

Decorators, Stage 2 update: Statically Analyzable

Daniel Ehrenberg
Igalia, in partnership with Bloomberg
March 2019 TC39

Popularity

- The JS ecosystem makes heavy use of
 - TypeScript "experimental" decorators
 - Babel "legacy" decorators

i.e. roughly, the proposal as of 2014

- 2015: Proposal switched to descriptors

many developers didn't

Excitement

- JS developers are really excited about decorators!
- Features people make use of or ask for:
 - Initially proposed features
 - ad-hoc implementation limitations
 - Features of the Stage 2 proposal
 - Interaction with private and fields
 - Scheduling callbacks on construction
 - Features not yet proposed (mixins, functions, let)

Some goals

- Decorators should be fast in implementations:
 - Transpilers
 - JS engines

- Decorators should be easy to use:
 - Using someone else's decorators
 - Writing your own

History of decorator proposals

In 2014/2015: TypeScript "experimental"/Babel "legacy"

TypeScript

Quick Start

Documentation


Download

Connect

Playground

Fork me on GitHub

TypeScript 3.3 is now available. [Download](#) our latest version today!

 This site uses cookies for analytics, personalized content and ads. By continuing to browse this site, you agree to this use.

[Learn more](#)

Documentation

Tutorials

What's New

Handbook

Declaration Files

Project Configuration

Decorators

Introduction

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members. Decorators are a [stage 2 proposal](#) for JavaScript and are available as an experimental feature of TypeScript.

NOTE Decorators are an experimental feature that may change in future releases.

To enable experimental support for decorators, you must enable the `experimentalDecorators` compiler option either on the command line or in your `tsconfig.json`:

Command Line:

```
tsc --target ES5 --experimentalDecorators
```

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }

  @enumerable(false)
  greet() {
    return "Hello, " + this.greeting;
  }
}
```

We can define the `@enumerable` decorator using the following function declaration:

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    descriptor.enumerable = value;
  };
}
```

2016: Descriptor-based decorators

```
interface MemberDescriptor {  
    kind: "property"  
    key: string,  
    isStatic: boolean,  
    descriptor: PropertyDescriptor,  
    extras?: MemberDescriptor[]  
    finisher?: (klass): void;  
}
```

```
interface ClassDescriptor {  
    kind: "class",  
    constructor: Function,  
    heritage: Object | null,  
    elements: MemberDescriptor[]  
}
```


2017-2019: Descriptor-based proposal grows and evolves

— — —

parameter descriptor	@decorator class	@decorator method()	@decorator field	@decorator get field() set field()
{				
kind:	"class"	"method"	"field"	"accessor" ⁴
elements:	Array of member descriptors ¹	-	-	-
key:	-	method name	property name	property name
placement:	-	"prototype" "static"	"own" "static"	"prototype" "static"
initialize:	-	-	Function used to set the initial value of the field ²	-
method:	-	method function	- ³	-
get:	-	-	-	getter function
set:	-	-	-	setter function
writable:	-	true	true	- ⁵
configurable:	-	true	true	true
enumerable:	-	false	false	false
}				

return descriptor (optional)	class	method()	field	getter/setter	Hooks
{					
kind:	"class"	"method"	"field"	"accessor"	"hook" ¹⁰
elements:	Array of member descriptors ⁶	-	-	-	-
key:	-	method name ⁸	field name ⁸	field name ⁸	-
placement:	-	"prototype" "static" "own"	"prototype" "static" "own"	"prototype" "static" "own"	"prototype" "static" "own"
extras:	-	Array of member descriptors ⁷	Array of member descriptors ⁷	Array of member descriptors ⁷	-
initialize:	-	-	Function used to set the initial value of the field	- ⁹	-
method:	-	method function	-	- ⁹	-
get:	-	-	-	getter	-
set:	-	-	-	setter	-
writable:	-	true false	true false	- ⁹	-
configurable:	-	true false	true false	true false	-
enumerable:	-	false true	false true	false true	-
start:	-	-	-	-	Function for effect ¹⁰
finish:	-	-	-	-	Function for effect ¹⁰
replace:	-	-	-	-	Function for replacement ¹⁰
}					

