Functions

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output. To use a function, you must define it somewhere in the scope from which you wish to call it.

See also the exhaustive reference chapter about JavaScript functions to get to know the details.

Defining functions

Function declarations

A **function definition** (also called a **function declaration**, or **function statement**) consists of the `function` keyword, followed by:

- The name of the function.
- A list of parameters to the function, enclosed in parentheses and separated by commas.
- The JavaScript statements that define the function, enclosed in curly braces, `{ /* … */ }`.

For example, the following code defines a function named `square`:

JS

```
function square(number) {
  return number * number;
}
```

The function `square` takes one parameter, called `number`. The function consists of one statement that says to return the parameter of the function (that is, `number`) multiplied by itself. The `return` statement specifies the value returned by the function, which is `number * number`.

Parameters are essentially passed to functions **by value** — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, **the change is not reflected globally or in the code which called that function**.

When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function, as shown in the following example:

JS

```
function myFunc(theObject) {
  theObject.make = "Toyota";
}

const myCar = {
  make: "Honda",
  model: "Accord",
  year: 1998,
};

console.log(myCar.make); // "Honda"
myFunc(myCar);
console.log(myCar.make); // "Toyota"
```

When you pass an array as a parameter, if the function changes any of the array's values, that change is visible outside the function, as shown in the following example:

JS

```
function myFunc(theArr) {
  theArr[0] = 30;
}

const arr = [45];
```

```
console.log(arr[0]); // 45
myFunc(arr);
console.log(arr[0]); // 30
```

Function declarations and expressions can be nested, which forms a *scope chain*. For example:

JS

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

See function scopes and closures for more information.

Function expressions

While the function declaration above is syntactically a statement, functions can also be created by a function expression.

Such a function can be **anonymous**; it does not have to have a name. For example, the function `square` could have been defined as:

JS

```
const square = function (number) {
  return number * number;
};

console.log(square(4)); // 16
```

However, a name *can* be provided with a function expression. Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces:

JS

```
const factorial = function fac(n) {
  return n < 2 ? 1 : n * fac(n - 1);
};

console.log(factorial(3)); // 6
```

Function expressions are convenient when passing a function as an argument to another function. The following example defines a `map` function that should receive a function as first argument and an array as second argument. Then, it is called with a function defined by a function expression:

JS

```
function map(f, a) {
  const result = new Array(a.length);
  for (let i = 0; i < a.length; i++) {
    result[i] = f(a[i]);
  }
  return result;
}

const numbers = [0, 1, 2, 5, 10];
const cubedNumbers = map(function (x) {
  return x * x * x;
}, numbers);
console.log(cubedNumbers); // [0, 1, 8, 125, 1000]
```

In JavaScript, a function can be defined based on a condition. For example, the following function definition defines `myFunc` only if `num` equals `0`:

JS

```
let myFunc;
if (num === 0) {
  myFunc = function (theObject) {
    theObject.make = "Toyota";
  };
}
```

In addition to defining functions as described here, you can also use the `Function` constructor to create functions from a string at runtime, much like `eval()`.

A **method** is a function that is a property of an object. Read more about objects and methods in Working with objects.

Calling functions

Defining a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function `square`, you could call it as follows:

JS

```
square(5);
```

The preceding statement calls the function with an argument of `5`. The function executes its statements and returns the value `25`.

Functions must be *in scope* when they are called, but the function declaration can be hoisted (appear below the call in the code). The scope of a function declaration is the function in which it is declared (or the entire program, if it is declared at the top level).

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function. The `showProps()` function (defined in Working with objects) is an example of a function that takes an object as an argument.

A function can call itself. For example, here is a function that computes factorials recursively:

JS

```
function factorial(n) {
  if (n === 0 || n === 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
```

```
  }
}
```

You could then compute the factorials of `1` through `5` as follows:

JS

```
console.log(factorial(1)); // 1
console.log(factorial(2)); // 2
console.log(factorial(3)); // 6
console.log(factorial(4)); // 24
console.log(factorial(5)); // 120
```

There are other ways to call functions. There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime.

It turns out that *functions are themselves objects* — and in turn, these objects have methods. (See the `Function` object.) The `call()` and `apply()` methods can be used to achieve this goal.

