Back to Week 1

JS | Functions Advanced

LESSON

Learning goals

After this lesson, you will be able to:

- understand what are and how to use function expressions,
- understand that functions can be passed to other functions as arguments,
- understand JavaScript async behavior and need for callbacks,
- understand and use anonymous functions,
- understand how to use arrow functions,
- use the `arguments` object.

Introduction

As already mentioned in the *Functions Intro* lesson, we can declare functions in 3 different ways: as *function declaration (aka statements)*, as *function expressions* and as an *instance of the global Function object constructor*.

We learned the proper syntax for function declarations would be the following:

```
function calcSum(x, y) {
  return x + y;
}

calcSum(12, 23); // => 35
```

Explain this code

where:

- `function` is a reserved keyword and serves to show that a particular block of code to be executed will be wrapped under a certain name;
- the name we gave to this function is `calcSum`;
- `(x, y)` are parameters to be passed in the function and to be used in between `{...}` which represents the function body; these are placeholders which will be replaced by some real values at the moment when this function gets executed;
- the `return` statement shows what will be the returned value from the function. Also, we said that the `return` is the last piece of code to be executed within the function and that if by accident, we add some code after the `return` it won't be executed.
- `calcSum(12, 23)` is the function call, when the function gets executed with some real values passed instead of *x* and *y*. `12 and 23` are function arguments.

Function expression

Now when we know how to declare and invoke named functions (function statements), it is going to be very easy to

explain how to do the same with function expressions. Before we show you how to work with function expression, we need to mention briefly that functions are treated as so-called `first-class objects` in JavaScript. It must be very confusing to see that - functions are objects?!? But by the end of this course, you will know how and why "everything in JavaScript is an object", including functions. If you are curious to know more about this topic, check the extra resources section later. For now, keep going.

So if functions are treated as objects, that means they can be assigned to variables (meaning, they can be stored in variables). And that brings us to *function expressions*.

```
// function declaration (statement)
function welcomeMessage(message) {
  return message;
}

// function assigned to a variable
const greeting = welcomeMessage('So nice to have you here! Welcome!');

console.log(greeting); // So nice to have you here! Welcome!
```

Explain this code

Why would we want to do this? For example, we could pass the variable `greeting` to some other function as an argument.

The above is a *"schooly way"* to explain function expressions. Here is how you could and you will do it yourself now when you know that functions can be stored in variables:

```
// function expression - is a function without name stored in a variable
const welcomeMessage = function (message) {
  console.log(message);
};

welcomeMessage('So nice to have you here! Welcome!');
// => So nice to have you here! Welcome!
```

Explain this code

Function declaration vs. function expression

A function declaration is a named function and can be stored in a variable if needed (example with `greeting`).
A function expression is an un-named (or so-called anonymous) function that is stored in the variable.
Both can be reused throughout the code.

Function expressions and declarations do the same thing pretty much. In both examples, to call them you do the same (`welcomeMessage()`), and then they execute whatever code is inside their code block (or how we call it - the function body) `{..}` .

So what's the catch, why we have to know both?

Well, there is a difference and it is not related to what they do, but how they are executed. The order in which code gets executed is what determines the difference between these two. Check this:

```
// function declaration (statement)

checkFuncDeclaration(); // => Func declaration CAN be invoked before it's defined.

function checkFuncDeclaration() {
  console.log('Func declaration CAN be invoked before it is defined.');
}
```

Explain this code

```
// function expression

checkFuncExpression(); // => ReferenceError: checkFuncExpression is not defined

const checkFuncExpression = function () {
  console.log('Func expression CAN NOT be invoked before it is defined.');
};
```

Explain this code

To understand why and how this happened, we will introduce the concept of hoisting.

> Hoisting is a concept related to the way how programming language gets interpreted by a machine that executes it. This concept is related to any type of variables and data types, and it is not correlated to the functions only.

Hoisting

(*Will will be covering the topic of hoisting much deeper soon, this is just the intro*).

When JavaScript code gets executed, it happens from top to bottom and from left to the right side. Meaning whatever we wrote on line 1 in our code snippet will be executed before anything that comes on line 2, and whatever is on line 2 will be executed before line 3 and so on. This implies, to use a variable, you need to declare it first. However, that is not the case when it comes to function declarations. Function declarations get hoisted (lift) to the top of the code before any other code gets executed.
How this process of lifting function declarations happens? Before any code runs, your JavaScript code needs to be interpreted (translated into browser understandable code). In this process of interpretation, function declarations get hoisted on the top of the code. That is why we were able to call the function before we even declared it. So when it comes to the execution phase, function declarations are already interpreted and loaded, which means they can be used although they will be defined at some point later on.

