Session 6 JS

Mohamed Emary

December 10, 2024

1 Functions

Function is a block of code that can be called by name. The code inside a function is executed when the function is invoked. Functions are used to perform specific tasks.

Functions can take parameters and return values.

Syntax:

```
function function_name(parameters){
  // code to be executed

  // return value
}
```

Example:

```
function calc() {
  console.log(5 + 6);
}

calc() // 11
```

We can even call `calc()` before the function definition (hoisting).

Hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope before code execution.

```
calc() // 11

function calc() {
  console.log(5 + 6);
}
```

We can add our own parameters to the function.

```
function calc(a, b) {
  console.log(a + b);
}

calc(5, 6) // 11
calc(10, 20) // 30
calc(3) // NaN
```

The `calc(3)` will return `NaN` because `b` is `undefined`, so `3 + undefined` is `NaN`. To solve this issue we can use default values.

```
function calc(a, b = 0) {
  console.log(a + b);
}

calc(3) // 3 because b is 0
```

Default Function Value

Always make the parameter with the default value at the end, or give all the parameters a default value.

1.1 Function Expression

A function expression is a function that is assigned to a variable.

```
var fun = function (a, b) {
  console.log(a + b);
}

fun(5, 6) // 11
```

Function Expression Hoisting

Function expressions are not hoisted. The JS engine doesn't know anything about the function expression until it's called.

```
fun(5, 6) // Error

var fun = function (a, b) {
console.log(a + b);
}
```

1.2 Return

The `return` statement ends the function execution and specifies a value to be returned to the function caller.

`return` can be used in both function declaration and function expression.

Any code after `return` will not be seen or executed by the JS engine.

```
function fun(num1, num2) {
  return num1 * num2;
}
```

```
4
5  var result = fun(5, 6);
6  console.log(result); // 30
7
8  var fun1 = function (num1, num2) {
9    return num1 * num2;
10 }
11
12 var result1 = fun1(5, 6);
13 console.log(result1); // 30
```

1.3 Number of Parameters

If you try `console.log(1, 2, 3, 4, 5)` you will get `1 2 3 4 5`. This is because our `console.log()` function can take any number of parameters.

To allow a function to take any number of parameters we can use the `...` (rest) operator.

```
1  function fun(...num) {
2    console.log(num);
3  }
4
5  fun(1, 2, 3, 4, 5) // [1, 2, 3, 4, 5]
```

1.4 Function Inside Function

We can also have a function inside a function.

```
1  function outer() {
2    function inner() {
3      console.log('Inside fun1');
4    }
5
6    inner()
7  }
8
9
10 outer()
11 inner() // Error
```

If you try to call `inner()` outside of `outer()` you will get an error because `inner()` is not in the global scope.

If you define another function inside `inner()` then you can't access that function from outside `inner()`.

```
1  function outer() {
2    function inner() {
3      function insideInner() {
4        console.log('Inside insideInner');
5      }
6
7      insideInner() // Works
```

```
  }

  insideInner() // Error
  inner()
}
```

1.5 IIFE (Immediately Invoked Function Expression)

An IIFE is a function that runs directly after it's created.

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

It can also take parameters.

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

1.6 Function Constructor (Not Used)

We can also create a function using a function constructor.

Function constructor takes parameters as strings and the last parameter is the function body.

```
var fun = new Function('a', 'b', 'console.log(a + b)');
fun(5, 6) // 11
```

> ***This is just for your information. We don't use this method in real-world applications, as it has a very bad performance.***

1.7 Arrow Functions

Arrow functions allow us to write shorter function syntax.

```
var fun = () => 'Hello';
console.log(fun()); // Hello
```

If your function has multiple lines you need to use `{}`.

```
var fun = (x, y) => {
  var sum = x + y;
  console.log(sum);
}

fun(5, 6); // 11
```

If your function has only one parameter you can skip the `()` and the `return` keyword.

```
var fun = x => x * x;
var result = fun(5);
console.log(result); // 25
```

1.8 Callback Functions

A callback function is a function that takes another function as an argument.

```
function fun(callback) {
  callback();
}
```

Example:

```
function fun(funName, data) {
  funName(data);
}

function print(data) {
  console.log(data);
}

fun(print, 'Hello'); // Hello
```

This code will call the `print` function with the data `Hello`.