Language-level issues with decorators

Complex to write decorators

- Need to understand `Object.defineProperty` deeply
 - Even with original decorators
 - Ecosystem abstraction layers

- With Stage 2 proposal, expanded descriptor language

Difficult to extend over time

- Past discussion about "elements"
- Passing new fields back from decorators
- Mixin feature request

Ember's experience
and issues with the
original decorators
proposal

Why are descriptor-based decorators slow?

Transpiler implementations

- Lots of code generated
- Elements array shows up big
- Traverse many dynamic data structures
 - Including in the constructor
 - Even without class elements
- It takes time to allocate class declaration
 - Lots of descriptors to create and parse
- Unclear how to generate good code

Native implementations

- Fundamentally, similar overhead!
- Several things observable which weren't before:
 - E.g., Initializer thunks
- Class declaration may run multiple times and have different shapes different times
- Fancy optimizations harder to do in interpreter (but the JIT may do them)
- Best case: Class structure available when generating bytecode
- Thanks for the detailed analysis, Sathya Gunasekaran!

**It's nice if it's easier
to see what's going on**

**Decorators tend to
have a fixed shape**

Built-in decorators and composition

The idea

- Basic building blocks: Built-in decorators
- Compose to make JavaScript-defined decorators
- @decorators are separate, static, lexically scoped

Built-in decorators

- @wrap
 - @register
 - @expose
 - @initialize
-
- More in follow-on proposals

Decorators defined in JavaScript

- Compose built-in decorators
- Pass computed arguments to them

```
decorator @foo { @bar @baz @bing }
```

@register

@register

```
class C {  
  @register(f) method() { }  
}
```

```
class C {  
  method() { }  
}  
f(C.prototype, "method");
```


@register

```
@register(f)
class C { }
```

```
class C {
    method() { }
}
f(C);
```

@defineElement

```
import { @defineElement } from
  "./defineElement.mjs";

@defineElement('my-class')
class MyClass extends HTMLElement {
  /* ... */
}
```

```
// defineElement.mjs

export decorator @defineElement(
  name, options) {
  @register(klass =>
    customElements.define(
      name, klass, options))
}
```

@wrap

@wrap

```
class C {  
  @wrap(f) method() { }  
}
```

```
class C {  
  method() { }  
}  
C.prototype.method =  
  f(C.prototype.method);
```

@wrap

```
@wrap(f)  
class C { }
```

```
class C { }  
C = f(C);
```

@logged

```
import { @logged } from "./logged.mjs";
class C {
  @logged
  method(arg) {
    this.#x = arg;
  }
  @logged
  set #x(value) { }
}
new C().method(1);
// starting method with arguments 1
// starting set #x with arguments 1
// ending set #x
// ending method
```

```
// logged.mjs
export decorator @logged {
  @wrap(f => {
    const name = f.name;
    function wrapped(...args) {
      console.log(`starting ${name}
with arguments ${args.join(", ")}`);
      f.call(this, ...args);
      console.log(`ending ${name}`);
    }
    wrapped.name = name;
    return wrapped;
  })
}
```

@initialize

@initialize

```
class C {  
    @initialize(f) a = b;  
}
```

```
class C {  
    constructor() {  
        f(this, "a", b);  
    }  
}
```


@initialize

```
@initialize(f)
class C { }
```

```
class C {
  constructor() {
    f(this);
  }
}
```

@bound

```
import { @bound } from "./bound.mjs";

class Foo {
  x = 1;
  @bound method() {
    console.log(this.x);
  }
  queueMethod() {
    setTimeout(this.method, 1000);
  }
}

new Foo().queueMethod();
// logs 1, not undefined
```

```
// bound.mjs
export decorator @bound {
  @initialize((instance, name) =>
    instance[name] =
      instance[name].bind(instance))
}
```

