JS Functions

Asst.Prof. Dr. Umaporn Supasitthimethee
ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

JavaScript: The Definitive Guide, Seventh Edition, by David Flanagan

Functions

- A function is a **block of JavaScript code that is defined once** but may be **executed**, or invoked, **any number of times**.
- **JavaScript functions are parameterized:** a function definition may include a list of identifiers, known as parameters, that work as local variables for the body of the function.
- If a **function** is assigned to **a property of an object**, it is **known as a method** of that object.
- When a function is invoked on or through an object, that object is the invocation context or `this` value for the function.
- Functions designed to initialize a **newly created object** are called **constructors**.
- In JavaScript, **functions are objects**, and they can be manipulated by programs. JavaScript can **assign functions to variables and pass them to other functions**

Function Declarations

Function declaration:

- the function keyword
- the name of the
- a list of parameters to the function
- the JavaScript statements that define the function, enclosed in curly brackets, {...}.

```
function name([param1[, param2[, ..., paramN]]]) {
               statements

}
```

Function Expressions

Function expressions look a lot like function declarations, but they appear within the context of a larger expression or statement, and *the name is optional*. However, a name can be provided with a function expression.

Function expression: function expression defines a function and assign it to a variable

```
//functions as Values
const getRectangleArea = function(width, height) {
            return width * height;
};
```

Named function expression : Function expressions can include names, which is useful for recursion.

```
//functions as Values
let fact = function factorial(n) {
                  console.log(n) ;
                  if (n <= 1) {
                    return 1;
                  }
                 return n * factorial(n - 1);
             } ;
fact (5); //120
```

Calling Functions

- *Defining* a function does not *execute* it. Defining it 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.

//06_Functions/script1.js

```
//function declaration
function square (side) {
        return side * side;
}

square(3); //calling functions
```

```
//function expression
let area=function square(){
        return side* side;
}
area(3); //calling functions
```

Functions must be *in scope* when they are called, but the function declaration can be hoisted:

```
square(3); //hoisting

function square (side) {
        return side * side;
}
```

function hoisting only works with function *declarations* — not with function *expressions*.

Arrow Function Expressions

Arrow Function expression is a compact alternative to a traditional function expression but is limited and can't be used in all situations.

```
// Traditional Function (no arguments)
let a = 4;
let b = 2;
function (){
    return a + b + 100;
}

// Arrow Function
let a = 4;
let b = 2;
() => a + b + 100;
```

```
//No param
() => expression

// One param
param => expression

// Multiple param
(param1, paramN) => expression

// Multiline statements
param1 => {
        statement1;
        …
        statementN;
}

(param1, paramN) => {
        statement1;
        …
        statementN;
}
```

Comparing traditional functions to arrow functions

```
// Traditional Function (one argument)
function  (a){
    return a + 100;
}
// Arrow Function Break Down
// 1. Remove the word "function" and place arrow
between the argument and opening body bracket
(a)=> {
    return a + 100;
}

// 2. Remove the body braces and word "return" --
the return is implied.
(a)=> a + 100;

// 3. Remove the argument parentheses
a => a + 100;
```

```
// Traditional Function (multiple arguments)
function (a, b){
       return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Function (multiline statements)
function (a, b){
        let chuck=42;
        return a + b + chuck;
}

// Arrow Function
(a, b) => {
        let chuck=42;
        a + b + chuck;
}
```

Primitive Parameter Passing

- **Primitive parameters** 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**.

```
function square (side) {
        return side * side;
}
```

Object Parameter Passing

- **Object parameter** (i.e., a non-primitive value, such as Array or a user-defined object) are passed to function and the function changes the object's properties, **that change is visible** outside the function.

```
function myFunc(theObject) {
  theObject.model = "A9999";
}
let product = {model: "A1001", price: 199};
console.log(mycar.model); // "A1001“

myFunc(mycar);
console.log(mycar.model); // "A9999"
```

Higher-Order Functions

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

//01_BasicJS/ script2.js

- JavaScript Functions are **first-class citizens**
  - be assigned to variables (and treated as a value)
  - be passed as an argument of another function
  - be returned as a value from another function

```
//1. store functions in variables

function add(n1, n2) {
  return n1 + n2
}
let sum = add

let addResult1 = add(10, 20)
let addResult2 = sum(10, 20)

console.log(`add result1: ${addResult1}`)
console.log(`add result2: ${addResult2}`)
```

```
//2. returned as a value from another function

function operator(n1, n2, fn) {
  return fn(n1, n2)
}

//3. Passing a function to another function
function multiply(n1, n2) {
  return n1 * n2
}

let addResult3 = operator(5, 3, add)
let multiplyResult = operator(5, 3, multiply)

console.log(`add result3 : ${addResult3}`)
console.log(`multiply result: ${multiplyResult}`)
```

Function scope

- **Variables defined inside a function** cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
- However, **a function can access all variables and functions defined inside the scope in which it is defined.**
- In other words, **a function defined in the global scope can access all variables defined in the global scope.**
- **A function defined inside another function** can also **access all variables defined in its parent function,** and **any other variables to which the parent function has access**.

Function scope and Nested Functions

//06_Functions/script2.js

```
// The following let variables are defined in the global scope
let mid = 20;
let final = 5;

let fname = 'Ada';

// sum function is defined in the global scope
function sum() {
  return mid + final;
}

console.log(`#1 sum: ${sum()}`); // Returns 25
mid = 10;
console.log(`#2 sum: ${sum()}`); // Returns 15

function getScore() {
  let mid = 10;
  let final = 30;

  //yourScore is nested function
  function yourScore() {
    return fname + ' scored ' + (mid + final);
  }
  return yourScore();
}
console.log(getScore()); // Returns "Ada scored 40"
```

Nested Functions and closures

- You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.
- **A closure is an expression (most commonly, a function)** that can have free variables together with an environment that binds those variables (that "closes" the expression).
- Since a **nested function is a closure**, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

Closures

- Closures are one of the most powerful features of JavaScript.
- JavaScript allows for **the nesting of functions** and grants the inner function **full access to all the variables and functions defined inside the outer function** (and all other variables and functions that the outer function has access to).
- However, **the outer function does not have access** to the variables and functions defined **inside the inner function.** This provides a sort of encapsulation for the variables of the inner function.

Closures

//06_Functions/script3.js

```
let getScoringPass = function (scores) {
  //bind and store "scores" argument to use in the nested "cuttingPoint" function
  function cuttingPoint(cuttingScore) {
    return scores.filter((score) => score >= cuttingScore);
  }
  return cuttingPoint;
};
//fn_cuttingPoint1 and fn_cuttingPoint2 are instance closure functions
//that bind to each their outer parameter "scores"
let fn_cuttingPoint1 = getScoringPass([50, 15, 32, 80, 100]);
console.log(fn_cuttingPoint1(50)); //[ 50, 80, 100 ]
let fn_cuttingPoint2 = getScoringPass([-10, -15, -53, -97, -32]);
console.log(fn_cuttingPoint2(-30)); //[ -10, -15 ]
```

Using the arguments object

- The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

```
arguments[i]
```

where *`i`* is the ordinal number of the argument, starting at `arguments[0]`. The total number of arguments is indicated by `arguments.length`.

Array-like" means that `arguments` has a length property and properties indexed from zero, but it doesn't have Array's built-in methods like forEach() or map().

