ES6 +A: Angular v2.0 extensions to ES6 Traceur

# Goal

For Angular v2.0 we are rethinking everything. One area of pain is writing JS which is succinct and minifiable. For this reason we are looking to ES6 and Traceur as a transpiler story to get the features from the language which will allow our users to build better apps.

* The use of traceur should be optional for us as well as for our users. This implies that anything which traceur enables should be a better syntax but can be equally written in pure ES5.
* In Angular v2.0 we would like to use classes a lot more than in Angular v1.x. ES6 has a better class syntax than ES5.
* Angular's goal is to be declarative everywhere it makes sense. For this reason we need to enable annotation syntax in traceur. This syntax needs to translate cleanly to human-readable and human-writable ES5.
* Angular deals with server data in form of a JSON. We would like to provide a way to assert structural types on the JSON as well as on the Angular API. For this reason we would like to enable runtime contracts based on types in traceur. These asserts would only be enabled in development mode.
* The output of the traceur should contain closure annotations so that it can be compiled with existing closure code. This will allow a projects to start using ES6 piecemeal.

[Goal](#heading=h.4raumpo138k2)

[Annotations](#heading=h.ambwele793xv)

[Glossary:](#heading=h.9f8i2meoc06s)

[Angular Use Case](#heading=h.af86gjw3q9it)

[Example:](#heading=h.j8hx66ygzudq)

[Goals](#heading=h.e7vgoqbd2hpw)

[Annotations](#heading=h.mp4ukbw5r5y9)

[Decorators](#heading=h.8qh9bcxqc3kw)

[Decorating Function Parameters](#heading=h.bnxi8dzd8688)

[Decorators can Interfere](#heading=h.slh6pvdl65vu)

[Decorators encourage global variables.](#heading=h.4jlqzj2l7g96)

[Example](#heading=h.2djmb27119d5)

[Decorator Summary](#heading=h.6ay0dyyupyo3)

[Contracts / Dynamic Types](#heading=h.8bcf31k1ne9q)

[Static Types](#heading=h.qjzshbj4hjwr)

[Contracts](#heading=h.82n8yy8ufvuh)

[Contracts and Static Types](#heading=h.hlxwhp7qae44)

[Example](#heading=h.db3qgyq7jslw)

[Nominal, structural, free typing](#heading=h.6m3q8hph9vd0)

[Nominal](#heading=h.dle7jdukwspl)

[Structural](#heading=h.krxj1yw2j9wp)

[Free-form](#heading=h.a0koyyz8trgz)

[Compose](#heading=h.sv8aukgo8kdq)

[Types as annotation for DI](#heading=h.g3qeifllo5en)

[Closure Output](#heading=h.6qqe72oqt4tk)

# Annotations

### Glossary:

* Annotations: allow attaching meta-data to the code.

## Angular Use Case

In angular we would like to have the ability to attach meta-data to types. This meta-data would direct the angular framework when the type should get instantiated, how, and what methods should be called and when.

### Example:

|  |
| --- |
| @NgDirective('[ng-bind]')  class NgBind {  @Inject([Element])  constructor(element) {  this.element = element  }  @NgMapExpression('ng-bind')  setText(value) {  this.element.textContent = value;  }  } |

The annotations in the above code example direct Angular to:

* Instantiate class NgBind when ng-bind attribute is encountered in an HTML template.
* During instantiation we pass the current DOM element to NgBind constructor.
* Whenever the expression on ng-bind attribute changes Angular will call the setText method.

## Goals

While comparing different approaches it is important to keep these goals in mind:

* Angular will be written in ES6 and compiled into ES5 using traceur. The users of angular can write their code in ES6 or in ES5. If they are in ES5 then it is important that the code they need to write is something which a human could be expected to write. The syntax has to be straightforward without any complicated rules.
* The annotations / decorators need to be composable in a predictable manner. That means that adding more annotations / decorators should not have surprising side effects.

## Annotations

Annotations could be implemented in the following way.

|  |
| --- |
| function Inject(args) { this.args = args; }  function NgDirective(selector) { this.selector = selector; }  function NgMapExpression(attrName) { this.attrName = attrName; }  function NgBind(element) {  this.element = element;  }  NgBind.annotations = [  new NgDirective('[ng-bind]'),  new Inject([Element])  ];  NgBind.prototype.setText = function(value) {  this.element.textContent = value;  }  NgBind.prototype.setText.annotations = [  new NgMapExpression('ng-bind')  ]; |

Note:

* The resulting ES5 code is something that we can expect a user to write
* Adding more annotations does not remove/overwrite/destroy the existing ones.
* There is a well known location where the annotations are placed. This aids in discoverability / reusability, for example in the case of a RAD tool.

