JavaScript for App Development

By Mark Lassoff, Founder Framework Television

Section 9: ES6 Functions

After completing this section you will:

- ☐ Be able to use default parameters
- ☐ Understand how to use rest parameters
- ☐ Create anonymous functions
- ☐ Write functions using the function constructor notation
- ☐ Use and Apply the lambda function notation

Watch This: JavaScript for Application Development Section 009 Video

As always your course videos are available on YouTube, Roku, and other locations. However, only those officially enrolled have access to this course guide, can submit assignments, work with the instructor, and get this guide.

Watch this section video at **https://www.youtube.com/watch?v=BI4cGg--Tt4**

Introduction

We previously looked at functions in Javascript, but now we are going to examine them from an ES6 perspective.

ES6 added a number of features to functions that allow us to do some pretty exciting things. The justification for the changes was to make JavaScript more of a functional programming language (which

is a language based mostly on mathematical functions). Even if you don't care about functional programming, I think you're going to enjoy these new features because they make functions a lot more flexible.

> It's important to keep in mind that you'll often work with code that you did not write. It's important to be familiar with all forms of function notation so you recognize and understand the work of other developers. Coding is often a team sport!

Traditional Functions Review

Before we get into the changes that came with ES6, let's review the traditional function notation in Javascript. Let's start by coding an output `div` and add `script` tags.

```
<div id="output"></div>
<script>

</script>
```

We are now ready to start coding our function. We'll begin by creating the function signature which consists of the function name and the keyword `function`. We'll name this function "greetings".

```
<div id="output"></div>
<script>
  function greetings(){

  }
</script>
```

Next, let's create a message variable and assign it the string "Greetings!". After initializing the variable, the function writes the value to our output `div` using the `innerHTML` property by assigning the message variable to it.

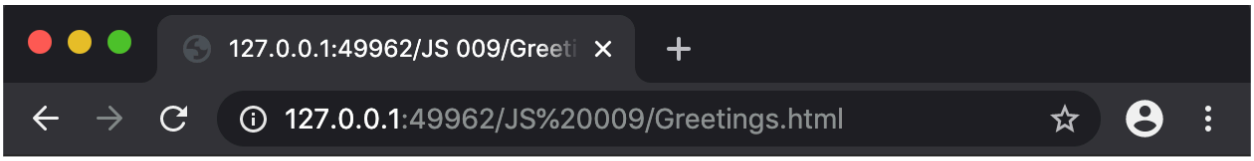
```
<div id="output"></div>
  <script>
    function greetings(){
      let message = "<h1>Greetings!</h1>";
      document.getElementById("output").innerHTML =
message;
    }
  </script
```

Of course, this function is not going to execute until it's called. Remember that you should always the function itself and the separate function call that executes the function. Essentially, the function is memorized by the Javascript processor and is executed only when called.

Let's add the function call.

```
<div id="output"></div>
  <script>
    function greetings(){
      let message = "<h1>Greetings!</h1>";
      document.getElementById("output").innerHTML =
message;
    }
    // Function call
    greetings();
  </script>
```

As you can see the output in HTML displays the value of the message variable, "Greetings!".



Greetings!

The previous function is an example of a simple Javascript function. However, many functions are parametrized which means they receive an argument from the function call that the function processes.

Parameterized Functions

We'll name the parameter passed to the function `userName`. In the function, we'll concatenate `userName` with additional text and assign the string to the `message` variable. This way, the message output by the functions is "Greetings [value of userName]!".

This time, the function call has to send the parameter. The value sent is placed between the parentheses in the function call.

In the code below, the parameter "Mark" is passed to the function, assigned to userName and then used in the `message` variable.

```
<div id="output"></div>
  <script>
    function greetings(userName){
      let message = "<h1>Greetings " + userName + "!</h1>";
      document.getElementById("output").innerHTML =
message;
    }
    // Function call
    greetings("Mark");
  </script>
```

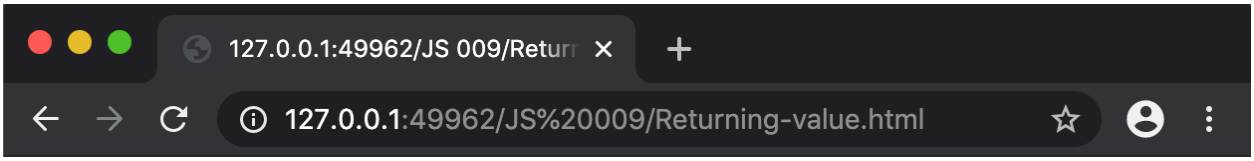
Returning a Value

In addition to some functions being parametrized many functions return a value. when a function returns a value, we need something to catch that return.

