Chapter 19: Functions

Functions in JavaScript provide organized, reusable code to perform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions, arguments, parameters, return statements and scope in JavaScript.

Section 19.1: Function Scoping

When you define a function, it creates a *scope*.

Everything defined within the function is not accessible by code outside the function. Only code within this scope can see the entities defined inside the scope.

```
function foo() {
  var a = 'hello';
  console.log(a);  // => 'hello'
}

console.log(a);  // reference error
```

Nested functions are possible in JavaScript and the same rules apply.

```
function foo() {
  var a = 'hello';

  function bar() {
    var b = 'world';
    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
  }

  console.log(a);  // => 'hello'
  console.log(b);  // reference error
}

console.log(a);  // reference error
console.log(b);  // reference error
```

When JavaScript tries to resolve a reference or variable, it starts looking for it in the current scope. If it cannot find that declaration in the current scope, it climbs up one scope to look for it. This process repeats until the declaration has been found. If the JavaScript parser reaches the global scope and still cannot find the reference, a reference error will be thrown.

```
var a = 'hello';

function foo() {
  var b = 'world';

  function bar() {
    var c = '!!';

    console.log(a);  // => 'hello'
    console.log(b);  // => 'world'
    console.log(c);  // => '!!'
    console.log(d);  // reference error
  }
```

```
}
```

This climbing behavior can also mean that one reference may "shadow" over a similarly named reference in the outer scope since it gets seen first.

```
var a = 'hello';

function foo() {
  var a = 'world';

  function bar() {
    console.log(a);  // => 'world'
  }
}
```

Version ≥ 6

The way JavaScript resolves scoping also applies to the `const` keyword. Declaring a variable with the `const` keyword implies that you are not allowed to reassign the value, but declaring it in a function will create a new scope and with that a new variable.

```
function foo() {
  const a = true;

  function bar() {
    const a = false;  // different variable
    console.log(a);   // false
  }

  const a = false;    // SyntaxError
  a = false;          // TypeError
  console.log(a);     // true
}
```

However, functions are not the only blocks that create a scope (if you are using `let` or `const`). `let` and `const` declarations have a scope of the nearest block statement. See here for a more detailed description.

Section 19.2: Currying

Currying is the transformation of a function of `n` arity or arguments into a sequence of `n` functions taking only one argument.

Use cases: When the values of some arguments are available before others, you can use currying to decompose a function into a series of functions that complete the work in stages, as each value arrives. This can be useful:

- When the value of an argument almost never changes (e.g., a conversion factor), but you need to maintain the flexibility of setting that value (rather than hard-coding it as a constant).
- When the result of a curried function is useful before the other curried functions have run.
- To validate the arrival of the functions in a specific sequence.

For example, the volume of a rectangular prism can be explained by a function of three factors: length (`l`), width (`w`), and height (`h`):

```
var prism = function(l, w, h) {
    return l * w * h;
}
```

A curried version of this function would look like:

```
function prism(l) {
    return function(w) {
        return function(h) {
            return l * w * h;
        }
    }
}
```

Version ≥ 6

```
// alternatively, with concise ECMAScript 6+ syntax:
var prism = l => w => h => l * w * h;
```

You can call these sequence of functions with `prism(2)(3)(5)`, which should evaluate to 30.

Without some extra machinery (like with libraries), currying is of limited syntactical flexibility in JavaScript (ES 5/6) due to the lack of placeholder values; thus, while you can use `var a = prism(2)(3)` to create a partially applied function, you cannot use `prism()(3)(5)`.

Section 19.3: Immediately Invoked Function Expressions

Sometimes you don't want to have your function accessible/stored as a variable. You can create an Immediately Invoked Function Expression (IIFE for short). These are essentially *self-executing anonymous functions*. They have access to the surrounding scope, but the function itself and any internal variables will be inaccessible from outside. An important thing to note about IIFE is that even if you name your function, IIFE are not hoisted like standard functions are and cannot be called by the function name they are declared with.

```
(function() {
   alert("I've run - but can't be run again because I'm immediately invoked at runtime,
          leaving behind only the result I generate");
}());
```

