Decorators

Stage: 3

Decorators are a proposal for extending JavaScript classes which is widely adopted among developers in transpiler environments, with broad interest in standardization. TC39 has been iterating on decorators proposals for over five years. This document describes a new proposal for decorators based on elements from all past proposals.

This README describes the current decorators proposal, which is a work in progress. For previous iterations of this proposal, see the commit history of this repository.

Introduction

Decorators are *functions* called on classes, class elements, or other JavaScript syntax forms during definition.

```
@defineElement("my-class")
class C extends HTMLElement {
  @reactive accessor clicked = false;
}
```

Decorators have three primary capabilities:

1. They can **replace** the value that is being decorated with a *matching* value that has the same semantics. (e.g. a decorator can replace a method with another method, a field with another field, a class with another class, and so on).
2. They can provide **access** to the value that is being decorated via accessor functions which they can then choose to share.
3. They can **initialize** the value that is being decorated, running additional code after the value has been fully defined. In cases where the value is a member of class, then initialization occurs once per instance.

Essentially, decorators can be used to metaprogram and add functionality to a value, without fundamentally changing its external behavior.

This proposal differs from previous iterations where decorators could replace the decorated value with a completely different type of value. The requirement for decorators to only replace a value with one that has the same semantics as the original value fulfills two major design goals:

- **It should be easy both to use decorators and to write your own decorators.** Previous iterations such as the *static decorators* proposal were complicated for authors and implementers in particular. In this proposal, decorators are plain functions, and are accessible and easy to write.
- **Decorators should affect the thing they're decorating, and avoid confusing/non-local effects.** Previously, decorators could change the decorated value in unpredictable ways, and also add completely new values which were unrelated. This was problematic both for *runtimes*, since it meant decorated values could not be analyzed statically, and for *developers*, since decorated values could turn into completely different types of values without any indicator to the user.

In this proposal, decorators can be applied to the following existing types of values:

- Classes
- Class fields (public, private, and static)
- Class methods (public, private, and static)
- Class accessors (public, private, and static)

In addition, this proposal introduces a new type of class element that can be decorated:

- Class *auto accessors*, defined by applying the `accessor` keyword to a class field. These have a getter and setter, unlike fields, which default to getting and setting the value on a private storage slot (equivalent to a private class field):

```
class Example {
  @reactive accessor myBool = false;
}
```

This new element type can be used independently, and has its own semantics separate from usage with decorators. The reason it is included in this proposal is primarily because there are a number of use cases for decorators which require its semantics, since decorators can only replace an element with a corresponding element that has the same semantics. These use cases are common in the existing decorators ecosystem, demonstrating a need for the capabilities they provide.

Motivation

You might be wondering "why do we need these at all?" Decorators are a powerful metaprogramming feature that can simplify code significantly, but can also feel "magical" in the sense that they hide details from the user, making what's going on under the hood harder to understand. Like all abstractions, in some cases decorators can become more trouble than they're worth.

However, one of the main reasons decorators are still being pursued today, and *specifically* the main reason class decorators are an important language feature, is that they fill a gap that exists in the ability to metaprogram in JavaScript.

Consider the following functions:

```
function logResult(fn) {
  return function(...args) {
    try {
      const result = fn.call(this, ...args);
      console.log(result);
    } catch (e) {
      console.error(result);
      throw e;
    }
    return result;
  }
}

const plusOne = logResult((x) => x + 1);

plusOne(1); // 2
```

This is a common pattern used in JavaScript every day, and is a fundamental power in languages that support closures. This is an example of implementing the [decorator pattern] in plain JavaScript. You can use `logResult` to add logging to any function definition easily, and you can do this with any number of "decorator" functions:

```
const foo = bar(baz(qux(() => /* do something cool */)))
```

In some other languages, like Python, decorators are syntactic sugar for this pattern - they're functions that can be applied to other functions with the `@` symbol, or by calling them directly, to add additional behavior.

So, as it stands today, it is possible to use the decorator *pattern* in JavaScript when it comes to functions, just without the nice `@` syntax. This pattern is also *declarative*, which is important - there is no step between the definition of the function and decoration of it. This means it's not possible for someone to accidentally use the undecorated version of the function, which could cause major bugs and make it very difficult to debug!

