JavaScript For Web

Week 2, Lecture 2 - JavaScript Fundamentals: functions

Instructor: Jason Xu

Today's Overview

- What is Nesting
- How to define and invoke different kinds of functions.
- How to use the return value.
- What function scope is.
- **Tutorial**: Assignment 1, Exercise 4

Nesting

It is perfectly OK to put one `if...else` statement inside another one — to nest them. For example, we could update our weather forecast application to show a further set of choices depending on what the temperature is:

```
...

if (choice === "sunny") {
  if (temperature < 86) {
    para.textContent = `It is ${temperature} degrees outside — nice and sunny. Let's go out to the beach, or the park, and get an ice cream.`
  } else if (temperature >= 86) {
    para.textContent = `It is ${temperature} degrees outside — REALLY HOT! If you want to go outside, make sure to put some sunscreen on.`
  }
}

...
```

What is a function in JavaScript

Reusable blocks of code

- Allow you to store a piece of code that does a single task inside a defined block, and then call that code whenever you need it using a single short command — rather than having to type out the same code multiple times.
- We've already seen examples of built-in functions, such as `console.log()`

Functions vs **methods**

- Functions that are part of objects are called methods. In short, they are same thing.

Invoking functions

- To use a function after it has been defined, you've got to run — or invoke — it. This is done by including the name of the function in the code somewhere, followed by parentheses.

```
// Define a function
function favoriteAnimal(animal) {
  return animal + " is my favorite animal!"
}

// Invoke(Run) a function
console.log(favoriteAnimal('Goat'))
console.log(favoriteAnimal('Cat'))
```

Function Declaration

To create a function we can use a `function` keyword.

- The `function` keyword goes first.
- Then goes the name of the function.
- Then a list of parameters between the parentheses `()` (comma-separated).
- It's optional to have parameters, but you need to keep the parentheses all the time because it's part of the syntax. `()`
- Finally the code of the function, also named “the function body”, between curly braces `{}`.

```
function name(parameter1, parameter2, ... parameterN) {
 // body
}
```

For example:

```
function showMessage() {
  console.log('Hello everyone!')
}
```

To invoke/run this function, it can be called by its name: `showMessage()`

```
showMessage()
```

Local Variables

A variable declared inside a function is called local variable, which is only visible inside that function.

```
function showMessage() {
  let message = "Hello, everyone!" // local variable

  console.log(message)
}

showMessage() // Hello, I'm JavaScript!

console.log(message) // <-- Error! The variable is local to the function
```

Global Variables

Variables declared outside of any function are called global variables.

```
let userName = 'John'

function showMessage() {
  userName = "Bob" // (1) changed the outer variable

  let message = 'Hello, ' + userName
  console.log(message)
}

console.log( userName ) // John before the function call

showMessage()

console.log( userName ) // Bob, the value was modified by the function
```

Global Variables continued

If a same-named variable is declared inside the function then it shadows the outer one. For instance, in the code below the function uses the local `userName`. The outer one is ignored:

```
let userName = "John"

function showMessage() {
  let userName = "Bob" // (1) changed the outer variable

  let message = "Hello, " + userName
  console.log(message)
}

showMessage()

console.log( userName ) // John, unchanged, the function did not access the outer variable
```

Comment: It's a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Function parameters

We can pass arbitrary data to functions using **parameters**. In the example below, the function has a parameter `animal`

```
// Define a function
function favoriteAnimal(animal) {
  return animal + ' is my favorite animal!'
}
```

When a value is passed as a function parameter, it's also called an **argument**.

```
// Invoke(Run) a function
console.log(favoriteAnimal('Goat'))
console.log(favoriteAnimal('Cat'))
```

Function parameters continued

- If a function is called, but an argument is not provided, then the corresponding value becomes undefined.

```
console.log(favoriteAnimal()) // undefined is my favorite animal!
```

- We can specify the default value (to use if omitted) value for a parameter in the function declaration, using `=`:

```
function favoriteAnimal(animal = 'Goat') {
  return animal + ' is my favorite animal!'
}

console.log(favoriteAnimal()) // Goat is my favorite animal!
```

- Sometimes it makes sense to assign default values for parameters at a later stage after the function declaration.

```
function favoriteAnimal(animal) {
  if(animal === undefined) {
    animal = 'Goat'
  }

  return animal + ' is my favorite animal!'
}
```

Returning a value

- A function can return a value back into the calling code as the result. The simplest example would be a function that sums two values:

```
function sum(num1, num2) {
  return num1 + num2
}

let result = sum(1, 2)
console.log(result)
```

- It is possible to use return without a value. That causes the function to exit immediately.

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return
  }

  console.log( "Showing you the movie" )
  // ...
}
```

- Checking the parameters and return value for build-in functions/methods in JavaScript. For example `Date.getHours()`

Function scope

Scope is a **very important** concept when dealing with functions.

- When you create a function, the variables and other things defined inside the function are inside their own separate scope, meaning that they are locked away in their own separate compartments, unreachable from code outside the functions.
- If a variable is declared inside a code block `{...}`, it's only visible inside that block.
- Scopes can also be layered in a hierarchy, so that child scopes have access to parent scopes.

```
let varInGlobal

function my_function() {
  let varInFunction

  if(...) {
    let varInConditionalScope
  } else {
    let anotherVarInAnotherConditionalScope
  }
  ...
}
```

Function Expression

Function Expression is a function, created inside an expression or inside another syntax construct. Here, the function is created on the right side of the “assignment expression” `=`:

```
// Function Expression
let sum = function(a, b) {
  return a + b
}
```

Function Declaration vs Function Express

- A Function Declaration can be called earlier than it is defined.

```
sayHi("John") // Hello, John

function sayHi(name) {
  console.log( `Hello, ${name}` )
}
```

- A Function Expression is created when the execution reaches it and is usable only from that moment.

```
sayHi("John") // error!

let sayHi = function(name) {
  console.log( `Hello, ${name}` )
}
```

ES6: Arrow function

```
let sayHi = function(name) {
  console.log( `Hello, ${name}` )
}
```

There's another very simple and concise syntax for creating functions, that's often better than Function Expressions. It's called "arrow functions"

```
let sayHi = (name) => {
  console.log(`Hello, ${name}`)
}

sayHi(`John`)
```

In order words, it's the shorter version of function expression.

```
let sum = (a, b) => a + b

/* This arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

console.log(sum(1, 2)) // 3
```

My comment: practice arrow function.

Thank you