Function hoisting

Consider the example below:

JS

```
console.log(square(5)); // 25

function square(n) {
  return n * n;
}
```

This code runs without any error, despite the `square()` function being called before it's declared. This is because the JavaScript interpreter hoists the entire function declaration to the top of the current scope, so the code above is equivalent to:

JS

```
// All function declarations are effectively at the top of the scope
function square(n) {
  return n * n;
}

console.log(square(5)); // 25
```

Function hoisting only works with function *declarations* — not with function *expressions*. The following code will not work:

JS

```
console.log(square(5)); // ReferenceError: Cannot access 'square'
before initialization
const square = function (n) {
  return n * n;
};
```

Recursion

A function can refer to and call itself. It can be referred to either by the function expression or declaration's name, or via any in-scope variable that refers to the function object. For example, consider the following function definition:

JS

```
const foo = function bar() {
  // statements go here
};
```

Within the function body, you can refer to the function itself either as `bar` or `foo`, and call itself using `bar()` or `foo()`.

A function that calls itself is called a *recursive function*. In some ways, recursion is analogous to a loop. Both execute the same code multiple times, and both require a condition (to avoid an infinite loop, or rather, infinite recursion in this case).

For example, consider the following loop:

JS

```
let x = 0;
// "x < 10" is the loop condition
while (x < 10) {
  // do stuff
  x++;
}
```

It can be converted into a recursive function declaration, followed by a call to that function:

JS

```
function loop(x) {
  // "x >= 10" is the exit condition (equivalent to "!(x < 10)")
  if (x >= 10) {
    return;
  }
  // do stuff
  loop(x + 1); // the recursive call
}
loop(0);
```

However, some algorithms cannot be simple iterative loops. For example, getting all the nodes of a tree structure (such as the DOM) is easier via recursion:

JS

```
function walkTree(node) {
  if (node === null) {
    return;
  }
  // do something with node
  for (let i = 0; i < node.childNodes.length; i++) {
    walkTree(node.childNodes[i]);
  }
}
```

Compared to the function `loop`, each recursive call itself makes many recursive calls here.

It is possible to convert any recursive algorithm to a non-recursive one, but the logic is often much more complex, and doing so requires the use of a stack.

In fact, recursion itself uses a stack: the function stack. The stack-like behavior can be seen in the following example:

JS

```
function foo(i) {
  if (i < 0) {
    return;
  }
  console.log(`begin: ${i}`);
  foo(i - 1);
  console.log(`end: ${i}`);
}
foo(3);

// Logs:
// begin: 3
// begin: 2
// begin: 1
// begin: 0
// end: 0
// end: 1
// end: 2
// end: 3
```

Immediately Invoked Function Expressions (IIFE)

An Immediately Invoked Function Expression (IIFE) is a code pattern that directly calls a function defined as an expression. It looks like this:

JS

```
(function () {
  // Do something
})();

const value = (function () {
  // Do something
  return someValue;
})();
```

Instead of saving the function in a variable, the function is immediately invoked. This is almost equivalent to just writing the function body, but there are a few unique benefits:

- It creates an extra scope of variables, which helps to confine variables to the place where they are useful.
- It is now an *expression* instead of a sequence of *statements*. This allows you to write complex computation logic when initializing variables.

For more information, see the IIFE glossary entry.

Function scopes and closures

Functions form a scope for variables—this means variables defined inside a function cannot be accessed from anywhere outside the function. The function scope inherits from all the upper scopes. For example, 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. On the other hand, the parent function (and any other parent scope) does *not* have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables in the inner function.

JS

```
// The following variables are defined in the global scope
const num1 = 20;
const num2 = 3;
const name = "Chamakh";

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}

console.log(multiply()); // 60

// A nested function example
function getScore() {
  const num1 = 2;
```

```
  const num2 = 3;

  function add() {
    return `${name} scored ${num1 + num2}`;
  }

  return add();
}

console.log(getScore()); // "Chamakh scored 5"
```

Closures

We also refer to the function body as a *closure*. A closure is any piece of source code (most commonly, a function) that refers to some variables, and the closure "remembers" these variables even when the scope in which these variables were declared has exited.

Closures are usually illustrated with nested functions to show that they remember variables beyond the lifetime of its parent scope; but in fact, nested functions are unnecessary. Technically speaking, all functions in JavaScript form closures—some just don't capture anything, and closures don't even have to be functions. The key ingredients for a *useful* closure are the following:

- A parent scope that defines some variables or functions. It should have a clear lifetime, which means it should finish execution at some point. Any scope that's not the global scope satisfies this requirement; this includes blocks, functions, modules, and more.

mdn web docs_

- The inner scope manages to survive beyond the lifetime of the parent scope. For example, it is saved to a variable that's defined outside the parent scope, or it's returned from the parent scope (if the parent scope is a function).
- Then, when you call the function outside of the parent scope, you can still access the variables or functions that were defined in the parent scope, even though the parent scope has finished execution.

