# Actionscript 3 Language Specification

Editor:
Jeff Dyer

Co-editors:
Edwin Smith
Gary Grossman
Francis Cheng

Revision History:

| | |
|---|---|
| November 14, 2005 | Intial proposal submited to Ecma TC39/TG1 |

## Purpose and status of this document

This document defines the Actionscript 3 (AS3) language that is designed to be forward compatible with the next edition of ECMA-262. It serves as Macromedia's initial proposal for the definition of Ecma-262 edition 4.

## Contents

# 1 Tutorial introduction

An Actionscript program consists of zero or more package definitions followed by zero or more directives, which includes non-package definitions and statements. Statements inside and outside of package definitions are evaluated in order, independent of their nesting inside a package.

## 1.1 Hello world

The following sections show various ways to implement simple programs such as the familiar 'hello, world' program in Actionscript 3. Here is the simplest of all.

```
print("hello, world")
```

This is a single expression statement that calls a function named print with the argument that is a literal string "hello, world". An expression statement does nothing but execute an expression.

## 1.2 Expressions

Here are some examples of expressions:

```
x = 1 + 2

x = y()

x = y..z

x = o.ns::id
```

Expressions evaluate to values.

- `1+2` evaluates to `3`
- `y()` evaluates to the result of calling the function `y` with no arguments
- `y..z` evaluates to the set of all properties identified by z in the value of y and y's descendants.
- `o.ns::id` evaluates to the value of property `'ns::id'` of the value of `'o'`

## 1.3 Statements

Statements are executed in the order that they appear in a block. Some statements change control-flow by abrupt completion (e.g. break and continue) or iteration (e.g. while and do).

```
for each ( x in o ) {
    print(x)
}
```

## 1.4 Variables

```
var x = 10
const PI = 3.1415
```

Variables define properties whose values can change at runtime. They can be defined with either the `var` keyword or the `const` keyword. A variable that is defined with the `var` keyword may be assigned to by any code that can access it. A variable that is defined with the `const` keyword may only be written to while its owning object is being initialized.

## 1.5 Functions

```
function hello() {
    print(“hello, world”)
}
hello()
```

Functions define properties whose values can be called. Depending on where a function is defined it results in a property whose value is a function closure or a method. A function closure is a first class object that can be treated as a collection of properties or a callable object. A method is tightly bound to the object that it is associated with. The `this` reference of a function is bound to the base object of the call expression, or the global object if none is specified.

[editorial: add example]

A method is a function that is tightly bound to an object. Methods can be extracted from their instance, but unlike function closures the value of `this` always refers to the instance it is extracted from.

## 1.6 Classes

```
class Greeter {
    var saying = “hello, world”
    function hello() {
        print(saying)
    }
}
var greeter : Greeter = new Greeter
greeter.hello()
```

A class is an object that can be used as a constructor of instances that share the same type and properties. Class definitions are used to define the fixed properties of a class object and shared by its instances. Property definitions that are marked with the `static` attribute become properties of the class object, and those that are not become properties of instances of the class.

Class and instance properties are either slots or methods. A method is defined by a function definition inside a class definition. A method has a definition (called a method trait) that is shared between all instances of the same type. Unlike an ordinary function object, a method is tightly bound to the object it is associated with. Whenever and however it gets invoked, the meaning of the expression `this` is always the same. In fact, methods can be extracted from their instance and treated as first class objects (called method closures), much like function objects can be. There is one important difference between a function closure and a method closure. With a method closure the `this` reference gets bound into the object so that whenever it is invoked the original `this` is used. With a function closure `this` is generic and will refer to any object the function happens to be associated with when it is invoked.

Slots are defined by variable definitions inside a class definition. An instance variable has a definition (called a slot trait) that is shared between all instances of the same type, but a unique location in each object.

## 1.7 Interfaces

```
interface Greetings {
    function hello()
    function goodmorning()
```

```
}

class Greeter implements Greetings {
    public function hello() {
        print(“hello, world”)
    }
    public function goodmorning() {
        print(“goodmorning, world”)
    }
}
var greeter : Greetings = new Greeter()
greeter.hello()
```

An interface defines a contract between an instance and code that uses that instance. When a class implements an interface, it is telling the world that it guarantees that it will provide the methods declared in that interface. An implementing method must be declared `public`, in which case it will implement all unimplemented interface methods with the same identifier.

## 1.8 Packages

```
package actors {
    public class Greeter {
        public function hello() {
            print(“hello, world”)
        }
    }
}
import actors.Greeter
var greeter : Greeter = new Greeter
greeter.hello()
```

In this example, the import directive makes the class Greeter visible to the global code that contains the import directive. Packages are useful for organizing frameworks (or toolkits, or APIs) into set of related definitions (e.g. classes, namespaces, interfaces, functions, variables). Client code can import all or parts of a package to get access to the functionality it provides without flooding its global namespace with unneeded names. Packages in Actionscript are very similar to packages in Java and namespaces in C# and C++.

## 1.9 Namespaces

```
package actors {
    public namespace English
    public namespace French
    public class BilingualGreeter {
        English function hello() {
            print("hello, world")
        }
        French function hello() {
            print("bonjour, le monde")
                    // ? French speakers, correct me here
        }
    }
}
import actors.*
var greeter : BilingualGreeter = new BilingualGreeter
use namespace English     // Make all identifiers in the English namespace
                          // visible
greeter.hello()           // Invoke the English version
greeter.French::hello()   // Invoke the French version
```

Namespaces are useful for controlling the visibility of a set of properties independent of the major structure of the program. Packages, classes and interfaces along with their implicitly defined access control namespaces allow authors to control the visibility of names in parallel with the organization of those packages, classes and interfaces. But it is sometimes necessary to control the names independent of the lexical structure of a program. Examples of this include:

- making the public interface of a set of classes look different to different client modules
- evolving a class over time without changing the behavior of existing programs
- providing privileged access to a limited set of clients

Use packages to give or gain access to a set of features. Use namespaces to give or gain access to a particular facet, version, or privileges independent of the structure of a program.

# 2 Design perspective

It is sometimes difficult to understand design decisions without understanding the perspective of the designers. Here are the major viewpoints that have grounded the design changes introduced in ed. 4.

## 2.1 Compatibility with existing programs

Ecmascript was originally designed for and used by consumers of host objects models. Being one of the most widely used programming languages, it is important that existing programs continue to work as before in systems that are updated to support the new language definition.

*Therefore, programs written for ed. 3, compact profile, or E4X must behave the same in ed. 4*

## 2.2 Compatibility with existing object models

Through 10 years of use Ecmascript has come under great pressure to also become a language for creating object models. This is a natural consequence of the need for application and tool developers to extend and override the functionality of the built-in objects provided by host environments. A few examples include: HTML, Flash, Acrobat, and VoiceXML

These embeddings contain host objects with behaviors that can only be approximated with the features of ed. 3, and even then in a way that is inefficient and fragile.

*Therefore, make it possible to create object models such as the ed. 3 built-ins, HTML DOM and Actionscript API in ed. 4. Moreover, make it natural to give these object models behavior like the existing object models, as well as make them robust and efficient.*

## 2.3 Controlling the visibility of names

It is a well known problem that naming conflicts arise when independently created libraries are used by a single application. It is also common that the meaning of a name must be different for different uses of a single component.

*Therefore, minimize the occurrence of naming conflicts when independently created libraries are used by a single application, and make it possible to resolve those conflicts when they do occur. Furthermore, make it possible for users to select the meaning of names between versions and uses.*

## 2.4 Choosing between reliability and flexibility

Whereas the original purpose of Ecmascript was to provide a scripting language for automating web pages and other hosted applications where lenient runtime behavior is preferred and scripts are small enough that performance is often not a concern, libraries written in Ecmascript can be very large and complex, and be constrained by aggressive performance requirements. These libraries are often created ahead of time using IDEs and stand-alone compilers. In this case developers are willing to give up some flexibility to be guaranteed that certain kinds of errors will not occur at runtime, and that their code will run as efficiently as possible.

Also, it is desirable when targeting low powered platforms to minimize the amount of processing that must occur to execute programs on the client.

*Therefore, allow developers to trade flexibility and compatibility for reliability and efficiency by choosing a well defined subset of Ecmascript that can be compiled ahead-of-time for more aggressive compile-time semantic analysis and optimization.*

# 3 Phases and dialects of interpretation

There are two dialects of the language described by this specification, one a subset of the other. These languages differ only in that one has additional verification rules. The more permissive language is called the standard dialect, and the more restrictive language is called the strict dialect.

## 3.1 Parsing

The parsing phase translates the source code of a program into an internal format suitable for verification. The syntax rules of the language are described using grammar productions throughout this specification.

## 3.2 Verifying

The verification phase ensures that the program obeys the static semantics of the language. In the standard dialect verification may be done anytime before a construct is first evaluated. In the strict dialect verification must happen before any part of the program is evaluated.

The differences in the verification rules of the standard dialect and the strict dialect mean that some programs that would verify in the standard language will not verify in the strict language. However, all programs that verify in the strict language will verify and run with the same behavior in the standard language.

### 3.2.1 Compile time constant expressions

A compile time constant expression is an expression whose value can be determined at compile time (during verification), before any part of the program has been executed. Compile time constant expressions consist of the following sub-expressions:

- Literals such as null, Number, Boolean and String literals
- References to properties whose values are compile-time constants
- Operators whose result can be computed at compile time

Expressions in certain contexts are required to be compile time constant expressions.

- type annotations
- default parameter initialisers
- inheritance clauses references
- attributes

Of these, inheritance clause references and attributes must not have forward references.

## 3.3 Evaluating

The evaluation phase takes the parsed, verified program and evaluates it to produce side effects in its host environment, and a final value. The semantics of evaluation are the same for both dialects of the language.

## 3.4 Strict verification

The goal of strict mode is reliability of new programs. The strict language is a subset of the standard language by adding three kinds of constraints:

- Expressions have static types and type errors are verification errors
- Common programming errors are caught by additional verification rules
- Verification errors are reported ahead-of-time

### 3.4.1 Type errors

Here is an example of a program that is valid in the standard dialect but not valid in the strict dialect,

```
class A {}
class B extends A {}
var a : A = new B
var b : B = a        // type error, static type of 'a' is A,
                     // which is incompatible with B
```

In the standard dialect this program has no error, since type errors are runtime errors and the runtime value of `a` is an instance of `B`, which is clearly a member of the type B.

### 3.4.2 Strict errors

The strict dialect adds various semantic errors to catch common programming mistakes that are allowed in the standard dialect for the sake of compatibility and flexibility.

Strict mode only verification errors fall in these categories:

- function call signature matching
- duplicate definition conflicts
- unbound references
- dynamic addition of properties on sealed objects
- writing to const variables
- deleting fixed properties
- comparison expressions with incompatible types
- unfound packages

# 4 Fundamental Concepts

This section describes the abstract data structures and procedures used throughout this specification..

## 4.1 Notation

Here is an informal description of the notation used in this chapter...

The definitions labeled `struct`, `type` and `proc` are for specification purposes only. None are directly accessible to program code. The definitions labeled class are built-in objects that are accessible to program code. However, built-in class definitions might have intrinsic properties that are inaccessible to program code.

## 4.2 Data

### 4.2.1 Objects

Every program visible value is an object. An object is a collection of properties.

```
struct Object {
    delegate   : Object
    properties : Map<Name,Object>
    type       : Traits
    slots      : List<Object>
}
```

### 4.2.2 Names

```
struct Name {
    namespace  : Namespace
    identifier : String
    attr       : Boolean
}
```

### 4.2.3 Prototypes and delegates

A prototype object is an object that becomes the implicit delegate shared by all instances of a particular class or function. An object's delegate is either of the same type as that object or of Object. A class prototype is an instance of that class, while the prototype's delegate is an instance of Object.

### 4.2.4 Properties

A property associates a name with a value or method. A method can be either a get or set accessor or an ordinary method. Fixed properties cannot be redefined or deleted. Dynamic properties are created at runtime and can be redefined and deleted. Internally, fixed properties are expressed as traits. Dynamic properties are expressed as a map between names and values.

### 4.2.5 Slots

A slot is a location inside an instance used to store the value of a variable property. A slot is allocated for each variable declaration.

```
slots : List<Object>
```

### 4.2.6 Traits

A trait is a fixed property shared by all instances of the same type. The collection of traits defines the invariants of the object's type. For this reason use the traits object to describe the type of an object. Traits are declared in the definition of the class used to create an object.

```
class A
{
    var x
    function m() { }
    function get y() { return 10 }
    function set y(v) { }
}
```

Each member of this class definition causes a trait to be added to the traits object for instances of A. When an instance is created by class A, the resulting object has the properties x, m and y, implemented by traits for var x, function m, function get y and function set y.