However, there is a place where we *can't* use this pattern at all - objects and classes. Consider the following class:

```
class MyClass {
  x = 0;
}
```

How would we go about adding the logging functionality to `x`, so that whenever we get or set it, we log that access? You could do it manually:

```
class MyClass {
  #x = 0;

  get x() {
    console.log('getting x');
    return this.#x;
  }

  set x(v) {
    console.log('setting x');
    this.#x = v;
  }
}
```

But if we're doing this a lot, it would be a pain to add all those getters and setters everywhere. We could make a helper function to do it for us *after* we define the class:

```
function logResult(Class, property) {
  Object.defineProperty(Class.prototype, property, {
    get() {
      console.log(`getting ${property}`);
      return this[`_${property}`];
    },

    set(v) {
      console.log(`setting ${property}`);
      this[`_${property}`] = v;
    }
  })
}

class MyClass {
  constructor() {
    this.x = 0;
  }
}

logResult(MyClass, 'x');
```

This *works*, but if we use a class field it would overwrite the getter/setter we defined on the prototype, so we have to move the assignment to the constructor. It's also done in multiple statements, so the definition itself happens over time and is not declarative. Imagine debugging a class that is "defined" in multiple files, each one adding different decorations as your application boots up. That may sound like a really bad design, but it was not uncommon in the past before classes were introduced! Lastly, there's no way for us to do this with *private* fields or methods. We can't just replace the definition.

Methods are a *little* better, we could do something like this:

```
function logResult(fn) {
  return function(...args) {
    const result = fn.call(this, ...args);
    console.log(result);
    return result;
  }
}

class MyClass {
  x = 0;
  plusOne = logResult(() => this.x + 1);
}
```

While this *is* declarative, it also creates a new closure for each instance of the class, which is a lot of additional overhead at scale.

By making class decorators a language feature, we are plugging this gap and enabling the decorator pattern for class methods, fields, accessors, and classes themselves. This allows developers to easily write abstractions for common tasks, such as debug logging, reactive programming, dynamic type checking, and more.

Detailed Design

The three steps of decorator evaluation:

1. Decorator expressions (the thing after the `@`) are *evaluated* interspersed with computed property names.
2. Decorators are *called* (as functions) during class definition, after the methods have been evaluated but before the constructor and prototype have been put together.

3. Decorators are *applied* (mutating the constructor and prototype) all at once, after all of them have been called.

> The semantics here generally follow the consensus at the May 2016 TC39 meeting in Munich.

1. Evaluating decorators

Decorators are evaluated as expressions, being ordered along with computed property names. This goes left to right, top to bottom. The result of decorators is stored in the equivalent of local variables to be later called after the class definition initially finishes executing.

2. Calling decorators

When decorators are called, they receive two parameters:

1. The value being decorated, or `undefined` in the case of class fields which are a special case.
2. A context object containing information about the value being decorated

Using TypeScript interfaces for brevity and clarity, this is the general shape of the API:

```
type Decorator = (value: Input, context: {
  kind: string;
  name: string | symbol;
  access: {
    get?(): unknown;
    set?(value: unknown): void;
  };
  private?: boolean;
  static?: boolean;
  addInitializer(initializer: () => void): void;
}) => Output | void;
```

`Input` and `Output` here represent the values passed to and returned from a given decorator. Each type of decorator has a different input and output, and these are covered below in more detail. All decorators can choose to return nothing, which defaults to using the original, undecorated value.

The context object also varies depending on the value being decorated. Breaking down the properties:

- `kind`: The kind of decorated value. This can be used to assert that the decorator is used correctly, or to have different behavior for different types of values. It is one of the following values.
  - `"class"`
  - `"method"`
  - `"getter"`
  - `"setter"`
  - `"field"`
  - `"accessor"`

- `name`: The name of the value, or in the case of private elements the *description* of it (e.g. the readable name).
- `access`: An object containing methods to access the value. These methods also get the *final* value of the element on the instance, not the current value passed to the decorator. This is important for most use cases involving access, such as type validators or serializers. See the section on Access below for more details.
- `static`: Whether or not the value is a `static` class element. Only applies to class elements.
- `private`: Whether or not the value is a private class element. Only applies to class elements.
- `addInitializer`: Allows the user to add additional initialization logic to the element or class.

