Modules for JavaScript

Simple, Compilable, and Dynamic Libraries on the Web

David Herman
Mozilla Research
dherman@mozilla.com

Sam Tobin-Hochstadt *
Northeastern University
samth@ccs.neu.edu

Abstract

Building reusable libraries and reliable, maintainable programs requires modular design, yet JavaScript currently provides little support for modularity. In this paper, we present the design of a module system for JavaScript. Our design is currently the basis for the module system in the next version of the JavaScript standard.

The design provides a simple model for programmers and supports developing both client-side applications in the browser as well as standalone, server-side applications. Modules in our system are lexically scoped, may be nested and recursive, and can be statically loaded from external sources, allowing existing programs to be refactored naturally. In addition to static modules, our design provides a flexible mechanism for dynamically loading code that maintains isolation from untrusted modules. Finally, the system supports programmatic transformation and validation of code, supporting emerging practices in the JavaScript community.

1. Introduction

Today, JavaScript is used in all manner of settings, from tiny web page animations to huge applications with hundreds of thousands of lines of code and hundreds of millions of users. New libraries appear all the time, and modern applications make use of a wide variety of frameworks. Yet all of this development takes place in a language that provides next to no support for modularity, a key tool for building reusable and reliable software: JavaScript has no module system.

In this paper, we present the design of a module system for JavaScript. Our design is driven by our support of three key features: simplicity, to make it easy for programmers to adopt; static scope, to support compilation and reasoning about programs, and dynamic loading, to support execution of new modules in controlled environments. To make our system *simple*, we avoid restrictions on how modules can be composed: modules may be nested inside other modules, be mutually recursive, and be easily loaded from external files. We do not require complex metadata such signatures, headers, or linking specifications. To make our system *compilable*, we provide static scope with compile-time errors for unbound variables even in the presence of remote code loading. To support *dynamic loading*, we provide an expressive system of module loaders, with control over the sharing of resources and privileges, validation for security, and support for languages such as Caja and CoffeeScript that compile to JavaScript.

The primary use case for JavaScript programs is currently client-side dynamic web applications, hosted in web browsers. Accordingly, our design focuses on this use case, from separating file names from module names to supporting isolation for creating mashups. However, JavaScript is rapidly growing in other environments, and we have avoided baking web-only assumptions into our design.

The state of JavaScript modularity Of course, modern JavaScript programmers do not entirely forego modularity. Instead, they use existing language mechanisms to build abstractions that mimic module systems. The primary tools are closures and objects, often combined in what is known as the *module pattern*. Using an example from node.js [Dahl 2011], the essence of the pattern is:

```
var Server = (function() {
   var handler = function(req, res) { ... };
   return {
      go: function() {
         http.createServer(handler).listen(80);
         console.log("Server running!");
      }
   };
})();
```

This pattern uses objects, here with the property `go`, to represent modules, and creates a closure to encapsulate private data such as the `handler` variable. Additionally, standard practice is to have only one global name for a module (here `Server`), with all other parts of the library available as properties of that single globally-bound object. The `http` library is a global variable, presumably created in similar fashion.

While this pattern does provide namespace management and encapsulation, it falls short on several fronts. First, as a pattern rather than a language element, it requires effort on

* Supported by a gift from the Mozilla Foundation.

the part of the programmer to use correctly and to recognize when reading code. Second, although the pattern reduces pollution of the global environment, it does not eliminate it—every module adds a name to the top-level environment, and the module author, rather than the client, chooses the name. Third, the abstraction offered by the module pattern can be violated easily if private variables escape, or if the "exports" of the module are mutated. Fourth, compilers have a difficult time using this pattern to drive optimization, even when the programmer ensures that private state is hidden or that exports are immutable.

To alleviate these problems, a variety of JavaScript libraries and frameworks have been proposed to support modularity in JavaScript. In particular, CommonJS [Dangoor et al. 2009] provides a module system built on top of existing JavaScript constructs. However, the CommonJS module system and related designs are limited in several ways: they cannot extend the syntax of the language, they do not provide new scoping constructs, and they aim primarily at server-side environments. By proposing a revision to the language, we can lift these restrictions and provide a simpler, cleaner, and more expressive design.

The basic structure of our design is outlined in our version of the `Server` example:

```
module Server {
    module http = require "http";
    function handler(req, res) { ... };
    export function go() {
        http.createServer(handler).listen(80);
        console.log("Server running!");
    }
}
```

Now, `http` is a module instead of a global variable, exports and imports are clear and declarative, and the structure of the program is apparent to both programmers and tools.

To describe our system, we begin with an overview of the design space for JavaScript modules (§2) and an exploration of our design with additional examples (§3). We then describe the detailed semantics of both static modules (as seen here) and dynamic loading (§4–5). Finally, we discuss related work and conclude.