As you might imagine, the same process doesn't happen with the function expressions. They get interpreted in the exact line where they are written so they can be used only after they are defined first.

Function declarations or function expressions - which one is better to use?

Is there a strict rule which one to use, or which one is much better? No. What matters is **consistency**. Whichever you decide to use, stick with it. When the time comes, and you join a dev team, respect the set rules - if they use declarations, you shall do the same, and the same stands for expressions. This all brings us to the same again - just be consistent.

One thing does stand out in favor of function expressions - since you are forced to define a function expression to use it, your code will have much leaner and better structure. This gives you better control over the flow of the app. You can't use a function unless it is already defined, so there is always a certain logic in arranging code statements in a particular sequence.

Things can get a bit messier with declarations since you can call them from wherever in your code. This won't force you to think structurally and be consistent.

However, for now, this shouldn't be a matter of your worry. Understanding the basics and knowing how to write and use functions should be in your main focus.

Callbacks

As we already mentioned, functions are first-class objects and, as such, can be stored in variables. One more characteristic that goes hand in hand with being a first-class object - functions can be passed as parameters (arguments) to other functions. In that case, we are talking about `callbacks`.

Let's now focus on understanding why and when we would want to apply it.

> Callback functions are used to make sure a particular code doesn't execute until another code has already finished execution.

Now, there can be a question: if JavaScript code gets executed line by line from top to bottom, how is then possible that some function can execute before some other if they are in the correct order in the file?

Let's create an example:

```
function eatDinner() {
  console.log('Eating the dinner!');
}

function eatDessert() {
  console.log('Eating the dessert!');
}

eatDinner(); // <== Eating the dinner!
eatDessert(); // <== Eating the dessert!
```

Explain this code

Great, we know how this works - our computers execute line by line so they will first execute `eatDinner()` and then `eatDessert()` because this is the order in which we called these functions.

Delayed execution of one of the functions

But, you know, sometimes the dinner prep can take a reaallllly long time and then if we don't handle the situation properly, we could be in the situation to get the dessert before the dinner is even served.
We will simulate this situation by just delaying the execution od `eatDinner()` function for a couple of seconds using the `setTimeout()` JavaScript method.

Let’s update function 1:

```
function eatDinner() {
  setTimeout(function () {
    console.log('Eating the dinner!');
  }, 1000);
}

function eatDessert() {
  console.log('Eating the dessert!');
}

eatDinner();
eatDessert();

// the console:
// Eating the dessert!
// Eating the dinner!
```

Explain this code

As we can see, we called first `eatDinner()` function and then `eatDessert()` but in the console, the first output is from the second function. It is not that JavaScript didn’t execute our functions in the order we wanted, this still happened. However, JavaScript didn’t wait for the response from the first function since it took a bit longer, so instead, it moved to execute the second function and then, after 1000 milliseconds (that’s the delay we passed to the *setTimeout*), it executed the first function.

When and why any function would be postponed/delayed in the real world?

Well, on many occasions. Here is one example: you could be waiting for some data from your database or API to display it to the users. So imagine that it takes a bit longer to get that data. The idea was to get the data and then display it to the user. You didn’t expect that the data could take a bit, so your code doesn’t have any mechanism to ensure that users won’t get just a blank page instead of a page with the data from the database. The worst-case scenario is that some error renders on the page. This would be an awful user experience and something we can’t allow to happen. What we have to do is to ensure that the function that renders the page to users doesn’t get executed before the function that gets the data doesn’t really get that data that we want to display to users.

And that is when the **callbacks** get in the game. By passing one function as an argument to some other function, we are actually making sure that the first function executes and returns some value so that value can be used in the second function.

> Callbacks are the way to make sure that some piece of code doesn’t execute before some other code hasn’t finished executing.

Now let’s use the knowledge we just got and make sure we get that dinner before the dessert comes.

This is example of how callbacks work:

```
function eatDinner(callback) {
  // the word "callback" is just placeholder
  // you can use any other word
  console.log('Eating the dinner!');
  callback();
}

function eatDessert() {
  console.log('Eating the dessert!');
}

eatDinner(eatDessert); // <== make sure when invoking callback func NOT tu use ()

// Eating the dinner!
// Eating the dessert!
```

Explain this code

When invoking a function with a callback, make sure not to use `()` when it comes to the callback function.

Anonymous functions

You already have seen these in the examples above, but let's give them some attention since they deserve it. 😉

- **An anonymous function is a function without a name.**
- **An anonymous function usually is not available to be used after its initial creation.** The reason to create a function without any name is that it will be used just in that very moment and never again in your code, so really no need to name them.