## Decorators

The decorators approach could be implemented in the following way.

|  |
| --- |
| function inject(fn, args) {  fn.inject = args;  return fn;  }  function NgDirective(fn, selector) {  fn.selector = selector;  return fn;  }  function NgMapExpression(fn, attrName) {  fn.mapExpression = attrName;  return fn;  }  var NgBind = NgDirective(inject(function NgBind(element) {  this.element = element;  }, ['Element']), {selector: '[ng-bind]'};  NgBind.prototype.setText = NgMapExpression(function(value) {  this.element.textContent = value;  }, 'ng-bind'); |

Note:

* The resulting ES5 code is something that we can expect a user to write. It is a bit hard to read, since the function is the first argument, but we could make decorating function the last argument to make it more readable. However, there is no easy way to decorate function parameters.
* Adding more decorators or adding them in the wrong order may change the behavior of the system. (because decorators can return delegates which hide annotations)
* Directives can execute code, but there is a limited amount of things they can do without resorting to global variables (this will be explored later).

### Decorating Function Parameters

An important goal is that decorators syntax in ES5 is something which an ES5 user using ES6 transpiler output could use. The tricky part becomes when we wish to decorate parameters in the function. In this example the class needs two Elements. One for the element where the directive is defined and one for the root of the application. Differentiating between the two Elements requires a way to decorate the elements so that they can be distinguished.

|  |
| --- |
| class DialogBox {  constructor(element, @AppRoot rootElement) {}  } |

Using annotations this may be expanded in ES5 as:

|  |
| --- |
| function AppRoot() {}  var DialogBox = function(element, rootElement) {}  DialogBox.parameters = [[Element], [Element, new AppRoot()]] |

Using decorators a possible solution may look like this:

|  |
| --- |
| function AppRoot(fn, pInfo) {  var parameters = fn.parameters;  if (!parameters) parameters = fn.parameters = [];  var annotations = parameters[pInfo.pos];  if (!annotations) annotations = parameters[pInfo.pos] = [];  return fn;  }  var DialogBox = AppRoot(  function(element, rootElement) {},  {pos: 1}); |

This is not something we can expect the users of directives in ES5 to write. The need to pass in additional object with the right structure is too cumbersome to expect that users would get it right most of the time.

### Decorators can Interfere

Decorators can return functions other than the function which they are decorating. In this example the log decorator would break the previous decorator which augments the function if they are applied in the wrong order.

|  |
| --- |
| class NgBind {  @log  @NgMapExpression('ng-bind')  setText(value) { ... }  } |

The issue here is that the log is implemented using a delegate

|  |
| --- |
| function log(fn) {  return function() {  return fn.call(this, arguments);  }  } |

### Decorators encourage global variables.

Decorators could be used to automatically register the type/function with the right part of the system. There are some issues with this approach:

1. *Global state*: Decorator does not get any additional references passed into it. This means that if the decorator wants to register the type with existing system it would have to rely on existing global variables to register the type with. Global variables are problematic.
2. *Reverse dependency on membership*:The dependency between the type and the group it belongs to should be from the group to the type, not from type to the group. If the membership is declared on decorator then each type can only be part of a single membership. This is counter to what is often needed in tests. In tests it is often desirable to create different test groups with different type memberships for testing. It seems restrictive that a type has external dependency on other part of the system, rather than the system having dependency on the type.

### Example

In this example if NgDirective decoration would try to register the NgBind type with the system it would have to rely on a global variable.

|  |
| --- |
| function NgDirective(type) {  globalDirectiveRegistry.register(type);  return type;  }  @NgDirective({ selector: '[ng-bind]' })  class NgBind { ... } |

Additional issue is that it is not possible to refer to the different directives in tests.

|  |
| --- |
| // uses global variable to register itself  @register(ExceptionHandler)  class LogExceptionHandler extends ExceptionHandler {}  // This @register will override.  @register(ExceptionHandler)  class RethrowExceptionHandler extends ExceptionHandler {}  it('should log exceptions', function(exceptionHandler) {  // Tests have no control over which ExceptionHandler is used  // One of these two tests will have to fail  doSomethingWhichThrows();  expect(exceptionHandler.errors).toEqual(['myError']);  });  it('should rethrow exceptions in tests', function() {  // Tests have no control over which ExceptionHandler is used  // One of these two tests will have to fail  try {  doSomethingWhichThrows();  throw 'should have thrown';  } catch(e) {  expect(e).toEqual('myError');  };  }); |

### Decorator Summary

Following summarizes why we would prefer to have Annotations instead of Decorators. Decorators could be used for these use cases:

1. *Monkey-patching*: Decorators can be used to monkey patch the types/functions with additional data. This can also be handled by annotations. The benefit of annotations is that they are on a well known location which encourages interoperability between libraries.
2. *Mix-ins*: Decorators can be used to add additional methods to a type.
3. *Delegating*: Decorators can create new functions which can delegate to the original function. This is a valid and useful use case, but it has the unintended consequence that it hides the monkey-patching data attachment from the user of the function.
4. *Registration*: Decorators can be used to register types with global registry. The issue here is that since the decorator function is global it can only get references to the registries which they themselves are global. In addition it creates a dependency in the wrong direction. From the type to the global registry rather than from a specific registry to a type. In addition there are often use cases when a single type needs to be registered with multiple registries.

