loop can get you into big trouble when you need to add an extra method to an array (or another object). Because for...in loops loop over all enumerable properties, this means if you add any additional properties to the array's prototype, then those properties will also appear in the loop.

#### Array.prototype.decimalfy = **function**() { **for** (**let** i = 0; i < **this**.length; i++) { **this**[i] = **this**[i].toFixed(2); } }; **const** digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; **for** (**const** index **in** digits) { console.log(digits[index]); }

Prints:

0

1

2

3

4

5

6

7

8

9

function() {

 for (let i = 0; i < this.length; i++) {

  this[i] = this[i].toFixed(2);

 }

}

Gross! This is why for...in loops are discouraged when looping over arrays.

NOTE: The forEach loop is another type of for loop in JavaScript. However, forEach() is actually an array method, so it can only be used exclusively with arrays. There is also no way to stop or break a forEach loop. If you need that type of behavior in your loop, you’ll have to use a basic for loop.

## **12. For...of loop**

The **for...of loop** is used to loop over any type of data that is *iterable*.

You write a **for...of** loop almost exactly like you would write a **for...in** loop, except you swap out in with of and you can drop the **index**.

#### **const** digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; **for** (**const** digit **of** digits) { console.log(digit); }

Prints:

0

1

2

3

4

5

6

7

8

9

This makes the for...of loop the most concise version of all the for loops.

TIP: It’s good practice to use plural names for objects that are collections of values. That way, when you loop over the collection, you can use the singular version of the name when referencing individual values in the collection. For example, for (const button of buttons) {...}.

But wait, there’s more! The for...of loop also has some additional benefits that fix the weaknesses of the for and for...in loops.

You can stop or break a for...of loop at anytime.

#### **const** digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; **for** (**const** digit **of** digits) { **if** (digit % 2 === 0) { **continue**; } console.log(digit); }

Prints:

1

3

5

7

9

And you don’t have to worry about adding new properties to objects. The for...of loop will only loop over the values in the object.

#### Array.prototype.decimalfy = **function**() { **for** (i = 0; i < **this**.length; i++) { **this**[i] = **this**[i].toFixed(2); } }; **const** digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]; **for** (**const** digit **of** digits) { console.log(digit); }

Prints:

0

1

2

3

4

5

6

7

8

9

## **13. Quiz: Writing a For...of Loop**

### **Directions:**

Write a for...of loop that:

* loops through each day in the days array
* capitalizes the first letter of the day
* and prints the day out to the console

Your code should log the following to the console:

Sunday

Monday

Tuesday

Wednesday

Thursday

Friday

Saturday

### **Your Code:**

/\*

\* Programming Quiz: Writing a For...of Loop (1-4)

\*/

#### const days = ['sunday', 'monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday'];

#### // your code goes here

#### for (const day of days) {

#### let full = day[0].toUpperCase() + day.slice(1);

#### console.log(full);

#### }

## **14. Spread... Operator**

The **spread operator**, written with three consecutive dots ( ... ), is new in ES6 and gives you the ability to expand, or *spread*, [iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators#Iterators) into multiple elements.

Let’s take a look at a few examples to see how it works.

#### **const** books = ["Don Quixote", "The Hobbit", "Alice in Wonderland", "Tale of Two Cities"]; console.log(...books);

Prints: Don Quixote The Hobbit Alice in Wonderland Tale of Two Cities

#### **const** primes = **new** Set([2, 3, 5, 7, 11, 13, 17, 19, 23, 29]); console.log(...primes);

Prints: 2 3 5 7 11 13 17 19 23 29

If you look at the output from the examples, notice that both the array and set have been expanded into their individual elements. So how is this useful?

NOTE: Sets are a new built-in object in ES6 that we’ll cover in more detail in a future lesson.

### 

### **Combining arrays with concat**

One example of when the spread operator can be useful is when combining arrays.

If you’ve ever needed to combine multiple arrays, prior to the spread operator, you were forced to use the Array’s concat() method.

#### **const** fruits = ["apples", "bananas", "pears"]; **const** vegetables = ["corn", "potatoes", "carrots"]; **const** produce = fruits.concat(vegetables); console.log(produce);

Prints: ["apples", "bananas", "pears", "corn", "potatoes", "carrots"]

This isn’t terrible, but wouldn’t it be nice if there was a shorthand way to write this code?

For example, something like…

NOTE: If you're following along by copy/pasting the code, then you've already declared the produce variable with the const keyword. The following code will try to re-declare and re-assign the variable, so depending on how you're running the code, it might throw an error. Remember that variables declared with const cannot be redeclared or reassigned in the same scope.

#### **const** produce = [fruits, vegetables]; console.log(produce);

Prints: [Array[3], Array[3]]

Unfortunately, that doesn’t work.

Instead of combining both arrays, this code actually adds the fruits array at the first index and the vegetables array at the second index of the produce array.

How about trying the spread operator?

