Ecma/TC39-TG1/2007/045

Evolutionary Programming and Gradual Typing in ECMAScript 4[1] (Tutorial)

30 November 2007

Lars T Hansen, Adobe Systems
(lhansen@adobe.com)

Introduction

ECMAScript 4 (ES4) provides a range of facilities for *evolutionary programming* – evolving a program in stages from a simple script to an ever larger and more reliable software system. The most important facility for evolutionary programming is the *gradual type system*; also important are *namespaces* and *packages*, *union types*, *generic functions*, and *reflection*.

Evolutionary programming has two aspects: making the code more robust, and working around robustness that gets in the way but cannot be removed.

The robustness of a program is improved by adding invariants to the program, or equivalently, restricting the range of its behavior. Starting with an untyped program built on ad hoc extensible objects and hand-coded data validation—a typical ES3 program—we can apply *structural types* and *type annotations* at key points to make validation faster and more reliable. We can also apply structural types to objects in order to *fix* their properties, thereby providing integrity and making type checking and data validation even faster. We can rewrite ad hoc typed objects as instances of *classes* when a new level of integrity is required, and when program-wide protocols are introduced the classes can even be constrained to match separately defined *interfaces*. Finally, we can use *packages* to hide implementation details, and we can tag functions, types, and variables with *namespaces* to prevent name clashes.

Working around robustness, on the other hand, usually means creating facilities that allow dissimilar aspects of the program—often authored by different programmers—to be treated uniformly. Faced with objects that are not of a desired type, but which behave like that type, we can *wrap* them so that they masquerade as instances of the type. If we have a set of unrelated types that should be treated the same, we can create a *union type* to give the set a name, effectively creating an after-the-fact interface. If we need to define a function that behaves differently on different argument types but won't be able to update its central definition every time a new argument type is added, we can create a *generic function* to which methods, specialized to new argument types, can be added later. Finally, *reflection* allows us to treat types as values and to discover facts about the program itself at run-time.

This tutorial uses a simple library as a running example to illustrate the evolution of a program from the ES3 "script" stage, via various levels of typing and rigor, to an ES4 package with greater guarantees of integrity and better performance potential than the original code. We then look at how union types, generic functions, and reflection can be used to work with a library whose code we can't modify.

Example

Our example is "Mlib", a simple library for a browser-embedded electronic mail facility. Mlib has three public functions, `send`, `receive`, and `get`. The `send` function accepts a single mail message and sends it to a server. The `receive` function retrieves a single mail message from the server, or a special "no mail" object if there are no messages waiting. The `get` function returns a sent or received message by its ID; the

[1] Based on the 2007-10-23 Proposed ECMAScript 4th Edition Language Overview, the (unpublished) 2007-09-20 Library Specification draft, and the ES4 wiki; see http://www.ecmascript.org/. This paper uses the term "ES4" to denote the Proposed ECMAScript 4 language. Details given in this paper are subject to future adjustments as ES4 evolves.

message may be cached locally or retrieved from the server. (We gloss over details of configuring Mlib with information about the server, among other things.)

Since Mlib is a library the public functions need to perform data validation on both the library client's data and data retrieved from the server.

There are two important data types. A *message* is an object with fields "`to`" (an array of addresses), "`from`" (an address), "`subject`" (a string), and "`body`" (a string), and "`id`" (a nonnegative integer). An *address* is an object with fields "`at`" (an array of two strings: the user name and the host name) and "`name`" (a string).

Version 1 of our program—shown on the left in Listing 1, below—makes use of simple ES4 facilities like `uint` and expression functions, but is otherwise ES3 code.

Structural Types for Data Validation

There is substantial noise in Version 1 from the data validation, so let's get rid of it by *annotating* the parameters of the API functions with constraints that perform the validation for us. ES4 provides *structural types* to describe values: record types and array types, which are all we need for the moment, but also function types and union types, which we will see later. There are also many predefined types in ES4, and our program will use `string` and `uint`. The new program is Version 2, and it is shown on the right in Listing 1, below.

Version 2 defines the types `Addr` and `Msg`. These are structural *record types* that are pretty much self-describing: they look like object literals, but with type expressions in the value positions instead of the usual value expressions. Version 2 also uses structural *array types*, which look like array literals but with type expressions instead of value expressions.

The key facilitating feature in Version 2 is the type operator "`like`", which is used to perform shape testing of data. For reasons of backward compatibility with ES3, the type of an ES4 object literal like this:

```
{ to:      [{at: ["es4-discuss","mozilla.com"], name: "ES4 forum"}],
  from:    {at: ["lhansen","adobe.com"], name: "Lars"},
  subject: "ES4 tutorials available",
  body:    "...",
  id:      10 }
```

isn't `Msg`, but rather the empty object type `{}`, since the properties introduced in the literal are all deletable. So if we want the `send` function to accept such a literal then the type constraint on the `msg` parameter must be that `msg` is `like Msg`; without `like` the program would signal an error at run-time.

The result of a `like` test only holds until the datum is modified (just as is the case for the manually written verification code in Version 1). No error is signalled even if the datum is later modified in a way that would contradict the result of the earlier shape test. That is, the `like` test is really just an assertion, testing that the datum looks a certain way at a certain time.