Traits express the type of an instance. All traits are copied down to the derived traits objects. All traits must be implemented. Interface members are abstract and so there traits must be implemented in any class that inherits them.

```
struct Traits {
    super      : Traits
    interfaces : Traits[]
    traits     : Trait[]
}

type Trait = {
    ConstantTrait, SlotTrait, GetterTrait, SetterTrait, MethodTrait
}

struct ConstantTrait {
      name   : Name
      type   : Traits
      value  : Object
}

struct SlotTrait {
      name   : Name
      type   : Traits
      slotid : Uint
}

struct GetterTrait {
      name   : Name
      type   : Traits
      method : Method
}

struct SetterTrait {
      name   : Name
      type   : Traits
      method : Method
}

struct MethodTrait {
      name       : Name
      resulttype : Traits
      paramtypes : List<Traits>
```

```
        method     : Method
    }
```

### 4.2.7 Methods

```
struct Method {
    param_types : List<Traits>
    return_type : Traits
    code : List<Byte>
    exceptions : List<Exception>
    traits : List<Trait>
    has_arguments : Boolean
    has_activation : Boolean
    has_restargs : Boolean
    optional_count : Uint
    optional_values : List<Object>
}

struct Exception {
    start : Uint
    stop : Uint
    target : Uint
    type : Traits
}

struct MethodClosure {
    this : Object
    method : Method
}
```

### 4.2.8 Lexical environment

```
struct Environment {
    scopes : List<Frame>
}

type Frame = { GlobalFrame, Class, Instance, WithFrame, Activation }
```

### 4.2.9 References

A Reference is an internal value used to express the evaluation of an expression that includes a name. References can be unqualified or qualified by a namespace, involve multiple qualified names (multiname), or target a specific object (dot or bracket reference). References resulting from super expressions are limited to the super type of the object containing the current code.

```
struct ObjectReference {
    base : ObjectOpt
    multiname : List<Name>
    limit : Traits
    attr : Boolean
}

struct LexicalReference {
    env : Environment
    multiname : List<Name>
    limit : Traits
    attr : Boolean
}

type Reference = { ObjectReference, LexicalReference }

struct LimitedBase {
    base : Object
    limit : Traits
}
```

### 4.2.10 Native classes

#### 4.2.10.1 Object

An object has intrinsic traits to support call and construct expressions.

```
class Object
{
    intrinsic const typeofString : String
    intrinsic const sealed : Boolean
    intrinsic const defaultValue : Object
    intrinsic const defaultHint : String
    intrinsic function hasProperty(name:Name) : Boolean
    intrinsic function read(name:Name,limit:Type) : Object
    intrinsic function write(name:Name,limit:Type,value:*,
        expand:Boolean) : Boolean
    intrinsic function expand(name:Name,value:*,
        super:Boolean) : Boolean
    intrinsic function delete(name:Name,super:Boolean) : Boolean
    intrinsic function enumerate() : List
}
```

#### 4.2.10.2 Function

A function has intrinsic traits to support call and construct expressions.

```
class Function extends Object
{
    intrinsic const environment : Frame[]
    intrinsic function call(this,...args) : *
    intrinsic function construct(...args) : *
}

class MethodClosure extends Function
{
    intrinsic const boundThis : Object
    intrinsic function call(...args) : Object
    intrinsic function construct(...args) : Object
}

class PrototypeFunction extends Function
{
    public var prototype : Object
}
```

#### 4.2.10.3 Class

A class has intrinsic traits to support explicit conversion and construct expressions.

```
class Class extends Object
{
    intrinsic const environment : Frame[]
    intrinsic const super : Class
    intrinsic const itraits : Traits
```

```
    intrinsic function call(...args) : Object
    intrinsic function construct(...args) : Object
    public const prototype : Object
}
```

#### 4.2.10.4 Namespace

A namespace has intrinsic traits to support comparison with other namespaces.

```
class Namespace extends Object
{
    public const name : String
    public const prefix : String
}
```

## 4.3 Procedures

The notation we use to describe abstract procedures is a pseudo-Actionscript. A few differences from real Actionscript are: (1) missing keywords in definitions; (2) access to internal data structures and intrinsic names; and (3) the use of non-existent values such as *none*.

### 4.3.1 Reading

```
readReference( refOrObj : * ) : Object
{
    if( refOrObj is LexicalReference )
    {
        i = 0
        while( i < env.size() )
        {
            obj    = env[i]
            name   = refOrObj.name
            traits = obj.type
            result = traits.intrinsic::read(obj,name,traits)
            if( result != none )
            {
                break
            }
            i++
        }
    }
    else
    if( ref is ObjectReference )
    {
        obj    = refOrObj.base
        name   = refOrObj.name
        traits = refOrObj.limit
        result = traits.intrinsic::read(obj,name,traits)
    }
    else
    {
        result = refOrObj
    }
    return result
}
```

### 4.3.2 Writing

```
proc writeReference( refOrObj, value )
{
    if( refOrObj is LexicalReference )
    {
        name = refOrObj.name
        i = 0
        while( i < env.size() )
        {
            obj    = env[i]
            traits = obj.type
            result = traits.intrinsic::write(obj,name,traits,value,false)
            if( result == okay )
            {
                break
            }
            i++
        }
        if( result != okay )
        {
            // obj and traits are already set to the outer scope
            result = traits.intrinsic::write(obj,name,traits,value,true)
        }
    }
    else
    if( ref is ObjectReference )
    {
        obj    = refOrObj.base
        name   = refOrObj.name
        traits = refOrObj.limit
        result = traits.intrinsic::write(obj,name,traits,value,true)
    }

    if( result != okay )
    {
        throw new Reference("unable to write to reference")
    }
}
```

### 4.3.3 Calling

```
// callReference allows the implementing type to limit the
// search for the callee to the fixed traits of this, or
// super, if necessary. E.g. class XML

callReference( refOrObj:*, args:Array ):*
{
    if( refOrObj is LexicalReference )
    {
        name = refOrObj.name
        i = 0
        while( i < env.size() )
        {
            obj    = env[i]
            traits = obj.type
            fun    = traits.intrinsic::read(obj,name,traits)
            if( fun != none )
            {
                self = null
```

```
                break
            }
            i++
        }
    }
    else
    if( refOrObj is ObjectReference )
    {
        obj    = refOrObj.base
        name   = refOrObj.name
        traits = refOrObj.limit
        fun    = traits.intrinsic::read(obj,name,traits)
        self   = obj
    }
    else
    {
        fun  = refOrObj
        self = null
    }

    // Do the call
    if( fun != none )
    {
        result = fun.intrinsic::call(self,args)
    }
    return result
}
```

### 4.3.4 Constructing

```
/* constructReference allows the implementing type to limit the
   search for the callee to the fixed traits of this, or
   super, if necessary. E.g. class XML
*/

constructReference( refOrObj:*, args:Array ) : *
{
    if( refOrObj is LexicalReference )
    {
        name = refOrObj.name
        i = 0
        while( i < env.size() )
        {
            obj    = env[i]
            traits = obj.type
            fun    = traits.intrinsic::read(obj,name,traits)
            if( fun != none )
            {
                self = null
                break
            }
            i++
        }
    }
    else
    if( refOrObj is ObjectReference )
    {
        obj    = refOrObj.base
        name   = refOrObj.name
        traits = refOrObj.limit
        fun    = traits.intrinsic::read(obj,name,traits)
```

```
            self   = obj
        }
        else
        {
            fun  = refOrObj
            self = null
        }

        // Do the call
        if( fun != none )
        {
            result = fun.intrinsic::construct(args)
        }
        return result
    }
```

### 4.3.5 Deleting

```
    deleteReference( refOrObj:* ) : Boolean
    {
        if( refOrObj is LexicalReference )
        {
            result = true  /* default result */
            i = 0
            while( i < env.size() )
            {
                obj    = env[i]
                name   = refOrObj.name
                traits = obj.type
                result = traits.intrinsic::delete(obj,name,traits)
                if( result != none )
                {
                    break
                }
                i++
            }
        }
        else
        if( ref is ObjectReference )
        {
            obj    = refOrObj.base
            name   = refOrObj.name
            traits = refOrObj.limit
            result = traits.intrinsic::delete(obj,name,traits)
        }
        else
        {
            result = refOrObj
        }
        return result
    }
```

### 4.3.6 Naming

```
    proc makeAttributeName( name:Name )
    {
        /* Set the attr flag of a name
        */
    }

    proc isAttribute( name:Name )
    {
```

```
    /* Return the attr flag of name
    */
}

proc makeMultiname( namespaces:Namespaces[], str:String )
{
    /* Create a set of qualified names from a string
       and a set of qualifiers
    */
}
```

### 4.3.7 Typing

```
proc typeOfString( obj:* ) : String
{
/*
    Return the ed. 3 typeof string
*/
}

proc resultType( fun:Function )
{
/*
    Return the result type of a function object
*/
}

proc typeOfThis( frame:ParameterFrame )
{
/*
    Return the type of this for a parameter frame
    Instance methods have type of the instance
    Function closures have a type of Object
    All others throw an exception
*/
}

proc hasThis( frame:ParameterFrame )
{
/*
    Return true if the given parameter frame is a method
    and has a bound this value
*/
}

proc referenceType(base:Object,name:Name,limit:Type,attr:Boolean)
{
/*
    Return the type of the property referenced by the
    specified parameters
*/
}
```

### 4.3.8 Converting

```
proc toString( obj:* ) : String
{
/*
    This procedure implements the semantics described in
```

```
    Ecma-262, section 9.8, ToString()
*/
}

proc toNumber( obj:* ) : Number
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 9.3, ToNumber()
*/
}

proc toInt( obj:* ) : int
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 9.5, ToInt32()
*/
}

proc toUint( obj:* ) : uint
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 9.6, ToUint32()
*/
}
```

### 4.3.9 Checking

```
proc isStrict()
{
/*
    If compiling as a strict dialect, return true
    Otherwise return false
*/
}


proc verifyType( type1:Type, type2:Type ) : void
{
/*
    The rules for type checking are describe in section 6.5

    If isStrict() and type1 is a subtype of Object {
        If type1 is not a subtype of type2 and
           type2 is not a subtype of Boolean and
           type1 is not a subtype of Number or type2
                        is not a subtype of Number
           Throw a TypeError
    }
    Else {
        Do nothing because type errors are runtime errors
    }
    Return
*/
}
```

### 4.3.10 Operating

```
proc bitwiseNot(v1:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.4.8
*/
}

proc multiply(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.5.1
*/
}

proc divide(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.5.2
*/
}

proc remainder(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.5.3
*/
}

proc add(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.6.1
*/
}

proc subtract(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.6.2
*/
}

proc shiftLeft(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.7.1
*/
}

proc shiftRight(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.7.2
*/
}
```

```
proc shiftRightUnsigned(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.7.3
*/
}

proc lessThan(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.8.1
*/
}

proc lessThanOrEquals(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.8.3
*/
}

proc hasProperty(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.8.7
*/
}

proc instanceof(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.8.6
*/
}

proc asType( obj:*, type:Type ) : type
{
/*
    If obj implicitly converts to type,
    Return obj implicitly converted to type
    Else If type includes null
        Return null
    Else
        Return obj explicitly converted to type
*/
}

proc isType( obj:*, type:Type ) : Boolean
{
/*
    If obj is in the value set of type
        Return true
    Return false
*/
}

proc equals(v1:*,v2:*)
```

```
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.9.1
*/
}

proc strictEquals(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.9.4
*/
}

proc bitwiseAnd(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.10
*/
}

proc bitwiseXor(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.10
*/
}

proc bitwiseOr(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.10
*/
}

proc logicalAnd(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.11
*/
}

proc logicalXor(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.11
*/
}

proc logicalOr(v1:*,v2:*)
{
/*
    This procedure implements the semantics described in
    Ecma-262, section 11.11
*/
}
```

# 5 Names

A name consists of a string and a namespace. Names are introduced into a particular scope by a definition. Those definitions are referred to by names that result from expressions.

The qualified forms result in a single name consisting of the given qualifier and identifier. The unqualified forms result in a set of names consisting of strings qualified by the open namespaces.

The visibility of an identifier is controlled by the set of open namespaces. The set of open namespaces includes all of the implicitly opened namespaces and the user opened namespaces. The implicitly opened namespaces are:

- Public namespace
- Internal namespace for the current package
- Private namespace for the current class
- Protected namespaces for the current class
- The user opened namespaces are controlled by the use namespace directives that are in scope. For example,

```
namespace mx = "http://macromedia.com/mx"
use namespace(mx)
o.m()
```

In this example, the reference to `o.m()` will involve the names qualified by the namespace `mx` as well as the implicitly opened namespaces: public, internal, etc.

The terms *namespace* and *qualifier* are used interchangeably when talking about qualified names.

## 5.1 Definition names

A name introduced by a definition might get its qualifier from one of various sources

- top-level definitions in a package have the package name as their qualifier
- top-level definitions outside of a package have the public namespace as their qualifier
- interface members have the interface name as their qualifier
- dynamic property names have the public namespace as their qualifier
- definitions inside a class have the internal namespace of the current package as their qualifier, unless a namespace attribute is specified
- a definition with a namespace attribute has its corresponding namespace as its qualifier
- a definition with an access control attribute has the implicitly defined namespace for that access specifier as its qualifier

It is an error to introduce a name with an identifier that has already been defined in an open namespace in the same scope, but with a different qualifier.

## 5.2 Reference names

Reference names result from various forms of expressions. The two main distinctions in these forms are whether the name is qualified or unqualified, and whether the identifier is a literal identifier or an expression.

The following table shows the kinds of references that include qualified and unqualified, literal and expression names.

| | Literal | Expression |
|---|---|---|
| Unqualified | `o.id, id` | `o[expr]` |
| Qualified | `o.q::id, q::id` | `o.q::[expr], q::[expr]` |

- A qualified or unqualified literal identifier is equivalent to the dynamic form with its expression operand replaced by a string literal representing the literal identifier
- An unqualified expression reference results in multiple names (called a multiname), one for every open namespace combined with the string value of the expression `expr`
- A qualified expression reference results in a qualified name that consists of the value of the qualifier `q` combined with the string value of the expression `expr`

*[edit: show examples of multinames]*

### 5.3 Name lookup

An expression involving a name results in an internal reference value used by certain operators to perform actions. To describe name lookup we distinguish between two types of references: those that include a base object (object references), and those that do not (lexical references.)

Looking up a reference involves determining its ultimate qualified name (in the case of unqualified references) and its base object.

#### 5.3.1 Object references

Object references result from expressions involving the dot or bracket operators. They may be qualified or unqualified. The following table shows various forms of object references.

| | Literal | Expression |
|---|---|---|
| Unqualified | `o.id` | `o[expr]` |
| Qualified | `o.q::id` | `o.q::[expr]` |

We use the expression form of references to describe the name lookup semantics. However, every literal name can be rewritten as an expression name by the following steps.

- If the expression is an unqualified literal name, then replace the dot operation `o.id` with a bracket operations of the form `o['id']`
- Otherwise the expression is a qualified literal name, so replace the operand of the dot operation with the dot operation `o.q::['id']`

##### 5.3.1.1 Unqualified object references

```
o[expr]
```

This is a reference to a property of the value of the expression o that has a name that matches one of the names of the set of names (multiname) composed in the following way:

- Let id be the string value of the expression `expr`
- Let `m` be an empty set of names
- For each namespace `q` in the set of open namespaces
    - Let `n` be a name with the qualifier `q` and the identifier `id`
    - Add `n` to the set of names `m`
- Return `m`

The single name of a multiname reference `r` is determined by the following steps:

- Let `t` be the least derived type of `x` that contains at least one of the names in the multiname set `m` of the reference `r`
- Let `m'` be the intersection of the set of names `m` and the property names in `t`
- Let `n` be the set of names in the most derived type of `x` and in `m'`
- If `n` is empty, return the name in `m` that is qualified by the public namespace
- If `n` contains one name, then return that name
- Report an ambiguous reference error

The base object of this reference is the value of the expression `o`.

#### 5.3.1.2 Qualified object references

```
o.q::[expr]
```

This is a reference to a property inside the value of o that matches a single name. Because the qualifier is explicit, the qualified name is straightforward to compute.

- Let `ns` be the value of the expression `q`
- Let `id` be the string value of the expression `expr`
- Return the qualified name consisting of the namespace `ns` and the identifier `id`

The base object of this reference is the value of the expression `o`.

### 5.3.2 Lexical references

```
q::[expr]
q::id
id
```

Lexical references result from expressions involving a name but no base object. Whether a lexical reference is qualified or unqualified, with a literal identifier or expression, it results in a search of the scope chain of the lexical environment until either a match is found or the last scope is searched.

The scope chain might include the following kinds of scopes:

- Code inside a `with` statement will have a `with` frame as the inner most scope on the scope chain
- Code inside a function definition will have an activation object on its scope chain

- Code inside an instance method will have the instance `this` object on its scope chain
- Code inside of a class definition, including in instance and static methods, will have the class objects of its base classes and the current class on the scope chain. The inner most class object corresponds to the most derived class, and the outermost class object corresponds to the Object class
- Code everywhere has the global object as the outer most object on its scope chain

The base object of a lexical reference is computed by the following steps:

- Let s be the list of scopes enclosing the reference being evaluated
- Let n be the qualified name or set of qualified names that result from the operation described in section 4.3.1.1
- Search the scopes in s starting from the innermost scope and continuing outwards until a scope is found that contains a property that matches n, or all scopes have been searched
- If a match is found, return the scope that contains the matching property
- Report a property not found error

# 6 Types

A type is a set of values. In the standard dialect expressions have values and properties have types. To be used as a value of a property, that value must be a member of the property's type, or be implicitly convertible to the property's type. For example,

```
var x : Number = 10
```

The value 10 is a member of the set of values that are implicitly convertible to a value in Number's value set, so the assignment is allowed.

In the strict dialect both expressions and properties have types. To be used to compute the value of a property, the expression must have a type that is compatible with the type of the property. One way to think about static types of expressions and values is that the static type is a conservative approximation of the set of values that will result from that expression.

The value set of a type is normally determined by the inheritance relationship between the type and its descendants. So for example,

```
class A {}
class B extends A {}
class C extenda B implements D, E {}
```

Values that are instances of class C are members of the types A, B, C, D and E. Values that are instances of class B are members of the types A and B. Values that are instances of the class A are members of the type A. All class instances are implicitly members of the type Object.

## 6.1 Run time versus compile time type

We sometimes refer to a class or interface that helps to define the structure of a value as the value's type. What we really mean is that that value is a member of that class or interface type. This distinction is subtle but important. Since a value might belong to any number of unrelated types to say that it is of a particular type is misleading.

In dynamically typed languages expressions don't have types; they have values whose types may change each time the expression is evaluated.

Statically typed languages make the important simplification of requiring expressions to have a known type, even if it is a very general one, when it is compiled. In this way the suitability of an expression can be checked against its use before it is ever actually run. The cost of this added reliability is the loss of flexibility that comes from not having to think about the types of values.

```
function f( o : Object ) {
    var x : Number
    x = o           // Allowed in the standard dialect
}
f(10)               // No problem, x gets set to 10
```

Other places where the differences between dynamic and static type checking can be seen are property access, and method invocation.

```
function f( o : Object ) {
    o.g()
    return o.x
```

```
}
```

Whereas in a static type system, the binding for a method call or property read, would need to be known at compile-time, the standard dialect always defers that checking until runtime.

The strict dialect has a hybrid type system. Normally static type rules are used to check the compatibility of an expression with its destination type but there are a few special cases. For example, when an expression on the right side of an assignment expression consists of an reference to an property with no type, name lookup is deferred to run time. When an object reference has a base object that is an instance of a dynamic class, the reference is checked at runtime. These dynamic typing features are useful when strict dialect programs are interoperating with dynamic features such as XML objects.

### 6.2 Untyped versus typed properties

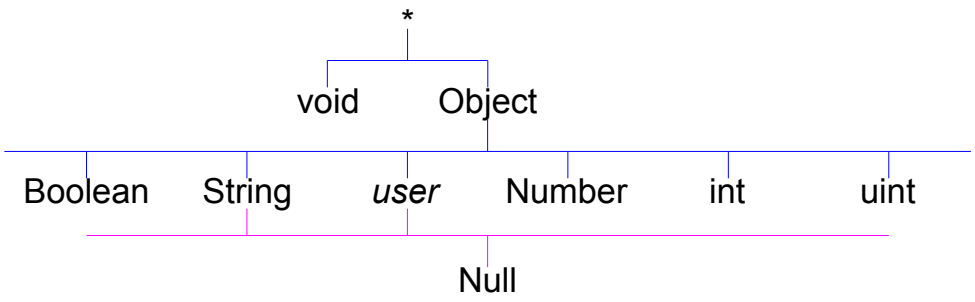
A property without a type annotation or with the annotation `*` (as in, `var x : *`) is said to be untyped. Writing to an untyped property will always succeed since an untyped property can hold any value. Expressions that read from an untyped property are said to be untyped expressions. Assignment from an untyped expression may or may not succeed at runtime depending on whether its value can be implicitly converted to the destination type. Nevertheless, in the strict dialect assignments from untyped expressions are always type checked at runtime as in the standard dialect.

Use untyped properties when you want to store the result of an untyped expression or `undefined` as one of the values, or when you want to defer type checking to runtime.

### 6.3 Fundamental types

There are three fundamental program types (void, Null, and Object). What makes these types fundamental is that they their union includes all possible values in the language. Null includes null, void includes undefined, and Object includes every other value. Null and void are different because they do not have object like properties (e.g. toString, valueOf), and they both have values that represents “no value”.

The relationships between these fundamental types is important when we talk about their value sets and the meaning of type annotations (as in, var x : T = v). If the type of v is a subtype of T (including T), then the assignment of v to x is considered type safe. In the strict dialect, the type of v is the static type of the expression v; in the standard dialect the type of v is the run time (true) type of the value the results from the expression v.




The type Null includes one value. It is the value that results of the primary expression `null`. The value `null` is used to represent the idea “no value” in the context of a nullable typed reference. Not all types are nullable (include null). Those that do not implicitly convert the value null to the default value of that type.

The type void includes one value. It is the value that is the initial value of the global property `undefined` and the result of the unary expression `void 0`. The value undefined is used to represent the idea "no property" or "no value" in the context of an untyped reference.

While the need for two types that represent the idea of "no value" seems strange to programmers familiar with statically typed object oriented languages, in this language the distinction is useful for representing the absence of a property or the absence of a value of an untyped property versus the absence of a typed property. Here is an example,

```
dynamic class A {
var x : String
var y
}
var a : A = new A
print(a.x)    // null
print(a.y)    // undefined
print(a.z)    // undefined
a.y = 10
a.z = 20
print(a.y)    // 10
print(a.z)    // 20
```

When dealing with dynamic instances, there is little difference between a property that doesn't exist and a property with no type and no value. But there is a difference between a property that has a type and one that doesn't. This is one of the reasons for the existence of both types Null and void.

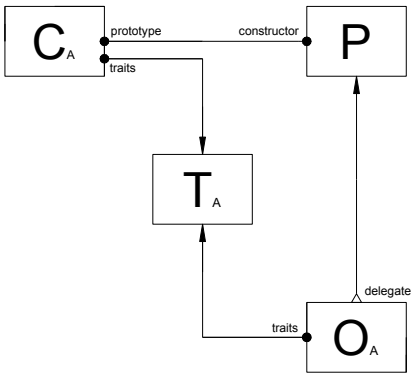
*Note: In Ecmascript ed. 3 program visible values where instances of one of six unrelated types (Undefined, Null, Boolean, Number, String and Object). Conversions were provided to translate a value from one type to another. Ed. 4 provides the same conversions between the primitive types (void/Undefined, Null, Boolean, String, Number, int and uint)*

### 6.3.1 Object types

All program visible types other than void and Null derive from type Object. This means that all values (except undefined and null) have properties that can be accessed by object references without the need to be wrapped in an object as they were in Ecmascript ed. 3.

## 6.4 Class types

A class refers to a type or a value depending on its use.




```
class A
{
    static var x
    var y
    prototype var z
}
var a : A           // A means type A
a = new A           // A means value A
```

The value is a class object has the form shown in the

drawing above. The class object is $C_A$. When used as a type it evaluates to its instance traits ($T_A$). When used in a new expression it serves as a factory object with a special method that create a new instance ($O_A$), with its internal delegate property pointing to the class object's prototype (P), and its internal traits property pointing to the class object's instance traits ($T_A$).

## 6.5 Interface types

An interface name can only be used where a type is expected.

```
interface I{}
var x : I           // I means type I
x = new I           // Error, I is not a value
```

## 6.6 Type conversions

A type conversion is the translation of a value to a value that is a member of a specific destination type. When the original value is a member of the destination type the value is unchanged. We call this an identity conversion.

Type conversions occur at runtime in various contexts:

- assignment expressions, argument lists and return statements
- `as` operator
- other operators

The result of the conversion depends on the context of the expression that yields the value to be converted:

| | |
|---|---|
| `var x : T = v` | Implicit conversion to T |
| `var y : T = v as T` | v or null |
| `var z : T = v + 10` | Conversion according to the rules of the operator |

### 6.6.1 Static types and conversion

Normally the static type checking of the strict dialect will guarantee that runtime exceptions will not occur during a conversion. This is enforced by the general rule:

Every value of the static type of an expression must be a value of the destination type of its use as determined by the `is` operator. For example, the following result in verification errors in the strict dialect:

```
var n : Number = "20"   // error, cannot coerce String to Number
class B extends A {}
var b : B = new A       // error, cannot coerce A to B
var n : Number = 30
var s : String = n      // error, cannot coerce Number to String
```

In all three cases, the static type check can be overridden with an explicit cast to the destination type.

```
var n : Number = Number("20")
class A {}
```

```
class B extends A {}
var a : A
var b : B = B(a)
var n : Number = 30
var s : String = String(n)
```

There are three special cases where static types are ignored, possibly allowing runtime errors to occur:

- coercions from an untyped expression to any type
- coercions from any type to type Boolean
- coercions between different numeric types

An explicit cast to a user defined type is only useful in the strict dialect. This is because the effect of an explicit cast is to defer type checking until runtime, which is already the case in the standard dialect. This is not necessarily the case for built-in types that have special conversion behavior.

### 6.6.2 Implicit conversions

Implicit conversions occur when a value is assigned to a property, passed as an argument to a function, or returned from a function.

When the destination type is a user defined type T, an implicit conversion will succeed if the value is an instance of a class that is T or is derived from T. If an implicit conversion does not succeed, then a type error is thrown.

When the destination type is a primitive type, the implicit conversion is described by the corresponding abstract procedure (e.g. toString() and toNumber().) The following table shows some implicit conversion results:

| Value | String | Number | int | uint | Boolean | Object |
|---|---|---|---|---|---|---|
| {} | “[object Object]” | NaN | 0 | 0 | true | {} |
| "string" | "string" | NaN | 0 | 0 | true | "string" |
| "10" | "10" | 10 | 10 | 10 | true | "10" |
| null | null | 0 | 0 | 0 | false | null |
| undefined | null | NaN | 0 | 0 | false | null |
| true | "true" | 1 | 1 | 1 | true | true |
| false | "false" | 0 | 0 | 0 | false | false |
| 0 | ”0” | 0 | 0 | 0 | false | 0 |
| 1 | ”1” | 1 | 1 | 1 | true | 1 |
| -1 | ”-1” | -1 | -1 | 2E+32-1 | true | -1 |
| 1.23 | "1.23" | 1.23 | 1 | 1 | true | 1.23 |
| -1.23 | "-1.23" | -1.23 | -1 | 2E+32-1 | true | -1.23 |
| NaN | "NaN" | NaN | 0 | 0 | false | NaN |

User defined types do not have built-in conversion operators, so implicit and explicit conversions behave the same at runtime. Specifically, if a value is not a member of the destination type, then no conversion exists, implicit or explicit, and a runtime exception will result in a cast expression and the default value of the destination type (which is null) will be the result of an as expression.

## 6.7 Type operators

### 6.7.1 Operator is

The `is` operator appears in expressions of the form:

```
v is T
```

The `is` operator checks to see if the value on the left hand side is a member of the type on the right hand side. For user defined types and most built-in types, `is` returns `true` if the value is an instance of a class that is or derives from the type on the right hand side, otherwise it returns `false`. For built-in numeric types the result cannot be determined by the class of the value. The implementation must check the actual value to see if it is included in the value set of the type.

The following table shows the results of using various values and types with the `is` operator:

| Value | String | Number | int | uint | Boolean | Object |
|---|---|---|---|---|---|---|
| {} | false | false | false | false | false | true |
| "string" | true | false | false | false | false | true |
| "10" | true | false | false | false | false | true |
| null | false | false | false | false | false | false |
| undefined | false | false | false | false | false | false |
| true | false | false | false | false | true | true |
| false | false | false | false | false | true | true |
| 0 | false | true | true | true | false | true |
| 1 | false | true | true | true | false | true |
| -1 | false | true | true | false | false | true |
| 1.23 | false | true | false | false | false | true |
| -1.23 | false | true | false | false | false | true |
| NaN | false | true | false | false | false | true |

### 6.7.2 Operator as

The `as` operator appears in expressions of the form:

```
v as T
```

The purpose of the `as` operator is to guarantee that a value is of certain type, and if not indicate so by returning the value `null`.

Note: It is common usage to assign the result of an as expression to a property with the same type in that expression. If the destination type does not include null, the assignment will convert null to the default value of that destination type (e.g. false for as Boolean and 0 for as Number). This results in loss of information about whether the original value is included in that type. Programs that need to distinguish between when a value is the default value and an incompatible value, must assign the result to a property of type Object, check for null, and then downcast to ultimate destination type.

The steps used to evaluate the as operator are:

- Let $v$ be the value of the left operand

- Let *T* be the value of the right operand
- If *T* is not of type Type
  - Throw a TypeError
- If *v* is of type *T*
  - Return the value v
- Else
  - Return the value **null**

# 7 Variables

A variable defines a slot with a name and a type.

