JavaScript Essentials

Functions

Table of Contents

1. Overview
2. Defining functions
3. Calling functions
4. Function call stack
5. Recursive function
6. Q&A

Lesson Objectives

- Understand the fundamentals concepts behind JavaScript functions
- Understand the two syntax of functions (function declaration and function expression)
- Able to differentiate between function declaration and function expression
- Able to invoke function and understand what will happened under the hood
- Understand recursion mechanism to solve coding problem

Section 1

Overview

Overview

- Another essential concept in coding is **functions**, which 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. In this article we'll explore fundamental concepts behind functions such as basic syntax, how to invoke and define them, scope, and parameters.

Overview

- In JavaScript, you'll find functions everywhere. In fact, we've been using functions all the way through the course so far; we've just not been talking about them very much. Now is the time, however, for us to start talking about functions explicitly, and really exploring their syntax.
- Pretty much anytime you make use of a JavaScript structure that features a pair of parentheses — () — and you're **not** using a common built-in language structure like a for loop, while or do...while loop, or if...else statement, you are making use of a function.

Overview – Summary

- Functions are essentials part in any Programming Language especially JavaScript
- Functions allow you to store a piece of code that does a single task inside a defined block, give that block a name then call that code (reuse) whenever you need
- Functions that we have encountered so far are: split, join…

Section 2

Defining functions

Defining functions - Function declarations

- A **function definition** (also called a **function declaration**, or **function statement**) consists of the function keyword, followed by:
- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly brackets, {...}.

```
1 function square(number) {
2   return number * number;
3 }
```

Defining functions - Function declarations

- The function square takes one parameter, called number. The function consists of one statement that says to return the parameter of the function (that is, number) multiplied by itself. The statement return specifies the value returned by the function:

```
1 | return number * number;
```

Defining functions - Function declarations

- Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.
- If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

Defining functions - Function declarations

- If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
  theObject.make = 'Toyota';
}

var mycar = {make: 'Honda', model: 'Accord', year: 1998};
var x, y;

x = mycar.make; // x gets the value "Honda"

myFunc(mycar);
y = mycar.make; // y gets the value "Toyota"
                // (the make property was changed by the function)
```

Defining functions - Function expressions

- While the function declaration above is syntactically a statement, functions can also be created by a function expression.
- Such a function can be **anonymous**; it does not have to have a name. For example, the function square could have been defined as:

```
1 | const square = function(number) { return number * number }
2 | var x = square(4) // x gets the value 16
```

Defining functions - Function expressions

- However, a name *can* be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

```
1 const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) }
2
3 console.log(factorial(3))
```

Defining functions - Function expressions

- Function expressions are convenient when passing a function as an argument to another function. The following example shows a map function that should receive a function as first argument and an array as second argument.

```
1  function map(f, a) {
2    let result = []; // Create a new Array
3    let i; // Declare variable
4    for (i = 0; i != a.length; i++)
5      result[i] = f(a[i]);
6    return result;
7  }
```

Defining functions - Function expressions

- In the following code, the function receives a function defined by a function expression and executes it for every element of the array received as a second argument.

```
function map(f, a) {
  let result = []; // Create a new Array
  let i; // Declare variable
  for (i = 0; i != a.length; i++)
    result[i] = f(a[i]);
  return result;
}
const f = function(x) {
   return x * x * x;
}
let numbers = [0, 1, 2, 5, 10];
let cube = map(f,numbers);
console.log(cube);
```

Defining functions - Function expressions

- In JavaScript, a function can be defined based on a condition. For example, the following function definition defines myFunc only if num equals 0:

```
1  var myFunc;
2  if (num === 0) {
3    myFunc = function(theObject) {
4      theObject.make = 'Toyota';
5    }
6  }
```

Defining functions - Function expressions

- In addition to defining functions as described here, you can also use the Function constructor to create functions from a string at runtime, much like eval().
- A **method** is a function that is a property of an object. Read more about objects and methods in Working with objects.

Practice defining functions

Defining functions - Overview

- In JavaScript there are 2 ways we can use to declare a function: function declaration and function expression
- In function declaration (or function **statement**), keyword function must be the very first keyword in a statement (there is no other keyword or symbols before it)
- Also in function declaration, function name is required
- In function **expression**, function must appear inside an expression and function name is optional
- **Note:** use function declaration for beginner

Section 3

Calling functions

Calling functions

- *Defining* a function does not *execute* it. Defining it simply names the function and specifies what to do when the function is called.
- **Calling** the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

