# Decorators Design Doc

**Attention - this doc is public and shared with the world!**

**Contact:** lpardosixtos@microsoft.com
**Contributors:** Luis Pardo
**Last updated:** Oct 30, 2024
**Status:** Inception | **Draft** | Accepted | Done
**Tracking bug:** Implement the decorators proposal [42202709] - Chromium

## LGTMs needed

| Name | Write (not) LGTM in this row |
|---|---|
| syg | |
| verwaest | |
| You? | |
| ... | |

## Prototype CLs:

1. [decorators] Add decorators context and access object. (5818208) · Gerrit Code Review
2. [decorators] Support class decorator syntax in the parser. (5763660) · Gerrit Code Review
3. [decorators] Add class decorators logic (5777465) · Gerrit Code Review
4. [decorators] Add class member parsing support (5827469) · Gerrit Code Review
5. [decorators] Add class element decorator logic (5837979) · Gerrit Code Review
6. [decorators] Rewrite bytecode generator class members initialization (5910395) · Gerrit Code Review
7. [decorators] Add class method/accessor decorator logic (5922269) · Gerrit Code Review
8. [decorators] Add auto-accessor decorators logic (5924163) · Gerrit Code Review

**Mega change for reference**:
[decorators] DO-NOT-MERGE. Decorators end-to-end mega change. (5936390) · Gerrit Code Review

## Background

The decorators proposal has reached stage 3. This doc describes the following changes to v8 required for the implementation:

1. Support for the decorator application syntax (@).
2. Support decorators in generated bytecode.

The support for the `accessor` keyword that is also included in the decorators proposal is explained in Design doc. Accessor keyword support.

## Design

### Decorator List

The proposal is not limited to a single decorator per decorated object, so we store the decorators in a decorator list that will be iterated later.

#### Parsing

Implementation-wise, the decorators list is a `ZonePtrList` of expressions, each of them representing an individual decorator. The decorator list is the result of parsing expressions of the form

```
DecoratorList :
  DecoratorList Decorator
```

Where the valid decorator expressions are

```
Decorator :
  @ DecoratorMemberExpression
  @ DecoratorParenthesizedExpression
  @ DecoratorCallExpression
DecoratorMemberExpression :
  IdentifierReference
  DecoratorMemberExpression.IdentifierName
  DecoratorMemberExpression.PrivateIdentifier
DecoratorParenthesizedExpression :
  ( Expression )
DecoratorCallExpression :
  DecoratorMemberExpression Arguments
```

Note that `DecoratorMemberExpression` cannot be composed by expressions of types `DecoratorParenthesizedExpression` or `DecoratorCallExpression`, hence the following expressions are invalid.

```
@foo(x).y class C{}
@foo(x).#y class C{}
@(foo).y class C{}
```

Similarly, `DecoratorCallExpression` must be formed by `DecoratorMemberExpression`, hence the following expressions are invalid:

```
@foo(x)(y) class C{}
@(foo)(y) class C{}
```

## Decorator Context Object

The decorator context object is a synthetic object containing information about the decorator property and which is passed as an argument to the decorator. It has the following shape

```
{
  kind: ‘class’ | ‘method’ | ‘getter’ | ‘setter’ | ‘field’ | ‘accessor’,
  access: <DecoratorAccessObject>, // present if kind !== ‘class’
  private: true | false, // present if kind !== ‘class’
  static: true | false, // present if kind !== ‘class’
  name: <String>,
  addInitializer: <addInitializerFunction>
}
```

For the implementation:

1. we initialize all the strings above by adding them to the `NOT_IMPORTANT_INTERNALIZED_STRING_LIST_GENERATOR` macro in heap_symbols.h.
2. We must create a new instance for each decorator call, otherwise we would introduce a non-spec compliant communication channel between the decorators.
3. We can create a new object and add the properties every time a decorator is called or cache 2 maps in the native context, one for the class kind and one for class properties.

### addInitializer function

The `addInitializer` function allows the user to add extra initializers that will be executed later, depending on the property type.

- **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

We define the `addInitializer` function as a torque builtin. The use of this function must follow the next restrictions:

1. According to the spec, a new decorator context object is created for each decorator call, and a new `addInitializer` function instance is created for each decorator context object. We need to follow this behavior because this function is exposed to JS and reusing instances would open a non-spec compliant communication channel to the users.
2. Each `addInitializer` function is created with a synthetic context with one extra context slot storing an `ArrayList`, which will be used to store the initializers.
3. According to the spec, each `addInitializer` function must close around a `decorationState` value that is initialized with `false` and is set to `true` after the decoration is executed. We implement this by setting the context of the `addInitializer` function to the native context after the decorator is executed.

`addInitializer` torque implementation:

```
transitioning javascript builtin AddInitializer(
    js-implicit context: Context, receiver: JSAny)(initializer: JSAny):
JSAny {
  if (Is<NativeContext>(context)) {
    ThrowTypeError(MessageTemplate::kInvalidDecoratorAddInitializer);
  }
  if (!Is<Callable>(initializer)) {
    ThrowCalledNonCallable(initializer);
  }
  let initializerList = *ContextSlot(
      context,
      AddInitializerContextSlots::kInitializerListContextSlot);
  initializerList =
      ArrayListAdd(initializersList, initializer);
  *ContextSlot(
      context, AddInitializerContextSlots::kInitializerListContextSlot) =
          initializerList;
  return Undefined;
}
```

## Decorator Access Object

Per spec, the decorator access object is exposed to js as part of the decorator context for decorators of any kind other than “class”. The decorator access object’s functions close around the `name` of the decorator property and has the following shape:

```
{
  get: (obj)=>obj.name // Not present for setters
  set: (obj, value)=>{obj.name = value} // Not present for getters/methods
  has: (obj)=>{name in obj}
}
```

On the implementation, we use builtins for the `get`, `set` and `has` functions with the name in a context slot and initialize decorator access objects from a map cached in the native context.

### Private methods and accessors

Private methods and accessors are not stored as properties keyed by the private name.

The access functions for private instance methods will have both the object and brand in the context and perform a brand check before returning the object, this is safe as private properties cannot be deleted.

The access functions for private static methods will store the constructor itself instead of the brand, and perform an equality check.

## Class decorators

Class decorators can be applied class statement, expressions, as well as exported and default exported classes, example:

```
@foo class C{}
let C = @foo class {}
@foo export class C{}
export @foo class C{}
@foo export default class C{}
@foo export default class {}
export default @foo class C {}
export default @foo class {}
```

**Note:** The spec explicitly prohibits the use of decorators both before and after `export` or `export default`.

## Parsing

Implementation-wise, we modify the behavior in the parser when an `@` token appears in a place where there is logic to handle either the `export` token or the `class` token. Note that we don't know if the decorator list will be followed by a valid expression until we've finished parsing it. Once we've parsed the decorator list and verified that it is a valid use, we pass it as an argument to the functions parsing export or class statements/expressions.

The decorator list is eventually stored in a new field in `ClassLiteral`.

## Bytecode generator

We extend the `ClassBoilerplate` object with a new field containing an ArrayList of extra class initializers,

```
extern class ClassBoilerplate extends Struct {
  arguments_count: Smi;
  static_properties_template: Object;
  static_elements_template: Object;
  static_computed_properties: FixedArray;
  instance_properties_template: Object;
  instance_elements_template: Object;
  instance_computed_properties: FixedArray;
  extra_class_initializers: ArrayList;
}
```

The new field is initialized to an empty `ArrayList` in `ClassBoilerplate::New`.

Class decorators are evaluated outside of the class scope. So we visit them in `BytecodeGenerator::VisitClassLiteral` right before entering the class scope, we store the visited values in a register list that is passed to `BytecodeGenerator::BuildClassLiteral`.

We implement the [ApplyDecoratosToClassDefinition](#) operation as a Runtime function whose arguments are:

1. The class constructor.
2. The class boilerplate.
3. The decorator functions.

Runtime_ApplyDecoratosToClassDefinition will do the following steps:

1. Load `extra_class_initializers` from `class_boilerplate`.
2. For each `decorator`:
    a. Initialize a new function object `add_initializer_function` using the `addInitializer` builtin and synthetic context `add_initializer_context` containing `extra_class_initializers` in its `kInitializerListContextSlot` slot.
    b. Initialize the `decorator_context` object with the initializer function.
    c. Call the decorator with `decorator` as the receiver and `class_constructor` and `decorator_context` as arguments. Store the result in `class_constructor`.
    d. Set `extra_class_initializers` to `add_initializer_context.get(kInitializerListContextSlot)`.
    e. Set `add_initializer_function` context to the current native context.
3. Set `class_boilerplate.extra_class_initializers` to `extra_class_initializers`.
4. Return `class_constructor`.

Per spec, the class initializers are executed as the last step before exiting the class's running execution context. This is implemented as another runtime with two arguments: `class_constructor` and `class_boilerplate`. It loads `extra_class_initializers` and executes each element with no arguments and `class_constructor` as the receiver.

## Class field decorators (non-auto-accessors).

Decorators can be applied to class fields with the following syntax:

```
class C {
  @foo  x = 1;
  @bar static y = 2;
}
```

Class field decorators don't receive a `value` argument and their return value is not a direct replacement of the property. Instead, the return value is added to a per-property *initializers* list, different from the "*extra initializers"* list used by the `addInitializers` list.

## Parsing

We parse the decorator list by adding a peek check for the `@` token at the top of `ParserBase::ParseClassPropertyDefinition` before we start parsing the class property; after the decorator list is parsed, we parse the class property as usual. If the class property is successfully parsed then we wrap the decorator list into a `DecoratorInfo` object with the following definition:

```
class DecoratorInfo : public ZoneObject {
  ZonePtrList<Expression> decorators_;
  Variable* initializer_lists;
}
```

and store it in this property's `ClassLiteralProperty` instance.

The `initializer_lists` variable will reserve a context slot in the class scope that we will use to share the initializer information between the class literal evaluation (when the initializers are collected) and the class literal property evaluation (when the initializers are applied).

## Bytecode generator

Class field decorators are applied after all other property decorators and before initializing private methods, starting with the static class fields. When executed, decorators receive the `undefined` as a decorator context object as arguments.

Currently, there is no code traversing through the instance or static elements at class evaluation time (`BytecodeGenerator::BuildClassLiteral`) when the field decorators must be executed. However, we can access the list through the `static_initializer_` and `nstance_members_initializer_function_` function literals in `ClassLiteral`. We iterate the properties and execute individual calls to a new `Runtime_ApplyDecoratorsToFieldDefinition` runtime function. `Runtime_ApplyDecoratorsToFieldDefinition` returns a struct object with references to both: the initializers and the extra initializers; this object is stored in the `initializers_lists` variables of the respective property.

**Note:** If we don't want to iterate through all the elements every time, we could add a new list with only the decorated properties.

Per spec, the *"extra_initializers"* are loaded from the `initializer_lists` variable and applied when the regular initializers are executed in `BytecodeGenerator::BuildClassProperty` with the initial value as argument. The (non-extra) "*initializers"* are called without arguments after the property has been assigned to the receiver.

## Class methods decorators (methods, accessors and auto-accessors).

Decorators can be applied to class methods with the following syntax:

```
class C {
  @foo f() {}
  @foo get x() {}
  @foo set x() {}
  @foo accessor x;
  @bar static f() {}
  @bar static get x() {}
  @bar static set x() {}
  @bar static accessor x;
}
```

Non-auto-accessor decorators return a callable object that overwrites the value of the decorated object in the prototype.

Auto-accessor are a combination of field and method decorators and return an object of the following shape:

```
{
  get: Callable;
  set: Callable;
  init: init
}
```

where the `get` and `set` entries can overwrite the auto-generated getter and setter and `init` must be added to this property's *"initializer list"*, which is different from the "*extra initializer list"* used by the `addInitializer` function.

To avoid introducing more runtime calls, we reserve 2 extra space in the ClassBoilerplate to keep track of the initializers and extra initializers introduces by methods.

```
extern class ClassBoilerplate extends Struct {
  arguments_count: Smi;
  static_properties_template: Object;
  static_elements_template: Object;
  static_computed_properties: FixedArray;
  instance_properties_template: Object;
  instance_elements_template: Object;
  instance_computed_properties: FixedArray;
  extra_class_initializers: ArrayList;
  extra_instance_initializers: ArrayList;
  extra_static_initializers: ArrayList;
}
```

## Parsing

For auto-accessors we follow the same parsing logic as with fields. For non-auto-accessors,  we store a pointer to the decorator list directly in the `ClassLiteralProperty` instead of creating a `DecoratoInfo` object because, in this case, the decorators and extra initializers are executed at class definition time.

## Bytecode generator

According to the spec, class elements are iterated in the order they appeared in the body but as of today, private and public methods are stored in different lists by the parser, and iterated at different times by the bytecode generator. So applying decorators at the time each method is currently declared wouldn't be spec compliant. Additionally, decorators are called before each method is set in the respective homeObject, so decorators applied to methods overridden by a second declaration must also be executed in order, e.g.

```
function g() {console.log("I was executed first")}
function h() {console.log(“I was executed second”)}

class C {
  @g foo() {return 1;}
  @h foo() {return 2;}
  foo() {return 42}
}

// console output:
//     I was executed first
//     I was executed second

let c = new C();
// c.foo() === 42
```

Currently, methods are evaluated in the following order:

1. Non-accessors private methods.
2. Public Methods.
3. Accessor private methods. (Generating bytecode).

**Note**: We only need to execute the decorators in order, so it would be spec compliant to iterate the methods in any order, compute their function values, and then iterate them again to apply decorators in the right order; however, we'd need add complexity to the bytecode generator and reserve more registers to keep track of repeated methods. Instead, the following algorithm describes a way to do it in a single pass.

We want to merge the 3 steps above into a single one so we can execute the decorators in declaration order. We start by merging 1 and 3.

**Prerequisite:** We need private field names to already be assigned at the time we initialize the computed names, we introduce a new `private_fields` list in the `ClassLiteral` class so that private field members are stored separately at parsing time and can be iterated before this algorithm is executed.

Step 1 mentioned above iterates the list of getters and setters evaluating the methods but performing other steps for accessors. The current algorithm initializes an `ordered_private_accessors` hash table with literal objects as keys and pairs `<getter: ClassLiteralProperty, setter: ClassLiteralProperty>` and the iterates the private methods list assigning the value of the method to following these steps:

A. If the accessor's name is not in the table, add a new entry in the table with only one of the pair values populated.
B. If the accessor's name is already in the table, fill the existing empty spot in the pair with this property.
C. For auto-accessors, create a new entry with both elements already populated.

The `ordered_private_accessors` list is iterated in step 3, the values in each pair are visited and used to create an `AccessorPair` (via a Runtime function), the result is stored in the corresponding `private_name_var` of the getter if defined, and in the `private_name_var` of the setter otherwise,

We change the algorithm by replacing the `ordered_private_accessors` table with a table whose values are single "ClassLiteralProperties". For each `private_accessor` in the private accessors:

1. Let `name` be `private_property.name`. If `name` is not in `ordered_private_accessors`:
    a. If `private_property.kind` is `GETTER` or `SETTER`, visit the function literal and apply the decorators with a runtime function. Assign the result to `` `private_property.private_name_var` `` and perform `ordered_private_accessors.add((name, private_property))`.
    b. Else, if `private_property.kind` is `AUTOACCESSOR` . Visit the auto-generated getter and setter function literals and apply the decorators with a runtime function. Use these values to create an `AccessorPair` and store the result in `private_property.private_name_var`.
2. Else.
    a. Let `existing_property` be `ordered_private_accessors.get(private_property.name)`.
    b. Visit `existing_property` and apply the decorators with a runtime function, use the result together with the value in `existing_property.private_name_var` to create a new `AccessorPair` with a Runtime function call.
    c. Store the result in the `private_name_var` field of this pair's getter.

With this we guarantee that private members are evaluated in the order they were declared. To complete the algorithm, we iterate both the private members and public members lists at the same time. Always prioritizing the property with the lowest beginning position.

### Runtime_ApplyDecoratorsToAutoAccessorDefinition

We add a new `Runtime_ApplyDecoratorsToAutoAccessorDefinition` that is called from the bytecode and wraps the decorators application logic, such as:

1. Creating a decorator context object.
2. Creating an `addInitializer` function instance.
3. Calling the decorator function.
4. Verifying that the return value is a valid object:
    a. If it has a `get` field, assert it is callable or undefined. If it's callable, overwrite the existing getter.

   b. If it has a `set` field, assert it is callable or undefined. If it's callable, overwrite the existing setter.
   c. If it has an `init` field, assert it is callable and push the value to this property's initializer list.

Auto-accessor properties initializers list behave as regular properties initializers list.

The return value of `Runtime_ApplyDecoratorsToAutoAccessorDefinition` will be a JSObject with pointers to the updated accessor object and the initializer lists that we can query in the bytecode generator to assign to the respective variables.

### Runtime_ApplyDecoratorsToMethodOrAccessorDefinition

We add a new `Runtime_ApplyDecoratorsToClassMethod` that is called from the bytecode and wraps the decorators application logic, such as:

1. Creating a decorator context object.
2. Creating an `addInitializer` function instance.
3. Calling the decorator function.
4. Verifying that the return value is a callable object.

Non-static extra initializers added through the `addInitializer` function will be called at class element initialization time with the class instance object as receiver. We can achieve this by adding a new `Variable` into the `InitializeClassMembersStatement`, the extra initializers list is initialized with a runtime call in `BytecodeGenerator::BuildClassLiteral` and is stored in the variable once all the decorators have been executed. Note that it is possible that `InitializeClassMembersStatement` doesn't exist because the class lacks initializers, in this case we call runtime to instantiate a new JS function that we'll run the extra initializers and store it in the constructor's class field initializer property slot.

Static extra class initializers are executed on class definition, so we initialize an `ArrayList` with a runtime call in `BytecodeGenerator::BuildClassLiteral`, fill it in the `Runtime_ApplyDecoratorsToAutoClassMethod` and iterate through it before executing the class initializers.

## Implementation summary

New string constants:

- **src/init/heap-symbols.h** (*String definitions).*
    - NOT_IMPORTANT_INTERNALIZED_STRING_LIST_GENERATOR
- **src/roots/static-roots.h**

Cached DecoratorAccess and DecoratorContext objects:

- **src/objects/js-objects.h** (*object declaration*).
- **src/objects/js-objects-inl.h** (*object definition*).
- **src/objects/objects-inl.h** (*object casting*).
- **src/init/bootstrapper.cc** (*object map initialization*).
- **src/objects/contexts.h** (*object map caching*).
- **src/heap/factory.h(cc)** (*object construction).*

DecoratorAccess get/set/has and DecoratorContext addInitializer.

- (New) **src/builtins/builtins-decorators.h**. (*Context slots C++ enum).*
- (New) **src/builtins/decorators.tq (*function definitions***, *Context slots torque enum).*
- **src/roots/roots.h** (*SharedFunctionInfo caching)*.
- **src/heap/setup-heap-internal.cc** (*SharedFunctionInfo initialization).*
- **src/codegen/code-stub-assembler.cc** (*Private get support).*
- **src/codegen/code-stub-assembler.h** (*Private get support).*
- **src/builtins/builtins-internal-gen.cc** (*Private get support).*
- **src/objects/fixed-array.tq** (*ArrayList torque declaration).*
- **src/builtins/base.tq** (*PrivateGet torque support, error messages).*
- **src/common/message-template.h** (*New error messages).*
- **test/unittests/test-utils.h** (*Decorator flag unittest support).*

Parsing:

- **src/parsing/scanner-inl.h** (*@ token support).*
- **src/parsing/token.h** *(@ token support).*
- **src/parsing/parser-base.h**
  - ClassInfo (*Add DecoratorList class member).*
  - New Parsing Methods.
    - ParseDecoratorList
    - ParseTopLevelDecoratorList
    - ParseClassDecoratorList
    - ParseDecoratorExpression
  - ParsePrimaryExpression
    - Add case for parsing @ token for decorated class expressions.
  - ParseClassDeclaration, ParseClassExpression, ParseClassLiteral.
    - Handle passed decorator list and @ token.
  - ParseClassPropertyDefinition
    - Handle decorators in class members.
- **src/parsing/preparser.h** (*Preparser decorator list supports currently no special handling).*
  - Synthetic variable support
- **src/parsing/parser.h(cc)**
  - ParseModuleItem, ParseExportDefault, ParseExportDeclaration (*Exported decorated class support).*
  - (New) NewDecoratorInfoLogic and DecoratorInfo synthetic variable support.
  - Add method/accessor initializer variable to the class instance initializer.

- **src/ast/ast.h(cc)**
    - ClassLiteral (*Add decorator list class field).*
    - NewDecoratorList factory method.
    - (New) DecoratorInfo class.
    - ClassLiteralProperty (*Decorator info/decorator list class field support).*

Bytecode generator:

- **src/interpreter/bytecode-generator.cc.**
    - VisitClassLiteral
        - Visit decorators.
    - BuildClassLiteral
        - Call runtime to apply decorators to class definition.
        - Call runtime to apply extra initializers (Could be moved to a torque builtin call).
        - Iterate fields and apply decorators. (One runtime call for each class member)
        - Change method/accessor evaluation order, call runtime to apply decorators (One call for each class member).
    - BuildClassProperty
        - Run class field and auto-accessors initializers and extra-initializers.
- **src/objects/literal-objects.h(cc/tq)** (*Initialize initializer lists in the class boilerplate).*
- **src/runtime/runtime-classes.cc** (Could be moved to a new runtime-decorators file).
    - Added Runtime functions:
        - Runtime_ApplyDecoratorsToClassDefinition
        - Runtime_ApplyDecoratorsToFieldDefinition
        - Runtime_ApplyDecoratorsToPublicMethodOrAccessorDefinition
        - Runtime_ApplyDecoratorsToPrivateMethodOrAccessorDefinition
        - Runtime_ApplyDecoratorsToPrivateAutoAccessorDefinition
        - Runtime_ApplyDecoratorsToPublicAutoAccessorDefinition
        - Runtime_RunExtraInitializers
        - Runtime_RunDecoratorFieldOrAutoAccessorInitializers
        - Runtime_RunDecoratorFieldOrAutoAccessorExtraInitializers
        - Runtime_RunStaticMethodExtraInitializers
        - Runtime_MakeRunExtraInitializersFunction

## References

- Github issue: GitHub - tc39/proposal-decorators: Decorators for ES6 classes
- Draft spec: ECMAScript Language Specification Comparator (arai-a.github.io)