A variable declared with the `const` rather than `var` keyword, is read-only outside of the variable's intialiser if it is not an instance variable, and outside of the instance constructor if it is an instance variable. It is a verifier error to assign to a const variable outside of its writable region.

Variables exist in the following objects:

- global object, inside an outside of a package
- class objects
- instance objects
- activation objects

## 7.1 Variable modifiers

When allowed by the context of the definition, the following attributes modify a variable definition.

- Access control namespaces
- User defined namespaces
- `static`
- `prototype`

Access control and visibility control namespaces specify the namespace part of the variables name.

The `static` attribute may only be used inside a class definition and causes the variable to become a trait of the class object rather than the instance object.

The `prototype` attribute may only be used inside a class definition and causes the variable to be added to the class's prototype object and a get and set accessor to be added to the instance traits of the class. The purpose of the accessor methods is to simulate the behavior of accessing prototype properties in ES3.

## 7.2 Variable types

All variables have a type. A type annotation on a variable definition limits the set of values that can be stored in that variable. If no type is specified, the implied type is Object. A type annotation must be compile-time constant expression that evaluates to a class or interface value. The actual value used to represent the type of the variable is the instance traits of the referenced class or interface.

When a value is assigned to a variable an implicit conversion to the variables type is performed on the value. A type error occurs if there is no implicit conversion of that value to the variables type. In the strict dialect such errors are verification errors, in the standard dialect type errors are runtime errors.

# 8 Functions

A function is a callable object. In general functions consist of a block of code, a set of traits, and a list of scopes. Instance methods are functions that also consist of a receiver object that `this` references are bound to.

## 8.1 Function modifiers

When allowed by the context of the definition, the following attributes modify a variable definition.

- Access control namespaces
- User defined namespaces
- `static`
- `final`
- `override`
- `native`
- `prototype`

Access control and visibility control namespaces specify the namespace part of the function name.

The `static` attribute may only be used inside a class definition and causes the function to become a trait of the class object rather than the instance object.

The `final` attribute may only be used on a non-`static` function definition inside a class. A function modified by `final` cannot be overridden.

The `override` attribute may only be used on a non-`static` function definition inside a class. A function modified by `override` will override a method with the same name and signature as a non-final method of a base class.

The `native` attribute may be used to indicate that that the function is implemented in an implementation defined way. The compiler should generate native stubs for functions that have this attribute.

The `prototype` attribute may only be used inside a class definition and causes a variable initialized to a function object to be added to the class's prototype object, and a get and set accessor to be added to the instance traits of the class. The purpose of the accessor methods is to simulate the behavior of accessing prototype functions in ES3.

## 8.2 Function signatures

A function signature includes the number and types of its parameters and its result type. Like variable type annotations, the types of a function signature affect the implicit conversion of argument and return values when calling to and returning from a function. Function signatures are also used to match inherited methods to methods in a derived class.

## 8.3 Function objects

Global and nested functions can be used a constructors in instantiation expressions. For example,

```
function A() { this.x = 10 }
var o = new A
print(o.x)     // prints 10
```

Function objects have a property named prototype whose value is used to initialize the intrinsic delegate property of the objects is creates. The prototype property has a default value of a new instance of the class Object. Building on the example above,

```
function A() { this.x = 10 }
function B() {}
B.prototype = new A
var o = new B
print(o.x)     // prints 10
```

The value of `o` is an instance of `B` which delegates to an instance of `A` which has a property named `x` with value of `10`.

Constructor methods inside of a class are also used to create objects. But unlike constructor functions, constructor methods create objects with a set of fixed properties (traits) associated with its class, and a delegate that is also an instance of its class.

```
class A {
    var x
    function A() { this.x = 10 }
}
var o = new A
print(o.x)     // prints 10
```

There are some subtle differences between this example and the one involving a function constructor above:

- x is a fixed property of each instance of A rather than a dynamic property
- A.prototype is an instance of A rather than an instance of Object
- The expression A(expr) does not call the function A defined in class A. It results in an explicit conversion of the value of expr to the type A

Class methods are functions that are defined with the attribute static inside of a class definition. A class method cannot be used as a constructor and does not define this. Class methods are in the scope of the class object they are defined in.

Instance methods are functions that are defined without the static attribute and inside a class definition. Instance methods are associated with an instance of the class they are defined in. Instance methods can override or implement inherited class or interface methods, and always have a value bound to this.

The value of `this` in an instance method is the value of the instance the method is belongs to. When an instance method is extracted from an object, a method closure is created to bind the value of `this` to that host object. Assignment of the method closure to property of another object does not affect the binding of `this`. For example,

```
class A {
    var x
    function A() { this.x = 10 }
    function m() { print(this.x) }
}
```

```
var a = new A()
var o = { x : 20 }
o.m = a.m
o.m()   // prints 10
```

# 9 Classes

A class is a type, a constructor of objects of that type, and a singleton object for sharing state and behavior. It is used as a constructor to create like instances. It is used as a type to constrain the value of properties. It is used as a singleton object to contain shared properties.

Classes are introduced with class definitions. A class definition can directly extend one other class definition and implement multiple interface definitions. The language does not support the concept of abstract classes and so a class must implement every interface method it inherits.

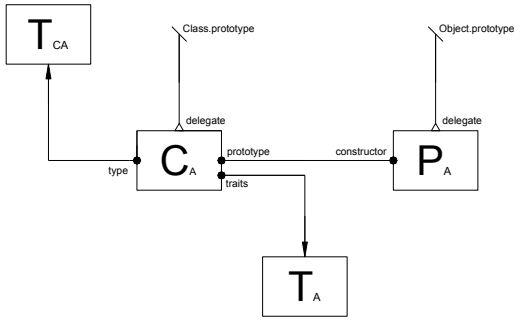
## 9.1 Class modifiers

Class definitions may be modified by these attributes

| | |
|---|---|
| dynamic | Allow properties to be added to instances at runtime |
| final | Must not be extended by another class |
| internal | Visible to references inside the current package (default) |
| public | Visible to references everywhere |

The default modifiers for a class definition are internal, non-dynamic, and non-final.

## 9.2 Class objects




Class objects have the basic structure shown in this drawing. The drawing illustrates the shape of the class object that results from this simple class definition,

```
class A {}
```

### 9.2.1 Prototypes

Every object has a prototype object that it is used to match references at runtime. This prototype is called the delegate of the object. Delegation is a simple way to add shared properties to a group of related objects at runtime.

Prototype objects are always instances of the dynamic class Object and therefore can always be extended by the addition of dynamic properties. Unlike with function closures which have a prototype property that is a variable and can be reset to another object, classes have a prototype that is read-only and so always points to the same object.

### 9.2.2 Traits

Properties of a class definition are represented as traits of the class object and its instances. Think of a trait as a fixed property that is shared by all instances of a type. Class objects ($C_A$) are special in that they are a single instance with an internal type with a corresponding set of traits ($T_{CA}$). The internal type of a class object describes the static properties of the class definition. The instance traits ($T_A$) are shared by all instances created by the class object. They correspond to the instance properties of the class definition.

```
class A
{
    static var x
    var y
}
```

In this example, the definition for `x` contributes a trait to the class traits ($T_{CA}$), and the definition of `y` contributes a trait to the instance traits ($T_A$).

### 9.2.3 Methods

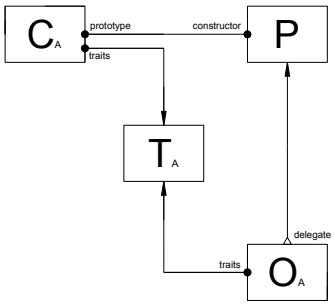
Each function definition inside of a class definition results in method inside the resulting class object or its instances. There are two special methods that are implicitly defined for each class: a class initializer; and an instance initializer. Code outside of a function definition gets placed in the class initializer, which is called when the class object is created. Instance variable initializers are placed in the instance initializer method, which is called when an instance of the class is created and before the user defined constructor is executed.

### 9.2.4 Slots

Traits introduced by variable definitions describe a property that holds a value unique to each instance. Therefore, each object has a fixed array of slots that store those values, one for each variable trait. This is true of class objects as well as instance objects.

### 9.2.5 Instances

All instances ($O_A$) created by a class object ($C_A$) will be given a traits ($T_A$) and delegate ($P_A$) object, as represented in this drawing




[edit: fix picture]

### 9.2.6 Inheritance

Each class inherits the instance traits of its base class. These traits are effectively copied down to the instance traits of the derived class. Classes that don't declare an explicit base class inherit the built-in Object class.

A class may also inherit the instance traits of one or more interfaces. Interface traits are abstract and so must be implemented by any class that inherits them.

Unlike in some other object oriented languages (e.g. Java), static properties of the base class are not inherited, but they are in scope in the static and instance methods of the derived class.

### 9.2.7 Scopes

Static properties are in scope of bodies of static and instance methods of the same class. Instance properties are in scope of the bodies of the instance methods. Instance properties shadow static properties with the same name. Static properties of base classes are in scope of static and instance methods of a class.

```
class A
{
    static var ax
}
class B extends A
{
    static var bx
}
class C extends B
{
    static var cx
    var ix
    function m()
    {
        var mx
        gx = 10
        ax = 20
        bx = 30
        cx = 40
        mx = 50
    }
}
var gx
o = new C
o.m()
```

Scopes:

```
{ mx } - activation scope
{ ix } - instance scope
{ cx } - static scope C
{ bx } - static scope B
{ ax } - static scope A
{ gx } - global scope
```

## 9.3 Class property attributes

Class properties may be modified by the following attributes

| | |
|---|---|
| static | Defines a property of the class object |

| | |
|---|---|
| prototype | Defines a property of the class prototype object |
| private | Visible to references inside the current class |
| internal (default) | Visible to references inside the current package |
| protected | Visible to references inside instances of the current class and derived classes |
| public | Visible to references everywhere |
| *AttributeExpression* | Namespace value is the qualifier for the name of the definition |

It is a syntax error to use any other attribute on a class property, unless otherwise specified in the section describing the specific type of property.

### 9.3.1 Static attribute

The `static` attribute means the current definition defines a property of the class object.

### 9.3.2 Prototype attribute

The prototype attribute enables the addition of a fixed property to the prototype object. It can be used on instance method and variable definitions. Prototype properties are accessible through the prototype object or through instances that delegate to that prototype. As in ES3, assigning to a prototype property of an instance will set a property of the same name in the instance. Prototype properties may be declared with a namespace like any other class property. The qualified name that results is the name of the instance accessor that forwards access to the prototype object.

A prototype function has on its scope chain the class object of its class, but no instance. Think of prorotype functions as function closures on the class prototype.

```
class A {
    static var x = 10
    A.prototype.f = function f() { print(x) }
}
```

[Implementation note:

The classes,

```
class A {
    prototype function f() { print("A.f") }
}
class B extends A {
    prototype function f() { print("B.f") }
}
```

are compiled as,

```
class A {
    public static var prototype : A
    internal var _f : Function
    internal var _f_isset : Boolean
    function get f() : Function {
        if( _f_isset ) {
            return _f
        }
        return A.intrinsic::delegate._f
```

```
                // intrinsic is a private namespace
                // used by the implementation to name
                // internal properties
    }
    function set f(_f : Function) : void {
        this._f = _f
        _f_isset = true
    }
    A.prototype = new A
    A.prototype.f = function () { print("A.f") }
}

class B extends A {
    public static var prototype : B
    B.prototype.f = function () { print("B.f") }
}
```

Prototype variables are implemented similarly. The main difference is that the internal variable that holds the prototype variable's state is given the declared type of the variable.

```
class A
{
    prototype var x : int = 10
}

class A {
    public static var prototype : A
    internal var _x : int
    internal var _x_isset : Boolean
    function get x() : int {
        if( _f_isset ) {
            return _x
        }
        return A.intrinsic::delegate._x
    }
    function set x(_x : Function) : void {
        this._x = _x
        _x_isset = true
    }
    A.prototype = new A
    A.prototype.x = 10
}
```

The following example illustrates the behavior of the prototype variable x.

```
var a1 = new A
var a2 = new A
print(a1.x)   // prints 10, shared initial value
print(a2.x)   // prints 10, shared initial value
A.prototype.x = 20
print(a1.x)   // prints 20, shared assigned value
print(a2.x)   // prints 20, shared assigned value
a2.x = 30
print(a1.x)   // prints 20, shared assigned value
print(a2.x)   // prints 30, individual assigned value
```

end implementation note]

### 9.3.3 Access control namespace attributes

Each access control attribute (private, internal, protected, and public) refers to namespace value with a unique, private namespace name. Access control is provided by the fact that code outside of the attributes access domain has no way to refer to that namespace value.

### 9.3.4 User defined namespace attributes

The value of an attribute expression that evaluates to a compile-time constant namespace is used as the qualifier of the definitions name.

```
namespace ns
class A
{
    ns var x
}
```

In this example the name of the definition of `x` is qualified by the namespace `ns`

- only one namespace attribute may be used per definition
- namespace attributes may not be used with an access control attribute

## 9.4 Class body

A class body may contain variable definitions, namespace definitions, function definitions, and statements.

```
class A
{
    static var x
    static function f() {}
    var y
    function g() {}
    print("class loaded")
}
```

- Definitions result in class or instance traits depending on whether the `static` attribute occurs in their definition
- Statements and initializers of static variables are added to the static initializer method of the class. The static initializer is called once, with the class is defined at runtime. The static initializer can be used to initialize variables of the class object and to invoke methods that are external to the current class
- Initializers of instance variables are added to the instance initializer method
- The scope chain of methods contained by the class body includes the class object, the base class objects (from most derived the least derived), and the global object.

Note: it is not an error to define a class and instance property with the same name. e.g.

```
class A {
    static var x
    var x
}
```

It is not an error to define a class property with the same name as a visible class property in a base class. e.g.

```
class A {
    static var x
}
class B extends A {
    static var x
}
```

]

### 9.5 Class variables

Class variables are defined using the `var` or `const` keywords.

```
class A
{
    var x
    const k = 10
}
```

The meaning of `var`, `const` follow from the general meaning described elsewhere in this specification.

| | |
|---|---|
| var | May be written to multiple times |
| const | May be written to only once |

`const` variable properties can be written to only once. The compiler uses a specific data flow analysis to determine if a const variable has been written to at the point of an assignment to that variable. A precise definition of this data flow algorithm can be found in section 2.3.1. Informally the effect of this algorithm can be seen in the following error cases

- It is an error to assign to a const instance or static variable in a statement that is outside of the instance or static initializer, respectively
- It is an error to assign to a const variable more than once in a sequence of statements with no control flow branches
- It is an error to assign to a const variable in more than one parallel control flow branch if the branch conditions are not compile-time constant expressions, or if that value of those branch conditions allow for one or more of those branches to be executed more than once.

