White Paper: Introducing the ActionScript® 4 Language

Bernd Mathiske, Avik Chaudhuri, Krzysztof Palacz, and Basil Hosmer

Adobe Systems Incorporated
AS4@adobe.com

November 1, 2012

© 2012 Adobe Systems Incorporated. All rights reserved.

Adobe ActionScript 4 Language White Paper Version 1.0

This document is protected under copyright law, furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

This guide contains links to third-party websites that are not under the control of Adobe Systems Incorporated, and Adobe Systems Incorporated is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Adobe Systems Incorporated provides these links only as a convenience, and the inclusion of the link does not imply that Adobe Systems Incorporated endorses or accepts any responsibility for the content on those third-party sites. No right, license, or interest is granted in any third party technology referenced in this guide.

This user guide is licensed for use under the terms of the Creative Commons Attribution Non-Commercial 3.0 License. This License allows users to copy, distribute, and transmit the user guide for noncommercial purposes only so long as (1) proper attribution to Adobe is given as the owner of the user guide; and (2) any reuse or distribution of the user guide contains a notice that use of the user guide is governed by these terms. The best way to provide notice is to include the following link. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/.

Adobe, Adobe AIR, ActionScript, and Flash are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. All other trademarks are the property of their respective owners.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users: The Software and Documentation are ''Commercial Items,'' as that term is defined at 48 C.F.R. §2.101, consisting of ''Commercial Computer Software'' and ''Commercial Computer Software Documentation,'' as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

1 Introduction and Motivation

The ActionScript® language has not changed since 2006. At the time of this writing it is still in the same version 3 without any significant updates. But the demands on the Adobe® Flash® platform and on its language have shifted. We now introduce a new version of ActionScript that is oriented at the requirements of larger and more performance-critical applications across a wider range of deployment scenarios than anticipated for its predecessor: ActionScript 4.

The Flash platform has a rich ecosystem into which it deploys programs in a variety of forms: packaged applications with a captive runtime, loaded applications on a standalone Flash Player or on a full Adobe AIR® installation, dynamically loaded applications in web browser plugins. Providing a true cross-platform developer experience across these different scenarios as well as the related ubiquity are primary attractions that give the Flash platform a great competitive edge.

In the following we analyze the prospects of each deployment mode and how it sets the stage for our language design. Next we derive guidance from major technological trends and from our focus audience: game developers. Then we summarize all of the above in high level guidelines for our language design.

1.1 Smartphones: Apps, Resource Constraints

The dominant distribution form of programs on smartphones is ‘‘apps’’, and the availability of Flash Player in mobile browsers is severely limited. Apps are statically compiled and even statically linked. This means that AS4 needs to be well-suited for static compilation and cannot rely on dynamic optimizations to achieve acceptable performance. This requirement is exacerbated by smartphones having relatively weak compute power. Furthermore, power consumption is a major concern and code performance improvement is a common prescription for this, since quicker execution reduces the amount of time any given stretch of code requires power.

For app deployment, we should continue to offer a form of native extensions that can employ functionality specific to one native platform in a non-portable way.

1.2 Tablets: Middle Ground

Initially, tablets are like a large smartphone without a phone function, or more precisely like a large iPod Touch. So they emphasize *apps* over *web* just like smartphones. But we have to be prepared for *web* also. Quite possibly, tablets will supplant large parts of PC install base, and then this will perpetuate the requirements we are facing now for desktops.

Already, we are seeing tablets with physical keyboards, which makes them a lot more similar to laptops. And for people who enjoy virtual keyboards more than physical ones, one can argue that tablets have been similar to laptops already. So either way, the result is pretty much equivalent to a laptop with touch input. And touch input is just arriving on laptops and even regular desktops, too. In other words, the differences between tablet, laptop, and desktop are blurring. As a result of this, we will continue to face devices with PC-class capabilities, in whatever physical shape, and this will remain relevant.

The real question is what artificial restrictions will be imposed on these platforms regarding runtime deployment, application code deployment, and dynamic code generation. Recently, we have experienced a trend towards more close control. Furthermore, Flash Player deployment is also constrained by our own capacity to conduct and support porting efforts, especially in the relatively fragmented mobile space, including tablets. However, for all we know, there will continue to be at least some tablets that admit Flash Player to browsers. Since it is hard to predict market share and constraints concerning all platform owners for

extended periods of time, we must stay nimble. This means keeping support for JIT deployments enabled, even for tablets.

On the other hand, there are more and more cheaper, low-end tablets. This perpetuates the alignment with smart phones. Apps are here to stay as deployment form as well.

So unsurprisingly, we see tablets positioned in the middle between smartphones and desktop, requiring both AOT and JIT deployment. Furthermore, the performance arguments made in reference to smartphones apply typically also to tablets, though lessened by their relatively larger compute and battery capacity.

1.3 Desktop: High End Clients

No matter how successful tablets become, desktops stay relevant. Many uses are better served by their larger form factor and their superior compute power and being mobile is not always required. Furthermore, the monetization of software and services on mobile devices is relatively weak, even though the cross-over point for mobile device numbers trumping PC numbers has been reached.

By definition, the desktop device class carries the high end for compute clients (including workstations). There did not use to be a choice between PCs and mobile devices. Now that there is, the higher investment in a PC implies indications concerning the user's objectives. Why would anyone choose a PC over a tablet or smartphone?

- Performance
- Screen real estate
- Memory
- Storage and storage bandwidth
- Network bandwidth

Today, the deployment of Flash Player is almost unrestricted on desktops. The plugin is virtually ubiquitous. And it requires a JIT to perform well. This gives us the opportunity to put features into our language that provide the ultimate performance experience, fully exploiting dynamic optimization.

For JavaScript, and also for ActionScript until now, dynamic compilation is first of all a compensating measure for inherent performance impediments. Aiming to eventually make performance a competitive differentiator for ActionScript, we now prefer to drive performance from a language design position that lets performance optimizations take off from higher ground.

Today's desktops have such abundance of memory that there is no question that we have to support 64-bit computing. This has to be taken into account for mass data types in ActionScript.

1.4 Servers: Outlook

For various reasons, the Flash platform is a client-centric technology without significant deployment on servers. Among those reasons, insufficient performance, the lack of concurrency, and the resulting poor hardware utilization have to be seen as major impediments. It is expected that these downsides will be mitigated and eventually overcome by our efforts on the client side. At least from a language and VM perspective there should then be no barrier to running (large) AS4 applications on servers. Besides, this also applies to server-side AS4 in cloud computing.

If and when AS4 will be running well on servers, the same arguments that are made today on desktops wrt. performance, memory, storage, and bandwidth will apply, only more so. We can count on 64-bit and

multi-core requirements. Furthermore, our language design should be on a trajectory that can lead to efficient expression of distributed programs.

1.5 Multi-Core and Many-Core

Mobile devices increasingly feature multi-core CPUs and on desktops they are already common place. With today's single-threaded ActionScript or even with AS "workers", this means unused hardware, poor utilization, categorically restricted performance. Furthermore, power efficiency is crucial on mobile devices and performance is a key driver for power savings. Combining these two observations, we cannot afford to leave the available hardware concurrency presented by multi-core CPUs untapped.

With frameworks like Stage3D and Starling, Flash Player is successful in harnessing the power of GPUs for rendering and making it available in a cross-platform fashion. But less graphics-oriented compute tasks are underserved. We need to tap more into the available parallelism on GPUs and GPGPUs to drive performance beyond multi-core capacity.

We are working towards integrating concurrent/parallel programming seamlessly into ActionScript. This already affects earlier features of AS4. We are strongly motivated to facilitate functional programming style. In particular, this includes suppressing mutability of values where possible, i.e. wherever tolerable.

1.6 HTML5: Redividing the Application Space

HTML, CSS, and JavaScript have become more expressive and performant and capable of fulfilling roles that used to be covered by Flash Player alone. Many analysts, bloggers and article commenters claim that the Flash platform will or at least should be displaced *completely*. Yet there is no compelling `technical` reason that reduces the remaining realm of the Flash platform overwhelmingly, and it can grow without serious challenge in areas where HTML/JavaScript will never be able to catch up. JavaScript programming has at least these limitations:

- It scales poorly in program size and in team size.
- IDE support is hindered by dynamic typing.
- Debugging effort is multiplied by the lack of static checking.
- The achievable performance ceiling is limited by dynamic features.
- It is ill-suited for static compilation and thus JavaScript apps often cannot be competitive with native apps.
- It is all too easy to write badly performing code, to fall into performance anti patterns.
- Innovation progress is throttled by standardization.
- Its implementation is fragmented and there are economic incentives for its standardization members and for its implementors to fragment it as well as to reconcile it. At the very least, we can expect its fragmentation to oscillate.

Despite all this, HTML5/JavaScript is here to stay. One cannot not serve the Web! Here it is our task to provide a superior alternative where HTML5/JavaScript is bound to lack. This amounts to a vast space, not just a niche. The Flash platform should be able to continue its role as technological trail blazer.

There is still a large overlap in the application space between HTML5 and the Flash platform and we do not expect it to shrink away entirely. We should be prepared to continue seeing Flash technologies used for general purpose web programming, ads, banners and so on. We should therefore keep a certain amount of

dynamic features to interoperate with web technologies and to provide smooth onramping of web programmers to ActionScript 4. The latter can also be facilitated with gradual typing and type inference.

However, even though we position Flash technologies as a general purpose programming platform, including web programming, our focus of attention has moved on to the following topic.

1.7 Game Development

Gaming is the ideal technological focus area for further developing the Flash platform.

- The Flash platform already enjoys a large user base in web and app gaming.
- This is a fast growing, revenue rich product area.
- Adobe as a creative design company must play in this space, since it is the pinnacle of creative design, posing the greatest challenges.
- The requirements for game programming subsume those of most other disciplines. If we master gaming, then the technology developed for it will be applicable to virtually everything else. This can be seen as trickle down tech transfer from the most demanding area of all.
- Game developers are early adopters, more forgiving wrt. initial problems in newly developed offerings than most other communities.

Besides media asset creation, game programming can generally be categorized into these main areas:

Category	*Example Tasks*	*Typical Programming Styles*
Game Logic	player actions, story line/script, UI, events	object-oriented, imperative
Numeric computation	physics, strategy, constraint solving	functional, imperative
Rendering	3D/2D shading, sprites, video, images, effects	declarative, constructive
Communication	network / distributed programming	object-oriented, reactive

In the short term, and in the scope of this text, we concentrate on the former two of these.

1.7.1 Social Gaming

Social games are booming. The market for them has become very competitive. The effort going into each individual game development is high. Team sizes often reach into several dozen developers, not rarely to more than a hundred. This clearly is programming in the large, which could benefit greatly from better language and tool support.

Frame rates convert into dollars. Here once again, improving Flash Player performance is key.

1.7.2 AAA Gaming

Some game engines run thousands of different tasks in a quasi-concurrent fashion. To allow developers to map these directly to threads way beyond the point of oversubscribing multi-core parallelism, we aim to offer a two-level thread model in a later version of ActionScript.

We also need to leverage all available hardware for numeric computation, whereas rendering is being served sufficiently well with API approaches.

ActionScript game code can be mixed with code written in other languages such as C++ when compiling the latter with FlasCC.

1.8 Conclusions and Goals

From the above, we gather these requirements:

Cross-platform: Cross-platform development remains the main attractor, even though the dynamics of device and OS regulations have shifted our deployment modes in recent year as follows.

Both JIT and AOT: The range of application deployment targets clearly indicates that both JIT and AOT are here to stay for the foreseeable future and that we should stay nimble in that regard anyway. This means that if AS4 is supposed to be a high performance language, it needs to be shaped so that great performance is possible without relying on dynamic optimizations.

Maximum performance: In order not only to catch up with competitors, but to make performance a defendable differentiator for AS4, we have to design it so that virtually nothing is in the way of performance maximization. Especially under the aspect of AOT compilation this means that static typing has to be the default setting and that it has to be absolutely stringent wherever possible unless the developer opts out explicitly (by using the `*` type). In general, we favor making non-dynamic features fast over whatever the consequences for dynamic features are. We also arrange the type hierarchy so that the source compiler can arrange autoboxing. Operations on primitive types do not have inherent control flow (when statically typed). Furthermore we introduce fixed-length arrays, and we plan for multi-dimensional arrays with VM-determined data layout.

Machine types: Primitive data types must translate directly to hardware operand types to provide maximum performance, but also to facilitate mapping C data types to ActionScript via FlasCC, and to enable us to develop a foreign interface. The latter will eventually make native extensions simpler to write and better performing, and it is intended to greatly simplify the implementation of the Flash Platform itself. To these ends, we should also offer user-defined value types that aggregate primitive types.

Hardare utilization: We carefully design the first version of AS4 for the later introduction of high-performance multi-threading and parallelism features, since offering these is necessary to provide competitive hardware utilization.

Programming in the large: Competitive pressures, in particular in game development, drive our developers towards larger efforts that require more coordination and more quality frontloading. This also enforces our choice to support static strong typing. Furthermore, our core language design must be suitable for the introduction of generics (parametric polymorphic typing) in the future. This will enhance expressiveness while maintaining static typing.

Both object-oriented and functional: Large program organization benefits from object-oriented style and a variety of well-known related patterns. Yet we want to promote functional programming style for more algorithmic program parts, especially when these involve concurrency. Reconciling the two styles, we aim at making object-oriented program parts more amenable to functional programing. Concretely this dictates giving immutability preference over mutability whenever possible. To this end we introduce two-phase constructors, which facilitate declaring and guaranteeing truly constant object fields, virtually without any loss of generality wrt. what constructors can express. Besides this feature, we are generally promoting immutability throughout the language to make developers more accustomed to paying attention to it. For example, function parameters are by default immutable in AS4.