/\*

\* Instructions: Use the spread operator to combine the `fruits` and `vegetables` arrays into the `produce` array.

\*/

#### const fruits = ["apples", "bananas", "pears"];

#### const vegetables = ["corn", "potatoes", "carrots"];

#### const produce = [];

#### console.log(produce);

## **15. ...Rest Parameter**

If you can use the spread operator to *spread* an array into multiple elements, then certainly there should be a way to bundle multiple elements back into an array, right?

In fact, there is! It’s called the *rest parameter*, and it’s another new addition in ES6.

### **Rest parameter**

The **rest parameter**, also written with three consecutive dots ( ... ), allows you to represent an indefinite number of elements as an array. This can be helpful in a couple of different situations.

One situation is when assigning the values of an array to variables. For example,

#### **const** order = [20.17, 18.67, 1.50, "cheese", "eggs", "milk", "bread"]; **const** [total, subtotal, tax, ...items] = order; console.log(total, subtotal, tax, items);

Prints: 20.17 18.67 1.5 ["cheese", "eggs", "milk", "bread"]

This code takes the values of the order array and assigns them to individual variables using [destructuring](https://classroom.udacity.com/nanodegrees/nd016/parts/11a45d59-bb81-44a9-be76-042c99e5f051/modules/cbf6deb8-d2cc-4757-b3a9-a1c58a4acd82/lessons/42383e89-ac6a-491a-b7d0-198851287bbe/concepts/7c7be588-31e9-4c62-9dad-ecf23f943b19#destructuring-values-from-an-array). total, subtotal, and tax are assigned the first three values in the array, however, items is where you want to pay the most attention.

By using the rest parameter, items is assigned the *rest* of the values in the array (as an array).

NOTE: You can think of the rest parameter like the opposite of the spread operator; if the spread operator is like unboxing all of the contents of a package, then the rest parameter is like taking all the contents and putting them back into the package.

### 

### **Variadic functions**

Another use case for the rest parameter is when you’re working with variadic functions. **Variadic functions** are functions that take an indefinite number of arguments.

For example, let’s say we have a function called sum() which calculates the sum of an indefinite amount of numbers. How might the sum() function be called during execution?

#### sum(1, 2); sum(10, 36, 7, 84, 90, 110); sum(-23, 3000, 575000);

There’s literally an endless number of ways the sum() function could be called. Regardless of the amount of numbers passed to the function, it should always return the total sum of the numbers.

### **Using the arguments object**

In previous versions of JavaScript, this type of function would be handled using the [arguments object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments). The **arguments object** is an array-like object that is available as a local variable inside all functions. It contains a value for each argument being passed to the function starting at 0 for the first argument, 1 for the second argument, and so on.

If we look at the implementation of our sum() function, then you’ll see how the arguments object could be used to handle the variable amount of numbers being passed to it.

#### **function** **sum**() { **let** total = 0; **for**(**const** argument **of** arguments) { total += argument; } **return** total; }

Now this works fine, but it does have its issues:

1. If you look at the definition for the sum() function, it doesn’t have any parameters.
   * This is misleading because we know the sum() function can handle an indefinite amount of arguments.
2. It can be hard to understand.
   * If you’ve never used the arguments object before, then you would most likely look at this code and wonder where the arguments object is even coming from. Did it appear out of thin air? It certainly looks that way.

### **Using the rest parameter**

Fortunately, with the addition of the rest parameter, you can rewrite the sum() function to read more clearly.

#### **function** **sum**(...nums) { **let** total = 0; **for**(**const** num **of** nums) { total += num; } **return** total; }

This version of the sum() function is both **more concise** and is **easier to read**. Also, notice the for...in loop has been replaced with the new [for...of loop](https://classroom.udacity.com/nanodegrees/nd016/parts/11a45d59-bb81-44a9-be76-042c99e5f051/modules/cbf6deb8-d2cc-4757-b3a9-a1c58a4acd82/lessons/42383e89-ac6a-491a-b7d0-198851287bbe/concepts/f1955923-744a-4906-8f64-1ddcb34c6da2#).

## **16. Quiz: Using the Rest Parameter**

### **Directions:**

Use the rest parameter to create an average() function that calculates the average of an *unlimited* amount of numbers.

TIP: Make sure to test your code with different values. For example,

average(2, 6) should return 4

average(2, 3, 3, 5, 7, 10) should return 5

average(7, 1432, 12, 13, 100) should return 312.8

average() should return 0

### **Your Code:**

|  |
| --- |
| *// your code goes here* function average() {   } console.log(average(2, 6)); console.log(average(2, 3, 3, 5, 7, 10)); console.log(average(7, 1432, 12, 13, 100)); console.log(average()); *//* function average(...nums) {  let total = 0;  let i = 0;  for (const num of nums) {  total += num;  i += 1;  }  if (total === 0 || NaN) {  return 0;  }  return total / i; } |

## **17. Video: Lesson 6 Summary**