The default value of a class or instance variable is the value of undefined coerced to the type of the variable.

#### 9.5.1 Static variables

Variables declared with the `static` attribute add a slot trait to the class traits and a slot to the class object. Because there is only one class object per class, there is also only one slot per static variable. Static variables, like static methods, are not inherited, but are accessible from within the body of the class definition and through an explicit reference to the defining class's name. Static variables are in scope for all static and instance methods of the defining class and classes that inherit the defining class.

Static const variables must either have an initializer or be definitely unassigned before being set in the static initializer method.

Note: unlike in Java and C#, static variables are not inherited by derived classes and so can not be referenced through derived class objects.

### 9.5.2 Instance variables

Variables declared without the `static` attribute add a slot trait to the instance traits of the class and a slot to each instance of the class. Instance variables are always `final` and must not be overridden or hidden by a derived class.

As with all class properties, the default qualifier for the variable is the `internal` namespace. Other qualifiers can be specified by other namespace attributes. Both instance and class variables are implicitly final. Any attempt to hide or override one in a derived class will result in a verification error.

## 9.6 Class methods

A method is function associated with a specific object. Unlike a function closure, a method is not a value and cannot be used apart from the instance it is bound to. The value of `this` inside a method is always the base object used to refer to the method, and always has the type of the class that implements the method, or subclasses.

### 9.6.1 Constructor methods

A function declared with the same identifier as the class it is defined in adds a constructor method to the class object. The constructor is called when a new instance of that class is created. A constructor may refer to the instance variables of the class that defines it.

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

A constructor is public by default and may be defined with the public namespace or with no namespace attribute. If no constructor is defined by a class definition, a default constructor is defined implicitly. No more than one constructor can be defined for a class.

*[Rationale: making a constructor always public reinforces the user model of classes, like functions, as constructors. While other languages allow constructors methods to be made inaccessible to keep outside code from creating instances, this use case was not deemed important enough to complicate the language design. It is a compatible change to allow explicit access control namespace attributes in a later edition.]*

If the body of a constructor contains a *SuperStatement*, that statement must occur before the first reference to `this` or `super`, and before any return or throw statement. If a call to the super constructor is not explicit, then one will be inserted before the first statement in the constructor body.

- It is a syntax error to call the super constructor more than once as determined by a specific data flow algorithm specified in section 2.3.2.
- It is a syntax error to specify a return statement with an expression
- It is a syntax error to specify a result type of a constructor

*Note: that there is no way to directly call the constructor of an indirect base class is intentional. This might lead to brittle or insecure programs.*

### 9.6.2 Static methods

Functions declared with the `static` attribute add a method trait to the class object traits. Static variables are in scope of a static method.

It is an error to for the `this` or `super` expression to appear in the body of a static method.

Unlike in Java and C#, static methods are not inherited by derived classes and so can not be referenced through derived class objects.

### 9.6.3 Instance methods

Functions declared without the `static` attribute add a method trait to the instance traits of a class object. Static and instance variables are in scope of an instance method. The value of `this` inside an instance method is the instance the method is bound to.

```
class A
{
    function m() { return this }
}
var a = new A
print(a==a.m())   // print true, this is the object 'm' is called on
```

In addition to the attributes defined for all class properties, the following attributes may be used on instance methods

| | |
|---|---|
| final | May not be overridden |
| override | Must override an inherited method |
| prototype | Defines a property of the prototype object |

The attribute `override` helps to avoid unintentional overriding of base class methods. It is a verifier error to use the `override` attribute on a function definition that does not override an inherited method. It is a verifier error to override an inherited method that is declared final. It is an error to define a method with a name that matches the name of an inherited method and without the override attribute.

The prototype attribute allows the addition of a fixed property to the prototype object, but not to the instance. Instance methods defined with the prototype attribute have function values that are compatible with Ecmascript edition 3 prototype functions.

```
class A
{
    prototype function f() { return this }
}
var a = new A
dynamic class B {}
var b = new B
b.f = a.f
b.f()   // prints "[object B]"
```

The instance of B becomes the value of `this`.

### 9.6.4 Accessor methods

A method defined with the `get` or `set` keyword, adds a get or set method trait to the instance or static traits of the defining class object. Accessor methods are called when the name of the accessor is used in a reference that reads or writes the value of that name.

```
class A
{
    private var _x
    function get x() { return _x }
    function set x(v) { _x = v }
}
var a  = new A
a.x = 10       // calls set accessor of A
print(a.x)     // prints 10, calls get accessor of A
```

Accessor methods are very similar in definition to regular methods. The differences are expressed by the following error conditions,

- Get methods must specify no parameters
- Set methods must specify just one parameter
- Get methods must return a value
- Set methods have a result type void by default
- Set methods must not specify a result type other than void
- Get methods must not specify the result type void
- If both a get and set method is defined with the same name, the parameter type of the set method and the result type of the get method must match

Note: accessor may only be defined at the top level of a class. They must not be nested inside another method, or defined outside of a class.

### 9.6.5 Inheriting instance methods

Instance methods are inherited by copying their instance traits down to the instance traits of the derived class.

#### 9.6.5.1 Overriding instance methods

Methods inherited from a class may be overridden in the derived class if the overriding method is given the `override` attribute and if its name, number and type of parameters, and return type match exactly. It is an error to attempt to override a method with a method that has the same name, but does not have the same number of parameters or parameters of different types or different return type.

#### 9.6.5.2 Implementing interface methods

Methods inherited from an interface must be implemented by a method with a name and signature that matches the inherited method. Interface methods are implemented by an instance method declared with the `public` attribute.

A method that has the `public` attribute implements all inherited interface methods with a matching identifier.

```
interface I
{
    function m()
}
interface J
{
    function m()
}

class A implements I,J
{
    public function m() { print("A.m") }
}
```

In this example, the definition of m in class A satisfies both interfaces I and J.

### 9.6.6 Method closures

Although a method is not a value by itself, it can be converted to a first class value called a method closure, through extraction. A method closure maintains the binding between a method and its instance. The user visible type of a method closure is Function.

```
class A
{
    function m() { return this }
}
var a  = new A
var mc : Function = a.m     // create a method closure from m and a
print(a==mc())   // print true, mc remembers its this
```

# 10 Interfaces

Interfaces provide a way for programs to express contracts between the producers and consumers of objects. These contracts are type safe, easy to understand and efficient to implement. Programs should not have to pay a significant performance penalty for using interfaces.

An interface is a type whose methods must be defined by every class that claims to implement it. Multiple interfaces can be inherited by another interface through the `extends` clause, or by a class through the `implements` clause. Instances of a class that implements an interface belong to the type represented by the interface. Interface definitions must only contain function definitions, which may include get and set methods.

Interface methods are not public by default, but are added to the public namespace by the implementing method definition.

## 10.1 Interface types

An interface definition introduces a type into the current scope. The interface type is described by a set of abstract method traits and a list of interfaces that it extends. This set of abstract traits must be fully implemented by any class that inherits the interface.

An interface name refers to the interface type when it is used in a type annotation or an inheritance clause of a class or interface definition.

```
interface I {}
class A implements I {}    // I refers to type I
var x : I  = new A         // In each of these uses too
print( x is I )
var y : I = x as I
```

When a reference is bound to an interface at compile-time, the value of that reference is always the compile-time interface value, even if the interface definition would be shadowed by another property at runtime. For example,

```
interface T {}
class A implements T {}
class B {}
function f() {
    var T = B
    var x = new A
    print(x is T)   // T refers to interface T, not var T, print's true
}
```

In this example, `T` in the `is` expression refers to the outer interface `T`, not the inner var `T`.

## 10.2 Interface methods

Classes that implement an interface method must use the `public` attribute to implement all interface methods that have the same identifier name.

```
interface I
{
    function f()
```

```
}
interface J
{
    function g()
}

class A implements I
{
    public function f() {}
    public function g() {}
}
```

This example shows a class that implements two inherited interfaces with public qualified methods.

### 10.2.1 Visibility of interface methods

Interface methods are visible when referenced through a property of the corresponding interface type, or through a reference to the implementing class or subclass.

```
var a : A = new A
a.f()     // okay, f is visible through an A as {public}::f
a.g()     // okay, f is visible through an A as {public}::g
var i : I = b
i.f()     // okay, f is still visible through an I as {I}::f
i.g()     // error, g is not visible through an I as {I}::g
```

References through an object with an interface type are multinames that contain only the names qualified by the interface namespace and its super interface namespaces. This means that the names in the open namespaces (including public) will not be visible through a reference with an interface typed base object. The motivation for this behavior is to express the idea of the interface as a contract between the producer and consumer of an object, and the contract being specified by the names in the interface namespace alone.

If the compile-time type of the base object is not an interface type, an unqualified reference will use the currently open namespaces (which includes `public`) to create a multiname in the normal way. Again, ambiguous references can be explicitly qualified with the interface name to avoid conflicts.

### 10.2.2 Inheritance of interface methods

The rules for implementing an inherited interface method are the same as the rules for overriding an inherited class method. Specifically, the name of the method, number and type of the parameters, and return type must match exactly.

It is a verification error if a class implements an interface method with a method whose name matches, but the parameter count or types, or return type do not match. It is a verifier error if a class inherits an interface method that it does not implement.

## 10.3 Interface example

Here is an example of how interfaces are defined and used.

```
interface T
{
    function f()
```

```
}

interface U
{
    function f()
    function g()
}

interface V extends T,U
{
    function h()
}

class A implements V
{
    public function f() {}   // implements {T,U}::f
    public function g() {}   // implements {U}::g
    public function h() {}   // implements {V}::h
}

var a : A = new A
var t : T = a
var u : U = a
var v : V = a

t.f()     // {T}::f referenced, T::f matched
u.g()     // {U}::g referenced, U::g matched
v.f()     // {T,U,V}::f referenced, {T,U}::f matched
v.g()     // {T,U,V}::g referenced, U::g matched
v.h()     // {T,U,V}::h referenced, V::h matched
a.f()     // {public,…}::f referenced, public::f matched

var o = a

o.f()     // {public,…}::f referenced, public::f matched
```

A few highlights of this example are:

- An implementing class must use `public` as an attribute to make the method implement all interface methods with a matching identifier
- The static type of the base object of a reference controls which interface names are open in that reference if that type is an interface type

# 11 Packages

A package definition introduces a top-level namespace, suitable for organizing collections of type definitions into APIs.

Unlike ordinary namespaces (hereafter referred to simply as namespaces), a package is a pure compile-time construct. A package directive qualifies the names of properties defined inside of it at compile-time; references to a package's member definitions are given fully qualified names at compile-time.

```
package mx.core
{
    class UIObject extends ...
    {
    }

    .
    .
    .
}
```

In this example, the fully qualified name for `UIObject` is `mx.core.UIObject`. An unqualified reference to `UIObject` will be fully-qualified as `mx.core.UIObject` by the compiler.

Package definitions may be discontinuous; the definition of a package may be spread over multiple package definitions, possibly in multiple source files.

The semantics of loading packages is outside of the language definition. The compiler and virtual machine will have access to the package definitions in files that have been loaded by the embedding tool or runtime.

[Compatibility note: In the Netscape proposal, packages are sealed values that could contain types and values, and could be dynamically loaded. This is problematic for large libraries because the author has no way to incrementally load a library. The current design does not have this restriction.]

## 11.1 Package namespace

The namespace name (the string used for equality comparision) of a package is the sequence of characters of its name. For example, the package in:

```
package mx.core {
    .
    .
    .
}
```

is given the namespace name "`mx.core`".

A package names are used to:

- qualify the names of top-level definitions in a package
- qualify the names of references to those definitions

import names into other packages.

```
package acme.core
{
    public class Widget { }    // qualifies Widget
}

import acme.core.*             // make visible all names in acme.core
var widget : acme.core.Widget  // distinguishes a reference to Widget
```

Packages exist only at compile-time. The static existence of packages allows us to give them certain properties that would not be possible if they could be manipulated at runtime. In particular,

- package names may have embedded dots
- fully qualified package references may and must be expressed using the dot operator rather than the usual :: syntax for qualified names

But because there is no runtime value for a package name, packages cannot be aliased or otherwise used in an expression that uses a runtime value.

When encountered in a valid context by the compiler, the meaning of a package name becomes fixed; any interpretation at runtime is no longer possible.

For this reason, a package name always shadows locally defined names, independent of the scope chain, when that package name is used on the left hand side of a dot operator.

```
package p
{
    public var x = 10
}
import p.x
function f()
{
    var p = { x : 20 }
    print(p.x)   // prints 10
}
f()
```

[Rationale: the alternative is to avoid such conflict by making it an error to define any name that has an identifier that matches the identifier of the lhs of the left most dot of a package name.]

Errors

- it is a strict error to import a package that cannot be found
- it is a strict error to reference a package property that cannot be found in an imported package

### 11.2 Package members

Definitions with the public attribute inside of a package definition are implicitly qualified by the package namespace. Every kind of definition except for package definitions may appear directly inside a package definition, including variable, function, namespace, class, and interface definitions.

### 11.2.1 Package property attributes

The visibility of a name defined inside of a package is controlled by the attributes that appear in that definition. Allowed attributes include,

| | |
|---|---|
| public | Qualified by the package namespace |
| internal | Qualified by the internal namespace for the current package [default] |

It is a syntax error for more than one of these attributes to appear in a definition.

## 11.3 Package import

The names of package members are made visible inside an external scope with an import directive. For example,

```
import mx.core.*
```

makes all public names defined in the package mx.core visible inside any scope that contains this directive. Individual names can be imported using an import directive with the fully qualified name to be imported. For example,

```
import mx.core.Image
```

has the effect of making the class mx.core.Image, but no other names defined inside package mx.core, visible to an unqualified reference.

