Decorators vs Metadata Annotations

Status: draft, URL: <http://goo.gl/vdPWFt>  
Authors: [misko@google.com](mailto:misko@google.com), [iminar@google.com](mailto:iminar@google.com)

*This document is published to the web as part of the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk) *folder*

# Objective

There are two competing proposals for associating metadata with code being made for ES7. Decorators and metadata annotations. While superficially similar, under the hood they have very different properties, with different strengths and weaknesses. In this doc, we compare the two approaches and their suitability for attaching metadata to code.

This document doesn't try to capture semantics and implementation details for the actual metadata storage. These a separate problem and the solution can very likely be the same for both the decorator and annotation approach of expressing the association between metadata and code.

# 

# Executive Summary

We believe that attaching metadata using annotations is preferable to attaching them using decorators. We perceive decorators as a more powerful and capable concept, however all of these additional capabilities come with major flaws and are undesirable in real application and in fact make reasoning about the code harder, which also impacts ability of VMs and tools to analyze, optimize and transform code. Because of these issues, the constraints that metadata annotations have are, in fact, guarantees that enable reasoning about the code and therefore should be seen as a feature rather than weakness.

|  |  |  |
| --- | --- | --- |
| **Properties / Proposal** | **Decorators** | **Annotations** |
| **Main Purpose** | augment code | describe code |
| **Behavior** | dynamic invocation | static declaration |
| **Ability to work with properties, classes, functions, ...** | yes | yes |
| **Ability to interact with the code being annotated/decorated** | yes | no |
| **Negative Impact on ability to reason about the runtime behavior of the code** | large | none / limited |
| **Composition** | problematic[[1]](#footnote-0) | yes |
| **Suitable for static analysis and tooling[[2]](#footnote-1)** | no | yes |
| **Impact on performance and VM optimizations** | likely | unlikely |
| **Additional capabilities** | many (metaprogramming, mixins, syntactic sugar, AOP)[[3]](#footnote-2) | annotating annotations |

# 

# Background / Prior Art

* Decorators [proposal](https://github.com/jonathandturner/brainstorming/blob/master/README.md), TC39 [notes](https://github.com/rwaldron/tc39-notes/blob/master/es6/2014-04/apr-10.md#decorators-for-es7).
* TypeScript discussion: [Issue#1557](https://github.com/Microsoft/TypeScript/issues/1557).
* AtScript [proposal](https://docs.google.com/document/u/0/d/11YUzC-1d0V1-Q3V0fQ7KSit97HnZoKVygDxpWzEYW0U/mobilebasic?viewopt=127).

# The end goal and two ways to get there

From our point of view the end goal is to add a capability to the language that would enable association of metadata with code via a developer-friendly syntax and capability to introspect this metadata at run time.

This goal can be achieved in at least two ways:

1. static **metadata annotations** that are *expressed* via syntactic sugar, which simply populates key-value store associated with the code entity (e.g. function, class, property, etc).
2. dynamic **decorators** that are *invoked* via syntactic sugar. Once the decorator is invoked it can do whatever it wants with the code entity, including populating key-value store associated with the code entity, augmenting the code entity or replacing the entity with a different entity.

At first sight, decorators look as a strict superset of metadata annotations. They can do the same and much more. Let's explore if this is really true.

# Decorators as (strict) superset of annotations

It is true that syntactically everything that could be expressed in annotations can be expressed in decorators. The argument here is not if the decorators are more powerful, but rather if the decorators' added benefits are useful and justifies the costs which they imply.

The core premise of this document is that for the purposes of associating metadata with code via decorators we lose guarantees that allow reasoning about the code by humans, VMs and tools.

Because of these implications, it is not correct to say that decorators are *strict superset* of metadata annotations. They are just an alternative way of associating metadata with a code unit, which comes with additional capabilities unrelated to this main task but also limitations that results in lower utility of this main task.

# Is lack of capabilities a capability on it's own?

<todo>

- intentional lack of capabilities creates guarantees that other capabilities can be built on top of

- examples: immutable data structures, promises, pure functions, const vs let

- why do we need these guarantees?: performance optimizations, reasoning about code, tooling (static analysis, documentation generation, minification, obfuscation)

</todo>

# Extra capabilities of decorators (compared to metadata annotations):

1. Decorators can dynamically change shape of classes
2. Decorators can register code with global registries
3. Decorators can wrap the function with other functions — for metaprogramming or AOP

All of these capabilities are result of a decorators being a regular code that gets invoked by the VM when code is being loaded. As currently proposed, the decorators have access only to three contexts: the code entity being decorated (e.g. function, class, property name and its descriptor), parameters optionally passed into the decorator at each call site and global context of the VM (or module in which decorators are defined).

