



[Home](#) → [The JavaScript language](#) → [Advanced working with functions](#)

📅 27th December 2019

Decorators and forwarding, call/apply

JavaScript gives exceptional flexibility when dealing with functions. They can be passed around, used as objects, and now we'll see how to *forward* calls between them and *decorate* them.

Transparent caching

Let's say we have a function `slow(x)` which is CPU-heavy, but its results are stable. In other words, for the same `x` it always returns the same result.

If the function is called often, we may want to cache (remember) the results to avoid spending extra-time on recalculations.

But instead of adding that functionality into `slow()` we'll create a wrapper function, that adds caching. As we'll see, there are many benefits of doing so.

Here's the code, and explanations follow:

```
1 function slow(x) {  
2   // there can be a heavy CPU-intensive job here  
3   alert(`Called with ${x}`);  
4   return x;  
5 }  
6  
7 function cachingDecorator(func) {  
8   let cache = new Map();  
9  
10  return function(x) {  
11    if (cache.has(x)) { // if there's such key in cache  
12      return cache.get(x); // read the result from it  
13    }  
14  
15    let result = func(x); // otherwise call func  
16  
17    cache.set(x, result); // and cache (remember) the result  
18    return result;  
19  };  
20 }  
21  
22 slow = cachingDecorator(slow);  
23  
24 alert( slow(1) ); // slow(1) is cached  
25 alert( "Again: " + slow(1) ); // the same  
26  
27 alert( slow(2) ); // slow(2) is cached  
28 alert( "Again: " + slow(2) ); // the same as the previous line
```



In the code above `cachingDecorator` is a *decorator*: a special function that takes another function and alters its behavior.

The idea is that we can call `cachingDecorator` for any function, and it will return the caching wrapper. That's great, because we can have many functions that could use such a feature, and all we need to do is to apply `cachingDecorator` to them.


By separating caching from the main function code we also keep the main code simpler.

The result of `cachingDecorator(func)` is a “wrapper”: `function(x)` that “wraps” the call of `func(x)` into caching logic:

```
function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x);
    cache.set(x, result);
    return result;
  };
}
```



The diagram consists of a large red curved arrow pointing from the `return` statement of the `cachingDecorator` function to the `func(x)` call inside the wrapper function. The word "wrapper" is written above the arrow, and the phrase "around the function" is written to the right of the arrow.

From an outside code, the wrapped `slow` function still does the same. It just got a caching aspect added to its behavior.

To summarize, there are several benefits of using a separate `cachingDecorator` instead of altering the code of `slow` itself:

- The `cachingDecorator` is reusable. We can apply it to another function.
- The caching logic is separate, it did not increase the complexity of `slow` itself (if there was any).
- We can combine multiple decorators if needed (other decorators will follow).

Using “func.call” for the context

The caching decorator mentioned above is not suited to work with object methods.

For instance, in the code below `worker.slow()` stops working after the decoration:

```
1 // we'll make worker.slow caching
2 let worker = {
3   someMethod() {
4     return 1;
5   },
6
7   slow(x) {
8     // scary CPU-heavy task here
9     alert("Called with " + x);
10    return x * this.someMethod(); // (*)
11  }
12 };
13
```



```
14 // same code as before
15 function cachingDecorator(func) {
16   let cache = new Map();
17   return function(x) {
18     if (cache.has(x)) {
19       return cache.get(x);
20     }
21     let result = func(x); // (**)
22     cache.set(x, result);
23     return result;
24   };
25 }
26
27 alert( worker.slow(1) ); // the original method works
28
29 worker.slow = cachingDecorator(worker.slow); // now make it caching
30
31 alert( worker.slow(2) ); // Whoops! Error: Cannot read property 'someMethod'
```

The error occurs in the line (*) that tries to access `this.someMethod` and fails. Can you see why?

The reason is that the wrapper calls the original function as `func(x)` in the line (**). And, when called like that, the function gets `this = undefined`.

We would observe a similar symptom if we tried to run:

```
1 let func = worker.slow;
2 func(2);
```

So, the wrapper passes the call to the original method, but without the context `this`. Hence the error.

Let's fix it.

There's a special built-in function method `func.call(context, ...args)` that allows to call a function explicitly setting `this`.

The syntax is:

```
1 func.call(context, arg1, arg2, ...)
```

It runs `func` providing the first argument as `this`, and the next as the arguments.