The status of our design This paper describes our work on Ecma TC39, the technical committee responsible for the specification of the JavaScript programming language, to design a module system for the next edition of the JavaScript standard. The system we present has been presented for inclusion in the next edition of the language standard,[1] and has received the consensus of the committee as the foundation of module system design for JavaScript.

We have developed an initial prototype implementation of our design in the Narcissus meta-circular JavaScript interpreter,[2] a testbed for JavaScript language design. Our prototype supports the use of local, scoped modules but not external modules and module loaders. Initial experience with the prototype is positive, but more work remains before it is ready for production use.

2. Design space

JavaScript poses many challenges and constraints for the design of a module system. Many of these challenges arise from unique characteristics of the web platform, JavaScript's most important use case. Here, we outline some of the key design considerations that inform our approach to a JavaScript module system.

2.1 Scripting convenience

As a scripting language, JavaScript owes much of its success to its accessibility and simplicity. A module system for a scripting language should impose a minimum of bureaucratic infrastructure on programs. In particular, the process of refactoring a global script into a reusable library should require very little decoration of the program with library metadata.

Moreover, to minimize the cost to developers of modularizing their scripts, a module system should allow them to divide up their program at arbitrary points. In particular, JavaScript should admit recursive modules. Module systems that disallow cyclic dependencies force programmers to make pervasive changes to the organization of their programs. The language should accommodate itself to the programmer's design, not the other way around.

2.2 Static scope in a dynamic environment

Modularizing a program involves grouping related definitions into logical units that can be shared and reused. Each module is essentially a collection of published definitions, whose exports form an API. Module imports and exports are therefore quite static in nature. Using this static information to provide compile-time binding and linking is valuable to programmers for catching early errors, and avoids making programs susceptible to dynamic variable capture based on a global namespace that may change at runtime.

At the same time, the web is a dynamic environment. Pages are loaded over the network, and scripts are processed by the browser and dynamically evaluated. The presence of `eval` in JavaScript and the ability to add new `script` elements dynamically to a page make it possible to load additional code at runtime.

Moreover, the nature of software distribution on the web mean that web application writers do not control the browser vendor and version used by the client, leading to platform inconsistencies. Because JavaScript programs are evaluated dynamically by the client's browser, applications must make

[1] The standardized language is referred to as ECMAScript.

[2] `https://github.com/mozilla/narcissus`

dynamic decisions based on the presence or absence of specific features and functionality.

Thus, a module system for JavaScript should provide static variable resolution and linking while also accommodating the dynamic usage patterns prevalent on and necessary for the web platform.

2.3 Naming on the web

The key to code reuse is to make as few assumptions as possible about the context in which clients will use a library. Modules help programmers achieve this goal by avoiding global variables and facilitating information hiding, but the system must still provide a mechanism for publishing modules and making them accessible to the rest of a program. This raises the question of module naming.

Clearly, at the scale of the web, creating a global registry for publishing modules would not only be difficult to design but completely impractical to make efficient. In the limit, such a registry would probably look like the Internet itself! Without any kind of registration mechanism, though, providing a unique name for a library relies on programmers following protocols such as Java's "reverse DNS" idiom, or user-hostile mechanisms such as globally unique identifiers. These approaches tend to result in inappropriate or even human-unreadable module names. Besides, a lightweight module system should make it easy to create a library without worrying about such issues.

2.4 Diverse delivery platforms

Another consideration in publishing modules is the question of retrieving modules from external resources. Traditionally, most languages or language implementations integrate into a local filesystem to find external modules. Sometimes this takes the form of explicit paths, and sometimes languages providing automatic mechanisms for finding modules in the filesystem based on their name. Some languages also use the file system canonicalize module references by file identity.

These approaches do not translate directly to the web. In general, web pages do not have access to a user's local filesystem. They do, however, need the ability to fetch modules from servers. JavaScript modules must therefore support loading from external URLs, and cannot depend on file system assumptions.

JavaScript is already capable of loading from external files using standard web APIs. Libraries such as RequireJS [Burke 2011] provide conveniences for retrieving JavaScript source files from remote servers and executing them via `eval`. But because the sequential control flow model of JavaScript demands non-blocking, asynchronous I/O, RequireJS is forced to employ callbacks to request the files. Painfully, this forces client programs to nest their contents within the body of the callback of the module request:

```
require(["a.js","b.js","c.js"],
    function(a, b, c) {
        ...
    });
```

This style imposes a heavy syntactic cost on client code. A module system for JavaScript should allow for simple, declarative loading of external files, but it must achieve this without introducing blocking I/O into the main event-processing loop of a web page.