Let's declare a variable `myMessage` and initialize it with the function call to `greetings()`. The resulting message, created by the function, will then be assigned to `myMessage`.

```
<div id="output"></div>
<script>
   function greetings(userName){
      let message = "<h1>Greetings " + userName + "!</h1>";
      return message;
   }
   // Function call
   let myMessage = greetings("Mark");
   document.getElementById("output").innerHTML =
myMessage;
</script>
```

Output on the web page should look like the screenshot below.



Greetings Mark!

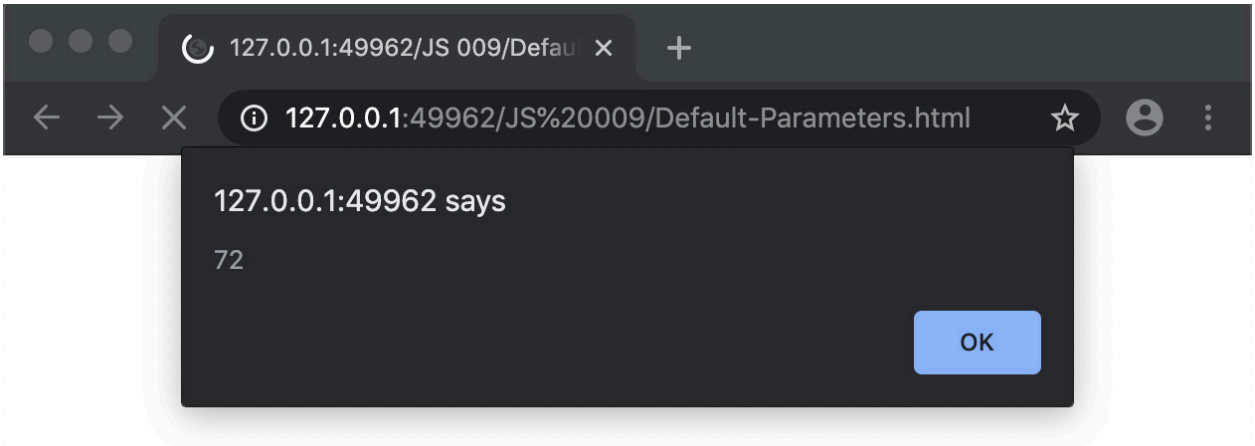
Default Parameters

Now that we've reviewed the "traditional" Javascript function, let's start examining what's new in ES6. First up is `default parameters`. Default parameters allow a parameter to be assigned by the function itself if no parameter is sent by the function call.

Let's code a new function called `multiply`. The function will require two parameters: `x` and `y`. It will return the product of both parameters. We'll embed the function in an `alert()` so we can review the result. Initially we'll send the arguments nine and eight to the function.

```
<script>
   function multiply(x,y) {
      return x * y;
   }

   alert(multiply(9,8));
</script>
```

The result of the code should appear similar to the screenshot below:



In the `alert` function the inside of the parenthesis is processed first. In this case, the `multiply` function is called. We pass the values `9` and `8` which are assigned to `x` and `y` respectively. Next, we multiply them together and return the product of the two numbers which is then displayed in our alert.

However, if we don't pass parameter values inside the function call, the returned value will be `undefined` which is, obviously, not desirable in most cases. This is where default parameters come in!

