Minutes of the: Ecma TC39-TG1
held in: Mountain View (Mozilla)
on: 13th of October 2005

Attendees

- Francis Cheng (Macromedia)
- Jeff Dyer (Macromedia)
- Brendan Eich (Mozilla)
- Gary Grossman (Macromedia)

Agenda

- Process for writing the ECMAScript Edition 4 specification
- Review of A-List features for ES4
- Drill down on ES4 Types feature
- Drill down on ES4 Classes feature

ES4 Spec Process

Brendan thinks it's important to have checkable or executable representation of the spec. He will investigate this offline.

Review of A List

Getter/Setters

Brendan: There is a widely used getter/setter syntax within object initializers used all over Firefox, different from the "function get / function set" syntax used in Waldemar's spec:

```
var o = {
    get p() { return ++this.x; },
    x:0,
    set p(nx) { return this.x = nx; },
    get * (id) { ... },
    set * (id, v) { ... }, };

 o.p => 1
 o.p => 2
 o.p = -1 => -1
 o.p => 0
```

The "get *" and "set *" syntax is like __resolve in ActionScript.

Brendan feels that this kind of extension is so popular and prevalent that we should attempt to standardize it.

Trailing Comma

```
var a=[1,2,] assert(a.length === 2)
var o={p:1, q:2,}              // Error
```

The ES3 spec forbids trailing commas in object initializers, but not in array initializers where it must be allowed (to make a hole at the end with two trailing commas). Brendan feels the spec is wrong and that we should standardize on permitting the trailing comma, since it is something people like to use in real code.

Drill down on types

We reviewed Section 6.5 of Jeff's language spec regarding Types.

Jeff: The distinctions between implicit conversions and explicit conversions is going to go away in our spec. Everything will be an implicit conversion. The implicit and explicit conversion tables will be merged together and will be equivalent. The conversion rules are pretty much the same as ECMA-262 Edition 3, with the exception of when null is assigned to a nullable type. Implicit conversions will convert to null. Explicit will convert to the string "null".

Brendan: All ints and uints can be expressed as doubles, but can't model the integer arithmetic and overflow, and we're not going to have exceptions for overflow.

Jeff: "is" table: How do you feel about value set checking at runtime? 1.23 is Number, but 1.23 is not int or uint.

Brendan: This is where we want "is" to be a predicate, not a converting operator. Whereas "as" converts if it can.

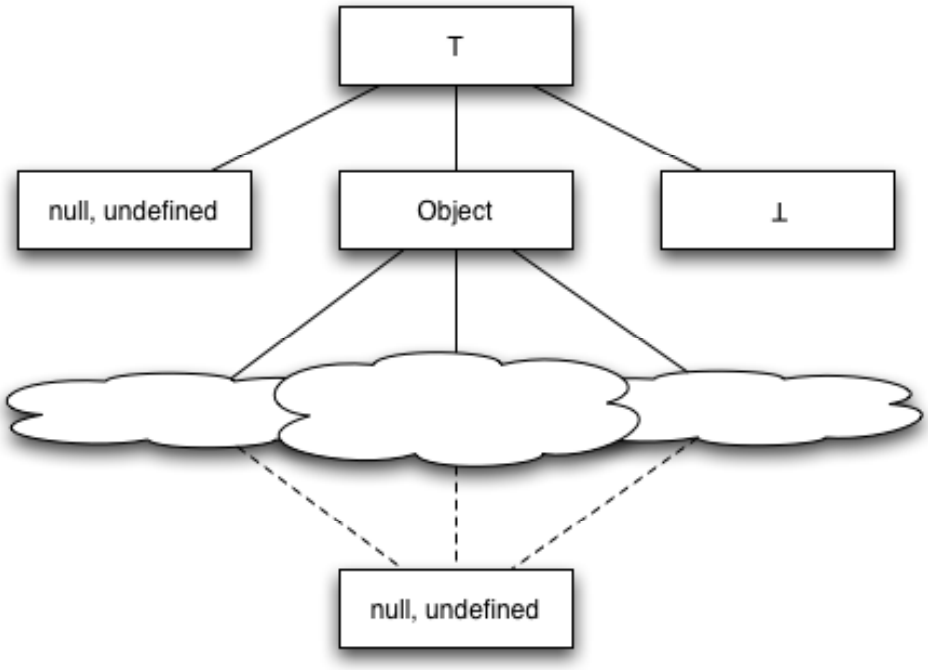
Brendan: Will int, uint be lowercase type names like in C?

