



[Home](#) → [The JavaScript language](#) → [JavaScript Fundamentals](#)

6th November 2019

Function expressions

In JavaScript, a function is not a “magical language structure”, but a special kind of value.

The syntax that we used before is called a *Function Declaration*:

```
1 function sayHi() {  
2   alert( "Hello" );  
3 }
```

There is another syntax for creating a function that is called a *Function Expression*.

It looks like this:

```
1 let sayHi = function() {  
2   alert( "Hello" );  
3 };
```

Here, the function is created and assigned to the variable explicitly, like any other value. No matter how the function is defined, it's just a value stored in the variable `sayHi`.

The meaning of these code samples is the same: "create a function and put it into the variable `sayHi`".

We can even print out that value using `alert`:

```
1 function sayHi() {  
2   alert( "Hello" );  
3 }  
4  
5 alert( sayHi ); // shows the function code
```



Please note that the last line does not run the function, because there are no parentheses after `sayHi`. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.

In JavaScript, a function is a value, so we can deal with it as a value. The code above shows its string representation, which is the source code.

Surely, a function is a special value, in the sense that we can call it like `sayHi()`.

But it's still a value. So we can work with it like with other kinds of values.

We can copy a function to another variable:



```
1 function sayHi() { // (1) create
2   alert( "Hello" );
3 }
4
5 let func = sayHi; // (2) copy
6
7 func(); // Hello // (3) run the copy (it works)!
8 sayHi(); // Hello // this still works too (why wouldn't it)
```

Here's what happens above in detail:

1. The Function Declaration (1) creates the function and puts it into the variable named `sayHi`.
2. Line (2) copies it into the variable `func`. Please note again: there are no parentheses after `sayHi`. If there were, then `func = sayHi()` would write *the result of the call* `sayHi()` into `func`, not *the function* `sayHi` itself.
3. Now the function can be called as both `sayHi()` and `func()`.

Note that we could also have used a Function Expression to declare `sayHi`, in the first line:

```
1 let sayHi = function() {
2   alert( "Hello" );
3 };
4
5 let func = sayHi;
6 // ...
```

Everything would work the same.

i Why is there a semicolon at the end?

You might wonder, why does Function Expression have a semicolon `;` at the end, but Function Declaration does not:

```
1 function sayHi() {
2   // ...
3 }
4
5 let sayHi = function() {
6   // ...
7 };
```

The answer is simple:

- There's no need for `;` at the end of code blocks and syntax structures that use them like `if { ... }`, `for { }`, `function f { }` etc.
- A Function Expression is used inside the statement: `let sayHi = ...;`, as a value. It's not a code block, but rather an assignment. The semicolon `;` is recommended at the end of statements, no matter what the value is. So the semicolon here is not related to the Function Expression itself, it just terminates the statement.

Callback functions

Let's look at more examples of passing functions as values and using function expressions.

We'll write a function `ask(question, yes, no)` with three parameters:

question

Text of the question

yes

Function to run if the answer is "Yes"

no

Function to run if the answer is "No"

The function should ask the `question` and, depending on the user's answer, call `yes()` or `no()` :

```
1 function ask(question, yes, no) {
2   if (confirm(question)) yes()
3   else no();
4 }
5
6 function showOk() {
7   alert( "You agreed." );
8 }
9
10 function showCancel() {
11   alert( "You canceled the execution." );
12 }
13
14 // usage: functions showOk, showCancel are passed as arguments to ask
15 ask("Do you agree?", showOk, showCancel);
```

In practice, such functions are quite useful. The major difference between a real-life `ask` and the example above is that real-life functions use more complex ways to interact with the user than a simple `confirm`. In the browser, such function usually draws a nice-looking question window. But that's another story.

The arguments `showOk` and `showCancel` of `ask` are called *callback functions* or just *callbacks*.

The idea is that we pass a function and expect it to be "called back" later if necessary. In our case, `showOk` becomes the callback for "yes" answer, and `showCancel` for "no" answer.

We can use Function Expressions to write the same function much shorter:

```
1 function ask(question, yes, no) {
2   if (confirm(question)) yes()
3   else no();
4 }
5
6 ask(
7   "Do you agree?",
8   function() { alert("You agreed."); },
9 )
```

```
10  function() { alert("You canceled the execution."); }  
    );
```

Here, functions are declared right inside the `ask(...)` call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of `ask` (because they are not assigned to variables), but that's just what we want here.

Such code appears in our scripts very naturally, it's in the spirit of JavaScript.

A function is a value representing an “action”

Regular values like strings or numbers represent the *data*.

A function can be perceived as an *action*.

We can pass it between variables and run when we want.

Function Expression vs Function Declaration

Let's formulate the key differences between Function Declarations and Expressions.

First, the syntax: how to differentiate between them in the code.