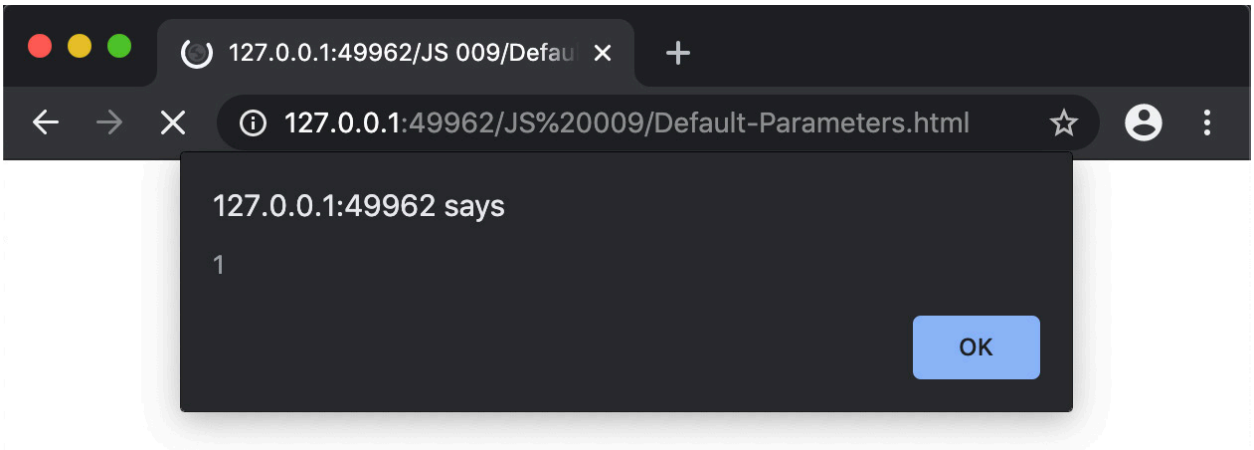
If we add default values, then this function will produce a result even if no values are passed to it. Let's create default values of `1` by assigning the value inside the function signature.

```
<script>
  function multiply(x=1,y=1) {
    return x * y;
  }

  alert(multiply(9,8));
</script>
```

If we run this code as is, we'll still get `72` as a result, because the parameters `9` and `8` are being passed to the function. But if we take the parameters out of the multiply function call and run the code in the browser:

```
<script>
  function multiply(x=1,y=1) {
    return x * y;
  }

  alert(multiply());
</script>
```

This is the result:



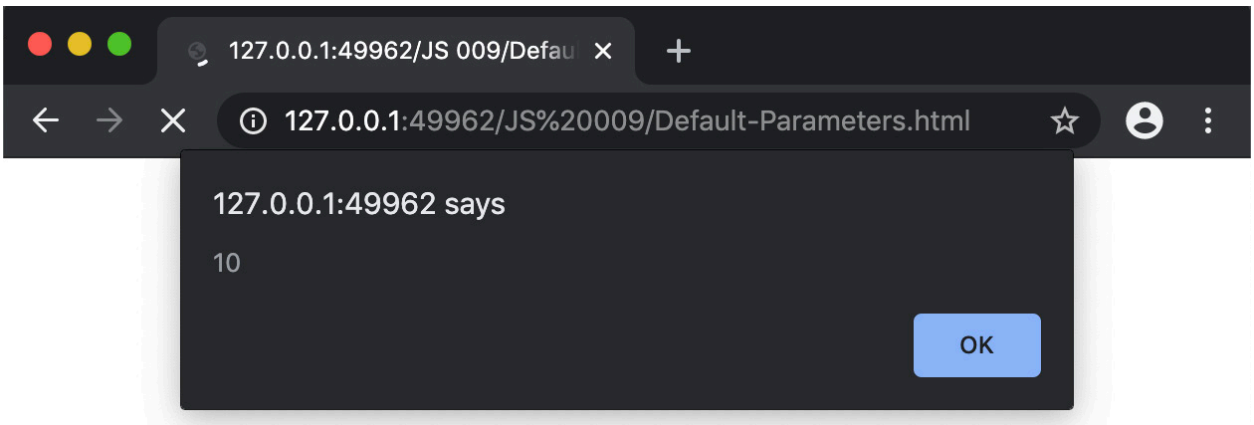
Because no value was passed to the parameters, the function assigned the value `1` to each of `x` and `y` and returned the product of the two values which is `1`.

But what if we passed only one value to the `multiply` function in the code above?

