Introspection API

*Status: (Draft, Final)*

*Authors: misko@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

1. Define introspection API to the Type and Metadata annotation at runtime.
2. Provide a reasonable fallback strategy for ES5, ES6, TypeScript v1, CoffeeScript users.

# Background

Frameworks need to be able to access annotation data at runtime. This allows the user of

the framework a more declarative approach to building applications. Declarative approach is a preferred approach in many scenarios and we believe it provides a productivity boost to the developers of the framework.

# Prior Art

Type and data annotations and their corresponding API can be seen in Java, Dart, and

TypeScript.

Many of these existing systems have a formal Reflection/Mirror API for retrieving and interacting with the objects. We think that JavaScript is different because basic introspection is already provided in the bracket ([]) and for-in-loop operators.

|  |
| --- |
| var obj = {a: 1, b: 2};  // Reflective access to data  for(var name in obj) {  var reflectiveValue = obj[name]  } |

This is in stark contrast to systems such as Java and Dart which rely on intermediate objects such as:

|  |
| --- |
| var obj = {a: 1, b: 2};  // need to get a hold of intermediate object  var instanceMirror = mirror(obj);  for(var fieldMirror in instanceMirror.fields) {  var reflectiveValue = fieldMirror.reflect;  } |

The main difference is that in JavaScript today there is no need for intermediary objects which wrap/represent the current level of reflection which is provided in the language today.

The reason why systems such as Dart and Java require intermediary objects is because these systems have separate namespace for variables/data and separate for Types. The code below has different semantics in JavaScript, vs Dart/Java;

|  |
| --- |
| class MyType {}  // passing a reference to a type  var type = MyType;  // JavaScript: perfectly OK  // Dart: MyType is a reference to type, where as  // 'type' is an instance of Type which represents the MyType  // Java: would have to be type = MyType.class;  // same semantics as Dart.  // Using the reference  var instance = new type();  // JavaScript: works as expected  // Dart/Java: Does not work since `new` can only be used with types  // not objects which represent the type. |

Languages which have separate namespace for Types and variables (such as Dart/Java) require introspection API (objects representing the types). Languages which treat Types as data (such as JavaScript) don't strictly require such API.

Just because reflective API is not strictly needed, it may still be a good idea to have such API. Here we would like to collect some pros/cons

|  |  |
| --- | --- |
| **PRO** | **CON** |
| Provides a layer of indirection. Such layer may have uses. | Inconsistent with JS prior art of reflective on fields and reflectively access these fields. (See bracket ([]) and for-in discussion above) |
|  | Provides a layer of indirection / Complicates the code with intermediary objects. It is unclear what value do these intermediary objects provide |

***It is our opinion and therefore an explicit goal of this proposal that the introspection API which we provide should be consistent with JavaScript prior art, which does not use formal intermediary objects for introspection.***

# Detailed Design

## Prior Art of ES5

In JavaScript Types are represented as constructor functions. The constructor functions can be passed around as data, and therefore can have any local name as shown in the this example.

|  |
| --- |
| class MyType {}  var type = MyType; // type and MyType are indistinguishable  var obj = new type();  expect(obj instanceof type).toEqual(true);  expect(obj instanceof MyType).toEqual(true); |

In addition JavaScript already has limited introspection API as shown here:

|  |
| --- |
| class MyType {  methodA() {}  }  var obj = new MyType();  expect(obj.constructor).toBe(MyType);  for (var name in obj.constructor.prototype) {  var method = obj.constructor.prototype[name];  var methodName = method.name;  var methodParameterSize = method.length;  } |

Using the above API we can already get a hold of the constructor function from an instance, as well as getting hold of all of the instance methods of a type. Finally from the method we can retrieve its name and its length (number of declared parameters). The important point here is that Function already defines limited set of introspection APIs. We can formally define it here using the TypeScript interface as:

|  |
| --- |
| interface Prototype {  constructor: Function;  }  interface Function {  name: string;  length: int;  prototype: Prototype;  call(self, arguments:any[]);  apply(self, ...arguments:any[]);  } |

## Introspection API proposal

There are two kinds of annotations which we need to keep and make available at runtime. These are metadata and type annotations. All existing introspection APIs are on the Function, and the proposal will follow this precedent, by extending Function with additional properties. It is important that the API is such which can easily be expressed in ES5 without the need for any special library.