This is another way to write IIFE. Notice that the closing parenthesis before the semicolon was moved and placed right after the closing curly bracket:

```
(function() {
   alert("This is IIFE too.");
})();
```

You can easily pass parameters into an IIFE:

```
(function(message) {
   alert(message);
}("Hello World!"));
```

Additionally, you can return values to the surrounding scope:

```
var example = (function() {
   return 42;
}());
console.log(example); // => 42
```

If required it is possible to name an IIFE. While less often seen, this pattern has several advantages, such as providing a reference which can be used for a recursion and can make debugging simpler as the name is included in the callstack.

```
(function namedIIFE() {
    throw error; // We can now see the error thrown in 'namedIIFE()'
}());
```

While wrapping a function in parenthesis is the most common way to denote to the JavaScript parser to expect an expression, in places where an expression is already expected, the notation can be made more concise:

```
var a = function() { return 42 }();
console.log(a)  // => 42
```

Arrow version of immediately invoked function:

Version ≥ 6

```
(() => console.log("Hello!"))(); // => Hello!
```

Section 19.4: Named Functions

Functions can either be named or unnamed (anonymous functions):

```
var namedSum = function sum (a, b) { // named
    return a + b;
}

var anonSum = function (a, b) { // anonymous
    return a + b;
}

namedSum(1, 3);
anonSum(1, 3);
```

> 4
> 4

But their names are private to their own scope:

```
var sumTwoNumbers = function sum (a, b) {
    return a + b;
}

sum(1, 3);
```

> Uncaught ReferenceError: sum is not defined

Named functions differ from the anonymous functions in multiple scenarios:

- When you are debugging, the name of the function will appear in the error/stack trace
- Named functions are hoisted while anonymous functions are not
- Named functions and anonymous functions behave differently when handling recursion
- Depending on ECMAScript version, named and anonymous functions may treat the function `name` property differently

Named functions are hoisted

When using an anonymous function, the function can only be called after the line of declaration, whereas a named function can be called before declaration. Consider

```
foo();
var foo = function () { // using an anonymous function
    console.log('bar');
}
```

> Uncaught TypeError: foo is not a function

```
foo();
function foo () { // using a named function
    console.log('bar');
}
```

> bar

Named Functions in a recursive scenario

A recursive function can be defined as:

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

//you could call 'say' directly,
//but this way just illustrates the example
var sayHelloTimes = say;

sayHelloTimes(2);
```

> Hello!
> Hello!

What if somewhere in your code the original function binding gets redefined?

```
var say = function (times) {
    if (times > 0) {
        console.log('Hello!');

        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> Hello!
> Uncaught TypeError: say is not a function

This can be solved using a named function

```
// The outer variable can even have the same name as the function
// as they are contained in different scopes
var say = function say (times) {
    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it uses the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> Hello!
> Hello!

And as bonus, the named function can't be set to `undefined`, even from inside:

```
var say = function say (times) {
    // this does nothing
    say = undefined;

    if (times > 0) {
        console.log('Hello!');

        // this time, 'say' doesn't use the outer variable
        // it's using the named function
        say(times - 1);
    }
}

var sayHelloTimes = say;
say = "oops";

sayHelloTimes(2);
```

> Hello!
> Hello!

The `name` property of functions

Before ES6, named functions had their `name` properties set to their function names, and anonymous functions had their `name` properties set to the empty string.

Version ≤ 5

```
var foo = function () {}
console.log(foo.name); // outputs ''

function foo () {}
console.log(foo.name); // outputs 'foo'
```

Post ES6, named and unnamed functions both set their `name` properties:

Version ≥ 6

```
var foo = function () {}
console.log(foo.name); // outputs 'foo'

function foo () {}
console.log(foo.name); // outputs 'foo'

var foo = function bar () {}
console.log(foo.name); // outputs 'bar'
```

Section 19.5: Binding `this` and arguments

Version ≥ 5.1

When you take a reference to a method (a property which is a function) in JavaScript, it usually doesn't remember the object it was originally attached to. If the method needs to refer to that object as **`this`** it won't be able to, and calling it will probably cause a crash.

You can use the `.bind()` method on a function to create a wrapper that includes the value of **`this`** and any number of leading arguments.

```
var monitor = {
  threshold: 5,
  check: function(value) {
    if (value > this.threshold) {
      this.display("Value is too high!");
    }
  },
  display(message) {
    alert(message);
  }
};

monitor.check(7); // The value of `this` is implied by the method call syntax.


var badCheck = monitor.check;
badCheck(15); // The value of `this` is window object and this.threshold is undefined, so value >
this.threshold is false

var check = monitor.check.bind(monitor);
check(15); // This value of `this` was explicitly bound, the function works.

var check8 = monitor.check.bind(monitor, 8);
check8(); // We also bound the argument to `8` here. It can't be re-specified.
```

When not in strict mode, a function uses the global object (`window` in the browser) as **`this`**, unless the function is called as a method, bound, or called with the method `.call` syntax.

```
window.x = 12;
```

```
function example() {
  return this.x;
}

console.log(example()); // 12
```

In strict mode `this` is `undefined` by default

```
window.x = 12;

function example() {
  "use strict";
  return this.x;
}

console.log(example()); // Uncaught TypeError: Cannot read property 'x' of undefined(...)
```

Version ≥ 7

Bind Operator

The double colon **bind operator** can be used as a shortened syntax for the concept explained above:

```
var log = console.log.bind(console); // long version
const log = ::console.log; // short version

foo.bar.call(foo); // long version
foo::bar(); // short version

foo.bar.call(foo, arg1, arg2, arg3); // long version
foo::bar(arg1, arg2, arg3); // short version

foo.bar.apply(foo, args); // long version
foo::bar(...args); // short version
```

This syntax allows you to write normally, without worrying about binding `this` everywhere.

Binding console functions to variables

```
var log = console.log.bind(console);
```

Usage:

```
log('one', '2', 3, [4], {5: 5});
```

Output:

```
one 2 3 [4] Object {5: 5}
```

Why would you do that?

One use case can be when you have custom logger and you want to decide on runtime which one to use.

```
var logger = require('appLogger');

var log = logToServer ? logger.log : console.log.bind(console);
```

Section 19.6: Functions with an Unknown Number of Arguments (variadic functions)

To create a function which accepts an undetermined number of arguments, there are two methods depending on your environment.

Version ≤ 5

Whenever a function is called, it has an Array-like arguments object in its scope, containing all the arguments passed to the function. Indexing into or iterating over this will give access to the arguments, for example

```
function logSomeThings() {
    for (var i = 0; i < arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

logSomeThings('hello', 'world');
// logs "hello"
// logs "world"
```

Note that you can convert `arguments` to an actual Array if need-be; see: Converting Array-like Objects to Arrays

Version ≥ 6

From ES6, the function can be declared with its last parameter using the rest operator (`...`). This creates an Array which holds the arguments from that point onwards

```
function personLogsSomeThings(person, ...msg) {
   msg.forEach(arg => {
       console.log(person, 'says', arg);
   });
}

personLogsSomeThings('John', 'hello', 'world');
// logs "John says hello"
// logs "John says world"
```

Functions can also be called with similar way, the spread syntax

```
const logArguments = (...args) => console.log(args)
const list = [1, 2, 3]

logArguments('a', 'b', 'c', ...list)
// output: Array [ "a", "b", "c", 1, 2, 3 ]
```

This syntax can be used to insert arbitrary number of arguments to any position, and can be used with any iterable(`apply` accepts only array-like objects).

```
const logArguments = (...args) => console.log(args)
function* generateNumbers() {
  yield 6
  yield 5
  yield 4
}

logArguments('a', ...generateNumbers(), ...'pqr', 'b')
// output: Array [ "a", 6, 5, 4, "p", "q", "r", "b" ]
```

Section 19.7: Anonymous Function

Defining an Anonymous Function

When a function is defined, you often give it a name and then invoke it using that name, like so:

```
foo();

function foo(){
    // ...
}
```

When you define a function this way, the JavaScript runtime stores your function in memory and then creates a reference to that function, using the name you've assigned it. That name is then accessible within the current scope. This can be a very convenient way to create a function, but JavaScript does not require you to assign a name to a function. The following is also perfectly legal:

```
function() {
    // ...
}
```

When a function is defined without a name, it's known as an anonymous function. The function is stored in memory, but the runtime doesn't automatically create a reference to it for you. At first glance, it may appear as if such a thing would have no use, but there are several scenarios where anonymous functions are very convenient.

Assigning an Anonymous Function to a Variable

A very common use of anonymous functions is to assign them to a variable:

```
var foo = function(){ /*...*/ };

foo();
```

This use of anonymous functions is covered in more detail in Functions as a variable

Supplying an Anonymous Function as a Parameter to Another Function

Some functions may accept a reference to a function as a parameter. These are sometimes referred to as "dependency injections" or "callbacks", because it allows the function your calling to "call back" to your code, giving you an opportunity to change the way the called function behaves. For example, the Array object's map function allows you to iterate over each element of an array, then build a new array by applying a transform function to each element.

```
var nums = [0,1,2];
var doubledNums = nums.map( function(element){ return element * 2; } ); // [0,2,4]
```

It would be tedious, sloppy and unnecessary to create a named function, which would clutter your scope with a function only needed in this one place and break the natural flow and reading of your code (a colleague would have to leave this code to find your function to understand what's going on).

Returning an Anonymous Function From Another Function

Sometimes it's useful to return a function as the result of another function. For example:

```
var hash = getHashFunction( 'sha1' );
```

```
var hashValue = hash( 'Secret Value' );

function getHashFunction( algorithm ){

    if ( algorithm === 'sha1' ) return function( value ){ /*...*/ };
    else if ( algorithm === 'md5' ) return function( value ){ /*...*/ };

}
```

Immediately Invoking an Anonymous Function

Unlike many other languages, scoping in JavaScript is function-level, not block-level. (See Function Scoping). In some cases, however, it's necessary to create a new scope. For example, it's common to create a new scope when adding code via a `<script>` tag, rather than allowing variable names to be defined in the global scope (which runs the risk of other scripts colliding with your variable names). A common method to handle this situation is to define a new anonymous function and then immediately invoke it, safely hiding you variables within the scope of the anonymous function and without making your code accessible to third-parties via a leaked function name. For example:

```
<!-- My Script -->
<script>
function initialize(){
    // foo is safely hidden within initialize, but...
    var foo = '';
}

// ...my initialize function is now accessible from global scope.
// There is a risk someone could call it again, probably by accident.
initialize();
</script>

<script>
// Using an anonymous function, and then immediately
// invoking it, hides my foo variable and guarantees
// no one else can call it a second time.
(function(){
    var foo = '';
}()) // <--- the parentheses invokes the function immediately
</script>
```

Self-Referential Anonymous Functions

Sometimes it's useful for an anonymous function to be able to refer to itself. For example, the function may need to recursively call itself or add properties to itself. If the function is anonymous, though, this can be very difficult as it requires knowledge of the variable that the function has been assigned to. This is the less than ideal solution:

```
var foo = function(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if (callAgain === true) foo(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
```

```
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Bad.
```

The intent here was for the anonymous function to recursively call itself, but when the value of foo changes, you end up with a potentially difficult to trace bug.

Instead, we can give the anonymous function a reference to itself by giving it a private name, like so:

```
var foo = function myself(callAgain){
    console.log( 'Whassup?' );
    // Less than ideal... we're dependent on a variable reference...
    if (callAgain === true) myself(false);
};

foo(true);

// Console Output:
// Whassup?
// Whassup?

// Assign bar to the original function, and assign foo to another function.
var bar = foo;
foo = function(){
    console.log('Bad.')
};

bar(true);

// Console Output:
// Whassup?
// Whassup?
```

Note that the function name is scoped to itself. The name has not leaked into the outer scope:

```
myself(false); // ReferenceError: myself is not defined
```

This technique is especially useful when dealing with recursive anonymous functions as callback parameters:

Version ≥ 5

```
// Calculate the Fibonacci value for each number in an array:
var fib = false,
    result = [1,2,3,4,5,6,7,8].map(
        function fib(n){
            return ( n <= 2 ) ? 1 : fib( n - 1 ) + fib( n - 2 );
        });
// result = [1, 1, 2, 3, 5, 8, 13, 21]
// fib = false (the anonymous function name did not overwrite our fib variable)
```

Section 19.8: Default parameters

Before ECMAScript 2015 (ES6), a parameter's default value could be assigned in the following way:

```
function printMsg(msg) {
```

```
  msg = typeof msg !== 'undefined' ? // if a value was provided
        msg :                        // then, use that value in the reassignment
        'Default value for msg.';    // else, assign a default value
  console.log(msg);
}
```

ES6 provided a new syntax where the condition and reassignment depicted above is no longer necessary:

Version ≥ 6

```
function printMsg(msg='Default value for msg.') {
    console.log(msg);
}
```

```
printMsg(); // -> "Default value for msg."
printMsg(undefined); // -> "Default value for msg."
printMsg('Now my msg in different!'); // -> "Now my msg in different!"
```

This also shows that if a parameter is missing when the function is invoked, its value is kept as `undefined`, as it can be confirmed by explicitly providing it in the following example (using an arrow function):

Version ≥ 6

```
let param_check = (p = 'str') => console.log(p + ' is of type: ' + typeof p);

param_check(); // -> "str is of type: string"
param_check(undefined); // -> "str is of type: string"

param_check(1); // -> "1 is of type: number"
param_check(this); // -> "[object Window] is of type: object"
```

Functions/variables as default values and reusing parameters

The default parameters' values are not restricted to numbers, strings or simple objects. A function can also be set as the default value `callback = function(){}`:

Version ≥ 6

```
function foo(callback = function(){ console.log('default'); }) {
    callback();
}

foo(function (){
    console.log('custom');
});
// custom

foo();
//default
```

There are certain characteristics of the operations that can be performed through default values:

- A previously declared parameter can be reused as a default value for the upcoming parameters' values.
- Inline operations are allowed when assigning a default value to a parameter.
- Variables existing in the same scope of the function being declared can be used in its default values.
- Functions can be invoked in order to provide their return value into a default value.

Version ≥ 6

```
let zero = 0;
function multiply(x) { return x * 2;}

function add(a = 1 + zero, b = a, c = b + a, d = multiply(c)) {
    console.log((a + b + c), d);
```

```
}

add(1);                 // 4, 4
add(3);                 // 12, 12
add(2, 7);              // 18, 18
add(1, 2, 5);           // 8, 10
add(1, 2, 5, 10);       // 8, 20
```

Reusing the function's return value in a new invocation's default value:

Version ≥ 6

```
let array = [1]; // meaningless: this will be overshadowed in the function's scope
function add(value, array = []) {
  array.push(value);
  return array;
}
add(5);         // [5]
add(6);         // [6], not [5, 6]
add(6, add(5)); // [5, 6]
```

`arguments` value and length when lacking parameters in invocation

The `arguments` array object only retains the parameters whose values are not default, i.e. those that are explicitly provided when the function is invoked:

Version ≥ 6

```
function foo(a = 1, b = a + 1) {
    console.info(arguments.length, arguments);
    console.log(a,b);
}

foo();        // info: 0 >> []     | log: 1, 2
foo(4);       // info: 1 >> [4]    | log: 4, 5
foo(5, 6);    // info: 2 >> [5, 6] | log: 5, 6
```

Section 19.9: Call and apply

Functions have two built-in methods that allow the programmer to supply arguments and the **`this`** variable differently: `call` and `apply`.

This is useful, because functions that operate on one object (the object that they are a property of) can be repurposed to operate on another, compatible object. Additionally, arguments can be given in one shot as arrays, similar to the spread (`...`) operator in ES6.

```
let obj = {
    a: 1,
    b: 2,
    set: function (a, b) {
        this.a = a;
        this.b = b;
    }
};

obj.set(3, 7); // normal syntax
obj.set.call(obj, 3, 7); // equivalent to the above
obj.set.apply(obj, [3, 7]); // equivalent to the above; note that an array is used

console.log(obj); // prints { a: 3, b: 5 }

let myObj = {};
myObj.set(5, 4); // fails; myObj has no `set` property
```

```
obj.set.call(myObj, 5, 4); // success; `this` in set() is re-routed to myObj instead of obj
obj.set.apply(myObj, [5, 4]); // same as above; note the array

console.log(myObj); // prints { a: 3, b: 5 }
```

Version ≥ 5

ECMAScript 5 introduced another method called **`bind()`** in addition to `call()` and `apply()` to explicitly set **`this`** value of the function to specific object.

It behaves quite differently than the other two. The first argument to `bind()` is the **`this`** value for the new function. All other arguments represent named parameters that should be permanently set in the new function.

```
function showName(label) {
    console.log(label + ":" + this.name);
}
var student1 = {
     name: "Ravi"
};
var student2 = {
     name: "Vinod"
};

// create a function just for student1
var showNameStudent1 = showName.bind(student1);
showNameStudent1("student1"); // outputs "student1:Ravi"

// create a function just for student2
var showNameStudent2 = showName.bind(student2, "student2");
showNameStudent2(); // outputs "student2:Vinod"

// attaching a method to an object doesn't change `this` value of that method.
student2.sayName = showNameStudent1;
student2.sayName("student2"); // outputs "student2:Ravi"
```

Section 19.10: Partial Application

Similar to currying, partial application is used to reduce the number of arguments passed to a function. Unlike currying, the number need not go down by one.

Example:

This function ...

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

... can be used to create another function that will always multiply by 2 and then add 10 to the passed value;

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

function factory(b, c) {
    return reversedMultiplyThenAdd.bind(null, c, b);
}

var multiplyTwoThenAddTen = factory(2, 10);
```

```
multiplyTwoThenAddTen(10); // 30
```

The "application" part of partial application simply means fixing parameters of a function.

Section 19.11: Passing arguments by reference or value

In JavaScript all arguments are passed by value. When a function assigns a new value to an argument variable, that change will not be visible to the caller:

```
var obj = {a: 2};
function myfunc(arg){
    arg = {a: 5}; // Note the assignment is to the parameter variable itself
}
myfunc(obj);
console.log(obj.a); // 2
```

However, changes made to (nested) properties *of* such arguments, will be visible to the caller:

```
var obj = {a: 2};
function myfunc(arg){
    arg.a = 5; // assignment to a property of the argument
}
myfunc(obj);
console.log(obj.a); // 5
```

This can be seen as a *call by reference*: although a function cannot change the caller's object by assigning a new value to it, it could *mutate* the caller's object.

As primitive valued arguments, like numbers or strings, are immutable, there is no way for a function to mutate them:

```
var s = 'say';
function myfunc(arg){
    arg += ' hello'; // assignment to the parameter variable itself
}
myfunc(s);
console.log(s); // 'say'
```

When a function wants to mutate an object passed as argument, but does not want to actually mutate the caller's object, the argument variable should be reassigned:

Version ≥ 6

```
var obj = {a: 2, b: 3};
function myfunc(arg){
    arg = Object.assign({}, arg); // assignment to argument variable, shallow copy
    arg.a = 5;
}
myfunc(obj);
console.log(obj.a); // 2
```

As an alternative to in-place mutation of an argument, functions can create a new value, based on the argument, and return it. The caller can then assign it, even to the original variable that was passed as argument:

```
var a = 2;
function myfunc(arg){
    arg++;
    return arg;
```

```
}
a = myfunc(a);
console.log(obj.a); // 3
```

Section 19.12: Function Arguments, "arguments" object, rest and spread parameters

Functions can take inputs in form of variables that can be used and assigned inside their own scope. The following function takes two numeric values and returns their sum:

```
function addition (argument1, argument2){
    return argument1 + argument2;
}

console.log(addition(2, 3)); // -> 5
```

`arguments` object

The `arguments` object contains all the function's parameters that contain a non-default value. It can also be used even if the parameters are not explicitly declared:

```
(function() { console.log(arguments) })(0,'str', [2,{3}]) // -> [0, "str", Array[2]]
```

Although when printing `arguments` the output resembles an Array, it is in fact an object:

```
(function() { console.log(typeof arguments) })(); // -> object
```

Rest parameters: `function (...parm) {}`

In ES6, the `...` syntax when used in the declaration of a function's parameters transforms the variable to its right into a single object containing all the remaining parameters provided after the declared ones. This allows the function to be invoked with an unlimited number of arguments, which will become part of this variable:

```
(function(a, ...b){console.log(typeof b+': '+b[0]+b[1]+b[2]) })(0,1,'2',[3],{i:4});
// -> object: 123
```

Spread parameters: `function_name(...varb);`

In ES6, the `...` syntax can also be used when invoking a function by placing an object/variable to its right. This allows that object's elements to be passed into that function as a single object:

```
let nums = [2,42,-1];
console.log(...['a','b','c'], Math.max(...nums)); // -> a b c 42
```

Section 19.13: Function Composition

Composing multiple functions into one is a functional programming common practice;

composition makes a pipeline through which our data will transit and get modified simply working on the function-composition (just like snapping pieces of a track together)...

you start out with some single responsibility functions:

Version ≥ 6

```
const capitalize = x => x.replace(/^\w/, m => m.toUpperCase());
const sign = x => x + ',\nmade with love';
```

and easily create a transformation track:

Version ≥ 6

```
const formatText = compose(capitalize, sign);

formatText('this is an example')
//This is an example,
//made with love
```

N.B. Composition is achieved through a utility function usually called `compose` as in our example.

Implementation of `compose` are present in many JavaScript utility libraries (lodash, rambda, etc.) but you can also start out with a simple implementation such as:

Version ≥ 6

```
const compose = (...funs) =>
  x =>
  funs.reduce((ac, f) => f(ac), x);
```

Section 19.14: Get the name of a function object

Version ≥ 6

ES6:

```
myFunction.name
```

Explanation on MDN. As of 2015 works in Node.js and all major browsers except IE.

Version ≥ 5

ES5:

If you have a reference to the function, you can do:

```
function functionName( func )
{
    // Match:
    // - ^           the beginning of the string
    // - function    the word 'function'
    // - \s+         at least some white space
    // - ([\w\$]+)   capture one or more valid JavaScript identifier characters
    // - \(          followed by an opening brace
    //
    var result = /^function\s+([\w\$]+)\(/.exec( func.toString() )

    return result ? result[1] : ''
}
```

Section 19.15: Recursive Function

A recursive function is simply a function, that would call itself.

```
function factorial (n) {
    if (n <= 1) {
        return 1;
    }
```

```
    return n * factorial(n - 1);
}
```

The above function shows a basic example of how to perform a recursive function to return a factorial.

Another example, would be to retrieve the sum of even numbers in an array.

```
function countEvenNumbers (arr) {
    // Sentinel value.  Recursion stops on empty array.
    if (arr.length < 1) {
        return 0;
    }
    // The shift() method removes the first element from an array
    // and returns that element. This method changes the length of the array.
    var value = arr.shift();

    // `value % 2 === 0` tests if the number is even or odd
    // If it's even we add one to the result of counting the remainder of
    // the array.  If it's odd, we add zero to it.
    return ((value % 2 === 0) ? 1 : 0) + countEvens(arr);
}
```

It is important that such functions make some sort of sentinel value check to avoid infinite loops. In the first example above, when `n` is less than or equal to 1, the recursion stops, allowing the result of each call to be returned back up the call stack.

Section 19.16: Using the Return Statement

The return statement can be a useful way to create output for a function. The return statement is especially useful if you do not know in which context the function will be used yet.

```
//An example function that will take a string as input and return
//the first character of the string.

function firstChar (stringIn){
    return stringIn.charAt(0);
}
```

Now to use this function, you need to put it in place of a variable somewhere else in your code:

Using the function result as an argument for another function:

```
console.log(firstChar("Hello world"));
```

Console output will be:

```
> H
```

The return statement ends the function

If we modify the function in the beginning, we can demonstrate that the return statement ends the function.

```
function firstChar (stringIn){
    console.log("The first action of the first char function");
    return stringIn.charAt(0);
    console.log("The last action of the first char function");
```

```
}
```

Running this function like so will look like this:

```
console.log(firstChar("JS"));
```

Console output:

```
> The first action of the first char function
> J
```

It will not print the message after the return statement, as the function has now been ended.

Return statement spanning multiple lines:

In JavaScript, you can normally split up a line of code into many lines for readability purposes or organization. This is valid JavaScript:

```
var
    name = "bob",
    age = 18;
```

When JavaScript sees an incomplete statement like `var` it looks to the next line to complete itself. However, if you make the same mistake with the `return` statement, you will not get what you expected.

```
return
    "Hi, my name is "+ name + ". " +
    "I'm "+ age + " years old.";
```

This code will return `undefined` because `return` by itself is a complete statement in JavaScript, so it will not look to the next line to complete itself. If you need to split up a `return` statement into multiple lines, put a value next to return before you split it up, like so.

```
return "Hi, my name is " + name + ". " +
    "I'm " + age + " years old.";
```

Section 19.17: Functions as a variable

A normal function declaration looks like this:

```
function foo(){
}
```

A function defined like this is accessible from anywhere within its context by its name. But sometimes it can be useful to treat function references like object references. For example, you can assign an object to a variable based on some set of conditions and then later retrieve a property from one or the other object:

```
var name = 'Cameron';
var spouse;

if ( name === 'Taylor' ) spouse = { name: 'Jordan' };
else if ( name === 'Cameron' ) spouse = { name: 'Casey' };

var spouseName = spouse.name;
```

In JavaScript, you can do the same thing with functions:

```
// Example 1
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = function(value){ /*...*/ };
else if ( hashAlgorithm === 'md5' ) hash = function(value){ /*...*/ };

hash('Fred');
```

In the example above, `hash` is a normal variable. It is assigned a reference to a function, after which the function it references can be invoked using parentheses, just like a normal function declaration.

The example above references anonymous functions... functions that do not have their own name. You can also use variables to refer to named functions. The example above could be rewritten like so:

```
// Example 2
var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = sha1Hash;
else if ( hashAlgorithm === 'md5' ) hash = md5Hash;

hash('Fred');

function md5Hash(value){
    // ...
}

function sha1Hash(value){
    // ...
}
```

Or, you can assign function references from object properties:

```
// Example 3
var hashAlgorithms = {
    sha1: function(value) { /**/ },
    md5: function(value) { /**/ }
};

var hashAlgorithm = 'sha1';
var hash;

if ( hashAlgorithm === 'sha1' ) hash = hashAlgorithms.sha1;
else if ( hashAlgorithm === 'md5' ) hash = hashAlgorithms.md5;

hash('Fred');
```

You can assign the reference to a function held by one variable to another by omitting the parentheses. This can result in an easy-to-make mistake: attempting to assign the return value of a function to another variable, but accidentally assigning the reference to the function.

```
// Example 4
var a = getValue;
var b = a; // b is now a reference to getValue.
var c = b(); // b is invoked, so c now holds the value returned by getValue (41)
```

```
function getValue(){
    return 41;
}
```

A reference to a function is like any other value. As you've seen, a reference can be assigned to a variable, and that variable's reference value can be subsequently assigned to other variables. You can pass around references to functions like any other value, including passing a reference to a function as the return value of another function. For example:

```
// Example 5
// getHashingFunction returns a function, which is assigned
// to hash for later use:
var hash = getHashingFunction( 'sha1' );
// ...
hash('Fred');


// return the function corresponding to the given algorithmName
function getHashingFunction( algorithmName ){
    // return a reference to an anonymous function
    if (algorithmName === 'sha1') return function(value){ /**/ };
    // return a reference to a declared function
    else if (algorithmName === 'md5') return md5;
}

function md5Hash(value){
    // ...
}
```

You don't need to assign a function reference to a variable in order to invoke it. This example, building off example 5, will call getHashingFunction and then immediately invoke the returned function and pass its return value to hashedValue.

```
// Example 6
var hashedValue = getHashingFunction( 'sha1' )( 'Fred' );
```

A Note on Hoisting

Keep in mind that, unlike normal function declarations, variables that reference functions are not "hoisted". In example 2, the `md5Hash` and `sha1Hash` functions are defined at the bottom of the script, but are available everywhere immediately. No matter where you define a function, the interpreter "hoists" it to the top of its scope, making it immediately available. This is **not** the case for variable definitions, so code like the following will break:

```
var functionVariable;

hoistedFunction(); // works, because the function is "hoisted" to the top of its scope
functionVariable(); // error: undefined is not a function.

function hoistedFunction(){}
functionVariable = function(){};
```