Why Callback Functions?

Callback functions are used to make sure that a function is not executed before another function has finished. For example calling an API and waiting for the response.

If JS faces a line of code that takes time to execute, it will not wait for it to finish and will continue executing the next line of code, so we use callback functions to run the code in the correct order.

There is other solutions to this problem like `Promises` and `async/await` which we will learn later.

1.9 `try`, `catch`, `finally`

`try` statement allows us to define a block of code to be tested for errors while it is being executed.

The advantage of `try` is that if an error occurs, the code will not stop, it will continue executing the next line of code.

`catch` statement allows us to define a block of code to be executed if an error occurs in the `try` block.

`finally` is an optional statement lets you execute code, after try and catch, regardless of the result.

Even if there is an error in the `catch` block, the `finally` block will be executed.

```
try {
  console.log(xyz);
} catch (e) {
  // console.error is similar to console.log but it's used for errors
  // to show the error with red highlight in the console
```

```
  console.error("Error: " + e);
} finally {
  console.log('Always executed');
}
```

This code will print `ReferenceError: xyz is not defined`, then `Always executed`.

2 Object

An object is a collection of key-value pairs. To define an object we use `{}`.

Objects are used to store multiple values of different types in a single variable.

```
var person = {
  name: 'Mohamed',
  age: 25,
  skills: ['HTML', 'CSS', 'JS'],
}

console.log(person); // prints the whole object
console.log(typeof person); // object
```

To access the object properties we can use `.` or `[]`.

```
console.log(person.name); // Mohamed
console.log(person['name']); // Mohamed
```

You can also add new properties to the object after defining it.

```
person.job = 'Developer';
console.log(person); // prints the object with the new property
```

To remove a property from an object we use the `delete` keyword.

```
delete person.job;
console.log(person); // prints the object without the job property
```

You can also define functions inside your object.

```
person.sayHello = function () {
  console.log('Hello, this is ' + person.name);
}

person.sayHello; // returns the function code
person.sayHello(); //Executes the functions: Hello, this is Mohamed
```

We can also define object inside an object, and access the inner object properties using `..`

```
person = {
  name: 'Mohamed',
  age: 25,
  skills: ['HTML', 'CSS', 'JS'],
  address: {
    city: 'Cairo',
    country: 'Egypt'
  }
```

```
}

console.log(person.address.city); // Cairo
```

As you see `address` is an object inside the `person` object.

2.1 Iterating Over Object Properties

To iterate over an object properties we use `for...in` loop.

```
var person = {
  name: 'Mohamed',
  age: 25,
  skills: ['HTML', 'CSS', 'JS'],
  address: {
    city: 'Cairo',
    country: 'Egypt'
  }
}

for (var key in person) {
  // Note: it doesn't print the inner object properties like city,
  ↪  country
  console.log(key); // name, age, skills, address
}

for (var key in person) {
  // name => undefined, age => undefined, ...
  console.log(key, " => ", person.key);

  // name => Mohamed, age => 25, ...
  console.log(key, " => ", person[key]);
}
```

The difference between using `person.key` and `person[key]` is that `person.key` will return `undefined` because there is no property called `key` in the object, while `person[key]` will return the value of the property that is stored in the `key` variable.

2.2 Assigning Object to Another Object

When you assign an object to another object, you are not copying the object, you are copying the memory reference to that object.

```
var person = {
  name: 'Mohamed',
  age: 25,
  skills: ['HTML', 'CSS', 'JS'],
}

var person2 = person;
person2.name = 'Ali';

```

```
console.log(person.name); // Ali
console.log(person2.name); // Ali
console.log(person1 == person2); // true
```

As you see, when we change the `name` property in `person2`, it also changes in `person`. This is because they are pointing to the same object in memory.

