Function and Array

Part 2

Functions: closer look

First-Class Functions

- JavaScript treats functions as first-class citizens.
- This means that functions are simply values.
- Functions are just special type of object.
- The typeof operator in JavaScript returns "**function**" for functions.
- JavaScript functions have both **properties** and **methods**

Higher-Order Functions

- A function that receives another function as argument, or returns a function, or both is called higher-order function.
- This is only possible because of first-class functions.

Passing Function as Parameter (Ex.1)

```
var handle_data = function (func) {
    var x = 2;
    var y = 3;
    return func(x, y);
}
var add = function (a, b) {
    return a + b;
}
var subtract = function (a, b) {
    return a - b;
}
console.log(handle_data(add));       // 5
console.log(handle_data(subtract));  // -1
```

Passing Function as Parameter (Ex.2)

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

Passing Function as Parameter (Ex.3)

```
setTimeout(
  function () {
    console.log('Execute later after
                 3 second');
  }, 3000
);
```

The arguments property

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
    var i;
    var max = -Infinity;
    for (i = 0; i < arguments.length; i++) {
        if (arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}
```

Function Hoisting (within scope)

```
console.log(sqr(5))
/* ... */
function sqr(n) {
  return n * n
}
```

Note: It works with function declaration only not for function expression

Returning a Function

```
function magic() {
    return function(x) {
        return x * 42;
    };
}

var answer = magic();
answer(1337); // 56154
```

ES6 Arrow Functions

An arrow function expression

- has a shorter syntax compared to function expressions
- does not have its own this, arguments, super, or new.target
- are always anonymous.

ES6 Arrow Functions

```
let add = function (x, y) {
    return x + y;
}
console.log(add(10, 20)); // 30
```

```
let add = (x, y) => x + y;
console.log(add(10, 20)); // 30;
```

Arrow Functions with Single Parameter

```
(p1) => { statements }
```

OR

```
p1 => { statements }
```

Arrow Functions with No Parameter

```
() => { statements }
```

Function and Variable Scope

```
var num1 = 20, num2 = 3;    // global variables

function multiply() {    // global function
  return num1 * num2;
}

multiply(); // Returns 60
```

Nested/Inner Function and Variable Scope

```
var country = "India";  // global variable

function getScore() {
    var score = 350;    // local variable

    function concat() {   // nested function
        return country + ' scored ' + score;
    }

    return concat();
}
getScore(); // Returns "India scored 350"
```

Nested/Inner Function and Variable Scope

- The inner function
  - can be accessed only from statements in the outer function
  - forms a **closure**
  - can use the arguments and variables of the outer function
- The outer function cannot use the arguments and variables of the inner function

Multiple Nested Functions

```
function A(x) {
    function B(y) {
        function C(z) {
            console.log(x + y + z);
        }
        C(3);
    }
    B(2);
}
A(1); // logs 6 (1 + 2 + 3)
```

Immediately Invoked Function Expression (IIFE)

- JavaScript function that runs as soon as it is defined
- It is a design pattern which is also known as a Self-Executing Anonymous Function

```
(
 function () { statements }
)
();
```

IIFE(Ex.1)

```
(function () {
    var aName = "John";
})();
// aName is not accessible from the outside scope

console.log(aName)
// "Uncaught ReferenceError: aName is not defined"
```

IIFE (Ex.2)

```
var result = (function () {
    var name = "Barry";
    return name;
})();

console.log(result); // "Barry"
```

IIFE (Ex.3)

```
let person = {
    firstName : 'John',
    lastName  : 'Doe'
};

(function (p) {
    console.log(p.firstName, p.lastName);
})(person);
```

IIFE Advantages

- Do not create unnecessary global variables and functions
- Functions and variables defined in IIFE do not conflict with other functions and variables even if they have same name
- Organizes JavaScript code
- Makes JavaScript code maintainable

Closure

Closure means that an inner function always has access to the vars and parameters of its outer function, even after the outer function has returned.

```
function init() {
    let name = 'DDU';                  // local variable
    function displayName() {  // inner function, a closure
        console.log(name);
    }
    return displayName;
}
let resFn = init();
resFn();
```

Observation: variable name exists even after completion of init function

Counter (Ex.1 Global Var)

```
var counter = 0;
function add() {
    counter += 1;
}
add();
add();
add();
// The counter should now be ???
//
// 3
```

Counter(Ex.2)

```
var counter = 0;
function add() {
    var counter = 0;
    counter += 1;
}
add();
add();
add();
// The counter should now be ???
//
// 0
```

Counter(Ex.3)

```
function add() {
    var counter = 0;
    counter += 1;
    return counter;
}
x = add();
x = add();
x = add();
// The "x" should now be ???
//
// 1
```

Counter & Closure (Ex.4.0)

```
function makeCounter() {
    var counter = 0;
    return function() { return counter += 1 }
}
var add = makeCounter();
add();
add();
add();
// The counter should now be ???
// 3
```

- *One important characteristic of closure is that outer variables can keep their states between multiple calls.*
- *Remember, inner function does not keep the separate copy of outer variables but it reference outer variables, that means value of the outer variables will be changed if you change it using inner function.*

Counter & Closure (Ex.4.1)