Using the arguments object

//06_Functions/script4.js

```
function printStudents(students) {
  let result = '';
  // iterate through arguments
  let separator = arguments[0];
  for (i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}
console.log(printStudents('.', 'Adam', 'John', 'Danai’));
//Adam.John.Danai.
```

Function Parameters

- Starting with ECMAScript 2015, there are two new kinds of parameters:
  - default parameters
  - rest parameters

Default Parameters

- In JavaScript, **parameters of functions default to undefined**.
- In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are undefined.
- With ***default parameters***, a manual check in the function body is no longer necessary. You can put the default value for any parameters in the function head

//06_Functions/script5.js

```
//default parameter
function who(name = 'unknown') {
  return name;
}
console.log(who()); //unknown
console.log(who('Umaporn’)); //Umaporn
```

Rest Parameters

- **Rest parameters** allow us to write functions that can be invoked with an indefinite number of arguments as **an array**
- Rest parameters are Array instances
- **Only the last parameter** in a function definition can be a **rest parameter**

```
//rest parameters
function sum(opName, ...theNumbers) {
  console.log(opsName); //'sum'
  return theNumbers.reduce((previous, current) => {
    return previous + current;
  });
}

console.log(sum('sum', 1, 2, 3)); //6
console.log(sum('sum', 1, 2, 3, 4, 5)); //15
```

Spread Parameters

- **Spread operator** takes the array of parameters and spreads them across the arguments in the function call.

```
function sum(num1, num2, num3) {
  return num1 + num2 + num3;
}
let nums = [5, 20, 15];
//spread parameter
console.log(sum(...nums)); //40
```

Destructuring Function Arguments into Parameters

- If you define a function that has parameter names **within square brackets**, you are telling the **function to expect an array value** to be passed for each pair of square brackets.
- As part of the invocation process, **the array arguments will be unpacked into the individually named parameters.**

```
//destructuring function arguments into parameters
function arrayAdd1(v1, v2) {
  return [v1[0] + v2[0], v1[1] + v2[1]];
}
console.log(arrayAdd1([1, 2], [3, 4])); // [4,6]

function arrayAdd2([x1, y1], [x2, y2]) {
  // Unpack 2 arguments into 4 parameters
  return [x1 + x2, y1 + y2];
}
console.log(arrayAdd2([1, 2], [3, 4])); // [4,6]
```

JavaScript Export

- The **export** statement is used when creating JavaScript modules to export live bindings to *functions*, *objects*, or *primitive values* from the module so *they can be used by other programs with the **import** statement.*
- There are **two types** of exports:
  - Named Exports (Zero or more exports per module)
  - Default Exports (One per module)

Module Export

```
// Exporting individual features
export let name1, name2, …, nameN; // also var, const
export let name1 = …, name2 = …, …, nameN; // also var, const
export function functionName(){...}
export class ClassName {...}
```

```
// Export list export { name1, name2, …, nameN };
// Renaming exports export { variable1 as name1, variable2 as name2, …, nameN };
```

```
// Default exports
export default expression;
export default function (…) { … } // also class, function
export default function name1(…) { … } // also class, function
export { name1 as default, … };
```

Module Import

The import statement cannot be used in embedded scripts unless such script has a type="module".

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { export1 , export2 } from "module-name";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export1 [ , [...] ] } from "module-name";
```

module-name: The module to import from. This is often a relative or absolute path name to the **.js** file containing the module.

```
//dataFuncExport.js
//named export
export const frontEndFramework = ['Vuejs', 'React', 'Angular’]
//or
const frontEndFramework = ['Vuejs', 'React', 'Angular’]
export { frontEndFramework }

export function greeting() {
  return 'Hello, function from another module’
}
//default export
export default function getInstructor() {
  return `Umaporn Supasitthimethee`;
}
```

```
<!--index.html -- > //run on browser
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>

</head>
<body>
  <script src="./main.js" type="module"></script>

</body>
</html>
```

//06_Functions/ExportImportModules/script.js

```
//subjectExport.js
const subject = 'INT201'
export {subject}
```

```
//main.js
import defaultExport, {greeting, frontEndFramework as frontEnd} from './dataFuncExport.js';

import {subject} from './subjectExport.js'

console.log(`Frontend Framework: ${frontEnd}`)
console.log(greeting())
console.log(defaultExport);
console.log(subject)
```