JavaScript Essentials

Functions

Table of Contents

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 - Problem

- Suppose we want to calculate the sum of an array [1, 2, 3, 4, 5]
- Loop can do this!!!

```
var arr = [1, 3, 4, 5];

var sum = 0;
for (var i = 0; i < arr.length; i += 1) {
   sum += arr[i];
}

console.log(sum);
```

```
13
```

Overview - Problem

- We then can use that algorithm to calculate the sum of scores, salaries, etc…

```
var scores = [10, 8, 7, 6, 9];

var sum = 0;
for (var i = 0; i < scores.length; i += 1) {
   sum += scores[i];
}

console.log(sum);
```

```
40
```

```
var salaries = [10, 18, 17, 16, 19];

var sum = 0;
for (var i = 0; i < salaries.length; i += 1) {
   sum += salaries[i];
}

console.log(sum);
```

```
80
```

Overview - Problem

- But we are repeating the same task here (looping array and add sum)

```
var scores = [10, 8, 7, 6, 9];

var sum = 0;
for (var i = 0; i < scores.length; i += 1) {
   sum += scores[i];
}

console.log(sum);
```

```
40
```

```
var salaries = [10, 18, 17, 16, 19];

var sum = 0;
for (var i = 0; i < salaries.length; i += 1) {
   sum += salaries[i];
}

console.log(sum);
```

```
80
```

Diffrence in variable, but same structure

Overview - Problem

- But we are repeating the same task here (looping array and add sum)

Overview

- What we are lookg for is another essential concept in coding: **function**
- Function allow you to store a piece of code that does a single task inside a defined block
- Then give it a name so you can call that code whenever you need it using a single short command
- No more repeat!!!

Overview

- In JavaScript, you'll find functions everywhere
- Pretty much, anytime you make use of a JavaScript structure that features a pair of parentheses — () — you are making use of a function.

Overview – Summary

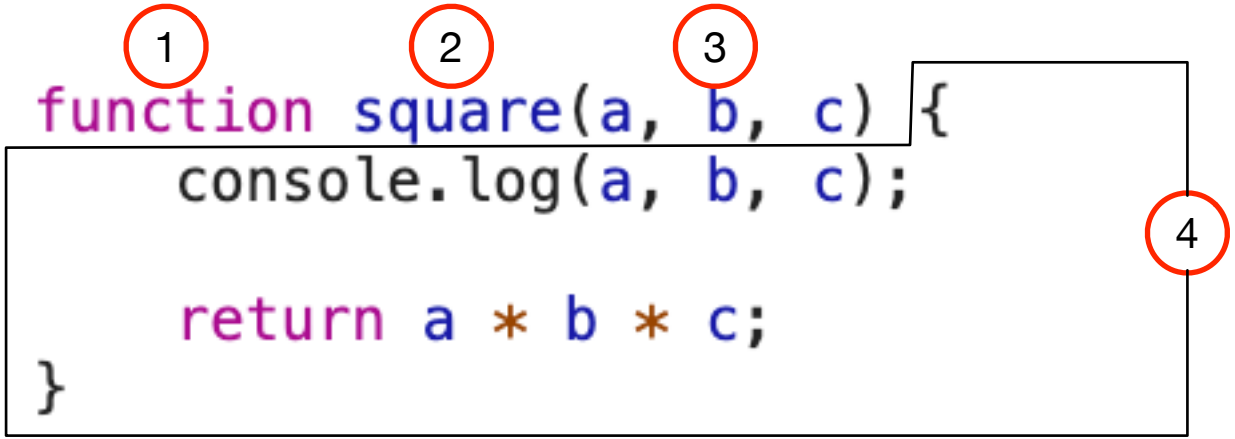
- Functions are essentials part in any Programming Language especially JavaScript
- Functions allow you to store a piece of code that does a single task inside a defined block, give that block a name then call that code (reuse) whenever you need
- Functions that we have encountered so far are: split(),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

Section 2

Defining functions

Defining functions - Function declarations

- A **function definition** consist of 4 elements:



Defining functions - Function declarations

- **Practice**: refactor below code to a function called sum so we can use it to calculate the sum of scores, salaries, etc…

```
var arr = [1, 3, 4, 5];

var sum = 0;
for (var i = 0; i < arr.length; i += 1) {
   sum += arr[i];
}

console.log(sum);
```

```
13
```

