Ecma/TC39/2010/036

Simple Modules

Goals

The primary purpose of this spec is to provide better modularity mechanisms for packaging, deploying, and sharing ECMAScript code.

Goals for language:	**Goals for hosts:**	**Goals that are out of scope:**	**Non-goals:**
Eliminate global object	Zero-admin provisioning	Security mechanisms and idioms	Primitives to enable experimentation with module systems
Orthogonality with existing features	Uniform location and retrieval	State isolation	Fully first-class modules (e.g., module expressions)
Compatibility with non-module code		Service isolation mechanisms	Multiple layers of abstraction
Moving ES to true lexical scope		Multiple instantiations	Definition by reflective evaluation, esp. via string concatenation
Simplicity and usability		Dynamic upgrade	Definition by implementation
Smooth refactoring path from non-module code			
Preserve web UX of online execution, interactive perf			
Standard mechanism for code sharing			
Definition by semantic specification			

Examples

See the simple modules examples page for some highlights.

Overview

This proposal describes an incompatible semantics, in the sense that there are ES5 programs that cannot be used unchanged with the new semantics. For compatibility, some sort of versioning is required to indicate that code is evaluated with the new semantics.

The semantics implicitly enables ES5 strict mode and lexical scope mode. In particular, `eval` cannot modify its container's scope, `with` is disallowed, and there is no global object on the scope chain. The semantics ensures that there are never any object environment records on the scope chain except in using `eval` to evaluate legacy code.

The host environment may provide extra-linguistic means, such as compilation switches or language dialects (again, see versioning) to import certain standard libraries by default.

Terminology

- *Application*: A sequence of scripts. (On the web, this corresponds to all the `script` tags on a web page, in the order they appear.)

- *Script*: A source code unit consisting of declarations (`module`, `var`, `function`, etc.) and statements. (On the web, this corresponds to a single `script` tag.)
- *Module*: A module is a source code artifact, i.e. the `module M { ... }` construct.
- *Module instance*: An instance of a module is an evaluated module; it is an internal construct (like scope chains or activation objects), not a first-class value.
- *Module instance object*: A module instance object is a first-class object that reflects the bindings of a module instance.
- *Module resource locator (MRL)*: A string that maps, in a host-dependent manner, to a module source text.
- *Module binding*: A binding in a scope chain record that maps to a module.
- *Declarative module binding*: A module binding in a declarative environment record.
- *Global environment record*: An environment record at the top of a scope chain, which dynamically reflects the

Syntax

```
ExportDeclaration ::= 'export' VariableStatement
                   |  'export' FunctionDeclaration

ImportDeclaration ::= 'import' ModuleSpecifier '.' '*' ';'
                   |  'import' ModuleSpecifier '.' ImportSpecifier ';'

ModuleSpecifier ::= Identifier
                  | ModuleSpecifier '.' Identifier

ImportSpecifier ::= '{' ImportSpecifierItem*(',') '}'

ImportSpecifierItem ::= Identifier (':' Identifier)?


ModuleDeclaration ::= 'module' Identifier '=' 'load' MRL+(',') ';'
                   |  'module' Identifier '=' ModuleSpecifier ';'
                   |  'module' Identifier '{' ModuleElement* '}'

MRL ::= StringLiteral
     |  '(' MRL ')'
     |  MRL '||' MRL

ModuleElement ::= Statement
               |  FunctionDeclaration
               |  ImportDeclaration
               |  ExportDeclaration
               |  ModuleDeclaration

Script ::= ScriptElement*

ScriptElement ::= Statement
               |  FunctionDeclaration
               |  ImportDeclaration
               |  ModuleDeclaration
```

Top level

The initial scope chain of an application consists of a global environment record, which contains only module bindings. The global environment record can acquire new entries at runtime, since the module loader may dynamically register new global modules. All subsequent frames in the scope chain are declarative environment records; the global environment record is effectively the only dynamically changing record.

The global object is essentially gone. It is only available via an accessor in the current module loader, or through evaluation of legacy code via `eval`.

A program is a sequence of scripts, evaluated from top to bottom. Each script is evaluated in a nested declarative environment record. **Note:** This means that subsequent scripts may refer to bindings in previous scripts, but not vice versa. This precludes mutual recursion between multiple scripts, but this means that scripts can be incrementally evaluated, each separately compiled.

At the top level of an individual script, variable and function declarations are hoisted to that script's declarative environment record.

After a script is compiled, and before it begins evaluation, all of the script's top-level modules are attached to the current module loader, and consequently accessible via the global environment record. This allows dynamic access to all top-level modules in all scripts. (Note that this does not make scripts mutually recursive, since these bindings are only attached after compilation.)

Lexical scope

In the initial scope of a module, the only bindings in scope are the declarative module bindings in its scope chain. The current module loader determines what modules, if any, are in the global scope by default.

In the initial scope of a module, the binding of `this` is to the instance object of that module.