```
var add = (function () {
    var counter = 0;
    return function() {
        return counter += 1
    }
}) ();
add();
add();
add();
// The counter should now be ???
// 3
```

Closure (Ex.5)

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
fn_inside = outside(3);

result = fn_inside(5);  // returns 8

result1 = outside(3)(5);   // returns 8
```

Note: Variable x remains in memory even after completion of function "outside"

Closure

- The inner function has access to the scope of the outer function
- the variables and functions defined in the outer function will live longer than the duration of the outer function execution, if the inner function manages to survive beyond the life of the outer function
- A closure is created when the inner function is somehow made available to any scope outside the outer function
- A closure is a function having access to the parent scope, even after the parent function has closed

Function Parameters

- Starting with ECMAScript 2015, there are two new kinds of parameters:
  1) *default parameters* and
  2) *rest parameters*

Default Parameters

```
function multiply(a, b = 1) {
  return a * b;
}

multiply(5); // 5
```

Rest Parameters

```
function multiply(multiplier, ...theArgs) {
    return theArgs.map(x => multiplier * x);
}

var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Predefined Functions (1)

JavaScript has several top-level, built-in functions:

- eval(): evaluates JavaScript code represented as a string
- uneval(): creates a string representation of the source code of an Object
- isFinite(): determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.
- isNaN(): determines whether a value is NaN or not. Note: coercion inside the isNaN function has interesting rules; you may alternatively want to use Number.isNaN(), as defined in ECMAScript 2015, or you can use typeof to determine if the value is Not-A-Number.
- parseFloat(): parses a string argument and returns a floating point number
- parseInt(): parses a string argument and returns an integer of the specified radix

Predefined Functions (2)

- decodeURI(): decodes a Universal Resource Identifier (URI) previously created by encodeURI or by a similar routine.
- decodeURIComponent(): decodes a Universal Resource Identifier (URI) component previously created by encodeURIComponent or by a similar routine.
- encodeURI(): encodes a Universal Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).
- encodeURIComponent(): encodes a Universal Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

Points to Remember

1. JavaScript function allows you to define a block of code, give it a name and then execute it as many times as you want.
2. A function can be defined using function keyword and can be executed using () operator.
3. A function can include one or more parameters. It is optional to specify function parameter values while executing it.
4. JavaScript is a loosely-typed language. A function parameter can hold value of any data type.
5. You can specify less or more arguments while calling function.
6. All the functions can access arguments object by default instead of parameter names.
7. A function can return a literal value or another function.
8. A function can be assigned to a variable with different name.
9. JavaScript allows you to create anonymous functions that must be assigned to a variable.

Array: some more methods

Remove an item (**splice**)

```
let fruits = ["Strawberry", "Banana", "Mango"]

let pos = fruits.indexOf('Banana')

let removedItem = fruits.splice(pos, 1)

// Array ["Banana"]

// ["Strawberry", "Mango"]
```

Note: Original array is **modified**

Insert and/or replace an item (**splice**)

```
const months = ['Jan', 'March', 'April', 'June'];
months.splice(1, 0, 'Feb');
// inserts at index 1
console.log(months);
// ["Jan", "Feb", "March", "April", "June"]
months.splice(4, 1, 'May');
// replaces 1 element at index 4
console.log(months);
// ["Jan", "Feb", "March", "April", "May"]
```

Check every items pass a criteria (**every**)

```
const arr = [1, 30, 39, 29, 10, 13];

console.log(arr.every(x => x < 40));

// true

console.log(arr.every(x => x < 30));

// false
```

Create new array which pass a criteria (**filter**)

```
const arr = [1, 30, 39, 29, 10, 13];

console.log(arr.filter(x => x < 30));

// [1, 29, 10, 13]

console.log(arr.filter(x => x < 20));

// [1, 10, 13]
```

Create new array (**map**)

The map() method creates a new array populated with the results of calling a provided function on every element in the calling array:

```
const array1 = [1, 4, 9, 16];

const array2 = array1.map(x => x * 2);

console.log(array2);

// [2, 8, 18, 32]
```

Find the first matching element (**find**)

```
const array1 = [1, 4, 9, 16];

const found = array1.find(x => x > 5);

console.log(found);

// 9
```

Find the index of first matching element (**findIndex**)

```
const array1 = [1, 4, 9, 16];

const foundIndex = array1.findIndex(x => x > 5);

console.log(foundIndex);

// 2
```

Apply a function to each element (**reduce**)

Apply a reducer function to each element to produce a single value:

```
const array1 = [1, 2, 3, 4];

const reducer = (accumulator, curValue) =>

                                   accumulator + curValue;

console.log(array1.reduce(reducer));

// 1 + 2 + 3 + 4 = 10

console.log(array1.reduce(reducer, 5));

// 5 + 1 + 2 + 3 + 4 = 15

// 5 is the initial value of an accumulator
```

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions
- https://www.w3schools.com/js/
- https://www.tutorialsteacher.com/javascript/javascript-function