There are other, subtle changes in Version 2. The use of structural types has improved `handleMessage`: where Version 1 would inspect any `result` field in the reply from the server to check for a "no data" return, Version 2 requires the `result` field to be a `string` as well. (The operator "`is`" tests whether its left operand is an instance of the type in its right operand.) On the other hand, the type annotation on the parameter `n` to the function `get` is weaker than the old check: the old check would reject values that were not already `uint`, whereas the new annotation allows any number type to be silently converted to `uint`. (Annotations and conversions are discussed in greater detail later.)

```
// Version 1

function send(msg) {
  validateMsg(msg)
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

function receive()
  handleMessage(-1)

function get(n) {
  if (!(uint(n) === n))
    throw new TypeError
  if (n in database)
    return database[n]
  return handleMessage(n)
}

var database = []

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (typeof msg != "object")
    throw new TypeError
  if (msg.result == "no data")
    return null
  validateMsg(msg)
  return database[msg.id] = msg
}

function validateMsg(msg) {
  function isObject(v)
    v != null && typeof v == "object"

  function isAddress(a)
    isObject(a) &&
    isObject(a.at) &&
    typeof a.at[0] == "string" &&
    typeof a.at[1] == "string" &&
    typeof a.name == "string"

  if (!(isObject(msg) &&
        isObject(msg.to) &&
        msg.to instanceof Array &&
        msg.to.every(isAddress) &&
        isAddress(msg.from) &&
        typeof msg.subject == "string" &&
        typeof msg.body == "string" &&
        typeof msg.id == "number" &&
        uint(msg.id) === msg.id))
    throw new TypeError
}
```

```
// Version 2

type Addr = { at:   [string, string],
              name: string }
type Msg  = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
              id:      uint }

function send(msg: like Msg) {

  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

function receive()
  handleMessage(-1)

function get(n: uint) {


  if (n in database)
    return database[n]
  return handleMessage(n)
}

var database = []

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null
  if (msg is like Msg)
      return database[msg.id] = msg
  throw new TypeError
}
```

Listing 1 The initial ES3-style program (Version 1) and the same program using ES4 structural types to perform data validation (Version 2)

Alert readers will have noticed that the annotation on the function `send` in Version 2 requires `msg` to contain a valid `id` field whose value will be overwritten immediately. It would be more natural just to require the argument to `send` to have `to`, `from`, `subject`, and `body` properties. For the sake of brevity, we'll call the type with these four fields `Msg0`, and we will use it to constrain the parameter to `send`. Making this change yields Version 2b, shown on the right in Listing 2.

```
// Version 2

type Addr = { at:   [string, string],
              name: string }




type Msg  = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
              id:      uint }

function send(msg: like Msg) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

...
```

```
// Version 2b

type Addr = { at:   [string, string],
              name: string }
type Msg0 = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
type Msg  = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
              id:      uint }

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

...
```

Listing 2 Using structural types (Version 2), and a relaxed structural interface to `send` (Version 2b)

Integrity Through Structural Types 1: Fixtures

One obvious weakness in Versions 1 and 2 is that data in the database are shared between the library and its client, and the client can—accidentally or maliciously—modify the message structures of messages both sent and received: it can remove fields, or it can store data of undesirable types in those fields. What we want are *fixtures*: fields of the objects that can't be removed, and whose types are known and enforced.

The library can enforce that the message objects it stores and returns always have the fields described by the type `Msg`. The key is always to create objects that have the type `Msg` (that aren't merely `like Msg`). We can implement that strategy by modifying the functions `send` and `handleMessage`, as shown in Version 3a (on the right in Listing 3, below).

The function `copyMsg` picks the untyped message apart (using the *destructuring forms* that are new in ES4) and then constructs a new object that has the desired fixtures, creating a `Msg` that contains `Addr` objects.

An object can be given a structural record type in two ways. The type can be appended to the initializer as an annotation: `{ at:..., name:... }: Addr`. The function `newAddr` inside `copyMsg` uses this technique. The type can also be the type argument of the `new` operator: `new Msg(...)`. The function `copyMsg` uses this technique. (The additional arguments to `new` are used to initialize the structure, and must be in the same order as the fields are in the type definition.) Both can accomplish the same thing, but sometimes one is more convenient than the other. In particular, the annotation syntax affects the deep structure of the literal: If the literal has nested object (or array) initializers, a nested record (or array) type will create fixtures in the nested objects as well as at the outermost level.

The situation is similar for array types, but with a twist. The type `[int]` means an array whose elements are all restricted to `int`. The type `[int,int]`, however, means an array whose elements 0 and 1 are restricted to `int` (we call this type of an array a *tuple*). In other words, the case with one element is special. The `new` operator can be used here too, as in `new [int](3)` to create an array of `int` of initial length 3.

```
// Version 2b

...

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

...

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null
  if (msg is like Msg)
    return database[msg.id] = msg
  throw new TypeError
}
```

```
// Version 3a

...

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = copyMsg(msg)
}

...

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null
  if (msg is like Msg)
    return database[id] = copyMsg(msg)
  throw new TypeError
}

function copyMsg(msg) {

  function newAddr({at:[user,host],name})
    ({at: [user,host], name:name}: Addr)

  function mapAddrs(addrs) {
    var dst = new [Addr]
    for (var i=0; i < addrs.length; i++)
      dst[i] = newAddr(addrs[i])
    return dst
  }

  var {to,from,subject,body,id} = msg
  return new Msg(mapAddrs(to),
                 newAddr(from),
                 subject,
                 body,
                 id)
}
```