Small and large memory: We can future-proof certain mass data types like arrays and strings by allowing them to range over 64 bits while at the same time allowing the VM to choose a space-efficient representation on constrained devices. However, we should also offer the usual 32-bit variants, since small arrays and strings are common.

Supplemental dynamic features: Every static type system has its limits wrt. expressiveness. We should supplement it with select dynamic features and a comprehensive reflection API. But we must be careful

to not compromise our other goals by doing so.

Type inference: Static typing does not always require stating types explicitly. The most suitable types that one might have written can more often than not be discovered by the compiler and inserted automatically. This mitigates the typing effort and visual impact caused by static typing compared to dynamic typing. However, type inference can also be useful even in dynamically typed code contexts, providing opportunistic performance improvements.

Adobe is not only going to update the ActionScript language but also going to renovate the libraries in Flash Player, moving from V11 versions to V12. In both cases, there will be a compatibility break that affects the source code level as well as the binary level. AS3 will not run on V12 and AS4 will not run on V11. AS4 and V12 will be a fresh new reinvention of the Flash platform, not just an incremental upgrade.[1]

However, both the AS4 language and the V12 APIs will look very similar and familiar to existing developers. This is of course intentional, to facilitate migration to our new platform version. Yet we believe that given its more concise and disciplined design, AS4 will be easier to learn than AS3 for many of those who will come to our platform as new developers.

1.9 Overview

To fulfil all of the above goals, it became necessary to drop some features from AS3. These are discussed in the next section. Other features, while kept, have been modified as explained in section 3. In principle, significant portions of AS3 code can be ported to AS4 given this knowledge alone. But we are aware that this is temporarily limited by some valued AS3 features having been removed without replacement. Eventually, there will be powerful substitutes that fit in well with the new language.

The following sections present new features in AS4. We first describe those designated for an initial release (section 4), then we sketch the next wave (section 5). Finally, we give an outlook on features (section 6) that are further out in our schedule. This is to provide a first impression of the overall direction and significance of the AS4/V12 platform.

The appendix contains implementation notes (section A) and a listing of base classes and interfaces (B), which includes core classes in the type hierarchy as well as a new reflection API.

In the remainder of this text we assume that the reader is already familiar with ActionScript 3.

2 Dropped Features Relative to ActionScript 3

In this section we list features of AS3 that are being dropped in AS4, motivate why they are being dropped, and suggest ways of porting AS3 code that relies on those features to AS4.

2.1 The `with` Construct

AS3 supports `with` statements à la ECMAScript 3. Such statements introduce dynamic scoping in the language, as the following example shows.

[1] AS3 and V11 content will continue to run in Flash Player, which will dynamically detect which runtime version to launch.

```
function f(o:Object) {
    var x:int = 6; // declares local variable x
    var y:int = 42;
    with(o) {
        x = 7; // updates o.x instead of x
        y = 0;
    }
    print(x, o.x); // 6, 7
    print(y); // 0
}
f({x : 6}); // passes an object with property x
```

Dynamic scoping has several drawbacks:

- Dynamic scoping is expensive to implement, since it requires maintaining a chain of run-time objects (a.k.a. scope chain) for resolving lexical references.

  In particular, dynamic scoping makes function closures (run-time representations of functions that are passed by value) heavyweight, since a function closure must capture the scope chain that is in effect at the point of function definition.

  But closures are often used to pass handlers in event-based programming.

  Furthermore, going forward, we want to encourage people to use closures (as part of a move to parallel programming using maps, for example).

  Finally, lightweight closures are required to implement our new object initialization protocol (see section 4.2).

- Dynamic scoping also causes the compiler to ignore static errors, thereby requiring dynamic checking to guard assignments to variables, to maintain type safety, and therefore security, at run-time. In other words, if we cannot ensure that a variable of type T will always hold values of type T, then the dynamic compiler / VM cannot safely allocate space for the variable based on T without guarding assignments to that variable; such guards potentially degrade performance.

It is instructive to note that for similar reasons, EcmaScript 5 strict mode makes it a syntax error to use `with` statements.

The main benefit of dropping `with` statements is that we recover lexical scoping, which enables compile-time optimizations.

- Lexical scoping is efficiently implementable by relying on a flat, compile-time map over variables (where variables declared in inner scopes shadow variables declared in outer scopes). This considerably simplifies and improves the performance of name lookup.
- Lexical scoping allows the compiler to eliminate dynamic checking on assignment to variables in scope. This is a big performance win. In fact, it is unfortunate that statically typed AS3 programs do not enjoy this benefit, since this performance win is part of the promise of static typing in general. Indeed the primary goal of static typing is proving, at compile-time, that dynamic typechecking is redundant: statically typed programs do not violate type safety.

A slow, but mostly functional hack for porting code that relies on `with` statements is as follows. Code inside a `with` block for object `o` can be transformed so that every lexical reference `x` is rewritten to a conditional that checks whether `x` can be found in `o` and branches to either `o.x` or `x`. Essentially, this amounts to inlining the lookup logic inside `with` statements.

Note that the keyword `with` is no longer required, although we may want to reserve it for future repurposing.

2.2 Namespaces and E4X

In AS3, namespaces enable a form of dynamic overloading, where different functions with the same name may be defined in different namespaces and called by supplying the namespaces at run time. Namespaces were introduced as part of the EcmaScript 4 effort, which was eventually abandoned (so that namespaces never made it into other JavaScript implementations).

Namespaces unify a lot of concepts in AS3: packages, access control modifiers, API versioning. However, arguably all of those concepts can be readily implemented without namespaces. At the same time, there do not seem to be existing applications that exploit the full expressive power of namespaces: for example, namespaces could provide a powerful mechanism to do capability-based programming, but no such framework seems to be in use by the AS3 developer community. As such, namespaces seem like ''a hammer looking for a nail'' that everybody pays for but only few, if at all, use.

The design and implementation of namespaces in AS3 suffer from many problems:

- When combined with static binding of function calls to function definitions based on static types, namespaces have several unintuitive implications, as the following example shows.

```
namespace N1; namespace N2;
class A {
  N1 function f() { print("in A"); }
}
class B extends A {
  N2 function f() { print("in B"); }
}

use namespace N1; use namespace N2;
// adds {N1,N2} to the set of possible qualifiers for every lexical reference

var b:B = new B();
b.N2::f(); // in B
b.f(); // in B
// desugars to b.{N1,N2}::f(), which early binds to b.N2::f() because b:B

var a:A = b;
a.N1::f(); // in A
a.f(); // in A
// desugars to a.{N1,N2}::f(), which early binds to a.N1::f() because a:A

var c:* = ... ? a : b;
c.f(); // in B
// desugars to c.{N1,N2}::f(), which late binds to c.N2::f() because c is a B-object
// this means that we cannot infer c:A
```

  Thus, namespaces preclude type inference for optimization: any precision lost when doing type inference may cause semantic unsoundness, i.e., may change the behavior of the program. Unfortunately, in an object-oriented language, subtyping is a key characteristic, so losing precision via subtyping is common and in fact desirable: it is not reasonable to require tracking exact types.

- Furthermore, namespaces make references heavyweight: every reference consists of not just an identifier but, in the worst case, a set of namespaces that need to be resolved at run time; such references may be ambiguous (i.e., they may resolve to different definitions), and throwing errors when they are ambiguous requires *all* definitions that possibly match to be considered during name resolution (as opposed to just

finding *some* definition that matches).

- Finally, it is painful to specify, understand, and implement namespaces. For example, it is tedious to work out what happens when namespaces are themselves defined in namespaces, which are in turn "opened" in some order (via `use namespace`; see the AS3 language specification for details on this and other issues. Not surprisingly, the AS3 compiler has numerous bugs around namespaces. Furthermore, the VM has several notions of names, including multinames and run-time qualified names, that add subcases to several core bytecodes; see the AVM2 overview for details. (In fact, it is a fair claim that almost nobody inside Adobe enjoys a full understanding of the internals of namespaces, as evident from numerous misunderstandings that keep coming up in various discussions on the topic: a situation that is untenable going forward.)

The main benefits of dropping namespaces are:

- huge reduction in complexity in the language specification and the VM implementation;
- performance improvements (both in time and in space) due to streamlining of names and the process of name lookup;
- soundness of type inference, i.e., the run-time semantics of the language does not depend on the precision of compile-time type information (which is crucial in a language with dynamic typing).

As long as a function defined in a namespace is always referenced by explicitly mentioning the namespace (or, all references to it are statically bindable), the namespace can be simulated by an extra parameter taken by the function and a corresponding extra argument passed to every call of the function.

Note that the keywords `namespace` and `use` become redundant, as do the operator `::`.

In AS3, packages are encoded with namespaces (with package "imports" encoded by namespace "uses"), and access controls such as `public`, `protected`, and `private` are also encoded with namespaces. In AS4, packages and access controls are primitive concepts, as in Java and C#.

See section 3.1.4 for more detail on packages and access control.

Configuration constants, which dictate conditional compilation, have thus far been defined in a special namespace `CONFIG`. The AS4 syntax for conditional compilation is explained in section 3.1.1.

Without namespaces, E4X cannot stand on its own, so we remove it. Consequently, we also remove the associated `XML`, `XMLList`, and `QNames` classes. Removing E4X also means removing a lot of cruft from the syntax (see section 2.15).

2.3 Global Object and Package-Level Variables and Functions

There is no notion of a global object in AS4, nor are there any package-level variables or functions. Instead, there is a *main class*, which is anonymous and is expandable. It is also prepopulated with a few convenience methods such as `trace()`.

See section 3.1 about program structure for further details.

2.4 Functions That Implicitly Bind `this`

In AS3, all functions, including non-method functions (e.g., anonymous functions, nested functions) implicitly take an additional `this` parameter.

- Thus, all functions can be used as methods, by attaching them (explicitly or implicitly) to objects. The calling convention is such that all calls pass an additional `this` argument, which means that non-method

functions are needlessly expensive, even though such functions are expected to become more common as we encounter more parallel-friendly code. Also, when there is no explicit `this` argument, currently the global object is implicitly passed, so without a global object (see above) we are left with an incomplete feature.

```
function F() { this.x = 0; }
var o:Object = { f: F }; // this in F binds to o
o.f();
print(o.x); // 0
o.x = 1;
{ f: F}.f(); // this in F binds to new object
print(o.x); // 1
```

- Furthermore, all functions can be used as constructors for new objects, but without prototypes (see below), the utility of this feature is greatly diminished.

```
function F() { this.x = 0; }
var o:Object = new F(); // this in F binds to o
print(o.x); // 0
```

In AS4, the keyword `this` may appear only in an instance method, and may be arbitrarily nested inside other functions. Functions that are not instance methods do not bind `this`, although they may have `this` in scope when they are nested inside instance methods.

In AS4, the keyword `new` must be followed by a class name resolved at compile-time.

Furthermore, functions as constructors do not serve as "cast" operators. The cast operator in AS4 is `as`, whose semantics is modified to throw an error upon cast failure (rather than returning `null`, which is not type-safe since `null` is not a valid value of all types, in particular value types). The `as` operator must be followed by a type, which could be a value type or a class name.

Complementing `as` is the operator `is`, which must also be followed by a type. AS4 maintains the invariant that `v is T` if and only if `v as T === v`, for any compile-time type `T` and run-time value v.

In a later AS4 version, we may provide instance methods of the `Class` class corresponding to `new`, `is`, and `as`, so that such operations can be applied to types computed at run-time.

The main benefits of dropping `this`-bindable functions is that closures and the calling convention becomes more lightweight, and thus, more efficient.

Functions that implicitly take `this` as a parameter can be simulated by methods in corresponding hidden classes.

2.5 The `undefined` Value

AS3 supports `undefined` as the only value of type `void`; it parallels `null`, which is a valid value of type `Object`. The type `*` includes `void` and `Object`; thus both `undefined` and `null` are valid values of type `*`.

The value `undefined` arises in the following cases in AS3:

- Whenever a dynamic property is not found in AS3, `undefined` is returned.

  In AS4, missing dynamic properties are denoted by `null` instead.

```
var o:* = { };
print(o.x); // undefined in AS3, null in AS4
```

- In AS3, arrays can have "holes," which are populated by `undefined`. Holes cannot exist in AS4 arrays.

- Whenever a function of return type `void` is called in AS3, `undefined` is returned as the result.

  In AS4, we throw a compile-time error if it is clear at compile time that a value of type `void` is being expected (because there is no such value anymore); on the other hand, if it is not clear at compile time that such a thing may happen, but it still happens at run time, we throw a run-time error.

  ```
  function f():void { }
  var r:* = f(); // OK in AS3, compile-time error in AS4
  print(r); // undefined

  var g:* = f;
  print(g()); // OK in AS3, run-time error in AS4

  function h():* { return; } // OK in AS3, compile-time error in AS4
  ```

  As a corollary of compile-time enforcement, return statements are linked to the type signature of the enclosing function: if the return type is the dynamic type, then having a `return` statement without an expression is a compile-time error.

  As an elaboration of run-time enforcement, if a function is called through the dynamic type, then the compiler makes it explicit in the emitted bytecode for the (slow) function call whether it expects a result or not (e.g., emitting `call` vs. `callvoid`); and as part of the implementation of the bytecode, the function signature is looked up to throw a run-time error whenever a value of type `void` is expected.

The main benefits of dropping `undefined` are reduced complexity in the VM, and a uniform interpretation of `*` as a boxed representation for value or reference types without additional conversion.

By closing all loopholes for creating uninitialized variables, there is no reason left to keep `undefined` in addition to `null`. In AS4, variables that are not explicitly initialized and are of type `Object` or one of its subtypes are initialized to `null`. All variables of non-`Object` types are initialized to a zero-like default value. The remaining possibility is a variable of type `*`. By defining its default value to be `null`, there is no room left for `undefined`. It would just act as a "second kind of `null`", which is superfluous and just complicates the language.