The dynamic invocation of user provided function and limited access to contextual information has several severe implications:

* **Promotes bad practices** - decorators can only interact with global state and object being decorated. Global state should be minimized, and is considered an anti-pattern. Decoration of code in AOP style has proven to be problematic in other platforms.
* **Unpredictable performance** - the decorator code must be executed during code load time. Since the code is free to do anything, no guarantee about predictable performance can be made (simply adding one decorator can result in a well performing code with no unexpected side effects, while another decorator, could make the system unusably slow).
* **Interferes with VM optimizations** - decorators can have (global) side-effects, as such they must be executed fully and in the order defined. This means that VMs will have a hard time performing optimizations, especially deferred parsing of code.
* **Interferes with tools analyzing code** - tools like TypeScript compiler and Closure compiler depend on their ability to statically analyze code. Since decorators are imperative in nature and have capability to change structure of objects, they make it difficult for the tools to understand the implications of using decorators without having a declarative way to describe the decorator (this could be solved by having type and metadata annotations for decorators :-) )

# Ignore / Under construction / Staging Area / Obsolete Stuff

[Annotations recognized by Closure](https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml#JSDoc_Tag_Reference)

## Summary of problems with Decorators

The reasons are:

* Order of applying decorators matter, which make them difficult to compose.
* Using decorators to register code with global registry is an anti-pattern
* Decorators are harder to reason about because they need to be executed.
* Having the ability to wrap functions using decorators has a very limited use as demonstrated by lack of adoption of AOP.
* Because decorators must execute they further complicate the code bootstrap process.
* Decorators make VM optimizations very difficult and static analysis of code impossible

## Using decorators and annotation for associating metadata with code

Let's take a simple example of in which we associate an annotation with parameters with a class or constructor of a class

|  |  |
| --- | --- |
|  | @Inject(Dependency1, Dependency2)  class MyClass {  constructor(dep1, dep2)  } |

The above examples can be implemented in two ways:

|  |  |
| --- | --- |
| **Decorator** | **Annotation** |
| function Inject(...deps) {  return function(type) {  var anno = type.annotation;  if (!ano) {  ano = new Map();  type.annotations = ano;  }  ano.set(Inject, deps);  }  } | function Inject(...deps) {  this.deps = deps;  } |

While not exactly same, the above two approaches are essentially equivalent. The advantage of annotation based approach is that it is simpler, shorter, has no side effects and easier to comprehend.

## 

## Issues with decorators

### Global Registry

One argument for decorators is that one could use them to register the components with a framework. Our opinion is that it is actually an anti-pattern because there is no way for the decorator to get a hold of the registry without relying on [global mutable state](https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=global%20state%20antipattern).

|  |
| --- |
| @Component(...)  class MyClass { ... } |

The problem arises because @Component has no easy way of getting hold of the registry, and is forced to rely on global state. For example:

|  |
| --- |
| import {globalRegistry} from '...';  function Component(...args) {  return function (type) {  globalRegistry.register(type, args);  }  } |

Now you can argue that globalRegistry is not really global, because the import system can inject a different instance into different files. But that does not work in practice for several reasons:

* imports are fixed per file
* imports can not be reconfigured once the code is loaded
* import configuration is deployment concern, not an application wiring concern.

There is a simple acid test. "Can I bootstrap two different applications on the same page using the same framework?" The short answer is no, because there is no way to control if the decorator should register the component with application A, B or both. And if you think that having two apps on the same page using same framework is not a common issue, consider that tests are a large collection of small apps on a single page.

Decorators encourage creation of global state.

### AOP has Limited Value

Another common use case is that decorators can wrap the function and provide added benefits such as exception handling or logging. It is true that being able to wrap a function can be convenient. However for the wrapped function to provide value, it needs to be able to get a hold of some external services. For example a logging function, needs to get a hold of a logger, an exception handling function needs to get a hold of an exception handler and so on. A function which can not coordinate with any external services is of limited value.

Here the decorator has the same issue as "Global Registry". The only way to get a hold of a service is to look it up in a global state. While there is only one console for logging and so global state is sufficient, different applications will have different exception handlers. There is no way of controlling which exception handler a decorator should get without resorting to global state. Decorators lack proper dependency management.

|  |
| --- |
| @log('something')  @handleException  function doSomething() { ... } |

|  |
| --- |
| // console is global, so it is sort of OK  import {console} from '...';  function log(name) {  return function (type) {  return function() {  console.log('before: ' + name);  var ret = type.apply(this, arguments);  console.log('before: ' + name);  return ret;  }  }  }  // Only one globalExceptionHandler can exist, not good!  import {globalExceptionHandler} from '...';  function handleException(name) {  return function () {  return function() {  try {  return type.apply(this, arguments);  } catch(e) {  globalExceptionHandler(e);  }  }  }  } |

Ignoring the issue of global state, decorators which wrap other functions are essentially providing AOP (aspect-oriented-programing) pointcuts. AOP has shown very limited value in practice. There are AOP systems which are useful, such as [GUICE AOP](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator), but those are based on annotations (not on decorators). Because they are based on annotations, the framework is responsible for pointcut (not wrapping functions). At runtime GUICE framework can properly inject the dependencies into tho pointcuts without resorting to global state.

### Order of decorators matter

Decorators are not very composable because some decorators can wrap functions. Wrapping functions will hide any monkey patched state from the previous decorator. It is not possible to know which decorators monkey patch and which are wrappers without also understanding their semantic meaning. This makes it difficult to reason about them.

Below are two examples of how one may write code, but one of them will produce a very unexpected behavior.

|  |
| --- |
| @Inject(...)  @Component(...)  @handleException()  class MyClass { ... }  // VS  @handleException()  @Component(...)  @Inject(...)  class MyClass { ... } |

## Could we make decorators work? (a.k.a. fixing decorators)

In order for decorators to be useful they have to address two issues:

1. Proper injection of dependencies into them
2. Order independent composition.

1. Order of declaration matters [↑](#footnote-ref-0)
2. For example documentation metadata (@deprecated) or tooling-only metadata (@immutable, @nosideffects, @override, @private) that should be safely stripped from production code since they have zero impact on runtime. [↑](#footnote-ref-1)
3. There are many possibilities for abuse here and very little options for providing guarantees from the language / platform. [↑](#footnote-ref-2)