Web applications are not the only consumers of JavaScript, even if they are the biggest. JavaScript is used as an embedded scripting languages in a number of environments such as the Java standard library [**?**]. Recently, JavaScript has rapidly gained popularity as a programming language for server programming, especially with the emergence of the node.js [Dahl 2011] server platform. On these platforms, modules should have access to the conventional filesystem as with traditional languages. Even on the web, programmers may wish to circumvent the usual URL schemes to use content delivery networks or custom library repositories. All of these needs call for flexible mechanisms that support a diversity of semantics for locating external modules.

2.5 Sharing and isolation

JavaScript programs operate on highly mutable data, including its standard libraries. Web applications routinely take advantage of this mutability to patch existing objects with new functionality. So in order to migrate existing applications to modules, as well as to allow modules to communicate usefully with one another, they must be able to share state.

At the same time, this pervasive mutability can be problematic. Web applications that combine separately developed and possibly mutually untrusted code often need to execute separate components in isolated contexts, to prevent unintended or malicious mutation. For example, security frameworks such as Google's Caja [Caja Team 2011] freeze standard objects to close off exploitable communication channels. Another example where isolation is desirable is in online integrated development environments (IDE's) such as ACE [Ajax.org 2007] or CodeMirror [Haverbeke 2007]. When a user executes a program they are editing, the executed code should not be able to affect the state of the IDE implementation.

Thus a module system for JavaScript should provide programmers with fine-grained control over sharing and isolation.

2.6 Extensible compilation

A final desideratum for a module system is the ability to tap into the loading process to execute custom compilation hooks. Modern production web applications typically employ build processes that manipulate JavaScript code in a number of ways:

- running style checkers such as JSLint [Crockford 2002] or JSHint [Kovalyov 2010];
- performing security analyses;

- running code optimizers such as Google's Closure compiler [Closure Tools Team 2009]; or
- compiling alternative source languages such as CoffeeScript [Ashkenas 2010].

Running build processes offline can be crucial for performance, but it introduces a roadblock in the development process. It also makes it difficult for developers of new languages, dialects, or code processors to acquire users, since it raises the barrier to entry for users who are accustomed to JavaScript's lightweight edit-reload development process. Letting programs hook into the compilation process would make it possible to deploy code processors as libraries, making it easy to explore and experiment with the tools.

3. Design overview

We begin by presenting a sequence of examples to illustrate the fundamental design of our system as it is seen by JavaScript programmers.

3.1 Importing and exporting

The simplest use of modules is as a collection of definitions, some of which are `exported`. The following `Math` module exports two bindings, `sum` and `pi`. The `three` binding is not exported, and therefore not accessible from outside the `Math` module.

```
module Math {
    export function sum(x, y) {
        return x + y;
    }
    export var pi = 3.141593;
    var three = 3;
}
```

To use the `Math` module, a client can import some or all of the exported bindings.

```
module Client {
    import Math.sum;
    sum(4,5);
}
```

Imports are possible at the top level of JavaScript programs, as well:

```
import Math.{sum, pi};
alert("2pi = " + sum(pi, pi));
```

All bindings in these examples are statically resolved; attempting to import `three` or reference a non-existent variable is a *compile-time* error, unlike in existing JavaScript programs where unbound top-level variables evaluate to the `undefined` value, or can be dynamically given a value via assignment before they are referenced.

Programmers can also import all bindings from a module, using `import Math.*`, and rename bindings on import, as in `import Math.{sum: plus}`, binding the name `plus`.

Modules can be nested within other modules, and exported just as with other bindings.

```
module Math {
    export module Arith {
        export function sum(x,y) ...
    }
    export module Trig {
        export function cos(x) ...
    }
}
```

Subsequent modules can then `import Math.Arith.sum` to bind the `sum` function in their scope.

Additionally, modules can simply be referenced, producing a *module instance object*, which is a first-class reflection of the exported bindings of a module.

```
module A = Math.Arith;
A.sum(3,4);
```

Here, the program indexes directly into `Arith` as an object. Although this technically uses the `A` module instance object as a first-class value, the syntactic structure of the variable reference is apparent and can be easily optimized by the compiler, just as if the programmer had imported `sum` directly. Of course, `A` can also be used in truly first-class fashion—as an argument to functions, as an element of data structures, or in any other way programmers can use JavaScript objects.

Module instances can contain mutable state, as in this implementation of a counter:

```
module Counter {
    var counter = 0;
    export function inc() { return counter++; }
    export function cur() { return counter; }
}
```

3.2 Recursive modules

Module nesting provides a natural home for recursive modules. Any module in a single scope can refer to and import from any other module in that scope. For example, the classic mutually-recursive `Even` and `Odd` can be defined by:

```
module Numbers {
    export module Even {
        import Odd.odd;
        export function even(x) {
            return x == 0 || odd(x - 1);
        }
    }
    export module Odd {
        import Even.even;
        export function odd(x) {
            return x != 0 && even(x - 1);
        }
    }
```

```
}
```

To determine scoping, all exports of all modules in a scope are collected prior to compiling the bodies of the individual modules. Initialization of module bindings is performed in program order, from top to bottom.