Note that without the `undefined` value, we do not have the unary `void` operator.

2.6 Prototypes

Prototypes provide yet another inheritance mechanism for objects (parallel to the usual inheritance mechanism provided by classes), as the following example shows:

```
class A {
  var x:int = 6;
}
class B extends A {}

B.prototype.x = 42;
B.prototype.y = 7;
A.prototype.y = 42;
A.prototype.z = 0;

var o:* = new B();
print(o.x,o.y,o.z); // 6, 7
```

Prototypes have several drawbacks when they coexist with classes.

- To support prototypes, every object in the language carries an extra reference to another object, irrespective of whether it is an instance of a dynamic class or not.
- Furthermore, the fact that some in-built methods of `Object` are dynamic properties accessed via prototype inheritance rather than class inheritance has several unintuitive implications.

```
class A {
  // shadow, not override! (trait properties shadow dynamic properties)
  public function toString() { return "some A"; }
}
class B extends A {
  // override! (trait properties override trait properties)
  override public function toString() { return "some B"; }
}
class C {
  // not shadow! (dynamic properties are public)
  function toString() { return "some C"; }
}
class D {
  // run-time error! (dynamic properties are ignored by compile-time checks)
  public var toString = null;
}
print(new A()); // some A
print(new B()); // some B
print(new C(), new C().toString()); // [object C], someC
print(new D()); // error!
```

As long as prototype chains are fixed, prototype inheritance can be simulated by class inheritance.

By dropping prototypes, objects become smaller, and in-built methods of Object are declared as trait properties in the class itself.

Note that the keyword `instanceof` is dropped.

2.7 Dynamic Classes

In AS3, every non-final static class can be extended by a dynamic class, which features dynamic properties. This approximates JavaScript, but falls short of modelling its entire generality, because dynamic properties in AS3 do not shadow static properties nor can they replace them nor can static properties be modified or removed dynamically.

In order to make the class system fit into the prototype system, AS3 models all functions as methods so that they can be assigned as values to properties. This means that every function has an implicit `this` parameter, which dynamically binds to the receiver object upon method invocation. And in case of a function invocation it dynamically binds to the global object.

In AS4 we cannot continue this scheme, because it is inefficient to have an extra parameter for every function call and because there is no such thing as a global object any more.

2.8 `ApplicationDomain`

In AS3, global definitions are organized into a hierarchy of application domains at run time, with complicated lookup and binding rules. This feature is useful in combination with dynamic loading.

Initially, both dynamic loading and application domains are absent in AS4. However, we plan to introduce these capabilities into AS4 later, after moderate redesign relative to AS3.

2.9 The `Array` Class

We eliminate the AS3 class `Array`. The class name is reused for fixed-length arrays (see section 4.4 and appendix B.1.6), which only retain a small subset of the functionality of AS3 arrays.

2.10 The `Proxy` Class

We eliminate the `Proxy` class in AS4, mainly because its only existing uses seem to be in Flex. We will re-design `Proxy` in a later version of AS4.

2.11 The `include` Construct

The `include` feature in AS3 allows fragments of code to be spliced into larger fragments of code. This feature is problematic for tooling; furthermore, the AS3 parser has several bugs around this feature. Thus, we drop it in AS4.

2.12 Automatic Semicolon Insertion

The optionality of semicolons causes complexity in the language definition and the lexer/parser. The language specification needs to be extremely pedantic about where a line break can occur or not to disambiguate what is meant (where the presence or absence of a required semicolon would immediately disambiguate). Overall, this feature in AS3 breeds implementation errors as well as, in some cases, program understanding errors.

```
var x = foo // ;
(y as T).bar();
```

Without the semicolon, the above code could be mistaken for:

```
var x = foo(y as T).bar();
```

2.13 Assignment Expressions

In AS4 assignments, e.g., `x = e`, `x += e`, `o.x = e`, `o.x += e`, are statements, not expressions. This means that they cannot appear inside conditions, argument lists, and so on.

This restriction avoids common programming errors.

```
if (x = y) { ... }
```

We are aware that this problem is greatly mitigated by the fact that there are no implicit conversions between values of type `bool` and numeric types. But we also believe that avoiding inline assignments in general fosters program readability and understanding, especially when programming in the large in teams.

Last but not least, the above restriction paves the way for named parameter passing in a future version of AS4.

```
function foo(x:int,y:String) { ... }
foo(y = "..", x = 5);
```

2.14 Rest Parameters

We eliminate rest parameters in AS4. Calling a function with rest parameters involves hidden costs that are not apparent to the programmer. In particular, any rest arguments need to be boxed and packed into an array. We believe that the programmer is better served when the full cost is explicit at the call site, i.e., the function takes an explicit optional array parameter instead.

2.15 List of Dropped Operators

The following AS3 operators are redundant in AS4, and are therefore dropped.

unsigned right shift `>>>` - cf. overloading of right shift, see section 3.5

`void` - cf. dropped feature: undefined

`instanceof` - cf. dropped feature: prototypes; operator `is` can be used to determine whether a value is an instance of a given class

`typeof` - to be replaced by a method in a reflection package

`delete` - it is no longer possible to remove object properties

`in` - the structural composition of objects can no longer be inspected without using the dedicated reflection API (see appendix B.2)

type casting operator `T(e)` - replaced by the operator `e as T`

attribute operator `@` - repurposed for metadata, cf. dropped feature: E4X

descendant operator `..` - cf. dropped feature: E4X

namespace qualifier operator `::` - cf. dropped feature: E4X

the rest parameter indicator `...` - cf. dropped feature: rest parameters

3 Modified Features Relative to ActionScript 3

Here we list features of AS3 that are kept in AS4 in some form, and explain how they have evolved.

3.1 Program Structure

An AS4 program consists of a set of files. Each file must contain an optional package directive, optional import directives, and then a sequence of class definitions and interface definitions.

A stand-alone program defines a set of classes and interfaces and one static method to call as entry point. This method must have no argments and return `void`. A library simply omits the mentioning of the entry point.

Program execution begins with the static initializer of the class that contains the static method entry point. Once this initializer has concluded, the specified static method is executed.

3.1.1 Conditional Compilation

Conditional compilation is dictated by configuration constants that are defined by the following syntax.

```
#debug = true;

#mac = true; #windows = false; #linux = false;

#platform_is_unix = mac || linux;
```

Any directive may be conditionally compiled by guarding the directive with a configuration constant that evaluates to a boolean value:

```
#test = debug && platform_is_unix;

#test
print("testing");

#test
class TestEnv { ... }

#test
function test(env) { ... }

#test {
    let env = new TestEnv();
    test(env);
}
```

We thus carry over the expressiveness of the AS3 conditional compilation model, with a more compact syntax that effectively assumes a unique configuration namespace, which is entered by # and applies throughout one directly following statement or expression.

Configuration constants can be only defined with constant value type `bool`.[2]

3.1.2 Static Initialization Code

All static initialization code needs to be marked `static`. This means that in addition to static variable definitions and static function definitions, any free-standing static initialization code in classes needs to be enclosed in blocks that are marked `static`, for example:

```
class A {
  let x:String = "...";
  function f():String { return x; }
  static {
    print(new A().f());
  }
}
```

Regarding their internal syntax, such blocks are treated just like static function blocks.

[2] Additional value types for configuration syntax may become available later on.

3.1.3 Block Scoping

Many of AS4's constructs involve code *blocks*, which delineate nested identifier scope visibility ranges with curly braces. Such blocks are subject to certain rules that govern what entity exactly is symbolically referenced and thus denoted by each and every stated identifier in any given program. Before we provide an overview of these scoping rules, let's have a look at the variety of different block constructs in AS4.

An interface has one block. A function body consists of a block. A substatement, e.g., a branch of an `if` statement, is also considered a block. Furthermore, the programmer can insert new blocks at will in any grammatical position that permits a statement. This includes nesting blocks inside blocks.

A class definition contains two textually overlayed, but conceptually distinct blocks:

the instance block which may contain non-static variable definitions and non-static function definitions,

the static block which may contain static function definitions, static variable definitions, and static statement blocks.

To assess the identifier scopes for any program, the compiler begins by building a global lexical environment that consists of classes and interfaces. Thereby the global lexical environment is always in scope. The compiler then builds lexical environments for the static block and the instance block of every class, and the block of every interface, by visiting these types in inheritance order. Finally, it builds lexical environments for function bodies and substatement blocks by visiting them in textual order, realizing the following scoping rules.

- When visiting the static block of a class, the lexical environment of the static block of the superclass is in scope (if it exists).
- When visiting the instance block of a class, the lexical environment of the instance block of the super class is in scope (if it exists), followed by the lexical environment of the static block of the current class.
- When visiting any other block, the lexical environment of the immediately nesting block is in scope.

When visiting a block, function definitions, variable definitions, and statements are visited in textual order. Thus, the compiler ''knows'' functions and variables as they are declared. Every definition introduces an entry in the lexical environment, and every lexical reference must bind to entries that appear in the lexical environment in scope. In other words, statements, variable initializers, and function bodies will not be able to forward-refer to functions and variables declared later in the scope.[3] Furthermore, if a lexical reference binds to an entry in an outer lexical environment, then a shadowing definition cannot appear in the lexical environment immediately enclosing that reference. Examples:

[3]We plan to implement forward references (respectively recursive bindings) for local functions in the next version of AS4. Until then, as a temporary limitation, mutually recursive local functions can only be expressed when nesting one function inside the other.

```
{
    var v0 = 0;
    if (b) {
        var v0 = 3;        // ok, shadow definition, variable can be redefined
        function i() {
            var y = v0;    // ok, y = 3;
        }
    }
}
{
    var v0 = 0;
    if (b) {
        function i() {
            // var y = v0; // error: outer v0 is shadowed below
        }
        var v0 = 3;

        function m(){
            var y = v0;  // ok, y = 3
        }
    }
}
{
    function g() {
        // f(); // error, f only defined below
    }

    let x = g();

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

3.1.4 Packages and Access Control

We assume that an AS4 compiler gains knowledge of package names that may appear in compilation units (obtained either by looking at package directives that appear in the compilation units in question, or via compiler switches). Based on this knowledge and the appearance of import directives in code, references to class names and interface names are fully qualified by package names as follows.

We say that `p.C` is open in some code if there is an import directive that appears textually earlier in some block that nests the code, and it is of the form `import p.C` or `import p.*`. This has the effect of making a lexical reference `C` possibly refer to any `public` class `C` in package `p` (the reference must be uniquely resolved at compile time).

```
package p {
  public class C { }
}
```

```
package q {
  import p.C;
  function f() {
    return new C(); // desugars to new p.C();
  }
}
```

We assume that the anonymous package is always open. Any code that is outside a package directive is implicitly considered to be part of the anonymous package in AS4 (see section 3.1.

```
package { // anonymous package, internally named %anon%
  public class C { }
}
package q {
  function f() {
    return new C(); // desugars to new %anon%.C();
  }
}
```

We retain the access control modifier `internal` with the same meaning as in AS3: it specifies that an identifier is visible package-wide. Furthermore, it remains optional as in AS3, i.e. not naming any modifier implies `internal`. Thus, a class `C` that is not specified to be `public` is available only to code in the package in which the class is defined (which could be the anonymous package, of course).

Inside classes, access controls have the following meanings:

- A `public` member `m` of a class can be accessed by any code.
- A `private` member `m` of a class can be accessed only by code in the class in which the member is defined.
- An instance `protected` member `m` that is defined or inherited by a class can be accessed by code in that class only on references whose type is that class or some subclass of it.

  ```
  class A {
    protected function f() { };
    function g(x:B) {
      this.f(); // OK, routes to f defined in A or its subclasses
      x.f(); // OK, routes to f defined in A or its subclasses
    }
  }
  class B extends A {
    function h(x:A) {
      this.f(); // OK
      this.g(this); // OK
      //x.f(); // not OK, may route to f defined in unrelated C (see below)
    }
  }
  class C extends A {
    override protected function f() { ... };
  }
  new B().h(new C());
  ```

- A static `protected` member `m` that is defined in a class can be accessed by code in that class and its subclasses. (Recall that static members defined in a class are not inherited by subclasses, but are still in scope.)

```
class A {
  static protected function f() { };
  static function g() {
    //C.f(); // not OK
    f(); // OK, desugars to A.f()
  }
}
class B extends A {
  static function h() {
    f(); // OK, desugars to A.f()
    //C.f(); // not OK
    C.g(); // OK
  }
}
class C extends A {
  static protected function f() { ... };
  static function g() {
    A.f(); // OK
    f(); // OK, desugars to C.f()
  }
}
```

- A member `m` of a class that is not specified to be `public`, `private`, or `protected` is effectively `internal`. Any `internal` member is available to any code in the package in which the class is defined. It is unavailable to every other package.

3.1.5 Constant Evaluation

During lexical environment building, the compiler visits initializers of `let`s in the order they appear; as these initializers are visited, if they are *compile-time constant* then their values are inlined. Compile-time constants are numeric, boolean, and string literals, references to other `let`s that are in scope (as dictated by block scoping) and have already been initialized with compile-time constants, and ''pure'' unary and binary operations over them (including arithmetic, bitwise, comparison, and logical operators, and casting and type-checking operators).

