Proposed ECMAScript 4th Edition – Language Overview

Revised 23 October 2007

The fourth edition of the ECMAScript language (ES4) represents a significant evolution of the third edition language (ES3), which Ecma approved as the standard ECMA-262 in 1999. ES4 is compatible with ES3 and adds important facilities for programming in the large (classes, interfaces, namespaces, packages, program units, optional type annotations, and optional static type checking and verification), evolutionary programming and scripting (structural types, duck typing, type definitions, and multimethods), data structure construction (parameterized types, getters and setters, and meta-level methods), control abstraction (proper tail calls, iterators, and generators), and introspection (type meta-objects and stack marks).

ES4 also upgrades ES3 in small ways by fixing bugs, improving support for regular expressions and Unicode, supplying richer libraries, and adding lightweight facilities like type-discriminating exception handlers, constant bindings, proper block scoping, destructuring binding and assignment, succinct function expressions and definitions, and array comprehensions.

We first present the goals of the ES4 working group (ECMA TC39-TG1) and discuss the key issue of compatibility between ES4 and programs written for ES3 before we describe ES4 in its entirety[1]. We omit tedious details. This document is not a tutorial; the reader should be familiar with ES3.

TG1 maintains a website at http://www.ecmascript.org.

[1] This document reflects the understanding of the language as of October 2007. TG1 is no longer accepting proposals and has started writing the language specification. The ES4 reference implementation is undergoing stabilization and is already passing large parts of its test suite, and this document describes the final language in most respects. That said, TG1 maintains a bug tracking system at http://bugs.ecmascript.org/, and ES4 is subject to further refinement as known and new bugs in the design are resolved.

The following licensing terms and conditions apply to this Language Overview document.

1. This Language Overview is made available to all interested persons on the same terms as Ecma makes available its standards and technical reports, as set forth at http://www.ecma-international.org/publications/.

2. All liability and responsibility for any use of this Language Overview rests with the user, and not with any of the parties who contribute to, or who own or hold any copyright in, this Language Overview.

3. THIS LANGUAGE OVERVIEW IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS LANGUAGE OVERVIEW, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

End of Terms and Conditions

Contents

I. Introduction

History ECMAScript 4th Edition (ES4) is a multi-paradigm programming language that is evolved from the 3rd Edition[2] (ES3), which in turn is based on JavaScript[3], a programming language for the web developed at Netscape Communications Corporation starting in 1995.

Work on ES4 commenced following the release of the 3rd Edition Specification in 1999. An interim report was released in 2003[4], after which work was halted. Subsets of the language defined by the interim report were then implemented by Adobe Systems (in ActionScript) and Microsoft (in JScript.NET).

The current Task Group 1 of Ecma Technical Committee 39 (Ecma TC39-TG1[5], hereinafter TG1) has met since the fall of 2005, basing their work on ES3, the interim ES4 report, and experiences with Adobe's and Microsoft's implementations. TG1 expects to complete the 4th Edition Specification in the fall of 2008.

This overview reflects the position of the majority in Ecma TC39-TG1. A minority in TG1 do not agree that the language presented here should become ES4. However, Ecma and ISO do not require unanimity to make standards, and no alternative proposal has been submitted to TG1. Therefore the language described in this overview continues to be proposed as ES4.

ES3 ES3 is a simple, highly dynamic, object-based language that takes its major ideas from the languages Self and Scheme. The programming style is a mixture of object-based and functional programming: The primary abstraction mechanisms in ES3 are *lexically scoped higher-order functions* and mutable *objects* whose *prototype objects* contain *methods* that are accessible through a *delegation* mechanism. ES3 has syntax reminiscent of C++ and Java™ and provides control structures like loops and exceptions; it also has a rich library of predefined data types—arrays, strings, numbers, dates, regular expressions, and more.

ES3 has been independently implemented several times and has found widespread use as the programming language for the web and as an extension language in desktop and other applications.

ES4 Though flexible and formally powerful, the abstraction facilities of ES3 are often inadequate in practice for the development of large software systems. ECMAScript programs are becoming larger and more complex with the adoption of Ajax programming on the web and the extensive use of ECMAScript as an extension and scripting language in applications. The development of large programs can benefit substantially from facilities like static type checking, name hiding, early binding and other optimization hooks, and direct support for object-oriented programming, all of which are absent from ES3.

Goals The goals of TG1 have been to make ES4 a language that is compatible with ES3 but more suitable for the development of large software systems; to facilitate the building of reusable ES4 libraries; to merge the strands of the language that were created when subsets of the 2003 interim report were independently implemented; to repair various shortcomings of the language; and to keep ECMAScript a pleasant language suitable for scripts and smaller programs.

II. Features at a Glance

OOP Object-oriented programming is supported in ES4 by *classes* and *interfaces* (jointly known as *nominal types*) as found in languages like Java. Class-based programming is widely taught and is a well-understood methodology for building programs and reusable libraries. In addition, the environment for ECMAScript programs on the web—the Document Object Model (DOM) of web browsers—is full of class- and

[2] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[3] http://developer.mozilla.org/en/docs/About_JavaScript
[4] http://www.mozilla.org/js/language/old-es4/
[5] http://www.ecma-international.org/memento/TC39-TG1.htm

interface-based data structures, while applications that use ECMAScript as the extension language in other environments often export object models that also are best realized as class-based structures.

The delegation style of programming used in ES3 is still fully supported in ES4, and in practice integrates well with class-based programming.

Finally, ES4 provides object-oriented programming in the style of Common Lisp, Dylan, and Cecil—where method definitions are decoupled from the definitions of the types they operate upon—through a multimethod mechanism. Multimethods also provide ES4 with a convenient hook for operator overloading.

Privacy Modularity, name hiding, and library construction are supported by *packages* and *namespaces*, which control access to names and—together with the type system—provide for robust APIs between separately implemented modules.

Types Programming in the large, verifiability in general, and performance are supported by a rich *type system*, where nominal types as defined by classes and interfaces are supplemented by a range of structural types that can describe the ad hoc data common in ES3 programs.

Variables can optionally be annotated with type constraints, and an implementation can perform static type checking and early binding. Critically, typed and untyped code can coexist in a predictable way, supporting multiple styles such as fully typed code; typed libraries and untyped client code; typed APIs (whether on the package, class, or function level) and untyped implementation code; and ad hoc use of type annotations as assertions.

Just as untyped code can become gradually typed as need dictates, ad hoc data can become structurally typed and nominally typed as need dictates: untyped data can be used for local one-off data structures; structural types can be applied when those data structures are exported; and the structural types can be changed to nominal types when performance, security, or program structure requires it.

Verification A program component can ask to be compiled in *strict mode*, which entails that the component is *verified* before it is executed. The verification pass performs static type checking and also verifies other static properties, notably: that all names that are referenced are known, that properties referenced on objects of known type are known to be present, that constants are not written to, and that comparisons do not mix types.

Verification in ES4 is modeled on static checkers and “lint” facilities in other languages, notably the verifier in ActionScript. A normative specification of verification facilitates interoperability among implementations to a greater degree than could be achieved without such a specification. (An implementation may choose not to support strict mode, and instead run every program in standard mode.)

Static type checking in strict mode is limited to checking that values of known types are not used in contexts requiring other types. Un-annotated variables and functions provide no information about the values they yield, and the implementation will inevitably check at run-time that those values are suitable for the operations that are applied to them (as is the case for all values in ES3 programs).

Speed and space optimization

Type annotations facilitate early binding from call sites to the methods that are called, potentially reducing the cost of method calls relative to ES3. Type annotations also allow a compiler to remove run-time type checks. Data types like `int`, `uint`, and `double` that map directly to modern hardware, coupled with type-specific operations on some data types, facilitate high-speed computation. Performance-oriented data types like `Vector` remove the overhead of very general data types like `Array`, at a slight cost in generality.

Programming in the small

Many small improvements and new features, including type-dispatched exception handling, block scoping, syntactic shorthands, and destructuring binding and assignment make the writing and reading of fragments of code simpler and more effortless.

Control Standard *iteration and itemization protocols* allow objects to expose iteration interfaces. The `for-in` and `for-each-in` statements make use of these protocols.

Generators provide a means of suspending control in a function to return a value to the caller, and letting the caller resume the computation later.

Proper tail calls allow recursive or mutually recursive functions to express iterative behavior. Tail calls make it possible to use procedural abstraction extensively without worrying about stack space usage.

Data ES4 provides parameterized classes, objects with getter and setter methods, typed object and array literals, meta-level hooks, reflection, destructuring, and type system facilities to make programmer-defined objects fit into the language as easily as predefined objects.

Self-hosting Though the ES3 definition does not state this fact, the predefined data types in ES3—`Object`, `Array`, and so on—are best realized as nominal data types: as classes and interfaces. Consequently, all the predefined types in ES4—from `Object` to `Map`—are described in the Standard as types in the language itself. So where *e.g.* `Array` was described as a collection of methods and properties in the 3rd Edition Specification, with prose and pseudocode providing the definition, `Array` is described as an ES4 class in the 4th Edition, and this description—using ES4 code to define the meaning of the methods precisely, necessarily augmented by descriptive prose—captures the full meaning of the data type.[6]

Self-hosting has been an explicit goal during the design of the language. Self-hosting has two positive effects: it significantly reduces the amount of magic required in the language definition, and it provides the users of the language with the same powerful facilities that the language itself requires to be specified, enabling the construction of a robust ecosystem of libraries and scripts for ES4.[7]

Portability ES4, like ES3 before it, is expected to see a variety of uses on a variety of machines.

Modern web browsers with full ES3 implementations run successfully on mobile phones with a few megabytes of memory and relatively slow hardware. Programs are transmitted as source text and all compilation takes place on the device, where short response times and interactivity are important. The full language must be supported—the web is not kind to subset implementations like the 3rd Edition Compact Profile[8]. Consequently, ES4 is implementable in its entirety on small devices, but at a cost: features that require sophisticated compile-time analyses are precluded in the language in general, not just in implementations on small devices. (For example, an implementation does not have to support strict mode.)

Likewise, some ES4 implementations are expected to exist in "hosted" environments, like the JVM or .NET. These environments do not always provide powerful control operators like first-class continuations or stack marks. Therefore ES4 is also precluded from incorporating such features.[9]

Formalism Above all else a good standard must be unambiguous, complete, and correct. The 3rd Edition Specification suffers to some extent from the use of an untestable, unspecified, low-level pseudocode to define much of

[6] A common fear when an implementation is used for specification is that the specification will be *too* specific—that implementation artifacts will constrain the specification too much. This has not been a problem for ES4 so far, for three reasons. First, the 3rd Edition Specification is more specific than most language standards. For example, ES3 defines order-of-evaluation in the libraries. The ES4 reference implementation simply captures such pre-existing specificity. Second, the implementation can be factored to avoid over-specificity. Third, the relatively high level of ES4 coupled with the self-referential nature of defining its own libraries requires the use of fewer implementation artifacts.

[7] Self-hosting may also benefit commercial ECMAScript implementations. Results from two companies suggest that implementing ECMAScript in itself, backed by advanced compiler and virtual machine technologies, leads to smaller and faster implementations.