3.3 Requiring external modules

Since the primary use case for JavaScript is currently deployment in client-side web applications, it is vital for programmers to load modules from external source, and even other web sites. The following example defines a module `JSON` based on the contents of a file loaded from the `json.org` site.

```
module JSON =
  require 'http://json.org/modules/json2.js';

alert(JSON.stringify({'hi': 'world'}));
```

One key element of the design evident here is that `json2.js` is not expected to supply the `JSON` module, but the *contents* of that module. This allows the `require`ing module to decide that name that will be used for the resulting module, and avoids name clashes without the need for complex naming conventions.

Of course, the external file can contain modules, which become *nested modules* when `required`, as in the following example:

```
module YUI =
  require 'http://yahoo.com/modules/yui3.js';

alert(YUI.dom.Color.toHex("blue"));
```

Here, the `yui3.js` file exports the `dom` module, which in turn has `Color`, another module, as an export.

3.4 Local and standard modules

Of course, not every module is loaded from a remote location over the web. JavaScript is increasingly used in server-side environments, where loading from the file system is the primary case, as in most other programming languages. Additionally, every instantiation of JavaScript provides a standard library, which modules must be able to `require`.

Our design supports both of these use cases. We indicated standard modules with the `@` symbol as a prefix—this ensures that no potentially valid URL overlaps with standard modules. Local modules can appear just as relative requires, without needing complex URL schemes. The following example demonstrates both of these features, assuming an environment where the standard library includes a `cmdline` binding, and a `Lexer` library is available.

```
module stdlib = require '@std';
module Lexer = require 'compiler/Lexer';

Lexer.scan(stdlib.cmdline[0])
```

3.5 Dynamic loading and evaluation

To reduce latency or make dynamic decisions, JavaScript code is often loaded dynamically during program execution. Our module system supports this using *module loaders*, which encapsulate module instances for reflective access. Module loaders can be used for secure encapsulation and many other purposes, but at their most basic, they allow dynamic loading of code, as in this example.

```
loader.load('http://json.org/json2.js',
    function(JSON) {
        alert(JSON.stringify([0, {a: 1}]));
    });
```

Here, in keeping with JavaScript's prohibition on blocking I/O, the `load` method takes a callback that is run once the `json2.js` code is fetched, compiled, and evaluated. The callback function is passed the reified module instance, which is simply an object with properties for each of the original module's exports.

Module loaders also support simply evaluating strings of code, as in

```
loader.eval("3 + 4")
```

which produces `7`, of course. Since `eval` is a synchronous method without a callback argument, the code provided to `eval` must not `require` any non-builtin modules; if this were allowed, then calls to `eval` would sometime block. For code that should be allowed to perform I/O, an asynchronous `evalAsync` method is provided.

3.6 Custom module loaders

Module loaders support more than just dynamic loading of JavaScript code in the manner of existing JavaScript systems. They also provide facilities to selectively share state with potentially-untrusted third parties, to reject calls to `load` or `eval` based on the source or URL of the code, and to transform programs from arbitrary other languages into JavaScript.

Isolating and sharing state A module loader encapsulates a mapping from module names to their instantiations. Since modules contain state, a module instantiation is a stateful object, and two module loaders which share an instantiation share the associated state. Given a loader, we can share an existing module with it:

```
module M { ... }
loader.defineModule("newM", M)
```

This example makes use of automatic reflection of `M` as a module instance. The use of `defineModule` makes sharing between module loaders explicit, and gives the programmer control over which state an inner module loader can see. In this case, `newM` is available to code evaluated in `loader`, but *not* `M`.

Since globally-available objects in JavaScript such as `Object` and `Array` are mutable, module loaders can also create new versions of these primitive objects to avoid creating unwanted channels of communication.

Checking and validating dynamic loads In addition to defining the scope and state available to dynamic loading and evaluation, module loaders can also interpose on that evaluation, potentially rejecting attempts to load from a particular site, or evaluate code that fails a required test.

For example, we can construct a module loader that rejects any attempt to use `XMLHttpRequest`, based on the static analysis of Guha et al. [2010]. Then this call to `eval` will be dynamically rejected:

```
loader.eval("new XMLHttpRequest()")
```

as will any attempt to load code that uses XHR, including via any further module loaders in the scope of `loader`.

