**AtScript has been replaced with TypeScript; see the** [**blog post**](http://blogs.msdn.com/b/typescript/archive/2015/03/05/angular-2-0-built-on-typescript.aspx)**.**

This document is historical.

*Status: Abandoned*

*Author:* [*misko@google.com*](mailto: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*

# Goal

JavaScript, the de facto language of the browser, has a large, thriving community. However, it is missing some features which would make it much more manageable for large-scale application development. The goal of AtScript is to enhance the language with these missing features without infringing upon its current capabilities.

## Enhancements

* **Type Annotations**: Types allow us to explicitly express contracts between different parts of the system and to give these contracts names. For large teams with large codebases, where no single person knows the whole codebase, types become a key way to discuss and reason about the collaboration between different components of the system.
* **Field Annotations**: Types are not only about contracts (API) but also about structure. In JavaScript the fields are implicit; they are created as a side effect of assignment. By giving the developer syntax to explicitly declare the fields, and warning upon implicit creation, we allow them to think about types in a more concrete way. By doing this we in no way sacrifice the flexibility of adding fields dynamically. We simply provide an optional tool to developers who can benefit from being more explicit.
* **Metadata Annotations**: Many things in programming are best expressed in a declarative fashion. Declarative annotations allow frameworks to understand the meaning of your code without forcing a complex inheritance or lifecycle API. Examples where this technique has been used in the past include dependency injection libraries and web frameworks.
* [**Type Introspection**](http://en.wikipedia.org/wiki/Type_introspection) **with Annotation Support**: When we have annotations, it’s important to provide a consistent API that developers and frameworks can leverage to gain access to this information at runtime. The reflection API should also be backwards compatible with ES5 and ES6.

## Philosophy

* **ES6 Baseline**: ES6, or ECMAScript 6, is the latest version of the Ecma International language popularly know as JavaScript. It includes many improvements such as a formal class declaration syntax, promises, modules and consistent variable scoping.
* **Backwards Compatibility**: Types, fields, metadata annotations and reflective access need to be added in a way which does not break the existing syntax or semantics of ES6/ES5. AtScript needs to be a more succinct way of expressing what is already possible in these languages. ES6/ES5 code must be valid AtScript code without any changes to the semantics. Any developer who has written ES6/ES5 code can immediately get started with AtScript. In other words, ES6/ES5 are strict subsets of AtScript. All the code you are used to writing today will work without alteration with AtScript, but now you can take advantage of the enhancements to the language which will allow you to express your ideas in a more concrete way.
* **Familiar Syntax**: We desire to use a syntax which has been tried, is time tested and is most likely to become the next standard. This means using ':' to indicate type annotations (proposed in ES4 and used by TypeScript), '@' for metadata annotations (used by java, dart and others) and 'name:Type;' for field annotations (used by java, dart, TypeScript and others).
* **Pragmatic Approach**: The goal is not to build a type system that is correct 100% of the time. Rather, we want to be flexible and useful in the vast majority of real-world scenarios.

**Semantics Agnostic**: Type system theory is a complex field with many tradeoffs. It is our explicit goal to leave the semantic discussion as well as the assertion system outside of the scope of AtScript. Instead we want to provide hooks for others to build [runtime type assertion](http://angular.github.io/assert/) libraries and static analyzer tools with their own semantics. In this way we believe that we can spur innovation in the area of type systems by lowering the barrier to entry.

# Type Annotations

## Syntax

Nominal types were originally proposed for JavaScript in the [ECMAScript 4 specification](http://www.ecma-international.org/activities/Languages/Language%20overview.pdf). Though this was never implemented in JavaScript, it was adopted and widely used via Adobe's ActionScript3 and inspired others. e.g. [TypeScript](http://www.typescriptlang.org/) adopted the syntax, but uses a structural type system semantic.

Our proposal is to use this previously vetted syntax by using a ':' and type declaration after the variable, parameter or function as seen below. Notice that translation to ES6 simply strips the type information.

|  |  |
| --- | --- |
| AtScript | ES6 |
| class MyClass {  methodA(name:string):int {  var length:int = name.length;  return length;  }  } | class MyClass {  methodA(name) {  var length = name.length;  return length;  }  } |

## 

## Runtime Type Assertions

While stripping out the types from the source allows us to turn AtScript into valid ES6, it does little to assert the validity of the code other than by providing documentation. One way to take advantage of the types is to have the transpiler, e.g. [traceur](https://github.com/google/traceur-compiler), generate type assertion statements as shown below. Notice that the transpiler is unaware of the semantics of the types, and it leaves all of that information to a third party [rtts library](http://angular.github.io/assert/). Leaving the assertions to a pluggable library will allow others to experiment with different type system semantics.