See the Decorator APIs section below for a detailed breakdown of each type of decorator and how it is applied.

3. Applying decorators

Decorators are applied after all decorators have been called. The intermediate steps of the decorator application algorithm are not observable--the newly constructed class is not made available until after all method and non-static field decorators have been applied.

The class decorator is called only after all method and field decorators are called and applied.

Finally, static fields are executed and applied.

Syntax

This decorators proposal uses the syntax of the previous Stage 2 decorators proposal. This means that:

- Decorator expressions are restricted to a chain of variables, property access with `.` but not `[]`, and calls `()`. To use an arbitrary expression as a decorator, `@(expression)` is an escape hatch.
- Class expressions may be decorated, not just class declarations.
- Class decorators may exclusively come before, or after, `export`/`export default`.

There is no special syntax for defining decorators; any function can be applied as a decorator.

Decorator APIs

Class Methods

```
type ClassMethodDecorator = (value: Function, context: {
  kind: "method";
  name: string | symbol;
  access: { get(): unknown };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;
```

Class method decorators receive the method that is being decorated as the first value, and can optionally return a new method to replace it. If a new method is returned, it will replace the original on the prototype (or on the class itself in the case of static methods). If any other type of value is returned, an error will be thrown.

An example of a method decorator is the `@logged` decorator. This decorator receives the original function, and returns a new function that wraps the original and logs before and after it is called.

```
function logged(value, { kind, name }) {
  if (kind === "method") {
    return function (...args) {
      console.log(`starting ${name} with arguments ${args.join(", ")}`);
      const ret = value.call(this, ...args);
      console.log(`ending ${name}`);
      return ret;
    };
  }
}

class C {
  @logged
  m(arg) {}
}

new C().m(1);
// starting m with arguments 1
// ending m
```

This example roughly "desugars" to the following (i.e., could be transpiled as such):

```
class C {
  m(arg) {}
}

C.prototype.m = logged(C.prototype.m, {
  kind: "method",
  name: "m",
  static: false,
  private: false,
}) ?? C.prototype.m;
```

Class Accessors

```
type ClassGetterDecorator = (value: Function, context: {
  kind: "getter";
  name: string | symbol;
  access: { get(): unknown };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;

type ClassSetterDecorator = (value: Function, context: {
  kind: "setter";
  name: string | symbol;
  access: { set(value: unknown): void };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => Function | void;
```

Accessor decorators receive the original underlying getter/setter function as the first value, and can optionally return a new getter/setter function to replace it. Like method decorators, this new function is placed on the prototype in place of the original (or on the class for static accessors), and if any other type of value is returned, an error will be thrown.

Accessor decorators are applied *separately* to getters and setters. In the following example, `@foo` is applied only to `get x()` - `set x()` is undecorated:

```
class C {
  @foo
  get x() {
    // ...
  }

  set x(val) {
    // ...
  }
}
```

We can extend the `@logged` decorator we defined previously for methods to also handle accessors. The code is essentially the same, we just need to handle additional `kind`s.

```
function logged(value, { kind, name }) {
  if (kind === "method" || kind === "getter" || kind === "setter") {
    return function (...args) {
      console.log(`starting ${name} with arguments ${args.join(", ")}`);
      const ret = value.call(this, ...args);
      console.log(`ending ${name}`);
      return ret;
    };
  }
}

class C {
  @logged
  set x(arg) {}
}

new C().x = 1
// starting x with arguments 1
// ending x
```

This example roughly "desugars" to the following (i.e., could be transpiled as such):

```js
class C {
  set x(arg) {}
}

let { set } = Object.getOwnPropertyDescriptor(C.prototype, "x");
set = logged(set, {
  kind: "setter",
  name: "x",
  static: false,
  private: false,
}) ?? set;

Object.defineProperty(C.prototype, "x", { set });
```

Class Fields

```ts
type ClassFieldDecorator = (value: undefined, context: {
  kind: "field";
  name: string | symbol;
  access: { get(): unknown, set(value: unknown): void };
  static: boolean;
  private: boolean;
  addInitializer(initializer: () => void): void;
}) => (initialValue: unknown) => unknown | void;
```

Unlike methods and accessors, class fields do not have a direct input value when being decorated. Instead, users can optionally return an initializer function which runs when the field is assigned, receiving the initial value of the field and returning a new initial value. If any other type of value besides a function is returned, an error will be thrown.

We can expand our `@logged` decorator to be able to handle class fields as well, logging when the field is assigned and what the value is.

```js
function logged(value, { kind, name }) {
  if (kind === "field") {
    return function (initialValue) {
      console.log(`initializing ${name} with value ${initialValue}`);
      return initialValue;
    };
  }

  // ...
}

class C {
  @logged x = 1;
}

new C();
// initializing x with value 1
```

This example roughly "desugars" to the following (i.e., could be transpiled as such):

```js
let initializeX = logged(undefined, {
  kind: "field",
  name: "x",
  static: false,
  private: false,
}) ?? (initialValue) => initialValue;

class C {
  x = initializeX.call(this, 1);
}
```

The initializer function is called with the instance of the class as `this`, so field decorators can also be used to bootstrap registration relationships. For instance, you could register children on a parent class:

```js
const CHILDREN = new WeakMap();

function registerChild(parent, child) {
  let children = CHILDREN.get(parent);

  if (children === undefined) {
    children = [];
    CHILDREN.set(parent, children);
  }

  children.push(child);
}

function getChildren(parent) {
  return CHILDREN.get(parent);
}

function register() {
  return function(value) {
    registerChild(this, value);

    return value;
  }
}

class Child {}
class OtherChild {}

class Parent {
  @register child1 = new Child();
  @register child2 = new OtherChild();
}

let parent = new Parent();
getChildren(parent); // [Child, OtherChild]
```

Classes

```
type ClassDecorator = (value: Function, context: {
  kind: "class";
  name: string | undefined;
  addInitializer(initializer: () => void): void;
}) => Function | void;
```

Class decorators receive the class that is being decorated as the first parameter, and may optionally return a new callable (a class, function, or Proxy) to replace it. If a non-callable value is returned, then an error is thrown.

We can further extend our `@logged` decorator to log whenever an instance of a class is created:

```
function logged(value, { kind, name }) {
  if (kind === "class") {
    return class extends value {
      constructor(...args) {
        super(...args);
        console.log(`constructing an instance of ${name} with arguments ${args.join(", ")}`);
      }
    }
  }

  // ...
}

@logged
class C {}

new C(1);
// constructing an instance of C with arguments 1
```

This example roughly "desugars" to the following (i.e., could be transpiled as such):

```
class C {}

C = logged(C, {
  kind: "class",
  name: "C",
}) ?? C;

new C(1);
```

If the class being decorated is an anonymous class, then the `name` property of the `context` object is `undefined`.

New Class Elements

Class Auto-Accessors

Class auto-accessors are a new construct, defined by adding the `accessor` keyword in front of a class field:

```
class C {
  accessor x = 1;
}
```

Auto-accessors, unlike regular fields, define a getter and setter on the class prototype. This getter and setter default to getting and setting a value on a private slot. The above roughly desugars to:

```
class C {
  #x = 1;

  get x() {
    return this.#x;
  }

  set x(val) {
    this.#x = val;
  }
}
```

Both static and private auto-accessors can be defined as well:

```
class C {
  static accessor x = 1;
  accessor #y = 2;
}
```

Auto-accessors can be decorated, and auto-accessor decorators have the following signature:

```
type ClassAutoAccessorDecorator = (
  value: {
    get: () => unknown;
    set(value: unknown) => void;
  },
  context: {
    kind: "accessor";
    name: string | symbol;
    access: { get(): unknown, set(value: unknown): void };
    static: boolean;
    private: boolean;
    addInitializer(initializer: () => void): void;
  }
) => {
  get?: () => unknown;
  set?: (value: unknown) => void;
  init?: (initialValue: unknown) => unknown;
} | void;
```

Unlike field decorators, auto-accessor decorators receive a value, which is an object containing the `get` and `set` accessors defined on the prototype of the class (or the class itself in the case of static auto-accessors). The decorator can then wrap these and return a *new* `get` and/or `set`, allowing access to the property to be intercepted by the decorator. This is a capability that is not possible with fields, but is possible with auto-accessors. In addition, auto-accessors can return an `init` function, which can be used to change the initial value of the backing value in the private slot, similar to field decorators. If an object is returned but any of the values are omitted, then the default behavior for the omitted values is to use the original behavior. If any other type of value besides an object containing these properties is returned, an error will be thrown.

Further extending the `@logged` decorator, we can make it handle auto-accessors as well, logging when the auto-accessor is initialized and whenever it is accessed:

```js
function logged(value, { kind, name }) {
  if (kind === "accessor") {
    let { get, set } = value;

    return {
      get() {
        console.log(`getting ${name}`);

        return get.call(this);
      },

      set(val) {
        console.log(`setting ${name} to ${val}`);

        return set.call(this, val);
      },

      init(initialValue) {
        console.log(`initializing ${name} with value ${initialValue}`);
        return initialValue;
      }
    };
  }

  // ...
}

class C {
  @logged accessor x = 1;
}

let c = new C();
// initializing x with value 1
c.x;
// getting x
c.x = 123;
// setting x to 123
```

This example roughly "desugars" to the following:

```js
class C {
  #x = initializeX.call(this, 1);

  get x() {
    return this.#x;
  }

  set x(val) {
    this.#x = val;
  }
}

let { get: oldGet, set: oldSet } =
Object.getOwnPropertyDescriptor(C.prototype, "x");

let {
  get: newGet = oldGet,
  set: newSet = oldSet,
  init: initializeX = (initialValue) => initialValue
} = logged(
  { get: oldGet, set: oldSet },
  {
    kind: "accessor",
    name: "x",
    static: false,
    private: false,
  }
) ?? {};

Object.defineProperty(C.prototype, "x", { get: newGet, set: newSet });
```

Adding initialization logic with `addInitializer`

The `addInitializer` method is available on the context object that is provided to the decorator for every type of value. This method can be called to associate an initializer function with the class or class element, which can be used to run arbitrary code after the value has been defined in order to finish setting it up. The timing of these initializers depends on the type of decorator:

- **Class decorators**: *After* the class has been fully defined, and *after* class static fields have been assigned.
- **Class static elements**
  - **Method and Getter/Setter decorators**: During class definition, *after* static class methods have been assigned, *before any* static class fields are initialized
  - **Field and Accessor decorators**: During class definition, immediately *after* the field or accessor that they were applied to is initialized
- **Class non-static elements**
  - **Method and Getter/Setter decorators**: During class construction, *before any* class fields are initialized
  - **Field and Accessor decorators**: During class construction, immediately *after* the field or accessor that they were applied to is initialized

Example: `@customElement`

We can use `addInitializer` with class decorators in order to create a decorator which registers a web component in the browser.

```
function customElement(name) {
  return (value, { addInitializer }) => {
    addInitializer(function() {
      customElements.define(name, this);
    });
  }
}

@customElement('my-element')
class MyElement extends HTMLElement {
  static get observedAttributes() {
    return ['some', 'attrs'];
  }
}
```

This example roughly "desugars" to the following (i.e., could be transpiled as such):

```
class MyElement {
  static get observedAttributes() {
    return ['some', 'attrs'];
  }
}

let initializersForMyElement = [];

MyElement = customElement('my-element')(MyElement, {
  kind: "class",
  name: "MyElement",
  addInitializer(fn) {
    initializersForMyElement.push(fn);
  },
}) ?? MyElement;

for (let initializer of initializersForMyElement) {
  initializer.call(MyElement);
}
```

Example: `@bound`

We could also use `addInitializer` with method decorators to create a `@bound` decorator, which binds the method to the instance of the class:

```
function bound(value, { name, addInitializer }) {
  addInitializer(function () {
    this[name] = this[name].bind(this);
  });
}

class C {
  message = "hello!";

  @bound
  m() {
    console.log(this.message);
  }
}

let { m } = new C();

m(); // hello!
```

This example roughly "desugars" to the following:

```
class C {
  constructor() {
    for (let initializer of initializersForM) {
      initializer.call(this);
    }

    this.message = "hello!";
  }

  m() {}
}

let initializersForM = []

C.prototype.m = bound(
  C.prototype.m,
  {
    kind: "method",
    name: "m",
    static: false,
    private: false,
    addInitializer(fn) {
      initializersForM.push(fn);
    },
  }
) ?? C.prototype.m;
```

Access and Metadata Sidechanneling

So far we've seen how decorators can be used to replace a value, but we haven't seen how the `access` object for the decorator can be used. Here's an example of dependency injection decorators which use this object via a metadata sidechannel to inject values on an instance.

```
const INJECTIONS = new WeakMap();

function createInjections() {
  const injections = [];

  function injectable(Class) {
    INJECTIONS.set(Class, injections);
  }

  function inject(injectionKey) {
    return function applyInjection(v, context) {
      injections.push({ injectionKey, set: context.access.set });
    };
  }

  return { injectable, inject };
}

class Container {
  registry = new Map();

  register(injectionKey, value) {
    this.registry.set(injectionKey, value);
  }

  lookup(injectionKey) {
    this.registry.get(injectionKey);
  }

  create(Class) {
    let instance = new Class();

    for (const { injectionKey, set } of INJECTIONS.get(Class) || []) {
      set.call(instance, this.lookup(injectionKey));
    }

    return instance;
  }
}

class Store {}

const { injectable, inject } = createInjections();

@injectable
class C {
  @inject('store') store;
}

let container = new Container();
let store = new Store();

container.register('store', store);

let c = container.create(C);

c.store === store; // true
```

Access is generally provided based on whether or not the value is a value meant to be read or written. Fields and auto-accessors can be both read and written to. Accessors can either be read in the case of getters, or written in the case of setters. Methods can only be read.

Possible extensions

Decorators on further constructs are investigated in EXTENSIONS.md.

Standardization plan

- [x] Iterate on open questions within the proposal, presenting them to TC39 and discussing further in the biweekly decorators calls, to bring a conclusion to committee in a future meeting
    - STATUS: Open questions have been resolved, decorators working group has reached general consensus on the design.
- [x] Write spec text
    - STATUS: Complete, available here.
- [x] Implement in experimental transpilers
    - STATUS: An experimental implementation has been created and is available for general use. Work is ongoing to implement in Babel and get more feedback.
        - [x] Independent implementation: https://javascriptdecorators.org/
        - [x] Babel plugin implementation (docs)
- [x] Collect feedback from JavaScript developers testing the transpiler implementation
- [x] Propose for Stage 3.

FAQ

How should I use decorators in transpilers today?

Since decorators have reached stage 3 and are approaching completion, it is now recommended that new projects use the latest transforms for stage 3 decorators. These are available in Babel, TypeScript, and other popular build tools.

Existing projects should begin to develop upgrade plans for their ecosystems. In the majority of cases it should be possible to support both the legacy and stage 3 versions at the same time by matching on the arguments that are passed to the decorator. In a small number of cases this may not be possible due to a difference in the capabilities between the two versions. If you run into such a case, please open an issue on this repo for discussion!

How does this proposal compare to other versions of decorators?

Comparison with Babel "legacy" decorators

Babel legacy-mode decorators are based on the state of the JavaScript decorators proposal as of 2014. In addition to the syntax changes listed above, the calling convention of Babel legacy decorators differs from this proposal:

- Legacy decorators are called with the "target" (the class or prototype under construction), whereas the class under construction is not made available to decorators in this proposal.

- Legacy decorators are called with a full property descriptor, whereas this proposal calls decorators with just "the thing being decorated" and a context object. This means, for example, that it is impossible to change property attributes, and that getters and setters are not "coalesced" but rather decorated separately.

Despite these differences, it should generally be possible to achieve the same sort of functionality with this decorators proposal as with Babel legacy decorators. If you see important missing functionality in this proposal, please file an issue.

Comparison with TypeScript "experimental" decorators

TypeScript experimental decorators are largely similar to Babel legacy decorators, so the comments in that section apply as well. In addition:

- This proposal does not include parameter decorators, but they may be provided by future built-in decorators, see EXTENSIONS.md.
- TypeScript decorators run all instance decorators before all static decorators, whereas the order of evaluation in this proposal is based on the ordering in the program, regardless of whether they are static or instance.

Despite these differences, it should generally be possible to achieve the same sort of functionality with this decorators proposal as with TypeScript experimental decorators. If you see important missing functionality in this proposal, please file an issue.

Comparison with the previous Stage 2 decorators proposal

The previous Stage 2 decorators proposal was more full-featured than this proposal, including:

- The ability of all decorators to add arbitrary 'extra' class elements, rather than just wrapping/changing the element being decorated.
- Ability to declare new private fields, including reusing a private name in multiple classes
- Class decorator access to manipulating all fields and methods within the class
- More flexible handling of the initializer, treating it as a "thunk"

The previous Stage 2 decorators proposal was based on a concept of descriptors which stand in for various class elements. Such descriptors do not exist in this proposal. However, those descriptors gave a bit too much flexibility/dynamism to the class shape in order to be efficiently optimizable.

This decorators proposal deliberately omits these features, in order to keep the meaning of decorators "well-scoped" and intuitive, and to simplify implementations, both in transpilers and native engines.

Comparison with the "static decorators" proposal

Static decorators were an idea to include a set of built-in decorators, and support user-defined decorators derived from them. Static decorators were in a separate namespace, to support static analyzability.

The static decorators proposal suffered from both excessive complexity and insufficient optimizability. This proposal avoids that complexity by returning to the common model of decorators being ordinary functions.

See V8's analysis of decorator optimizability for more information on the lack of optimizability of the static decorators proposal, which this proposal aims to address.

If the previous TC39 decorators proposals didn't work out, why not go back and standardize TS/Babel legacy decorators?

Optimizability: This decorator proposal and legacy decorators are common in decorators being functions. However, the calling convention of this proposal is designed to be more optimizable by engines by making the following changes vs legacy decorators:

- The incomplete class under construction is not exposed to decorators, so it does not need to observably undergo shape changes during class definition evaluation.
- Only the construct being decorated may be changed in its contents; the "shape" of the property descriptor may not change.

Incompatibility with [[Define]] field semantics: Legacy decorators, when applied to field declarations, depend deeply on the semantics that field initializers call setters. TC39 concluded that, instead, field declarations act like Object.defineProperty. This decision makes many patterns with legacy decorators no longer work. Although Babel provides a way to work through this by making the initializer available as a thunk, these semantics have been rejected by implementers as adding runtime cost.

Why prioritize the features of "legacy" decorators, like classes, over other features that decorators could provide?

"Legacy" decorators have grown to huge popularity in the JavaScript ecosystem. That proves that they were onto something, and solve a problem that many people are facing. This proposal takes that knowledge and runs with it, building in native support in the JavaScript language. It does so in a way that leaves open the opportunity to use the same syntax for many more different kinds of extensions in the future, as described in EXTENSIONS.md.

Could we support decorating objects, parameters, blocks, functions, etc?

Yes! Once we have validated this core approach, the authors of this proposal plan to come back and make proposals for more kinds of decorators. In particular, given the popularity of TypeScript parameter decorators, we are considering including parameter decorators in this proposal's initial version. See EXTENSIONS.md.

Will decorators let you access private fields and methods?

Yes, private fields and methods can be decorated just like ordinary fields and methods. The only difference is that the `name` key on the context object is only a description of the element, not something we can be used to access it. Instead, an `access` object with `get`/`set` functions is provided. See the example under the heading, "Access".

How should this new proposal be used in transpilers, when it's implemented?

This decorators proposal would require a separate transpiler implementation from the previous legacy/experimental decorator semantics. The semantics could be switched into with a build-time option (e.g., a command-line flag or entry in a configuration file). Note that this proposal is expected to continue to undergo significant changes prior to Stage 3, and it should not be counted on for stability.

Modules exporting decorators are able to easily check whether they are being invoked in the legacy/experimental way or in the way described in this proposal, by checking whether their second argument is an object (in this proposal, always yes; previously, always no). So it should be possible to maintain decorator libraries which work with both approaches.

What makes this decorators proposal more statically analyzable than previous proposals? Is this proposal still statically analyzable even though it is based on runtime values?

In this decorators proposal, each decorator position has a consistent effect on the shape of the code generated after desugaring. No calls to `Object.defineProperty` with dynamic values for property attributes are made by the system, and it is also impractical to make these sorts of calls from user-defined decorators as the "target" is not provided to decorators; only the actual contents of the functions.

How does static analyzability help transpilers and other tooling?

Statically analyzable decorators help tooling to generate faster and smaller JavaScript from build tools, enabling the decorators to be transpiled away, without causing extra data structures to be created and manipulated at runtime. It will be easier for tools to understand what's going on, which could help in tree shaking, type systems, etc.

An attempt by LinkedIn to use the previous Stage 2 decorators proposal found that it led to a significant performance overhead. Members of the Polymer and TypeScript team also noticed a significant increase in generated code size with these decorators.

By contrast, this decorator proposal should be compiled out into simply making function calls in particular places, and replacing one class element with another class element. We're working on proving out this benefit by implementing the proposal in Babel, so an informed comparison can be made before proposing for Stage 3.

Another case of static analyzability being useful for tooling was named exports from ES modules. The fixed nature of named imports and exports helps tree shaking, importing and exporting of types, and here, as the basis for the predictable nature of composed decorators. Even though the ecosystem remains in transition from exporting entirely dynamic objects, ES modules have taken root in tooling and found to be useful because, not despite, their more static nature.

How does static analyzability help native JS engines?

Although a JIT can optimize away just about anything, it can only do so after a program "warms up". That is, when a typical JavaScript engine starts up, it's not using the JIT--instead, it compiles the JavaScript to bytecode and executes that directly. Later, if code is run lots of times, the JIT will kick in and optimize the program.

Studies of the execution traces of popular web applications show that a large proportion of the time starting up the page is often in parsing and execution through bytecode, typically with a smaller percentage running JIT-optimized code. This means that, if we want the web to be fast, we can't rely on fancy JIT optimizations.

Decorators, especially the previous Stage 2 proposal, added various sources of overhead, both for executing the class definition and for using the class, that would make startup slower if they weren't optimized out by a JIT. By contrast, composed decorators always boil down in a fixed way to built-in decorators, which can be handled directly by bytecode generation.

What happened to coalescing getter/setter pairs?

This decorators proposal is based on a common model where each decorator affects just one syntactic element--either a field, or a method, or a getter, or setter, or a class. It is immediately visible what is being decorated.

The previous "Stage 2" decorators proposal had a step of "coalescing" getter/setter pairs, which ended up being somewhat similar to how the legacy decorators operated on property descriptors. However, this coalescing was very complicated, both in the specification and implementations, due to the dynamism of computed property names for accessors. Coalescing was a big source of overhead (e.g., in terms of code size) in polyfill implementations of "Stage 2" decorators.

It is unclear which use cases benefit from getter/setter coalescing. Removing getter/setter coalescing has been a big simplification of the specification, and we expect it to simplify implementations as well.

If you have further thoughts here, please participate in the discussion on the issue tracker: #256.

Why is decorators taking so long?

We are truly sorry about the delay here. We understand that this causes real problems in the JavaScript ecosystem, and are working towards a solution as fast as we can.

It took us a long time for everyone to get on the same page about the requirements spanning frameworks, tooling and native implementations. Only after pushing in various concrete directions did we get a full understanding of the requirements which this proposal aims to meet.

We are working to develop better communication within TC39 and with the broader JavaScript community so that this sort of problem can be corrected sooner in the future.