Listing 3 Sharing message data with the client code and the server (Version 2b), compared with copying messages into data structures that have the same structural type but fixed fields with known types (Version 3a)

Digression: The cost of *like*
Unlike an "`is`" type test, whose cost is often constant, the cost of a `like` is roughly proportional either to the size of its right-hand type argument (for record types) or to the size of its left-hand value argument (for array types). If the right-hand type is a record type then each field named in the type must be extracted from the object on the left-hand side and the tested against the expected field type. If the right-hand type is an array type then each array element must be read from the array on the left-hand side and tested against the array's base type. ES4 requires neither field nor element access to be performed in constant time, so the cost of a `like` test can be worse than proportional to the number of values extracted.

On the other hand, the implementation can optimize `like` testing in several ways. For example, it can keep a table that records compatibility between types so that individual field testing is not always necessary. In Listing 3, if the client code calls `send` on an instance of `Msg0` (and not just an ad hoc object) then an implementation ought to be able to perform the check in constant time.

Digression: When is an object like an array type?
Arrays are very rich objects—they have large method suites and unusual behavior around `length`, construction, and so on—so the conditions for an object to be `like` a structural array type are a little

involved. Generally we say that if an object looks like an array, methods and all, then it is `like` an array type, but the rules on what constitutes a method are loose.

Starting with the simple rules, structually typed array data are instances of hidden subtypes of `Array`. Arguments of type `[T]` are compatible with type `[*]`—instances of the former can be stored in variables annotated with the latter—but in general it is illegal to use an object of type `[D]` where `[B]` is required, even if `D` is a subtype of `B`.

The `like` operator is more permissive. An object `t` of type `[T]` is `like [P]` if every array element of `t` is `like P`. An object `q` of type `Q` (where `Q` is not an array type) is `like [P]` if `q` is `like` the predefined type `ArrayLike` and every array element of `q` is `like P`. In turn, `ArrayLike` is a record type that requires a `length` property of type `uint` and methods corresponding to the intrinsic methods of `Array`, but without constraints on the signatures of those methods.

As a consequence of these rules, `Vector.<T>` is not of type `[T]`, but it is `like [T]`.

Integrity Through Structural Types 2: Wrappers

Another option available to us is to *wrap* the message objects so that their fields can't be removed or given values of different types. The change is in `send` and `handleMessage`, while `copyMessage` disappears. Version 3b is shown on the right in Listing 4, below.

The operator `wrap` checks that its left operand value is `like` the right operand type – throwing a `TypeError` if it is not; thus we can remove the explicit `throw` in `handleMessage` – and then constructs an object that has fixtures like the right operand type. Accesses to the wrapper object result in accesses to the wrapped object as well.

The trap here is that run-time errors may later be signalled by the wrapper if the wrapped object has become inconsistent with the wrapper. That situation can arise if code that has unmediated access to the wrapped object bypasses the wrapper when changing the object. This is the case in our code: the caller of `send` has access to the object wrapped by `send`. Whether wrappers are the right solution depends on how and when we expect the data to change – on purpose, or accidentally; and in the course of a call, or over time.

```
// Version 2b

...

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

...

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null
  if (msg is like Msg)
    return database[msg.id] = msg
  throw new TypeError
}
```

```
// Version 3b

...

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg wrap Msg
}

...

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null

  return database[msg.id] = msg wrap Msg

}
```

Listing 4 Sharing message data with the client code and the server (Version 2b), compared with wrapping messages in data structures that have the same structural type but fixed fields with known types (Version 3b)

Digression: Wrapping on entry
One idiom that we're not using in our example is wrapping a data structure when a library API function is called but being careful not to retain the wrapper after the call returns. Suppose we add a `validateAddresses` method and an internal `validateAddress` helper method:

```
    function validateAddresses(as: wrap [Addr])
        as.every(validateAddress)

    function validateAddress(a: Addr)
        ...
```

Note how `validateAddress` can annotate its parameter with just the type name – it does not need to use "`like`". Yet the API function `validateAddresses` allows clients to pass ad-hoc data as long as those data are reasonable.

The key is that as long as the library does not retain a reference to the wrapper, there is no chance of later run-time errors if the client code updates the wrapped data structure in an incompatible way.

Digression: The cost of *wrap*
The cost of `wrap` is at least as bad as the cost of `like`, since a `like` test is required before the object can be wrapped. If a wrapper must be created then it will be a new object that has all the fixtures of the type we are wrapping to, but those fixtures may be implemented as getter/setter pairs and allocating them may be more expensive than allocating normal properties. When a getter reads a value from the wrapped object, that value may itself need to be wrapped, which includes another `like` test, and when a setter writes a value, the value will go through a type check on entry to the setter.

Though it would seem that a naïve wrapper implementation turns linear-time operations like inorder tree traversals into exponential-time operations because of the repeated recursive `like` tests, we are saved by the fact that structural types cannot be recursive—the cost of a `like` test with a record type is bounded by the size of the type, and is not related to the size of the structure.

Again, the implementation may be able to optimize away some tests, as outlined above for `like`, and wrappers won't be created for objects that already have the type of the wrapper. Implementations may use representations that are more efficient that getter/setter pairs. Implementations may also be able to discover quickly whether the wrapped object has changed and whether a new `like` test can be avoided.

Integrity Through Structural Types 3: Make it the Client's Problem

So far we have assumed that the library must accept any reasonable data structure and then deal with it as best it can. A more classical approach is to put the burden on the client by requiring the client to construct data of known types. The client code may become more complicated as a result, and in situations where the client code can't be changed it can become impossible to integrate it with upgraded libraries.

Of course the "burden" does not come without benefits. The library becomes simpler and more reliable, because the client guarantees the integrity of the data, relieving the library of that job. Also, compile-type type checking can verify statically that the client and the library are handling the same type, something that may be more difficult with `like`.

Version 3c (on the right in Listing 5) has a new `send` function, and the `Msg0` type is no longer needed.

```
// Version 2b

type Addr = { at:   [string, string],
              name: string }
type Msg0 = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
type Msg  = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
              id:      uint }

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg wrap Msg
}

...

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null
  if (msg is like Msg)
    return database[msg.id] = msg
  throw new TypeError
}
```

```
// Version 3c

type Addr = { at:   [string, string],
              name: string }




type Msg  = { to:      [Addr],
              from:    Addr,
              subject: string,
              body:    string,
              id:      uint }

function send(msg: Msg) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = msg
}

...

function handleMessage(n) {
  var msg =
    JSON.decode(receiveFromServer(n))
  if (msg is like { result: string } &&
      msg.result == "no data")
    return null

  return database[msg.id] = msg wrap Msg

}
```

Listing 5 Wrapping messages in data structures that have the same structural type but fixed fields with known types (Version 3b) compared with making it the client's problem (Version 3c)

Digression: Structural types

A vital point in the preceding discussions is that two structural types are equal when they have the same structure. Their names or defining directives do not matter. (This contrasts with the definitions of *nominal types*, like classes and interfaces, which introduce new, unique, named types.)

For example, the type names `Msg` and `Addr` do not have to be available to the client code, all the client code needs in order to create objects with the right fixtures is to know the structure of the object to be created. After that, the client code can apply its own structural type annotation. A function that constructs `Addr` instances could look like this:

```
    function makeAddr( user, host, name )
      ({ at: [user, host], name } : { at: [string,string], name: string })
```

The function `makeAddr` is the same as the function `newAddr` in Listing 3, above, but `makeAddr` does not have access to the type `Addr` and applies its own compatible type instead. This is not necessarily good programming style, but if Mlib does not export `Addr` it is a facility that gets the job done.

In fact, structural types are all about describing data that are "known" but whose type definition may not be available because it was never written down, or because it was not exported from its defining module, or because the data were generated by a separate program (as when Mlib receives an object from the mail server), or because the data originated in an ES3 program that can't be changed.

We've already seen that structural types can describe objects with named fields and arrays with array elements. They can also describe *functions* and *unions* of other types:

```
    type Mapper = function (*, uint, Object):*
    type AnyNumber = (int | uint | double | decimal | Number)
```

A function type describes the signature of a function: the number of arguments, their types, and the return type.

A union type makes a single type out of a collection of other types, which may be unrelated to each other. There are no instances of union types, so they are only useful for annotations and type testing. Commonly, a function that can operate on numbers has parameters annotated by `AnyNumber`, such as the `atan` method on the `Math` object:

```
    function atan(x: AnyNumber): FloatNumber ...
```

We'll look more at union types later in this tutorial.

A Complete API

We can now complete a typed API to Mlib by providing the API functions with return types.

```
// Version 3a

...

function send(msg: like Msg0) {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = copyMsg(msg)
}

function receive()
 handleMessage(-1)

function get(n: uint) {
  if (n in database)
    return database[n]
  return handleMessage(n)
}

...
```

```
// Version 4

...

function send(msg: like Msg0): void {
  msg.id = sendToServer(JSON.encode(msg))
  database[msg.id] = copyMsg(msg)
}

function receive(): Msg
  handleMessage(-1)

function get(n: uint): Msg {
  if (n in database)
    return database[n]
  return handleMessage(n)
}

...
```

Listing 6 Constraints only on function parameters (Version 3a) and constraints on return values in addition (Version 4)

A key point in Version 4 (on the right in Listing 6, above) is that though we have added return types to `receive` and `get`, we have not changed the types of `handleMessage` or `database`: those are still unannotated. The return types on `receive` and `get` guarantee that these functions return values of the correct type, but since `receive` and `get` obtain the values they return from untyped sources, the implementation will check the types at run-time when `receive` and `get` return.

Digression: About Annotations

So far we have seen annotations used in several contexts: on parameters; on return types; and on the fields of record types and the elements of array types. The meaning of an annotation is always at least an `is` test: when a value is stored in an annotated field, the value is tested against the type as with `is`, and if the test succeeds, the value is stored. (Otherwise a `TypeError` is thrown.) Sometimes, the annotation also implies a conversion: all numeric types (`int`, `uint`, `double`, `decimal`, and `Number`) are interconvertible, as are `string` and `String`, and `boolean` and `Boolean`. In fact, anything converts to `boolean`.

Annotations can also be used on variables, so we could easily imagine this:

```
    var database : [Msg]
```

Yet our programs are correct without annotating `database`, or `handleMessage`, or `copyMsg`. This ability to introduce annotations gradually, and only to the extent necessary to attain the desired level of checking, is key in making ES4 a language for evolutionary programming. The pattern that the preceding examples have used is one of *typed APIs, untyped code*. API methods are fully annotated and therefore perform type checking on input and output; private variables and functions remain untyped until the complexity of a module requires types to be added for additional error checking.

The "typed APIs, untyped code" pattern may be seen as the natural end point for a program, but it may also be just a stepping stone on the path to *fully typed* code, where every variable and parameter has a type annotation, all objects are instances of named classes, and `like` and `wrap` are no longer used. (Of course, typing being gradual in ES4, the client of a module can't usually tell if the code behind a typed API is fully typed, partially typed, or untyped.)

Prototype methods

Let's introduce a new object type, the "`Server`", to hold our methods; initially we will use an ES3 style constructor function. A sketch of the code is shown as Version 6a, on the left in Listing 7, below.

Anyone who has programmed in ES3 knows the problems we'll encounter: The prototype methods are not protected from being changed, overridden, or deleted; code and data shared by the `Server` methods but not meant for public access (like `database`) might need to be hidden somehow; the prototype methods can be taken out of their context and moved to other objects, where they may wreak havoc or it may be possible to trick them to reveal data; forgetting to add "`this.`" in the reference to a method breaks the program in mysterious ways; the list goes on. Finally, though `Server` is a type in a logical sense, instances of `Server` have the uninteresting type `{}`, as discussed earlier, and `Server` cannot be used as a type annotation.

Version 6b, on the right in Listing 7, attempts to solve some of these problems by using `const` methods in the prototype (thus protecting them from being overwritten in the prototype and shadowed in the `Server` instance), and by using a namespace "`sp`" ("server private") to hide prototype methods and instance data that are not meant to be publicly accessible.

Namespaces are special values that qualify names. I can have a definition of `send` in my namespace and you can have one in yours, and they will be entirely distinct. Namespaces are created by `namespace` directives (in Version 6b one is used to create the `sp` namespace), and the name of the namespace is then used to qualify definitions.

A name in a namespace is referenced by prefixing the name with the namespace: `sp::copyMsg`, for example. The `use namespace` pragma opens a namespace in the block scope of the pragma so that the qualification can be omitted as long as the name is unambiguous. The `send` method uses only the name `copyMsg`, but because the namespace `sp` is open the reference is really to `sp::copyMsg`.

So far so good, you say. But Version 6b's namespace `sp` is not protected—other code can open it too, and gain access to our "private" names—and the properties on the Server object itself, `database` and `host`, are in any case enumerable and the namespace value can therefore be obtained through enumeration. We can make those properties DontEnum (ES4 has a facility for that) and we can hide `sp` in a closure, ES3 style. Yet even if we do all that—and end up with code even more complex than Version 6b—the "public" prototype methods can still be used out of their context, and `Server` will still not be a useful type for annotations. Those two problems aren't solvable in our structural world. Instead, we need to look at *classes*.

```
// Version 6a


function Server(host:string) {
  this.host = host
  this.database = []
}



Server.prototype = {
  send:
    function (msg: like Msg0): void {
      msg.id =
        sendToServer(JSON.encode(msg))
      this.database[msg.id] =
        this.copyMsg(msg)
    },
  receive:
    function (): Msg
      this.handleMessage(-1),
  get:
    function (n: uint): Msg {
      if (n in this.database)
        return this.database[n]
      return this.handleMessage(n)
    },
  handleMessage:
    function (n) {
      var msg =
        JSON.decode(
          receiveFromServer(n))
      if (msg is like {result: string} &&
          msg.result == "no data")
        return null
      if (msg is like Msg)
        return this.database[id] =
          this.copyMsg(msg)
      throw new TypeError
    },
  copyMsg:
    function (msg) {
      ...
    }
}
```

```
// Version 6b

namespace sp

function Server(host:string) {
  this.sp::host = host
  this.sp::database = []
}
{
  use namespace sp

  Server.prototype = {
    const send:
      function (msg: like Msg0): void {
        msg.id =
          sendToServer(JSON.encode(msg))
        this.database[msg.id] =
          this.copyMsg(msg)
      },
    const receive:
      function (): Msg
        this.handleMessage(-1),
    const get:
      function (n: uint): Msg {
        if (n in this.database)
          return this.database[n]
        return this.handleMessage(n)
      },
    const sp::handleMessage:
      function (n) {
        var msg =
          JSON.decode(
            receiveFromServer(n))
        if (msg is like {result:string}&&
            msg.result == "no data")
          return null
        if (msg is like Msg)
          return this.database[id] =
            this.copyMsg(msg)
        throw new TypeError
      },
    const sp::copyMsg:
      function (msg) {
        ...
      }
  }
}
```

Listing 7 ES3 style constructor functions (Version 6a), with read-only, namespaced fixture methods on the prototype to provide better integrity (Version 6b)

Classes

A *class definition* is a syntactic shorthand that combines object construction, type definition, and constant method definition. Classes hide private data with implicit namespaces, prevent out-of-context use of methods, and class names can be used as type annotations. A class version of `Server` appears on the right in Listing 8, below.

```
// Version 6b

namespace sp

function Server(host:string) {
  this.sp::host = host
  this.sp::database = []
}
{
  use namespace sp

  Server.prototype = {
    const send:
      function (msg: like Msg0): void {
        msg.id =
          sendToServer(JSON.encode(msg))
        this.database[msg.id] =
          this.copyMsg(msg)
      },
    const receive:
      function (): Msg
        this.handleMessage(-1),
    const get:
      function (n: uint): Msg {
        if (n in this.database)
          return this.database[n]
        return this.handleMessage(n)
      },
    const sp::handleMessage:
      function (n) {
        var msg =
          JSON.decode(
            receiveFromServer(n))
        if (msg is like {result:string}&&
            msg.result == "no data")
          return null
        if (msg is like Msg)
          return this.database[id] =
            this.copyMsg(msg)
        throw new TypeError
      },
    const sp::copyMsg:
      function (msg) {
        ...
      }
  }
}
```

```
// Version 7a

class Server {
  function Server(h:string) {
    host = h
  }

  private const database = []
  private const host

  function send(msg: like Msg0): void {
    msg.id =
      sendToServer(JSON.encode(msg))
    database[msg.id] = copyMsg(msg)
  }

  function receive(): Msg
    handleMessage(-1)

  function get(n: uint): Msg {
    if (n in database)
      return database[n]
    return handleMessage(n)
  }

  private function handleMessage(n) {
    var msg =
      JSON.decode(receiveFromServer(n))
    if (msg is like {result:string} &&
        msg.result == "no data")
      return null
    if (msg is like Msg)
      return database[id] = copyMsg(msg)
    throw new TypeError
  }

  private function copyMsg(msg) {
    ...
  }
}
```

Listing 8 Defining a data type using ES3 style constructor functions (Version 6b) and ES4 style classes (Version 7a)

Comparing the class definition on the right in Listing 8 to the ES3 style definition on the left, there are many obvious differences. Foremost, the class definition has less clutter. Phrases that are explicit in the ES3 version—the extensive use of `this`, the fact that methods are `const`, the fact that there is a private namespace that is open by default—are implicit in the ES4 version. Simplifying further, methods in the ES4 version are defined by the same syntax as are top-level functions, whereas they are defined using a property-value syntax in the ES3 version, obscuring the fact that they are methods.

Operationally, the ES3 and ES4 versions are not quite equivalent:

- The ES4 methods will become properties on the class instance, whereas the ES3 methods are properties on the prototype object. Since the ES3 prototype properties are `const`, however, referencing them on the instance will always result in the method values from the prototype.
- The ES4 methods are DontEnum; the ES3 methods are enumerable.
- The ES4 instance variables—`database` and `host`—are DontDelete and ReadOnly and can neither be removed nor altered after initialization. The ES3 instance variables are neither; if the client code can gain access to the namespace "`sp`" then the instance variables can be altered or removed.
- The ES4 methods are *bound* to their instance. Reading e.g. the `receive` property of an instance of the ES4 class yields a function value which, when invoked, always has the same `this` value: the object from which the method was extracted.
- The ES4 instances can't be extended dynamically with new properties. To make instances extensible, the class definition must have the attribute `dynamic`. (Every class has a prototype object, like in ES3, and the prototype object can always be extended dynamically. A new property on the prototype will be visible to all instances, as in ES3.)

It is possible to express bound methods in ES3, but the idioms for doing so are clumsy. Using ES4 style `const` properties it is possible to make properties DontDelete and ReadOnly, but not at the same time as using the normal ES3 constructor functionality to install a prototype into a fresh object. There is no way in ES3 to make properties DontEnum, and no way to make objects non-dynamic.

In sum, ES4 classes offer better *integrity* at substantially greater *convenience* than do ES3 constructor functions. ES4 programs using classes are therefore more reliable than comparable ES3 programs.

Packaging It Up: Packages and Namespaces

If we go back to our earlier program where all the definitions were global, how can we maintain the privacy and integrity of our internal definitions, `handleMessage`, `database`, and so on. A client program that uses our library could easily[2] have defined names like that itself. We have already seen how namespaces can be used to make similar names distinct. Listing 9 below shows how namespaces and another ES4 facility—packages—can be used to similar effects to hide private names.

Packages, shown in Version 5a, are similar to packages in Java. A package has a name comprised of a list of identifiers separated by "`.`". The package has *public* definitions (those qualified as `public`) and *internal* definitions (all the others), and *importing* a package makes some or all of the public definitions visible in the importing scope. Finally, the full package name can be used to qualify any name. Two snippets of client code demonstrate the use of packages.

Version 5b uses namespaces to accomplish the same result as the packages in 5a. New in this program is the `use default namespace` pragma, which names the namespace that annotates definitions that do not have an explicit namespace annotation: The definition of `Addr` really defines `mail_private::Addr`. Two snippets of client code demonstrate the use of namespace.

