12th October 2019

Function object, NFE

As we already know, a function in JavaScript is a value.

Every value in JavaScript has a type. What type is a function?

In JavaScript, functions are objects.

A good way to imagine functions is as callable "action objects". We can not only call them, but also treat them as objects: add/remove properties, pass by reference etc.

The "name" property

Function objects contain some useable properties.

For instance, a function's name is accessible as the "name" property:

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

alert(sayHi.name); // sayHi
```

What's kind of funny, the name-assigning logic is smart. It also assigns the correct name to a function even if it's created without one, and then immediately assigned:

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

alert(sayHi.name); // sayHi (there's a name!)
```

It also works if the assignment is done via a default value:

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (works!)
}

f();
```

In the specification, this feature is called a “contextual name”. If the function does not provide one, then in an assignment it is figured out from the context.

Object methods have names too:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }

}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

There’s no magic though. There are cases when there’s no way to figure out the right name. In that case, the name property is empty, like here:

```
// function created inside array
let arr = [function() {}];

alert( arr[0].name ); // <empty string>
// the engine has no way to set up the right name, so there is none
```

In practice, however, most functions do have a name.

The “length” property

There is another built-in property “length” that returns the number of function parameters, for instance:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Here we can see that rest parameters are not counted.

The `length` property is sometimes used for introspection in functions that operate on other functions.

For instance, in the code below the `ask` function accepts a `question` to ask and an arbitrary number of `handler` functions to call.

Once a user provides their answer, the function calls the handlers. We can pass two kinds of handlers:

- A zero-argument function, which is only called when the user gives a positive answer.

- A function with arguments, which is called in either case and returns an answer.

To call `handler` the right way, we examine the `handler.length` property.

The idea is that we have a simple, no-arguments handler syntax for positive cases (most frequent variant), but are able to support universal handlers as well:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

}

// for positive answer, both handlers are called
// for negative answer, only the second one
ask("Question?", () => alert('You said yes'), result => alert(result));
```

This is a particular case of so-called polymorphism – treating arguments differently depending on their type or, in our case depending on the `length`. The idea does have a use in JavaScript libraries.

Custom properties

We can also add properties of our own.

Here we add the `counter` property to track the total calls count:

```
function sayHi() {
  alert("Hi");

  // let's count how many times we run
  sayHi.counter++;
}
sayHi.counter = 0; // initial value

sayHi(); // Hi
sayHi(); // Hi

alert( `Called ${sayHi.counter} times` ); // Called 2 times
```

A property is not a variable

A property assigned to a function like `sayHi.counter = 0` does *not* define a local variable `counter` inside it. In other words, a property `counter` and a variable `let counter` are two unrelated things.

We can treat a function as an object, store properties in it, but that has no effect on its execution. Variables are not function properties and vice versa. These are just parallel worlds.

Function properties can replace closures sometimes. For instance, we can rewrite the counter function example from the chapter Variable scope to use a function property:

```
function makeCounter() {
  // instead of:
  // let count = 0

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

The `count` is now stored in the function directly, not in its outer Lexical Environment.

Is it better or worse than using a closure?

The main difference is that if the value of `count` lives in an outer variable, then external code is unable to access it. Only nested functions may modify it. And if it's bound to a function, then such a thing is possible:

```
function makeCounter() {

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

So the choice of implementation depends on our aims.

Named Function Expression

Named Function Expression, or NFE, is a term for Function Expressions that have a name.

For instance, let's take an ordinary Function Expression:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

And add a name to it:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

Did we achieve anything here? What's the purpose of that additional `"func"` name?

First let's note, that we still have a Function Expression. Adding the name `"func"` after `function` did not make it a Function Declaration, because it is still created as a part of an assignment expression.

Adding such a name also did not break anything.

The function is still available as `sayHi()` :

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

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

There are two special things about the name `func` , that are the reasons for it:

1. It allows the function to reference itself internally.
2. It is not visible outside of the function.

For instance, the function `sayHi` below calls itself again with `"Guest"` if no `who` is provided:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // use func to re-call itself
  }
};

sayHi(); // Hello, Guest

// But this won't work:
func(); // Error, func is not defined (not visible outside of the function)
```

Why do we use `func` ? Maybe just use `sayHi` for the nested call?

Actually, in most cases we can:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

The problem with that code is that `sayHi` may change in the outer code. If the function gets assigned to another variable instead, the code will start to give errors:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Error: sayHi is not a function
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Error, the nested sayHi call doesn't work any more!
```

That happens because the function takes `sayHi` from its outer lexical environment. There's no local `sayHi`, so the outer variable is used. And at the moment of the call that outer `sayHi` is `null`.

The optional name which we can put into the Function Expression is meant to solve exactly these kinds of problems.

Let's use it to fix our code:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // Now all fine
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (nested call works)
```

Now it works, because the name `"func"` is function-local. It is not taken from outside (and not visible there). The specification guarantees that it will always reference the current function.

The outer code still has it's variable `sayHi` or `welcome`. And `func` is an "internal function name", how the function can call itself internally.

There's no such thing for Function Declaration

The "internal name" feature described here is only available for Function Expressions, not for Function Declarations. For Function Declarations, there is no syntax for adding an "internal" name.

Sometimes, when we need a reliable internal name, it's the reason to rewrite a Function Declaration to Named Function Expression form.

Summary

Functions are objects.

Here we covered their properties:

- `name` – the function name. Usually taken from the function definition, but if there's none, JavaScript tries to guess it from the context (e.g. an assignment).
- `length` – the number of arguments in the function definition. Rest parameters are not counted.

If the function is declared as a Function Expression (not in the main code flow), and it carries the name, then it is called a Named Function Expression. The name can be used inside to reference itself, for recursive calls or such.

Also, functions may carry additional properties. Many well-known JavaScript libraries make great use of this feature.

They create a "main" function and attach many other "helper" functions to it. For instance, the jQuery library creates a function named `$`. The lodash library creates a function `_`, and then adds `_.clone`, `_.keyBy` and other properties to it (see the docs when you want learn more about them). Actually, they do it to lessen their pollution of the global space, so that a single library gives only one global variable. That reduces the possibility of naming conflicts.

So, a function can do a useful job by itself and also carry a bunch of other functionality in properties.

✓ Tasks

Set and decrease for counter

importance: 5

Modify the code of `makeCounter()` so that the counter can also decrease and set the number:

- `counter()` should return the next number (as before).
- `counter.set(value)` should set the counter to `value`.
- `counter.decrease()` should decrease the counter by 1.

See the sandbox code for the complete usage example.

P.S. You can use either a closure or the function property to keep the current count. Or write both variants.

Open a sandbox with tests.

solution

Sum with an arbitrary amount of brackets

importance: 2

Write function `sum` that would work like this:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.S. Hint: you may need to setup custom object to primitive conversion for your function.

solution

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…)