Translating to JavaScript As the only language supported in all modern web browsers, JavaScript now sees wide use as a *target language*, with compilers written from CoffeeScript [Ashkenas 2010], Java [Google], Flapjax [Meyerovich et al. 2009], JavaScript itself [Closure Tools Team 2009], and many others. However, all of these translations currently take place ahead of time and outside the JavaScript toolchain, complicating deployment and development. With module loaders, we can define translation hooks, which can take arbitrary source code, translate it, and pass it on to their context for evaluation. For example, with a CoffeeScript module loader, we could call:

```
csLoader.load("mod.coffee", callback)
```

where `"mod.coffee"` is the file:

```
math =
  root:   Math.sqrt
  square: square
  cube:   (x) -> x * square x
```

The function `callback` is thus called with a module instance with the `math` export, and can use this just as if it were an ordinary JavaScript object. Translation via module loaders opens up the possibilities of new languages that can be written and deployed entirely as JavaScript libraries.

4. The core system

In this section, we present our design of the core module system. This includes module declarations with imports and exports, static scoping and linking, and evaluation. For now, we assume a single compilation and evaluation session with a single global environment. We revisit this assumption in Section 5, where we discuss dynamic loading and multiple global contexts.

4.1 Static scope

At its heart, JavaScript is a lexically scoped programming language. Unfortunately, the language also includes a few constructs that involve dynamic variable binding. These features are notorious for being difficult to use reliably and for degrading performance in modern optimizing JavaScript engines. They include:

- the `with` statement, which extends a lexical environment with a dynamically computed object;
- unprotected `eval`, which dynamically evaluates a string as a JavaScript program, and extends the local environment with the evaluated program's global variables; and
- the global object, which reflects the initial frame of the lexical environment as a mutable object that is exposed to JavaScript programs.

JavaScript implementations have recently begun to support a "strict" dialect, which disallows the use of `with` and protects calls to `eval` with a fresh environment record. However, a motivating goal of the revised standard is to support full static scoping, with compile-time checking of variable references and assignments. To that end, we make the additional restriction that JavaScript programs cannot dynamically remove global variables.

Modules play a central role in this new static semantics for JavaScript. Module definitions, as well as `import` and `export` declarations, are declarative constructs which are processed at compile-time. The complete hierarchy of declared modules in a JavaScript program is statically fixed, as is each module's set of exports.

4.2 Dynamic reflection

Modules are bound in the same environment as other variables. As such, they can easily be reflected as first-class *module instance objects* simply by referring to them in expressions:

```
module M {
    export var x = 1, y = 2, z = 3;
}
print(Object.keys(M)); // x, y, z
```

While any variable may be bound at runtime to a module instance object, only those variables bound via `module` carry the additional static information used for compile-time checking. We refer to such bindings as *static module bindings*. For example, consider the following declarations:

```
module M { ... }
import M.*;
```

The module `M` is declared via a `module` declaration and is thus a static module binding. This means that the `import` declaration below the declaration of `M` is legal. However, if we bind an ordinary variable to a module instance object, no such `import` is possible:

```
module M { ... }
var x = M;
import x.*; // compile-time error
```

Because `x` is assigned its value dynamically, its contents cannot be statically determined, so `import`ing from `x` would require dynamic scoping.

4.3 Resolution

In order to support static scoping, we introduce a *resolution* phase which precedes evaluation. The resolution process traverses a program with a lexical environment, checking for references or assignments to unbound variables and preventing evaluation from occurring if any are found. Resolution occurs in the context of a *global namespace*, which indicates the current set of global variables and is fixed at compilation time.

The scoping and linking semantics of modules is carefully designed to avoid complicated validation algorithms, even in the presence of recursive modules. This helps maintain fast compilation and also rules out certain problematic or ambiguous programs. The design relies on the following key restrictions:

1. It is illegal to export a module that is not defined locally.
2. The `export` construct does not allow wildcards for exporting sets of bindings (e.g., `export M.*;`).
3. Static module bindings can only be bound with the `module` form.

Restriction (1) ensures that module paths are acyclic and can be evaluated via a simple structural recursion. Restriction (2) rules out complex cyclic relationships in module export sets, including ambiguous declarations such as:

```
module M {
    export N.*;
}
module N {
    export M.*;
}
```

Restriction (3) deserves a closer look. Consider a module `M` that exports a child module `N`. A client can create a local module binding via a `module` declaration, from which the client can import subsequent bindings `x`, `y`, and `z`:

```
module N = M.N;
import N.{ x, y, z };
```

Attempting to bind `N` via `import` produces a compile-time error:

```
import M.N; // error
```

Similarly, the `import M.*;` convenience form only imports the variable exports of `M`, excluding any of its exported submodules.

Restriction (3) simplifies the resolution process, making it possible at each scope to process all `module` declarations before `import` declarations. This avoids tricky mutual dependencies between the module paths in `import` declarations and the modules that they bind, which could lead to paradoxical declarations such as:

```
module M {
    export module N {
        export module M { ... }
    }
}
import M.N;
import N.M;
```

The tree of module definitions is discovered at parse time, at which point the set of declared exports for each module is known. The process of statically resolving a module leaves it in a partially linked state:

1. Each `module` declaration is bound statically to its source definition. This can be fully resolved at this stage, and must be validated to reject cyclic definitions.
2. Each `import` declaration such as `import M.{ x: y };` is processed by locally binding `y` to the declared export `x` of the module referred to by `M`.
3. Each `export` declaration such as `export { x: N.y };` is partially linked by pointing the export `x` to the declared export `y` of the module referred to by `N`. If no such export of module `N` exists, the program is rejected.