|  |  |
| --- | --- |
| AtScript | ES6 |
| class MyClass {  methodA(name:string):int {  var length:int = name.length;  return length;  }  } | import \* as rtts from 'rtts';  class MyClass {  methodA(name) {  rtts.types(name, rtts.string);  var length =  rtts.type(name.length, rtts.int);  return rtts.returnType(length, rtts.int);  }  } |

### 

### Why use a runtime type system?

* It allows free mixing of existing libraries which do not have type annotations with new code which can take advantage of types. The incomplete type information would prevent useful static analysis of code but runtime type checks do not suffer from such limitations.
* The semantics of types can be left to the implementor of the rtts library. This means that different libraries can choose different strategies of identifying types. e.g. nominal vs structural. Our hope is that developers will be able to experiment with new ways of using types.
* Runtime type verification can be used to verify that the JSON returned from the server is of a valid structure, something which cannot be done statically.
* Since the type errors should be caught and resolved before pushing the code to production, all of the assertions can be stripped from production builds, completely removing the cost of these assertions.

### Prior Art

* This runtime type system is modeled after Dart’s runtime type assertion. Dart users working on large scale applications point out that optional types and runtime type assertions work well in practice.

## Generics

Generics allow developers to further constrain the type system of the language and are especially useful when used with data structures such as Maps, Arrays and Promises.

|  |  |
| --- | --- |
| AtScript | ES6 |
| class MyClass {  methodA(names:List<string>):List<int> {  var sizes = [];  for(var i = 0; i < names.length; i++) {  sizes[i] = names[i].length;  }  return sizes;  }  } | import \* as rtts from 'rtts';  class MyClass {  methodA(names) {  rtts.types(names, Array.of(rtts.string));  var sizes = [];  for(var i = 0; i < names.length; i++) {  sizes[i] = names[i].length;  }  return rtts.returnType(  sizes,  Array.of(rtts.int));  }  } |

The generic information is fed into the runtime type verification which can be used to further constrain the kind of types which are allowed. For example, it is possible to assert that the array consists of strings which are in turn CSS selectors, something which is difficult to do in most static type systems.

## Static Type Analysis and Type Inference

The runtime type system can further be supplemented by writing a static analyzer tool which, with the help of type inference, can identify some of the errors without actually executing the code. The helpfulness of the static tool will be proportional to the amount of types present in the source code, and will complement the runtime assertion.

### Prior Art

* Static code analysis is the defacto standard for type verification. There is a lot of prior art with which the developers are already familiar.

Many existing type systems, and Dart in particular, already introduced the concept of an analyzer which fills in the missing type information through inference and provides additional type checks. The analyzer can detect many of the errors statically and provide instant feedback to the developer. The combination of static analysis as well as runtime assertions have proven to be valuable in the Dart community.

# Field Annotations

As mentioned earlier, explicit field declarations are needed to fully describe the type’s structure. This syntax has been proposed in TypeScript, and should be familiar to most developers.

|  |  |
| --- | --- |
| AtScript | ES6 |
| class Point {  x:int;  y:int;  } | class MyClass {  constructor() {  this.x = null; // auto-generated  this.y = null; // auto-generated  }  } |

By specifying the fields in this way, the transpiler can generate the constructor which then pre-creates the fields. By creating the fields in the same order it will allow most virtual machines to optimize their hidden class system, thereby enhancing performance. Forgetting to declare a field will still produce working code, but the static analyzer could generate a warning to notify the developer of the potential mistake.

# Metadata Annotations

Annotations are a declarative way of attaching metadata to the code. The simplest form of annotation is the type information itself.

|  |  |
| --- | --- |
| AtScript | ES6 |
| class MyClass {  constructor(name:string, list:Array) {}  } | import \* as rtts from 'rtts';  class MyClass {  constructor(name, list) { }  }  MyClass.parameters = [  {is: rtts.string},  {is: Array}  ]; |

The parameter type information is retained at runtime and made available to the rest of the system. This is useful for frameworks, such as dependency injection, as shown in the next, more realistic example.

|  |  |
| --- | --- |
| AtScript | ES6 |
| class Inject {}  class Component {  selector:string;  constructor({selector:string}) {  this.selector = selector;  }  }  class Server {}  @Component({selector: 'foo'})  class MyComponent {  @Inject()  constructor(server:Server) {}  } | import \* as rtts from 'rtts';  class Inject {}  class Component {  selector;  constructor({selector}) {  this.selector = selector;  }  }  class Server {}  class MyComponent {  constructor(server) {}  }  MyComponent.parameters = [{is:Server}];  MyComponent.annotate = [  new Component({selector: 'foo'}),  new Inject()  ]; |

# 

# 

# Type Introspection

An explicit goal of AtScript is to have the type annotation data available at runtime to be used by projects such as various dependency injection libraries or AngularJS. At runtime a framework needs access to a function's annotations, parameter types (and their annotations) and return types. For a constructor we also need its field metadata including names, types and annotations. The type and meta-data annotations provide important control information for frameworks which would be more difficult and error prone to express imperatively.