References to package members are fully qualified using the dot operator. When the meaning of a simple name is ambiguous, a fully qualified name can be used to indicate the intended binding. For example,

```
import mx.core.*
import player.core.*

new Image    // error, mx.core.Image or player.core.Image?
new player.core.Image   // okay
```

[Java compatibility note: unlike in Java, an import directive is required to introduce a package name to a program even when fully qualified names are used. This is to decouple the language semantics of dot expressions from the host dependent behavior of introducing package names to a program. E.g.

```
print(x.y.z)
```

Here, is x.y a package name or a reference to y inside of an object referred to by x?

Given the dynamic nature of the language and the diversity of host environments, we chose to require the programmer to specify through an import statement which packages he intends to use.]

Visibility of package members outside of a package is controlled by access control namespaces. The default namespace of a package member is package internal. For example,

```
package acme.core
{
```

```
    public class Widget { }
    class WidgetImpl {}    // default namespace is internal
}

import acme.core.*
new WidgetImpl   // error, cannot find WidgetImpl
new Widget       // okay, pubic names are always visible
```

In this example, `class WidgetImpl` is in the internal package namespace for package `acme.core`. This namespace is always open inside of any definition of package `acme.core`, and never open or accessible outside of a definition of `acme.core`.

### 11.3.1 Single name aliases

A name alias can be provided for single name import directive to avoid ambiguity of unqualified references. E.g.

```
package acme.core
{
    public class Widget { }
}

package mx.core
{
    public class Widget { }
}

import AcmeWidget = acme.core.Widget
import MxWidget = mx.core.Widget
new AcmeWidget
new MxWidget
```

When an alias is specified, the original fully qualified name can be used to refer to the imported definition. It is also possible to use the original unqualified name as long as the resulting reference is not ambiguous.

## 11.4 Unnamed package

The unnamed package is defined by a package definition with no name specified. E.g.

```
package
{
}
```

The unnamed package is implicitly imported by all other packages and global code outside of any package. This makes it convenient for casual sharing of definitions between programs by making public definitions in the unnamed package always visible.

# 12 Namespaces

Namespaces are used to qualify names. E4X introduced the idea of explicitly qualifying names to reference properties of an XML object. XML namespaces allow markup with various meanings, but potentially conflicting names, to be intermixed in a single use. Packages in As3 provide such a capability. XML namespaces also allow names to be individually qualified to create sub-vocabularies relating to concerns secondary to the main purpose of the markup. Namespaces in As3 provide this capability; that is, controlling the visibility of names independent of the structure of the program. This is useful for giving trusted code special access privileges, and for distinguishing the meaning of a name between versions and uses.

## 12.1 Namespace values

Namespace definitions introduce a constant fixed property of type Namespace into the defining scope. The property is initialized to an implicit or explicit value. Regardless of how it is initialized, a namespace value consists of a namespace name used for equality comparison.

The following example shows the definition of several namespaces,

```
namespace N1
namespace N2 = N1
namespace N3 = 'http://www.ecma-international.org/namespace'
```

`N1` is given an anonymous namespace name. `N2` is an alias of `N1`. `N3` is given a namespace with the namespace name of `'http://www.ecma-international.org/namespace'`. When created by a namespace definition, the prefix of a namespace is initialized to the value undefined.

The set of attributes that may be used on a namespace definition is the same as the set that can be used on a variable definition.

## 12.2 Namespaces as attributes

When used as an attribute of a definition, a namespace specifies the namespace qualifier of that definitions name. E.g.

```
namespace N1
namespace N2
N1 var x : int = 10
N2 var x : String = "hello"
```

Here two distinct variables are defined one with the qualified name `N1::x` and the other with the qualified name `N2::x`. Referencing code can refer to one or the other of these names by explicitly qualifying references to `x` or by adding one or the other namespace to the set of open namespaces.

It is an error to use an user defined namespace as an attribute except in the top-level of a class definition.

## 12.3 Namespaces as qualifiers

References to a name qualified by a namespace can be explicitly qualified by that namespaces. E.g.

```
namespace N1
namespace N2
N1 var x : int = 10
N2 var x : String = "hello"

print(N1::x)
```

In this case the qualification is necessary because an unqualified reference to `x` would not match any visible definition of x, and therefore result in a runtime exception.

### 12.4 Open namespaces

The set of open namespaces determines the visibility of unqualified references. If the qualifier of a name is not in the set of open namespaces it will not be visible to an unqualified reference. Namespaces are added to the list of open namespaces by the `use namespace` directive. Building on the previous example,

```
namespace N1
namespace N2
N1 var x : int = 10
N2 var x : String = "hello"
use namespace N1
print(x)  // print 10
```

Here the namespace `N1` is added to the set of open namespaces. The unqualified reference to `x` matches any name that has the identifier `x` and qualified by one of the open namespaces, in this case `N1::x`.

It is a runtime error for more than one name to match an unqualified reference.

The set of open namespaces includes any namespace that is explicitly used in that block or an outer nested block, as well as the public, internal, protected, and private namespaces that are implicitly open in various contexts.

Bindings of explicitly used namespaces are preferred over names in the public namespace. This allows a public name to be given an open user defined namespace without making unqualified references ambiguous. E.g.

```
namespace N1
N1 var x : int = 10
public var x : String = "hello"
use namespace N1
print(x)  // okay, matches N1::x, even though public::x is also visible
```

### 12.5 Namespace examples

#### 12.5.1 Access control

```
class A {
    private namespace Key
    private friends = [ B ]
    function beMyFriend( suitor ) {
        for each( friend in friends )
        {
            if( suitor is friend ) return Key
        }
        return null
```

```
    }
    Key function makeMyDay()
    {
        print("making my day")
    }
}

class B {
    function befriendAnA(a:A) {
        var key : Namespace = a.beMyFriend(this)
        if( key != null )
        {
            a.key::makeMyDay()
        }
    }
}
```

### 12.5.2 Version control

```
package p {
public namespace V2
public class A {
    public function m() {}
    V2 function m() {}
}
}

import p.v1
import p.v2
import p.A

// version 1

class B extends A
{
    public function m() {}
}

// version 2

class B extends A
{
    public function m() {}
    V2 function m() {}
}

use namespace p.V2  // open p.V2, prefer it over public
var a : A = new B
a.m()
```

### 12.5.3 Vocabulary control

Namespace definitions allow multiple vocabularies to be defined in a single class. This is a kind of polymorphism that is independent of the class abstraction. It is useful when you have common functionality that has a more than one public interface. You could use sub classes to express the overridden behavior, but if there is more than one vocabulary that needs to be mixed in, the number of combinations quickly explodes.

```
package p {
public namespace French
public namespace Vegan
public class Person {
    public function sayIt() { /* say it in English */ }
    French function sayIt() { /* say it in French */ }
    public function eatIt() { /* eat steak */ }
    Vegan  function eatIt() { /* eat vegan */
}
}

import p.*

var person = new Person()

{
    use namespace French
    use namespace Vegan
    person.sayIt()   // speak French
    person.eatIt()   // eat vegan
}

{
    person.sayIt()   // speak English
    person.eatIt()   // eat meat
}
```

# 13 Lexical Structure

[Tbd: specification of lexical tokens]

## 13.1 Lexical

Lexical keywords are removed from the available program namespace during scanning. It is a syntax error to use any of these names except as indicated by the grammar. Syntactic keywords appear to the lexical scanner as identifier tokens, but are given special meaning in certain contexts by the parser.

Keywords:

```
as break case catch class const continue default delete do else extends
false finally for function if implements import in instanceof interface
internal is native new null package private protected public return super
switch this throw to true try typeof use var void while with
```

Identifiers that are syntactic keywords:

```
each get set namespace include dynamic final native override static
```

## 13.2 Syntactic

Identifiers with special meaning (become keywords) in certain syntactic contexts:

In a for-each-in statement between the ‘for’ token and the ‘(‘ token:

```
each
```

In a function definition between the ‘function’ token and an identifier token:

```
get set
```

As the first word of a directive:

```
namespace include
```

In an attribute list or wherever an attribute list can be used:

```
dynamic final native override static
```

It is a syntax error to use a syntactic keyword in a context where it is treated as a keyword. E.g,

```
namespace = “hello”
namespace()
```

In these cases the grammar requires an identifier after the ‘namespace’ keyword.

[Rationale:

Traditionally identifiers with special meaning have been set aside for exclusive use by the implementation. This simplifies the implementation, allows for more precise error reporting and makes the language simpler. Taken literally, this policy dictates that a language reserves all identifiers with special meaning anywhere in the language. The problem with this approach is that it breaks programs written for a version of the language that did not reserve those keywords, and takes commonly used names out of the set of available identifiers.

A case in point is the identifier 'namespace'. 'namespace' is a method name in E4X and a definition label in AS3. Making it a reserved word in AS3 would force the renaming of the E4X method to a less pleasing name. Because this change is incompatible with E4X it will force E4X to be revised incompatibly, breaking any program that uses the XML namespace method.

There are two alternatives to this traditional approach that are less disruptive to existing and future programs include:

- giving new keywords funny names (e.g. __namespace).
- limiting the context in which new keywords have special meaning

Aesthetically the first option is a non-starter leaving us with only the second option when compatibility is important. And because we are designing a language to be useful in environments with different compatibility requirements, we must choose the most compatible rule. Tool chains that favor simplicity over compatibility should provide code hinting and compiler errors or warnings to help users avoid naming pitfalls.

End Rationale]

# 14 Expressions

## 14.1 Identifiers

Identifiers may be either simple identifiers or qualified identifiers. Qualified identifiers result in a single name consisting of a namespace and a string. The string is specified by an expression or a literal identifier. The namespace is specified by an expression that precedes the : : punctuator. Simple identifiers result in one or more names that consist of the identifier string and each of the namespaces open in the scope of the expression. The resulting name value(s) are used to construct a Reference value specified by a larger expression.

### Syntax

```
Identifier
    Identifier
    each
    get
    include
    namespace
    set

PropertyIdentifier
    Identifier
    *

Qualifier
    PropertyIdentifier
    ReservedNamespace

SimpleQualifiedIdentifier
    PropertyIdentifier
    Qualifier :: PropertyIdentifier
    Qualifier :: Brackets

ExpressionQualifiedIdentifier
    ParenExpression :: PropertyIdentifier
    ParenExpression :: Brackets

NonAttributeQualifiedIdentifier
    SimpleQualifiedIdentifier
    ExpressionQualifiedIdentifier

QualifiedIdentifier
    @ Brackets
    @ NonAttributeQualifiedIdentifier
    NonAttributeQualifiedIdentifier
```

Expressions of the form

```
SimpleQualifiedIdentifier : Qualifier :: PropertyIdentifier
```

SimpleQualifiedIdentifier : ParenExpression :: PropertyIdentifier

are syntactically rewritten as

SimpleQualifiedIdentifier : Qualifier :: Brackets
SimpleQualifiedIdentifier : ParenExpression:: Brackets

respectively, where the expression between *Brackets* is a string literal with the same sequence of characters as the *PropertyIdentifier*.

**Verification**

Identifier :: **Identifier**
Identifier :: **each**
Identifier :: **get**
Identifier :: **include**
Identifier :: **namespace**
Identifier :: **set**

- Return the type String

PropertyIdentifier :: Identifier
Qualifier :: PropertyIdentifier
Qualifier :: ReservedNamespace
SimpleQualifiedIdentifier :: PropertyIdentifier

- Return the result of verifying the non-terminal symbol on right side of the production

SimpleQualifiedIdentifier :: Qualifier :: PropertyIdentifier

- Let *qual* be the result of verifying *Qualifier*
- Call verifyType(*qual,*Namespace)
- Return the type Name

SimpleQualifiedIdentifier :: Qualifier :: Brackets

- Let *qual* be the result of verifying *Qualifier*
- Call verifyType(*qual,*Namespace)
- Let *expr* be the result of verifying *Brackets*
- If *expr* is of type Name
  - Throw a VerifierError exception
- Return the type Name

ExpressionQualifiedIdentifier :: ParenExpression :: PropertyIdentifier

- Let *qual* be the result of verifying *ParenExpression*
- Call verifyType(*qual,*Namespace)
- Return the type Name

ExpressionQualifiedIdentifier :: ParenExpression :: Brackets

- Let *qual* be the result of verifying *ParenExpression*
- Call verifyType(*qual,*Namespace)
- Let *expr* be the result of verifying *Brackets*
- If *expr* is of type Name
  - Throw a VerifyError exception

- Return the type Name

NonAttributeQualifier : SimpleQualifiedIdentifier
NonAttributeQualifier : ExpressionQualifiedIdentifier

- Return the result of verifying the non-terminal symbol on right side of the production

QualifiedIdentifier : @ Brackets

- Verify *Brackets*
- Return the type Name

QualifiedIdentifier : @ NonAttributeQualifiedIdentifier
QualifiedIdentifier : NonAttributeQualifiedIdentifier

- Verify *NonAttributeQualifiedIdentiifer*
- Return the type Name

**Evaluation**

Identifier :: **Identifier**
Identifier :: **each**
Identifier :: **get**
Identifier :: **include**
Identifier :: **namespace**
Identifier :: **set**

- Return a new String value consisting of the sequence of characters of the token on the right side of the production

PropertyIdentifier :: Identifier

- Return the result of evaluating *Identifier*

PropertyIdentifier :: *

- Return the new instance String("*")

Qualifier :: PropertyIdentifier
Qualifier :: ReservedNamespace
SimpleQualifiedIdentifier :: PropertyIdentifier

- Return the result of evaluating the non-terminal symbol on right side of the production

SimpleQualifiedIdentifier :: Qualifier :: PropertyIdentifier

- Let *qual* be the result of evaluating *Qualifier*
- Let *str* be the result of evaluating *PropertyIdentifier*
- Return the new instance Name(*qual,str,*false)

SimpleQualifiedIdentifier :: Qualifier :: Brackets

- Let *qual* be the result of evaluating *Qualifier*
- Let *expr* be the result of evaluating *Brackets*
- If *expr* is of type Name
  - Throw a TypeError exception
- Let *str* be the result of calling String(*expr*)

- Let *name* be the instance Name(*qual*,*str*,false)
- Return *name*

ExpressionQualifiedIdentifier :: ParenExpression :: PropertyIdentifier

- Let *qual* be the result of evaluating *ParenExpression*
- Let *str* be the result of evaluating *PropertyIdentifier*
- Return the new instance Name(*qual*,*str*,false)

ExpressionQualifiedIdentifier :: ParenExpression :: Brackets

- Let *qual* be the result of evaluating *ParenExpression*
- Let *expr* be the result of evaluating *Brackets*
- If *expr* is of type Name
  - Throw a TypeError exception
- Let *str* be the result of calling String(*expr*)
- Let *name* be the instance Name(*qual*,*str*,false)
- Return *name*

NonAttributeQualifier : SimpleQualifiedIdentifier
NonAttributeQualifier : ExpressionQualifiedIdentifier

- Return the result of evaluating the non-terminal symbol on right side of the production

QualifiedIdentifier : @ Brackets

- Let *expr* be the result of evaluating *Brackets*
- If *expr* is of type Name
  - Let *name* be the set consisting of *expr*
- Else
  - Let *str* be the result of calling String(*expr*)
  - Let *namespaces* be the result of calling openNamespaces(*ctx*)
  - Let *name* be the result of makeMultiname(*namespaces*,*str*)
- Call makeAttributeName(*name*)
- Return *name*

QualifiedIdentifier : @ NonAttributeQualifiedIdentifier

- Let *name* be the result of evaluating *NonAttributeQualifiedIdentifier*
- Call makeAttributeName (*name*)
- Return *name*

QualifiedIdentifier : NonAttributeQualifiedIdentifier

- Let *name* be the result of evaluating *NonAttributeQualifiedIdentifier*
- Return *name*

## 14.2 Primary expressions

### Syntax

PrimaryExpression

**null**

**true**

**false**
**Number**
**String**
**this**
**RegularExpression**
QualifiedIdentifier
XMLInitializer
ReservedNamespace
ParenListExpression
ArrayInitialiser
ObjectInitialiser
FunctionExpression

A *PrimaryExpression* can be used wherever a *FullPostfixExpression* or a *FullNewSubExpression* can be used. This includes object creation, property access, and function invocation expressions.