- *Function Declaration*: a function, declared as a separate statement, in the main code flow.

```
1  // Function Declaration  
2  function sum(a, b) {  
3    return a + b;  
4  }
```

- *Function Expression*: a function, created inside an expression or inside another syntax construct. Here, the function is created at the right side of the “assignment expression” `= :`

```
1  // Function Expression  
2  let sum = function(a, b) {  
3    return a + b;  
4  };
```

The more subtle difference is *when* a function is created by the JavaScript engine.

A Function Expression is created when the execution reaches it and is usable only from that moment.

Once the execution flow passes to the right side of the assignment `let sum = function...` – here we go, the function is created and can be used (assigned, called, etc.) from now on.

Function Declarations are different.

A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is.

That's due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

And after all Function Declarations are processed, the code is executed. So it has access to these functions.

For example, this works:

```
1 sayHi("John"); // Hello, John
2
3 function sayHi(name) {
4   alert( `Hello, ${name}` );
5 }
```



The Function Declaration `sayHi` is created when JavaScript is preparing to start the script and is visible everywhere in it.

...If it were a Function Expression, then it wouldn't work:

```
1 sayHi("John"); // error!
2
3 let sayHi = function(name) { // (*) no magic any more
4   alert( `Hello, ${name}` );
5 };
```



Function Expressions are created when the execution reaches them. That would happen only in the line `(*)`. Too late.

Another special feature of Function Declarations is their block scope.

In strict mode, when a Function Declaration is within a code block, it's visible everywhere inside that block. But not outside of it.

For instance, let's imagine that we need to declare a function `welcome()` depending on the `age` variable that we get during runtime. And then we plan to use it some time later.

If we use Function Declaration, it won't work as intended:

```
1 let age = prompt("What is your age?", 18);
2
3 // conditionally declare a function
4 if (age < 18) {
5
6   function welcome() {
7     alert("Hello!");
8   }
9
10 } else {
11
12   function welcome() {
13     alert("Greetings!");
14   }
15
16 }
17
18 // ...use it later
19 welcome(); // Error: welcome is not defined
```



That's because a Function Declaration is only visible inside the code block in which it resides.

Here's another example:

```

1  let age = 16; // take 16 as an example
2
3  if (age < 18) {
4      welcome();           // \   (runs)
5                          // |
6      function welcome() { // |
7          alert("Hello!"); // |   Function Declaration is available
8      }                   // |   everywhere in the block where it's declared
9                          // |
10     welcome();           // /   (runs)
11
12 } else {
13     function welcome() {
14         alert("Greetings!");
15     }
16 }
17
18
19 // Here we're out of curly braces,
20 // so we can not see Function Declarations made inside of them.
21
22 welcome(); // Error: welcome is not defined

```

What can we do to make `welcome` visible outside of `if`?

The correct approach would be to use a Function Expression and assign `welcome` to the variable that is declared outside of `if` and has the proper visibility.

This code works as intended:

```

1  let age = prompt("What is your age?", 18);
2
3  let welcome;
4
5  if (age < 18) {
6
7      welcome = function() {
8          alert("Hello!");
9      };
10
11 } else {
12
13     welcome = function() {
14         alert("Greetings!");
15     };
16
17 }
18
19 welcome(); // ok now

```

Or we could simplify it even further using a question mark operator `?`:



```
1 let age = prompt("What is your age?", 18);
2
3 let welcome = (age < 18) ?
4   function() { alert("Hello!"); } :
5   function() { alert("Greetings!"); };
6
7 welcome(); // ok now
```

i When to choose Function Declaration versus Function Expression?

As a rule of thumb, when we need to declare a function, the first to consider is Function Declaration syntax. It gives more freedom in how to organize our code, because we can call such functions before they are declared.

That's also better for readability, as it's easier to look up `function f(...) {...}` in the code than `let f = function(...) {...};`. Function Declarations are more “eye-catching”.

...But if a Function Declaration does not suit us for some reason, or we need a conditional declaration (we've just seen an example), then Function Expression should be used.

Summary

- Functions are values. They can be assigned, copied or declared in any place of the code.
- If the function is declared as a separate statement in the main code flow, that's called a “Function Declaration”.
- If the function is created as a part of an expression, it's called a “Function Expression”.
- Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
- Function Expressions are created when the execution flow reaches them.

In most cases when we need to declare a function, a Function Declaration is preferable, because it is visible prior to the declaration itself. That gives us more flexibility in code organization, and is usually more readable.

So we should use a Function Expression only when a Function Declaration is not fit for the task. We've seen a couple of examples of that in this chapter, and will see more in the future.

[Previous lesson](#)[Next lesson](#)

Share  

 [Tutorial map](#)

Comments

- If you have suggestions what to improve - please [submit a GitHub issue](#) or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.

- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>` , for more than 10 lines – use a sandbox ([plnkr](#), [JSBin](#), [codepen](#)...)