Digression: Packages are Namespaces, Simplified

Packages are syntactic sugar for namespaces, and the general effect of a package can be accomplished by using namespaces explicitly. As Listing 9 shows, packages can be more convenient: there are fewer names to manage, fewer directives to write, and `public` stands out clearly regardless of what the package is called, whereas if the namespace mechanism is used then the namespace meant for public use is different from module to module. Packages can also be defined in disjoint chunks of source code, and the compiler takes care of hiding the package's internal namespace; using namespaces directly, this hiding must be handled manually.

[2] On the world-wide web, pretty much anything is bound to happen sooner or later. *Somebody* will run into this problem.

Namespaces are more general than packages, though: they are not just for hiding, but also for tagging names, so that multiple names can coexist easily, or so that names can be given an informative qualification. The ES4 predefined classes could have reserved a single namespace for system use, but reserves several, each with its own purpose: “`intrinsic`” for early-bindable names, “`reflect`” for the reflection API, “`meta`” for system hooks, and “`iterator`” for the interator and generator facilities. The result is not just that they can be combined, but that the reader knows what category of method she’s looking at.

```
// Version 5a

package org.ecmascript.mlib {




    type Addr = ...
    type Msg0 = ...
    type Msg  = ...

    public function send(msg: like Msg0)
      ...
    public function receive(): Msg ...
    public function get(n: uint): Msg ...

    var database = ...
    function handleMessage(n) ...
    function copyMsg(msg) ...
}
```

```
// Version 5b

namespace mlib

{
    namespace mail_private

    use default namespace mail_private
    use namespace mlib, mail_private

    type Addr = ...
    type Msg0 = ...
    type Msg = ...

    mlib function send(msg: like Msg0)
      ...
    mlib function receive(): Msg ...
    mlib function get(n: uint): Msg ...

    var database = ...
    function handleMessage(n) ...
    function copyMsg(msg) ...
}
```

```
// Client code

import org.ecmascript.mlib.*
send( { from: { ... },
        to: [ ... ],
        subject: ...,
        body: ... } )
var msg = receive()
```

```
// Client code (version 1)

use namespace mlib
send( { from: { ... },
        to: [ ... ],
        subject: ...,
        body: ... } )
var msg = receive()
```

```
// Client code (version 2)

org.ecmascript.mlib.send({ from: { ... },
                           to: [ ... ],
                           subject: ...,
                           body: ... } )
var msg = org.ecmascript.mlib.receive()
```

```
// Client code (version 2)

mlib::send( { from: { ... },
              to: [ ... ],
              subject: ...,
              body: ... } )
var msg = mlib::receive()
```

Listing 9 Name hiding: Version 4 modified to hide private names with packages (Version 5a) and namespaces (Version 5b)

Interfaces and Union Types

Going back to our mail server class, if we anticipate that there might be multiple server implementations, we can define an *interface*, call it `IServer`, that describes all the servers. The interface has method signatures, but no method bodies. Each server class—`HttpMailServer`, `TcpMailServer`—then declares that it *implements* `IServer`, and the class definition provides bodies for the methods.

The use of interfaces is shown on the left in Listing 10, below. That listing also illustrates that the interface name can be used as an annotation on variables and parameters, effectively stating that any type that implements `IServer` is an acceptable value. Interfaces should be familiar to anyone who has programmed in Java.

Suppose that, on the contrary, we we did not have the foresight to anticipate multiple server implementations, and we did not define the interface `IServer`. Maybe we even figured there would not be other kinds of servers than `HttpMailServer`. But then another programmer implemented `TcpMailServer`, which is compatible with `HttpMailServer` but defined as an unrelated class, and we would like to use it. The situation should not be unfamiliar to anyone who has worked on a large program.

If we want to use both servers in a program then we would like to treat them the same (since they are compatible), and we would like to use type annotations for greater reliability. We could change the source code—define the `IServer` interface and declare each server to implement that—but that's not always desirable or possible. We could try defining `IServer` as a record type, but as we've seen, record types cannot always capture the subtleties of classes. A union type works better, because it names a collection of unrelated types.

The right column of Listing 10 illustrates how a union type called `IServer` can be defined after the fact and used to annotate variables and parameters, in the same way the interface type `IServer` was used.

```
// server.es

interface IServer
{
    function send(msg: like Msg0): void
    function receive(): Msg
    function get(n: uint): Msg
}

// http.es                                       // http.es

class HttpMailServer implements IServer {        class HttpMailServer {
  ...                                              ...
}                                                }

// tcp.es                                        // tcp.es

class TcpMailServer implements IServer {         class TcpMailServer {
  ...                                              ...
}                                                }

// Client code                                   // Client code

                                                 type IServer =
                                                   (HttpMailServer | TcpMailServer)

var myServer: IServer = ...                      var myServer: IServer = ...
function addServer(srv: IServer) ...             function addServer(srv: IServer) ...
```

Listing 10 Classes implementing an interface (left) and unrelated classes collected as a union type (right)

Generic Functions

Software systems evolve over time and not always in predictable ways, and often consist of components created by several authors. Suppose we'll be builing a mail application that makes use of server components from several sources. All of the servers implement `IServer`, and in addition they all provide a connection status facility. The problem is that the connection status facilities are all different—`IServer` does not define it—and yet we'd like our mail application to use a single API to query a server's status. Among the several ways to solve the problem—editing the source code, wrapping each server object in a façade, writing a global function that tests the type of the server object and performs the appropriate action (illustrated on the left in Listing 11, below), adding a method to each server's prototype object—we can implement the new API non-invasively as a *generic function*.