4.4 Linking

After resolution completes, modules are fully linked by chasing export links down to their ultimate source definitions. This process must check for and reject cyclic or invalid exports.

4.5 Evaluation

Evaluation of modules proceeds top-down, like other block forms in JavaScript. At the start of evaluation, all module instance objects are pre-allocated, but their exports are uninitialized. At the top level of a module, the special JavaScript variable `this` is bound to that module's instance object. Analogous to the use of `this` in object constructors, the body of a module to store references to the module instance object.

4.6 External modules

Modules can be loaded from external resources via a URL:

```
module M = require 'http://example.com/m.js';
```

The module loader in charge of compiling the declaration determines how to load the source file and whether it has already been loaded (see Section 5). External modules that are `required` multiple times are only evaluated at their first evaluated `require` declaration.

The contents of an external module includes the *body* of the module, but not the module declaration itself. Crucially, this allows clients of a third-party library to decide for themselves what local name to provide for the library. This obviates the need for a global registry of names, or inconvenient and brittle naming practices such as Java's "reverse DNS" convention.

External modules are compiled and evaluated in an initial scope that includes only the global namespace.

5. Module loaders

This section describes the module loader API, which provides powerful facilities for dynamic code evaluation, compilation hooks, and code and state isolation (sandboxing).

In Section 4, we described the behavior of a single compilation and evaluation session. Module loaders generalize this behavior: compilation and evaluation are always performed in the context of a specific module loader, and a program may create multiple loaders. Module loaders encapsulate several pieces of information that affect compilation and evaluation behavior:

- a *global namespace*, mapping global variable names to values;
- a *module instance cache*, storing module instance objects for externally `required` modules; and
- evaluation and loading *hooks*, which provide custom behavior for compilation.

All compiled code is permanently associated with the loader with which it was compiled. This association is used to override the behavior of the JavaScript `eval` operator[3], as well as the `require` form.

5.1 The system module loader

A JavaScript host environment must provide a built-in *system module loader*, which provides system-specific loading and compilation behavior. In the context of a web browser, this loader is reflected as an object and made accessible to programs via a global `ModuleLoader` variable, which satisfies the API described in this section.

5.2 Managing global namespaces

Every module loader encapsulates a global namespace, which is a mapping from variable names to values. Programs compiled and evaluated by the loader use this global namespace to interpret global variables (see Section 4).

The global namespace can be dynamically extended with new bindings. However, such bindings have no effect on code that has already been compiled, since any references to those non-existent bindings would have been rejected during resolution. Newly added bindings *do* affect code that is subsequently compiled by the loader.

Module loaders expose the following methods for extending their global namespace.

```
loader.defineGlobal(name, val)
```

This method takes a variable name and a value and creates or updates a binding in the global namespace.

```
loader.defineModule(name, mod, key)
```

This method takes a module name and a module instance object and creates a module binding in the global namespace, throwing an exception if there is already a binding of the given name. The optional `key` argument specifies a key for the module instance cache.

5.3 Dynamic evaluation

One of the most important use cases for module loaders is dynamic loading and evaluation.

```
loader.load(url, callback, onerror)
```

This method takes a URL and two functions and initiates a request to load the module at the given URL asynchronously. The source is loaded from the given URL, compiled and evaluated by the module loader. If compilation and evaluation succeed, the `callback` function is called with the module instance object as its argument. Otherwise, the `onerror` callback is called with the error as its argument.

```
loader.evalAsync(src, callback, onerror)
```

This method takes a source string and two functions and initiates a request to evaluate the source. Because the source may contain `require` directives, evaluating the source may involve I/O, so the interface is asynchronous. The source is asynchronously compiled and evaluated by the module loader. If compilation and evaluation succeed, the `callback` function is called with the result of the evaluated program. Otherwise, the `onerror` callback is called with the error as its argument.

```
loader.eval(src)
```

This method takes a source string and immediately compiles and evaluates the source program. The program is disallowed from containing any `require` directives, so that compilation cannot perform I/O. If compilation and evaluation succeed, the result of the evaluated program is returned. Otherwise an exception is raised.

5.4 Creating custom loaders

The module loader API also allows the creation of custom loaders, which can override the default system loading and compilation behavior.

```
loader.create(resolver, base)
```

[3] Technically, JavaScript's `eval` is not an operator but a function. However, the rules that determine when a call to `eval` can be statically detected and given access to the lexical environment cause it to behave much like an operator.