```
/* block begins */

  //let x = y; // y not in scope
  let y = 1;
  let z = y + 1 as double;

  print(z);; // prints 2.0

/* block ends */
```

3.2 Type Hierarchy

The type hierarchy in AS4 is almost the same as in AS3, but there is a significant change at the top: value types are now directly under `*` rather than `Object`. Although `Object` is the root of all reference types, it does not include value types.

We distinguish between the notions of subtyping and conversion between types. Subtyping never involves changing the underlying representation, whereas conversion may do so.

Value types have no interesting subtyping relation among themselves, although some of them implicitly convert to others. Subtyping between reference types follows their inheritance relations in the type hierarchy. Finally, both value types and reference types implicitly convert to and from `*`.

3.2.1 The * Type and its Uses

When an entity (variable, field, etc.) is typed as `*`, then we mostly follow the traditional dynamic typing principle (as seen in various scripting languages, including JavaScript and former versions of ActionScript®) of making the program continue to run if at all possible no matter what the results. We call this a *dynamically typed code context* as opposed to a *statically typed code context*.

At the top of the hierarchy, `*` is above both value types (e.g. int, double) and reference types (`Object` and its subclasses).

The following methods are defined by class `*`:

```
/**
 * @return whether the receiver and the argument are identical.
 * Instances of Object are compared by reference.
 * They are identical if they resulted from the same instantiation.
 * Value type instances are never identical to Object instances.
 * Value type instances are identical to each other
 * if they have the same type and the same value.
 */
public native final function identical(other :*) :bool;

/**
 * @return an identity hash value for the receiver
 * Every receiver indistinguishable from another by "identical()"
 * has the same identity hash value.
 * Non-identical receivers may have different identity hash values.
 * In fact, these will differ with high probability.
 */
public native final function identityHash() :ulong;

public native function toString() :String;
```

These methods are inherited by all classes, for both reference types and value types, as `*` is at the top of the type hierarchy.

In addition to overriding the above, class `Object` defines these methods[4]:

```
/**
 * @return whether the receiver and the argument are equal
 *
 * In subclasses, equal() and equalityHash()
 * should either both be overriden or neither of them.
 */
public function equal(other :*) :bool {
    return this.identical(other);
```

[4] This is a temporary arrangement, which may change later, with the introduction of parametric interfaces.

```
}

/**
 * @return An equality hash value based on the receivers value contents
 */
public function equalityHash() :ulong {
    return this.identityHash();
}
```

The method `equal` is invoked by the `==` operator when comparing two instances of `Object` (in particular strings).

3.2.2 Value Types

AS4 provides the following builtin value types.

`bool` - the boolean type, a renaming of the AS3 `Boolean` type. The size of heap storage of `bool` values is not specified.

`byte` - new in AS4, 8-bit, unsigned machine integer type without arithmetic operations.

`int` - the name is retained from AS3, but with different semantics - it is a 32-bit, two's complement machine integer type with wraparound on overflow.

`uint` - the name is also retained but the type is given different semantics - it represents an unsigned, 32-bit machine integer type with wraparound on overflow.

`long` - new in AS4, 64-bit, signed machine integer types with wraparound on overflow.

`ulong` - new in AS4, 64-bit, unsigned machine integer types with wraparound on overflow.

`double` - IEEE 754 double precision floating point number, replacement for the `Number` type, which is absent in AS4.0.

`float` - IEEE 754 single precision floating point number.

All value type names are in short form and lowercase. No other value types are provided and AS4.0 does not allow user-defined value types (yet). Implicit conversions between numeric types are allowed, if such conversions do not result in any loss of precision. As a result, implicit conversions between signed and unsigned integer types are not allowed, unless it can be statically determined that no loss of precision will occur.

Unlike in AS3, but similarly to Java, there are no implicit conversions from and to `bool` values. This may require more typing, but it increases readability, making the intention to arrive at a binary distinction explicit. In particular, this means that the following statements do not work when `x` is of type `T` where `T` is not `bool`.

```
if (x) { ... } // error
var y = x || foo(); // error
```

This is the complete list of implicit coercions between value types in AS4:

From	To	Static Precondition
byte	int, uint, long, ulong, double, float	
int, uint	long, double	
int	uint, ulong	≥ 0
long	ulong	≥ 0
uint	int	$< 2^{31}$
ulong	long	$< 2^{63}$
long	double	$\geq -2^{53}, < 2^{53}$
ulong	double	$< 2^{53}$
int	float	$\geq -2^{24}, < 2^{24}$
uint	float	$< 2^{24}$
float	double	
double	float	literals only, see section 3.4 for restrictions

In addition, every type implicitly converts to `*`, and `*` implicitly converts to every other type. Furthermore, a type that is a subtype of another type implicitly converts to that type. Note that we don't consider `void` to be a type any more: it is merely a syntactic keyword.

3.2.3 Equality and Identity

AS4 retains the `==` and `===` operators but slightly modifies their semantics. The meaning of the identity operator `===` is fixed, defined by the language and not modifiable by user code. For reference types the `===` operator tests if two references point to the identical object. This is analogous to AS3, except that `String` is a reference type in AS4, and two references to strings with the same contents will not necessarily test as identical.

For value types the identity operator will perform value comparison. Informally, two values are identical if one can be replaced by the other without any observable effects. As a result, two numeric values of different types are not considered identical. Moreover, it is in general likely that an attempt to compare values of different types is unintentional and the possible source of a bug. Therefore, a compile error is generated if the two operands of `===` do not have a common super type besides `*`, which is the case for any pair of different value types as well as when attempting to compare a value type to a reference type. For instance, all of these expressions result in compile errors:

```
5 === new Object() // error
3 === 3.0          // error

let n :uint = 2;
-3 === n           // error
```

The `==` operator does allow its operands to be of different value types as long as generally permitted implicit coercions as listed in section 3.2.2 reach a common type. For example:

```
(1 << 31) as int == (1 < 31) as long;             // true
(1 as long) << 63 == (1 as long) << 63 as double; // true
(1 << 31) as int == (1 << 31) as uint;            // false
```

For comparisons where both arguments have reference types, `==` is syntactic sugar for the `equal()` method.

When no implicit coercion according to the rules laid out in section 3.2.2 is available, numeric comparison (`==,!=,<,<=,>,>=`) of signed and unsigned integer types results in a compilation error. This may be considered draconian, however it does prevent surprising cases occurring in C, such as the following evaluating to `1` (C's `true`).

```
let n: int = 1 << 31;
n + 1 == n as uint + 1
```

The identity comparison operator `===` generally translates directly to method `identical()` in class `*` (see section 3.2.1), with one exception that is necessary according to the above rules: if a value type is compared to a reference type, a *compile time* error occurs. To ensure *equivalent* program behavior under dynamic typing, when one or both of the operands are typed as `*`, and the actual runtime types are incomparable in the above sense, a *run time* error is thrown. Value types and reference types can only be intermixed when the programmer explicitly demands it. Here, this can be accomplished by choosing to call `identical()` directly instead of using the `operator`. Examples for all these situations:

```
let car :Car = new Car();
let person :Person = new Person();
car === person // translates to car.identical(person) => false

let m :int = 4;
let n :int = 5;
m === n // false
n === car // compile error

let x :* = n;
let y :* = car;
x === y // runtime error

x.identical(y) // false
y.identical(x) // false
x.identical(5) // true
```

The equality comparison operator `==` follows analogous rules.

```
let car :Car = new Car();
let person :Person = new Person();
car == person // translates to car.equal(person)

let m :int = 4;
let n :int = 5;
m == n // false
n == car // compile error

let x :* = n;
let y :* = car;
x == y // runtime error

x.equal(y) // false
y.equal(x) // false
x.equal(5) // true
```

3.3 Type Literals

In certain syntactic contexts in AS3, the name of a class denotes a runtime value that represents it for reflective purposes. AS4 also support this, but with less syntactic ambiguity, by requiring a leading operator, a colon, that bridges the gap between the type domain and the value domain. Examples:

```
import type.*;
...
let myClass :Class = :MyClass;
myClass.createInstance();
let myParameterTypes = []Type{:int, :bool, :MyClass};
```

See the reflection API in appendix B.2 for further information about package `type` andd its utility classes `Class` and `Type`.

3.4 Numeric Literals

Numeric literals with a decimal point are of type `double`.

```
let d1:double = 4.7;
let d2 = 4.7; // also double
```

To obtain values of type `float`, one can use casts or type annotations.

```
let f1 :float = 4.7;
let f2 = 4.7 as float;
```

Where a `double` program literal occurs, these are all possible interpretations:

1. The recipient (variable, parameter, field) is known as a `double`. This type checks trivially.
2. The recipient is known to be of type `float`. Then an implicit conversion occurs. Thus float literals do not need a trailing "f" or any other marker as in other languages.
3. The recipient is subject to type inference. Then the presence of the program literal forces the inferred type to be `double` (unless it is contradictory anyway), never `float`.

Why is case 2 OK even though a loss of precision occurs and as a general design rule we do not permit implicit conversions where this is the case? We allow an exception from the rule here, because a loss of precision also occurs in case 1! With floating point literals of any kind, there is always a hazard whether one happens to express a number that can actually be represented by the resulting bit pattern or not. One may type a number and the actually stored number differs in some decimal places. From this point of view, case 2 does not look worse than case 1. There would be a problem though, if the programmer were to implicitly expect a certain precision as expressed in your literal. This is why case 3 infers `double` from floating point literals, unless overruled by explicit typing. Thus the programmer only encounters extra loss of precision when explicitly asking for it.

Numeric literals in decimal form but without a decimal point are of type `int`, unless the literal value does not fit in an `int`, in which case they are of type `long`.

```
let i1 :int = 3;
let i2 = 3;
let i3:long = 1000000000000000;
let i4 = 1000000000000000; // inferred type is long
```

String literals are of type `String`. The rules for single and double quotes are the same as in AS3.

```
let s1 = "Hello World!";
let s2: String = "Hello!", responded the world;
let s3: String = "bye";
```

In initialization assignments, type annotations on the variable being defined can be used to request compile-time conversion. Alternatively, the cast operator syntax can be used (the cast operator on constants is interpreted by the compiler as a part of constant propagation, see 3.1.5)

```
let i2 :uint = 5; // this is fine
let i1 = 5 as uint;  // cast syntax
let l2 :long = 5;
let 1l = 5 as long;
```

In expression contexts the cast operator syntax can be used to obtain the desired type of the constant:

```
var hibits = (0xFFFFFFFF as ulong) << 32;
```

3.5 Arithmetic, Bitwise, and Comparison Operators

Integral types are `byte`, `int`, `uint`, `long`, and `ulong`. Numeric types are integral types, `double`, and `float`. Value types are numeric types and `bool`. Reference types are `Object` and its subclasses, including function types: thus, reference types are non-value types other than `*`.

The *least upper bound* (LUB) of two numeric types T_1 and T_2 is defined as a numeric type T_3 such that T_1 and T_2 implicitly convert to T_3, and for any other numeric type T_3' such that T_1 and T_2 also implicitly convert to T_3', we have that T_3 implicitly converts to T_3'.

The tables in the following subsections outline all possible typings for various binary and unary operators.

3.5.1 Binary `+`

Operands	*Result*	*Compile-Time Precondition*
T_1, T_2	T_3	T_1 and T_2 numeric but not both `byte`, LUB of T_1 and T_2 is T_3 (cast both operands to T_3)
T_1, T_2	String	T_1 or T_2 is `String` (cast both operands to String)
T_1, T_2	`*`	T_1 or T_2 is `*` and the other is `*` or numeric (unbox `*` operand(s), box result)

In particular, we have the following typings:

```
//byte + byte // ERROR
byte + int = int
int + int = int
uint + uint = uint
int + uint = long
int + double = double
//long + ulong // ERROR
//double + long // ERROR
int + String = String
String + * = String
int + * = *
```

3.5.2 Binary `-`, `*`, `/`, `%`

Operands	*Result*	*Compile-Time Precondition*
T_1, T_2	T_3	T_1 and T_2 numeric but not both `byte`, LUB of T_1 and T_2 is T_3 (cast to T_3)
T_1, T_2	`*`	T_1 or T_2 is `*` and the other is `*` or numeric (unbox `*` operands, box result)

3.5.3 Binary &, |, ^

Operands	*Result*	*Compile-Time Precondition*
T_1, T_2	T_3	T_1 and T_2 integral but not both `byte`, LUB of T_1 and T_2 is T_3 (cast both operands to T_3)
T_1, T_2	*	T_1 or T_2 is * and the other is * or integral (unbox * operand(s), box result)

3.5.4 Unary -

Operand	*Result*	*Compile-Time Precondition*
T	T	T numeric but not `byte`
*	*	(unbox * operands, box result)

3.5.5 Unary ~

Operand	*Result*	*Compile-Time Precondition*
T	T	T integral but not `byte`
*	*	(unbox * operands, box result)

3.5.6 Binary <<, >>

Left Operand	*Right Operand*	*Result*	*Compile-Time Precondition*
T	S	T	T integral but not `byte`, S is implicitly convertible to `int`
*	S	*	S is implicitly convertible to `int`

3.5.7 Binary ==, !=

Operands	*Result*	*Compile-Time Precondition*
`bool`, `bool`	`bool`	
`byte`, `byte`	`bool`	
T_1, T_2	`bool`	T_1 and T_2 numeric but not both `byte`, LUB of T_1 and T_2 is T_3 (cast both operands to T_3)
T_1, T_2	`bool`	T_1 and T_2 reference
T_1, T_2	`bool`	T_1 or T_2 is * (unbox * operand(s))

3.5.8 Binary <, >, <=, >=

Operands	*Result*	*Compile-Time Precondition*
T_1, T_2	bool	T_1 and T_2 numeric but not both `byte`, LUB of T_1 and T_2 is T_3 (cast both operands to T_3)
T_1, T_2	bool	T_1 or T_2 is * and the other is numeric or * (unbox * operand(s))

3.6 Object Literals

In addition to the AS3 constructor call syntax for object creation, viz. `new C` and `new C(arg1,...,argn)`, we additionally introduce the syntax `new C{x1=e1,...,xn=en}` and `new C(arg1,...,argn){x1=e1,...,xn=en}`, which effectively constitutes object literals.

As in AS3, an object literal with a missing argument list (in parentheses) is replaced by a call to the 0-ary constructor of the specified class. In addition, the assignment list (in braces) is desugared to a sequence of assignments to the corresponding fields of the newly created object. Syntactically, the class specified in an object literal must not be an array type or a function type.

3.7 Metadata

AS4 supports metadata, but changes the syntax from AS3. Syntactically, metadata definitions have the exact same shape of object literals, except that `@` is used to introduce them. For instance:

```
@Builtin
@Native(12.0){cls = "DateClass", instance = "DateObject",  gc = "exact", construct = "override"}
public class Date {
     ...
}
```

Semantically, only *conforming* object literals are allowed as metadata, with the leading `new` replaced by `@`. For now, an object literal is conforming if the expressions in it are compile-time values of primitive types, strings, or type literals (see section 3.3) or `null`. The name following the `@` symbol must be a valid class name and the class must be a subclass of `Object`. The classes above are declared as follows.

