ECMA International

ECMA/TC39-TG1/2002/3

Standardizing Information and Communication Systems

Minutes of the 13th TC39-TG1 Meeting held April 25th, 2002
Microsoft, Redmond, WA, USA

Present:

Rok Yu (Microsoft)
Herman Venter (Microsoft)
Eric Lippert (Microsoft)
Peter Torr (Microsoft)
Jeff Dyer (Compiler Company)
Waldemar Horwart (Netscape)
John Schneider (BEA) - Welcome!

Next meeting:

June 13th, Netscape, Mountain View, CA, USA

Agenda:

MIME Type
Class extensions vs prototype
Checked arithmetic
Hoisting to class scope
Indirect property lookup
Compile time constants and attributes
Definition conflicts
Overrides with namespaces

MIME Type:

As decided in October 2001, we re-affirmed our earlier decision to support **text/ecmascript** and **application/ecmascript** for existing Edition 3 content, and to add **text/ecmascript4** and **application/ecmascript4** for Edition 4 content. Server administrators who wish to set up an extension-to-MIME mapping can name their Edition 4 scripts differently from their Edition 3 scripts (eg, **.es** instead of **.js**) since the extension of external script files is not significant in a web browser.

Class extensions vs. prototype

Waldemar proposed going back to a prototype-based approach for extending classes. Herman points out that this causes problems for multi-threaded code, where one thread modifies the prototype and it is visible from other threads. JScript .NET does not allow modifications to prototypes in 'fast' mode (used for server-side processing) for just this reason.

The main difficulty with class extensions is the order of compilation / evaluation when a derived class clashes with an extension. For example:

```
class Base { /* ... */ }

class Derived extends Base { function foo() { /* ... */ } }

extends(Base) function foo() { /* ... */ }

var ob : Object = new Derived
var base : Base = new Derived

ob.foo() // which foo gets called?
base.foo() // which foo gets called?
```

The problem occurs when a program receives a `Derived` instance and the user invokes `foo` -- which `foo` gets called? One possible solution would be to add an implicit namespace to the `extends` that is implicitly `use`-d in the scope of the extension.

We decided that this form of extension was not required for Edition 4, and that we would move to a solution based on Edition 3. User-defined classes can be declared with a special attribute which gives them a writable `prototype` property, as per Edition 3. The default would not be to have such a property. The insecure nature of prototypes in Edition 3 was noted.

Checked arithmetic

Waldemar has issues with passing around the checked / unchecked state of the program at runtime to all the operators such as "plus". For JScript .NET, doubles are unchecked and the built-in types int, long, etc. are all checked and overflow if allowed. The built-in types for C# have two versions of each operation and the code generator calls the correct MSIL instruction. There is no way for a user-defined operator to provide checked and unchecked versions.

Waldemar would like to drop unchecked arithmetic (it is mostly a performance feature). Implicit conversions that overflow will fail, whilst explicit conversions will wrap around (this is currently how JScript .NET behaves). Floating-point expressions that result in a decimal portion of zero will be treated as integers, and the check for `is int` will return `true`.

Hoisting to class scope

Everyone agreed that declarations in blocks inside class declarations do no get hoisted into class scope. For example, a variable declared inside a block does not become a field of the class.

Indirect property lookup

Waldemar wants to prevent users from doing the following:

```
var s = "private::a" // A field scoped to the 'private' namespace
var x = ob[s] // Access someone else's private data -- oops!
```

Instead, he would prefer

```
var s = private::"a" // A special reference to this type's private field
        a
var x = ob[s] // Error; do not have access to another type's privates
```

Herman advocates a clean, orthogonal reflection mechanism that is a 'pay for play' feature. We discussed a light-weight reflection layer that would differ from the existing `for...in` and `[]` lookup, since they already have well-defined semantics and we do not wish to overload them again (for example, how would you enumerate properties of an object that was also a collection?). A basic example of a potential solution was given:

```
// Reflector is a new built-in object for doing reflection on 'ob'
var members = new Reflector(ob)

// Enumerate the public members of ob
for (var member in members)
    member.apply(ob) // apply the member to ob

// Retrieve a private field:
var f = members.GetField("private::x")
var x = f.apply(ob)

// or maybe
var y = Reflector(ob).GetPrivateField("a")
```

Rok will document this for our next meeting.

Compile time constants and attributes

Consider the following code:

```
const x = foo // an attribute

function bar()
{
    x const y = false // which 'x'?
    y const x = baz // hidden because 'y' is false
}
```

The usage of `x` is ambiguous. Herman wants scope hiding to be orthogonal to value hiding, so in the function above, the `y const x` definition hides the global definition of `x`, even though it itself turns out to be hidden. The variable `y` is then decorated with an attribute that does not exist. We will solve this problem by disallowing forward references to attributes.

We also discussed attributes decorated with the `compile` attribute. The values `true` and `false` can be treated as special namespaces, whilst compile-time constant expressions are compiled on demand. Declarations that end up being ignored, such as `false const bar` do not hide names from the enclosing scope. Instead they are placed in an 'un-nameable' namespace such that they can never be used. The `compile` attribute is no longer needed and can be removed.

Definition conflicts

Ordinarily, it is an error to define a name that already exists in a class hierarchy, for example:

```
class Base
{
    public var x
}

class Derived extends Base
{
    private var x // Error, x is defined in the base class
}
```

John points out that unifying namespaces with visibility modifiers is problematic because they are still used inconsistently in certain situations, for example `private`. In the above case it would be valid to have `x` declared twice in two user-defined namespaces (eg, `version1` and `version2`) but it is not valid to have it declared in the "special" namespaces `private` and `public`. This is inconsistent.

Overrides with namespaces

When a derived class overrides a method in a base class that is in namespaces `v1`, `v2`, etc. then as long as the override shares at least one namespace in common with the base class (eg, `v1`) then the method is overridden in all namespaces, including new ones added after the derived class was compiled. For example:

```
class A
{
    v1 v2 v3 function foo() { /* ... */ }
}

class B extends A
{
    // Although only v1 is mentioned, v2 and v3 are also overridden
    override v1 function foo() { /* ... */ }
}

class C extends A
{
    // C.foo is accessible from v1, v2, v3, baz, and bar namespaces
    override v1 baz bar function foo() { /* ... */ }
}
```

Two namespaces that are in use cannot contain the same name, even if that name is never called. Also, it is not legal to chain visibility modifies such as `public private function foo()`.