This method returns a new *child module loader* of `loader`. This parent-child relationship creates a chain of responsibility for loading. Both for loading external modules and dynamically evaluating source code, a child loader is given the opportunity to transform the source, but the result is then implicitly passed to the parent for subsequent processing. This ensures that child loaders cannot subvert any invariants enforced by parent loaders, and also allows for the composition of multiple levels of language abstraction.

The optional `resolver` argument is expected to be an object and provides the hooks for customizing the loading and compilation semantics of the loader. The optional `base` argument is a "base library," which we return to below.

5.5 Resolver objects

A resolver object can contain any of the following optional properties.

```
resolver.load(url, accept, reject, redirect)
```

This hook is called by the loader when an external module is required by `loader.load()` or by the compilation of a `require` directive. The `url` parameter is a canonicalized URL string. The `accept` callback accepts a string, allowing the hook to perform transformations on the source before passing it on to the parent loader. The `reject` callback accepts an optional error value, allowing the hook to signal a loading or compilation error. The `redirect` callback accepts a URL string, allowing the hook to forward the request to an alternate URL.

```
resolver.eval(src, accept, reject)
```

This hook is called by the loader when source code is evaluated by `loader.evalAsync()`, `loader.eval()`, or a use of the `eval` operator. The `src` parameter is the source program text. The `accept` callback accepts a string, again allowing the hook to transform the source language. The `reject` callback accepts an optional error value, allowing the hook to signal a compilation error.

```
resolver.resolve(url)
```

This hook allows custom loaders greater control over the canonicalization of module URL's. The hook is called by the loader before loading a URL; the hook returns an arbitrary value representing this URL's key into the loader's internal module instance cache. This makes it possible to resolve distinct URL's to the same key, effectively normalizing them to refer to the same shared module instance.

5.6 State isolation and base libraries

Module loaders facilitate state isolation by virtue of their separate global namespaces. Code that creates a module loader can selectively decide what bindings to share with the created loader via its `defineGlobal()` and `defineModule()` methods, as well as its `eval()` method.

The standard JavaScript library provides an extremely dynamic and mutable inheritance hierarchy for standard object types such as `Object`, `String`, and `Array`. Each of these types provides a shared *prototype* object. In practice, much of the code of the web takes advantage of the mutability of these prototypes in order to create new functionality that can be shared throughout a program. This means it would not be practical or even desirable to freeze the entire standard library.

In order to provide full isolation, then, it is necessary to be able to clone the original base library when creating a new module loader. However, in order for module loaders to communicate effectively with one another, it must also be possible to share a common base library between loaders. The `loader.create()` method therefore takes an optional `base` argument, which is an opaque object encapsulating the JavaScript standard library. Programmers can either obtain the base library of an existing loader:

```
loader.getBase()
```

or create a fresh base library:

```
loader.createBase()
```

This gives programmers fine-grained control over the sharing of state between loaders.

6. Related work

The literature on modules and modularity in programming languages is far too vast to summarize here. Below, we discuss the most closely related systems and those that have inspired our design.

Java Java provides dynamic loading controlled by class loaders [Liang and Bracha 1998]. Class loaders are both responsible for maintaining a mapping between class names and their implementations in code, and for loading new code dynamically when classes are requested. Additionally, class loaders form a hierarchy, with a class loader potentially delegating to its parent. The design of module loaders reproduces many of these design choices, in particular the coupling of loading with naming and the hierarchy. Of the four design goals set out by Liang and Bracha, type safety is not applicable, we do not provide lazy loading, and we accomplish user-defined extensibility and multiple communicating namespaces.

There are two key differences between module loaders and class loaders. First, sharing is handled differently. Class loaders share with their parent by delegating. Module loaders instead can share with any other module loader they can reference (likely their children), and share by explicitly giving access to some modules. This allows module loaders to create sub-loaders with fewer capabilities.

Second, module loaders can inspect, translate and potentially reject code loaded within them, even nested inside

multiple additional loaders. In contrast, class loaders can always directly define code, or delegate to the system loader.

Numerous modules systems have been proposed, specified, and implemented for Java [Corwin et al. 2003; JCP 2007; Jigsaw 2009; OSGi 2009; Strnisa 2008]. These systems tend to be far more complicated than ours, often involving deployment and packaging formats, metadata, versioning information, and linkage specifications. All of the systems are of course fully compilable. For dynamic loading, the systems rely on Java's class loader system described above.

ML The Standard ML module system [Milner et al. 1997], and related designs are widely influential and are one of the few designs that, like ours, supports arbitrary nesting of modules. Unlike our system, ML makes heavy use of explicit signatures, and like Scheme, supports richer static information, in particular types. Additionally, the ML module system is non-recursive and supports parameterized modules, not a goal of our system. Numerous proposals for adding recursive modules to SML exist [Dreyer and Rossberg 2008]; however they strive for expressiveness and static checking rather than simplicity. OCaml [Leroy 1997] supports a similar form of simple recursive modules, but requires predeclaration of signatures for all recursive modules.