Jeff: Yes

Brendan: Top and bottom types. I was arguing that Object should not be the top type. Arguably we have two bottom types, null and undefined, historically. The uninitialized variables have value undefined. null denotes a null reference, a null object. Kind of useful to have that distinction, or not, but now we're stuck with it.

Brendan: Undefined and null should not convert to Object. ToObject in Edition 3 on null/undefined throws a TypeError exception. In this table, it does not, so the table has a bug.

Brendan: It's cool that you are reasoning about types at compile time, but I don't think Object should be the top type. There should be a top type that includes null and undefined. Then we can restore the TypeErrors in the table for those coercions.



Nullable Types

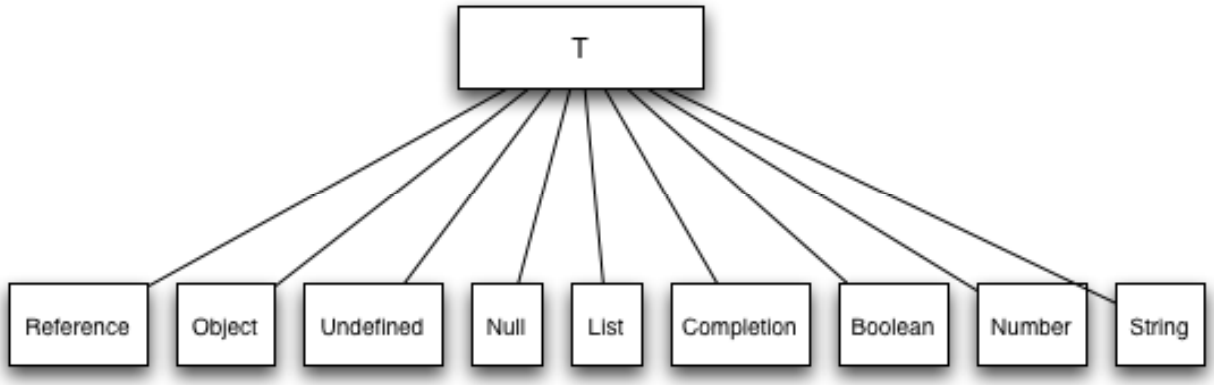
We explored the idea of using nullable types to resolve the type conversion issues with null and undefined.

```
String : { UTF-16 unichar sequence }
String? : String | null
Object? would mean any value of type Object, or null
Object! would mean Object and only Object, null not permitted
```

In ActionScript today, var x:Object, the default value is undefined. So the presence of Object is the same as the absence of a type annotation.

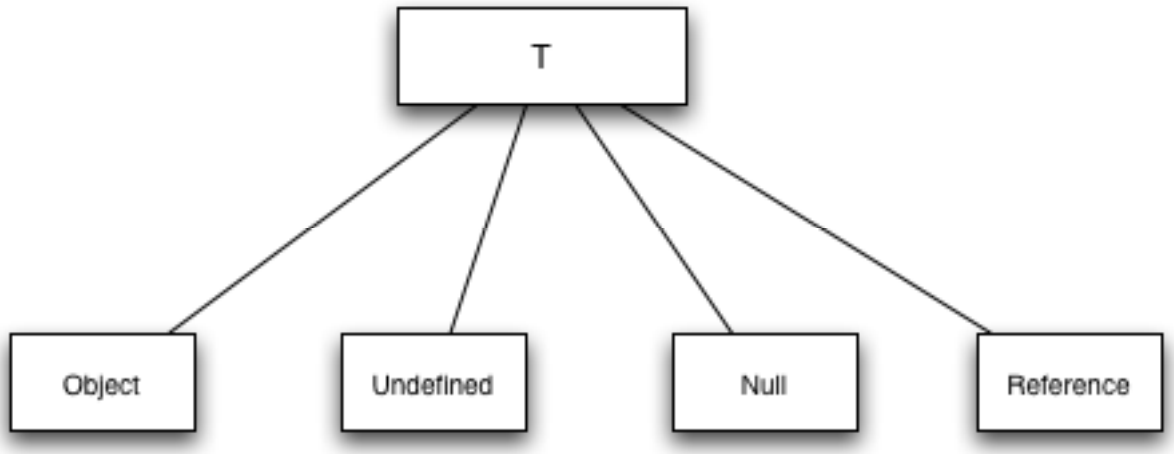
Object is currently the top type, therefore the table in 6.5.3 is not compatible with Edition 3. The effect of the current incarnation of the table is that we've moved Object up in the type tree, which causes some issues.