To put it simply, these two calls do almost the same:

```
1 func(1, 2, 3);
2 func.call(obj, 1, 2, 3)
```

They both call `func` with arguments `1`, `2` and `3`. The only difference is that `func.call` also sets `this` to `obj`.

As an example, in the code below we call `sayHi` in the context of different objects: `sayHi.call(user)` runs `sayHi` providing `this=user`, and the next line sets `this=admin`:



```
1 function sayHi() {
2   alert(this.name);
3 }
4
5 let user = { name: "John" };
6 let admin = { name: "Admin" };
7
8 // use call to pass different objects as "this"
9 sayHi.call( user ); // John
10 sayHi.call( admin ); // Admin
```

And here we use `call` to call `say` with the given context and phrase:



```
1 function say(phrase) {
2   alert(this.name + ': ' + phrase);
3 }
4
5 let user = { name: "John" };
6
7 // user becomes this, and "Hello" becomes the first argument
8 say.call( user, "Hello" ); // John: Hello
```

In our case, we can use `call` in the wrapper to pass the context to the original function:



```
1 let worker = {
2   someMethod() {
3     return 1;
4   },
5
6   slow(x) {
7     alert("Called with " + x);
8     return x * this.someMethod(); // (*)
9   }
10 };
11
12 function cachingDecorator(func) {
13   let cache = new Map();
14   return function(x) {
15     if (cache.has(x)) {
16       return cache.get(x);
17     }
18     let result = func.call(this, x); // "this" is passed correctly now
19     cache.set(x, result);
20     return result;
21   };
22 }
23
24 worker.slow = cachingDecorator(worker.slow); // now make it caching
25
26 alert( worker.slow(2) ); // works
27 alert( worker.slow(2) ); // works, doesn't call the original (cached)
```

Now everything is fine.

To make it all clear, let's see more deeply how `this` is passed along:

1. After the decoration `worker.slow` is now the wrapper `function (x) { ... }`.
2. So when `worker.slow(2)` is executed, the wrapper gets `2` as an argument and `this=worker` (it's the object before dot).
3. Inside the wrapper, assuming the result is not yet cached, `func.call(this, x)` passes the current `this (=worker)` and the current argument (`=2`) to the original method.

Going multi-argument with “func.apply”

Now let's make `cachingDecorator` even more universal. Till now it was working only with single-argument functions.

Now how to cache the multi-argument `worker.slow` method?

```
1 let worker = {
2   slow(min, max) {
3     return min + max; // scary CPU-hogger is assumed
4   }
5 };
6
7 // should remember same-argument calls
8 worker.slow = cachingDecorator(worker.slow);
```

Previously, for a single argument `x` we could just `cache.set(x, result)` to save the result and `cache.get(x)` to retrieve it. But now we need to remember the result for a *combination of arguments* (`min,max`). The native `Map` takes single value only as the key.

There are many solutions possible:

1. Implement a new (or use a third-party) map-like data structure that is more versatile and allows multi-keys.
2. Use nested maps: `cache.set(min)` will be a `Map` that stores the pair (`max, result`). So we can get `result` as `cache.get(min).get(max)`.
3. Join two values into one. In our particular case we can just use a string `"min,max"` as the `Map` key. For flexibility, we can allow to provide a *hashing function* for the decorator, that knows how to make one value from many.

For many practical applications, the 3rd variant is good enough, so we'll stick to it.

Also we need to replace `func.call(this, x)` with `func.call(this, ...arguments)`, to pass all arguments to the wrapped function call, not just the first one.

Here's a more powerful `cachingDecorator`:

```
1 let worker = {
2   slow(min, max) {
3     alert(`Called with ${min},${max}`);
4     return min + max;
5   }
6 };
7
8 function cachingDecorator(func, hash) {
9   let cache = new Map();
```



```

10  return function() {
11      let key = hash(arguments); // (*)
12      if (cache.has(key)) {
13          return cache.get(key);
14      }
15
16      let result = func.call(this, ...arguments); // (**)
17
18      cache.set(key, result);
19      return result;
20  };
21  }
22
23  function hash(args) {
24      return args[0] + ',' + args[1];
25  }
26
27  worker.slow = cachingDecorator(worker.slow, hash);
28
29  alert( worker.slow(3, 5) ); // works
30  alert( "Again " + worker.slow(3, 5) ); // same (cached)

```

Now it works with any number of arguments (though the hash function would also need to be adjusted to allow any number of arguments. An interesting way to handle this will be covered below).