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

- For example, if you define the function **sum** that take 3 parameters, you could call it as follows:

```
function sum(a, b, c) {
    return a + b + c;
}

// call function sum
// and pass a = 1, b = 2, c = 3 to it
sum(1, 2, 3);

6
```

Calling functions

- You can pass more or less number of parameter when calling functions

```
function sum(a, b, c) {
   return a + b + c;
}

// pass more parameters to sum
// extra parameters are discarded
sum(1, 2, 3, 4, 5);
```

```
6
```

```
function sum(a, b, c) {
   return a + b + c;
}

// pass less parameter to sum
// c = undefined since no matching parameter
sum(1, 2);
```

```
NaN
```

Defining functions - Function declarations

- Primitive parameters (such as a number) are passed to functions **by copy value**:

```
var a = 1;
var b = 2;

console.log('before swap: ', a, b);
function swap(a, b) {
  var tmp = a;
  a = b;
  b = tmp;
  console.log('inside swap: ', a, b);
}

swap(a, b);
console.log('after swap: ', a, b);
```

```
before swap:  1 2
inside swap:  2 1
after swap:  1 2
```

Defining functions - Function declarations

- Object parameters are passed to functions **by reference**:

```
var fresher = {
  name: 'Anh',
  clazz: 'Front-end'
}

function changeName(f) {
  f.name = 'Binh';
}

changeName(fresher);
console.log(fresher);
```

```
▶ {name: "Binh", clazz: "Front-end"}
```

Defining functions - Function declarations

- Same for array parameter:

```
var scores = [10, 8, 7, 6, 9];

function checkScores(array) {
  array.push(11); // same as scores.push(11)
}

checkScores(scores);
console.log(scores);
```

```
▶ (6) [10, 8, 7, 6, 9, 11]
```

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

- 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

- A **method** is a function that is a property of an object:

```
function log() {
  console.log('Hello');
}

var o = {
  log: log
};

o.log()
```

```
Hello
```

```
var o = {
  log: function log() {
    console.log('Hello');
  }
};

o.log()
```

```
Hello
```

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

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
- Once **called**, the program run at the first statement inside the body of the function