### Verification

PrimaryExpression : **null**

- Return the type Null

PrimaryExpression : **true**
PrimaryExpression : **false**

- Return the type Boolean

PrimaryExpression : **Number**

- Return the type Number

PrimaryExpression : **String**

- Return the type String

PrimaryExpression : **RegularExpression**

- Return the type RegExp

PrimaryExpression : QualifiedIdentifier

- Return the result of verifying *QualifiedIdentifier*

PrimaryExpression : XMLInitialiser
PrimaryExpression : ReservedNamespace
PrimaryExpression : ParenListExpression
PrimaryExpression : ArrayInitialiser
PrimaryExpression : ObjectInitialiser
PrimaryExpression : FunctionExpression

- Return the result of verifying the non-terminal symbol on the right side of the production

PrimaryExpression : this

- Let *frame* be the immediately enclosing ParameterFrame

- If *frame* is none
  - Throw a VerifyError
- Return the result of typeOfThis(frame)

**Evaluation**

PrimaryExpression : null

- Return the value `null`

PrimaryExpression : true

- Return the value true

PrimaryExpression : false

- Return the value false

PrimaryExpression : Number

- Return the Number value produced by lexical analysis of *Number*

PrimaryExpression : String

- Return the String value produced by lexical analysis of *String*

PrimaryExpression : this

- Let *frame* be the immediately enclosing ParameterFrame
- Return the value of **this** associated with *frame*

PrimaryExpression : RegularExpression

- Return the RegExp result of evaluating the expression produced by lexical analysis of *RegularExpression*

PrimaryExpression : QualifiedIdentifier

- Let *name* be the result of evaluating QualifiedIdentifier
- Let *ref* be an instance Reference(null,*name*,null)
- Return *ref*

## 14.3 Reserved namespace expressions

**Syntax**

ReservedNamespace
**public**
**private**
**protected**
**internal**

**Verification**

ReservedNamespace : public

- Return the value of type Namespace

ReservedNamespace : private
ReservedNamespace : protected

- If *ReservedNamespace* is not enclosed in a *ClassDefinition*
  - Throw a VerifyError
- Return the value of type Namespace

ReservedNamespace : internal

- If *ReservedNamespace* is not enclosed in a *PackageDefinition*
  - Throw a VerifyError
- Return the value of type Namespace

**Evaluation**

ReservedNamespace : public

- Return the public namespace

ReservedNamespace : private

- Return the private namespace of the enclosing class

ReservedNamespace : protected

- Return the protected namespace of the enclosing class

ReservedNamespace : internal

- Return the internal namespace of the enclosing package

## 14.4 Parenthesized expressions

**Syntax**

ParenExpression
**(** AssignmentExpression$^{allowIn}$ **)**

ParenListExpression
ParenExpression
**(** ListExpression$^{allowIn}$ , AssignmentExpression$^{allowIn}$ **)**

**Verification**

ParenExpression : ( AssignmentExpression$^{allowIn}$ )

- Return the result of verifying *AssignmentExpression*

ParenListExpression : ( ListExpression$^{allowIn}$ , AssignmentExpression$^{allowIn}$ )

- Verify *ListExpression*

- Return the result of verifying *AssignmentExpression*

**Evaluation**

ParenExpression : ( AssignmentExpression$^{allowIn}$ )

- Return the result of evaluating *AssignmentExpression*

ParenListExpression : ( ListExpression$^{allowIn}$ , AssignmentExpression$^{allowIn}$ )

- Evaluate *ListExpression*
- Let *ref* be the result of evaluating *AssignmentExpression*
- Return the result of readReference(*ref*)

## 14.5 Function expression

**Syntax**

FunctionExpression
    **function** FunctionCommon
    **function** Identifier FunctionCommon

**Verification**

FunctionExpression : function FunctionCommon
FunctionExpression : function Identifier FunctionCommon

- Return the result of verifying *FunctionCommon*

**Evaluation**

FunctionExpression : function FunctionCommon

- Return the result of evaluating *FunctionCommon*

FunctionExpression : function Identifier FunctionCommon

- Let *obj* be a new instance of Object
- Push *obj* onto the scope chain
- Let *fun* be the result of evaluating *FunctionCommon*
- Let *id* be the result of evaluating *Identifier*
- Add a property to *obj* with the name *id* and the value *fun* that is not writable and not deletable
- Pop *obj* from the scope chain
- Return *fun*

## 14.6 Object initialiser

**Syntax**

ObjectInitialiser
    **{** FieldList **}**

FieldList

«empty»
NonemptyFieldList

NonemptyFieldList
LiteralField
LiteralField , NonemptyFieldList

LiteralField
FieldName **:** AssignmentExpression$^{allowIn}$

FieldName
NonAttributeQualifiedIdentifier
**String**
**Number**

### Verification

ObjectInitialiser : { FieldList }

- Return the result of verifying *FieldList*

FieldList : empty

- Do nothing

FieldList : NonemptyFieldList

- Verify *NonemptyFieldList*

NonemptyFieldList : LiteralField

- Verify *LiteralField*

NonemptyFieldList : LiteralField , NonemptyFieldList

- Verify *LiteralField*
- Verify *NonemptyFieldList*

LIteralField : FieldName : AssignmentExpression

- Verify *FieldName*
- Verify *AssignmentExpression*

FieldName : NonAttributeQualifiedIdentifier

- Verify *NonAttributeQualifiedIdentifier*

FieldName : String
FieldName : Number

- Do nothing

### Evaluation

ObjectInitialiser : { FieldList }

- Let *obj* be the result of creating a new Object instance
- Return the result of evaluating *FieldList* with argument *obj*

FieldList : empty

- Return the value of the argument *obj*

FieldList : NonemptyFieldList

- Evaluate *NonemptyFieldList* with argument *obj*

NonemptyFieldList : LiteralField

- Evaluate *LiteralField* with argument *obj*

NonemptyFieldList : LiteralField , NonemptyFieldList

- Evaluate LiteralField with argument *obj*
- Evaluate NonemptyFieldList with argument *obj*

LIteralField : FieldName : AssignmentExpression

- Let *name* be the result of evaluating *FieldName*
- Let *ref* be the result of evaluating *AssignmentExpression*
- Let *val* be the value of referenceRead(*ref*)
- Call objectWrite(*obj*,*name*,*val*)

FieldName : NonAttributeQualifiedIdentifier

- Return the result of evaluating *NonAttributeQualifiedIdentifier*

FieldName : String

- Return the value of *String*

FieldName : Number

- Let *num* be the value of *Number*
- Return the result of calling String(*num*)

## 14.7 Array initialiser

An array initialiser is an expression describing the initialisation of an Array object, written in a form of a literal. It is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets. The elements need not be literals; they are evaluated each time the array initialiser is evaluated.

Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an *AssignmentExpression* (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined.

**Syntax**

ArrayInitialiser
**[** ElementList **]**

ElementList
«empty»
LiteralElement
**,** ElementList
LiteralElement , ElementList

LiteralElement
AssignmentExpression$^{allowIn}$

### Verification

An *ArrayInitialiser* is verified by verifying all non-terminals on the right hand side of each production. The result of verifying an *ArrayInitialiser* is the type Array.

### Evaluation

*ArrayInitialiser* expressions are evaluated as described in Ecma-262 edition 3.

## 14.8 XML initialisers

An XML initialiser is an expression describing the initialisation of an XML object, written in a form of a literal. It may specify an XML element, an XML comment, an XML PI, or a CDATA section using ordinary XML syntax. For XML elements, it provides the name, attributes and properties of an XML object.

### Syntax

XMLInitialiser
XMLMarkup
XMLElement
**< >** XMLElementContent **</ >**

XMLElementContent
XMLMarkup XMLElementContent$_{opt}$
XMLText XMLElementContent$_{opt}$
XMLElement XMLElementContent$_{opt}$
**{** Expression **}** XMLElementContent$_{opt}$

XMLElement
**<** XMLTagContent **XMLWhitespace$_{opt}$/>**
**<** XMLTagContent **XMLWhitespace$_{opt}$>** XMLElementContent **</** XMLTagName **XMLWhitespace$_{opt}$>**

XMLTagContent
XMLTagName XMLAttributes

XMLTagName
**{** Expression **}**
**XMLName**

XMLAttributes
**XMLWhitespace {** Expression **}**
XMLAttribute XMLAttributes
Empty

XMLAttribute
**XMLWhitespace** XMLName **XMLWhitespace$_{opt}$ = XMLWhitespace$_{opt}$ {** Expression **}**
**XMLWhitespace** XMLName **XMLWhitespace$_{opt}$ = XMLWhitespace$_{opt}$ XMLAttributeValue**

XMLElementContent
**{** Expression **}** XMLElementContent
**XMLMarkup** XMLElementContent
**XMLText** XMLElementContent
XMLElement XMLElementContent
Empty

### Verification

An *XMLInitialiser* is verified by verifying all non-terminals on the right hand side of each production. The result of verifying an *XMLInitialiser* is the type XML.

### Evaluation

*XMLInitialiser* expressions are evaluated as described in Ecma-357: Ecmascript for XML.

## 14.9 Super expression

SuperExpression limits the binding of a reference to a property of the base class of the current method. The value of the operand must be an instance of the current class. If *Arguments* is specified, its value is used as the base object of the limited reference. If no *Arguments* is specified, the value of `this` is used as the base object.

### Syntax

SuperExpression
**super**
**super** Arguments

SuperExpression may be used before a PropertyOperator in either a FullPostfixExpression or a FullNewSubexpression.

```
super.f(a,b,c)
super(o).f(a,b,c)
```

### Verification

SuperExpression : super
SuperExpression : super Arguments

- Let *frame* be the immediately enclosing ParameterFrame
- If *frame* is none
  - Throw a VerificationError
- Let *type* be the result of typeOfThis(frame)

- Let *limit* be *type*.super
- If *Arguments* is specified and not empty
  - Let *obj* be the result verifying *Arguments*
  - Call verifyType(*obj*,*limit*)
- Return the type *limit*

**Evaluation**

SuperExpression : super
SuperExpression : super Arguments

- Let *frame* be the immediately enclosing *ParameterFrame*
- Let *this* be the value of *frame*.this
- Let *type* be the value of *this*.type
- Let *limit* be *type*.super
- If *Arguments* is empty or not specified
  - Let *obj* be the value of *this*
- Else
  - Let *obj* be the result of evaluating *Arguments*
  - If *obj*.type is not a subtype of *limit*, then throw a TypeError
- Let *obj* be a new instance LimitedBase(*obj*,*limit*)

**Compatibility**

As2 supports only the first form of SuperExpression.

```
super.f(a,b,c)
```

This is equivalent in As2 to

```
this.constructor.prototype.__proto__.f.apply(this,arguments);
```

This differs from the As3 depending on the value of this, and whether the value of constructor, prototoype or __proto__ has been modified.

The second form of SuperExpression is included for the sake of future compatibility and completeness.

## 14.10 Postfix Expressions

**Syntax**

PostfixExpression
  FullPostfixExpression
  ShortNewExpression

A PostfixExpression may be used in a UnaryExpression, before ++ or -- in another PostfixExpression on the left hand side of an AssignmentExpression, or as a ForInBinding.

FullPostfixExpression
  PrimaryExpression
  FullNewExpression

FullPostfixExpression PropertyOperator
SuperExpression PropertyOperator
FullPostfixExpression Arguments
FullPostfixExpression QueryOperator
PostfixExpression [no line break] **++**
PostfixExpression [no line break] **--**

A FullPostfixExpression may be used as a PostfixExpression, or before a PropertyOperator or an Arguments in another FullPostfixExpression.

