Functions

Functions are one of the fundamental building blocks in JavaScript. A function in JavaScript is 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.

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 brackets, { /* … */ }

For example:

```
function square(number) {
  return 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.

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:

```
const square = function (number) {
  return number * number;
}
const x = square(4); // x gets the value 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:

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

console.log(factorial(3))
```

Function expressions are convenient when passing a function as an argument to another function. The following example shows a map function that should receive a function as the first argument and an array as a second argument:

```
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;
}
```

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.

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:

```
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.

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

```
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:

```
const a = factorial(1); // a gets the value 1
const b = factorial(2); // b gets the value 2
const c = factorial(3); // c gets the value 6
const d = factorial(4); // d gets the value 24
const e = factorial(5); // e gets the value 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 varies.

Function hoisting

Consider the example below:

```
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:

```
// 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 code below will not work.

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

```
  return n * n;
}
```

Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, 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.

Scope and the function stack

Recursion

A function can refer to and call itself. There are three ways for a function to refer to itself:

1. The function's name
2. `arguments.callee`
3. An in-scope variable that refers to the function

For example, consider the following function definition:

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

Within the function body, the following are all equivalent:

1. `bar()`
2. `arguments.callee()`
3. `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).

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:

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

Nested functions and 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 an expression (most commonly, a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

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**.

To summarize:

- 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

```
function outside(x) {
  function inside(y) {
    return x + y;
```

```
  }
  return inside;
}
const fnInside = outside(3); // Think of it like: give me a function that adds 3 to
                                whatever you give it
const result = fnInside(5); // returns 8
const result1 = outside(3)(5); // returns 8
```

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.

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

outside()(10); // returns 20 instead of 10
```

Closures

Closures are one of the most powerful features of JavaScript. JavaScript allows for the **nesting of functions** and grants **the inner function full access to all the variables and functions defined inside the outer function** (and all other variables and functions that the outer function has access to).
However, the outer function does *not* have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function.

Also, since 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.

```
const pet = function (name) {   // The outer function defines a variable called "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');

myPet(); // Returns "Vivie"
```

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:

`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.

```
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":

```
// returns "red, orange, blue, "
myConcat(', ', 'red', 'orange', 'blue');

// returns "elephant; giraffe; lion; cheetah; "
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');
```

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.

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 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:

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

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:

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

multiply(5); // 5
```

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.

```
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 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:

```
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.

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:

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

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

const p = new Person();
```

Predefined functions

JavaScript has many top-level, built-in functions. Some of them are:

- `eval()` - The `eval()` method evaluates JavaScript code represented as a string.
- `parseFloat()` - The `parseFloat()` function parses a string argument and returns a floating point number.
- `parseInt()` - The parseInt() function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).
- `isNaN()` - The `isNaN()` function determines whether a value is NaN or not.
- `isFinite()` - The global `isFinite()` function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.

return statement

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

```
function getRectArea(width, height) {
  if (width > 0 && height > 0) {
    return width * height;
  }
  return 0;
}

console.log(getRectArea(3, 4));
// expected output: 12

console.log(getRectArea(-3, 4));
// expected output: 0
```

The expression whose value is to be returned. If omitted, undefined is returned instead.