### Type Annotations

A Function can optionally implement the following interface which would allow for type introspection.

|  |
| --- |
| interface Parameters {  parameters: Function[];  }  interface Returns {  returns: Function;  } |

Not all Functions need to implement such interface. An example of retrieving the data at runtime.

|  |
| --- |
| **// TypeScript**  class Lock {}  class Door {  constructor(lock:Lock, string:string) {}  isLocked(): boolean;  } |
| **// Equivalent ES5**  Door.parameters = [Lock, string];  Door.prototype.isLocked.parameters = [];  Door.prototype.isLocked.returns = boolean; |
| **// Accessing at runtime**  if (Door.parameters) {  expect(Door.parameters).toEqual([Lock, string])  }  if (Door.returns) {  // constructors don't usually return anything  // (unless called with new operator)  expect(Door.returns).toEqual(void)  }  if (Door.prototype.isLocked) {  expect(Door.prototype.isLocked).toEqual(boolean);  } |

The parameters property is an array of constructor functions. Those functions represent the types of parameters as declared.

**Todo**: handling var args; handling of mixed arguments types

### Metadata Annotations

A Function can optionally implement the following interfaces which would allow for metadata introspections.

|  |
| --- |
| interface Annotate {  annotate: any[];  }  interface ParameterAnnotate {  parameterAnnotate: any[][];  } |

Not all Functions need to implement such interface. An example of retrieving the data at runtime.

|  |
| --- |
| **// TypeScript**  var singleton = 'singleton';  class Inject {}  @Inject()  class Door {  constructor(@singleton lock, string) {}  } |
| **// Equivalent ES5**  Door.annotate = [new Inject()];  Door.parameterAnnotate = [[singleton], []]; |
| **// Accessing at runtime**  if (Door.parameters) {  expect(Door.parameters).toEqual([Lock, string])  }  if (Door.returns) {  // constructors don't usually return anything  // (unless called with new operator)  expect(Door.returns).toEqual(void)  }  if (Door.prototype.isLocked) {  expect(Door.prototype.isLocked).toEqual(boolean);  } |

### Field annotations

## Backwards Compatibility

It is important that the type annotation can easily be define in a backward compatible way. This implies that the developers writing in ES5 should easily be able to patch this API on top of existing Functions without the need for a third party library. We may provide a library for creating class like constructs along with metadata annotations for easier development.

|  |
| --- |
| **// Equivalent ES5**  var Point = function(x, y) {  this.x = x;  this.y = y;  };  Point.annotate = [new Inject()]; |
| **// ES5 with helper library**  var Point = CLASS(function() {  ANNOTATE(Inject);  CONSTRUCTOR(function(x, y) {  this.x = x;  this.y = y;  });  }); |

## Nominal vs Structural Types at Runtime

Normally when checking for types one can use instanceof operator. This however can only verify nominal types. For structural types an alternative mechanism needs to be present.

|  |
| --- |
| var ArrayLike = function () { /\* constructor function \*/};  ArrayLike.isAssignableFrom = function(obj) {  return typeof obj.length == 'number';  };  var list:ArrayLike = {length: 0};  // implicitly calls ArrayLike.isAssignableFrom(list); |

## Primitive Types

A set of global types need to be defined which represent the primitive types.

|  |
| --- |
| var string = { isAssignableFrom: function(o) {  return typeof o == 'string';  } };  var number = { isAssignableFrom: function(o) {  return typeof o == 'number';  } };  var int = { isAssignableFrom: function(o) {  return typeof o == 'number' && o % 2 == 0;  } }; |

## Interfaces as Runtime Types

In its simplest form isAssignableFrom can be used to verify that an instance satisfies a given interface.

# Caveats

You may need to describe what you did not do or why simpler approaches don't work.

Mention other things to watch out for (if any).

# Security Considerations

How you’ll be secure

# Performance Considerations / Test Strategy

How you’ll be fast.

# Work Breakdown

Description of development phases and approximate time estimates.