These are the internal types in the ECMAScript Edition 3 specification. There is no explicit top type in the Edition 3 spec, but one could imagine there being an implicit one, T:



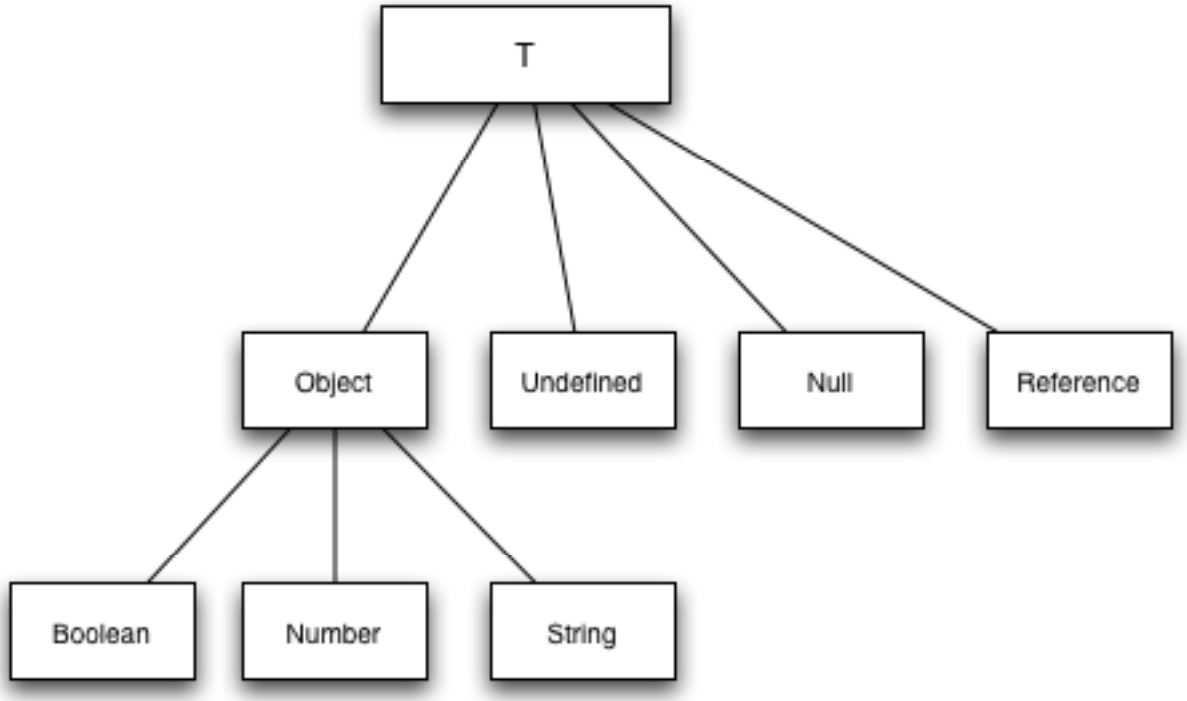
So, there are nine types in ES3, three of them being internal: Reference, List and Completion.

We're trying to make them all derive from Object, except for Null Undefined



Jeff: Completions are gone, due to the execution model of statements. Past statements pass in their completion values. Type is Object instead. Completion is three things: a value, a target, a state.

Brendan: Seems like there could be an internal spec type called Completion, don't see why not since it was in Edition 3.




```
   undefined is Boolean -> false   undefined as Boolean -> false   undefined as
String -> "undefined"   null as String -> null
```

Brendan: Why did Waldemar make string nullable? [Answer from Waldemar, via Brendan: by fiat, to match common use-cases, as Java did.]