Let's use `10` for our example.

```
alert(multiply(10));
```

The result is the following dialog box:

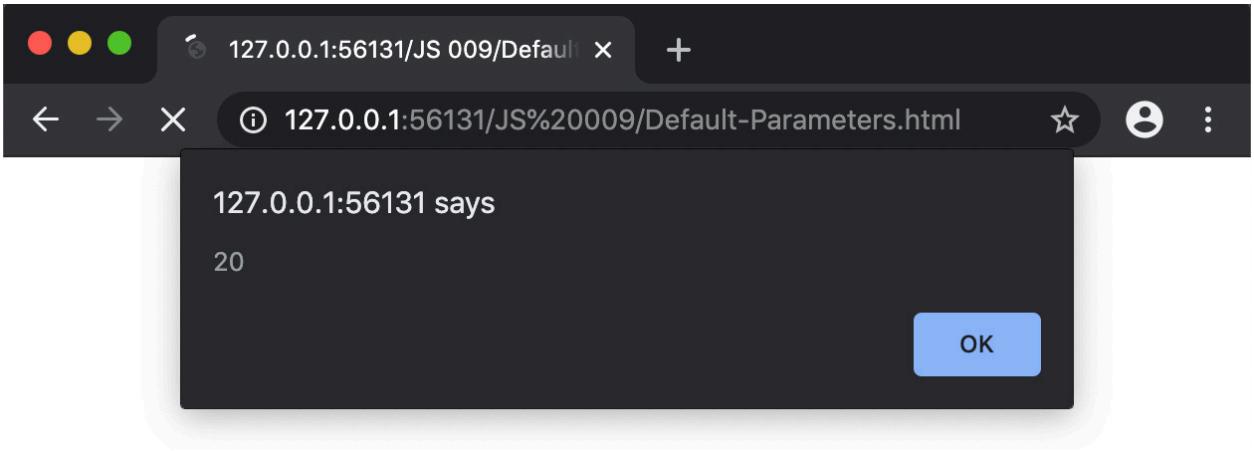


With only one parameter passed, `10` was assigned to the first parameter `x` and since `y` wasn't given a parameter, it was assigned the value of `1`. The function then returns the product of `10` and `1`.

For a bit of proof, we'll change the default value of y to `2`.

```
<script>
    function multiply(x=1,y=2) {
      return x * y;
    }

    alert(multiply(10));
  </script>
```

And as expected, the result is now `20`.



This can be very useful if you use a function frequently and you only need to pass parameters in certain circumstances. Without the ability to assign default parameters we'd have to write several separate versions of the function. With default parameters we are able to write a single version of the function that is applicable in more cases.

Rest Parameters

In addition to default parameters, ES6 has brought us another new feature called `rest parameters`. Rest parameters let us pass any number of values to a function. This can be useful when we want a function to work on an unknown number of parameters.

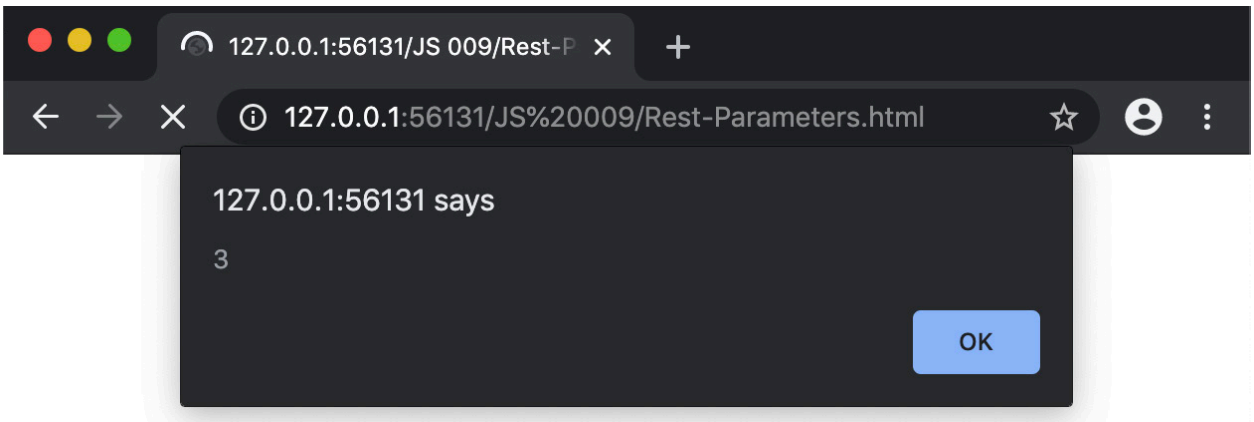
For our example, we'll write a function called `output`. We'll call our rest parameter `params` and write it with `...` preceding the name.

We'll use an `alert` and pass it `params.length` which will output the number of parameters we passed into the function.

Finally we'll code the function call with three random parameters.

```
<script>
   function output(...params) {
      alert(params.length);
   }

   output("x", "y", "z");
</script>
```

The result is an alert that shows `3`.