[8] http://www.ecma-international.org/publications/standards/Ecma-327.htm

[9] ES4 has a stack-mark facility, but it is optional.

its semantics—the remaining semantics are described by prose. Though perhaps precise enough in practice, the pseudocode is hard to read and hard to write—numerous errors have been found, some of which are captured in unofficial errata.[10] Sometimes the prose does not describe everything it needs to describe, but inadequate descriptions are difficult to identify until one attempts to implement the specification.

Early on it became clear that ES4 would be a substantially larger language than ES3. TG1 members—certain that it would be far too hard to debug the resulting document—did not wish to continue to use the formalism from the 3rd Edition Specification. A better formalism was called for. E4X[11] has led the way with a structured pseudocode but it, too, is untestable. The 2003 interim report used an Algol-like pseudocode that was substantially more formal than notations used for ES3 or E4X, and an execution engine for it was even constructed. In its recent work on ES4, the TG1 members have investigated the use of term rewriting languages, including Stratego.[12] In the end, however, TG1 has decided to build a complete reference implementation (RI) of ES4 and all its predefined classes in high-level languages, and to extract parts of the 4th Edition Specification directly from the source code of the RI. As a bonus, the RI allows both TG1 members and the community at large to experiment with the language as it is being designed.[13]

The RI is written primarily in Standard ML (for the language processor) and ECMAScript (for the predefined classes). The RI is tested on test suites contributed by TG1 members to assess its degree of compatibility with programs written for ES3.

III. Compatibility

Compatible? The goal of TG1 is that ES4 will be backward compatible with ES3 and that ES4 implementations will be able to run ES3 programs correctly. This goal has been met, with a small number of qualifications.

Errors *Execution errors* are observable in ECMAScript. For example, if the program calls a non-existent method, references an undefined variable, or `eval`s a syntactically malformed string as a program, then a well-defined exception is thrown. The exception can be caught by the program and the program can come to depend on the exceptions. ES4 adds new methods and variables and makes more syntactic phrases legal, so ES3 programs cannot always count on receiving the same exceptions in an ES4 implementation.

ES3 explicitly allows new methods and variables to be added in conforming implementations, so this hazard is not new.

Syntax Some identifiers that were legal names in ES3 (`let`, `yield`, `cast`, `is`, and a few more) are *keywords* in ES4. Other keywords in ES4 were *future reserved words* in ES3, and correct ES3 programs do not use them (`class`, `interface`, `public`, `private`, and many others), though some implementations allow them to be used as names. Sometimes the new keywords are contextual and can continue to be used as names, but in general an ES4 implementation that must be able to process all ES3 programs must be told which dialect—ES3 or ES4—it is looking at, so that it knows whether to treat these identifiers as keywords or not.

The mechanism that supplies the dialect information will depend on the environment. In a web browser the information comes from the MIME type of the script or from a `version` parameter on the `SCRIPT` tag in the document.[14] New web pages that choose to use ES4 will have to specify the dialect.

Behavior Behavioral changes in ES4 are the result of specifying behavior more precisely than in ES3 (thus reducing the scope for variation among implementations), or of introducing true incompatibilities.

[10] http://www.mozilla.org/js/language/E262-3-errata.html
[11] http://www.ecma-international.org/publications/standards/Ecma-357.htm
[12] http://www.stratego-language.org/Stratego/WebHome
[13] http://www.ecmascript.org/download.php
[14] Script MIME types are defined by RFC 4329: http://www.ietf.org/rfc/rfc4329.txt.

TG1 has worked on the assumption that variation among ES3 implementations entails a license to specify behavior more precisely for ES4 where that seems important. For example, ES3 does not specify the order of enumeration in a `for-in` loop, but Netscape Navigator implemented enumeration order as insertion order and Internet Explorer followed Netscape's lead. Once the two leading implementations agreed, programmers started depending on the behavior, and TG1 has decided to require enumeration order to be insertion order.

TG1 has adopted the principle that a true incompatibility can be justified only if it is likely to fix many more programs than it breaks. Thus some incompatibilities are simple bug fixes, as when the `arguments` object is made to behave like a true `Array` or when the value of the `length` property of certain predefined functions has been changed to follow a general rule (where previously there were a few ad hoc variations among functions). Other incompatibilities are more subtle, as when the value of `this` is propagated unchanged to calls to lexically nested functions, or when the literal syntax for regular expressions results in a new `RegExp` object every time the literal is evaluated, instead of just once during the lifetime of the program.

Impact Is it possible to assess the likely impact of the changes in ES4 on current environments? The main concerns are: incompatibilities between ES4 and ES3; incompatibilities among implementations; and incompatibilities between ES4 and the environments that will use it (importantly, but not exclusively, the web).

The incompatibilities between ES4 and ES3 have already been discussed, and it is the opinion of TG1 that they are benign, though undoubtedly some programs somewhere will depend on syntax or behavior that has changed.[15] Again, the rule of thumb has been a belief that bug fixes are necessary and that breaking some programs can be accepted if there is a net benefit because many more programs start working.

Incompatibilities among implementations can be the result of a poor standards document, inadequate engineering practices, or simply different choices among vendors about whether standards compliance is more important than compatibility with existing ECMAScript implementations (some of which are not compatible with ES3). Of all these factors, TG1 has the most control over the quality of the Standard, and we are attempting to produce a high-quality specification by means discussed in the previous section.

The web The final concern, incompatibilities between ES4 and its environment, is primarily a question about whether ES4 is compatible with ES3 and whether ES4 implementations are compatible with each other. In part it is also a question about whether ES4 implementations can align themselves with each other in such a way that they can support the same existing and new content in a timely manner.

Historically, a new feature has become a part of the Standard when one vendor adds that feature to its dialect and other vendors follow with compatible implementations in their dialects. With ES4, the reference implementation and its test suite together fulfill this role of lead implementation, giving all vendors a head start for producing compatible dialects. The reference implementation creates the opportunity for vendors to coordinate feature development and compatibility testing before any one implementation is released. This kind of collaboration will potentially have a stabilizing effect that was not possible in earlier editions.

Note The following sections mark incompatibilities between ES3 and ES4 with a "**Note:**".

[15] It is hard to predict when scripts will break. It is possible to spider the web, looking for pages that rely on certain behavior either statically or dynamically. Many important pages can be reached this way—yet many cannot, and some behavior cannot easily be searched for in this manner.

IV. Values

Values There are three kinds of *run-time values* in ES4: *objects*, and two special values called **null** ("no object") and **undefined** ("no value").

There are also *compile-time values*, including types, namespaces, constants, variables, program units, and packages. These will be discussed later.

Classes A *class* describes an object by presenting those *properties* (fields) of the object that are always present (the *fixed properties* or *fixtures*), including variables, constants, and methods[16]:

```
class C {
    var val                              // a variable property
    var large = Infinity                 // a variable property
    const x = 3.14                       // a constant property
    function f(n) { return n+val*2 }     // a method property
}
```

Instances An object is created by *instantiating* a class. Any class can be instantiated several times and the objects that result are related to each other: they are of the same *type*, and that type is determined by the class.

new Instantiation of a class C is generally accomplished by a `new` expression:

```
new C
```

The expression creates a new instance of C and initializes it: properties are given their initial values if those are described in the class, and otherwise given the value **undefined**. If the class has a special *constructor* method then that method is called. The `new` expression can pass arguments to the constructor:

```
new C( 1, 2, 3 )
```

Properties Properties are *named*, have *attributes*, and hold values. Variable properties can be updated; constant and method properties cannot. Method properties always hold function values (behavior).

The property attributes are DontDelete (the property is fixed), DontEnum (the property is withheld from property enumeration), and ReadOnly (the property value is constant). Properties defined in the class are DontDelete and DontEnum. Properties added dynamically are neither. Constant properties and method properties are ReadOnly.

Dynamic properties

Instances of a class that is described as `dynamic` can hold properties that are not defined by their class, and these properties can be removed and added dynamically:

```
dynamic class C {
}
c = new C
c.x = 37   // adds a property
delete c.x // removes it again
```

Dynamic properties are deletable, enumerable, and writable (none of their attributes are set).

Whether a class is dynamic or not depends only on whether it is directly designated `dynamic`. (For example, `Object` is dynamic but its subclass `int` is not.)

[16] Objects can also be described by *record types* and *array types*, which are introduced later and which have additional uses, such as asking about the shapes of objects.

Virtual properties

Properties can be implemented not as storage locations that hold values but as *virtual properties*: as pairs of functions, a *getter* that produces a property value and a *setter* that receives a new property value.

```
class C {
    function get x() { ... }
    function set x(value) { ... }
}
```

(A getter without a setter implements a read-only property.) Virtual properties can't be added dynamically even if their class is dynamic.

Dynamic classes can also provide *catch-all* getters and setters that handle references to all dynamic properties (see the chapter "Behavior").

```
class D {
    meta function get(name) { ... }
    meta function set(name, value) { ... }
}
```

o.property The properties of an object are referenced by their names, so if `o` is an instance of the class C defined earlier (under "*Classes*") then `o.val` references the object's variable property `val`, and `o.f` references the method property `f`.

Builtins Predefined classes describe generic data (`Object`), numbers (`byte`, `int`, `uint`, `double`, and `decimal`), truth values (`boolean`), constant character sequences (`string`), mutable value sequences (`Array` and `Vector`), behavior (`Function`), binary relations (`Map`), time stamps (`Date`), regular expressions (`RegExp`), names (`Name`, `Namespace`, `NamespaceSet`, and `NamespaceSetList`), and errors (`Error`, `EncodingError`, `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `UndefinedError`, and `URIError`).

Note: ES3 constructor functions—like `Number`—are just function bindings in the top-level environment of ES3, and they can be changed by the program. In ES4 the constructor functions have become classes, which are bound as constants and cannot be changed.

Wrappers The classes `double`, `boolean`, and `string` are paired with *wrapper classes* called `Number`, `Boolean`, and `String` whose instances store instances of the classes they are paired with. The wrapper classes are present mainly for backward compatibility, but are also more flexible than the objects they wrap.

Note: ES3 has a notion of *primitive values* that are not true objects, and a method call on a "primitive string" results in the allocation of a new instance of the wrapper class `String`, on which the method is subsequently called. Since all values in ES4 (except **null** and **undefined**) are true objects, this implicit wrapping no longer takes place. This is not expected to be an important issue for compatibility since the `string` and `String` classes share a single prototype object; code written for ES3 will continue to work in ES4. Furthermore, the ES4 objects that replace the primitive values of ES3 provide catch-all methods that absorb any attempt to set a property and return **undefined** as the value of any unknown property, since this is what the wrapper mechanism in ES3 effectively does.

Inheritance A class D can optionally *extend* another class B. B is the *superclass* (sometimes called the *base class*) and D is the *subclass* (or *derived class*). We say that D *inherits* properties from B. The definition of D can add properties that are not named in B, and D can replace behavior defined in B by *overriding* B's methods. The purpose of inheritance is to allow common aspects of a group of classes to be factored into one common base class.

```
class C {
    var val
    function f(n) { return n+1 }
}
class BetterC extends C {
    var large = 123456                    // new value property
    override function f(n) { return n*3 }  // overridden method property
}
```

A class that does not explicitly extend any other class is said to extend the class `Object`. As a consequence, all the classes that exist form an *inheritance hierarchy* that is a tree, the root of which is `Object`.

A class designated as `final` cannot be extended, and `final` methods cannot be overridden:

```
class C {
    final function f(n) { return n*2 }
}
final class D extends C {
    function g() { return 37 }
}
```

Metaclass A class is reflected at run-time as a *class object*, which in turn is described by a hidden *metaclass*. The class definition adds properties to its metaclass by designating the properties *static*:

```
class C {
    static var val = 37                // variable property on metaclass
    static function f(n) { return n+1 } // method property on metaclass
}
```

The static properties are available as `C.val` and `C.f`.

Though the metaclass is hidden, it can be inspected through the use of the reflection facility.

Prototype Every class object has a constant property `prototype` that holds a reference to an object that is the *prototype* of the instances of the class, and there is a hidden reference from each instance to its prototype. The prototype holds values that are shared among all the objects that reference it. Properties on an object's prototype appear to casual observers as properties on the object itself: reading a property from the object will find the property in the prototype if it is defined there and not in the object.

The prototype of an instance of a class C is always an instance of a hidden class C'. C' is similar to C—notably, it has the same name and properties as C—but C' is always `dynamic`, and it has a different prototype value (namely, the prototype object of the base class of C). The value of `Object'.prototype` is **null**.

A class definition can add properties to its prototype object by giving them the `prototype` attribute:

```
class C {
     prototype function toString() { return "this is a C" }
}
```

`var`, `const`, and `function` properties can be placed in the prototype using this mechanism. Properties defined with the `prototype` attribute are always DontEnum but are neither DontDelete nor ReadOnly.

Note: In ES3 the prototype of `RegExp` is an `Object`, not a `RegExp'`. ES4 fixes this anomaly.

Shared prototypes

Backward compatibility with ES3 requires that some predefined classes share prototypes. The prototypes of `byte`, `int`, `uint`, `double`, and `decimal` are the same as that of `Number`; the prototype of `string` is that of `String`; and the prototype of `boolean` is that of `Boolean`. This sharing is not expressible in the language and user programs cannot implement it for their own classes.

Literals

Instances of some predefined classes can be written down as literal text:

- `1` and `3i` are both `int`
- `2147483648`, `0x20`, and `35u` are all `uint`
- `37.5`, `7e4`, `8d`, and `8589934592` are all `double`
- `17.5m` is `decimal` (the `m` is for "money")
- `"ecmascript"` and `"""E262-3 "ECMAScript""""` are `string`
- `true` and `false` are `boolean`
- `null` and `undefined` are the special non-object values **null** and **undefined**
- `/[a-z_][a-z0-9_]*/ig` is a `RegExp`
- `{x:10}` is an `Object` with a single property called `x` whose value is 10
- `{const ns::x: 10}` is an `Object` with a single ReadOnly property called `ns::x` whose value is 10
- `{get x() { return global_x }}` is an `Object` with a read-only virtual property `x`
- `function(n) { return n+1 }` is a `Function`
- `[10,20]` is an `Array` of two elements.

Literals always create new instances of the objects they denote, but in the case of `byte`, `int`, `uint`, `double`, `decimal`, `string`, and `boolean` it is not possible to detect whether a new instance is created or an old instance is reused as these classes are final, non-dynamic, and have no variable properties.

Note: Though the literal `1` would result in a "primitive number" (a `double`) in ES3, the ES4 behavior is entirely compatible because all number types are interconvertible, share a single prototype object, and `int` arithmetic overflows to `double` as appropriate. (`uint` arithmetic overflows in the same way.)

Constructor functions

Instances of `Object` can be created in one more way. If the value of an expression *E* is a `Function` (call it `fn`) then the expression "`new` *E*" creates a plain `Object` and calls `fn` to initialize the object (which is dynamically bound to the special name `this` inside the function body). Any arguments passed in the constructor expression are passed on to `fn`. In this context `fn` is called a *constructor function*. There is an added power here: the value of `fn.prototype`, which is normally just an empty `Object`, can be changed before construction to reference any object. In this way it is possible to construct objects that have prototype hierarchies not described by a class structure.

V. Types

Types

Types are compile-time values that describe run-time values. A (run-time) value `v` "has" type T if T describes `v`. The value **null** has the type `null`; the value **undefined** has the type `undefined`; and an object has the type that is either the class of which the object is an instance, or the *structural type* that annotated the object when it was created.

A value `v` also has type T if the actual (allocated) type of `v` is U and U is a *subtype* of T.

Interfaces

An *interface* gives an additional type to an object. An interface describes a collection of method properties; a class can declare that it *implements* an interface and then back that up by providing definitions for the methods in the interface. Then instances of the class also have the type of the interface.

```
interface I { function f() }
interface J { function g(n) }

class C implements I, J {
    function f() { return "foo" }
    function g(n) { return n+2 }
}
```

Record and array types

Objects can also be described by ad hoc *structural types* that are independent of classes and interfaces. For example, `{ x: int, y: int }` is a *record type*; any object that has fields `x` and `y`, each of type `int`, has this type, even if they have other fields too.

Similarly, `[int]` is an *array type* that describes `Array` objects whose elements must all have type `int`. `[int,string]` is a *tuple type* that describes `Array` objects whose first element must be `int` and whose second element must be `string` (and whose remaining elements are unconstrained).

Note the similarity of the type syntax to the literal syntax that would be used to create objects described by the types.

Record and array types can be instantiated either by the `new` operator

```
new [ int ](7)
new { x:int, y:string }( 3, "foo" )
```

(where the arguments are used to indicate the `Array` length and the initial property values, respectively; in the latter case the order of arguments corresponds to the order of properties in the type) or by appending the types to literals creating the objects:

```
[ 1,2,3 ] : [ int ]
{ x:10, y:"foo" } : { x:int, y:string }
```

It is important to realize that the object value `{ x:3, y:4 }` does not have the type `{ x:int, y:int }` since an object has a type only if the properties described by the type are always present with their respective types. The properties of `{ x:3, y:4 }` can be deleted and can hold non-`int` values.

Record types may describe virtual properties, but the type does not reveal that the property is virtual: a getter `g` returning `T` is described simply as `g:T` in the record type.

Annotations

The record types above show the general *annotation syntax* used in the language: a colon followed by a type expression. A type annotation can be added to a property, to a function parameter, to a function itself (applying to the function's return value), to a variable, or to an object or array initializer.

A program that stores a value of type other than `T` into a property annotated with type `T` (or tries to return it from a function annotated with type `T`) is in error. The error can sometimes be detected before the program runs—see the chapter "Verification"—but at the latest it will be detected when the program stores a value with the wrong type into the property or tries to return it from a function.

An un-annotated property, parameter, function, or variable is implicitly annotated by the type `*` ("any").

Function types

Structural *function types* describe functions and methods. The type

```
function (int, string): boolean
```

describes a `Function` object that takes two arguments, one `int` and the other `string`, and which returns a `boolean`. If a function is a *method* it is possible to describe the type of objects of which it may be a

method by using a special `this` annotation, and if it takes optional arguments their types may also be described using special syntax:

```
function (this:C, int, string=): boolean
```

The preceding function can be a method only on `C` objects, and its second argument is optional but must be a `string` if it is present.

The return type of a function can be written `void`, which means that no return value is allowed.

Rest arguments can also be represented in the type, and constraints can be placed on the types of the rest arguments:

```
function (int, ...): boolean
function (int, ...[string]): void
```

Union types A *union type* is a collection of other types. It never describes an object, but when used as a constraint on a property it means "any of these types". The type `(int,string)` means "`int` or `string`".

"Any" The type written `*` (pronounced "any") is a structural type that describes any value. In most respects it is compatible with but not identical to the union `(null,undefined,Object)`.[17]

Type definitions

Class and interface definitions define types. Types can also be renamed or abbreviated by means of special *type definitions*, which are especially helpful in giving names to structural types:

```
type Point = { x: int, y: int }
```

Defined types are reflected at run-time as objects, and if the name of the type is `T` then the constant binding `T` holds one of these reflected objects at run-time.

Type meta-objects

Any type description also defines a *type meta-object* describing that type. A type meta-object is an instance of a hidden class that implements one of the *meta-object interfaces* `ClassType`, `InterfaceType`, `AnyType`, `UnionType`, `RecordType`, `ArrayType`, `FunctionType`, `NullType`, `UndefinedType`, or `ParameterizedType`.

If `s` is an instance of the class `Shape` then the value of the expression `intrinsic::typeOf(s)` is an object that implements the `ClassType` interface and that describes the elements of the `Shape` class.

(The reflection facilities in ES4—the type meta-objects and the meta-level hooks described later—do not provide a full-fledged meta-object protocol for the language.)

Predefined union types

The language predefines several convenient union types:

```
type AnyString = (string,String)
type AnyBoolean = (boolean,Boolean)
type AnyNumber = (byte,int,uint,double,decimal,Number)
type FloatNumber = (double,decimal)
```

Deep types Type descriptions can nest types within other types. The previous examples show that class types like `int` can appear inside record and array types, but record and array types also nest directly, forming types with deep structure.

[17] For type-theoretic reasons it is convenient to have a separate "any" type at the top of the type hierarchy.

```
type Person = { name:{ last:string, first:string }, born:Date, spouse:* }
```

This example highlights a restriction on structural types: they cannot be recursive. Though the natural type annotation for `spouse` is `Person`, such an annotation would be illegal.[18] The prohibition on recursive types applies also to mutually recursive types.

Nullability As shown earlier, class names can be used as annotations on, for example,. a variable, and the annotation allows only instances of the class (or a subclass) to be stored in the variable. But there is a subtlety: the value **null**, meaning "no object", is implicitly a member of the value sets of class and structural types, so **null** can be stored in the variable also. The ability to store **null** is occasionally the source of run-time errors. ES4 provides a facility for excluding **null** from the legal values for annotated variables:

```
var v : C! = ...
```

The postfix "`!`" can be read as "not null", and the variable is said to be *non-nullable*. Variables that can accept **null** are *nullable.*

The postfix "`!`" operator is a general operator on types. If its argument is a union type that contains the type `null` it results in a union type with `null` removed. (Otherwise it has no effect.)

When a class C is defined the type that is created is in fact `(C,null)`, that is, the union of the class type itself with the `null` type. Thus applying the "`!`" to the type removes the `null` type, leaving only the class type.

There is also an operator for creating a union of any type `t` with `null`, written "`t?`"—this is exactly the same as writing `(t,null)`, just more readable.

The final feature of nullability is that a class can be defined so that the type that is created does not admit `null` by default. This is done by suffixing "!" to the class name in the class definition:

```
class int! { ... }
```

Such a definition means that variables annotated with the type `int` will not accept **null** values (though those annotated with `int?` will).

Whether a class is non-nullable depends only on whether it is directly designated non-nullable; a non-nullability designation has no impact on the class's subclasses.

Initialization ES4 provides default values for some predefined non-nullable classes (`byte`, `int`, `uint`, `double`, `decimal`, `string`, and `boolean`), but otherwise the default value for a property of any class, record, or array type is always **null**. Non-nullable properties therefore have no default value, and must be initialized before they are ever accessed.

In a class the constructor method must initialize instance properties before they can be read by any method. ES4 does not require implementations to implement the data flow analyses that are necessary to determine whether a property is initialized before it is accessed—on-line compilation on small devices would then be too costly—but instead requires non-nullable properties to be initialized before the constructor body is entered. Special syntax in the head of the constructor allows the initialization to take place:

[18] There are technical reasons for this having to do with the complexity of run-time type checking. Recursive types can be described only by classes and interfaces.

```
class C {
    function C(p) : x = p
    {
        ...
    }
    var x : D!        // must be initialized
}
```

In the initializer "`x = p`" the scope of the left-hand-side includes the instance being constructed, whereas the scope of the right-hand-side includes the class object and the constructor parameters. Notably, `this` cannot be used on the right-hand-side as a reference to the object being initialized.

The initializer can call the constructor in the superclass using the syntax `super(...)`. If the superclass constructor takes non-optional arguments then it must be called explicitly.

Subtypes

An object that has type `T` also has type `U` if `T` is a *subtype* of `U` (written `T <: U`). Intuitively, T is a subtype of `U` if `T` is *more specific* than `U`, that is, `T` describes fewer values than `U` because `T` has more constraints.

A key point of subtyping is that if a variable is annotated with a type `U` then it will also accept any value of type `T` if `T` is a subtype of `U`.

Without getting technical or exhaustive, the following rules for subtyping hold:

- any type is a subtype of itself;
- any type is a subtype of `*`;
- the only member of the type `null` is the value **null**, and the only member of the type `undefined` is the value **undefined**;
- if D is a subclass of B then D is a subtype of B;
- if C implements the interface I then C is a subtype of I;
- a type is the same as the union type containing just that type;
- a union type A is a subtype of another union type B if every member of A is a subtype of some member of B;
- a record type A is a subtype of another record type B if all of B's properties are also in A, with equal types; and
- a function type A is a subtype of another function type B if A has the same number of fixed parameters as B, each of A's parameter types is a supertype of the corresponding parameter of B, and the return type of A is a subtype of the return type of B.

"like"

The special type operator `like` is used to test whether an object "looks like" a particular type even without being of that type. For example,

```
var v: like { x: int, y: int }
```

will accept both of these values:

```
{ x:3, y:3 } : Point
{ x:3, y:3 }
```

even though the latter does not have fixed properties `x` and `y`. The trick is that `like` examines the properties present in the value, and those properties' values, and if the properties and values conform to the type then the type test succeeds. Thus `like` is a weaker type test than the standard test for annotations.

The value that is tested may subsequently be changed so that it no longer conforms to the type, but such a change has no effect on the type test, which only tests the value's structure at a particular time.

Rationale: "Like" types and "wrap" annotations (discussed below) are unusual features in a programming language, and were the outcome of a year's discussion about the extent to which structural types can be

said to describe classical ES3 data objects, which have no fixtures and can't be said to *have* any particular structural type except `{}`. The issue is important, because a key use case for structural types is at the interface between typed code (an upgraded Ajax library or toolkit) and untyped code (existing client code for that library). Library authors would like to evolve their code to use types and perhaps strict-mode verification internally; client programmers would like to make use of new features but don't have the resources or desire to upgrade their code. As a compromise, the library authors can provide APIs that check (by means of `like`) that the client is passing plausible data, and either wrap the data to static types or just use the check for what it is: a precondition.

```
function low32( x: like { p: int } ) {
    return x.p & 0x1F
}

low32( { p: 12 } )              // Succeeds
low32( { p: "foo" } )           // Fails

class C { var p: int, q: int }

low32( new C )                  // Succeeds

function myAPI( x: wrap { p: int } ) {
    function fn(k: { p: int })  // Note, neither "like" nor "wrap"
        ...
    fn(x)                       // *Always* succeeds
}

myAPI( { p: 12 } )              // Succeeds
```

"wrap" The type annotation `wrap` performs a test like `like`, and if the manifest type of the value is not a subtype of the type tested against, then a wrapper object is constructed that does have the correct type. The following program is type-correct, which is not the case if `wrap` is replaced by `like`:

```
var v: wrap { x: int, y: int } = { x: 10, y: 20 }
var w: { x: int, y: int } = v
```

`wrap` is also an operator; it is described later.

Conversions The language defines conversions between values of different predefined types. The conversions make the language more pleasant to use at the cost of slightly relaxing the constraints introduced by the type annotations. The members of `AnyString` interconvert, as do the members of `AnyBoolean` and the members of `AnyNumber`. In addition, any value in the language converts to a member of `AnyBoolean` (**undefined**, **null**, zero, **NaN**, and the empty string converting to **false** and everything else converting to **true**).

No other conversions take place automatically. User code must call functions explicitly to convert from one type to another. By convention, the `meta::invoke` method on a class object acts as a converter to that type of the class, so `string(x)` converts `x` to `string`.[19]

Parameterized types

A *parameterized type* is a template for new types and is defined by adding type parameters to class, interface, type, and function definitions:

[19] Most predefined classes have conversion methods that accept any type, but `meta::invoke` can have type constraints like other methods, and nothing in the language prevents `meta::invoke` from being used for other purposes.

```
class Pair.<T> {
    var first: T, second: T
}
type Box.<T> = { value: T }
function f.<T>( x:T ): T ...
```

A parameterized type is instantiated by supplying concrete types for its parameters:

```
new Pair.<int>(3, 4)
f.<int>(37)
var v: Box.<boolean> = new Box.<boolean>(true)
```

The predefined types `Map`, `Vector`, `IteratorType`, and `ControlInspector` (among others) are parameterized.

The parameterized types in ES4 do not allow for type parameter constraints or variance annotations. However, nothing precludes the inclusion of these in a future edition of the language.

VI. Structure

Names

A *name* is a pair of a namespace and a string. Some names are *unqualified*: their namespace value is **null**. All other names are *qualified* by their namespace. Two names are equal if their respective namespaces and strings are equal.

Most often names appear directly in program text, and there is convenient syntax for them. The unqualified name whose string is `"x"` is written simply `x`. Namespaces can be named, so if `ns` is the (unqualified) name of a namespace then `ns::x` is a qualified name whose string is `"x"`. Furthermore, if a variable written `x` holds a string *s* then `ns::[x]` is a qualified name whose string is *s*.

Names can be reflected at run-time as instances of the predefined class `Name`, and arbitrary names can be created by `new` expressions, but such names have few uses.

Namespaces

Namespaces are compile-time values created by a `namespace` directive. The following lines create namespace constants `ns1`, `ns2`, `ns3`, and `ns4`; the first two are anonymous (and equal), the last two have program-defined content (and are also equal):

```
namespace ns1
namespace ns2 = ns1
namespace ns3 = "www.ecma-international.org"
namespace ns4 = "www.ecma-international.org"
```

Namespaces can be reflected at run-time as instances of the predefined class `Namespace`, and the name bound by the `namespace` directive denotes a run-time constant binding whose value is such an object. Arbitrary namespaces can be created by `new` expressions.

Predefined namespaces

ES4 predefines and reserves the namespaces `__ES4__`, `intrinsic`, `iterator`, and `meta`. The `__ES4__` namespace is always *open* in ES4 programs (*i.e.*, all names defined in it are visible to the program without qualification—more about this later), and the qualified names of the others are `__ES4__::intrinsic`, `__ES4__::iterator`, and `__ES4__::meta`, respectively.[20]

[20] The `__ES4__` namespace provides backward compatibility with code written for ES3. Since ES4 introduces some constant bindings—like `decimal`—that might have been used as variable names in ES3, ES3 programs must be loaded in a scope where these identifiers are not visible without qualification. Therefore top-level names introduced by ES4 are bound in the namespace `__ES4__`, and ES3 code is loaded in a scope where this namespace is not open. A similar mechanism may be used in future editions of ECMAScript, and the language now reserves identifiers of the form `__ES<n>__` for any nonnegative integer `n`.

The `intrinsic` namespace is used for bindings in the global object and in the predefined classes that are available for early binding. The only way user code can introduce a new binding in this namespace is if the new binding is a method in a subclass of a predefined class and overrides an inherited method in the `intrinsic` namespace.

The `iterator` namespace is used to segregate all the names introduced by the iterator and generator protocols from other names.

The `meta` namespace is used for system protocols; for example, `meta::get` is a catch-all getter and `meta::invoke` is the method that will be called if the containing object is called as a function. User code cannot use this namespace for purposes other than implementing these system protocols.

In addition, the names `internal`, `public`, `protected`, and `private` denote namespace values, but the values depend on the context of the use.

Bindings

Names are *bound* by the language directives `package`, `class`, `interface`, `type`, `var`, `const`, `function`, `const function`, `let`, `let const`, and `let function`, by `let` blocks and `let` expressions, by function parameters, and at run-time by the `with` and `eval` statements.

For example, a `class` definition binds its class name and a `var` directive binds its variable names.

Some binding directives may be annotated in some contexts by a namespace. The annotation is prefixed to the binding directive—the following line defines the class whose name is `ns::C`:

```
ns class C
```

If a namespace annotation is not used then a default namespace is supplied where applicable: in a class or package the default is `internal`, in an interface the default is `public`. The default can be overridden with a pragma that supplies the default to use in the pragma's scope:

```
use default namespace ns
```

Binding objects and scopes

The kind of the binding directive (*e.g.*, `var` or `let`) determines the *binding object* into which the binding is introduced. At each program point there are two binding objects that can receive new bindings, a *variable object* and a *block object*.

New variable objects are created for three phrases in the language: the program, the class, and the function.

New block objects are created for *blocks*—regions of source text that are usually delimited by a matching pair of curly braces—and for some statements and expressions that bind names.

The region of text associated with a particular binding object is known as the *scope* of bindings bound by that object. [21]

A new binding object is created for a scope every time the scope is entered during execution, as when a function is called (activated) or a block in the body of a loop is entered repeatedly.

For example, a `function` definition introduces a new variable object (every time the function is activated) that is associated with the function (including the parameter list) and a new block object that is associated with the function's body. `var` and `function` bindings in the function body create bindings in the variable object (so their scopes are the entire function); `let` bindings create bindings in the block object (and their scope is just the body).

[21] In other words, ES4 uses lexical block scoping.

Some scopes are block-like but are not delimited by curly braces. A `for` loop can bind a name (typically used for iteration control) whose scope is the head of the loop and the loop body. The following loop binds the names `i` and `limit`:

```
for ( let i=0, limit=x.length ; i < limit ; i++ )
    sum += x[i]
```

The binding objects of a program form a tree, with the variable object for the program (the global object) at the root and the variable and block objects for other program phrases intermingled along the branches.

A name is *visible* in its entire scope if it is not *shadowed* by a binding in a scope nested inside the scope of the name. Typically this means that a block contains a nested block that provides a new binding for the same name:

```
{
  let a = 10, b=1
  {
    let a = 20
    print(a+b)              // prints 21
  }
  print(a)                  // prints 10
}
```

Run-time introduction of bindings

Some phrases in the language can introduce arbitrary names into variable objects at run-time. The `eval` operator can create new bindings in its innermost variable object, and assignment statements can create arbitrary bindings in the global variable object.

"with"

The `with` statement introduces a new scope object that is not a binding object. This object has arbitrary content at run-time, and it is consulted during the lookup of names in the body of the statement:

```
let x = 5
with ({x: 10})
    print(x)                // prints 10
```

The `with` statement does not violate block scoping, but it will in general be impossible to say whether a reference to a variable in the body of the `with` refers to a propety bound by the `with` or a name in an outer scope.

ES4 also provides a restricted form of `with` that exposes only a set of known names and makes static analysis (whether by machine or human) possible again. The new form is described later in the chapter "Syntax".

Name lookup and open namespaces

A reference to a name is resolved by looking for the name in the innermost binding object that is in effect at the point of reference and then in every nesting binding object, from innermost to outermost. Name lookup is parameterized by the set of *open namespaces*. The program can open a namespace in a block with a `use namespace` directive. All unqualified names in the block then become implicitly qualified by that namespace, in addition to being implicitly qualified by all namespaces that were previously open. A reference to an unqualified name can therefore result in multiple names being found. If more than one name is found in a particular scope then the reference is ambiguous; this is an error. For example:

```
ns var x
var x
{
  use namespace ns
  x += 5              // error!
}
```

During name lookups, namespaces opened in inner blocks take precedence over namespaces opened in outer blocks, and an unqualified reference is unambiguous even if it is found with multiple qualifications, as long as the qualifying namespaces are opened in different blocks. Furthermore, namespaces that are explicitly opened with `use namespace` take precedence over those that are implicitly opened, such as `internal`.

package A package is a compile-time value comprised of a package name and two namespaces known as the package-internal and the package-public namespaces. The two namespaces are created by the ECMAScript implementation and are not visible to the program, but the package introduces a block scope in which these two namespaces are both open. The definitions in the package are evaluated in this block scope.

The variable object in a package is the global variable object—package definitions are global definitions, qualified by namespaces. `class`, `function`, and `var` bindings in the package can be qualified by either an ordinary namespace value or one of the special namespaces `internal` and `public`, which are just aliases for the package-internal and package-public namespaces.

```
package org.ecmascript.experiment {
    internal var v;
}
...
package org.ecmascript.experiment {
    public function f(k) { v += k; return v-1 }
}
```

As the example shows, a package can be defined across disjoint pieces of text. Each fragment has a separate block scope but they all share the same variable object.

The user program can `import` a package into a scope. This entails simply opening the package's public namespace in that scope, so that the global bindings qualified by that namespace become visible.

```
import org.ecmascript.experiment.*
f(37) // calls org.ecmascript.experiment.f
```

Bindings can also be imported singly, they can be renamed on import, and the package's full name can be used to qualify a name when the name is referenced.

A package can be local to a program fragment[22] by designating it as `internal package`; this avoids name clashes when code from multiple sources is mixed for effect.

class and interface

`class` and `interface` definitions create compile-time values that are always reflected as run-time values that are stored in the global variable object. Syntactically, `class` and `interface` definitions can only appear at the top level or directly inside a package.

When a `class` is instantiated a new variable object and a new block object are created for the instance; the variable object receives `var`, `const`, and `function` bindings from the class, whereas the block object receives any `let` and `type` bindings at the top level of the class definition.

An `interface` creates neither a variable object nor a block object; its use is only as a constraint on the `class`es that implement it.

type `type` definitions create compile-time values that are always reflected as run-time values that are bound in the innermost block object:

[22] Think "file".

```
type Point = { x: int, y: int }
```

(Type definitions may have special scoping rules; TG1 are still working on the details.)

var, const

`var` and `const` directives introduce mutable and constant bindings in the innermost variable object, respectively:

```
var x = 10
const color = 0xFEFEFE
```

`var` bindings can be redundantly defined (provided the variables are not annotated). Just as in ES3, the binding is created in the variable object when that object is created but the initialization takes place only when the lexical location of the binding statement is reached.

`var` bindings can be made by `for` loops (as in ES3):

```
for ( var i=0 ; i < 10 ; i++ )
    sum += i
```

functions

Function definitions create variable bindings holding function values in the innermost variable object. Functions have a *parameter list*, which is a list of names that will be bound in the function's variable object and whose initial values will be the arguments passed to the function in the function call.

```
function f(x) {
    return x+x
}
```

If the function is annotated with parameter types or return types then the variable binding for the function will have an annotation expressing the function's type (in the form of a structural function type). The annotations on the function serve as checks on parameter values, on the return value, and on values stored in the parameter variables by the function body. The annotation on the variable binding prevents the value from being updated to a value that is not type-compatible with the initial function definition.

```
function f(x: int): int {
    return x+x
}
```

Constant function bindings in the variable object are created by the phrase `const function`.

Functions can have optional arguments, where the parameter list describes default values to be substituted for missing arguments in the call:

```
function f(x, y=0, z=0) ...
```

Functions can also have rest arguments, where extra parameters are collected into an `Array` which becomes the value of that parameter:

```
function f(x, y, z, ...others) ...
```

A function is an object; the property `length` on this object is the number of fixed and optional arguments accepted by the function.

generic functions

A *generic function* is a function object that performs run-time dispatch to one of several attached *methods* based on the actual types of the argument values and the signatures of the available methods. (Generic functions are often called *multimethods*.) Method selection is such that the *most specific* method is chosen: from among the methods that can be applied to the arguments, the most specific method will be the one

whose signature is more specifically matched to the arguments than all the other methods in at least one argument position.[23] (Some calls to generic functions may be ambiguous and may cause run-time exceptions.)

A generic function must first be defined by a body-less `generic function` phrase:

```
generic function intersect(s1, s2);   // No body here
```

Then methods can be attached:

```
generic function intersect(s1: Shape, s2: Shape) {
    // general intersection method
}

generic function intersect(s1: Rect, s2: Rect) {
    // fast intersection for rectangles
}
```

Generic functions can be `static`, and static generic functions can be `final`, preventing methods from being added outside the class definition.

If a call to a generic function is not ambiguous then there is an ordered set of methods that can be applied (in which set the most specific method is the first). Less specific methods in the ordered set can be called from within more specific methods by calling the special function `nextMethod`.

let, let const `let` and `let const` directives introduce bindings into the innermost block object (unlike `var` and `const`, which introduce bindings into the variable object):

```
let x = 10
let const color = 0xFEFEFE
```

`let` also has an expression form:

```
x = 10 + let (a=f(k)) a*(a-1)
```

and a block form:

```
let (a=f(k), b=b) {
    b *= 2
    return a*(a-1) + b
}
```

The expression and block forms evaluate the initializing expressions in scopes that do not include the new bindings, as the latter example illustrates.

`let` bindings can also be made by `for` loops, and `catch` clauses in `try-catch` statements and `case` clauses in `switch type` statements `let`-bind the names they introduce.

let function The phrase `let function` binds the function in the block object (unlike `function`, which binds the function in the variable object), and provides block-local scoping for functions.

Program units

Program fragments can be bundled into named *program unit*s. An ES4 implementation will load ("resolve") a unit on request. A program asks for a unit to be resolved by using an *unresolved unit* pragma:

[23] Generic functions in ES4 are similar in most respects to those in the Dylan programming language.

```
use unit URLParser "http://www.mycompany.com/lib/URLParser"
```

The preceding line asks for the unit `URLParser` to be loaded if it hasn't already been loaded, and provides a locator for the unit (here in the form of a URL).[24]

The unit itself is defined by a *resolved unit* pragma, which contains code and further unresolved unit pragmas:

```
unit URLParser {
    use unit Splitter "http://www.mycompany.com/lib/splitter"
    use unit Unicode "http://www.mycompany.com/lib/Unicode"
    package com.mycompany.urlparser {
        ...
    }
}
```

Program units provide multiple facilities: a mechanism for declaring dependencies between program fragments; a mechanism for loading a program fragment on demand (and loading it just once); a mechanism for bundling several source code fragments that together form a consistent whole (in the sense that the unit contains no references to undefined types, for example; this is important in strict mode); and finally a mechanism for bundling several source code fragments that can be loaded together, thereby reducing the pressure on busy servers.

VII. Behavior

Versioning TG1 has defined two mechanisms to aid environments that need to handle both ES3 and ES4 code (and which will handle future language editions as well).

ES4 implementations will have a global constant `__ECMASCRIPT_VERSION__` whose value is an integer specifying the latest edition of the language handled by the implementation (*i.e.*, `4` in ES4 implementations).

TG1 has also made use of the provisions of RFC 4329 by requiring that ES4 scripts whose content are described by a MIME type must use a MIME type with a "`version=4`" modifier, either during transport or in the `SCRIPT` tag in a web browser (for example).

Type annotations and type checking

Type annotations on locations (whether variables, parameters, or properties, and irrespective of the binding form or the binding object) act as run-time assertions—*type checks*—on the types of values stored into those locations, whether by the program or by the implementation. If a type check fails, the implementation throws a `TypeError` exception.

Strict mode programs additionally make use of the type annotations for static type checking and verification.

Programs that use type annotations in principle perform more run-time type checking than programs that do not use annotations, and hence may run more slowly. An ES3 implementation performs type checking as part of the evaluation of primitive operations and some predefined methods; an ES4 implementation performs all the checks the ES3 implementation would perform and an additional check every time a value is stored into a typed location.

The reason for introducing type annotations into ECMAScript is their benefit to programming in the large, not their potential benefits to performance. Programs that check more often—that have more assertions—will tend to have fewer errors and be more easily debugged, all other things being equal.[25] Some

[24] The interpretation of the locator depends entirely on the host environment. In a web environment a URL could naturally be considered a location from which to fetch source code by HTTP, but that behavior is not required by ES4.
[25] For once, "all other things" *are* equal, since type annotations are entirely optional in ECMAScript.

invariants captured by type annotations can also be added to ES3 programs as ad hoc pre- and post-conditions on functions, for example, but doing so requires significant programmer discipline and potentially incurs greater run-time cost than do type annotations in ES4.

In practice, good implementations of ECMAScript will be able to remove some redundant type checks, and it is plausible that programs or program fragments that are fully statically typed—whose every variable and function have a type annotation—will have fewer dynamic type checks than corresponding untyped code, and will run faster. Additionally, it is much harder for an ES3 implementation to remove redundant ad hoc pre- and post-conditions than it is for an ES4 implementation to remove redundant type checks.

Pragmas

A *pragma* is a compiler directive that can only appear at the beginning of a program fragment or at the beginning of a block. Its effect is limited to the fragment or block, and it can be overridden in nested blocks. Some of the pragmas are:

- `use default namespace`, which sets the default for namespace qualification in definitions;
- `use namespace`, which opens a namespace so that it qualifies otherwise unqualified names;
- `use strict` and `use standard`, which select the language dialect (provided strict mode is supported by the particular implementation); and
- `use decimal`, which selects a `DecimalContext` object containing rounding and precision settings for decimal arithmetic

Iteration

An object is *iterable* if it has the structural type `IterableType` (`iterator` is a predefined namespace):

```
type IterableType.<T> = {
    iterator::get: function (boolean=): IteratorType.<T>
}

type IteratorType.<T> = {
    next: function(): T
}
```

An *iterator* is an object that matches both `IterableType`[26] and `IteratorType` and in addition is its own iterator (because the `iterator::get` method returns the object), for example:

```
class Fibber {
    iterator function get(deep:boolean = false): IteratorType.<int>
        this
    public function next(): int {
        [a,b] = [b,a+b]
        return a
    }
    private var a=0, b=1
}
```

Printing a lot of Fibonacci numbers is easy:

```
for ( i in new Fibber )
    print(i)
```

Occasionally it is useful for the `next` method to signal that there are no more values; it does so by throwing the singleton value `iterator::StopIteration` (whose type is `iterator::StopIterationClass`):

[26] The optional argument to `iterator::get` expresses the client's preference for whether iteration should be deep (go down the prototype chain) or shallow.

```
        public function next(): int {
            if (b >= cutoff)
                throw iterator::StopIteration
            [a,b] = [b,a+b]
            return a
        }
```

More generally an object is *itemizable* if it has the structural type `ItemizableType`:

```
type ItemizableType.<K,V,I> = {
    iterator::getKeys:    function (boolean=): IteratorType.<K>,
    iterator::getValues:  function (boolean=): IteratorType.<V>,
    iterator::getItems:   function (boolean=): IteratorType.<I>
}
```

If an object can be said to contain key-value pairs then `getKeys` returns an iterator that iterates across keys, `getValues` returns one that iterates across values, and `getItems` returns one that iterates across the key-value pairs.

As seen below, the `for-in` statement uses `iterator::get` to obtain an iterator, and `for-each-in` uses `iterator::getValues`.

Any `Object` in ES4 is both iterable and itemizable, with the `getItems` iterator returning a `[key,value]` pair.

"for in" statement

The for-in statement enumerates the names of the properties of an object and its prototype (without duplicates); it is present in ES3 but has been changed in three respects.

First, the value that is the operand of `for-in` can be **null** or **undefined**. In that case, the loop body is not entered.

Second, in ES4 the local enumeration order—the order in which properties of any one object in the prototype chain is enumerated—is defined to be the order of insertion of those properties into that object. Objects in the prototype chain must be visited in order, starting with the object that is the operand of the `for-in` statement.

Third, the `for-in` statement in ES4 is based on the new iterators. The default iterator for the class `Object` implements ES3-compatible `for-in` enumeration, but other classes need not, and the predefined class `Vector` provides an iterator across its array elements only, ignoring the prototype. The `for-in` statement will use this iterator. In fact, this statement:

```
for ( i in obj )
    print(i)
```

is equivalent to this:

```
let ($it = o.iterator::get(true)) {
    while (true) {
        try {
            i = $it.next()
        }
        catch (e: iterator::StopIterationClass) {
            break;
        }
        print(i)
    }
}
```

Only public properties whose DontEnum attribute is not set are enumerated by the default iterator.

Note: With the change in enumeration order TG1 merely codifies reality: Netscape Navigator made enumeration order be insertion order, Microsoft Internet Explorer followed suit, and now the web depends on it. As for the handling of **null** and **undefined**, Internet Explorer allowed it and now other browsers do as well.

"for each" statement

The new `for-each-in` statement (originally introduced in E4X) behaves like the `for-in` statement in all respects save one: it iterates over the values of properties of an object instead of their names. The following produces the string "123"

```
let s = ""
for each ( let n in [1,2,3] )
    s += n
```

To accomplish this the `for-each-in` statement retrieves its iterator by the `iterator::getValues()` method instead of `iterator::get()`.

Generators A function containing the new keyword `yield` is a *generator*: when it is called, it binds its parameters to their values and immediately returns a new *generator-iterator* object which can be called repeatedly to yield new values.

```
function fringe(tree) {
    if (tree is like {left:*, right:*}) {
       for (let leaf in fringe(tree.left))
           yield leaf
       for (let leaf in fringe(tree.right))
           yield leaf
   }
   else
       yield tree
}
```

Try it:

```
let tree = { left: { left: 37, right: 42 }, right: "foo" }
for ( let x in fringe(tree) )
    print(x)
```

It prints `37`, `42`, and `"foo"`.

Operator overloading through global multimethods

A number of generic functions ("multimethods") corresponding to the built-in binary and unary operators are predefined in the intrinsic namespace: `intrinsic::===`, for example. The ECMAScript implementation predefines methods on these generic functions to handle many predefined types—these methods cannot be replaced or superseded. The program is free to add methods to these functions that handle additional types, however, and when the implementation evaluates an operator for which an additional method has been defined, the method will be invoked if its signature matches the types of the operator's operands. For example,

```
class Complex! { ... }

generic intrinsic function +( a: Complex, b: Complex )
    new Complex( a.real + b.real, a.imag + b.imag )
```

```
generic intrinsic function +( a: Complex, b: AnyNumber )
    a + Complex(b)

generic intrinsic function +( a: AnyNumber, b: Complex )
    Complex(a) + b
```

"&&=" and "||=" operators

ES4 introduces assignment operators for the logical "&&" and "||" operators. These assignment operators are short-circuiting; if the value of the left-hand-side determines the value of the result then the right-hand-side is not evaluated.

Tail calls

ES4 has *proper tail calls*—the requirement that a function that calls another function and then just returns the latter's result, or tail calls another function and then returns nothing, must do so in a way that does not accumulate control stack. Proper tail calls facilitate extensive use of procedural abstraction: they allow recursive or mutually recursive functions to express iterative control behavior, that is, computations that do not consume stack space. State machines, interpreters, delegation-style programming, and macro systems are examples of such computations. Without proper tail calls, programmers are restricted to using loops or `goto`'s to express iteration. The use of procedural abstraction for iteration requires the use of un-abstract control structures[27] to consumption of control stack space, among other things.

Whether a call is in *tail position* is primarily specified syntactically—a call inside a `try` block with an associated `catch` or `finally` block is not in tail position, for example—but also depends on whether type conversions or type checks may have to be performed when a function returns to its caller. If `f` calls `g` in a tail position but the declared return type of `g` is not a subtype of the return type of `f`, or if the return type of `g` is not known and the return type of `f` is not `*`, then the call is not a tail call.

Meta-level hooks

ES4 provides hooks that allow the program to take on some of the responsibilities of the implementation in certain situations by implementing methods in the reserved `meta` namespace. The meta-level methods must always be fixtures and cannot be defined in prototype objects.

An object that implements the method `meta::invoke` receives control through that method when the object is called as a function. The predefined classes use this hook to implement type converters.

An object that implements the methods `meta::get`, `meta::set`, `meta::has`, and `meta::delete` receives control when the program tries to read, write, query, or remove a dynamic property on the object, and allows the object fully to control how dynamic properties are handled. The predefined classes `Array` and `Vector` use this hook to handle setting and getting of array properties.

Reformed "with" statement

The `with (o) { ... }` statement in ES3 extends the lexical scope in its body at run-time with all the properties of the object `o`, which is consulted every time a variable lookup reaches that scope.

ES4 allows the statement head to be annotated by a type constraint for `o` in the form of a record, class, or interface type. The constraint serves to specify exactly those fields that should be found in the object, and their types: `o` must have all the fixed fields described by the type, and they must be of exactly the types described by the type—subtypes or supertypes are not allowed. (`o` may have more fields, but they will be ignored during name lookup.) For example, this prints 21:

```
let o = new Point(1, 2)      // Point has x and y fixtures
let x = 10, y = 20
with (o): {x: int}
    print(x + y)
```

[27] Like "Cheney on the MTA", http://home.pipeline.com/~hbaker1/CheneyMTA.html

Arithmetic

ES4 provides five number representations: `byte`, `int`, `uint`, `double`, and `decimal`. In principle, only `decimal` is needed, but the others are present to meet special needs:

- `byte` represents an 8-bit unsigned integer. Its intended use is space optimization: as the element type for the parameterized `Vector` type.
- `int` and `uint` represent 32-bit signed and unsigned integers, respectively.[28] Their intended use is time and space optimization: taking advantage of the integer hardware on current CPUs, and as element types for the parameterized `Vector` type.
- `double` corresponds to the "primitive number" in ES3 and represents a 64-bit IEEE 754 binary floating-point number. It is needed for backwards compatibility, is suitable for most computations, and takes advantage of the floating-point hardware on current CPUs.
- `decimal` represent a 128-bit IEEE 754r decimal floating-point number. It provides intuitive base-10 arithmetic to high precision (34 decimal digits) and is in a sense the "right thing". Base-10 floating point is not yet widely implemented in hardware, as it is an emerging standard, but vendors are committing to it and TG1 expects it to be supported efficiently eventually.

Arithmetic operations generally convert their operands to a representation that can represent all operands, then perform the operation in that representation, and finally generate a result also in that representation. If an operation on `byte`, `int`, or `uint` overflows then the result will be in a "better" representation; `byte` operations overflow to `uint`, whereas `int` and `uint` operations overflow to `double`.

`decimal` arithmetic is always performed in a *decimal context* that specifies the rounding and precision. Decimal contexts are scoped lexically and are introduced by the `use decimal` pragma, whose argument is an instance of the predefined `DecimalContext` class:

```
ctx = new DecimalContext(12, "half_even")  // 12 digits of precision,
                                           //   round to even
{
   use decimal ctx
   x = a + b                 // "+" computed to 12 digits of precision,
                             //   rounding to even if necessary
}
```

Note that `byte`, `int`, `uint`, `double`, and `decimal` are not subtypes of `Number`; instead, all six types are members of the union type `AnyNumber`. The primary reason for this is that `Number`, for reasons of backward compatibility, must represent a `double`, and it is not natural for some of the other types (nor for possible future types, like complex numbers or arbitrary-precision integers) to be subtypes of `Number`.

"this"

ES3 does not preserve the value of `this` across function calls, and the consequence is that functions that are methods on objects cannot call even lexically nested helper functions without capturing the value of `this` in a variable and referencing that variable instead. ES4 takes the chore out of this style by specifying that `this` is preserved if a call is made by name to a function that is lexically nested inside any other. The following program prints 10:

```
function f() {
    function g() { print(this.x) }
    g()
}
v = { x: 10, f: f }
v.f()
```

The program will print "undefined' in ES3, because `this` in the function `g` is a reference to the global object in ES3.

[28] `int` and `uint` were present "under the hood" in ES3 as the Int32 and Uint32 types that were used by bitwise operators and in a few other places, like in Array indices.

"is" operator

The `is` operator—`v is t`—tests whether the value `v` is of the type `t`. The result is **true** or **false**. The right-hand-side `t` must be a type expression. In the rare cases where a reflected type object must be used it is possible to use the reflection facility, *e.g.*, `typeOf(v).isSubtypeOf(T)`, for some type object `T`.

"cast" operator

The `cast` operator— `v cast t`—checks that the value `v` is of the type `t`, and throws an exception otherwise. If it succeeds, the *static* type of the expression is the type on its right-hand-side, so its effect is mainly useful in strict-mode programs. The right-hand-side `t` must be a type expression. (The dynamic effects of `cast` can be simulated using `is`.)

"wrap" operator

The wrap operator—`v wrap t`—returns `v` if "`v is t`", a new object `o` such that "`o is t`" if "`v is like t`", and otherwise throws an exception. The object `o` shares state with `v`; reading from `o` reads from `v`, and writing to `o` writes to `v`. However, if `v` is changed into an object that is no longer "like" `t`, then reading from or writing to `o` may result in an exception being thrown.

"eval" operator and the "eval" function

ES4 defines `eval(s)` as an operator[29] that has the same behavior as `eval` in ES3: it has access to the lexical environment of its use and can normally introduce new names into that environment. (If the code that is using `eval` is compiled in strict mode, eval is prohibited from introducing new names. This is the one case in ES4 where the run-time meaning of a program changes in strict mode.)

ES4 additionally defines `eval` as a function bound in the global environment that can be called indirectly through that environment (*e.g.*, as `window.eval(s)` in a web browser) or whose value can be read and passed around, stored in other variables and so on, and called as a plain function:

```
var e = eval
...
e(myprog)
```

In both of these cases, the evaluation takes place in the global environment in which the `eval` function is defined.

Any other use of `eval` results in the implementation throwing an `EvalError` at run-time.

arguments

The `arguments` facility is deprecated in ES4.[30] The primary purpose of the facility is to capture rest arguments; ES4 introduces the parameter syntax "`...rest`" for that purpose:

```
function normalize(ref, ...args) {            // args has rest arguments
    for ( let i=0 ; i < args.length ; i++ )
        args[i] /= ref
    return args
}
```

A secondary purpose of `arguments` is to provide access to the current function through the property `arguments.callee`; ES4 introduces the syntax `this function` for that purpose. (ES4 also introduces the syntax "`this generator`" to allow the program to access the current generator-iterator; the `arguments` facility has no provision for this.)

TG1 recognizes that the `arguments` facility will remain in use in some environments indefinitely—notably on the web. Thus ES4 specifies that `arguments` is no longer something like a plain `Object`, but

[29] The syntax must still be that of a function call, so `eval "10"` is illegal.

[30] At the time of writing, the `arguments` facility is the only feature deprecated in ES4.

something like an `Array`, notably in the sense that if `a` is an `arguments` object then "`a is Array`" and "`a instanceof Array`" are both true.

Note: The change in the nature of the arguments object from `Object` to `Array` is considered a bug fix.

"typeof" operator

In ES4, `typeof` returns `"null"` for the value **null** (in ES3 it returns `"object"`), and it returns `"function"` for every object that has a `meta::invoke` fixture, including all the predefined classes.[31]

Note: The change in the handling of **null** is considered a bug fix; it will break some existing content but will in general be an improvement.

VIII. Verification

Two modes ES4 programs can be compiled in *standard mode* or in *strict mode*. The mode is selected by a pragma (`use strict` or `use standard`); the default mode is standard.

Implementations are not required to support strict mode, and implementations that don't should silently ignore the strict and standard pragmas.

Strict With the exception of a detail involving the `eval` operator, strict mode does not change the run-time meaning of programs. Instead, strict mode determines whether a program should be allowed to run at all: strict mode programs are subject to *verification* before execution, and if verification fails then the program is not run.

Verification Verification affects whether a program is considered "legal". Some programs that are legal in standard mode—and might even make perfect sense—cannot be proven safe in strict mode, so are rejected. In addition to static type checking, the strict mode also statically detects and rejects certain programming errors common in ES3 programs (*e.g.* calling functions with the wrong number of arguments and assigning to nonexistent variables).

Strict mode is not intended as a way to alter or improve performance, and strict mode programs may even run slower due to the extra static analysis phase. Strict mode is only meant as a way for programmers to indicate their desire for more extensive and conservative error analysis. All programs, regardless of mode, must execute as though they were performing all dynamic checks.

In practice, compilers can often remove type checks from type-annotated programs that pass strict mode verification.[32]

Type checking

The verifier computes the static type of each expression. Values of unknown type are generally allowed everywhere and will undergo run-time type checks when appropriate. For values of known type, the verifier checks that the type of a value is compatible with its use.

Programs can use the `cast` operator to assign a static type to an expression; at run-time, a check is made that the value is actually of the asserted static type.

Likely semantic errors

The verifier rejects programs that:

- call functions with the wrong number of arguments;
- reference variables that are not known (except in the presence of ES3-style `with` statements);
- reference unknown properties on non-dynamic objects of known type;

[31] The `typeof` operator is simplistic and has limited utility in ES4 programs. TG1 expects that ES4 programs will use `is` instead. Some compatibility aspects of `typeof` are still being discussed.

[32] Nothing stops the compiler from removing type checks from standard mode programs as well.

- add properties to non-dynamic objects of known type;
- write to `const` variables and properties;
- delete fixed properties on objects of known type;
- compare expressions whose types are incompatible;
- reference packages that are not known.

IX. Syntax

ES4 introduces some syntactic shorthand. Here is a summary.

"switch type" statement

The `switch type` statement dispatches on the manifest type of its discriminant value to the first clause that is annotated with a matching type:

```
switch type (v) {
case (s: string) { ... }
case (d: Date) { ... }
}
```

Unlike the normal `switch` statement, clauses do not fall through to the next one, but jump to the end of the `switch type` statement. The statement can be rewritten as a sequence of `if-else` statements, using the `is` operator for the type test. A "default" case is written as `case (x:*) { ... }`.

The names introduced in the clause heads are `let`-bound and scoped in their respective clause bodies.

"try-catch" statement

ES3 provides exception handling in the form of the `try-catch-finally` statement. There can be only one `catch` clause, which will catch any exception. ES4 extends this mechanism by providing for multiple `catch` clauses, each of which may be annotated by a type expression; a thrown value that is a subtype of the annotation on a clause will be caught by that clause:

```
try { ... }
catch (exn: EvalError) { ... }
catch (exn: TypeError) { ... }
finally { ... }
```

Exceptions not caught by any clause are automatically re-thrown, significantly reducing the hassle of handling specific exceptions, relative to ES3.

The extended `try-catch` syntax can be implemented as a syntactic transformation into the old syntax along with a `switch type` statement.

Note: The scope of the name bound by a `catch` clause is the clause's associated block. The 3rd Edition Specification uses language that implies that the name is bound by a `with`-like construct; this detail is observable if the caught value is a function and is called from within the clause's block (because the binding object then becomes the `this` object in the call). TG1 has considered this to be a bug; ES4 requires that the `catch` variable is bound as if by `let`.

Expression closures

If the body of a function definition or function expression is a single expression whose value is returned, then the braces enclosing the body and the `return` keyword may be omitted. This tends to reduce clutter.

```
function square(n) n*n                      // a definition

function bind(f, t)                         // a definition
    function (...args)                      // an expression
        Function.apply(f, t, args)
```

Expression closures can be named, too:

```
var v = function fact(n)
            n < 2 ? n : n*fact(n-1)
```

Note: The scope of the function name bound by a named function expression is the entire function. The 3rd Edition Specification uses language that implies that the name is bound by a `with`-like construct; this detail is observable if the function calls itself recursively because the binding object then becomes the `this` object in the call. TG1 has considered this to be a bug; ES4 requires that the name is bound as if by `let`.

Array comprehensions

`Array` literals can be written as comprehensions taking values from iterators, with the keyword `for` following an initial expression inside the square brackets. An array containing the first 10 squares:

```
[i * i for (i in naturals(10))]
```

(Here, `naturals` is an iterator providing natural numbers up to its argument.)

The variables that take on iterator values are bound locally in the comprehension and can be typed. Iterators can be nested and can have final conditions. The keyword `each` can be used to iterate over values rather than names.

```
[i * j for (i in naturals(8))
         for (j in naturals(8))
           if (i != 0 && j != 0)]

[i * i for each (i in [1, 3, 5, 7])]
```

Destructuring assignment and binding

The left-hand-side of the = operator may be a record or array pattern, signifying that the pattern on the left should be used to extract properties from the value on the right. Some simple cases:[33]

```
({x, y} = {x: 42, y: 37})          // x and y are updated
({x: a, y: b} = {x: 42, y: 37})    // a and b are updated
[x,y,z] = [1,2,3]                  // x, y, and z are updated
```

Note the similarity in syntax between destructuring patterns and the literals that create the objects being destructured.

Destructuring can be used in binding forms like `let`, `const`, `var`, parameter lists, and binding clauses of `switch type` and `catch`:

```
let {x, y} = {x: 42, y: 37}        // x and y are bound and initialized

function f( { x, y } ) ...         // ditto

switch type (obj) {
case ({x, y}: Point) { ... }       // ditto
case ([r,g,b]: RGB) { ... }        // r, g, and b are bound
}
```

Types can be given to the bound names by annotating the pattern:

```
let [x,y,z]:[int] = ...            // x, y, and z are all int
```

[33] The parentheses are required at statement level for syntactic disambiguation. In practice a binding form will most often be used and the parentheses will not be needed.

Patterns can be deep, so the following statement traverses a value and binds and initializes the variables `a`, `b`, `c`, `d`, and `e`:

```
let { x: {a, b}, y: [c, d], z: e } = ...
```

The apparent syntactic clumsiness of pattern annotation is alleviated by the use of type definitions, which are then shared among the value producers and value consumers:

```
type T = { x:{ a:int, b:double }, y: Array, z: string }

function g(): like T {
    return { x: { a: 3, b: 7.0 }, y: [1,2,3,4,5], z: "foo" }
}

let { x: { a, b }, y: [c, d], z: e }: T = g()
```

"this function"

The value of the expression `this function` is always the function object in which the expression is evaluated. It saves the programmer from naming every function, knowing the name of the function in which the code is embedded, or using the deprecated phrase `arguments.callee`.

"this generator"

The value of the expression `this generator` is always the generator-iterator object created by the initial call to the generator in which the `this generator` expression occurs.

"type" The expression `type T`, where `T` is a type expression, evaluates to a type meta-object representing `T`.

Suffixes Suffixes on numeric literals denote their type: `-7i`, `1u`, `37d`, `14.5m`.

Triple-quoted strings

Strings that are triple-quoted (starting and ending with a triplet of quotes) can contain line breaks and unquoted quote characters.

```
    """this string starts here

it has a "blank" line before this one
and ends here:"""
```

Backslash-newline in string

A common extension to ES3 syntax is to allow strings to be broken across several source lines by allowing a backslash character followed by a newline to read as nothing. ES4 standardizes this behavior.

String indexing

A character can be fetched from a string using bracket syntax, as for `Array`, no longer requiring an explicit call to the `charAt` method:

```
    "ecmascript"[3]       // evaluates to "a"
```

Slicing `String`s, `Array`s and `Vector`s can be *sliced* (as with their `slice` methods) using an extension of bracket syntax, where colon-separated fields denote the start, the end, and the stepping:

```
    "ecmascript"[5:2:-1]   // evaluates to "sam"
```

Slices are just values; they cannot be used on the left-hand-side of an assignment operator.

Readable regular expressions

Regular expressions are notoriously hard to read, and ES4 allows them to be written on multiple lines, to use whitespace for readability, and to contain comments, if the `x` flag is present. Submatches and back-references can also be named, and there are provisions for Unicode character classes. Here's a simple URL parser:

```
/# The protocol is an alpha name followed by colon and double slash
 (?P<protocol> [a-z]+)
 :
 \/\/
 # The host is two or more dot-separated alpha names
 (?P<host> [a-z]+ (?: \. [a-z]+ )+)
 # The path is optionally present
 (?P<path> (?: \/ [a-z]+ )* \/? )
/x
```

The match result's `protocol`, `host`, and `path` properties will hold the submatches.

Semicolon insertion

Two changes are made to the rules for automatic semicolon insertion: A semicolon is now inserted following the final right parenthesis of the condition of a `do-while` statement; and a newline is allowed between a `return` statement and the expression whose value is returned.

Note: The change in semicolon insertion following `return` may create backward compatibility problems, which are being investigated.

Trailing commas

A trailing comma is now allowed in object literals: `{a:10,}`.

Keywords ES4 allows the use of keywords in contexts where they were prohibited in ES3, *e.g.*, as field names in objects: `{function:37}`, `o.try`, and `ns::catch` are all legal.

Note: The purpose of this change is to make integration with host environments simpler. Most references from ECMAScript programs to host data use property access syntax. When keywords are allowed as property names fewer restrictions are placed on the naming of properties in the host environment, and the host environment interface will not be affected when new keywords are added to ECMAScript.

Operators ES4 allows the use of operator symbols as names in a few contexts, *e.g.*, `intrinsic::===`. For every operator there is a global intrinsic definition whose value is a generic function (multimethod) that implements the operator behavior. Programs can pick them up as names:

```
new Map.<int,string>( intrinsic::===, function (x) x )
```

The operator symbols can also be used as names in function definitions, so extending a global operator generic function with a new method for user-defined types is straightforward:

```
generic intrinsic function ===(a: MyType, b: MyType)
    a.getID() === b.getID()
```

E4X ES4 does not incorporate ECMA-357, "ECMAScript for XML", but it does make the E4X syntax reserved for future use. (ES4 also reserves the syntax for the "`default xml namespace`" pragma and requires that the `typeof` operator return `"xml"` for XML objects.)

Unicode ES4 incorporates Unicode version 5 (ES3 was built on Unicode version 2) and provides the ability to directly encode Unicode characters outside the Basic Multilingual Plane with the `\u{...}` syntax, which can be used in all contexts where the existing `\uXXXX` syntax can be used.

ES4 also requires that Unicode format control characters (class Cf) not be stripped from program text.

Note: Disallowing the stripping of Cf characters is a bug fix. ES3 requires Cf characters to be stripped, but it turns out that the only implementation that does strip them receives many complaints about the behavior from its users, who find it convenient to place Cf characters in strings and regular expressions.

Linefeed ES4 requires every implementation to treat CR, CRLF, and LF equivalently as single line terminators, and to transform each of these sequences into a single LF character in contexts where the character value is observable by the program.

X. Predefined Classes and Objects

Changes The predefined classes and objects have undergone substantial changes from ES3, though most of the changes are visible in their definition, not in their functionality: the predefined classes and types are all expressed as ES4 code.

There are several new classes, some new global functions, and some new methods in existing classes.

Note: There are not many new methods, and they are marked DontEnum and thus are unlikely to upset existing code. However, programs that explicitly test whether an object has a property with the name of the new method will find that it does, and programs that use such a test to determine whether to install their own methods may experience that the value given to the property in ES4 is not compatible with their own.

Self-hosting All the predefined types in ES4[34] are in general described as types in the language itself. So where *e.g.* `Array` was described as a collection of methods and properties in the 3rd Edition Specification, it is described as an ES4 class in the 4th Edition.

Predefined objects— the `Math` object, the global object, and all the top-level functions, constants, and variables—are also described in the language.

When certain primitive facilities are not effectively describable in the language, the Standard resorts to other mechanisms, usually prose or traps to the language processor part of the reference implementation.

Note: The constructor functions in ES3—`Object` and so on—are stored in global variables. In ES4 they are stored in global constants. This may break programs that try to give new values to global types, but at the same time, ES3 is not consistent—though the ES3 user program may have replaced `Boolean`, say, the implementation will still pick up the original implementation of `Boolean` and use that in some cases. Sometimes the 3rd Edition Specification sanctions this (with the phrase "The [[Prototype]] property of the newly constructed object is set to the original `Boolean` prototype object, the one that is the initial value of `Boolean.prototype`"); other times not. TG1 has decided that the best way to make the language consistent is to make the predefined types (or indeed any types) not replaceable.

Numbers There are new numeric types: `byte`, `int`, `uint`, `double`, and `decimal`. These were described earlier in the chapter "Behavior".

Vector `Vector` is a new parameterized class that provides dense, 0-based, monotyped, bounds-checked, and optionally fixed-length arrays. `Vector` provides the same methods as `Array`, but in a typical implementation `Vector` will be faster and use less space, and the cost in convenience is slight.

```
function dot(xs: Vector.<double>, ys: Vector.<double>): double {
    let result = 0d
    for ( let i=0, limit=xs.length ; i < limit ; i++ )
        result += xs[i] * ys[i]
    return result
}
```

[34] Consult the library definition for the full list.

Map

`Map` is a new parameterized class that maps arbitrary keys to arbitrary values, with type constraints on both keys and values. Effectively, it is a hash table. It can be customized through the use of program-supplied hashing and equality testing functions.

```
function identityFactory() {
    let ids = new Map.<*,int>
    let id = 0
    return function (obj) {
        let probe = ids.get(obj)
        if (probe is AnyNumber)
            return probe
        ids.put(obj, ++id)
        return id
    }
}
```

Static generic helper methods

Most methods on the ES3 types `Array`, `Function`, and `String` are *generic*: their `this` object does not have to be of the type `Array`, `Function`, or `String`, respectively, and the methods can be transferred to other kinds of object. ES4 provides counterparts for all these methods on the respective class objects (and also on `string`) so that an explicit transfer of the method is not necessary. All methods take the "receiver" object as the first argument. For example,

```
string.slice(obj, 10, 20)
Function.apply(obj, thisObj, myArgs)
```

Static generic methods, like all static methods, are also not subject to being overridden by subclasses, so programs that use them can always depend on their behavior.

Early binding, static type checking, and predictable behavior with "intrinsic"

ES3 programs are highly dynamic: prototype methods can be changed, ad hoc objects can be constructed easily, and data are converted to fit the signature of a method more often than not. Sometimes this is not what is needed, and ES4 provides a cluster of facilities to enable early binding to fixture methods (and properties) in instances of predefined classes, non-converting type checking in those methods, and facilities for preventing injection of unwanted methods.

Public methods in ES4 are all `prototype` methods for compatibility with ES3. ES4 also provides *fixture* methods corresponding to the public methods; fixture methods are immutable and can at most be overridden in subclasses. The fixture methods in the predefined classes are defined in the namespace `intrinsic`. Programs can use the `intrinsic` qualifier in the method call to bypass the public definition:

```
s.intrinsic::charCodeAt(10)
```

Or programs can open the `intrinsic` namespace in a scope, in which case `intrinsic` bindings will be preferred:

```
{
   use namespace intrinsic
   s.charCodeAt(10)         // finds the intrinsic one
}
```

The `prototype` methods usually perform type conversions; the `intrinsic` methods require that types actually match (remember that all numeric types convert to `double`):

```
intrinsic function charCodeAt(pos: double = 0): double
    string.charCodeAt(this, pos);
```

Run-time type checking will cause an error if `intrinsic::charCodeAt` is called with a non-numeric argument.

In the expression `s.intrinsic::charCodeAt(n)`, if the type of the receiver object `s` is known (usually because it comes from a variable that has a type annotation, or is returned from a function that has a type annotation), then strict mode verification will reject a call to `intrinsic::charCodeAt` with a value for `n` known not to be a number, and the compiler can in general perform *early binding* to the method. The known type of the receiver lets the compiler deduce the layout of the type, and any method call on the receiver to a fixture method (like any `intrinsic` method) can use a constant offset into the class to find the method, making for a faster call.

Finally, a method that is defined as `final` cannot be overridden by a subclass (and therefore has known behavior if the type of the receiver object is known), and an implementation can inline it if it can early-bind it.

There is nothing magic about the namespace `intrinsic` that enables all this machinery (which is implementation-dependent in any case); normal programs can make use of it as well, with their own namespaces.

Immutable globals

In ES3 it is possible to redefine the values of the global variables `undefined`, `NaN`, and `Infinity`. These bindings are constant in ES4. (`NaN` and `Infinity` are still `double` values, but they convert correctly to `decimal`.)

Note: This change is considered a bug fix.

Reflection

The *reflection system* provides a reflection of the type system. Every type in the language can be reflected as a *type meta-object*; each meta-object is an instance of a hidden class but it implements one of the predefined reflection interfaces, *e.g.*, `ClassType`.

The global function `intrinsic::typeOf` returns a type meta-object for any value.

The reflection system can sometimes be used to perform operations that are otherwise inexpressible. For example, the `is` operator can have only a type expression on the right-hand-side and therefore cannot test a value against an arbitrary type object. But if `v` is the value and `t` is the type, then

```
intrinsic::typeOf(v).intrinsic::isSubtypeOf(t)
```

does just that. Similarly, to construct an instance of a structural type `t` that's known only at run-time, the type meta-object's `construct` method can be used:

```
t.intrinsic::construct(3, "foo")
```

ControlInspector

`ControlInspector` is a new parameterized class that lets user code place annotations on the continuation (essentially, the program stack) and later step through those annotations. It can be helpful for debugging tools, security mechanisms, and similar introspection tasks.

Date

The `Date` class has been upgraded in several ways. Chiefly, there are methods for producing and consuming ISO-8601 format time strings, and `Date` instances keep track of their age with nanosecond precision and can be used as low-cost high-precision timers.

JSON JSON[35] is a popular, lightweight, language-independent, textual representation for structured data. It uses a subset of ES3 literal syntax. ES4 provides serialization of data to JSON format through the `toJSONString` protocol and deserialization through the static method `string.parseJSON`.

DontEnum The `propertyIsEnumerable` method (defined in `Object` and available on every object in the system) is extended to take a second, optional argument, whose value if supplied should be a `boolean`. The property named by the first argument has its enumerability status set to the value of the second (so if the second argument is **true** then the DontEnum bit is set to **false**, and vice versa).

The typical use case is to add a method to an object and designate it DontEnum so that it will not be subject to enumeration by unsuspecting `for-in` loops:

```
o.myMethod = function () ...
o.propertyIsEnumerable("myMethod", false)
```

Math The methods on the Math object will return results in the representation of their arguments, if that is possible and reasonable, and not always convert to `double`. For example, `Math.pow(2,2)` will return the `int` value 4, not the `double` value 4.

The constant properties on the `Math` object—`PI`, `E`, and so on—are still the `double` values required for backward compatibility with ES3, but eponymous properties are defined on the `decimal` and `double` classes as well, with appropriate precision for each representation.

Similarly, the constant properties on the Number object—`NaN`, `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, `MAX_VALUE`, and `MIN_VALUE`—are also defined on `decimal` and `double`, with appropriate precision for each representation.

Regular Expressions

Regular expressions have been compatibly extended in ES4 with facilities for clarity (ignoring whitespace and comments); matching only at the current location in the input; giving names to submatches and referencing those names in backreferences and in the match results; Unicode character classes; and subtraction and intersection of character sets. Finally, two minor fixes from existing implementations have been incorporated: `RegExp` objects can be called as functions (equivalent to invoking their `exec` method), and unescaped slashes can occur inside character sets.

hashcode A global function `intrinsic::hashcode` computes a hash value for any value, where the only requirement is that two objects have the same hash code if they are equal by the `===` operator.

global The global object is now available through an intrinsic top-level constant binding called `global`.

uint-specific operations

A number of global functions are defined in the `uint32ops` namespace: arithmetic (`add`, `sub`, `mul`, `div`, `mod`, `neg`), parallel bit operations (`and`, `or`, `xor`, `not`), and shifts and rotates (`sll`, `srl`, `sra`, `ror`, `rol`). They take `uint` operands and return `uint` results, and they are intended for use in those few programs that need to operate directly on such values without normal overflow handling. The intent is that some implementations will inline them and provide high performance.

[35] http://www.json.org

Change log

2007-10-23 *Title*: Added the word “proposed”. *Introduction*: Added paragraph about majority/minority views. *Features at a Glance*: Removed “modern” as describing class-based OOP. Clarified the use of prose in the 4th Edition Specification. *Change log*: Added this section.

2007-10-22 Initial release.