JS Working 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

Higher-Order Functions

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

- 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 function 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. Passing a function to another function
function operator(n1, n2, fn) {
  return fn(n1, n2)
}
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}`)
```

```
//3. return function as value of another function
function sayGoodBye(){
    return 'Good bye'
}
function doSomething(){
    return sayGoodBye
}
let doIt=doSomething() //let doIt=sayGoodBye
console.log(doIt())
```

Pass Function to other functions

filter() creates a new array with all elements that pass the test implemented by the provided function.

1. Arrow function passing to filter function

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);
console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

2. Callback function passing to filter function

```
function isMorethanFive(value) {   return value > 5 }
const filterNums = [12, 5, 8, 130, 44].filter(isMorethanFive);
// filterNums is [12, 8, 130, 44]
```

3. Inline callback function passing to filter function

```
const nums = [-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
const primeNums =  nums.filter(function (num) {
    for (let i = 2; num > i; i++) {
      if (num % i === 0) {
        return false
      }}
    return num > 1
  })
// primeNums is [ 2, 3, 5, 7, 11, 13 ]
```

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.
- In JavaScript, **functions are objects**, and they can be manipulated by programs. JavaScript can **assign functions to variables and pass them to other functions**
- JavaScript function definitions **can be nested within other functions,** and they have access to any variables that are in scope where they are defined.

https://developer.mozilla.org/en-US/docs/Glossary/Function

Function Types

- An **anonymous function** is a function without a function name. Only function expressions can be anonymous, function declarations must have a name

```
// Anonymous function created as a function expression
function () {}

// Anonymous function created as an arrow function
() => {}
```

- A **named function** is a function with a function name

```
// Function declaration
function foo() {}

// Named function expression
Const barFn=function bar() {}

// Arrow function
const barAF = () => {}
```

https://developer.mozilla.org/en-US/docs/Glossary/Function

Function Types

- An **inner (nested) function** is a function inside another function

```
function addSquares(a, b) {
   function square(x) {
      return x * x
   }
      return square(a) + square(b)
}
```

```
// Arrow function
const addSquares2 = (a, b) => {
      const square = (x) => x * x
      return square(a) + square(b)
}
```

https://developer.mozilla.org/en-US/docs/Glossary/Function

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

```
// 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
}
const score = getScore()
console.log(score()) // Returns "Ada scored 40"
```

Closures

- Closures are one of the most powerful features of JavaScript.
- A closure is **the combination of a function bundled together** (enclosed) with references to its surrounding state (the lexical environment).
- 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.
- 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.

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

Closures

```
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 ]
```

Closures

Returning multiple values from a function using an object

```
function counter() {
  let count = 0
  function increment() {
    return count++
  }
  function decrement() {
    return count--
  }
  function getCount() {
    return count
  }
  return {
    increment,
    decrement,
    getCount
  }
}
```

```
const c = counter()
c.increment()
console.log(c.getCount()) //1
c.increment()
console.log(c.getCount()) //2
c.decrement()
console.log(c.getCount()) //1
```

Using the arguments object

- The arguments object is a local variable available within all non-arrow functions. You can refer to a function's arguments inside that function by using its 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

```
function printNumbers1(num1, num2, num3) {
  console.log(`argument length: ${arguments.length}`)
  console.log(arguments[0]) //5
  console.log(arguments[1]) //10
  console.log(arguments[2]) //15
}
printNumbers1(5, 10, 15)
```

```
function printNumbers1(num1, num2, num3) {
  for (const argu of arguments) {
    console.log(argu)
  }
}
printNumbers2(5, 10, 15) //5, 10, 15
```

```
function updateArgument(x, y) {
  console.log(x) //10
  arguments[0] = 555
  console.log(x) //555
}
updateArgument(10, 5)
```

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

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

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

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

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(opsName, ...theNumbers) {
  console.log(opsName) //'sum'
  let total = 0
  for (const num of theNumbers) {
    total += num
  }
  return total
}

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

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

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
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Unpacking elements in array passed as a function parameter

- 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.**

```
function arrayAdd1([x1], [y1]) {
  return x1+y1
}
const a = [5, 8]
const b = [2, 7]
console.log(arrayAdd1(a, b)) // 7
```

```
function arrayAdd2([x1, y1], [x2, y2]) {
  return x1 + x2 + y1 + y2
}
console.log(arrayAdd2([1, 2], [3, 4])) // 10
```

```
function arrayAdd3([x1, y1], [x2, y2]) {
  return [x1 + x2, y1 + y2]
}
const x = [1, 2]
const y = [3, 4]
console.log(arrayAdd3(x, y)) // [4,6]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Unpacking properties from objects passed as a function parameter

- Objects passed into function parameters can also be unpacked into variables, which may then be accessed within the function body.

```
const students = {
  studentId: 64001,
  displayName: 'jsGuy',
  fullName: {
    firstName: 'Somchai',
    lastName: 'DeeJai'
  }
}
```

```
function studentId({ studentId }) {
  return studentId
}

console.log(studentId(students)) //64001
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Unpacking nested object properties passed as a function parameter

- **Nested objects** can also be unpacked. The example below shows the property `displayName` and `fullname.firstName` being unpacked.

```
const students = {
  studentId: 64001,
  displayName: 'jsGuy',
  fullName: {
    firstName: 'Somchai',
    lastName: 'DeeJai'
  }
}
```

```
function getFirstName({ displayName, fullName: { firstName } })
{
  return `${displayName} is ${firstName}`
}

console.log(getFirstName(students)) //jsGuy is Somchai
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Unpacking Combined Array and Object Destructuring passed as a function parameter

- Array and Object destructuring can be combined.

```
const person = [
  { id: 1, name: 'Suda' },
  { id: 2, name: 'Surapong' },
  { id: 3, name: 'Somchai' }
]
console.log(getPersonName(person)) // Surapong
```

```
function getPersonName([, { name }]) {
  return name
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment