6 Ways to Declare JavaScript Functions

Updated November 2, 2019

javascript function

26 Comments

A **function** is a parametric block of code defined once and called multiple times later. In JavaScript a function is composed and influenced by many components:

- JavaScript code that forms the function body
- The list of parameters
- The variables accessible from the lexical scope
- The returned value
- The context `this` when the function is invoked
- Named or an anonymous function
- The variable that holds the function object

- `arguments` object (or missing in an arrow function)

This post teaches you six approaches to declare JavaScript functions: the syntax, examples and common pitfalls. Moreover, you will understand when to use a specific function type in certain circumstances.

Table of Contents

1. Function declaration
    - 1.1 A regular function
    - 1.2 Difference from function expression
    - 1.3 Function declaration in conditionals
2. Function expression
    - 2.1 Named function expression
    - 2.2 Favor named function expression
3. Shorthand method definition
    - 3.1 Computed property names and methods
4. Arrow function
    - 4.1 Context transparency
    - 4.2 Short callbacks
5. Generator function
6. One more thing: new Function
7. At the end, which way is better?

1. Function declaration

> "**A function declaration** is made of `function` keyword, followed by an obligatory function name, a list of parameters in a pair of parenthesis `(para1, ..., paramN)` and a pair of curly braces `{...}` that delimits the body code."

An example of function declaration:

```
// function declaration
function isEven(num) {
  return num % 2 === 0;
}
isEven(24); // => true
isEven(11); // => false
```

`function isEven(num) {...}` is a function declaration that defines `isEven` function, which determines if a number is even.

The function declaration **creates a variable** in the current scope with the identifier equal to the function name. This variable holds the function object.

The function variable is **hoisted** up to the top of the current scope, which means that the function can be invoked before the declaration (see this chapter for more details).

The created function is **named**, which means that the `name` property of the function object holds its name. It is useful when viewing the call stack: in debugging or error messages reading.

Let's see these properties in an example:

```
// Hoisted variable
console.log(hello('Aliens')); // => 'Hello Aliens!'
// Named function
console.log(hello.name)       // => 'hello'
// Variable holds the function object
console.log(typeof hello);    // => 'function'
function hello(name) {
  return `Hello ${name}!`;
}
```

The function declaration `function hello(name) {...}` create a variable `hello` that is hoisted to the top of the current scope. `hello` variable holds the function object and `hello.name` contains the function name: `'hello'`.

1.1 A regular function

The function declaration matches for cases when a regular function is needed. Regular means that you declare the function once and later invoke it in many different places. This is the basic scenario:

```
function sum(a, b) {
  return a + b;
}
sum(5, 6);            // => 11
([3, 7]).reduce(sum) // => 10
```

Because the function declaration creates a variable in the current scope, alongside regular function calls, it is useful for recursion or detaching event listeners. Contrary to function expressions or arrow functions, that do not create a binding with the function variable by its name.

For example, to calculate recursively the factorial you have to access the function inside:

```
function factorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}
factorial(4); // => 24
```

Inside `factorial()` a recursive call is being made using the variable that holds the function: `factorial(n - 1)`.

It is possible to use a function expression and assign it to a regular variable, e.g. `const factorial = function(n) {...}`. But the function declaration `function factorial(n)` is compact (no need for `const` and `=`).

An important property of the function declaration is its hoisting mechanism. It allows using the function before the declaration in the same scope.

Hoisting is useful in some situations. For example, when you'd like to see how the function is called at the beginning of a script, without reading the function implementation. The function implementation can be located below in the file, so you may not even scroll there.

You can read more details about function declaration hoisting here.

1.2 Difference from function expression

It is easy to confuse the function declaration and function expression. They look very similar but produce functions with different properties.

An easy to remember rule: the *function declaration* in a statement always **starts** with the keyword `function`. Otherwise it's a *function expression* (see 2.).

The following sample is a function declaration where the statement **starts** with `function` keyword:

```
// Function declaration: starts with "function"
function isNil(value) {
  return value == null;
}
```

In case of function expressions the JavaScript statement **does not start** with `function` keyword (it is present somewhere in the middle of the statement code):

```
// Function expression: starts with "const"
const isTruthy = function(value) {
  return !!value;
};
// Function expression: an argument for .filter()
const numbers = ([1, false, 5]).filter(function(item) {
  return typeof item === 'number';
});
// Function expression (IIFE): starts with "("
(function messageFunction(message) {
  return message + ' World!';
})('Hello');
```

1.3 Function declaration in conditionals

Some JavaScript environments can throw a reference error when invoking a function whose declaration appears within blocks `{...}` of `if`, `for` or `while` statements.
Let's enable the strict mode and see what happens when a function is declared in a conditional:

```
(function() {
  'use strict';
  if (true) {
    function ok() {
      return 'true ok';
    }
  } else {
    function ok() {
      return 'false ok';
    }
  }
  console.log(typeof ok === 'undefined'); // => true
  console.log(ok()); // Throws "ReferenceError: ok is not defined"
})();
```

When calling `ok()`, JavaScript throws `ReferenceError: ok is not defined`, because the function declaration is inside a conditional block.

The function declaration in conditionals is allowed in non-strict mode, which makes it even more confusing.

As a general rule for these situations, when a function should be created by conditions - use a function expression. Let's see how it is possible:

```
(function() {
  'use strict';
  let ok;
  if (true) {
    ok = function() {
      return 'true ok';
    };
  } else {
    ok = function() {
      return 'false ok';
    };
  }
  console.log(typeof ok === 'function'); // => true
  console.log(ok()); // => 'true ok'
})();
```

Because the function is a regular object, assign it to a variable depending on the condition. Invoking `ok()` works fine, without errors.

2. Function expression

> "**A function expression** is determined by a `function` keyword, followed by an optional function name, a list of parameters in a pair of parenthesis `(para1, ..., paramN)` and a pair of curly braces `{ ... }` that delimits the body code."

Some samples of the function expression:

```
const count = function(array) { // Function expression
  return array.length;
}
const methods = {
  numbers: [1, 5, 8],
  sum: function() { // Function expression
    return this.numbers.reduce(function(acc, num) { // func. expressio
      return acc + num;
    });
  }
}
count([5, 7, 8]); // => 3
methods.sum();    // => 14
```

The function expression creates a function object that can be used in different situations:

- Assigned to a variable as an object `count = function(...) {...}`
- Create a method on an object `sum: function() {...}`
- Use the function as a callback `.reduce(function(...) {...})`

The function expression is the working horse in JavaScript. Usually, you deal with this type of function declaration, alongside the arrow function (if you prefer short syntax and lexical context).

2.1 Named function expression

A function is anonymous when it does not have a name (`name` property is an empty string `''`):

```
(
```

```
  function(variable) {return typeof variable; }
).name; // => ''
```

This is an anonymous function, which name is an empty string.

Sometimes the function name can be inferred. For example, when the anonymous is assigned to a variable:

```
const myFunctionVar = function(variable) {
  return typeof variable;
};
myFunctionVar.name; // => 'myFunctionVar'
```

The anonymous function name is `'myFunctionVar'`, because `myFunctionVar` variable name is used to infer the function name.

When the expression has the name specified, this is a **named function expression**. It has some additional properties compared to simple function expression:

- A named function is created, i.e. `name` property holds the function name
- Inside the function body a variable with the same name holds the function object

Let's use the above example, but set a name in the function expression:

```
const getType = function funName(variable) {
  console.log(typeof funName === 'function'); // => true
  return typeof variable;
}
console.log(getType(3));     // => 'number'
console.log(getType.name);   // => 'funName'

console.log(typeof funName); // => 'undefined'
```

`function funName(variable) {...}` is a named function expression. The variable `funName` is accessible within function scope, but not outside. Either way, the property `name` of the function object holds the name: `funName`.

2.2 Favor named function expression

When a function expression `const fun = function() {}` is assigned to a variable, some engines infer the function name from this variable. However, callbacks might be passed as anonymous function expressions, without storing into variables: so the engine *cannot determine its name*.

It is reasonable to favor named functions and avoid anonymous ones to gain benefits like:

- The error messages and call stacks show more detailed information when using the function names
- More comfortable debugging by reducing the number of *anonymous* stack names
- The function name says what the function does
- You can access the function inside its scope for recursive calls or detaching event listeners

3. Shorthand method definition

> "**Shorthand method definition** can be used in a method declaration on object literals and ES2015 classes. You can define them using a function name, followed by a list of parameters in a pair of parenthesis `(para1, ..., paramN)` and a pair of curly braces `{ ... }` that delimits the body statements."

The following example uses a shorthand method definition in an object literal:

```
const collection = {
  items: [],
  add(...items) {
    this.items.push(...items);
  },
  get(index) {
    return this.items[index];
  }
};
collection.add('C', 'Java', 'PHP');
collection.get(1) // => 'Java'
```

`add()` and `get()` methods in `collection` object are defined using short method definition. These methods are called as usual: `collection.add(...)` and `collection.get(...)`.

The short approach of method definition has several benefits over traditional property definition with a name, colon `:` and a function expression `add: function(...) {...}`:

- A shorter syntax is easier to understand
- Shorthand method definition creates a named function, contrary to a function expression. It is useful for debugging.

The `class` syntax requires method declarations in a short form:

```
class Star {
  constructor(name) {
    this.name = name;
  }
  getMessage(message) {
    return this.name + message;
  }
}
const sun = new Star('Sun');
sun.getMessage(' is shining') // => 'Sun is shining'
```

3.1 Computed property names and methods

ECMAScript 2015 adds a nice feature: computed property names in object literals and classes.
The computed properties use a slight different syntax `[methodName]() {...}`, so the method definition looks this way:

```
const addMethod = 'add',
  getMethod = 'get';
const collection = {
  items: [],
  [addMethod](...items) {
    this.items.push(...items);
  },
  [getMethod](index) {
    return this.items[index];
  }
};
```

```
collection[addMethod]('C', 'Java', 'PHP');
collection[getMethod](1) // => 'Java'
```

`[addMethod](...) {...}` and `[getMethod](...) {...}` are shorthand method declarations with computed property names.

4. Arrow function

> "**An arrow function** is defined using a pair of parenthesis that contains the list of parameters `(param1, param2, ..., paramN)`, followed by a fat arrow `=>` and a pair of curly braces `{...}` that delimits the body statements."

When the arrow function has only one parameter, the pair of parentheses can be omitted. When it contains a single statement, the curly braces can be omitted too.

Let's see the arrow function basic usage:

```
const absValue = (number) => {
  if (number < 0) {
    return -number;
  }
  return number;
}
absValue(-10); // => 10
absValue(5);   // => 5
```

`absValue` is an arrow function that calculates the absolute value of a number.

The function declared using a fat arrow has the following properties:

- The arrow function does not create its execution context, but takes it lexically (contrary to function expression or function declaration, which create own `this` depending on invocation)
- The arrow function is anonymous. However, the engine can infer its name from the variable holding the function.

- `arguments` object is not available in the arrow function (contrary to other declaration types that provide `arguments` object). You are free to use rest parameters (`...params`), though.

4.1 Context transparency

`this` keyword is a confusing aspect of JavaScript (check this article for a detailed explanation on `this`).
Because functions create own execution context, often it is difficult to detect `this` value.

ECMAScript 2015 improves `this` usage by introducing the arrow function, which takes the context lexically (or simply uses `this` from the immediate outer scope). This is nice because you don't have to use `.bind(this)` or store the context `var self = this` when a function needs the enclosing context.

Let's see how `this` is inherited from the outer function:

```
class Numbers {
  constructor(array) {
    this.array = array;
  }
  addNumber(number) {
    if (number !== undefined) {
       this.array.push(number);
    }
    return (number) => {
      console.log(this === numbersObject); // => true
      this.array.push(number);
    };
  }
}
const numbersObject = new Numbers([]);
const addMethod = numbersObject.addNumber();

addMethod(1);
addMethod(5);
console.log(numbersObject.array); // => [1, 5]
```

`Numbers` class holds an array of numbers and provides a method `addNumber()` to insert new numbers.
When `addNumber()` is called without arguments, a closure is returned that allows inserting numbers. This closure is an arrow function that has `this` as

`numbersObject` instance because the context is taken lexically from `addNumbers()` method.

Without the arrow function, you have to manually fix the context. It means using workarounds like `.bind()` method:

```
//...
    return function(number) {
      console.log(this === numbersObject); // => true
      this.array.push(number);
    }.bind(this);
//...
```

or store the context into a separated variable `var self = this`:

```
//...
    const self = this;
    return function(number) {
      console.log(self === numbersObject); // => true
      self.array.push(number);
    };
//...
```

The context transparency can be used when you want to keep `this` as is, taken from the enclosing context.

4.2 Short callbacks

When creating an arrow function, the parenthesis pairs and curly braces are optional for a single parameter and single body statement. This helps in creating very short callback functions.

Let's make a function that finds if an array contains `0`:

```
const numbers = [1, 5, 10, 0];
numbers.some(item => item === 0); // => true
```

`item => item === 0` is an arrow function that looks straightforward.

Note that nested short arrow functions are difficult to read. The convenient way to use the shortest arrow function form is a single callback (without nesting).

If necessary, use the expanded syntax of arrow functions when writing nested arrow functions. It's just easier to read.

5. Generator function

The generator function in JavaScript returns a Generator object. Its syntax is similar to function expression, function declaration or method declaration, just that it requires a star character `*`.

The generator function can be declared in the following forms:

a. Function declaration form `function* <name>():`

```
function* indexGenerator(){
  var index = 0;
  while(true) {
    yield index++;
  }
}
const g = indexGenerator();
console.log(g.next().value); // => 0
console.log(g.next().value); // => 1
```

b. Function expression form `function* ():`

```
const indexGenerator = function* () {
  let index = 0;
  while(true) {
    yield index++;
  }
};
const g = indexGenerator();
console.log(g.next().value); // => 0
console.log(g.next().value); // => 1
```

c. Shorthand method definition form `*<name>():`

```
const obj = {
  *indexGenerator() {
```

```
    var index = 0;
    while(true) {
      yield index++;
    }
  }
}
const g = obj.indexGenerator();
console.log(g.next().value); // => 0
console.log(g.next().value); // => 1
```

In all 3 cases the generator function returns the generator object `g`. Later `g` is used to generate a series of incremented numbers.

6. One more thing: new Function

In JavaScript functions are first-class objects - a function is a regular object of type `function`.
The ways of the declaration described above create the same function object type. Let's see an example:

```
function sum1(a, b) {
  return a + b;
}
const sum2 = function(a, b) {
  return a + b;
}
const sum3 = (a, b) => a + b;
console.log(typeof sum1 === 'function'); // => true
console.log(typeof sum2 === 'function'); // => true
console.log(typeof sum3 === 'function'); // => true
```

The function object type has a constructor: `Function`.
When `Function` is invoked as a constructor `new Function(arg1, arg2, ..., argN, bodyString)`, a new function is created. The arguments `arg1, args2, ..., argN` passed to constructor become the parameter names for the new function and the last argument `bodyString` is used as the function body code.

Let's create a function that sums two numbers:

```
const numberA = 'numberA', numberB = 'numberB';
const sumFunction = new Function(numberA, numberB,
   'return numberA + numberB'
```

```
);
sumFunction(10, 15) // => 25
```

`sumFunction` created with `Function` constructor invocation has parameters `numberA` and `numberB` and the body `return numberA + numberB`.

The functions created this way don't have access to the current scope, thus closures cannot be created. They are always created in the global scope.

One *possible* application of `new Function` is a better way to access the global object in a browser or NodeJS script:

```
(function() {
   'use strict';
   const global = new Function('return this')();
   console.log(global === window); // => true
   console.log(this === window);   // => false
})();
```

Remember that functions **almost never** should be declared using `new Function()`. Because the function body is evaluated on runtime, this approach inherits many `eval()` usage problems: security risks, harder debugging, no way to apply engine optimizations, no editor auto-complete.

7. At the end, which way is better?

There is no winner or loser. The decision which declaration type to choose depends on the situation.

There are some rules however that you may follow in common situations.

If the function uses `this` from the enclosing function, the arrow function is a good solution. When the callback function has one short statement, the arrow function is a good option too, because it creates short and light code.

For a shorter syntax when declaring methods on object literals, the shorthand method declaration is preferable.

`new Function` way to declare functions normally should not be used. Mainly because it opens potential security risks, doesn't allow code auto-complete in editors and loses the engine optimizations.

Do you prefer arrow functions or function expressions? Tell me why in a comment below!

Like the post? Please share!

Suggest Improvement

About Dmitri Pavlutin

Tech writer and coach. My daily routine consists of (but not limited to) drinking coffee, coding, writing, coaching, overcoming boredom 😉.

Recommended reading:

When 'Not' to Use Arrow Functions

javascript arrow function function

Arrow Functions Shortening Recipes in JavaScript

javascript arrow function es2015