ML module systems typically do not support dynamic loading, and treat the mapping of modules to files or other resources as outside the scope of the module system. Alice ML [Rossberg 2007] is an exception here, supporting static loading of code from remote locations as well as dynamically-invokable components and component managers, similar to our module loaders. However, these component managers apply only to the first-class component system in Alice ML, as opposed to the entire module system. Additionally, the system in Alice ML is significantly more involved to support the ML type system.

Scheme and Racket Module systems designs in the Scheme community, as in Chez Scheme [Waddell and Dybvig 1999] and Racket [Flatt 2002], typically resemble our design—first-order, untyped, with internal linking. The complexities of these systems comes from supporting macros, a far richer form of static information than is present in JavaScript, where modules only manage names. However, the need to support macros means that recursive modules are confined to separate systems, as in Racket's units [Findler and Flatt 1998; Flatt and Felleisen 1998], where macros live in signatures rather than modules [Owens and Flatt 2006]. Our design attempts to combine the simplicity of the first-order systems with the convenience of recursive modules.

The design of Racket also inspired two key aspects of our module loader design. First, the use of `defineModule` to explicitly share module instances between module loaders is taken from the design of Racket's namespaces [Flatt et al. 1999] where the operation is named `attach`. Second, the language-integrated support for compilers is inspired by a related feature in Racket's module system [Tobin-Hochstadt et al. 2011], which also supports the definition of entirely new languages.

Other dynamic languages Other dynamic languages such as Ruby [Matsumoto 2001] and Python [Van Rossum and Drake 2009] provide simple module systems. Typically, these systems consist of syntactic sugar over mutable hash tables, providing no encapsulation and limiting static reasoning about scope. These systems do not provide the sandboxing and isolation features of module loaders.

7. Conclusion

Today, JavaScript programs stretch to hundreds of thousands of lines and use numerous independently-developed libraries, yet JavaScript programmers must manage without a module system. In this paper, we present a module system that is simple for programmers, handles the complexities of deployment on the web, and supports flexible dynamic loading of code. We can also extend the capabilities of JavaScript to support state isolation and language-integrated translation and compilation from other languages to JavaScript. Our module system is under development for the next version of the JavaScript standard.

References

Ajax.org. Ajax.org Cloud9 Editor, 2007. `http://ace.ajax.org/`.

Jeremy Ashkenas. CoffeeScript, 2010. `http://coffeescript.org/`.

James Burke. RequireJS, 2011. `http://requirejs.org/`.

Caja Team. Google Caja, 2011.

Closure Tools Team. Google Closure compiler, 2009. `http://code.google.com/closure/compiler/`.

John Corwin, David F. Bacon, David Grove, and Chet Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 241–254, 2003.

Douglas Crockford. JSLint, 2002. `http://www.jslint.com/`.

Ryan Dahl. node.js, 2011. `http://nodejs.org`.

Kevin Dangoor et al. CommonJS, 2009. `http://commonjs.org/`.

Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 307–320, 2008.

Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, pages 94–104, 1998.

Matthew Flatt. Composable and compilable macros: you want it when? In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, ICFP '02, pages 72–83, 2002.

Matthew Flatt and Matthias Felleisen. Units: cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 236–248, 1998.

Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, pages 138–147, 1999.

Google. Google Web Toolkit. `http://code.google.com/webtoolkit/`.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 126–150, 2010.

Marijn Haverbeke. CodeMirror, 2007. `http://codemirror.net/`.

JCP. Improved modularity support in the Java™ programming language, 2007. JSR 294, `http://jcp.org/en/jsr/detail?id=294`.

Jigsaw. Project Jigsaw, 2009. `http://openjdk.java.net/projects/jigsaw/`.

Anton Kovalyov. JSHint, 2010. `http://jshint.com/`.

Xavier Leroy. *The Objective Caml system, Documentation and User's guide*, 1997. URL `http://caml.inria.fr/`.

Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 36–44, 1998.

Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly Media, 2001.

Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, 2009.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, May 1997.

OSGi. OSGi service platform core specification, 2009. `http://www.osgi.org/`.

Scott Owens and Matthew Flatt. From structures and functors to modules and units. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 87–98, 2006.

Andreas Rossberg. *Typed Open Programming—A higher-order, typed approach to dynamic modularity and distribution*. Phd thesis, Universität des Saarlandes, Saarbrücken, Germany, January 2007.

Rok Strnisa. Fixing the Java module system, in theory and in practice. In *Proceedings of FTfJP*, pages 88–99. Radboud University, July 2008.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation.*, PLDI '11, 2011.

Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, 2009.

Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 203–215, 1999.