There are two changes:

- In the line `(*)` it calls `hash` to create a single key from `arguments`. Here we use a simple “joining” function that turns arguments `(3, 5)` into the key `"3,5"`. More complex cases may require other hashing functions.
- Then `(**)` uses `func.call(this, ...arguments)` to pass both the context and all arguments the wrapper got (not just the first one) to the original function.

Instead of `func.call(this, ...arguments)` we could use `func.apply(this, arguments)`.

The syntax of built-in method `func.apply` is:

```
1 func.apply(context, args)
```

It runs the `func` setting `this=context` and using an array-like object `args` as the list of arguments.

The only syntax difference between `call` and `apply` is that `call` expects a list of arguments, while `apply` takes an array-like object with them.

So these two calls are almost equivalent:

```

1 func.call(context, ...args); // pass an array as list with spread syntax
2 func.apply(context, args);   // is same as using apply

```

There's only a minor difference:

- The spread syntax `...` allows to pass *iterable* `args` as the list to `call`.
- The `apply` accepts only *array-like* `args`.

So, these calls complement each other. Where we expect an iterable, `call` works, where we expect an array-like, `apply` works.

And for objects that are both iterable and array-like, like a real array, we technically could use any of them, but `apply` will probably be faster, because most JavaScript engines internally optimize it better.

Passing all arguments along with the context to another function is called *call forwarding*.

That's the simplest form of it:

```
1 let wrapper = function() {  
2   return func.apply(this, arguments);  
3 };
```

When an external code calls such `wrapper`, it is indistinguishable from the call of the original function `func`.

Borrowing a method

Now let's make one more minor improvement in the hashing function:

```
1 function hash(args) {  
2   return args[0] + ',' + args[1];  
3 }
```

As of now, it works only on two arguments. It would be better if it could glue any number of `args`.

The natural solution would be to use `arr.join` method:

```
1 function hash(args) {  
2   return args.join();  
3 }
```

...Unfortunately, that won't work. Because we are calling `hash(arguments)` and `arguments` object is both iterable and array-like, but not a real array.

So calling `join` on it would fail, as we can see below:

```
1 function hash() {  
2   alert( arguments.join() ); // Error: arguments.join is not a function  
3 }  
4  
5 hash(1, 2);
```

Still, there's an easy way to use array `join`:

```
1 function hash() {  
2   alert( [].join.call(arguments) ); // 1,2  
3 }  
4
```

```
5  
  hash(1, 2);
```

The trick is called *method borrowing*.

We take (borrow) a join method from a regular array (`[].join`) and use `[].join.call` to run it in the context of `arguments` .

Why does it work?

That's because the internal algorithm of the native method `arr.join(glue)` is very simple.

Taken from the specification almost “as-is”:

1. Let `glue` be the first argument or, if no arguments, then a comma `", "` .
2. Let `result` be an empty string.
3. Append `this[0]` to `result` .
4. Append `glue` and `this[1]` .
5. Append `glue` and `this[2]` .
6. ...Do so until `this.length` items are glued.
7. Return `result` .

So, technically it takes `this` and joins `this[0]` , `this[1]` ...etc together. It's intentionally written in a way that allows any array-like `this` (not a coincidence, many methods follow this practice). That's why it also works with `this=arguments` .

Decorators and function properties

It is generally safe to replace a function or a method with a decorated one, except for one little thing. If the original function had properties on it, like `func.calledCount` or whatever, then the decorated one will not provide them. Because that is a wrapper. So one needs to be careful if one uses them.

E.g. in the example above if `slow` function had any properties on it, then `cachingDecorator(slow)` is a wrapper without them.

Some decorators may provide their own properties. E.g. a decorator may count how many times a function was invoked and how much time it took, and expose this information via wrapper properties.

There exists a way to create decorators that keep access to function properties, but this requires using a special `Proxy` object to wrap a function. We'll discuss it later in the article [Proxy and Reflect](#).

Summary

Decorator is a wrapper around a function that alters its behavior. The main job is still carried out by the function.

Decorators can be seen as “features” or “aspects” that can be added to a function. We can add one or add many. And all this without changing its code!

To implement `cachingDecorator` , we studied methods:

- `func.call(context, arg1, arg2...)` – calls `func` with given context and arguments.
- `func.apply(context, args)` – calls `func` passing context as `this` and array-like `args` into a list of arguments.

The generic *call forwarding* is usually done with `apply` :

```
1 let wrapper = function() {  
2   return original.apply(this, arguments);  
3 };
```

We also saw an example of *method borrowing* when we take a method from an object and `call` it in the context of another object. It is quite common to take array methods and apply them to `arguments` . The alternative is to use rest parameters object that is a real array.