Let's take a look at the following function expression:

```
const calcSum = function(a, b){
  return a + b;
}
```

anonymous function

We stored the above defined function into the variable so we can reuse it. But sometimes we really don't need to reuse functions. Especially when we pass functions as arguments to other functions. Let's see how that works.

Anonymous functions as other function's arguments

Anonymous functions can be used as an *arguments passed to another function*.
Here are some examples:

Example 1:

```
function printName(name, anonFunc) {
  anonFunc(name);
}

printName('sandra', function (name) {
  console.log(name[0].toUpperCase() + name.slice(1));
});

// => Sandra
```

Explain this code

```
function getLength(str, anonFunc) {
  anonFunc(str);
}

getLength('aleksandra', function (str) {
  console.log(`${str} has ${str.length} letters.`);
});

// => aleksandra has 10 letters.

getLength('nick', function (str) {
  console.log(`${str} has ${str.length} letters.`);
});
// => nick has 4 letters.
```

Explain this code

Very often, we will use anonymous functions as arguments in the `setTimeout()` JavaScript native method:

```
setTimeout(function () {
  console.log('I am anonymous function and I will execute after 1 second.');
}, 1000);

// ... nothing happens for 1 second
// => I am anonymous function and I will execute after 1 second.
```

Explain this code

Since anonymous functions are not available to be used later, if, for some reason, we have a need to use them, we should give them a proper function declaration or function expression structure. Then we will be able to reference them and use them whenever we need to do so.

```
function notifyUser() {
  console.log('I am anonymous function and I will execute after 1 second.');
}

setTimeout(notifyUser, 1000);
```

Explain this code

Arrow functions

According to the official **MDN arrow function docs**, an arrow function expression is a syntactical alternative to a regular function expression.

This update was introduced with ES6 and its main goal is to introduce simpler syntax and help with some earlier challenges related to the scope with nested functions/methods (we will learn much more about scope in one of the next lessons, but in this one we will just demo how it really looks).

Let's see how arrow functions look like:

```
// function expression syntax
const greeting = function (name) {
  console.log(`Hello, ${name}!`);
};

// arrow function syntax
const greeting = name => {
  return name;
};
```

Explain this code

As we can see, the keyword `function` is omitted, the parameter doesn't have braces around (although this changes when we have more than one parameter), and there's `=>` arrow between the parameter and the body (`{...}`) of the function.

But this function can, even more, be shortened - since we return only one expression (there is only one line of code in the body), so we can **omit the braces and skip the return** since it's implicit:

```
const greeting = name => `Hello, ${name}!`;

greeting('Ana'); // => Hello, Ana!
```

Explain this code

So much cleaner and shorter!
In case **there are no parameters passed then empty parentheses are mandatory**:

```
const greeting = () => console.log('Hello there!');
```

Explain this code

As a conclusion:

If the right side is only a one-line expression, we can omit the curly braces and the `return` statement is omitted as well (the `return` statement is implied). However, if we need to write multiline statements in the function, then we can do it using the curly braces `{...}` and in that case, we have to use the `return` statement as well.

The `this` keyword and the matter of a scope

This concept will be explained in one of the next lessons much deeper, but for now, let's see what we can do to simplify it, so you understand the basics.

Let’s take the next example - a simple object with two 3 properties, where one of them is a method:

```
const user = {
  name: 'ana',
  age: 29,
  getOlder: function () {
    console.log(this);
    console.log(this.name);
  }
};

user.getOlder();
// => { name: 'ana', age: 29, getOlder: [Function: getOlder] }
// => ana
```

Explain this code

As shown, the keyword `this` refers to the object (user) itself. So if we would like to get Ana older for a year, we could update our code as follows:

```
const user = {
  name: 'ana',
  age: 29,
  getOlder: function () {
    this.age += 1;
    console.log(this.age);
  }
};

user.getOlder();
// => 30
```

Explain this code

Cool! Now, let’s add the `setInterval()` JS native method to make Ana “older” every second for one year:

```
const user = {
  name: 'ana',
  age: 29,
  getOlder: function () {
    setInterval(function () {
      this.age += 1;
      console.log(this.age);
    }, 1000);
  }
};

user.getOlder();
// => NaN
// => NaN
// => NaN
// ...
```

Explain this code

Hm... what changed??? Let’s output the `this` keyword again:

```
const user = {
  name: 'ana',
  age: 29,
  getOlder: function () {
    setInterval(function () {
      console.log(this);
    }, 1000);
  }
};

user.getOlder();
// Timeout {
//   _idleTimeout: 1000,
//   _idlePrev: null,
//   _idleNext: null,
//   _idleStart: 145,
//   _onTimeout: [Function],
//   _timerArgs: undefined,
//   _repeat: 1000,
//   _destroyed: false,
//   [Symbol(refed)]: true,
//   [Symbol(asyncId)]: 5,
//   [Symbol(triggerId)]: 1
// }
```


