ECMA/TC39-TG1/01/14

ECMA TC 39 TG 1 Meeting

Date:	27/08/2001
Location:	Microsoft, Redmond, WA
Convenor:	Peter Torr (MS)
Attendees:	Waldemar Horwat (Netscape)
	Patrick Beard (Netscape)
	Herman Venter (Microsoft)
	Eric Lippert (Microsoft)
	Peter Torr (Microsoft)

Preliminaries

The minutes of the last meeting, whilst late, were deemed OK by everyone.

ECMAScript 4 Processes

We had a small discussion about how we are progressing on the standard. We decided that the convenor should be responsible for nagging people to get assigned tasks done. We also decided to regularly have phone conversations between meetings. As far as actually writing the document goes, Waldemar wishes to spend the next few months writing up the core, and then individual chapters will be handed off to individual members for completion. We will use a process similar to ECMAScript Edition 3, where chapters go through various stages, etc. The current schedule for discussing the document is as follows:

* Chapters 9 and 10: August
* Chapters 5, 8, 11, and 12: September
* Chapters 13, 14, and 15: October
* Chapters 16, 17, 18, 19, and 20: November
* Chapters 21, 22, and 23.1: December
* Cleanup : January - March
* Chapters 1, 2, 3, and 4 can be written in parallel

Waldemar believes this schedule is aggressive, but achievable. There is a good chance of success.

Upcoming Communications

We will have a telephone conversation at 10 am on Friday, 14th of September. I will send out information about the call (number, meeting ID, etc.) closer to the date.

Our next meeting will NOT be at HP with the rest of the TGs, as Herman will not be able to attend. Instead, we will meet at Netscape on Monday, 1st of October, at 9:30 am.

Hoisting Algorithm

After discussing various options, we settled on a simple algorithm that preserves legacy code but enables true block scoping for ECMAScript 4 programs. The algorithm is as follows:

1. If all declarations of a variable are not decorated (they have no type annotations or other attributes), it is treated as an ECMAScript 3 declaration and it is hoisted to the enclosing function or global scope

2. If one or more of the declarations of a variable are decorated, the declaration is hoisted to the enclosing function or global scope, but the hoisted value will always generate an error when accessed

3. If a compiler can detect that access to a hoisted variable will always fail (ie, it is covered by #2 above), it may give a compile-time error; otherwise, the error occurs at run-time

One important implication of this algorithm is that a declaration in an inner scope will shadow declarations at an outer scope, thus making them inaccessible. For example:

```
// ---------- Begin Code ---------- \\

// Global x
var x : int = 42

function foo(b)
{
      if (b)
      {
            // Declare x local to the if block. Not only is this x
            // not accessible outside of this block, but the global
            // x is not accessible elsewhere in the function
            var x : int = 123
      }

      // Illegal; doesn't access the global x because the x declared
      // above is hoisted to function scope as a value that always
      // throws when accessed
      // This may be caught at compile-time or run-time
      print(x)
}

// ---------- End Code ---------- \\
```

It was also decided that loop control variables would be scoped to the block comprising the loop, not to the enclosing scope.

The .class Built-in Property
===========================

We decided to remove .class from the Netscape proposal as it was not clear what use the resultant value would be without a defined reflection mechanism in ECMAScript. There are no other areas of the proposal that are affected by this, but we will re-visit it in a future edition of the language.

Implicit Late Binding

Oftentimes, a script author may have a variable of t type T, but they know it will contain a reference to a sub-type of T, say U. It would be very convenient for them to access members of U directly from t, without performing a cast. For example:

```
// ---------- Begin Code ---------- \\

// Base class; no interesting members
class T {}

// Derived class adds a function 'f'
class U extends T
{
      function f() {}
}

// Factory method that is typed to return a T,
// but in some situations returns a U instead
function GetObject(flag) : T
{
      if (flag)
            return new U
      else
            return new T
}

// Declare variable of type T, but the author knows it
// will contain a U at run-time. This type may also have
// been inferred by the compiler if no explicit type was
// given.
var t : T = GetObject(true)

// Access U's method without an explicit cast
t.f()

// ---------- End Code ---------- \\
```

In situations such as the above, the compiler should emit a warning for the access to t.f, stating that it may fail at run-time, and generate a late-bound lookup for f. In this example, the lookup will succeed, but in other examples it may fail.