**Verification**

FullPostfixExpression : PrimaryExpression
FullPostfixExpression : FullNewExpression

- Return the result of verifying the right hand side of the production

FullPostfixExpression : FullPostfixExpression PropertyOperator

- Let *base* be the result of verifying *FullPostfixExpression*
- Let *name* be result of verifying *PropertyOperator*
- Return the result of referenceType(*base*,*name*,null,false)

FullPostfixExpression : SuperExpression PropertyOperator

- Let *base* be the result of verifying *SuperExpression*
- Let *name* be result of verifying *PropertyOperator*
- Return the result of referenceType(*base*.this,*name*,*base*.limit,false)

FullPostfixExpression : FullPostfixExpression Arguments

- Let *fun* be the result of verifying *FullPostfixExpression*
- Let *args* be the result of verifying *Arguments*
- If isStrict()
  - Call verifyType(*fun*,Function)
  - Let *types* be the value *fun*.types
  - If *args*.length is not equal to *types*.length, throw a VerifyError exception
  - For each type in *args*, call verifyType(*args*[i],*types*[i])
- Return the result of resultType(*fun*)

FullPostfixExpression : FullPostfixExpression QueryOperator

- Let *type* be the result of verifying *FullPostfixExpression*
- Return the result of verifying *QueryOperator* passing the argument *type*

FullPostfixExpression : PostfixExpression [no line break] ++
FullPostfixExpression : PostfixExpression [no line break] --

- Let *type* be the result of verifying *PostfixExpression*
- Call verifyType(*type*,Number)
- Return type Number

**Evaluation**

FullPostfixExpression : PrimaryExpression

- Return the result of evaluating *PrimaryExpression*

FullPostfixExpression : FullNewExpression

- Return the result of evaluating *FullNewExpression*

FullPostfixExpression : FullPostfixExpression PropertyOperator

- Let *ref* be the result of evaluating *FullPostfixExpression*
- Let *base* be the result of readReference(*ref*)
- Let *name* be the result of evaluating *PropertyOperator*
- Return the new instance Reference(*base*,*name*,null,false)

FullPostfixExpression : SuperExpression PropertyOperator

- Let *limited* be the result of evaluating *SuperExpression*
- Return the new instance Reference(*limited*.this,*name*,*limited*.type,false)

FullPostfixExpression : FullPostfixExpression QueryOperator

- Let *ref* be the result of evaluating *FullPostfixExpression*
- Let *obj* be the result of readReference(*ref*)
- Return the result of evaluating *QueryOperator* passing the argument *obj*

FullPostfixExpression : FullPostfixExpression Arguments

- Let *ref* be the result of evaluating *FullPostfixExpression*
- Let *args* be the result of evaluating *Arguments*
- Return the result of callReference(*ref*,*args*)

FullPostfixExpression : PostfixExpression [no line break] ++

- Let *ref* be the result of evaluating *PostfixExpression*
- Let *val* be the result of readReference(*ref*)
- Let *num1* be the result of Number(*val*)
- Let *num2* be the result of evaluating the expression *num1* + 1
- Call writeReference(*ref*,*num2*)
- Return *num1*

FullPostfixExpression : PostfixExpression [no line break] –

- Let *ref* be the result of evaluating *PostfixExpression*
- Let *val* be the result of readReference(*ref*)
- Let *num1* be the result of Number(*val*)
- Let *num2* be the result of evaluating the expression *num1* - 1
- Call writeReference(*ref*,*num2*)
- Return *num1*

### 14.11 New expressions

A new expression results in the invocation of the intrinsic construct method of the value computed by the expression that follows the new keyword. Arguments, if specified, are passed to the construct method. If no arguments are specified, the parentheses may be omitted.

**Syntax**

FullNewExpression
    **new** FullNewSubexpression Arguments

A FullNewExpression may be used as a FullPostfixExpression, or as a FullNewSubexpression.

FullNewSubexpression
    PrimaryExpression
    FullNewExpression
    FullNewSubexpression PropertyOperator
    SuperExpression PropertyOperator

A FullNewSubexpression may be used between the `new` keyword and the Arguments in a FullNewExpression, before a PropertyOperator in another FullNewSubexpression, or as a ShortNewSubexpression.

ShortNewExpression
    **new** ShortNewSubexpression

A ShortNewExpression may be used as a PostfixExpression, or as a ShortNewSubexpression (that is, after the `new` keyword in another ShortNewExpression.)

ShortNewSubexpression
    FullNewSubexpression
    ShortNewExpression

A ShortNewSubexpression may be used after the `new` keyword in a ShortNewExpression.

**Verification**

FullNewExpression : new FullNewSubexpression Arguments

- Let *fun* be the result of verifying *FullNewSubexpression*
- Let *args* be the result of evaluating *Arguments*
- If isStrict()
  - Call verifyType(*fun*,Function)
  - Let *types* be the value *fun*.types
  - If *args*.length is not equal to *types*.length, throw a VerifyError exception
  - For each type in *args*, call verifyType(*args*[i],*types*[i])
- Return the result of calling resultType(fun,new)

FullNewSubxpression : PrimaryExpression
FullNewSubxpression : FullNewExpression

- Return the result of verifying the non-terminal symbol on the right side of the production

FullNewSubxpression : FullNewSubexpression PropertyOperator

- Let *base* be the result of verifying *FullNewSubexpression*
- Let *name* be the result of evaluating *PropertyOperator*
- Return the result of calling propertyType(base,name,null,false)

FullNewSubxpression : SuperExpression PropertyOperator

- Let *limited* be the result of evaluating *SuperExpression*
- Let *name* be the result of evaluating *PropertyOperator*
- Return the result of calling propertyType (*limited*.this,*name*,*limited*.type,false)

ShorNewExpression : new ShortNewSubexpression

- Let *ref* be the result of verifying *ShortNewSubexpression*
- Return the result of calling resultType(fun,new)

ShorNewSubexpression : FullNewSubexpression
ShorNewSubexpression : ShortNewExpression

- Return the result of verifying the non-terminal symbol on the right side of the production

**Evaluation**

FullNewExpression : new FullNewSubexpression Arguments

- Let *ref* be the result of evaluating *FullNewSubexpression*
- Let args be the result of evaluating *Arguments*
- Return the result of constructReference(*ref*,*args*)

FullNewSubxpression : PrimaryExpression

- Return the result of evaluating *PrimaryExpression*

FullNewSubxpression : FullNewExpression

- Return the result of evaluating *FullNewExpression*

FullNewSubxpression : FullNewSubexpression PropertyOperator

- Let *ref* be the result of evaluating *FullNewSubexpression*
- Let *base* be the result of readReference(*ref*)
- Let *name* be the result of evaluating *PropertyOperator*
- Return the new instance Reference(*base*,*name*,null,false)

FullNewSubxpression : SuperExpression PropertyOperator

- Let *limited* be the result of evaluating *SuperExpression*
- Return the new instance Reference(*limited.this*,*name*,*limited.type*)

ShorNewExpression : new ShortNewSubexpression

- Let *ref* be the result of evaluating *ShortNewSubexpression*
- Return the result of constructReference(*ref*,null)

ShorNewSubexpression : FullNewSubexpression

- Return the result of evaluating *FullNewSubexpression*

ShorNewSubexpression : ShortNewExpression

- Return the result of evaluating *ShortNewExpression*

## 14.12 Property accessors

### Syntax

PropertyOperator
    . QualifiedIdentifier
    Brackets

Brackets
    **[** ListExpression$^{allowIn}$ **]**

### Verification

PropertyOperator : . QualifiedIdentifier

- Return the result of verifying *QualifiedIdentifier*

PropertyOperator : Brackets

- Return the result of verifying *Brackets*

Brackets : [ ListExpression ]

- Verify *ListExpression*
- Return the type Name

### Evaluation

PropertyOperator : . QualifiedIdentifier

- Return the result of evaluating *QualifiedIdentifier*

PropertyOperator : Brackets

- Return the result of evaluating *Brackets*

Brackets : [ ListExpression ]

- Let *val* be the result of evaluating *ListExpression*
- If *val* is of type Name
    - Let *name* be the set of names consisting of *val*
- Else
    - Let *str* be the result of calling String(*val*)
    - Let *namespaces* be the result of calling openNamespaces(*ctx*)
    - Let *name* be the value of makeMultiname(*namespaces,str*)
- Return *name*

## 14.13 Query operators

### Syntax

QueryOperator
    .. QualifiedIdentifier
    . ( ListExpression[allowIn] )

### Verification

QueryOperator : .. QualifiedIdentifier

- Let *type* be a named argument to this verifier
- Call verifyType(*type,*XML)
- Verify *QualifiedIdentifier*
- Return type XMLList

QueryOperator: . ( ListExpression )

- Let *type* be a named argument to this verifier
- Call verifyType(*type,*XML)
- Verify *ListExpression*
- Return type XMLList

### Evaluation

*QueryOperator* expressions are evaluated as described in Ecma-357 (E4X)

## 14.14 Call expressions

### Syntax

Arguments
    ( )
    ( ListExpression[allowIn] )

ArgumentList[β]
    AssignmentExpression[β]
    ArgumentList[β] , AssignmentExpression[β]

### Verification

Arguments : ()

- Return an empty array of types

Arguments : ( ArgumentList )

- Let *argTypes* be an empty array of types
- Verify *ArgumentList* passing the argument *argTypes*
- Return *argTypes*

ArgumentList : AssignmentExpression

- Let *argTypes* be a named argument to this verifier
- Let *type* be the result of verifying *AssignmentExpression*
- Call push(*argTypes ,type*)

ArgumentList : ArgumentList , AssignmentExpression

- Let *args* be the result of evaluating *ArgumentList* with argument *argTypes*
- Let *type* be the result of evaluating *AssignmentExpression*
- Call push(*argTypes ,type*)

**Evaluation**

Arguments : ()

- Return an empty Array

Arguments : ( ArgumentList )

- Let *args* be an empty Array
- Evaluate *ArgumentList* passing the argument *args*

ArgumentList : AssignmentExpression

- Let val be the result of evaluating *AssignmentExpression*
- Call push(*args,val*)
- Return

ArgumentList : ArgumentList , AssignmentExpression

- Evaluate *ArgumentList* passing the argument *args*
- Let *val* be the result of evaluating *AssignmentExpression*
- Call push(*args,val*)
- Return

## 14.15 Unary expressions

**Syntax**

UnaryExpression
PostfixExpression
**delete** PostfixExpression
**void** UnaryExpression
**typeof** UnaryExpression
**++** PostfixExpression
**--** PostfixExpression
**+** UnaryExpression
**-** UnaryExpression
**- NegatedMinLong**
**~** UnaryExpression
**!** UnaryExpression

A UnaryExpression may be used where ever a MultiplicativeExpression may be used, and in another UnaryExpression after the `void` or `typeof` keywords or after the **+**, **-**, **~**, and **!** punctuators.

**Verification**

UnaryExpression : PostfixExpression

- Return the result of verifying *PostfixExpression*

UnaryExpression : delete PostfixExpression

- Verify *PostfixExpression*
- Return the type Boolean

UnaryExpression : void UnaryExpression

- Verify *UnaryExpression*
- Return the type void

UnaryExpression : typeof UnaryExpression

- Verify *UnaryExpression*
- Return the type String

UnaryExpression : ++ PostfixExpression
UnaryExpression : -- PostfixExpression
UnaryExpression : + PostfixExpression
UnaryExpression : - PostfixExpression

- Let *type* be the result of verifying *PostfixExpression*
- Call verifyType(*type,*int)
- Return the type Number

UnaryExpression : - NegatedMinLong

- Return the type Number

UnaryExpression : ~ UnaryExpression

- Let *type* be the result of verifying *UnaryExpression*
- Call verifyType(*type,*int)
- Return the type int

UnaryExpression : ! UnaryExpression

- Let *type* be the result of verifying *UnaryExpression*
- Call verifyType(*type,*Boolean)
- Return the type Boolean

**Evaluation**

UnaryExpression : PostfixExpression

- Return the result of evaluating *PostfixExpression*

UnaryExpression : delete PostfixExpression

- Let *ref* be the result of evaluating *PostfixExpression*
- If *ref* is of type Reference
  - Return the result of calling deleteReference(*ref*)
- Else
  - Return **true**

UnaryExpression : void UnaryExpression

- Let *ref* be the result of evaluating *UnaryExpression*
- Call readReference(*ref*)
- Return **undefined**

UnaryExpression : typeof UnaryExpression

- Let *ref* be the result of evaluating *UnaryExpression*
- If *ref* is a Reference and *ref.base* is null
  - Let *val* be the value **undefined**
- Else
  - Let *val* be the result of readReference(*ref*)
- Return the result of typeOfString(*val*)

UnaryExpression : ++ PostfixExpression

- Let *ref* be the result of evaluating *PostfixExpression*
- Let *val* be the result of readReference(*ref*)
- Let *num1* be the result of Number(*val*)
- Let *num2* be the result of calling add(*num1*,1)
- Call writeReference(*ref,num2*)
- Return *num2*

UnaryExpression : -- PostfixExpression

- Let *ref* be the result of evaluating *PostfixExpression*
- Let *val* be the result of readReference(*ref*)
- Let *num1* be the result of Number(*val*)
- Let *num2* be the result of evaluating the expression *subtract(num1,*1)
- Call writeReference(*ref,num2*)
- Return *num2*

UnaryExpression : + PostfixExpression

- Let *ref* be the result of evaluating *PostfixExpression*
- Let *val* be the result of readReference(*ref*)
- Return the result of calling Number(*val*)

UnaryExpression : - PostfixExpression

- Let *ref* be the result of evaluating *PostfixExpression*
- Let *val* be the result of readReference(*ref*)
- Let *num* be the result of Number(*val*)
- If *num* == NaN, then return NaN
- Return the result of the expression multiply(-1,*num*)

UnaryExpression : - NegatedMinLong

- Return the Number value **-2E63**

UnaryExpression : ~ UnaryExpression

- Let *ref* be the result of evaluating *UnaryExpression*
- Let *val* be the result of readReference(*ref*)
- Let *int32* be the result of int(*val*)
- Return the result of bitwiseNot(*int32*)

UnaryExpression : ! UnaryExpression

- Let *ref* be the result of evaluating *UnaryExpression*
- Let *val* be the result of readReference(*ref*)
- Let *bool* be the result of Boolean(*val*)
- If *bool* == **true**
  - Return **false**
- Return **true**

### 14.16 Binary expressions

The binary expressions are left associative have relative precedence as specified in the grammar: LogicalOrExpression has the lowest precedence and MultiplicativeExpression has the highest precedence.

#### 14.16.1 Multiplicative expressions

**Syntax**

MultiplicativeExpression
UnaryExpression
MultiplicativeExpression ***** UnaryExpression
MultiplicativeExpression **/** UnaryExpression
MultiplicativeExpression **%** UnaryExpression

**Verification**

MultiplicativeExpression : UnaryExpression

- Return the result of verifying *UnaryExpression*

MultiplicativeExpression : MultiplicativeExpression ***** UnaryExpression
MultiplicativeExpression: MultiplicativeExpression **/** UnaryExpression
MultiplicativeExpression: MultiplicativeExpression **%** UnaryExpression

- Let *x* be the result of evaluating *MultiplicitiveExpression*
- Call verifyType(x,Number)
- Let *y* be the result of evaluating *UnaryExpression*
- