problematic. Built-in constructors are typically defined using an implementation language such as C++. They allocate and initialize private object representations whose unique structure is known to the associated built-in methods which are also defined using the implementation language. This works when a built-in constructor is directly invoked using the `new` operator, but when "subclassing" such constructors using JavaScript's ad hoc prototype inheritance scheme, the `new` operator is applied to the subclass constructor (typically coded in JavaScript) which allocates an ordinary object rather than the private object representation expected by the inherited built-in methods. Wirfs-Brock [2013] attempted to avoid this problem when specifying the max-min class semantics. The semantics of `new` was split into separable allocation and initialization phases. Object allocation was performed by `new` first invoking a specially named `@@create` method that would typically be provided by a built-in superclass and not overridden by subclasses. Object initialization occurred after allocation and was orchestrated by the subclass constructor. It would typically make a `super` call to its superclass constructor to perform any necessary superclass-specific initialization and then perform any subclass-specific initialization. When properly coded, this enables a built-in superclass to allocate its unique private object structure before passing the object to the subclass constructor which can use its initialization code to add subclass properties to the superclass-provided object.

The problem that was identified in 2014 was that the objects created by `@@create` methods are uninitialized, and a buggy or malicious class constructor might invoke a built-in superclass method (likely implemented in C++) on an uninitialized object—probably with catastrophic results. Wirfs-Brock had assumed that all such objects would internally keep track of their initialization state and that the corresponding built-in methods would be required to check if they were being applied to an uninitialized object. Mozilla's Boris Zbarsky [2014] pointed out that browsers have thousands of such methods, and the two phase approach would require updating for each method the DOM specifications and implementations for every browser. This motivated development of a single phase allocation/initialization approach [Wirfs-Brock et al. 2014c,d] and another proposal [Herman and Katz 2014] which retained the two phases but passed the constructor arguments to both the `@@create` method and the constructor. These and other alternatives were hotly debated throughout the remainder of 2014, and for awhile it appeared that a lack of consensus might either delay the planned June 2015 publication of ES6 or force complete removal of classes from that edition. However, in January 2015 a TC39 consensus formed around a variation of the single phase approach [TC39 2015a; Wirfs-Brock 2015b]. This experience reïnforced TC39's resolve to require more and earlier implementor feedback on post-ES6 new features.

21.3.4 Modules. A complex aspect of the ES4 designs had been the package and namespace constructs for structuring large programs and libraries. By the time $ES4_2$ was abandoned, significant problems had been identified [Dyer 2008b; Stachowiak 2008b] with those mechanisms, and it was clear that they would not be suitable for Harmony. At that time, ad hoc modularity solutions based on the module pattern (§13.2) were being adopted by influential JavaScript developers [Miraglia 2007; Yahoo! Developer Network 2008]. In January 2009, Kris Kowal and Ihab Awad presented a module-pattern–inspired design [Awad and Kowal 2009; Kowal and Awad 2009a] to TC39 [2009c]. Their design eventually evolved to become the CommonJS module system used by Node.js.

Kris Kowal and and Ihab Awad, in their original proposal and a subsequent revision [Kowal 2009b; Kowal and Awad 2009b], included syntactic-sugaring alternatives that might overlay their module design without changing the proposal's dynamic semantics. Awad [2010a; 2010c] then developed a different proposal that drew from both the CommonJS work and the Emaker modules of the E Language [Miller et al. 2019] that were being used by the Secure ECMAScript Caja Project [2012]. Within TC39, these proposals were called "first-class module systems" because they manifest modules as dynamically constructed first-class runtime entities that provide a new computational

abstraction mechanism. For example, in Awad's proposal, multiple instances of a module may simultaneously exist, each initialized with different parameter values.

Brendan Eich [2009c] described an alternative approach:

> The alternative for Harmony is a syntactic special form, an import directive for example, that can be analyzed when the program is parsed (not executed), so the implementation can preload all dependencies before execution to avoid blocking on an import (or a later data dependency), or else an awkward non-blocking import to preserve JS's run-to-completion execution model.

This alternative approach was called a "static" or "second-class module" system. A second-class module system provides mechanisms for structuring application code rather than mechanisms for defining new computational abstractions. Sam Tobin-Hochstadt [2010] explained:

> . . . in a language with state, you want to be able to divide your program up into modules without changing its behavior. If you have a stateful piece of code, and you move it into its own module, that shouldn't change things any more than any other refactoring. If you need to repeatedly create fresh state, ES provides nice mechanisms for that as well. Similarly, if you have one module that imports A, and you divide it into two, both of which now import A, that refactoring shouldn't change how your program works.

Dave Herman and Sam Tobin-Hochstadt developed a "Simple Modules" design [Herman 2010b,c,f; Herman and Tobin-Hochstadt 2011; Tobin-Hochstadt and Herman 2010] for second-class Harmony modules. The basic idea was that modules were units of code that could share lexical bindings. Syntax was used to delimit the units of code and to identify which bindings would be shared. TC39 extensively debated the merits of the two approaches until Awad [2010b] recommended that TC39 focus its efforts on the Herman/Tobin-Hochstadt proposal.

Their design had `module` declarations that assigned a lexical identifier to the module and either included the module code or identified an external resource containing the code. An `export` keyword prefixed declarations whose binding were to be exposed outside of the module, for example:

```
module m1 {          // an internal module
  export var x = 0, y=0;
  export function f() {/* ... */};
}
module m2 {          // another internal module in same source file
  export const pi = 3.1415926;
}
module mx = load "http://example.com/js/x.js";
            // String literal identifies an external module
// ... code that imports and uses bindings from m1, m2, and mx.
```

Original Harmony Simple Modules Proposal

Module declarations could also be nested. An external module such as `x.js` could consist of only a module body without the surrounding module-declaration syntax. An `import` declaration is used to make a binding exported by a module lexically accessible to an importing module. Code that uses the above example modules might have `imports` like the following:

```
import m1.{x, f};  //import two exported bindings from m1
import m2.{pi: PI};//import a binding and rename it for local access
import mx.*;       //import all of the bindings exported by mx
import mx as X;    //Locally bind X to an object whose properties bind
                   //to the exports of mx
```

Original Harmony Simple Modules Proposal

Module declarations, string literal external module specifiers, and declarative export/import definitions enable static determination of a closed set of interdependent modules whose shared lexical bindings can be linked prior to the execution of any code. Circular dependencies are allowed. When execution starts, modules are initialized in a specified deterministic order and temporal dead zones ensure run-time errors if any circular dependencies are impossible to initialize.

The syntax evolved [Herman et al. 2013], but the basic idea of statically linkable modules with shared lexical bindings remained. A major change was the elimination of explicit module declaration syntax, module identifiers, and internal/nested modules. Harmony modules are defined one per source file and are identified using literal string resource specifiers. The elimination of module identifiers required changing the `import` syntax, and wildcard imports were eliminated as being too error prone. Wildcard imports were replaced with an alternative form that exposes an open-ended set of imports as properties of a single namespace object rather than as individual lexical bindings. The above `import` examples expressed using the final syntax look like this:

```
import {x, f} from "m1.js";     //import two exported bindings from m1
import {pi as PI} from "m2.js";//import a binding. Rename it for local access
import * as X from "mx.js";     //Locally bind X to a namespace object whose
                                //properties (*) map to the exports of mx.js
//additional import forms
import from "my.js";   //import my.js only for initialization effects
import z from "mz.js"; //import the single default binding exported by mz.js
```

ES2015

The elimination of `module` declarations and the addition of the default binding `import` form were a late change to the design. Node.js adoption had been unexpectedly rapid, and it had widely exposed CommonJS modules to the JavaScript developer community. TC39 was getting negative community feedback [Denicola 2014] and had concerns that de facto standardization of CommonJS modules might overshadow the Harmony design. The `export default` form was added to accommodate developers who were accustomed to the single export pattern[97] used in many CommonJS modules. TC39 module champions also began to evangelize [Katz 2014] Harmony modules to Node.js developers.

The initial Simple Modules proposal had included the concept of a module loader [Herman 2010e] which provided the semantics of incorporating modules into a running JavaScript program. The intent was that the ECMAScript specification would define the language level syntax and semantics of modules, the runtime semantics of module loading, and a module loader API which would provide JavaScript programmers with mechanisms to control and extend the loader semantics. The loading process was ultimately envisioned [Herman 2013b] to be a pipeline consisting of five stages: normalize, resolve, fetch, translate, and link. The loader begins by normalizing a module identifier. It then progresses through the retrieval and preprocessing of module source code, determining module interdependencies, linking imports to exports, and finally initializing the interdependent modules. The intent was that the module loader would be extremely flexible and fully support the asynchronous I/O model of Web browsers. At JSConf 2011, Dave Herman demonstrated [Leung 2011] a proof-of-concept module loader which extended the translation stage to load CoffeeScript and Scheme code as modules of a JavaScript-based Web page.

In order to fully understand the module loading process and how to specify it, Dave Herman worked with Jason Orendorff at Mozilla to prototype a module loader reference implementation [Orendorff and Herman 2014] using JavaScript code. In December 2013, Herman [2013a] completed an initial rewrite of Orendorff's JavaScript code into specification pseudocode, and in

[97] CommonJS modules typically export a single object that is used as a namespace.

January 2014 Allen Wirfs-Brock [2014a] had a preliminary integration of the pseudocode into the ES6 draft. Wirfs-Brock found that the asynchronous nature of the module loader added significant new complexity and potential nondeterminism to the ECMAScript specification. This was made worse by the loader API which allowed user programs to inject arbitrary JavaScript code into the module loading process. By the middle of 2014, the additional complexity of asynchronous module loading and a stream of difficult-to-resolve design issues with the API appeared to put the 2015 target release of ES6 in jeopardy.

Early in the development of the simple module proposal, Allen Wirfs-Brock [2010] had observed that the semantics of module scoping and linking could be separated from the loader pipeline. Previous editions of ECMA-262 had defined the syntax and semantics of JavaScript source code but did not address how it was accessed. That was left as a responsibility of the environments that hosted a JavaScript engine. At the September 2014 TC39 meeting [TC39 2014b], Wirfs-Brock argued that a similar approach could be used for modules. ECMA-262 did not need to include the specification of a module loading pipeline. If ECMA-262 assumed that the source code for modules was already available, it would be sufficient to specify the syntax and semantics of individual modules and the semantics of linking imported bindings to exported bindings. A host environment, such as a browser, could provide an asynchronous loading pipeline but its definition would be decoupled from the language specification. Removing the loader pipeline also implied the removal of the loader API. TC39 accepted this argument, and Wirfs-Brock was able to incorporate a nearly complete language-level specification of modules into the October 2014 specification draft [Wirfs-Brock et al. 2014b]. The separation of module semantics from the loader pipeline enabled the WHATWG to focus on specifying how ECMAScript modules would integrate with the Web platform [Denicola 2016].

21.3.5 Arrow Functions. ES2015 introduces a concise form of function definition expressions commonly called "arrow functions." An arrow function is written as a formal parameter list followed by the token => followed by a function body, for example:

```
(a, b) => {return a+b}
```

If there is only a single parameter, the parentheses may be omitted and if the body is a single `return` statement, the braces and the `return` keyword may be omitted, for example:

```
x => x /* an identity function */
```

Unlike other function definition forms, arrow functions do not rebind `this` and other implicit function-scoped bindings. This makes arrow functions convenient in situations where an inner function needs full access to the implicit bindings of its outer function.

The primary motivation for arrow functions was the frequent need to code verbose function expressions as call-back arguments to platform and library API functions. In JavaScript 1.8, Mozilla [2008b] had implemented[98] "expression closures" which retained use of the `function` keyword and permitted only a brace-free single expression body. TC39 discussed similar shorter formulations, which replaced `function` with symbols such as λ, *f*, \, or # [Eich 2010b; TC39 Harmony 2010c], but could not reach consensus on any of them.

There was also interest within TC39 [Herman 2008] in providing "lambda functions" with streamlined semantics such as support for *proper tail calls*[g] and Tennent's [1981] correspondence principle.[99] The proponents argued that such functions would be useful for implementing both language- and library-defined control abstractions. In an es-discuss post early in the Harmony

[98] Based upon an $ES4_2$ proposal [TC39 ES4 2006c]

[99] Within a function, wrapping a code sequence with another function that is immediately called should produce the same effect as directly executing the original code sequence.