Jeff: Our users want it too, it's useful to be able to say no value.

Brendan: If we want the type system to work without special cases for values, it will make the spec and implementation broken or ugly.

```
   Boolean <: Object   Boolean? <: Object?   Null <: Boolean?
```

Conversion table that maps type to type using capitalized type names. Then some cases collapse, and do get TypeError conversion. Get Edition 3 behavior. We should systematize things and use set-theoretic type system, so we can reason about them.

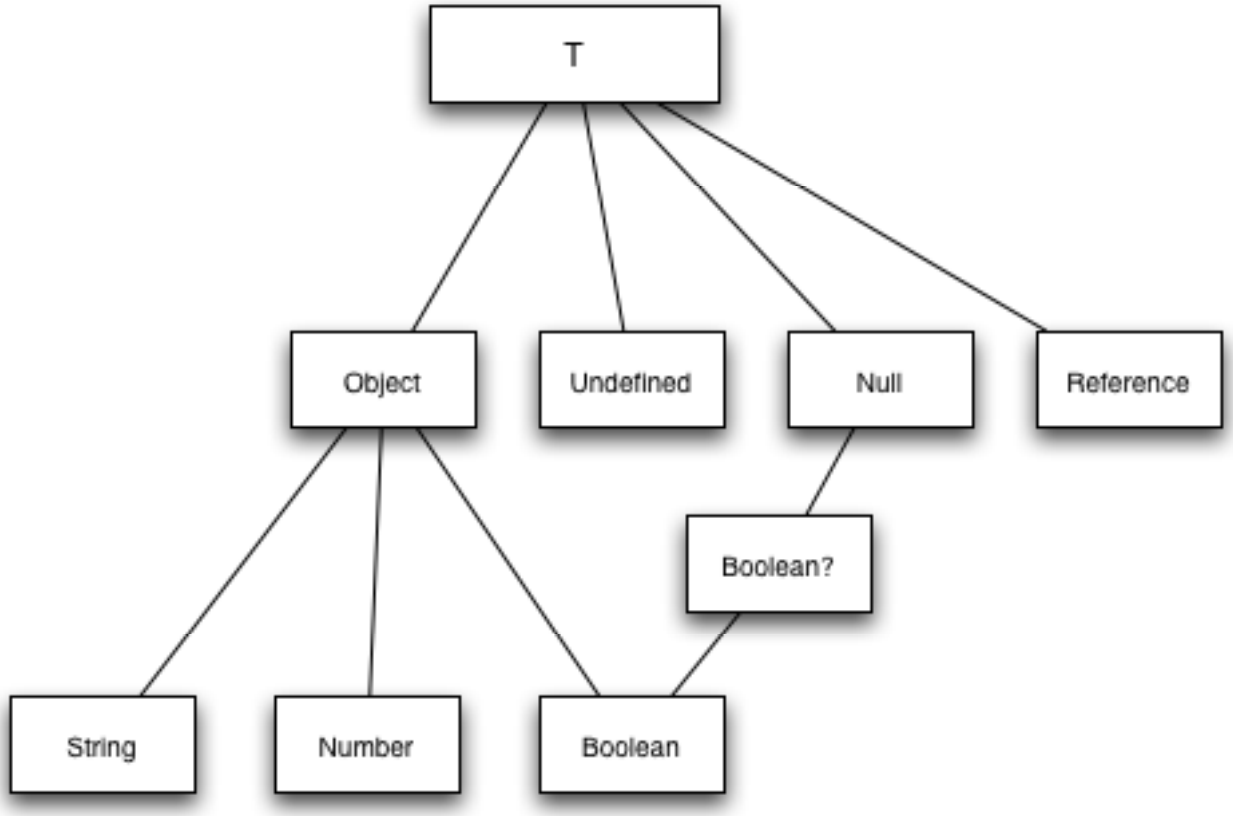
Soft spots:

- `null` to `String`
- `undefined` to `Object`

Use cases to solve:

- `String~` - always does conversion to String
- `String!` - can be string and only string
- `String?` - nullable

Brendan: Some languages are building in preconditions, postconditions, invariants I prefer the type system to enforce things, so that people don't have to write boilerplate code to validate parameters.



Right now, in Jeff's language spec, Object and the lack of a type annotation are equivalent. Should we change that, so that not having a type annotation means something different, i.e. type Any?

For instance, type Object could be sealed instead of dynamic, so that the language is safer by not allowing Object to be a free-for-all. Then, accessing unknown properties of a reference of type Object would be an error, whereas it would have the standard ES3 behavior if the type annotation was Any or absent.

```
var x   var y:Object   x.foo -> undefined   y.foo -> error?
```

Type Unions

Brendan thinks we should have some ability for type unions in the language.

```
var x : Number | Null
```

It would be good to have something like this because otherwise developers end up enforcing their invariants on their own, with the potential for buggy code.

Recap from types discussion

- Revisit Any vs. Object, try to work Any into the type system. Jeff will investigate.

- The biggest issue is null/undefined with String and Object.

Classes

Brendan: Lots of people write me, asking me not to add classes to the language. Now, I'm resigned to doing classes, and also I want them in the language for two reasons: bootstrapping and empowerment. But, we must be careful to avoid the C++ black hole.

Duck Typing

```
function m (s:Shape) {   print(s.radius); }
```

Suppose this is a block, and you've tested if (s is Circle), and don't want to rename it. This kind of duck typing is convenient; Python users, etc. like it.

Is whether this is an error controlled by whether you are in strict mode (Bang)?Or is it dependent on whether you are using sealed or dynamic classes?

Sealed vs. dynamic

In Jeff's spec, a class is sealed by default. The "dynamic" attribute may be used to make a class dynamic. Sealed classes are not expando; they have no internal hashtable and properties cannot be added at runtime. Dynamic classes have ES3-like behavior; they can have declared properties/methods in the class definition, but dynamic properties may be added too.

Sealed and dynamic classes alike have a prototype object. The prototype object is currently an instance of generic Object, although we may want to change that to be an instance of the class. However, if the prototype of a sealed class was an instance of the sealed class, then the prototype object is basically useless and vestigial, since you can't add properties to it.

If it is an error to access undefined properties on a sealed object with the "." operator, it's kind of weird since you can't get to properties declared on the prototype chain. This is only a compile-time error in the strict dialect.

Jeff is leaning is to make all objects expando by default, since they are already really expando due to the prototype chain.

However, this would be unpopular since it means that we basically can't intercept property not found conditions, so typos would get past the system.

Jeff entertained the idea of always making accessing an unfound property a ReferenceError, for every object, instead of returning undefined. That would be a pretty big break from ES3 behavior though.

The "dynamic" attribute means 3 things rolled up in one:

- The object is not sealed.
- The object is not checked.
- Undefined property accesses will not throw a ReferenceError exception.

Jeff wants to explore the idea of breaking these three aspects of dynamic out into separate attributes. Suppose there was a sealed attribute, and an unchecked attribute? After some discussion, the straw man of making all objects dynamic by default was rejected. There are good developer productivity reasons for making objects sealed by default.

Ultimately, we agreed that dynamic should continue to mean the 3 things it does today.

Prototype

We decided that the prototype of a class should be an instance of that class.

One weird idiosyncracy in the ES3 spec is that the RegExp prototype is a vanilla Object.

Another weirdness is that arguments is an Object, not a true Array... it should probably just be an Array in the ES4 spec.

If something has a [[Call]], typeof is supposed to say function. So what's typeof RegExp in the new spec? It's "function" in SpiderMonkey, but "object" in many other implementations.

Sealing built-in classes

```
var x:Number = 42;
... x.f = function() { global = this; }
x.f();
```

It is desirable to seal certain built-in classes, like Number, Boolean and String. Then they can be optimized and represented using their machine representations, instead of a wrapper object a la ES3.

(image here)

Jeff: If we didn't have a dynamic attribute, we could have a "primitive" attribute which means sealed.

Brendan: Ajax toolkits have a tendency to decorate Number and String to extend them. So, it's important to preserve that capability.

