


Exploring EcmaScript Decorators

 medium.com/google-developers/exploring-es7-decorators-76ecb65fb841

Iterators, generators and array comprehensions; The similarities between JavaScript and Python continue to increase over time and I for one could not be more excited. Today we're going to talk about the next Pythonic proposal for ECMAScript—Decorators, by Yehuda Katz.

Update 07/29/17: Decorators are advancing at TC39. The latest work on them can be found in the proposals repo. Several new examples are also now up.

Update 10/02/18: A command line utility to upgrade your scripts from the legacy decorators proposal to the new one is now available.

The Decorator Pattern

What the heck are decorators anyway? Well, in Python, decorators provide a very simple syntax for calling higher-order functions. A Python decorator is a function that takes another function, extending the behavior of the latter function without explicitly modifying it. The simplest decorator in Python could look like this:

That thing at the very top (`@mydecorator``) is a decorator and isn't going to look that different in ES2016 (ES7), so pay attention! :).

```
@mydecorator
def myfunc():
    pass
```

``@`` indicates to the parser that we're using a decorator

while *mydecorator* references a function by that name. Our decorator takes an argument (the function being decorated) and returns the same function with added functionality.

Decorators are helpful for anything you want to transparently wrap with extra functionality. These include memoization, enforcing access control and authentication, instrumentation and timing functions, logging, rate-limiting, and the list goes on.

Decorators in ES5 and ES2015 (aka ES6)

In ES5, implementing imperative decorators (as pure functions) is quite trivial. In ES2015 (previously ES6), while classes support extension, we need something better when we have multiple classes that need to share a single piece of functionality; something with a better method of distribution.

Yehuda's decorators proposal seeks to enable annotating and modifying JavaScript classes, properties and object literals at design time while keeping a syntax that's declarative.

Let's look at some ES2016 decorators in action!

ES2016 Decorators in action

So remember what we learned from Python. An ES2016 decorator is an expression which returns function and can take a target, name and property descriptor as arguments. You apply it by prefixing the decorator with an `@` character and placing this at the very top of what you are trying to decorate. Decorators can be defined for either a class or property.

Decorating a property

Let's take a look at a basic Cat class:

```
class Cat {  
  meow() { return `${this.name} says Meow!`; }  
}
```

Evaluating this class results in installing the meow function onto `Cat.prototype`, roughly like this:

```
Object.defineProperty(Cat.prototype, 'meow', {  
  value: specifiedFunction,  
  enumerable: false,  
  configurable: true,  
  writable: true  
});
```

Imagine we want to mark a property or method name as not being writable. A decorator precedes the syntax that defines a property. We could thus define a `@readonly` decorator for it as follows:

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

and add it to our meow property as follows:

```
class Cat {  
  @readonly  
  meow() { return `${this.name} says Meow!`; }  
}
```

A decorator is just an expression that will be evaluated and has to return a function. This is why `@readonly` and `@something(parameter)` can both work.

Now, before installing the descriptor onto `Cat.prototype`, the engine first invokes the decorator:

```
let descriptor = {  
  value: specifiedFunction,  
  enumerable: false,  
  configurable: true,  
  writable: true  
};  
  
// The decorator has the same signature as `Object.defineProperty`,  
// and has an opportunity to intercede before the relevant  
// `defineProperty` actually occurs  
descriptor = readonly(Cat.prototype, 'meow', descriptor) || descriptor;  
Object.defineProperty(Cat.prototype, 'meow', descriptor);
```

Effectively, this results in meow now being read only. We can verify this behaviour as follows:

```
var garfield = new Cat();
garfield.meow = function() {
  console.log('I want lasagne!');
}
```

// Exception: Attempted to assign to readonly property

Bees-knees, right? We're going to look at decorating classes (rather than just properties) in just a minute, but let's talk about libraries for a sec. Despite its youth, libraries of 2016 decorators are already beginning to appear including <https://github.com/jayphelps/core-decorators.js> by Jay Phelps.