Arrays

We had another long discussion about arrays, but came to a conclusion this time. We decided the type hierarchy would look like this:

```
     Array
       |
    Array[T]
      / \
   T[]   List[T]
```

```
    |
   T[n]
```

Where:

* Array denotes any kind of array, with elements of any type
* Array[T] denotes any kind of array, with elements of type T
* T[] denotes a non-resizable of any size, with elements of type T
* T[n] denotes a non-resizable of size n, with elements of type T
* List[T] denotes a resizable of any size, with elements of type T

The rules for constructing / converting these are as follows:

* new T[5](a, b, c, d, e)

  Creates a fixed-size array of 5 items, all of type T

* new List[T](a, b, c, d, e)

  Creates a resizable array with 5 initial items, all of type T

* new List[T](length : 5)

  Creates a new resizable array with an initial length of 5 elements,
  all of type T (they have the default value)

* T[](existingArray)

  Converts an existing array to a fixed-size array of type T. If
  existingArray is not already a fixed-size array of T, this creates
  a copy of the array (you lose object identity)

* List[](existingArray)

  Converts an existing array to a resizable array of type T. If
  existingArray is not already a resizable array of T, this creates a

  copy of the array (you lose object identity)

Since there was some confusion and much white-board action during the discussion of arrays, everyone agreed to try writing some "typical" user code using this proposed syntax, and provide feedback to the list before the telephone conversation on September 14th. I will be sending out reminders!

We also had some discussions about parameterised types in general. For Microsoft's implementation, when the compiler sees an expression such as the following:

```
      new expr[val]
```

if <expr> resolves to a type, and <val> resolves to an integer, a new array of <expr> things is created with size <val>. Otherwise, ECMAScript 3 rules apply, and the <val> property is retrieved from <expr>, then called as a constructor. For example:

```
// ---------- Begin Code ---------- \\
var ob = { x : function() { this.bar = 'hello' } }

// An array of 5 integers
var vector = new int[5]

// An object with a 'bar' property with the value 'hello'
var hello = new ob["x"]

// ---------- End Code ---------- \\
```

Parameterised types need to be able to override the [] operator to allow things such as:

```
      var t = new Tree[10]
```

to declare a Tree of Objects with an initial size of 10 nodes, whilst:

```
      var t = new Tree[String]
```

would declare a Tree of Strings with a default initial size. To allow this, all types will have a default [] operator that returns an array, but individual types could override / overload the [] operator to return different values as needed.

Positional and Named Arguments

We had a long discussion about this, but again came to an agreement. The canonical example for Microsoft is the Office Object Model, which was designed with VB in mind. There are many methods in Office that take 20 or more parameters, all of which are both positional and named, and all have default values. JScript users would like to be able to call these the same way VB users can. One of the canonical examples for Netscape is the new List[T](...) example above -- it can take a variable number of arguments, or a named argument.

We decided on the following:

```
 * The formal parameter declaration must be in the following order:

      [positional-arguments] [positional-varargs] [named-arguments]

   or

      [positional-arguments] named-varargs

   and where positional-arguments can be broken down into:

      [required-positional-arguments] [optional-positional-arguments]

   + To declare a required positional argument, one uses:

        identifier [: Type]

   + To declare an optional position argument, one uses:

        identifier [: Type] = default

   + To declare positional varargs, one uses:
        ... identifier [: Type]

     where if specified, Type must be an array type, defaulting to
Object[]

   + To declare a named argument (which must be optional), one uses:

        named identifier [: Type] = value

   + To declare named varargs, one uses:

        ... named identifier

     where a type is not allowed; <identifier> is an ECMAScript
Object with
     expando properties representing the additional named arguments.

 * The caller must specify all positional arguments first (including
varargs),
   then any named arguments last. Conceptually, the callee will
receive three
   lists of parameters: A list of positional parameters, a list of
named
   parameter names, and a list of named parameter values. The
implementation must
   then map those to actual arguments of the function call.
```

Unicode Format Control Characters
=================================

Due to issues with not pulling out FCCs, we decided to reverse last month's decision, and continue with ECMAScript Edition 3's rules about stripping FCCs from String and Regular Expression literals.

Additional notes from Waldemar
==============================

We decided that a class has no inherent visible properties (not even toString) but can be converted to a string using the internal [[ToString]] method. The reason for this is a clash between a class's static members and its inherent properties (the instance members it has by virtue of being an instance of the type Type).

"new Object" will actually return an instance of a hidden subtype (I'll call it "Z" in this message) of Object. Objects of type Z are the only ones for which one can access their dynamic properties using the "a.x" notation. Other user-defined classes can be declared dynamic, which has the effect of providing a [] operator that supports reading, writing, and deleting dynamic properties. These properties would not be visible using the "a.x" notation. I don't think we need the "fixed" attribute any more. A prototype of an object of type Z must be either null or another object of type Z. Prototypes are not defined on objects other than type Z.

Herman asked that the binary operator dispatch algorithm be extended to consider implicit coercions. To avoid hijacking, one of the two operands must still be an instance of the class (or its subclasses) that defined the operator. The C# draft should contain more information. I'll look into this and see if I can make it work.

I'd like to add some more rationale for killing .class. The problem I had with it was that it compromised abstraction -- a factory method could be declared to return objects of class A but actually return objects of hidden subclasses of A. With unrestricted .class users could tell what subclasses these are and then prevent the factory method from being evolved.