Function expressions and arrows

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*:

```
function sayHi() {
  alert( "Hello" );
}
```

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

It looks like this:

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

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` :

```
function sayHi() {
  alert( "Hello" );
}

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.

It is a special value of course, 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:

```
function sayHi() {   // (1) create
  alert( "Hello" );
}

let func = sayHi;    // (2) copy

func(); // Hello     // (3) run the copy (it works)!
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:

```
let sayHi = function() { ... };

let func = sayHi;
// ...
```

Everything would work the same. Even more obvious what's going on, right?

Why there’s a semicolon at the end?

There might be a question, why does Function Expression have a semicolon `;` at the end, and Function Declaration does not:

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

let sayHi = function() {
  // ...
};
```

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. The semicolon `;` is recommended at the end of statements, no matter what is the value. So the semicolon here is not related to the Function Expression itself in any way, 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()`:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

function showOk() {
  alert( "You agreed." );
}

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

// usage: functions showOk, showCancel are passed as arguments to ask
ask("Do you agree?", showOk, showCancel);
```

Before we explore how we can write it in a much shorter way, let’s note that in the browser (and on the server-side in some cases) such functions are quite popular. The major difference between a real-life implementation 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 a function usually draws a nice-looking question window. But that’s another story.

The arguments 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 the “yes” answer, and `showCancel` for the “no” answer.

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

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  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 see what is what in the code.

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

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

- *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 =":

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

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 from then on.

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 is usable in the whole script/code block.

In other words, when JavaScript *prepares* to run the script or a code block, it first looks for Function Declarations in it and creates the functions. We can think of it as an "initialization stage".

And after all of the Function Declarations are processed, the execution goes on.

As a result, a function declared as a Function Declaration can be called earlier than it is defined.

For example, this works:

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

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

…If it was a Function Expression, then it wouldn't work:

```
sayHi("John"); // error!

let sayHi = function(name) {  // (*) no magic any more
  alert( `Hello, ${name}` );
};
```

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

When a Function Declaration is made within a code block, it is visible everywhere inside that block. But not outside of it.

Sometimes that's handy to declare a local function only needed in that block alone. But that feature may also cause problems.

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

The code below doesn't work:

```
let age = prompt("What is your age?", 18);

// conditionally declare a function
if (age < 18) {

  function welcome() {
    alert("Hello!");
  }

} else {

  function welcome() {
    alert("Greetings!");
  }

}

// ...use it later
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:

```
let age = 16; // take 16 as an example

if (age < 18) {
  welcome();               // \   (runs)
                           //  |
  function welcome() {     //  |
    alert("Hello!");       //  |  Function Declaration is available
  }                        //  |  everywhere in the block where it's declared
                           //  |
  welcome();               // /   (runs)

} else {

  function welcome() {     //  for age = 16, this "welcome" is never created
    alert("Greetings!");
  }
}

// Here we're out of figure brackets,
// so we can not see Function Declarations made inside of them.

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.

Now it works as intended:

```
let age = prompt("What is your age?", 18);

let welcome;

if (age < 18) {

  welcome = function() {
    alert("Hello!");
  };

} else {

  welcome = function() {
    alert("Greetings!");
  };

}

welcome(); // ok now
```

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

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  function() { alert("Hello!"); } :
  function() { alert("Greetings!"); };

welcome(); // ok now
```

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, the one we used before. It gives more freedom in how to organize our code, because we can call such functions before they are declared.

It's also a little bit 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 (we've seen an example above), then Function Expression should be used.

Arrow functions

There’s one more very simple and concise syntax for creating functions, that’s often better than Function Expressions. It’s called “arrow functions”, because it looks like this:

```
let func = (arg1, arg2, ...argN) => expression
```

…This creates a function `func` that has arguments `arg1..argN`, evaluates the `expression` on the right side with their use and returns its result.

In other words, it’s roughly the same as:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
}
```

…But much more concise.

Let’s see an example:

```
let sum = (a, b) => a + b;

/* The arrow function is a shorter form of:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

If we have only one argument, then parentheses can be omitted, making that even shorter:

```
// same as
// let double = function(n) { return n * 2 }
let double = n => n * 2;

alert( double(3) ); // 6
```

If there are no arguments, parentheses should be empty (but they should be present):

```
let sayHi = () => alert("Hello!");

sayHi();
```

Arrow functions can be used in the same way as Function Expressions.

For instance, here’s the rewritten example with `welcome()`:

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");

welcome(); // ok now
```

Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure.

They are very convenient for simple one-line actions, when we're just too lazy to write many words.

Multiline arrow functions

The examples above took arguments from the left of `=>` and evaluated the right-side expression with them.

Sometimes we need something a little bit more complex, like multiple expressions or statements. It is also possible, but we should enclose them in figure brackets. Then use a normal `return` within them.

Like this:

```
let sum = (a, b) => {  // the figure bracket opens a multiline function
  let result = a + b;
  return result; // if we use figure brackets, use return to get results
};

alert( sum(1, 2) ); // 3
```

More to come

Here we praised arrow functions for brevity. But that's not all! Arrow functions have other interesting features. We'll return to them later in the chapter Arrow functions revisited.

For now, we can already use them for one-line actions and callbacks.

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.

Arrow functions are handy for one-liners. They come in two flavors:

1. Without figure brackets: `(...args) => expression` – the right side is an expression: the function evaluates it and returns the result.
2. With figure brackets: `(...args) => { body }` – brackets allow us to write multiple statements inside the function, but we need an explicit `return` to return something.

Tasks

Rewrite with arrow functions

Replace Function Expressions with arrow functions in the code:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
);
```

solution

Comments