JavaScript Functions

Definition
Invocation
Arguments
Scope
Anonymous Functions
Closures

Functions

- Functions are the fundamental modular unit of JavaScript.
- Function objects are linked to **`Function.prototype`** (which is itself linked to **`Object.prototype`**).
- Functions can be stored in variables, objects, and arrays. Functions can be passed as arguments to functions, and functions can be returned from functions.
- function(a){…} —> define
- function() —> call

Function Literal

- Functions can be created using Literals
- var add = function (a, b) { return a + b;}
- A function literal can appear anywhere that an expression can appear.

Function Invocation

- There are four patterns of invocation in JavaScript: the method invocation pattern, the function invocation pattern, the constructor invocation pattern, and the apply invocation pattern

Method Invocation

- When a function is stored as a property of an object , it is called a method

```
var myObject = {
    value: 0;
    increment: function (inc) {
        this.value += typeof inc ===
'number' ? inc : 1;
} };
myObject.increment(  );
```

Function Invocation

- When a function is not the property of an object, then it is invoked as a function:

```
function add(a,b){return a+b}
var sum = add(3, 4);    // sum is 7
```

Constructor Invocation

- Functions that are intended to be used with the **`new`** prefix are called constructors.

```
var Quo = function (string) {
    this.status = string;
};
// Give all instances of Quo a public method
// called get_status.
Quo.prototype.get_status = function (  ) {
    return this.status;
};
// Make an instance of Quo.
var myQuo = new Quo("confused");
```

Arguments

- A bonus parameter that is available to functions when they are invoked is the **`arguments`** array.

- **`arguments`** is not really an array. It is an array-like object. **`arguments`** has a **`length`** property, but it lacks all of the array methods.

```
var sum = function (   ) {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
}
    return sum;
};
```

Scope

- Variables defined inside a function cannot be accessed from anywhere outside the function(Function scope), because the variable is defined only in the scope of the function.

  - a = 5 ;
    function(){b=5 ; console.log(a)};
    b ;

- *F*unction 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.

Anonymous Functions

- Such a function can be **anonymous**; it does not have to have a name.

- const square = function(number) { return number * number }

```
var x = square(4) // x gets the value 16
```

Nested Function & Closures

- You may nest a function within another function. The nested (inner) function is private to its containing (outer) function.

- It also forms a *closure*. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created.

- 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.

Example Closure

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // returns 13
b = addSquares(3, 4); // returns 25
c = addSquares(4, 5); // returns 41
```

- The inner function can be accessed only from statements in 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.

Example Closure

```
function makeAdder(x) {
    return function(y) {
        return x + y;
    };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

- In essence, `makeAdder` is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

- `add5` and `add10` are both closures. They share the same function body definition, but store different lexical environments. In `add5`'s lexical environment, `x` is 5, while in the lexical environment for `add10`, `x` is 10.