Section 3

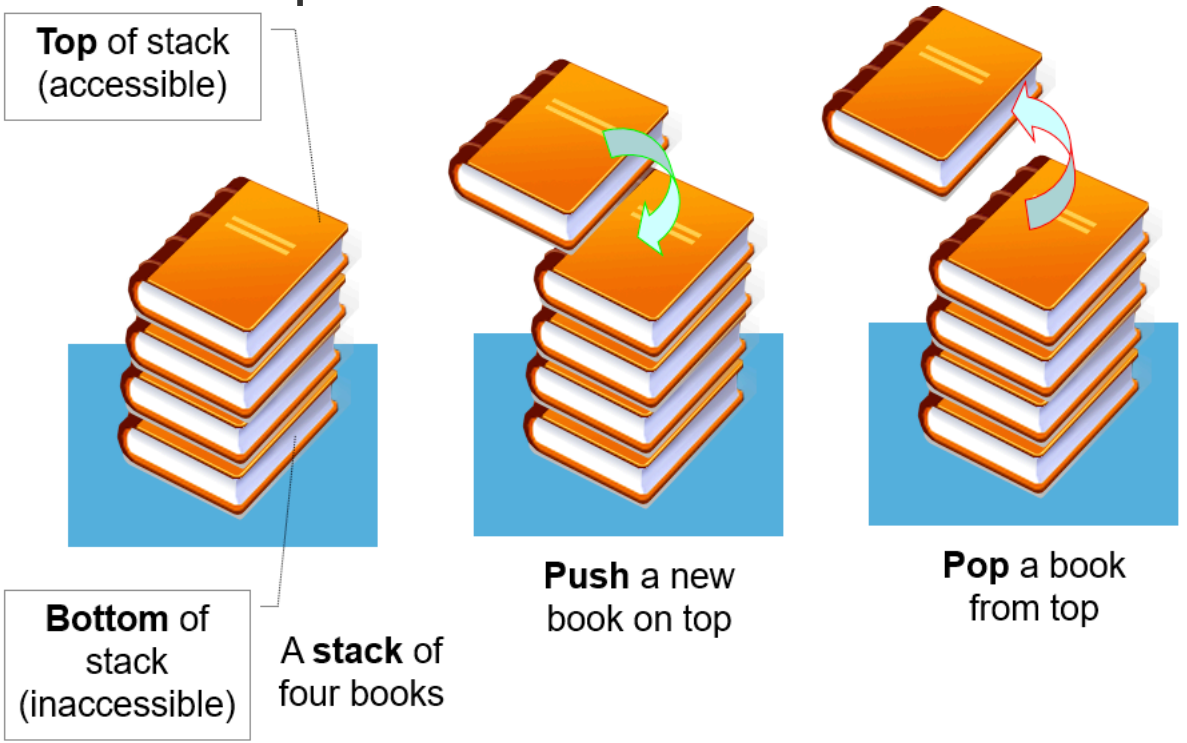
Function Call Stack

Function Call Stack

- A **call stack** is a mechanism for JavaScript to keep track of 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

- Think of Stack like a pile of book:



Function Call Stack

- When function is called, JavaScript 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, JavaScript 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

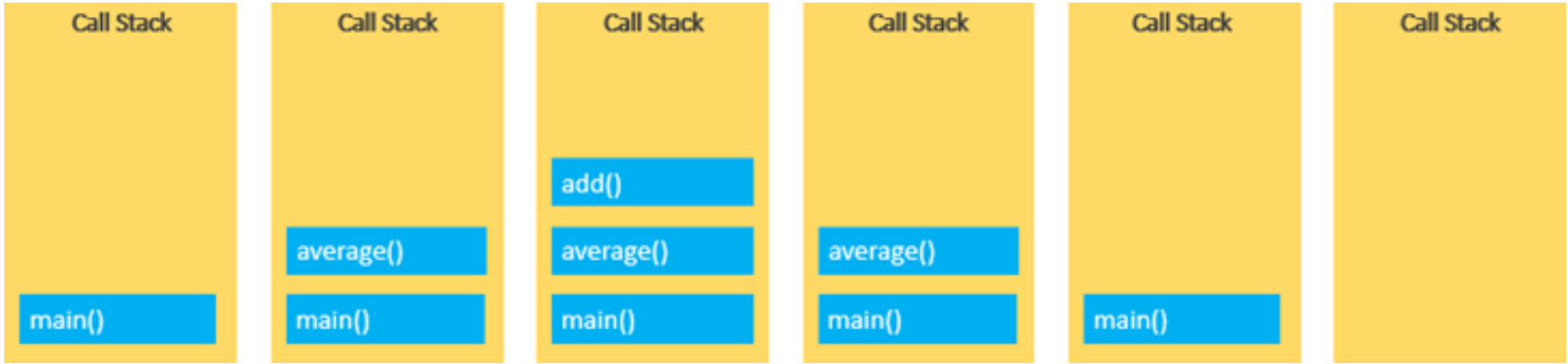
- Try running below code, what is the output ?

```
function average() {
  console.log('inside average');
  add();
}
function add() {
  console.log('inside add');
}
function main() {
  console.log('inside main');
  average();
}

main();
```

Function Call Stack

- How Call Stack work



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 pile of book (the first come in will be on top and the first to be removed is **also** on top)

Section 5

Recursive function

Recursive function

- Remember “divide and conquer” in How to think like Programmer?
- 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

- Recursion usage:

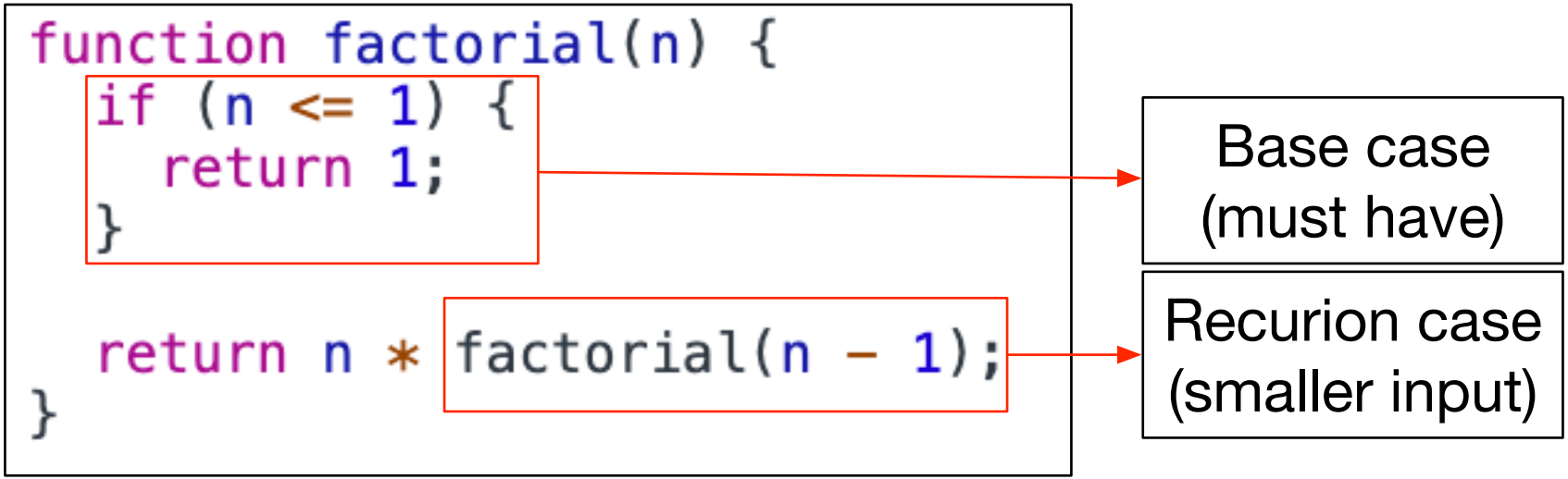
```
function loop(i) {
  if (i > 10) {
    return;
  }

  console.log(i);
  loop(i + 1);
}

loop(0);
```

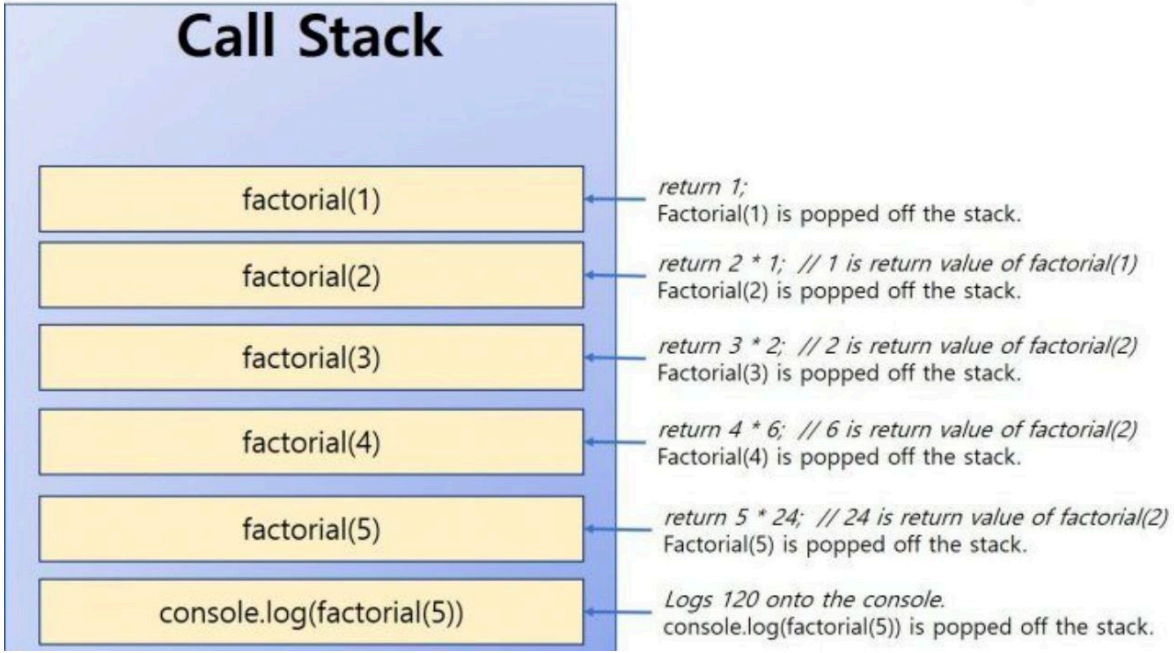
Recursive function

- Calculate factorial:



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

- Make sure to add Base case:

```
function factorial(n) {
  return n * factorial(n - 1);
}

factorial(5);
```

```
▶Uncaught RangeError: Maximum call
stack size exceeded
```

Recursive function

- Or if input is too big:

```
function factorial(n) {
  if (n <= 1) {
    return 1;
  }

  return n * factorial(n - 1);
}

factorial(10000000);
```

```
▶Uncaught RangeError: Maximum call stack size exceeded
```

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