```
public class Builtin {}

public class Native {
  var cls :String;
  var instance :String;
  var gc :String;
  var construct :String;
  let version :double;
  function Native(version :double) { this.version = version; }
}
```

3.8 Strings

AS4 strings represent immutable sequences of Unicode code points. The length of the sequence is represented as an `int`. A Unicode ‘‘code point’’ is represented as a `uint` value in the $[0, 17 * 2^{16}]$ range.

```
public class String extends Object {
    public native final function length() :int; // the number of code points
    public native final function codePointAt(index :int):uint;
    public native final function indexOf(string:String):int;
    override public native final function equal(other :*):bool;
    ...
}
```

The language does not specify which Unicode encoding is used internally by its implementation. Indexed access to code points is provided via the `codePointAt()` method. However, due to the internal use of variable-length encodings (e.g. UTF-8), it is not guaranteed nor is it likely to have $O(1)$ complexity. For fast iteration, it is therefore not advisable to use a loop with a counter index.

```
let s :String = ...; // large String

for (int i = 0; i < s.length; i++) {
    let codePoint = codePointAt(i);   // SLOW!
    ...codePoint...
}
```

There is no way to iterate quickly over the characters of a string, but we intend to add one later. Until then, a string can be rapidly converted to an array of code points:

```
let codePoints :[]uint = s.toCodePointArray(); // quite quick

for (int i = 0; i < codePoints.length; i++) {
    let codePoint = codePoints[i]; // FAST
    ...codePoint...
}
```

Unlike in AS3, identity comparison of `String` objects does not result in content comparison. For instance, the language does not guarantee that `'Hello' + (function():String return "World";)()` is identical to `'HelloWorld'`.

The absolute maximum length of strings is $2^{32} - 1$ as the return type of `String.length()` is `int`. However, the actual maximum size that can be created in a program is limited to whatever the current runtime allows.

3.9 `Vector` becomes `ArrayList`

AS3 has a variable length array type called `Vector`, which is parametrized by its element type. We provide a very similar construct that is now called `ArrayList`. However, the type syntax no longer contains a ".". For example:

```
var v :Vector.<int> = new Vector.<int>(4,false); // old syntax
var l :ArrayList<int> = new ArrayList<int>(4); // new syntax
```

The option to dynamically fix the length of a vector is preserved, but there is no way to change the length again once it is fixed. This method in class `ArrayList` replaces the field `Vector.fixed`.

```
/**
 * Prohibit length mutation of this array list going forward.
 * @return an array that contains the same elements as the receiver
 * Repeated calls may return the identical array.
 * The implementation may or may not have copy overhead.
 * Aliasing may occur when accessing the resulting array
 * and the original array list as well.
 *
 * This is a temporary measure for our first release.
 * We will follow up with a parametrically typed version later.
 * This later version will also always provide no-copy aliasing.
 */
public native function fixLength() :*;
```

The constructor no longer has the length fixing boolean argument:

```
var v :Vector.<int> = new Vector.<int>(4,true); // old syntax

var l :ArrayList<int> = new ArrayList<int>(4); // new syntax
```

```
l.fixLength();
```

There is no object literal syntax for array lists. However, a static method is provided for bulk initialization with an array.

```
/**
 *  @return an array list that is populated with the array elements
 *
 * This is a temporary measure for our first limited language release.
 * We will follow up with a parametrically typed alternative constructor later.
 */
public static native function create(array :*) :*;
```

We would like to provide generic programming against both array lists and arrays with a common interface. Unfortunately, the first version of AS4 will not let us express this. However, we plan to introduce this capability in a later release with generalized parametric types.

3.10 The `override` Method Attribute

Use of the `override` attribute for methods that do override other methods is optional in AS4. We believe that IDEs can indicate the overriding relationship and give related warnings as needed. However, we still want to give programmers who do not use an IDE the option to declare their intent visibly.

3.11 For-in enumeration

The syntax and semantics of `for-in` enumeration have been modified to enable more aggressive optimizations and facilitate correctness.

```
let v :ArrayList<String> = ...; // length 5
for (i :int in v) {
  print(v[i]); // executes 5 times
  if (...) v.push("...");
}
```

In the above code, `i` ranges over `0..4`, even though the length of the array list may change during execution of the loop.

For-in enumeration over property maps and dictionaries similarly work on a fixed key set: the underlying data structure may grow (or shrink), without affecting the values bound to keys or the number of times the loop is executed.

For-in enumeration over arrays obviously works on a fixed key set as well, because arrays cannot be grown (or shrunk) in AS4.

Additionally, we drop `for-each-in` enumeration. This feature will be replaced by ‘‘comprehensions’’ in a later version of AS4. Meanwhile, common AS3 usecases can be ported by a combination of `for-in` enumeration and indexing operations.

4 New Features

The following first batch of new features for AS4 provides essentials for:

- laying a sound foundation for functional programming in an object-oriented setting,
- enabling high performance computing.

4.1 The `let` Declaration

In preparation of greater use of concurrency and larger scale software engineering in ActionScript, and to help program correctness in general, we strive to make the declaration of constant values easy and to make immutability the default wherever an explicit indication regarding mutability is not given.

In AS3, variables are customarily declared with the keyword `var` and constants with `const`. However, constants are not really guaranteed to be constant, i.e. it is often possible to assign a value to them several times and they sometimes can be read before written to.

In AS4, we replace the `const` keyword with `let` and we reserve `const` for interesting future uses such as making whole data structures immutable. We hope that the comparative brevity of `let` will encourage much more frequent usage than `var` and we back up the usefulness of `let` with these strong guarantees for every variable declared with it:

- Within each active scope and extent, it has exactly one immutable value that can ever be observed.
- It cannot be read before it is initialized.
- Every control flow path leads to an initialization or to a state in which it is guaranteed that its value will never be read. The latter can for example occur when an exception is thrown between its declaration and its initialization.
- If there are multiple initializations, then the values they assign can differ, but it must be provable at compile time that exactly one will execute in every possible control flow that remains in scope.
- The compiler ensures all of the above by throwing errors wherever any of it does not hold.

In short: such constants are truly constant. No value mutation can ever be programmatically observed. This holds for local variables and for class instance fields. Even though function parameters are not preceded by either `let` or `var`, they are always treated as if they were declared with `let`, i.e. they are truly constant as well.

Both our commitment to promoting immutability and the choice of the keyword `let` instead of `const` are deeply rooted in the design philosophy that AS4 is a multi-style language, which is both object-oriented and functional. We are aware that `const` would be a more familiar keyword for most of our existing users and that its name immediately indicates immutability. However, the relevance of these arguments fades quickly with the number of lines of AS4 code written or read. Looking further ahead, they are outweighed by the brevity of `let`, which matches `var` and provides even indentation, and by its stronger association with functional programming. This is consistent with our intention to make AS4 surpass today's mainstream object-oriented languages (JavaScript, Java, C#, Objective C, ActionScript 3) in this important regard: functional programming is no longer a mere appendix to object-oriented programming, it is going to be the primary style used for algorithm encoding, because it is usually superior in performance and correctness. This holds especially in concurrent settings.

Nevertheless, for well-known reasons, object-oriented code is still expected to dominate much of programming, as it facilitates programming in the large. So in the spirit of greater harmony with functional programming, AS4 provides improved support for both static and instance field immutability. The above guarantees apply

to these as well. Whereas this is straight forward for static fields, a novel approach is needed for instance fields to prevent situations like the following:

```
class A {
    function f() {
    }

    function A() {
        // compile error in AS4:
        this.f(); // reads x before initialization!
    }
}
class B extends A {
    let x:int;

    override function f() {
        print(this.x);
    }

    function B(x) {
        super(); // calls f()
        this.x = x;
    }
}
```

Here, the presumable "constant" `x` is read before the program assigns a value to it. Other program languages typically offer solutions along these lines:

- The variable `x` is already pre-initialized with a default value (zero) before any user code runs. This means that two distinct values are observable. Hardly a constant! Although static or dynamic optimization can often determine that most code only sees the variable after its final assignment, considerable user code complications remain.
- The above program is rejected by the compiler as it recognizes that `x` is not a true constant. (This is the case in AS4, but see below for why this is problematic and how we solve the problem).
- An enhanced (flow-sensitive/context-sensitive) type system lets `x` be addressable only downstream from its initialization statement. In such a system, the above program would still not type check.

In the latter two cases, the user would be forced to write a factory class. Much boilerplate! The next section presents our solution to this important problem.

4.2 2-Phase Constructors and Truly Constant Fields

In a constructor, every variable field declared by `var` is initialized with a zero-like default value before any user code in the constructor is executed. This holds even when an explicit initializer is given, since it could indirectly read the same variable that is about to be initialized. Constant fields declared by `let` can also have an initializer assignment, but the compiler ensures that they cannot be read during the execution of this statement.

Constant initialization can be deferred to the constructor body. Still, each such deferred constant must be initialized exactly once in every possible code path of the constructor. Example:

```
class A {
    var v :int;
    let dateStamp :Date = Date.current();
    let x :X;

    function A(x :X) {
        this.x = x;
    }
}
```

The variable field v is initialized to zero. Next, the constant field *dateStamp* is initialized to the current data. Finally, the constant field x must be initialized in the constructor or the compiler will post an error.

In order to support the common programming practice of publishing of `this` inside the constructor, constructors are split into two phases. In the first, default phase, `this` can only be used to write properties of the object being constructed. Any read operations involving `this` are statically rejected, in particular, `this` cannot be passed as an argument, used to call methods (other than the constructor of the superclass), or aliased with other variables. In the second, optional phase, the use of `this` is unconstrained. The second phase is expressed in the `defer` block. Example:

```
class B extends A {
    let y :Y;

    function B(x :X, y :Y, k: Key, d :Dictionary) {
        super(x);
        this.y = computeSomething(x, y);
        defer {
            d.add(key, this);
        }
    }
}
```

```
class C extends B {
    let z :Z;

    function C(x :X, y :Y, k: Key, d :Dictionary, z :Z) {
        super(x, y, k, d);
        this.z = z;
        print(d);
        defer {
            print(this);
        }
    }
}
```

It is of course possible to extend a class with a constructor that has a `defer` block with a class whose constructor does not have one:

```
class D extends C {
    function D(x :X, y :Y, k: Key, d :Dictionary, z :Z) {
        super(x, y, k, d, z);
    }
}
```

The `defer` keyword creates a closure that captures the current environment and schedules this closure to

run after all initialization has completed. The `defer` blocks in an inheritance chain are run in inheritance order.

This new language features accomplishes:

- No part of `this` can ever escape from the initialization part of the constructor.
- It is thus impossible to observe the value of an uninitialized field.
- In the `defer` block one can immediately use the constructed object without first leaving the constructor context.
- For many use cases it is therefore not necessary to write a factory method or class.

4.2.1 Constructor Implementation

It is instructive to inspect in more detail how constructors with `defer` blocks translate to conventional language features. The general form of the translation scheme, shown below, relies on closures. However, it is important to note that such closures can always be inlined away, and an AOT compiler can inline them away at compile time (as illustrated later).

For illustration purposes we pretend here that AS4 already has inline anonymous closures. Please be aware that this feature will be introduced in a later version and its syntax may change by then. In reality the compiler creates code for this behavior directly, without the intermediate step of source code as shown here.

```
static function A.<init>(this :A, x :X) :()=>void {
    // A's init code:
    this.v = 0;
    this.dateStamp :Date = Date.current();
    this.x = x;
    return function():void {
        // A's deferred code
    };
}

static function A.<prep>(this :A, x :X) {
    k = A.<init>(this, x);
    k();
}

static function B.<init>(this :B, x :X, y :Y, k: Key, d :Dictionary) :()=>void {
    k = A.<init>(this, x); // call super
    this.y = computeSomething(x, y); // B's init code
    return function():void {
        k();
        d.add(key, this); // B's deferred code
    };
}

static function B.<prep>(this :B, x :X, y :Y) :()=>void {
    k = B.<init>(this, x, y);
    k();
}
```

```
static function C.<init>(this :C, x :X, y :Y, k: Key, d :Dictionary, z :Z)
                     :()=>void {
    k = B.<init>(this, x, y, k, d); // call super

    // C's init code:
    this.z = z;
    print(d);

    return function():void {
        k();
        print(this); // C's deferred code
    };
}

static function C.<prep>(this :C, x :X, y :Y, k: Key, d :Dictionary, z :Z) {
    k = C.<init>(this, x, y, k, d, z);
    k();
}
```

And an instantiation statements for each of these classes translate to:

```
a = Heap.allocate(VM.instanceSize(A));
A.<prep>(a, x);

b = Heap.allocate(VM.instanceSize(B));
B.<prep>(b, x, y, k, d);

c = Heap.allocate(VM.instanceSize(C));
C.<prep>(c, x, y, k, d, z);
```

The above implements that in case of `c`, these phases execute in this listed order:

1. A's init code
2. B's init code
3. C's init code
4. A's deferred code
5. B's deferred code
6. C's deferred code

Both the AOT compiler and the JIT can optimize the above code with directed, aggressive inlining so that compared to constructors without `defer` blocks no extra call overhead actually occurs at runtime. To arrive at such deep inlining, the optimizer needs to be capable of relatively simple escape analysis and of reverting closure abstraction.

In particular, the AOT compiler does not need to emit intermediate closures at all. For example, the constructor for class `C` above can be inlined to the following:

```
static function C(this :C, x :X, y :Y, k: Key, d :Dictionary, z :Z) :void {
  this.x = x;
  this.y = computeSomething(x, y);
  this.z = z;
  print(d);
  d.add(key, this);
  print(this);
}
```

4.2.2 Exceptions in 2-Phase Constructors

What if an exception occurs in a constructor? If it is thrown in the init block and it is not caught, then it escapes the entire constructor, leaving the program with no reference to the object being constructed whatsoever. However, all other side-effects on the rest of the program that already occurred by means of the constructor's code remain in effect.

If the exception is caught, then its handling code is part of the init block. The compiler must ensure that also in this case all the above rules for `let`-declared fields apply. Each of them must be assigned exactly once throughout the entire actual code path and cannot be read. Also, `this` must not escape from exception handlers or `finally` blocks.

If an exception occurs in a `defer` block and it is not caught in the same block, then it is propagated all the way out of the constructor. This means that no `defer` block can catch another one's escaping exception. In such an event, some user-desired invariants may get broken, but at least system safety remains intact, even if the new object has been leaked, since it is already fully initialized.

4.3 Static Verification of Constructors and Constants

This section summarizes static verification around `super` and `let`, to support the above new features.

4.3.1 `super`

If a `super` call does not appear in constructor, a default `super` call (with no arguments) is inserted at the beginning of the constructor.

Otherwise a `super` call must appear in the beginning of the constructor, and there must not be any other `super` call in the constructor.

4.3.2 `let`

The compiler needs to prove that a `let` is not read before it is written, and that it is written exactly once.

At runtime, it should be sufficient to disallow writes to a `let` through dynamic code, and have no restriction on its reads.

Local `lets` The compiler ensures that there is one and only one write to a `let` as follows:

1. The *FunctionBody* of a function defined in the same scope as the `let`, or any other scope enclosed by that scope, must not write to the `let`.

2. In the control-flow graph of the scope, there must not be any path from the declaration of the `let` to the end of the scope on which there is zero or multiple *Assignment*s to the `let`.

The compiler ensures that there is no read to a `let` on any path in the control-flow graph of the scope between the declaration of a `let` and the one and only one write to the `let`, as follows:

1. No function defined in the same scope as the `let` that reads the `let` or recursively calls such a function, is read (or called) on the path.
2. There is no read of the `let` on the path.

Static `lets` The compiler ensures that there is one and only one write to a `let` as follows:

1. The *FunctionBody* of a function defined in the same scope as the `let`, or any other scope enclosed by that scope, must not write to the `let`. Furthermore, the `let` must not written through a static member reference outside the scope. (Note that writes may still be possible in dynamically typed code: e.g., `{dyn:* = A; A.x = ...}`.)
2. In the control-flow graph of the scope, there must not be any path from the declaration of the `let` to the end of the scope on which there is zero or multiple *Assignment*s to the `let`.

The compiler ensures that there is no read to a `let` on any path in the control-flow graph of the scope between the declaration of a `let` and the one and only one write to the `let`, as follows:

1. There is no read, write, or call of anything defined outside the class on the path, including possibly via dynamically typed code. (Everything defined outside the class is assumed to read the `let`. We can narrow this assumption in some cases.)
2. No function defined in the same scope as the `let` that reads the `let` or recursively calls such a function, is read (or called) on the path.
3. There is no read of the `let` on the path.

Instance `lets` The compiler ensures that there is one and only one write to a `let` as follows:

1. The *FunctionBody* of a non-constructor function defined in the same scope as the `let`, or any other scope enclosed by that scope, must not write to the `let`. Furthermore, a *DeferStatement* in a constructor must not write to the `let`. Finally, the `let` must not written through an instance member reference outside the scope. (Note that writes may still be possible in dynamically typed code: e.g., `{dyn:* = this; dyn.x = ...}`.)
2. In the control-flow graph of the constructor, there must not be any path between the super statement and the defer statement on which there is zero or multiple *Assignment*s to the `let`.

The compiler ensures that there is no read to a `let` on any path in the control-flow graph of the constructor between the super statement and the one and only one write to the `let`, as follows:

1. There is no read of `this` (other than an access of an instance field) and there is no read (or call) of any instance function on the path. (Every instance function is assumed to read the `let`. We can narrow this assumption in some cases.)
2. No function defined in the constructor that reads the `let` or recursively calls such a function, is read (or called) on the path.
3. There is no read of the `let` on the path.

4.4 Fixed Length Arrays

In AS3, the class `Array` stands for a versatile yet poorly performing general purpose collection type with indexing. Often, AS3 performance can receive a boost by replacing code using `Array` with code using `Vector` instead. And we intend to hold on to `Vector` (renaming it to `ArrayList` notwithstanding). However, in many cases, there is no need for even the few remaining features of `Vector` either. Often, the length of an array/vector is known to be fixed. Then further code optimizations can ensue and we can obtain even better performance. This will hold in particular in the presence of concurrency.

Another reason to introduce fixed length arrays is to increase program correctness, by encoding a relevant invariant in the type system.

Arrays are not type-compatible with array lists (vectors). Neither type is a subtype of the other. However, we provide conversion routines between the two types as described in section 3.9.

4.4.1 Declaration and Initialization

This example declares and initializes an array of `int` values of length 4.

```
let a :[]int = new [4]int;
```

The length must not be omitted. Every array must have a definitive length.

All array elements are automatically initialized to the default value of the element type. In this example, this is zero. For object types, it is `null`. However, an explicit initializer can be given:

```
let a = new []int{computeHeight(), 2, computePrice(weight, count), 4};

let streetNames = new []String{"Anza", "Balboa", "Chavez"};
```

The element type must be explicit as shown, and the element count must be omitted.

In a future release, the type declaration `[]int` will be interpreted as syntactic sugar for `Array<int>`. Either notation will then be allowed interchangeably. However, realizing this immediately would mean precedent for generalized generics and we would be forced to introduce related features prematurely.

4.4.2 Indexing and Length

Array elements can be accessed with familiar indexing syntax:

```
value = a[index];
a[index] = value;
```

These are the permitted index types: `int` and `uint`. Each of them causes the access in question to be translated internally in a slightly different way. This is disambiguated by the compiler which can regard this as operator overloading.

The meaning of an array access remains the same across all index types as long as it succeeds. The reported or inserted value will be the same and the location addressed in the array will be the same irrespectively. However, bounds checking mechanics may differ: in case of an unsigned index, checking against indices less than zero is naturally omitted.

In principle, arrays can have upto $2^{32} - 1$ elements. However, runtime memory management may reduce this at will by rejecting allocations that exceed present capacity or address ranges. Arrays are thought of as having this method returning instance length:

```
public native final function get length() :int;
```

We may later introduce an additional array type that has instances of greater lengths in the 64-bit address range.

4.4.3 Arrays of Arrays

Arrays of arrays can be declared by appending additional `[]` bracket pairs. In the following example only the first dimension gets created and initialized. No elements that represent the additional dimensions are created at this point. They have to be assigned individually in subsequent steps.

```
let a2 :[][]String = new [4][]String;
a2[0] = new [5]String;
a2[0][3] = "Hello";

let t :String = a2[1][1]; // null pointer exception
let s :String = a2[1]; // type error
let b :[]String = a2[1]; // OK
```

This change in the construction of the array creates all the substructures as well:

```
// 4 arrays of arrays with 5 strings each:
let a2 :[][]String = new [4][5]String;

a2[0][3] = "Hello";
let t :String = a2[1][1]; // succeeds
```

Additional dimensions can be added. This is only limited by resource capacity and the developers appetite for complexity.

```
let a3 :[][][]String = new [4][5][8]String; // OK
let a3 :[][][]String = new [][5][8]String; // compile error
let a3 :[][][]String = new [4][][8]String; // compile error
let a3 :[][][]String = new [5][4][]String; // OK
```

Note that the brackets are a type prefix. They denote a type of arrays of what follows them. This is like in the Go language.

Initializer syntax for arrays of arrays is as follows:

```
let a1 = new []int{1, 2, 3, 4, 5, 6};
let m2 = new [][]int{{1, 2, 3}, {4, 5, 6}};

let a3 = new [][][]int{{{{1, f()}, {1, 2}, {1, 2}},
                        {{{1, g()}, {1, 2}, {1, 2}}};
```

A tree data structure such as an array of arrays differs from a multi-dimensional array, which is one flat, coherent aggregate without any graph structure. We will introduce this feature in a later release and we have shaped the above so that it is forward compatible with our plans in this regard. The prospect of adding multi-dimensional arrays to the language is the reason we did not simply define comma syntax as sugar for accessing arrays of arrays.

To preserve the direct applicability of JSON syntax in AS4, we have array literals such as these:

```
let numbers = [1, 2, 3, 4, 5];
let names = ["David", "John", "Peter"];
```

They desugar to:

```
let numbers :[]* = new []*{1, 2, 3, 4, 5};
let names :[]* = new []*{"David", "John", "Peter"};
```

4.4.4 Explanation of Syntax

The syntax for array types has `[]` as prefix (following a modern trend: see the language Go) rather than postfix (which is the traditional choice).

Consider what happens with postfix syntax in, say, Java. To initialize a 4-sized array of `String`-arrays, and initialize one of those arrays to a 5-sized `String` array, one may write:

```
String[][] a = new String[4][]; a[0] = new String[5];
```

But this scheme cannot be generalized to arbitrary types instead of String. For example, if one replaced String by String[], one should have been required to write the following, which surprisingly is a syntax error!

```
var a:String[][][] = new String[][4][]; a[0] = new String[][5];
```

So Java compromises by requiring a more complicated interpretation of array type syntax. In other words, array types are not considered to be of the form `T[]`, where `T` could be any type including an array type, but instead array types are of the form `T[]...[]`, where `T` is a non-array type.

Unfortunately, the above scheme is brittle.

As we have larger and larger types in the language (function types and array types in this version, possibly generics in a future version), it would make sense to have type macros, which would then be expected to expand seamlessly (without causing syntax errors as above). And moving forward, we may actually want sizes in array types, like Go (a performance-oriented language), which opens up better code optimization opportunities. Thus, we would like to have `String [4][5]` in addition to `String[][]` to describe the type of an array. In such a scenario:

```
type T = String[5];
... = new T[4]
```

would macro-expand to:

```
... = new String[5][4];
```

but actually mean:

```
... = new String[4][5];
```

This order reversal is fairly unintuitive. To solve this problem, Go has `[]` as a prefix rather than suffix type operator. This reads well enough (think: `Array<int>`), lets one write `new [4][5] String` to mean 4-sized array of 5-sized `String`-arrays, and lets one have a simple type language where `[] T` indeed means "array of `T`" for all `T`, even array types.

There is another, deeper reason (which is related to the above reasons). Postfix array notation feels natural for programmers who have learned to parse `String[4][5]` en bloc, rather than parsing it as `(String[4])[5]`. But this idea breaks down when we add another postfix operator to the language. In particular, we may want to introduce a type of the form `T!` in a later version of AS4, that denotes the type of non-null references of type `T`. How would that look if we had postfix array notation? Suppose we start with:

```
... = new String[4][];
```

But we want to actually be more specific, and say that we want a 4-sized array of non-null `String`-arrays. We would then have to write something like:

```
... = new (String[])![4]
```

This has nothing to do with the fact that we chose `!` to be a postfix operator. If we try to make it a prefix operator, the interaction actually becomes worse.

A readability argument may be made against the new syntax, but it mostly comes from brain-print. Following the reasoning above, it is no surprise that other modern languages are beginning to reverse this convention. It might well be that in some years the prefix array notation will become the norm and people will simply get used to it and argue why it feels "readable".

We are not the first to note this problem. As mentioned above, Go already does this. Other proposals have appeared in the literature. For example, C++ type declaration syntax has been rewritten to be saner.[5] The authors use `[][] int` syntax in their cleanup of C++ declarations, and their overall goals include readability. As another example, the C type declaration syntax has been explained and some fixes suggested.[6] The author uses "array of `int`" syntax, the English equivalent of `[] int`, in his systematic explanation of C's confusing declaration syntax. (Careful readers of the second paper will note that the simplified syntax in the appendix leaves array declarations as they are; the first paper cites the second and states that the second paper (chronologically first) didn't go far enough.)

In summary, there is no getting around the fact that postfix arrays make sense only because we don't realize that we are parsing the dimensions "in reverse" by habit, but that breaks down when we have anything else that needs to interleave seamlessly with this parsing rule. The problem is not with postfix per se, but with the fact that we additionally desire the textual ordering of dimensions to be the same as the textual ordering of access operations. This problem is guaranteed to surface when we try to add more type operators to the language, as we will invariably do to expose more and more optimization opportunities to the programmer. (That is a design theme that we are going to adhere to moving forward.)

4.5 Function types

One of the larger holes in AS3's type system is that all functions have the same type, `Function`. For almost all uses besides *-typing, it can be closed completely in AS4 using the following syntax.