@expose

@expose

```
class C {  
  @expose(f) #x;  
}
```

```
class C {  
  @register(proto =>  
    f(proto,  
      "#x",  
      instance => instance.#x,  
      (instance, value) =>  
        instance.#x = value ))  
  #x;  
}
```

@show

```
import { FriendKey, @show } from
  "./friend.mjs";

let key = new FriendKey;
export class Box {
  @show(key) #contents;
}

export function setBox(box, contents) {
  return key.set(box, "#x", contents);
}

export function getBox(box) {
  return key.get(box, "#x");
}
```

```
export class FriendKey {
  #map = new Map();
  expose(name, get, set) {
    this.#map.set(name, { get, set });
  }
  get(obj, name) {
    return this.#map.get(name).get(obj);
  }
  set(obj, name, value) {
    return this.#map.get(name)
      .set(obj, value);
  }
}

export decorator @show(key) {
  @expose((target, name, get, set) =>
    key.expose(name, get, set))
}
```

Common features of decorators

Syntax

- Uses DecoratorList syntax before an existing construct
- Doesn't change the syntax of what's decorated
- No early errors, but rather runtime errors

Semantics/phasing

- Decorator arguments are evaluated at runtime
 - In classes: Interspersed with computed property names
- In spec-land, they run at runtime
 - But it's always apparent which built-in decorators are used
- When code is executing multiple times, different arguments, but the same built-in decorators

Potential future built-in decorators

Decorators for other syntactic forms

- Functions
- Parameters
- Variable declarations
- Blocks
- Labels
- Numeric literals
- Object literals
- Object properties
- (not expressions)

Built-in support for common scenarios

— — —

- @bound
- @tracked
- @reader
- @set

Error checking

- @assertClass
- @assertMethod
- @assertField
- @assertPrivate
- etc

Property descriptor/placement change

- @own
- @prototype
- @static
- @enumerable
- @nonenumerable
- @writable
- @nonwritable
- @configurable
- @nonconfigurable

Statically change the structure of the class

- Adding a private field
 - Converting a field to an accessor
 - Adding a mixin (even to a base class)
 - etc.
-
- May use some kind of descriptor (as input, not output)
 - Or, may use some yet-to-be-designed mini-language
 - Could be a separate declaration form

**Implementation notes:
Some complexity, but more optimizable**

Could implement this directly

- Can be executed dynamically
 - Similar to the previous proposal
 - This is how the spec will be written

- But, to take advantage of the guarantees and optimize...

JavaScript
compilation becomes
non-local

Transpilers: `.decorators.json`

- Describes which built-in decorators a composed decorator breaks down into
- Referenced when compiling a file which uses the decorators
- Separate runtime representation used for arguments

- Different from how tooling works now, but seems doable

Custom decorators in tooling

- Decorators could be a good basis for more general macros
- To start: Follow lead of [babel-plugin-macros](#) and let tools define decorators which are tree transforms
 - Then, they can be composed!
 - So, prototype proposed built-in decorators

Native JS implementations: Bytecode based on dependencies

- Classes with decorators are lazy-parsed if their decorator has not yet been parsed
- Cached bytecode is invalidated when imported decorators change

- Generally positive feedback from browsers
- Different from how engines work now, but a much better starting point for optimization than the last proposal

Next steps

Recommendations for authors of decorators today

- The original decorators proposal is broadly supported
 - The newer January 2019 "Stage 2" proposal is not
 - **Just keep using "legacy"/"experimental" decorators**
 - Tools may need to maintain support for set-based fields until it's possible to transition to Stage 3 decorators
-
- Goal of this proposal: For **users** (not authors) of decorators, upgrading to standard should be a codemod

Prototyping this proposal

- Write specification
 - Implement these new decorators in Babel
 - Including some "post-MVP" decorators
 - Try out the new decorators
 - Collect feedback
-
- Propose for Stage 3 after some months of stability+use