JavaScript

Laiba Imran

Functions

JavaScript functions are **defined** with the function keyword.
You can use a function **declaration** or a function **expression**.

```
function myFunction(a, b) {
  return a * b;
}

const x = function (a, b) {return a * b};
let z = x(4, 3);


const x =  (a, b) => {return a * b};
let z = x(4, 3);
```

Functions

JavaScript functions can be used as values:

```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3);
```

Can be used as expression:

```
function myFunction(a, b) {
  return a * b;
}

let x = myFunction(4, 3) * 2;
```

Functions as Objects

The typeof operator in JavaScript returns "function" for functions.

But, JavaScript functions can best be described as objects.

JavaScript functions have both properties and methods.

The arguments.length property returns the number of arguments received when the function was invoked:

```
function myFunction(a, b) {
  return arguments.length;
}
```

Functions as Objects

```
<!DOCTYPE html>
<html>
<body>

<p>The arguments.length is:</p>

<p id="demo"></p>

<script>
function myFunction(a, b) {
  return arguments.length;  //Output=2
}
document.getElementById("demo").innerHTML = myFunction(4, 3);
</script>

</body>
</html>
```

Functions as Objects

The toString() method returns the function as a string:

```
function myFunction(a, b) {
  return a * b;
}

let text = myFunction.toString();
```

Function Parameters

A JavaScript function does not perform any checking on parameter values (arguments).

JavaScript function definitions do not specify data types for parameters.

JavaScript functions do not perform type checking on the passed arguments.

JavaScript functions do not check the number of arguments received.

Default Parameters (ES6)

If a function is called with missing arguments (less than declared), the missing values are set to undefined.

Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter

Default Parameters

```
<p>Setting a default value to a function parameter.</p>
<p id="demo"></p>
<script>
function myFunction(x, y) {
  if (y === undefined) {  y = 2; }
  return x * y;
}
document.getElementById("demo").innerHTML = myFunction(4);
</script>
```

Default Parameters

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>
<script>
function myFunction(x, y = 10) {
  return x + y;    //15
}
document.getElementById("demo").innerHTML = myFunction(5);
</script>
</body>
</html>
```

Parameters Pass by Value

Primitive data types are passed by value.

```
function modifyPrimitive(num) {
    num = num + 10; // Modify the copy of the value
    console.log("Inside function:", num); // Output: 15
}
let originalNum = 5;
modifyPrimitive(originalNum);
console.log("Outside function:", originalNum); // Output: 5
```

Parameters Pass by Reference

Objects are passed by reference.

```
function modifyObject(obj) {
    obj.name = "Bob"; // Modify the property of the original object
    console.log("Inside function:", obj); // Output: { name: "Bob" }
}

let person = { name: "Alice" };
modifyObject(person);
console.log("Outside function:", person); // Output: { name: "Bob" }
```

The Arguments Object

JavaScript functions have a built-in object called the arguments object.
The argument object contains an array of the arguments used when the function was called (invoked).

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

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

The Arguments Object

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>

<script>
function findMax() {
  let max = -Infinity;
  for(let i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
document.getElementById("demo").innerHTML = findMax(4, 5, 6);
</script>
</body>
</html>
```

Function Rest Parameter

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

Function Rest Parameter

```
<!DOCTYPE html>
<html>
<body>
<p id="demo"></p>

<script>
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;   //15
}
let x = sum(1,2,3,4,5);
document.getElementById("demo").innerHTML = x;

</script>
</body>
</html>
```

Function Spread Parameter

The spread parameter (...) expands an iterable (like an array) into more elements.
This allows us to quickly copy all or parts of an existing array into another array

```
function sum(a, b, c) {
    return a + b + c;
}

const number = [1,2,3];
let x = sum(...number);
```

Function Spread Parameter

```
const one = [1,2,3];
const two = [4,5,6];

const x = one.concat(two);
Const x = [...one, ...two];
```

Function Object Methods

In JavaScript, functions are object methods.

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}

// This will return "John Doe":
person.fullName();
```

Function call()

The call() method is a predefined JavaScript method.

It can be used to invoke (call) a method with an owner object as the first argument (parameter).

With call(), an object can use a method belonging to another object.

```
const person = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
const person1 = {
  firstName:"John", lastName: "Doe"
}
const person2 = {
  firstName:"Mary", lastName: "Doe"
}

person.fullName.call(person1); //John Doe
```

Function call()

```
<p id="demo"></p>

<script>
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + "," +
country;
  }
}
const person1 = {
  firstName:"John", lastName: "Doe"
}
const person2 = {
  firstName:"Mary", lastName: "Doe"
}

document.getElementById("demo").innerHTML =
person.fullName.call(person1, "Oslo", "Norway");
</script>
```

Function apply()

The apply() method is similar to the call() method.
The difference is:

The call() method takes arguments separately.

The apply() method takes arguments as an array.

The apply() method is very handy if you want to use an array instead of an argument list.

Function apply()

```
<p id="demo"></p>
<script>
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city + ","
+ country;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}
document.getElementById("demo").innerHTML =
person.fullName.apply(person1, ["Oslo", "Norway"]);
</script>
```

JavaScript Classes

Use the keyword class to create a class.

Always add a method named constructor():

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
```

It is not an object, but rather a template for objects.

JavaScript Classes

```
<p id="demo"></p>
<script>
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);

document.getElementById("demo").innerHTML =
myCar1.name + " " + myCar2.name;
</script>
```

JavaScript Class Methods

```
<p id="demo"></p>
<script>
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age(x) {
    return x- this.year;
  }
}

let year = 2000;
const myCar = new Car("Ford", 2014);
document.getElementById("demo").innerHTML =
"My car is " + myCar.age(year) + " years old.";
</script>
```

JavaScript Class Inheritance

```
<p id="demo"></p>
<script>
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a ' + this.model;
  }
}
const myCar = new Model("Ford", "Mustang");
document.getElementById("demo").innerHTML = myCar.show();
</script>
```

JavaScript Class Getter Setter

```
<p id="demo"></p>

<script>
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  get cnam() {
    return this.carname;
  }
  set cnam(x) {
    this.carname = x;
  }
}
const myCar = new Car("Ford");
document.getElementById("demo").innerHTML = myCar.cnam;
</script>
```

JavaScript Function Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope (to the top of the current script or the current function).

In JavaScript, a variable can be declared after it has been used.
In other words; a variable can be used before it has been declared.

```
x = 5; // Assign 5 to x

let elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                         // Display x in the
element

var x; // Declare x
```

JavaScript Function Hoisting

JavaScript only hoists declarations, not initializations.

```
<p id="demo"></p>

<script>
var x = 5;  // Initialize x

elem = document.getElementById("demo");      // Find an element
elem.innerHTML = "x is " + x + " and y is " + y;  // Display x and y

var y = 7;  // Initialize y
</script>
```

JavaScript Class Hoisting

Unlike functions, and other JavaScript declarations, class declarations are not hoisted.
That means that you must declare a class before you can use it:

```
//You cannot use the class yet.
//myCar = new Car("Ford") will raise an error.

class Car {
  constructor(brand) {
    this.carname = brand;
  }
}

//Now you can use the class:
const myCar = new Car("Ford")
```