```
function foo(i :int, s :String) :double {...}
let f :(int, String) => double = foo;
let d :double = f(1, "Hello");
```

The parentheses must always be present, even when the parameter list is empty:

```
function bar() :void {...}
let g :() => void = bar;
g();
```

4.5.1 Subtyping of function types

Recall that if a value of type T flows to a variable of type S, then the compiler checks that T is a subtype of S, or T is implicitly convertible to S.

A function type of the form $(T_1,\ldots,T_n) => T$ is considered a subtype of a function type of the form $(S_1,\ldots,S_n) => S$ whenever S_1 is a subtype of T_1, ..., S_n is a subtype of T_n, and T is a subtype of S. This is consistent with the intuition that a function that takes less precise types and returns a more precise type can act like a function that takes more precise types and returns a less precise type.

[5] http://dl.acm.org/citation.cfm?id=240964.240981
[6] http://dl.acm.org/citation.cfm?id=947627

As usual, a function type is implicitly convertible to and from `*`. Crucially, though, function types that have `*` as parameter or result types are not implicitly convertible to and from function types of the same shape that have non-`*` types in those positions: such implicit conversion would require wrapping functions with boxing/unboxing operations, which is deemed too expensive in time and space. (We might consider this feature later.)

4.5.2 Default Parameters

The type of a function with default parameters is a subtype of the function's full signature, which has all parameters specified as non-default.

4.6 Type Inference

In AS4, the programmer need not specify types at all declaration sites: missing types can be inferred. This is important because of several reasons:

- Dynamically typed code is not obligated to be optimized at all by the compiler and runtime; indeed, dynamically typed code is expected to be very slow, so the cost of not writing a type in AS4 is much more than in AS3.
- As types become more expressive, they also become more verbose (e.g., function types and array types in AS4), and it becomes more and more tedious to write down static types to ensure good performance.
- With richer numeric types and tighter typechecking around numeric types in AS4, it is sometimes better to let the compiler figure out the best possible representation for a location rather than specify it explicitly.

Type inference infers static types where possible, and may fall back to inferring the dynamic type (emitting warnings).

Type inference is limited to the following cases:

- Type inference can always infer types of `let`s.
- Type inference can always infer types of local variables.
- Type inference can always infer return types of functions.
- The programmer must specify parameter types of functions: type inference will fall back to the dynamic type for missing parameter types.
- The programmer must specify types of any non-`let` instance variables and static variables: type inference will fall back to the dynamic type for missing types of non-`let` instance variables and static variables.

This design encourages developers to transform more instance and static `var` declarations to `let` declarations, and to leave it to the compiler to figure out best possible representations for local variables and returns.

Furthermore, this design allows type inference to be implemented by a simple, standard data flow analysis: the type of an expression is computed from the types of its parts, and the type inferred for a location is the union of the types of all expressions that may flow to that location.

Note that in separate compilation scenarios, it is recommended practice to annotate types for all non-private fields and non-private methods of classes, including types for `let`s and returns that could have been inferred.

As an illustrative example, type inference allows the omission of all types other than parameter types in the following code, without degrading performance.

```
function foo(i :int, s :String) // :double
 {
   let x = 2; // x :int
   var r = x; // r :double, since LUB(int,double) is double
   r *= 0.5;
   return r;
 }

 let f = foo; // f :(int, String) => double
 var d = f(1, "Hello"); // d :double
```

The LUB relation defined earlier for numeric types is generalized to arbitrary types, as follows.

For reference types A and B, the LUB is the least common ancestor in the inheritance hierarchy.

For function types, the LUB is a function type that takes the intersection of the parameter types and the LUB of the result type.

The intersection of two types T_1 and T_2 is defined as T if T is one of T_1 and T_2, and is a subtype of the other. In particular, if T_1 and T_2 are numeric types then their intersection is defined only when $T_1 = T_2$. If T_1 and T_2 are reference types then their intersection is defined only when $T_1 = T_2$ or one is an ancestor of the other in the inheritance hierarchy.

5 Upcoming Features

So far, we have just described the intended feature set of the first version of AS4. In this section we present additional features that we want to introduce as quickly as possible thereafter.

Note that we cannot guarantee that these or any other features will appear in any particular form or order. Everything we have written from here on is to be considered as somewhat speculative.

5.1 The `Symbol` Class for String Interning

Instances of `String` objects are not interned, neither explicitly nor implicitly. If interning behavior is expected, the `Symbol` class must be used.

```
public restricted class Symbol extends String {
    /**
     * "Intern" a given string.
     *
     * 1. If the identical string is already in the global intenring table,
     *    return it.
     * 2. Otherwise, if a string of equal content is already in the table,
     *    return it.
     * 3. Otherwise, enter a copy of the string typed as \code{Symbol}
     *    into the table, then return this resident unique representative
     *    of all strings of equal content.
     *
     * The cost of the initial copying can quickly be amortized
```

```
     * when comparing symbols to each other by identity
     * instead of by equality as with non-interned strings.
     *
     * We assume that string interning is predominantly going to be used
     * for relatively short strings, which mitigates the initial interning cost.
     *
     * Explicit coercion of a value to \code{Symbol} (\code{value as Symbol})
     * translates to a call to this method.
     * If necessary, this is preceded by implicit coercion to \code{String}
     * (\code{value as String as Symbol}).
     */
    public static final function fromString(string :String) :Symbol;

    override public function equal(other :*):bool
    {
        return this.identical(other);
    }
    ...
}
```

All string literals are already `Symbol` instances. Additional `Symbol` instances can be produced from Strings by explicit coercion, as shown below or by direct calls to the above constructor.

```
function world():Symbol { return "World"; }

let s1 :Symbol = "Hello"; // OK, because string literals are Symbols already
let s2 :Symbol = "Hello" + world(); // Error, concatenating computed symbols produces a String

let string1 :String = "Hello" + world();
let string2 :String = "Hello" + world();
print(string1 == string2); // true
print(string1 === string2); // false

let symbol1 :Symbol = string1 as Symbol;
let symbol2 :Symbol = string2 as Symbol;
print(symbol1 == symbol2); // true
print(symbol1 === symbol2); // true
```

`Symbol` extends `String`, but `String` is otherwise not extensible by user code without certain restrictions, which we explain in the following section.

5.2 Restricted Classes

Generalizing the concept behind class `Symbol`, we introduce the keyword `restricted` as class attribute. A *restricted* class can be extended by subclasses, but those must not declare any additional instance fields. Furthermore, every subclass of a restricted class must be a restricted class, too.

Class `String` is a restricted class and class `Symbol` extends it (see above).

Restricted classes can be used for encoding relevant invariants, which can thus be statically guaranteed by the type checker. That a string is guaranteed to be interned when typed as a `Symbol` is just one example. Other conceivable (user-defined) examples are: strings that comply with a certain lexical grammar (e.g. ‘‘well-formed’’ identifiers), lists that are assumed to be sorted, acyclical graphs, etc. Of course, the user has

to be careful that the guaranteed invariants actually hold and cannot be broken by mutation of constituent variables.

5.3 Enumeration Types

In AS3, finite sets of discrete, enumerated values are customarily represented by string constants, which is a poor choice for various obvious reasons. Alternatively, one can use integer numbers, but this is not ideal either. There is a pattern that provides a more type-safe and comfortable encoding, but it requires a lot of boiler plate code and strict adherence to certain protocol. Example:

```
public final class RbgColor // AS3 enum pattern code:
{
    private var _ordinal:int;

    public function get ordinal():String
    {
        return _ordinal;
    }

    private var _name:String;

    public function get name():String
    {
        return _name;
    }

    public function RgbColor(ordinal:int, name:String)
    {
        _ordinal = ordinal;
        _name = name
    }

    public static const RED:RgbColor = new RgbColor("red");
    public static const GREEN:RgbColor = new RgbColor("green");
    public static const BLUE:RgbColor = new RgbColor("blue");
}

var myColor:RbgColor = someCondition() ? RgbColor.BLUE : someOtherColor();
...
switch (myColor.ordinal)
{
    case RED:
        ...
        break;
    case GREEN:
        ...
        break;
    default:
        trace(myColor.name);
        break;
}
```

In AS4, this will soon be condensed as follows:

```
public enum Color { // AS4 code:
    RED, GREEN, BLUE
}

var myColor :RbgColor = someCondition() ? RgbColor.BLUE : someOtherColor();
...
switch (myColor) {
    case RED:
        ...
        break;
    case GREEN:
        ...
        break;
    default:
        trace(myColor);
        break;
}
```

There are more elaborate patterns for enum constants that have extra payload properties, and there are equally powerful syntactic enhancements in languages such as Haxe, C# , Java, and others. We will take further inspiration from these and later extend AS4's enum capabilities accordingly as an extension of the above.

5.4 Abstract Classes and Methods

The new keyword `abstract` is used as an attribute for both classes and methods. An *abstract* class cannot be instantiated directly. Only its non-abstract subclasses can.

An *abstract* method must not have a function body and it must not be native. The first non-abstract subclass of its declaring class must override it with a full method.

Constructor functions cannot be abstract.

A class that contains any abstract method must be declared as an abstract class.

5.5 Constructor, Method, and Function Overloading

AS4 will allow several different constructors with different parameter lists in the same class. And then, more generally, methods or functions with different function signatures can also have the same name, under certain conditions (to be specified later). This is also known as ad-hoc polymorphism, in that the compiler disambiguates direct calls to such methods or functions by discerning function signatures at compile time.

5.6 ActionScript Workers

We intend to reintroduce workers and to allow them in every deployement mode in AS4, not just in JIT-based deployments as in AS3. It is likely that we will update the Worker API to better leverage AS4 language features.

6 Future Features

Last but not least, let us briefly look further ahead towards features that we have not fully designed yet, but that we are particularly keen on introducing to AS4.

Immutable Data Structure Types

User-Defined Value Types

Generics (Parameteric Polymorphism)

Collection Library

Module System

Multi-Threading

Nested Transactions on Versioned Data Structures

Data Parallelism

We will elaborate on these once we have completed designs in hand.

A Implementation Notes

A.1 Automatic Boxing and Unboxing

In AS4 we distinguish between value types, reference types, and `*`. Inhabitants of value types are directly laid out in memory, registers, and on the stack, and are compared by value; and furthermore, their types are tracked by the compiler. Inhabitants of reference types are represented by pointers into memory, and are compared by reference; and furthermore, their types are manifest in the payload.

The `*` type is a (tagged) union of value types and reference types. Inhabitants of `*` are represented by a pointer into memory (''boxing''). (Crucially, this means that unlike in AS3, inhabitants of value types and inhabitants of `*` always have distinct representations, and are never confused. Type inference at the source-code level ''unboxes'' as many inhabitants of `*` as possible, but the run-time does not need to worry about such type inference.) Boxing a value makes the type manifest, and boxing a reference is a no-op. Furthermore, inhabitants of `*` behave exactly like their unboxed versions modulo boxing.

Subtyping and conversion are distinct notions in AS4. Subtyping does not change representation and semantics; e.g., class B is a subtype of class A if B is a subclass of A, so a reference of type B can be freely passed to a context of type A.

Implicit conversion, on the other hand, changes representation or semantics; e.g., `int` is implicitly convertible to `double` or `*`. Crucially, the compiler makes all implicit conversions explicit (via coercions). This means that the runtime works on explicitly typed code (possibly with coercions): it never needs to know about implicit conversions. In particular, the compiler cleanly separates fast paths and slow paths. So the runtime can generate fast machine code for statically typed parts of the source code without any type inference (and is *not* obligated to generate optimized machine code for dynamically typed parts of the source code).

A.2 Verification

As a result of the above separation of duties of the compiler and the run-time, a bytecode verifier for AS4 (unlike AS3) need not do any type inference and/or bytecode rewriting. All it does is typecheck the bytecode, following the typechecking performed by the source-level compiler.

This model enables a source-level compiler to freely and aggressively perform type-based optimizations: for example, it can rely on the types of library classes that a program is compiled against, even those such classes may not be included in the generated bytecode. At run time, those type assumptions are validated by the verifier, which means that different versions of such library classes can be dynamically loaded and the generated bytecode continues to run as long as the types it relies on at compile-time remain invariant at run-time.

B Base Classes and Interfaces

The following listings of are primarily just illustrations of what the eventual base classes and interfaces might look like. These sketches are not to be taken as promises of any specific functionality. Furthermore, this is just a subset of the classes that we intend to provide. If a familiar class from AS3 is not listed here, this neither implies that we have dropped it nor the opposite.

B.1 Core Classes and Interfaces

These at compile time predefined and at runtime preloaded core classes and interfaces can be addressed without any package qualification.

B.1.1 *

```
/**
 * The top type, i.e. super type of all types,
 * represented by <code>*</code> in source code.
 * Also the super class of all classes.
 */
internal abstract class *
{
    protected function *
    {
    }

    public abstract function toString() :String
    {
        throw new TypeError();
    }

    /**
     * @return an identity hash value for the given receiver.
     *
     * It is guaranteed that repeated calls with the identical receiver
     * will always return the same value.
     *
     * The probability that two such hash values are the same for different receivers
     * is very low for instances of <code>Object</code>
     * and reasonably low for numeric values.
     * In neither case is it zero.
     */
```

```
    public abstract function identityHash() :ulong;

    /**
     * @return whether the <code>other</code> value is identical to the receiver.
     *
     * In contrast to the <code>===</code> operator,
     * this method never throws a type error,
     * but returns <code>false</code> instead.
     */
    public abstract function identical(other :*) :bool;
}
```

B.1.2 Primitive

```
/**
 * Common part of numeric types..
 */
internal abstract class Primitive extends *
{
    protected Primitive()
    {
    }

    ... // implementation detail omitted
}
```

B.1.3 int

```
/**
 * The primitive value type of 32-bit integer numbers
 * and simultaneously the box class thereof.
 */
public final class int extends Primitive
{
    public static let MAX :int =  2147483647;
    public static let MIN :int = -2147483648;

    /**
     * Invoked when a <code>*</code>-typed <code>int</code> value
     * is the receiver of <code>identical()<code>.
     * <p>
     * <code>let v :int = ...;</code>
     * <code>let x :* = v;<code>
     * Then <code>x.identical()</code> lands here by virtual dispatch.
     */
    override public function identical(other :*) :bool { ... }