Generic functions are shown on the right in Listing 11. First we define the generic function in a central location—it is body-less, the actual body is comprised of methods that will be added later. The generic function `stat` takes a single argument that is constrained to be an instance of `IServer`.

Then we add a *method* to `stat` for each server type. The methods are added in a distributed fashion; in our program there is a file containing auxiliary code for each server type (e.g., `httpaux.es` for the `HttpMailServer`), and that file that is the natural home for methods operating on that server. The methods are not aware of each other, and when a new server is added to the system, a new `stat` method is defined along with the server's auxiliary code.

(At this time, generic functions have a limitation: they only dispatch on class, interface, and union types, not on record or array types.)

```
// mailapp.es

function stat(srv: IServer) {
  switch type (srv) {
  case (h: HttpMailServer) {
    ...
  }
  case (t: TcpMailServer) {
    ...
  }
  case (x: *) {
    throw new TypeError
  }
  }
}

// Client code

var srv: IServer = ...
var status = stat(srv)
```

```
// mailapp.es

generic function stat(srv: IServer);

// httpaux.es

generic function stat(h: HttpMailServer){
    ...
}

// tcpaux.es

generic function stat(t: TcpMailServer){
   ...
}

// Client code

var srv: IServer = ...
var status = stat(srv)
```

Listing 11 Adding a type-dispatching function after the fact, as a centralized function (left) and as an equivalent generic function (right)

Digression: multi-way dispatch
Generic functions are more powerful than the `stat` example reveals. Unlike class and prototype methods, which select the concrete method based only on the type of the receiver object, a generic function selects the concrete method based on all the argument types. We can use this in our mail application. Suppose we add support for a a new type of addresses, but only some servers support it directly. The generic function `useServer` (Listing 12, below) tests whether a server can send to a particular address (and otherwise redirects it to a slower proxy server), and is just a multi-way type dispatch.

```
generic function useServer(server, addr);

generic function useServer(server:*, addr:*)
    g_proxyServer

generic function useServer(server: SmtpMailServer, addr:*)
    server

generic function useServer(server: TcpMailServer, addr: LocalAddr)
    server
```

Listing 12 Multi-way type dispatch in generic functions

The generic function provides a default behavior where the arguments are unconstrained ("`*`"); that action is to redirect the message. Then we add the fact that `SmtpMailServer` can handle any address, and that `TcpMailServer` can handle `LocalAddr` type addresses (whatever they may be).

Reflection

The term "reflection" is used to describe a number of loosely related facilities in ES4. Most of these are related to inspecting data and types:

- The `for-in` loop enumerates the "enumerable" properties of an object and its prototypes.
- The `propertyIsEnumerable` method tests whether a property is enumerable, and can be used to change the property's enumerability status.
- The `in` operator tests whether an object or its prototypes contains a property.
- The `hasOwnProperty` method tests whether an object itself contains a property.
- Types are available as run-time values: `Object` is itself an object. Collectively, objects that represent types are called *type meta-objects*. A type meta-object always exports an appropriate, known interface. The most general such interface is `Type`, which provides operations such as `isSubtypeOf` and `isLikeType`. Subinterfaces of Type include `NominalType` (with its own subinterfaces `ClassType` and `InterfaceType`), `RecordType`, `ArrayType`, `FunctionType` and `UnionType`. `Object` implements `ClassType`; the type `{x:int}` implements `RecordType`.
- The global function `typeOf` returns the type meta-object for its argument value.
- The operator "`type`" creates type meta-objects from type expressions: `type {x:int}` yields a type meta-object that implements `RecordType`.
- The operator "`new`" can be applied to arbitrary type meta-objects at run time.

All of the type meta-object interfaces, their methods, and the global `typeOf` function are defined in the namespace "`reflect`".

Reflection is useful mainly when a program is manipulating objects of an unknown or uncertain type, or when types are not available until run-time. As a simple example (Listing 13, below), suppose the mail program has a factory for server components. The `addServer` method accepts an `IServer` class object, but the class will be instantiated by the factory and the factory will need to know how to do this. Recently it became necessary to extend the constructor argument list from two to four arguments, but some server components—some of them not even maintained by us—haven't changed yet. We'll use reflection to discover how to construct a server instance from a class object.

```
function addServer(srvClass) {
  use namespace reflect
  var clsName = srvClass.name()
  for (let m in srvClass.staticMembers()) {
    if (m is FunctionType && m.name() == clsName) {
      let nargs = 0
      for (let a in m.argTypes())
         nargs++
    }
  }
  serverDatabase.push({ class:srvClass, newConstructor:nargs == 4 })
}
```

Listing 13 Using reflection to discover whether a constructor takes four arguments

The `reflect::staticMembers` method on `ClassType` objects returns an iterator across the static members of the class (optionally taking a set of namespaces; the default is the set consisting just of the public namespace). The `for-in` loop extracts all values from this iterator.

Change Log

29-Nov-2007 *Initial version.*

Licensing terms, Conditions, and Copyright