|  |  |
| --- | --- |
| AtScript | ES5 |
| @Component()  class MyApp {  server:Server;  @Bind('name') name:string;  @Event('foo') fooFn:Function;  @Inject()  constructor(@parent server:Server) {}  greet():string {}  } | function MyApp() {}  MyApp.properties = {  'server': { is: Server },  'name': { is:string,  annotate: [new Bind('name']},  'fooFn': { is:Function,  annotate:[new Event('foo')]}  }  MyApp.annotate = [  new Component(),  new Inject()  ];  MyApp.parameters = [  {is:Server, annotate:[parent]}  ];  MyApp.prototype.greet = function() {}  MyApp.prototype.greet.returns = string; |

### Notes

* AtScript annotation syntax is just a shorthand of placing the same information in ES5. It would be reasonable for an ES5 developer to write these annotations manually. A helper library could even be provided.
* Annotations can only be placed on functions.
  + An annotation placed on a class is an annotation placed on the class’ constructor function.
  + An annotation placed on a field gets moved to the constructor function.
* All annotations are translated as properties on a function.
* Each type or meta-data annotation translates as follows:
  + @Foo(args)becomes new Foo(args)
    - Instantiates a new instance of the given type, allowing parameter passing.
  + @bar becomes bar
    - Reuses the existing instance.

NOTE: Mechanisms to limit the amount of runtime type annotations available at production is needed to prevent delivering dead data and bloating the applications. This is outside the scope of this document.

# Current Status

The Angular team started working on AtScript over a year ago in an effort to produce a more maintainable and easier to understand Angular code base. The goal was to also allow applications to be written in AtScript syntax, but AtScript, or ES6 should not be required when writing an angular app, so it was important to keep backwards compatibility with ES5.

Currently we are in the midst of writing Angular v2 on top of the AtScript syntax. Here is an inventory of what is working today:

* [Traceur](https://github.com/google/traceur-compiler) transpiler:
  + Consumes the AtScript syntax. (type annotations, meta-data annotations, fields)
  + Generates ES5 code with runtime type assertion statements for the [rtts library](http://angular.github.io/assert/) and with meta-data annotations.
  + Transpiles AtScript into Dart code. (Does not do semantics matching, requires developer to use facades)
* [RTTS](http://angular.github.io/assert/) Library:
  + Asserts nominal types.
  + Asserts structural or custom types.
  + Partial support for generics. Generics support in arrays and maps only.

We would like to add:

* A stand alone tool for static analysis of the type system similar to Dart analyzer or TypeScript.
* More support for generics in the runtime type system, beyond arrays and maps.
* ES5+[ClosureAnnotation](https://developers.google.com/closure/compiler/docs/js-for-compiler) output to allow integration with existing Closure projects.
* Source maps support for easier debugging and translation of stack trace line numbers.

# Prior Art

The AtScript has been influenced by TypeScript and Dart. Here we would like to discuss why these existing solutions do not meet our needs.

**TypeScript**

* Types are analyzed statically only.
  + Static analysis makes it difficult to have optional types, since static analysis can not analyze what is not typed. (type inference has its limits)
  + Static analysis can not be used to assert that the JSON returned from the server has valid structure.
* Lacks meta-data annotations
* Provides no mechanism to access the annotations at runtime

**Dart**

* Dart is semantically different from JavaScript. The resulting Dart2JS code uses JS as more of a VM than a language. The result is that Dart2JS code can not easily interoperate with JS code. Only primitive types (such as strings, numbers, arrays, maps) can be passed between Dart and JS. Complex things such as classes can not be marshaled easily.
  + Existing JavaScript libraries can not be used from Dart in a straightforward way. Limiting reuse of prior art.

# Future of AtScript and TypeScript

Both of these languages are solving similar problems. The fundamental difference today is that TypeScript is based on ES5 and AtScript is explicitly based on ES6. Once TypeScript aligns with ES6, we intend to be a strict superset of TypeScript. We are [collaborating with the TypeScript](http://blogs.msdn.com/b/typescript/archive/2014/10/22/typescript-and-the-road-to-2-0.aspx) team on this effort.

# Standards

Our plan is to use AtScript for [building Angular and optionally let users of Angular leverage AtScript](https://docs.google.com/document/d/1cUTD8oVzfpwFqX5tMxHTifKO8uJm5VddwmB0aVQMxpI) to build their own applications. AtScript is independent of Angular and as such we think it would be useful for other non-Angular projects. Once we prove the usefulness of this approach and get a better handle on the corner cases by building real world applications, we would like to offer it to the standardization bodies. We think it is generally agreed upon that types are something people would like to see in JS. What seems to be under contention is: What do these types mean? AtScript specifically defines syntax, but stays clear of the semantic discussion, therefore we think it would be a good introductory step towards official types in JavaScript.