JavaScript Decorators: What They Are and When to Use Them

sitepoint.com/javascript-decorators-what-they-are/

April 19, 2018

With the introduction of ES2015+, and as transpilation has become commonplace, many of you will have come across newer language features, either in real code or tutorials. One of these features that often has people scratching their heads when they first come across them are JavaScript decorators.

Decorators have become popular thanks to their use in Angular 2+. In Angular, decorators are available thanks to TypeScript, but in JavaScript they're currently a stage 2 proposal, meaning they should be part of a future update to the language. Let's take a look at what decorators are, and how they can be used to make your code cleaner and more easily understandable.

What is a Decorator?

In its simplest form, a decorator is simply a way of wrapping one piece of code with another — literally "decorating" it. This is a concept you might well have heard of previously as **functional composition**, or **higher-order functions**.

This is already possible in standard JavaScript for many use cases, simply by calling on one function to wrap another:

```
function doSomething(name) {
  console.log('Hello, ' + name);
}

function loggingDecorator(wrapped) {
  return function() {
    console.log('Starting');
    const result = wrapped.apply(this, arguments);
    console.log('Finished');
    return result;
  }
}

const wrapped = loggingDecorator(doSomething);
```

This example produces a new function — in the variable `wrapped` — that can be called exactly the same way as the `doSomething` function, and will do exactly the same thing. The difference is that it will do some logging before and after the wrapped function is called:

```
doSomething('Graham');
// Hello, Graham

wrapped('Graham');
// Starting
// Hello, Graham
// Finished
```

How to Use JavaScript Decorators

Decorators use a special syntax in JavaScript, whereby they are prefixed with an `@` symbol and placed immediately before the code being decorated.

Note: at the time of writing, the decorators are currently in "Stage 2 Draft" form, meaning that they are mostly finished but still subject to changes.

It's possible to use as many decorators on the same piece of code as you desire, and they'll be applied in the order that you declare them.

For example:

```
@log()
@immutable()
class Example {
  @time('demo')
  doSomething() {
    //
  }
}
```

This defines a class and applies three decorators — two to the class itself, and one to a property of the class:

- `@log` could log all access to the class

- `@immutable` could make the class immutable — maybe it calls `Object.freeze` on new instances
- `@time` will record how long a method takes to execute and log this out with a unique tag.

At present, using decorators requires transpiler support, since no current browser or Node release has support for them yet. If you're using Babel, this is enabled simply by using the transform-decorators-legacy plugin.

Note: the use of the word "legacy" in this plugin is because it supports the Babel 5 way of handling decorators, which might well be different from the final form when they're standardized.

Why Use Decorators?

Whilst functional composition is already possible in JavaScript, it's significantly more difficult — or even impossible — to apply the same techniques to other pieces of code (e.g. classes and class properties).

The decorator proposal adds support for class and property decorators that can be used to resolve these issues, and future JavaScript versions will probably add decorator support for other troublesome areas of code.

Decorators also allow for a cleaner syntax for applying these wrappers around your code, resulting in something that detracts less from the actual intention of what you're writing.

Different Types of Decorator

At present, the only types of decorator that are supported are on classes and members of classes. This includes properties, methods, getters, and setters.

Decorators are actually nothing more than functions that return another function, and that are called with the appropriate details of the item being decorated. These decorator functions are evaluated once when the program first runs, and the decorated code is replaced with the return value.

Class member decorators

Property decorators are applied to a single member in a class — whether they are properties, methods, getters, or setters. This decorator function is called with three parameters:

- `target` : the class that the member is on.
- `name` : the name of the member in the class.
- `descriptor` : the member descriptor. This is essentially the object that would have been passed to Object.defineProperty.

The classic example used here is `@readonly` . This is implemented as simply as:

```
function readonly(target, name, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

Literally updating the property descriptor to set the “writable” flag to false.

This is then used on a class property as follows:

```
class Example {
  a() {}
  @readonly
  b() {}
}

const e = new Example();
e.a = 1;
e.b = 2;
// TypeError: Cannot assign to read only property 'b' of object '#<Example>'
```

But we can do better than this. We can actually replace the decorated function with different behavior. For example, let’s log all of the inputs and outputs:

```
function log(target, name, descriptor) {
  const original = descriptor.value;
  if (typeof original === 'function') {
    descriptor.value = function(...args) {
      console.log(`Arguments: ${args}`);
      try {
        const result = original.apply(this, args);
        console.log(`Result: ${result}`);
        return result;
      } catch (e) {
        console.log(`Error: ${e}`);
        throw e;
      }
    }
  }
  return descriptor;
}
```

This replaces the entire method with a new one that logs the arguments, calls the original method and then logs the output.

Note that we’ve used the spread operator here to automatically build an array from all of the arguments provided, which is the more modern alternative to the old `arguments` value.

We can see this in use as follows:

```
class Example {
  @log
  sum(a, b) {
    return a + b;
  }
}

const e = new Example();
e.sum(1, 2);
// Arguments: 1,2
// Result: 3
```

You'll notice that we had to use a slightly funny syntax to execute the decorated method. This could cover an entire article of its own, but in brief, the `apply` function allows you to call the function, specifying the `this` value and the arguments to call it with.

Taking it up a notch, we can arrange for our decorator to take some arguments. For example, let's re-write our `log` decorator as follows:

```
function log(name) {
  return function decorator(t, n, descriptor) {
    const original = descriptor.value;
    if (typeof original === 'function') {
      descriptor.value = function(...args) {
        console.log(`Arguments for ${name}: ${args}`);
        try {
          const result = original.apply(this, args);
          console.log(`Result from ${name}: ${result}`);
          return result;
        } catch (e) {
          console.log(`Error from ${name}: ${e}`);
          throw e;
        }
      }
    }
    return descriptor;
  };
}
```

This is getting more complex now, but when we break it down we have this:

- A function, `log`, that takes a single parameter: `name`.
- This function then returns a function that *is itself a decorator*.

This is identical to the earlier `log` decorator, except that it makes use of the `name` parameter from the outer function.

This is then used as follows:

```
class Example {
  @log('some tag')
  sum(a, b) {
    return a + b;
  }
}

const e = new Example();
e.sum(1, 2);
// Arguments for some tag: 1,2
// Result from some tag: 3
```

Straight away we can see that this allows us to distinguish between different log lines using a tag that we’ve supplied ourselves.

This works because the `log('some tag')` function call is evaluated by the JavaScript runtime straight away, and then the response from that is used as the decorator for the `sum` method.

Class decorators

Class decorators are applied to the entire class definition all in one go. The decorator function is called with a single parameter which is the constructor function being decorated.

Note that this is applied to the constructor function and not to each instance of the class that is created. This means that if you want to manipulate the instances you need to do so yourself by returning a wrapped version of the constructor.

In general, these are less useful than class member decorators, because everything you can do here you can do with a simple function call in exactly the same way. Anything you do with these needs to end up returning a new constructor function to replace the class constructor.

Going back to our logging example, let’s write one that logs the constructor parameters:

```
function log(Class) {
  return (...args) => {
    console.log(args);
    return new Class(...args);
  };
}
```

Here we are accepting a class as our argument, and returning a new function that will act as the constructor. This simply logs the arguments and returns a new instance of the class constructed with those arguments.

For example:

```
@log
class Example {
  constructor(name, age) {
  }
}

const e = new Example('Graham', 34);
// [ 'Graham', 34 ]
console.log(e);
// Example {}
```

We can see that constructing our Example class will log out the arguments provided and that the constructed value is indeed an instance of `Example`. Exactly what we wanted.

Passing parameters into class decorators works exactly the same as for class members:

```
function log(name) {
  return function decorator(Class) {
    return (...args) => {
      console.log(`Arguments for ${name}: args`);
      return new Class(...args);
    };
  }
}

@log('Demo')
class Example {
  constructor(name, age) {}
}

const e = new Example('Graham', 34);
// Arguments for Demo: args
console.log(e);
// Example {}
```

Real World Examples

Core decorators

There's a fantastic library called Core Decorators that provides some very useful common decorators that are ready to use right now. These generally allow for very useful common functionality (e.g. timing of method calls, deprecation warnings, ensuring that a value is read-only) but utilizing the much cleaner decorator syntax.

React

The React library makes very good use of the concept of Higher-Order Components. These are simply React components that are written as a function, and that wrap around another component.

These are an ideal candidate for using as a decorator, because there’s very little you need to change to do so. For example, the react-redux library has a function, `connect`, that’s used to connect a React component to a Redux store.

In general, this would be used as follows:

```
class MyReactComponent extends React.Component {}
export default connect(mapStateToProps, mapDispatchToProps)(MyReactComponent);
```

However, because of how the decorator syntax works, this can be replaced with the following code to achieve the exact same functionality:

```
@connect(mapStateToProps, mapDispatchToProps)
export default class MyReactComponent extends React.Component {}
```

MobX

The MobX library makes extensive use of decorators, allowing you to easily mark fields as Observable or Computed, and marking classes as Observers.

Summary

Class member decorators provide a very good way of wrapping code inside a class in a very similar way to how you can already do so for freestanding functions. This provides a good way of writing some simple helper code that can be applied to a lot of places in a very clean and easy-to-understand manner.

The only limit to using such a facility is your imagination!