There are many decorators there in the wild. Check how well you got them by solving the tasks of this chapter.

✓ Tasks

Spy decorator

importance: 5

Create a decorator `spy(func)` that should return a wrapper that saves all calls to function in its `calls` property.

Every call is saved as an array of arguments.

For instance:

```
1 function work(a, b) {  
2   alert( a + b ); // work is an arbitrary function or method  
3 }  
4  
5 work = spy(work);  
6  
7 work(1, 2); // 3  
8 work(4, 5); // 9  
9  
10 for (let args of work.calls) {  
11   alert( 'call:' + args.join() ); // "call:1,2", "call:4,5"  
12 }
```

P.S. That decorator is sometimes useful for unit-testing. Its advanced form is `sinon.spy` in [Sinon.JS](#) library.

[Open a sandbox with tests.](#)

solution

Delaying decorator

importance: 5

Create a decorator `delay(f, ms)` that delays each call of `f` by `ms` milliseconds.

For instance:

```
1 function f(x) {
2   alert(x);
3 }
4
5 // create wrappers
6 let f1000 = delay(f, 1000);
7 let f1500 = delay(f, 1500);
8
9 f1000("test"); // shows "test" after 1000ms
10 f1500("test"); // shows "test" after 1500ms
```

In other words, `delay(f, ms)` returns a "delayed by `ms`" variant of `f`.

In the code above, `f` is a function of a single argument, but your solution should pass all arguments and the context `this`.

[Open a sandbox with tests.](#)

solution

Debounce decorator

importance: 5

The result of `debounce(f, ms)` decorator should be a wrapper that passes the call to `f` at maximum once per `ms` milliseconds.

In other words, when we call a “debounced” function, it guarantees that all future calls to the function made less than `ms` milliseconds after the previous call will be ignored.

For instance:

```
1 let f = debounce(alert, 1000);
2
3 f(1); // runs immediately
4 f(2); // ignored
5
6 setTimeout( () => f(3), 100); // ignored ( only 100 ms passed )
7 setTimeout( () => f(4), 1100); // runs
8 setTimeout( () => f(5), 1500); // ignored (less than 1000 ms from the last ru
```

In practice `debounce` is useful for functions that retrieve/update something when we know that nothing new can be done in such a short period of time, so it's better not to waste resources.

[Open a sandbox with tests.](#)

solution

Throttle decorator

importance: 5

Create a “throttling” decorator `throttle(f, ms)` – that returns a wrapper, passing the call to `f` at maximum once per `ms` milliseconds. Those calls that fall into the “cooldown” period, are ignored.

The difference with `debounce` – if an ignored call is the last during the cooldown, then it executes at the end of the delay.

Let’s check the real-life application to better understand that requirement and to see where it comes from.

For instance, we want to track mouse movements.

In a browser we can setup a function to run at every mouse movement and get the pointer location as it moves. During an active mouse usage, this function usually runs very frequently, can be something like 100 times per second (every 10 ms).

We’d like to update some information on the web-page when the pointer moves.

...But updating function `update()` is too heavy to do it on every micro-movement. There is also no sense in updating more often than once per 100ms.

So we’ll wrap it into the decorator: use `throttle(update, 100)` as the function to run on each mouse move instead of the original `update()`. The decorator will be called often, but forward the call to `update()` at maximum once per 100ms.

Visually, it will look like this:

1. For the first mouse movement the decorated variant immediately passes the call to `update`. That’s important, the user sees our reaction to their move immediately.
2. Then as the mouse moves on, until `100ms` nothing happens. The decorated variant ignores calls.
3. At the end of `100ms` – one more `update` happens with the last coordinates.
4. Then, finally, the mouse stops somewhere. The decorated variant waits until `100ms` expire and then runs `update` with last coordinates. So, quite important, the final mouse coordinates are processed.

A code example:

```
1 function f(a) {
2   console.log(a);
3 }
4
5 // f1000 passes calls to f at maximum once per 1000 ms
6 let f1000 = throttle(f, 1000);
7
8 f1000(1); // shows 1
9 f1000(2); // (throttling, 1000ms not out yet)
10 f1000(3); // (throttling, 1000ms not out yet)
11
12 // when 1000 ms time out...
13 // ...outputs 3, intermediate value 2 was ignored
```

P.S. Arguments and the context `this` passed to `f1000` should be passed to the original `f`.

[Open a sandbox with tests.](#)

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](#)...)