Performance note: to avoid unnecessary space consumption, closures should not capture any non-module bindings from the portion of the scope chain outside the current module.

Compilation

The only dynamic portion of the scope chain is the global environment record. Each script is compiled in the context of a current state of the global environment record. If a script contains any free variables, a compile-time error is reported and the script is not evaluated.

Module binding and resolution

Modules can be lexically nested, but are static declaration forms, not expression forms.

Modules are bound in the same lexical environment as variables and functions, i.e., in the scope chain.

It is a static error to declare a binding with the same name as a module binding at the same level of scope, e.g.:

```
{
    module m { ... }
    function m() { ... } // error
}
```

Module name resolution is simply lookup in the scope chain.

Each static module has a mapping from its nested module declarations to their corresponding static modules.

Dotted module name resolution selects nested modules, which involves a statically computable lookup.

Compatibility note: object environment records never contain bindings to static modules. If an object environment record happens to contain a module property (e.g., because a module object flowed into legacy non-strict code), it contains no static information. Note that this is harmless since non-strict code cannot do static imports.

Pragmatics note: nested modules make the “reverse DNS” naming convention (as used in Java) unnecessary; in fact, it doesn’t even work, since you can’t create an open, extensible module called e.g. `org` or `com`. Library writers can simply give their library a single identifier name such as `jQuery`. If clients need two different modules with the same name, they can disambiguate by placing them in different parent modules.

Import and export

Imports can only appear at the outermost scope of a module or script body. Their bindings are in scope for their entire containing block.

It is a static error for two import declarations or an import declaration and another kind of declaration to attempt to bind the same identifier in the same scope.

Import declarations

Imported bindings may not be mutated within the module. Because all module exports are write-once, it is possible for a module to import a binding that has not yet been set (which results in an exception). However, once an imported binding has been initialized, its value will never change.

- `ImportDeclaration ::= ‘import’ ModuleSpecifier ‘.’ ‘*’ ‘;’`

Binds all the exported names from the specified module in the current scope.

- `ImportDeclaration ::= ‘import’ ModuleSpecifier ‘.’ ImportSpecifier ‘;’`

Binds the identifiers specified in the import specifier, in the current scope, to the corresponding exports from the specified module.

- `ModuleSpecifierItem ::= Identifier ‘:’ Identifier`

Binds the export specified by the first identifier locally with the name specified by the second identifier.

- `ModuleSpecifierItem ::= Identifier`

Binds the specified export.

Export declarations

- `ExportDeclaration ::= ‘export’ (VariableStatement | FunctionDeclaration)`

Exports the binding(s) from this statement. The bindings are `const`, i.e., write-once.

Dependencies

A module may refer to or import from (and thus depend on) any other module in scope. This allows cyclic dependencies between any two modules in a single scope (such as the top-level scope) by virtue of being in scope of one another.

Module instance objects

Module instance objects are a runtime reflection of a module instance as a first-class object. Instance objects are *almost* frozen, except that their properties may internally change from undefined to defined during the initial evaluation of their corresponding module instance, when it initially assigns to its exports. But their properties cannot be explicitly updated in importing code, they cannot receive new properties, etc. If an own-property is read before its corresponding export has been defined, an exception is thrown.

A module instance object has a prototype chain, which ultimately leads to the `Object` prototype (possibly going through some prototype object with standard methods for module instance objects).

The own-properties of a module instance object are precisely its exports (including nested modules).

Determining whether a module instance object has a particular export is possible via `hasOwnProperty`.

Evaluation

A module body is evaluated in textual order (from top to bottom).

If a module export is dereferenced before it is initialized, an exception is thrown.

Module loading

The current module loader determines how to interpret MRL’s. The disjunction syntax:

```
MRL || MRL
```

informs the module loader to try each MRL in turn until one can successfully retrieve its module source.

Dynamic module loading

Dynamic module loading is possible via module loaders.

Compatibility

Host environments may provide a binding through the global object to give legacy scripts convenient dynamic access to the current module loader. On the web, this would enable patterns such as:

```
<button onclick="modules.MyApp.handleClick(this)">Click Me</button>
```

Enhancements

- Private modules
  - modules should be automatically public
  - may wish to allow creating internally private modules
  - private names proposal may be applicable

- packages
  - like modules but open and extensible
  - can only contain other packages or modules
  - can only be contained within packages
  - inline: `package p { ... }`
  - initially empty: `package p;`
  - modules can be declared with '.'-separated names; the portion before the final '.' must be a package (created if doesn't exist yet)
  - enables the reverse-DNS pattern
  - establishes a conceptual space for things like offline-JS directories, CLASSPATH, etc

- import "all except"
- automatic module-wrapping conveniences for upgrading legacy code, e.g.:

```
<script type="text/javascript?version=1.5" module="jQuery" bindings="this.jQuery" />
```

strawman/simple_modules.txt · Last modified: 2010/05/14 15:51 by dherman