    /**
     * Invoked when a <code>*</code>-typed <code>int</code> value
     * is the receiver of <code>identityHash()<code>.
```

```
     * <p>
     * <code>let v :int = ...;</code>
     * <code>let x :* = v;<code>
     * Then <code>x.identityHash()</code> lands here by virtual dispatch.
     */
    override public function identityHash() :ulong { ... }

    /**
     * Invoked when a <code>*</code>-typed <code>int</code> value
     * is the receiver of <code>toString()<code>.
     * <p>
     * <code>let v :int = ...;</code>
     * <code>let x :* = v;<code>
     * Then <code>x.toString()</code> lands here by virtual dispatch.
     */
    override public function toString() :String { ... }

    public static function abs(x :int) :int { ...}

    public static function max(x :int, y :int) :int
    {
        return (x > y) ? x : y;
    }

    public static function min(x :int, y :int) :int
    {
        return (x < y) ? x : y;
    }

    ... // more API
}
```

B.1.4 double

```
public final class double extends Primitive
{
    /**
     * The smallest positive value
     * that is distinguishable from zero on the current platform.
     */
    public static function get MIN() :double { ... }

    public static let MAX    :double = 1.79769313486231571e+308;

    public static let E      :double = 2.718281828459045;
    public static let LN10   :double = 2.302585092994046;
    public static let LN2    :double = 0.6931471805599453;
    public static let LOG10E :double = 0.4342944819032518;
    public static let LOG2E  :double = 1.442695040888963387;
    public static let PI     :double = 3.141592653589793;
    public static let SQRT2  :double = 1.4142135623730951;
```

```
/**
 * Implements <code>identical()</code> for primitive double values.
 * <p>
 * <code>let v :double = ...;</code>
 * <code>v.identical(other)</code> translates to <code>doubleIdentical(v, other)</code>
 */
public static function doubleIdentical(value :double, other :*) :bool { ... }

/**
 * Invoked when a <code>*</code>-typed <code>double</code> value
 * is the receiver of <code>identical()<code>.
 * <p>
 * <code>let v :double = ...;</code>
 * <code>let x :* = v;<code>
 * Then <code>x.identical()</code> lands here by virtual dispatch.
 */
override public function identical(other :*) :bool { ... }

/**
 * Invoked when a <code>*</code>-typed <code>double</code> value
 * is the receiver of <code>identityHash()<code>.
 * <p>
 * <code>let v :double = ...;</code>
 * <code>let x :* = v;<code>
 * Then <code>x.identityHash()</code> lands here by virtual dispatch.
 */
override public function identityHash() :ulong { ... }

/**
 * Invoked when a <code>*</code>-typed <code>double</code> value
 * is the receiver of <code>toString()<code>.
 * <p>
 * <code>let v :double = ...;</code>
 * <code>let x :* = v;<code>
 * Then <code>x.toString()</code> lands here by virtual dispatch.
 */
override public function toString() :String { ... }

public function abs() :double { ... }

public static function max(x :double, y :double) :double
{
    return (x > y) ? x : y;
}

public static function min(x :double, y :double) :double
{
    return (x < y) ? x : y;
}
```

```
    public native function acos  (x :double) :double;
    public native function asin  (x :double) :double;
    public native function atan  (x :double) :double;
    public native function ceil  (x :double) :double;
    public native function cos   (x :double) :double;
    public native function exp   (x :double) :double;
    public native function floor (x :double) :double;
    public native function log   (x :double) :double;
    public native function round (x :double) :double;
    public native function sin   (x :double) :double;
    public native function sqrt  (x :double) :double;
    public native function tan   (x :double) :double;

    public native static function atan2 (y :double, x :double) :double;
    public native static function pow   (x :double, y :double) :double;

    public native static function random() :double;

    ... // more API
}
```

B.1.5 Other Primitive Types

Class `uint`, `ulong`, and `long` look quite similar to `int`. Class `byte` and `bool` also follow this general pattern, with natural differences in which operations are offered. Class `float` looks quite similar to `double`.

B.1.6 Array

```
/**
 * Super class of all array types (e.g. []int).
 * Hosts common element type-independent functionality.
 *
 * Will be parameterized as soon as this becomes possible.
 */
internal class Array extends Object
{
    // ASC intrinsifies this method to a bytecode.
    /**
     * @return the number of elements in this array
     */
    public final native function get length() :int;

    /**
     * Copies elements from the specified range in the given source array
     * into this array at the specified index.
     *
     * In case an error is thrown, no copying takes place.
     *
     * @param source The array to copy from
```

```
     * @param fromIndex The index in the source array to start copying from
     * @param toIndex The index in this array to start copying to
     * @param numElements The number of elements to copy in sequence
     * @throws ArgumentError If the source element type is not
     *                       a sub-type of the target element type
     * @throws RangeError If any of the specified copying actions violate an array boundary
     */
    public final function copyRangeFrom(source :Array, fromIndex :int,
                                        toIndex :int, numElements :int) :void { ... }

    /**
     * Copies elements from this array into the given array.
     * Starts at index 0 in both arrays.
     * Stops at whichever array length is less.
     *
     * @throws ArgumentError If the element type of this array is not
     *                       a sub-type of the target element type
     */
    public final function copyFrom(source :Array) :void { ... }

    // We may generalize this concept later for other kinds of objects,
    // once we have interfaces with type parameters
    /**
     * @return A shallow copy of this array
     */
    public final function clone() :Array { ... }

    ... // more API
}
```

B.1.7 ArrayList

This class is very similar to `Vector` in AS3 and essentially has the same role in AS4.

```
/**
 * A list that can be indexed like an array.
 */
public class ArrayList extends Object
{
    // ASC intrinsifies this method to a bytecode.
    /**
     * @return the number of elements in this array list
     */
    public final native function get length() :int;

    /**
     * Change the length of this array list to the given number of elements.
     *
     * @param value The requested new length
     *
     * @throws RangeError If the length has already been fixed
```

```
                        and the requested length differs from the current length
     * @throws RangeError If the argument is negative
     */
    public native function set length(value :int) :void;

    /**
     * Prohibit length mutation of this array list going forward.
     *
     * @return an array that contains the same elements as the receiver
     *
     * Repeated calls may return the identical array.
     * The implementation may or may not have copy overhead.
     * Aliasing may occur when accessing the resulting array
     * and the original array list as well.
     *
     * This is a temporary measure for our first release.
     * We will follow up with a parametrically typed version later.
     * This later version will also always provide no-copy aliasing.
     */
    public native function fixLength() :*;

    /**
     * @return Whether the length of this array list is fixed
     */
    public function isLengthFixed() :bool { ... }

    /**
     *  @return an array list that is populated with the array elements
     *
     * This is a temporary measure for our first limited language release.
     * We will follow up with a parametrically typed alternative constructor later.
     */
    public static native function create(array :*) :*;

    ... // more API; much like Vector in AS3
}
```

B.1.8 String

```
public restricted class String extends Object
{
    /**
     * Create a string from the given array of 32-bit unicode "code points",
     * i.e. character codes.
     *
     * @param The sequence of unicode characters to be represented by this string
     */
    public native function String(codePoints :[]uint);

    // ASC intrinsifies this method to a bytecode.
    /**
```

```
     * @return The number of unicode characters in this string
     */
    public native function get length() :int;

    /**
     * @return Whether the number of unicode characters in this string is zero
     */
    public function get isEmpty() :bool
    {
        return length == 0;
    }

    /**
     * @return A contents-based hash function value for this string
     *
     * It is guaranteed that invocations on multiple strings all return the same hash value
     * if these strings are equal according to method <code>String.equal()</code>
     *
     *  @see equal()
     */
    override public native function equalityHash() :ulong { ... }

    /**
     * @return Whether The given other string represents
     *                 the same sequence of unicode characters as this string
     */
    override public native function equal(other :*) :bool;

    /**
     * Retrieve a single character code at a specific position in this string.
     *
     * @param The index of the code point to retrieve
     * @return A 32-bit Unicode "code point", i.e. character code
     * @throws RangeError If the index is out of the range
     *                    from <code>0</code> to <code>(this.length - 1)</code>.
     */
    public native final function codePointAt(index :int) :uint;

    /**
     * @return the codepoints encoded by the string as an array of Unicode 32 code point values
     */
    public native final function toCodePointArray() :[]uint;

    ... // a lot more API follows here; much like String in AS3
}
```

B.2 The Reflection API

For now, AS4 only provides basic reflection functionality that does not require explicit treatment of security aspects in its API. All access to non-public properties is turned off. We will further elaborate on these reflection features in subsequent releases of AS4.

All of the following classes are declared in the special AS4 package named `type`.

B.2.1 MetaDataHolder

```
/**
 * A language element that may be annotated with metadata,
 * which can be queried, returning an array of inspectable objects.
 *
 * @see Type
 * @see Method
 * @see Field
 */
public interface MetadataHolder
{
    /**
     * @return Deep copies of all metadata objects that this holder has been annotated with
     *
     * Note to implementors:
     *    The return value must never be null.
     *    Return an empty array when no metadata are present.
     */
    function metadata() :[]Object;

    /**
     * Look up metadata associated with a specific annotation class for this holder.
     *
     * @param The class whose metadata annotation to look up
     * @return A metadata object for the specified annotation class
     *         or null if no such annotation exists
     *
     * If several annotations with the same class exist, only the first one of them will be reported.
     */
    function metadataFor(metadataClass :Class) :Object
}
```

B.2.2 Type

```
/**
 * Runtime representation of a type,
 * providing basic reflection.
 * <p>
 * @see ObjectType
 * @see Function
 */
public class Type implements MetadataHolder
{
    /**
     * @return The most precise actual runtime type of the given <code>value</code>
     * @param value The value of which its runtime type is being queried here
     * <p>
```

```
     * The result is always an instance of <code>FunctionType</code>,
     * <code>ArrayType</code>, or <code>Class</code>.
     * The latter holds for primitive values.
     * For instance, <code>Type.get(3)</code> returns <code>:int</code>,
     * <code>Type.get(true)</code> returns <code>:bool</code>.
     */
    public static native function get(value :*) :Type;

    /**
     * @see MetadataHolder
     */
    public final native function metadata() :[]Object;

    /**
     * @see MetadataHolder
     */
    function final native metadataFor(metadataClass :Class) :Object;

    /**
     * @return The fully qualified name describing this type
     */
    public final function name() :String { ... }

    public static function fromName(name :String) :Type { ... }
    override public function toString() :String { ... }
    public native function superType() :Type;
    public native function isSubTypeOf(superType :Type) :bool;
    ...
}
```

B.2.3 ObjectType

```
/**
 * Common part of the runtime representation of a class or an interface,
 * providing basic reflection.
 */
public abstract class ObjectType extends Type
{
    public native function hasInterface(i :Interface) :bool;
    public native function interfaces() :[]Interface;

    public final native function instanceMethods() :[]Method;
    ...
}
```

B.2.4 Class

```
/**
 * Runtime representation of a class,
 * providing basic reflection.
```

```
 */
public final class Class extends ObjectType
{
    /**
     * @return The actual runtime class of the given <code>value</code>
     * @param value The value of which its runtime class is being queried here
     */
    public static native function get(value :Object) :Class

    /**
     * Much faster check than Type.isSubTypeOf().
     * @return whether this class is a sub-class of the given class
     * Note that the sub-type relationship is also regarded as true if both classes are identical.
     */
    public native function isSubClassOf(c : Class) :bool;

    public native function get superClass() :Class;
    public native function staticMethods() :[]Method;
    public native function staticFields() :[]Field;
    public native function instanceFields() :[]Field;
    ...
}
```

B.2.5 Interface

```
/**
 * Runtime representation of an interface,
 * providing basic reflection.
 */
public final class Interface extends ObjectType
{
    ...
}
```

B.2.6 Field

```
/**
 * A field in a class.
 */
public class Field implements MetadataHolder
{
    /**
     * @see MetadataHolder
     */
    public final native function metadata() :[]Object;

    /**
     * @see MetadataHolder
     */
    function final native metadataFor(metadataClass :Class) :Object;
```

```
    public function get declaringClass() :Type { ... }
    public function get name() :String { ... }
    public function get type() { ... }
    ...
}
```

B.2.7 Method

```
/**
 * A method in a class or an interface.
 */
public class Method implements MetadataHolder
{
    /**
     * @see MetadataHolder
     */
    public final native function metadata() :[]Object;

    /**
     * @see MetadataHolder
     */
    function final native metadataFor(metadataClass :Class) :Object;

    public function get declaringType() :Type { .. }
    public function get name() :String { ... }
    public function get signature() :FunctionSignature { ... }
    ...
}
```

B.2.8 FunctionSignature

```
/**
 * A function signature, comprising of a parameter type list and a return type.
 */
public final class FunctionSignature
{
    public static function create(parameterTypes :[]Type, returnType :Type) :FunctionSignature { ... }
    public final function get parameterTypes() :[]Type { ... }
    public final function get returnType() :Type { ... }
    public final function name() { ... }
    ...
}
```

B.2.9 FunctionType

```
/**
 * Runtime representation of a function closure type,
 * providing basic reflection.
```

```
 */
public final class FunctionType extends ObjectType
{
    public function get signature() :FunctionSignature { ... }
    ...
}
```

B.2.10 ArrayType

```
/**
 * Runtime representation of an array type,
 * providing basic reflection.
 */
public final class ArrayType extends ObjectType
{
    public final function get elementType() :Type { ... }

    /**
     * @return The actual array type of the given <code>value</code>
     * @param value The value of which its array type is being queried here
     */
    public static function get(value :Array) :ArrayType { ... }

    /**
     * @return A new instance of this array type,
     *         with the given length and with all elements initialized
     *         to zero-like default values
     */
    public native function createInstance(length :int) :Array;

    ...
}
```