Let's change the parameters we pass into the function call as follows:

```
output(10.2, 8.5, 7.1, 6.5, 1.1);
```

We'll receive an alert with 5. The type of the parameters doesn't matter for now as long as they are all the same type. Right now, our function is essentially counting the parameters in the function call.

Normally, we might loop through the parameters that are passed to the function.

```
<script>
   function output(...params) {
      alert(params.length);
      for(i=0; i < params.length; i++){
         alert(params[i]);
      }
   }

   output("x", "y", "z");
</script>
```

This function will output an alert with the value of each parameter we pass into the function call, one at a time. You can see how this would be useful when you don't know how many parameters your function is going to receive. For example, you could create a function that averages the grades of an entire class and the function would work with any sized group. Using rest parameters, you can pass in any number of parameters to the function and it can average them.

Do This: Discuss on Slack

Visit the course Slack discussion and cite examples of how you might use Rest parameters. When might you have a situation in which you don't know how many parameters would be passed to a function?

Anonymous Functions

Anonymous functions are not entirely new but ES6 takes the concept more robust. Let's examine the structure of an anonymous function:

```
<script>
    let greetings = function() { return "Greetings Stranger!" }

    alert(greetings());
  </script>
```

Here we called the function by its variable name but the function itself is anonymous. Anonymous functions can also receive parameters:

```
<script>
    let greetings = function(name) { return "Greetings " + name }

    alert(greetings("Brett"));
  </script>
```

The justification for assigning your anonymous functions to a variable is mainly traceability. Previously, an anonymous function was anonymous in the call stack—making it difficult to debug. Now the anonymous function is associated with the name and traceable by that name when debugging.

Function Constructor

The function constructor provides yet another way of creating functions. Let's take the example of our `greetings` function and add the function constructor concept.

```
<script>
    let greetings = new Function("name", "return ('Greetings ' 
    + name)");
    alert(greetings("Valerie");
 </script>
```

The version of the `greetings` function above is exactly the same functionally as the `greetings` function used previously. Here the `new` keywod declares a new function, `name` is the function's parameter and we'll return the string `Greetings + name`. It is essentially a very compact way to create a function.

> In this example we used the single quotes to surround the string `Greetings` because it's already inside double quotes.

We then are able to call the function as usual by using the variable name `greetings`.

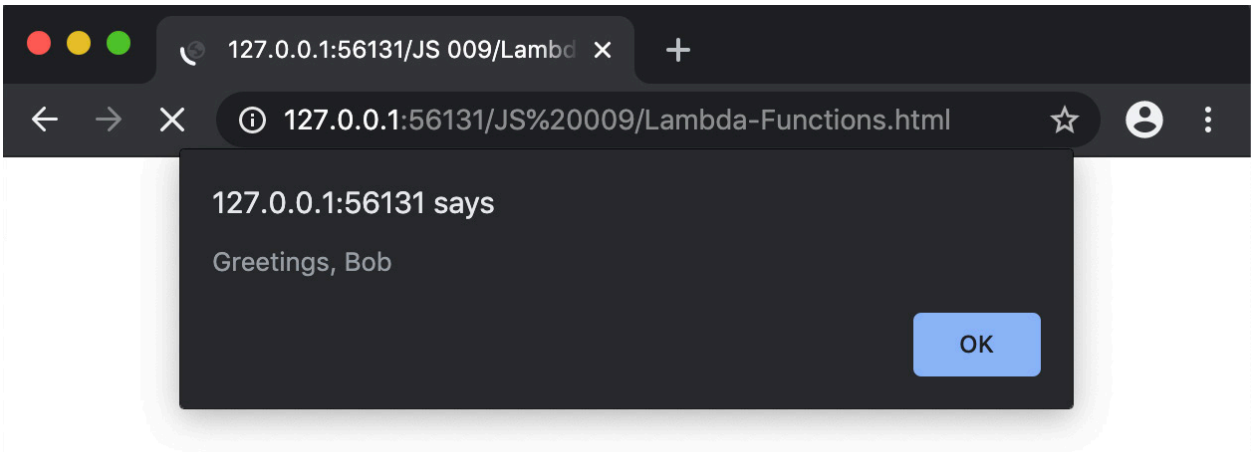
Lambda Functions

Lambda functions are functions that use the arrow notation—another option for creating functions. The lambda function has three parts: the parameters, the fat arrow (which is the lambda notation) and the statements.