To solve this issue we can use the `Object.assign()` method.

```
var person = {
  name: 'Mohamed',
  age: 25,
  skills: ['HTML', 'CSS', 'JS'],
}

var person2 = Object.assign({}, person);
console.log(person == person2); // false

person2.name = 'Ali';

console.log(person.name); // Mohamed
console.log(person2.name); // Ali
```

The `console.log(person == person2)` statement returned `false` because although the properties are the same, they are stored in different memory locations.

Modifying object inside a function:

```
function changeName(obj) {
  obj.name = 'Ahmed';
}

changeName(person);
console.log(person.name); // Ahmed
```

3 Array

Arrays are used to store multiple values in a single variable. By logic these values are of the same type, but in JS you can store different types in the same array.

To define an array we use `[]`, and to get the length of the array we use `length` property.

```
var names = ['Mohamed', 'Ali', 'Ahmed'];
console.log(names); // ['Mohamed', 'Ali', 'Ahmed']
console.log(names.length); // 3
```

To add an element to the end of the array we use `push()` method, and to add an element to the beginning of the array we use `unshift()` method.

Both `push()`, `unshift()` can accept multiple parameters, for example `names.push('Omar', 'Khaled')`, or `names.unshift('Omar', 'Khaled')`.

To remove the last element from the array we use `pop()` method, and to remove the first element from the array we use `shift()` method.

```
names.push('Omar', 'Khaled');
console.log(names); // ['Mohamed', 'Ali', 'Ahmed', 'Omar', 'Khaled']

names.unshift('Tarek');
console.log(names); // ['Tarek', 'Mohamed', 'Ali', 'Ahmed', 'Omar',
↪  'Khaled']
```

To remove an element from the array we use `splice()` method.

`splice(start, deleteCount, item1, item2, ...)` will remove `deleteCount` elements starting from the `start` index, and will add `item1`, `item2`, ... at the `start` index.

```
names.splice(1, 2);
console.log(names); // ['Tarek', 'Ahmed', 'Omar', 'Khaled']

names.splice(1, 0, 'Ali', 'Mohamed'); // Add Ali, Mohamed at index 1
console.log(names); // ['Tarek', 'Ali', 'Mohamed', 'Ahmed', 'Omar',
↪  'Khaled']
```

To reverse the array we use `reverse()` method.

```
names.reverse();
console.log(names); // ['Khaled', 'Omar', 'Ahmed', 'Mohamed', 'Ali',
↪  'Tarek']
```

To sort the array we use `sort()` method.

```
var nums = [1, 50, 200, 5, 100, 10];
nums.sort();
console.log(nums); // [1, 10, 100, 200, 5, 50]
```

As you can see the array is sorted as strings, to sort it as numbers we can use a compare function.

```
nums.sort(function (a, b) {
  return a - b;
})

console.log(nums); // [1, 5, 10, 50, 100, 200]
```

What is happening here is that the `sort()` function inside the sort method subtracts `a` from `b`, if the result is negative it means that `a` is smaller than `b`, if the result is positive it means that `a` is greater than `b`, and if the result is zero it means that `a` is equal to `b`.

You can also reverse the sorting by subtracting `b` from `a`, so instead of `return a - b` you can use `return b - a`.

To loop over array elements use `.forEach()` method.

```
// Print all the elements in the array
nums.forEach(function (num) {
  console.log(num);
})
```

To filter the array elements use `.filter()` method.

```
// Filter numbers greater than 50
var filtered = nums.filter(function (num) {
  return num > 50;
})

console.log(filtered); // [200, 100]
```

`indexOf()` method returns the index of the first occurrence of the element in the array, if the element is not found it will return `-1`.

```
console.log(nums.indexOf(100)); // 4
console.log(nums.indexOf(500)); // -1
```

`lastIndexOf()` method returns the index of the last occurrence of the element in the array, if the element is not found it will return `-1`.

```
var nums = [1, 50, 200, 100, 5, 100, 10];
nums.push(100);
console.log(nums.lastIndexOf(100)); // 6
```