This should be disallowed:

```
(1).foo = "bar"
```

But it is desirable to extend Number and String via the prototype object. Namespaces as versioning

Brendan expressed interest in the idea of using namespaces as a versioning mechanism. We discussed Macromedia's internal experience with trying to use namespaces as a versioning mechanism, and running into various difficulties and shelving it.

In Macromedia's spec, only one namespace attribute can be used on a definition. This restriction was made to simplify the implementation, but also because developers were having a hard time understanding what it meant to decorate a definition with multiple namespace attributes.

It is certainly demanding to have to put multiple namespace attributes on every definition in your framework classes, adding a new one every time you release a new version.

Brendan wondered whether a definition could be both public and in a namespace. The Macromedia experience was that versioning worked best if the definitions were in a namespace and were not public, and then user code would use the desired version namespace. Finalization of built-in classes

The final attribute on a class definition makes it an error to subclass it, just like Java/C#.

In Macromedia's spec, String, Boolean and Number are final, since they may be represented by primitives at runtime, so subclassing them would be problematic. Brendan is fine with that.

XML and XMLList are also final. That also seems to make sense.

We may have swung the pendulum too far out, however, when we made Array final. Brendan favors making Array non-final.

Should also consider whether Date, RegExp, etc. should be final or not.

Explicit

The explicit attribute is from the ES4 Draft Proposal. When it is used on a definition in a package, it means that references to that definition must be fully qualified, even if the definition has been imported.

The use case is to reduce pollution of the user's open namespaces.

Brendan: I don't think we really need explicit...

Public, Private, Internal, Protected

The public, private, internal, protected access modifiers are all implemented using namespaces in the Macromedia implementation.

Brendan: If public, protected, private, internal can be implemented by namespaces, and that's basically the best way to do it, maybe we should say that in the specification. Protected namespace bug

Macromedia's first cut at implementing "protected" was to introduce a protected namespace for each class. Class A would declare a protected namespace. Subclass B would declare its own protected namespace, and would "use" both the A and B protected namespaces.

```
class A { protected int foo; } class B extends A { } class C extends B { }
```

The problem is that B can now see protected members on instances of C! That means if you have a common UIComponent base class with a protected update method, a Button component could invoke the protected update method of a ComboBox.

It seems like a solution is to copy traits from base classes into subclasses. If B only opens its own protected namespace, but copies A's protected traits into its own namespace, that might solve the problem.

Open definition namespaces

Since a class doesn't create its own public namespace, this C++-style "member hiding" is not allowed:

```
class A { public var x; } class B extends A { public var x; }
```

This is an error, and that makes sense.

However, local name collisions are also reported as errors:

```
class A { public var x; } class B extends A { private var x; }
```

This is an error, even though it doesn't seem like it has to be. If it wasn't an error, the user would get confusing ambiguity errors.

The scope overrides private::x, public::x should work, however.

Static inheritance not supported

The Macromedia spec does not support static inheritance, i.e.

```
class A { public static int x; } class B extends A {} print(B.x);
```

It was troublesome to implement because class members are defined using traits and slots just like instance members, just on the class object instead of on object instances. The aliasing was not simple to implement. We had it working for awhile using prototype chain, but that seemed like a kludge.

Brendan: C++ ran into ugly problems like contravariance of return types. This stuff scares me.

However, the Macromedia spec supports lexical references to static members from within a class, even superclass statics. Is this good to continue to allow?

Brendan would like to get rid of asymmetries in the model if possible. We will think about whether lexical references to statics should be disallowed.

It is already disallowed to reference a static member using an instance. It's permitted in Java and C++. This seems OK, C# disallows this too.

const

We are coming to the conclusion that initializers should be required on const.

For instance const members, you can write to const as much as you want during constructor.

Non-public constructors

We think non-public constructors, while requested by people wanting to do things like private constructors for singletons, are too complex to put in because in ECMAScript, the class object and constructor method don't lead a really separate existence and it's difficult for them to have different names.

We want to future-proof it though... so we require that a constructor's access specifier match the class's access specifier.

Super statements

Super statements are not required to be the first line in the constructor, but they are required to occur before any statement that references this. Jeff has a data flow analysis algorithm for determining this.

Must watch out for indirect eval, a.eval(s), called via a super expression.