I am not aware of other use cases, but would like to hear about them.

# Contracts / Dynamic Types

## Static Types

For static types to be useful most of the system needs to be typed. If a function declares that it needs an argument which is a string the compiler must be able to statically verify that all the call sites do indeed pass in strings. This is complex, since the parameter may be a computed value from another function. We then have to recursively verify that such a function returns a string. The net effect of this is that types spread throughout the system. This creates several issues:

* 3rd party libraries are mostly untyped. This requires that the some form of extern file is created for the third party library. This extern file is just a cast and it does not actually prove that it the 3rd party library conforms to the type.
* Server side JSON responses also need to follow expected type, but there is no way to verify them at compile time.

## Contracts

The above two issues can be solved through runtime assertions which validate types. It is better to think of these runtime-types as contracts to better differentiate them from static types.

By inserting a check into development code which verifies that the parameter is of the expected type we can verify that the function is called with the expected type without requiring that the callers of the function need to declare the type.

These runtime checks have a performance side-effect, but they can be removed in production, once the developer is convinced that the application functions correctly. The checks also provide an early warning system in tests.

## Contracts and Static Types

It is the goal that contracts work along side static types. The same annotation/syntax which declares type for static checking, should be the one used to generate the runtime assertion.

### Example

|  |
| --- |
| function concat(a:string, b:string) {  return a+b;  } |

Would generate these assertion at runtime:

|  |
| --- |
| function concat(a:string, b:string) {  assert(typeof a == 'string');  assert(typeof b == 'string');  return a+b;  } |

## Nominal, structural, free typing

### Nominal

The most commonly understood type system is nominal. In this case an object is an instance of a Type if it is either a direct instance of a type or it inherits from the type.

|  |
| --- |
| assert(a instanceof MyType); |

### Structural

JavaScript is a dynamic language. A method 'x' can be invoked on any instance which has method 'x'. For this reason dynamic systems can better be thought of as structurally typed. An object conforms to a given type if it implements all of its methods.

|  |
| --- |
| // simplified code  function isStructuralType(obj, type) {  for(var name in type.prototype) {  if (typeof obj[name] != typeof type.prototype[name]) {  return false;  }  }  return true;  }  assert(isStructuralType(a, MyType)); |

### Free-form

In JavaScript some types are even looser than structural. For a lack of a better name we will call them free-form. In this case a developer can specify a function which can be used to assert the validity of the type. For example ArrayLike.

|  |
| --- |
| // simplified code  function isArrayLike(obj, itemType) {  if (!isNumber(obj.length)) return false;  for(var i = 0; i < obj.length; i++) {  if (!isOfType(obj[i], itemType)) return false;  }  return true;  }  // For example jQuery would pass this type.  assert(isArrayLike(a, Node)); |

### Compose

Because JavaScript does not have type annotations, libraries often use the parameters in a very free flow way. For this reason types need to be composable in the form of union. Types also need to be parameterized, as in the above example where ArrayLike takes additional type which proves that it is an array of Nodes.

## Types as annotation for DI

Type information also needs to be preserved at runtime for the purposes of dependency injection.

|  |
| --- |
| class MyClass {  @inject  constructor(a:TypeA, b:TypeB) {}  } |

should generate something like this

|  |
| --- |
| function MyClass(a:TypeA, b:TypeB) {}  MyClass.parameterTypes = [TypeA, TypeB]; |

NOTE: It is not clear to me yet, if we should generate types for all methods and let the minification process tree-shake, or if we should only generate the methods which are in some way annotated.

# Closure Output

The generated output from the traceur needs to be human readable. Preferably preserving as much of the original formatting, spacing, comments as possible. This is important because libraries such as angular should be available in two formats.

1. A human readable development format the developer may choose to work with ES5 in which case the compiled version of library needs to be readable for debugging purposes.
2. Production version which is fully minified.

In addition to preserving the formatting traceur should also be able to generate output which is decorated with the Google Closure Compiler annotations. This is important so that the ES6 can be incrementally introduced to an existing Closure Compiler ES5 project.