```
1 | square(5);
```

Calling functions

- The preceding statement calls the function with an argument of 5. The function executes its statements and returns the value 25.
- There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime.
- It turns out that *functions are themselves objects*—and in turn, these objects have methods. (See the Function object.) One of these, the apply() method, can be used to achieve this goal.

Calling functions - Overview

- Functions do not run until you call it
- To call (invoke) a function use syntax: functionName(para)
- Make sure the functionName is in scope
- You can pass parameter to function calls any number you like at runtime. If there is more parameter at run time then the excess parameter will be dropped. If there is less then other parameter will be undefined
- Once called, the program at the first statement inside the body of the function

Practice calling functions

Section 4

Function Call Stack

Function Call Stack

- A **call stack** is a mechanism for an interpreter (like the JavaScript interpreter in a web browser) to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc.
- **Stack** follow LIFO principles (last in first out)

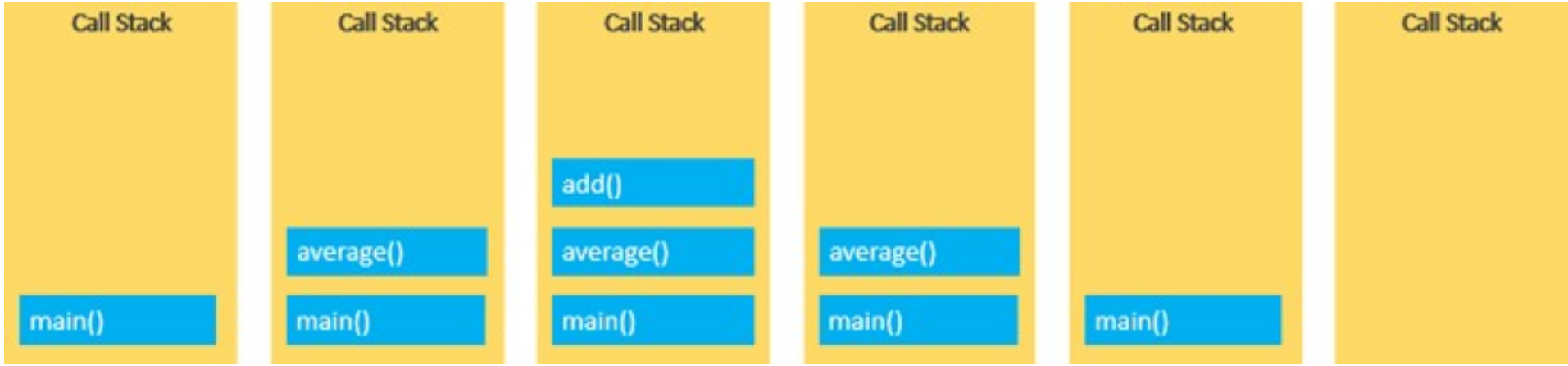
Function Call Stack

- When a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function.
- Any functions that are called by that function are added to the call stack further up, and run where their calls are reached.
- When the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing.
- If the stack takes up more space than it had assigned to it, it results in a "stack overflow" error.

Function Call Stack

- JavaScript engine uses a **call stack** to manage execution contexts: the Global Execution Context and Function Execution Contexts.
- The call stack works based on the LIFO principle i.e., last-in-first-out.
- When you execute a script, the JavaScript engine creates a Global Execution Context and pushes it on top of the call stack.
- Whenever a function is called, the JavaScript engine creates a Function Execution Context for the function, pushes it on top of the Call Stack, and starts executing the function.
- If a function calls another function, the JavaScript engine creates a new Function Execution Context for the function that is being called and pushes it on top of the call stack.
- When the current function completes, the JavaScript engine pops it off the call stack and resumes the execution where it left off in the last code listing.
- The script will stop when the call stack is empty.

Function Call Stack



Function Call Stack

Practice Function Call Stack

Function Call Stack - Summary

- JavaScript must keeps track what function is currently being run and what functions are called from within that function, etc.
- Call Stack is the mechanism used for tracking
- Call Stack is like a stack of disk (the first come in will be on top and the first to be removed is also on top)

Section 5

Recursive function

Recursive function

- The act of a function calling itself, recursion is used to solve problems that contain smaller sub-problems. A recursive function can receive two inputs: a base case (ends recursion) or a recursive case (resumes recursion).
- A **recursive function** is a function that calls itself

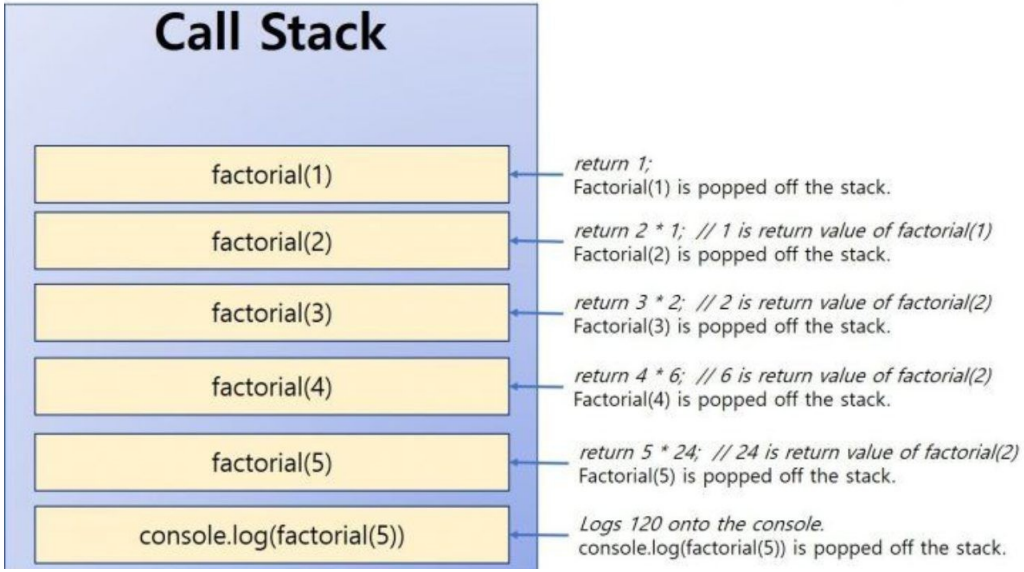
Recursive function

- Loop with recursion