> You will also here Lambda functions referred to as arrow functions. This is, of course, due to the arrow notation (=>) used inside the function itself.

```
<script>
    let greetings = (name) => (“Greetings, “ + name);
    alert(greetings(“Bob”));
  </script>
```

The result is as expected.



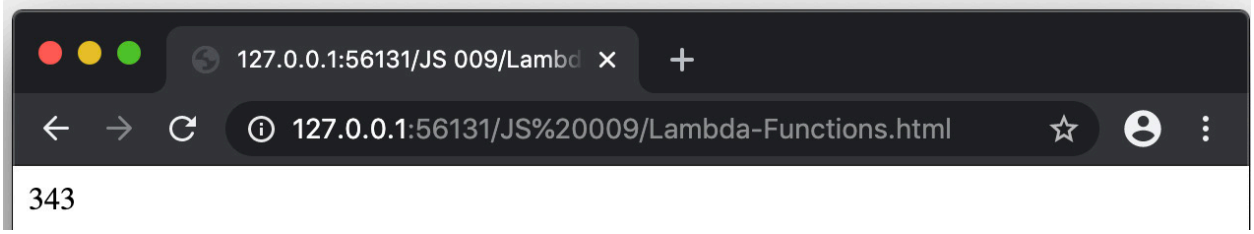
In the above example, `name` is the parameter, the fat arrow `=>` essentially replaces all the function syntax and `“Greetings, “ + name` is the returned string value.

Let’s examine a second example:

```
<div id=”output”></div>
  <script>
    let cube = (myValue) => (myValue * myValue * myValue);
    document.getElementById(“output”).innerHTML = cube(7);
  </script>
```

Here is the result displayed in the browser:

```
<div id="output"></div>
<script>
    let cube = (myValue) => (myValue * myValue * myValue);
    document.getElementById("output").innerHTML = cube(7);
</script>
```



The function cubes the parameter by multiplying it by itself three times. The correct result, 343, is returned and displayed.

Lambda functions are typically used if a function is short and doesn't require multiple lines. Lambda functions are designed to be easy to see and understand at a glance.

Do This: Practice New Function Notation

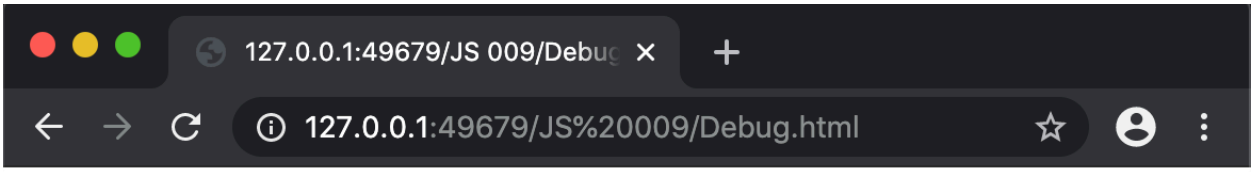
For practice, start a new document and save it as `small-talk.html`

Using the code above as a guide, write three separate functions. For the first use the notation for anonymous functions. For the second function, use a function constructor and, for the final function, use lambda function notation.

Each function should output the string "Hello [name]" where a name is the parameter passed to the function.

Debug This:

There are errors in this code preventing it from displaying our welcome message correctly. Fix the errors, so the information displays correctly in your browser like this:



Welcome,We're very glad to have you in this course. ,45,

Feel free to ask us any question!

Here's the code to debug

```
<div id="output"></div>
  <script>
    let addParagraphs = function(params) {
      for(i=1; i < params.length; i++){
        document.getElementById("output").innerHTML +=
        params;
      }
    };
    addParagraphs(
      "Welcome",
      "We're very glad to have you in this course. ",
      45,
      "<p>Feel free to ask us any question!</p>"
      );
  </script>
```

Submit This: My To-Dos

Create an HTML5 document from scratch that is correctly formed and coded that shows 5-10 items on your to-do list. There should only be a `div` element in your properly formed HTML. You'll add the output to the `<div>` using Javascript.

Store your to-do items in a single array of string values called `toDoList`.

The output should include an `h1` that contains the text `TO-DO:` which is written by using a function called `todo` . Add your individual to-do items to the `div` from a single function called `addTodos`.

Each to-do item should be inside an individual `p` element in the HTML.