The following is a typical example of a closure:

JS

```
// The outer function defines a variable called "name"
const pet = function (name) {
  const getName = function () {
    // The inner function has access to the "name" variable of the outer
function
    return name;
  };
  return getName; // Return the inner function, thereby exposing it to outer
scopes
};
const myPet = pet("Vivie");

console.log(myPet()); // "Vivie"
```

It can be much more complex than the code above. An object containing methods for manipulating the inner variables of the outer function can be returned.

JS

```
const createPet = function (name) {
  let sex;

  const pet = {
    // setName(newName) is equivalent to setName: function (newName)
    // in this context
    setName(newName) {
      name = newName;
    },

    getName() {
      return name;
    },

    getSex() {
      return sex;
    },

    setSex(newSex) {
      if (
        typeof newSex === "string" &&
        (newSex.toLowerCase() === "male" || newSex.toLowerCase() ===
"female")
      ) {
```

```
        sex = newSex;
      }
    },
  };

  return pet;
};

const pet = createPet("Vivie");
console.log(pet.getName()); // Vivie

pet.setName("Oliver");
pet.setSex("male");
console.log(pet.getSex()); // male
console.log(pet.getName()); // Oliver
```

In the code above, the `name` variable of the outer function is accessible to the inner functions, and there is no other way to access the inner variables except through the inner functions. The inner variables of the inner functions act as safe stores for the outer arguments and variables. They hold "persistent" and "encapsulated" data for the inner functions to work with. The functions do not even have to be assigned to a variable, or have a name.

JS

```
const getCode = (function () {
  const apiCode = "0]Eal(eh&2"; // A code we do not want outsiders to be
able to modify…

  return function () {
    return apiCode;
  };
})();

console.log(getCode()); // "0]Eal(eh&2"
```

In the code above, we use the IIFE pattern. Within this IIFE scope, two values exist: a variable `apiCode` and an unnamed function that gets returned and gets assigned to the variable `getCode`. `apiCode` is in the scope of the returned unnamed function but not in the scope of any other part of the program, so there is no way for reading the value of `apiCode` apart from via the `getCode` function.

Multiply-nested functions

Functions can be multiply-nested. For example:

- A function (`A`) contains a function (`B`), which itself contains a function (`C`).
- Both functions `B` and `C` form closures here. So, `B` can access `A`, and `C` can access `B`.
- In addition, since `C` can access `B` which can access `A`, `C` can also access `A`.

Thus, the closures can contain multiple scopes; they recursively contain the scope of the functions containing it. This is called *scope chaining*. Consider the following example:

JS

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

In this example, `C` accesses `B`'s `y` and `A`'s `x`. This can be done because:

1. `B` forms a closure including `A` (i.e., `B` can access `A`'s arguments and variables).
2. `C` forms a closure including `B`.
3. Because `C`'s closure includes `B` and `B`'s closure includes `A`, then `C`'s closure also includes `A`. This means `C` can access *both* `B` *and* `A`'s arguments and variables. In other words, `C` *chains* the scopes of `B` and `A`, *in that order*.

The reverse, however, is not true. `A` cannot access `C`, because `A` cannot access any argument or variable of `B`, which `C` is a variable of. Thus, `C` remains private to only `B`.

Name conflicts

When two arguments or variables in the scopes of a closure have the same name, there is a *name conflict*. More nested scopes take precedence. So, the innermost scope takes the highest precedence, while the outermost scope takes the lowest. This is the scope chain. The first on the chain is the innermost scope, and the last is the outermost scope. Consider the following:

JS

```
function outside() {
  const x = 5;
  function inside(x) {
    return x * 2;
  }
  return inside;
}

console.log(outside()(10)); // 20 (instead of 10)
```

The name conflict happens at the statement `return x * 2` and is between `inside`'s parameter `x` and `outside`'s variable `x`. The scope chain here is `inside` => `outside` => global object. Therefore, `inside`'s `x` takes precedences over `outside`'s `x`, and `20` (`inside`'s `x`) is returned instead of `10` (`outside`'s `x`).

Using the 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:

JS

```
arguments[i];
```

where `i` is the ordinal number of the argument, starting at `0`. So, the first argument passed to a function would be `arguments[0]`. The total number of

arguments is indicated by `arguments.length`.

Using the `arguments` object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will be passed to the function. You can use `arguments.length` to determine the number of arguments actually passed to the function, and then access each argument using the `arguments` object.

For example, consider a function that concatenates several strings. The only formal argument for the function is a string that specifies the characters that separate the items to concatenate. The function is defined as follows:

JS

```
function myConcat(separator) {
  let result = ""; // initialize list
  // iterate through arguments
  for (let i = 1; i < arguments.length; i++) {
    result += arguments[i] + separator;
  }
  return result;
}
```

You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

JS

```
console.log(myConcat(", ", "red", "orange", "blue"));
// "red, orange, blue, "

console.log(myConcat("; ", "elephant", "giraffe", "lion", "cheetah"));
// "elephant; giraffe; lion; cheetah; "

console.log(myConcat(". ", "sage", "basil", "oregano", "pepper",
"parsley"));
// "sage. basil. oregano. pepper. parsley. "
```

Note: The `arguments` variable is "array-like", but not an array. It is array-like in that it has a numbered index and a `length` property. However, it

> does *not* possess all of the array-manipulation methods.

See the `Function` object in the JavaScript reference for more information.

Function parameters

There are two special kinds of parameter syntax: *default parameters* and *rest parameters*.

Default parameters

In JavaScript, parameters of functions default to `undefined`. However, in some situations it might be useful to set a different default value. This is exactly what default parameters do.

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`.

In the following example, if no value is provided for `b`, its value would be `undefined` when evaluating `a*b`, and a call to `multiply` would normally have returned `NaN`. However, this is prevented by the second line in this example:

```
JS
function multiply(a, b) {
  b = typeof b !== "undefined" ? b : 1;
  return a * b;
}

console.log(multiply(5)); // 5
```

With *default parameters*, a manual check in the function body is no longer necessary. You can put `1` as the default value for `b` in the function head:

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

```
console.log(multiply(5)); // 5
```

For more details, see default parameters in the reference.

Rest parameters

The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

In the following example, the function `multiply` uses *rest parameters* to collect arguments from the second one to the end. The function then multiplies these by the first argument.

JS

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

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

Arrow functions

An arrow function expression (also called a *fat arrow* to distinguish from a hypothetical `->` syntax in future JavaScript) has a shorter syntax compared to function expressions and does not have its own `this`, `arguments`, `super`, or `new.target`. Arrow functions are always anonymous.

Two factors influenced the introduction of arrow functions: *shorter functions* and *non-binding* of `this`.

Shorter functions

In some functional patterns, shorter functions are welcome. Compare:

JS

```
const a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

const a2 = a.map(function (s) {
  return s.length;
});

console.log(a2); // [8, 6, 7, 9]

const a3 = a.map((s) => s.length);

console.log(a3); // [8, 6, 7, 9]
```

No separate this

Until arrow functions, every new function defined its own `this` value (a new object in the case of a constructor, undefined in strict mode function calls, the base object if the function is called as an "object method", etc.). This proved to be less than ideal with an object-oriented style of programming.

JS

```
function Person() {
  // The Person() constructor defines `this` as itself.
  this.age = 0;

  setInterval(function growUp() {
    // In nonstrict mode, the growUp() function defines `this`
    // as the global object, which is different from the `this`
    // defined by the Person() constructor.
    this.age++;
  }, 1000);
}

const p = new Person();
```

In ECMAScript 3/5, this issue was fixed by assigning the value in `this` to a variable that could be closed over.

JS

```
function Person() {
  // Some choose `that` instead of `self`.
  // Choose one and be consistent.
```

```
  const self = this;
  self.age = 0;

  setInterval(function growUp() {
    // The callback refers to the `self` variable of which
    // the value is the expected object.
    self.age++;
  }, 1000);
}
```

Alternatively, a bound function could be created so that the proper `this` value would be passed to the `growUp()` function.

An arrow function does not have its own `this`; the `this` value of the enclosing execution context is used. Thus, in the following code, the `this` within the function that is passed to `setInterval` has the same value as `this` in the enclosing function:

JS

```
function Person() {
  this.age = 0;

  setInterval(() => {
    this.age++; // `this` properly refers to the person object
  }, 1000);
}

const p = new Person();
```