```
1  function loop(x) {
2     if (x >= 10)
3        return;
4     loop(x + 1);
5  };
6  //Using ECMAScript 2015 arrow notation
7  const loop = x => {
8     if (x >= 10)
9        return;
10    loop(x + 1);
11 };
```

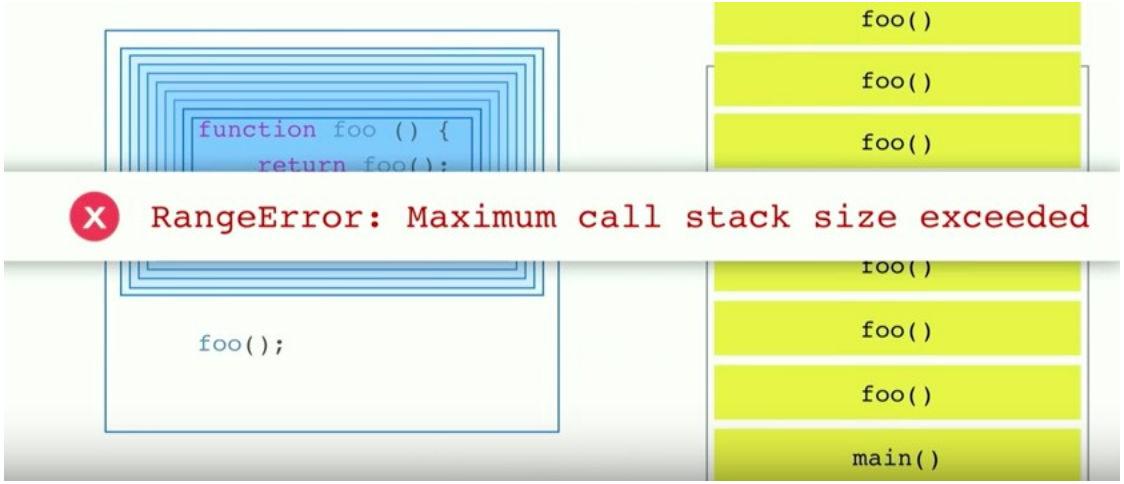
Recursive function

- In fact, recursion itself uses a stack: the function stack. The stack-like behavior can be seen in the following example:



Recursive function

- Resource is limited



Practice recursion

Recursive functions - Overview

- Recursion is the act of a function calling itself.
- Recursion provide an elegant mechanism to loop and solve complex problem
- Recursion allow to reduce the complexity of a problem at hand
- Always remember to add base case inside recursive functions
- Recursion itself use Call Stack and that Stack is limited. Too may recursion call may lead to **Stack Overflow**

Thank you

Q&A