Similar to our attempt at readonly props above, it includes its own implementation for `@readonly`, just an import away:

```
import { readonly } from 'core-decorators';

class Meal {
  @readonly
  entree = 'steak';
}

var dinner = new Meal();
dinner.entree = 'salmon';
// Cannot assign to read only property 'entree' of [object Object]
```

It also includes other decorator utilities such as `@deprecated`, for those times when an API requires hints that methods are likely to change:

Calls console.warn() with a deprecation message. Provide a custom message to override the default one. You can also provide an options hash with a url, for further reading.

```

import { deprecate } from 'core-decorators';

class Person {
  @deprecate
  facepalm() {}

  @deprecate('We stopped facepalming')
  facepalmHard() {}

  @deprecate('We stopped facepalming', { url: 'http://knowyourmeme.com/memes/facepalm' })
  facepalmHarder() {}
}

let captainPicard = new Person();

captainPicard.facepalm();
// DEPRECATION Person#facepalm: This function will be removed in future versions.

captainPicard.facepalmHard();
// DEPRECATION Person#facepalmHard: We stopped facepalming

captainPicard.facepalmHarder();
// DEPRECATION Person#facepalmHarder: We stopped facepalming
//
// See http://knowyourmeme.com/memes/facepalm for more details.
//

```

Decorating a class

Next let's look at decorating classes. In this case, per the proposed specification, a decorator takes the target constructor. For a fictional `MySuperHero` class, we can define a simple decorator for it as follows using a `@superhero` decoration:

```

function superhero(target) {
  target.isSuperhero = true;
  target.power = 'flight';
}

@superhero
class MySuperHero() {}

console.log(MySuperHero.isSuperhero); // true

```

This can be expanded further, enabling us to supply arguments for defining our decorator function as a factory:

```
function superhero(isSuperhero) {  
  return function(target) {  
    target.isSuperhero = isSuperhero  
  }  
}
```

```
@superhero(true)  
class MySuperheroClass() { }  
console.log(MySuperheroClass.isSuperhero) // true
```

```
@superhero(false)  
class MySuperheroClass() { }  
console.log(MySuperheroClass.isSuperhero) // false
```

ES2016 Decorators work on property descriptors and classes. They automatically get passed property names and the target object, as we'll soon cover. Having access to the descriptor allows a decorator to do things like changing a property to use a getter, enabling behaviour that would otherwise be cumbersome such as automatically binding methods to the current instance on first access of a property.

ES2016 Decorators and Mixins

I've thoroughly enjoyed reading Reg Braithwaite's recent article on [ES2016 Decorators as mixins](#) and the precursor [Functional Mixins](#). Reg proposed a helper that mixes behaviour into any target (class prototype or standalone) and went on to describe a version that was class specific. The functional mixin that mixes instance behaviour into a class's prototype looked like this:


```

function mixin(behaviour, sharedBehaviour = {}) {
  const instanceKeys = Reflect.ownKeys(behaviour);
  const sharedKeys = Reflect.ownKeys(sharedBehaviour);
  const typeTag = Symbol('isa');

  function _mixin(clazz) {
    for (let property of instanceKeys) {
      Object.defineProperty(clazz.prototype, property, { value: behaviour[property] });
    }
    Object.defineProperty(clazz.prototype, typeTag, { value: true });
    return clazz;
  }
  for (let property of sharedKeys) {
    Object.defineProperty(_mixin, property, {
      value: sharedBehaviour[property],
      enumerable: sharedBehaviour.propertyIsEnumerable(property)
    });
  }
  Object.defineProperty(_mixin, Symbol.hasInstance, {
    value: (i) => !!i[typeTag]
  });
  return _mixin;
}

```

Great. We can now define some mixins and attempt to decorate a class using them. Let's imagine we have a simple `ComicBookCharacter` class:

```

class ComicBookCharacter {
  constructor(first, last) {
    this.firstName = first;
    this.lastName = last;
  }
  realName() {
    return this.firstName + ' ' + this.lastName;
  }
};