It seems that we “lost” access to the properties of the object “user”, since `this` keyword is now referring to the `setInterval()` method. Simplified, `this` inside `setInterval()` refers to the `setInterval()`. So what can we do to get access to the properties we need? The simplest way is to use the `arrow` function syntax since it *binds* the scope to the object itself. (The *bind* will also be a bit more clear later. In simple words, thanks to `=>`, we now again have access to the “user” properties.)

```
const user = {
  name: 'ana',
  age: 29,
  getOlder: function () {
    setInterval(() => {
      this.age += 1;
      console.log(this.age);
    }, 1000);
  }
};

user.getOlder();
// 30
// 31
// 32
// 33
```


This is one of the most important features that ES6 brought, and you will be using it heavily.

The `arguments` object

Inside the body of a function, you can access an object called `arguments`. This object represents all the arguments passed to a function. The specific thing related to it is that this is an `array-like object`. According to the MDN,

"array-like" means that arguments have a length property and properties indexed from zero, but it doesn't have Array's built-in methods like forEach() (and others which we will cover soon).

```
function printSomething() {
  console.log(arguments);
}

printSomething('hi');

// [Arguments] { '0': 'hi' }
```

Explain this code

We can use the square bracket `[]` to access the arguments: `arguments[0]` returns the first argument, `arguments[1]` returns the second one, and so on. We can also use the `length` property of the arguments object to determine the number of arguments.

```
function printSomething() {
  console.log('arguments length: ', arguments.length);
  console.log('all: ', arguments);
  console.log('first arg: ', arguments[0]);
  console.log('second arg: ', arguments[1]);
}

printSomething('hi', 'there');

// arguments length:  2
// all:  [Arguments] { '0': 'hi', '1': 'there' }
// first arg:  hi
// second arg:  there
```

Explain this code

How to use the `arguments`?

The `arguments` can be used in the `for` loop:

```
function printArgs() {
  for (let i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}

printArgs('hey', 'there', 'ironhacker');

// hey
// there
// ironhacker

printArgs(1, 77, { name: 'milo' }, ['cat', 'dog']);

// 1
// 77
// { name: 'milo' }
// [ 'cat', 'dog' ]
```

Explain this code

Keep in mind that `arguments` is an object so `.forEach()` and other array specific methods can't be used on it. If you try to use array methods on it, you will get an error similar to this one:

```
TypeError: arguments.forEach is not a function
```

Explain this code

However, you should treat it as an array when using it in your code. Although it has some limitations, this "array-like object" can be easily turned into an array, if needed:

```
const args = Array.from(arguments);
```

Explain this code

Check the example here:

```
function useArgsAsArr() {
  const argsArr = Array.from(arguments);

  argsArr.forEach(el => console.log(el));
}

useArgsAsArr('i', 'am', 'iterated', 'with', 'forEach');

// i
// am
// iterated
// with
// forEach
```

Explain this code

Bonus - First-class citizens

This is not in the scope of this topic, so we will just briefly explain the terminology, and furthermore, you will understand why we mentioned it here. In programming, when something is called `first-class` means that specific entity (in our case, the functions), inherits the same operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, modified, and assigned to a variable. **Source - First-class citizen (Wikipedia)**.

In conclusion, in JavaScript, functions are first-class objects because they can have properties and methods just like any other object. The main difference is that functions can be invoked (called) to perform a specific activity, and that is not the case when it comes to objects themselves.

Summary

In this lesson, you have learned one more way of defining functions in JavaScript, and that is functions as expressions. In the core, there is no huge difference between function declaration and expression, but when it comes to interpreting them in by our computers. We learned that the function declarations get hoisted and can be called before even defined in the code, which could but doesn't have to be always a positive side. Function expressions enforce a better structured code.

We learned that functions could be passed as arguments to other functions, and then we saw how that looks in the case of closures.
We learned that some functions would be used only once, so no need to name them (aka anonymous functions).
The ES6 brought a much nicer and shorter way of writing functions using the arrow function expression syntax.
And finally, we saw that functions by default have access in their bodies to the `arguments`, array-like object, and we saw a couple of use cases.

Extra resources

- **Everything in js is an object? (Google search)**
- **Arrow function expressions**
- **The arguments object**

Mark as completed

PREVIOUS
← CodeWars

NEXT
JS | Arrays - Map, Reduce & Filter →