```

This may well be the world's most boring character, but we can define some mixins which would provide behaviours granting `SuperPowers` and a `UtilityBelt`. Let's do this using Reg's mixin helper:

```

const SuperPowers = mixin({
  addPower(name) {
    this.powers().push(name);
    return this;
  },
  powers() {
    return this._powers_pocessed || (this._powers_pocessed = []);
  }
});

const UtilityBelt = mixin({
  addToBelt(name) {
    this.utilities().push(name);
    return this;
  },
  utilities() {
    return this._utility_items || (this._utility_items = []);
  }
});

```

With this behind us, we can now use the `@` syntax with the names of our mixin functions to decorate the `ComicBookCharacter` with our desired behaviour. Note how we're prefixing the class with multiple decorator statements:

```

@SuperPowers
@UtilityBelt
class ComicBookCharacter {
  constructor(first, last) {
    this.firstName = first;
    this.lastName = last;
  }
  realName() {
    return this.firstName + ' ' + this.lastName;
  }
};

```

Now, let's use what we've defined to craft a Batman character.


```

const batman = new ComicBookCharacter('Bruce', 'Wayne');
console.log(batman.realName());
// Bruce Wayne

batman
  .addToBelt('batarang')
  .addToBelt('cape');

console.log(batman.utilities());
// ['batarang', 'cape']

batman
  .addPower('detective')
  .addPower('voice sounds like Gollum has asthma');

console.log(batman.powers());
// ['detective', 'voice sounds like Gollum has asthma']

```

These decorators for classes are relatively compact and I could see myself using them as an alternative to function invocation or as helpers for higher-order components.

Note: @WebReflection has some alternatives takes on the mixin pattern used in this section which you can find in the comments [here](#).

Enabling Decorators via Babel

Decorators (at the time of writing) are still but a proposal. They haven't yet been approved. That said, thankfully Babel supports transpilation of the syntax in an experimental mode, so most of the samples from this post can be tried out with it directly.

If using the Babel CLI, you can opt-in to Decorators as follows:

```
$ babel --optional es7.decorators
```

or alternatively, you can switch on support using a transformer:

```
babel.transform("code", { optional: ["es7.decorators"] });
```

There's even an [online Babel REPL](#); enable decorators by hitting the "Experimental" checkbox. Try it out!

Interesting experiments

Paul Lewis, who luckily enough I sit next to, has been [experimenting](#) with decorators as a means for rescheduling code that reads and writes to the DOM. It borrows ideas from Wilson Page's FastDOM, but provides an otherwise small API surface. Paul's read/write decorators can also warn you via the console if you're calling methods or properties during a @write (or change the DOM during @read) that trigger layout.

Below is a sample from Paul's experiment, where attempting to mutate the DOM inside a @read causes a warning to be thrown to the console:

```
class MyComponent {
  @read
  readSomeStuff () {
    console.log('read');

    // Throws a warning.
    document.querySelector('.button').style.top = '100px';
  }

  @write
  writeSomeStuff () {
    console.log('write');

    // Throws a warning.
    document.querySelector('.button').focus();
  }
}
```

Go try Decorators now!

In the short term, ES2016 decorators are useful for declarative decoration and annotations, type checking and working around the challenges of applying decorators to ES2015 classes. In the long term, they could prove very useful for static analysis (which could give way to tools for compile-time type checking or autocompletion).

They aren't that different from decorators in classic OOP, where the pattern allows an object to be decorated with behaviour, either statically or dynamically without impacting objects from the same class. I think they're a neat addition. The semantics for decorators on class properties are still in flux, however keep an eye on Yehuda's repo for updates.

Library authors are currently discussing where Decorators may replace mixins and there are certainly ways in which they could be used for higher-order components in React.

I'm personally excited to see an increase in experimentation around their use and hope you'll give them a try using Babel, identify repurposeable decorators and maybe you'll even share your work like Paul did :)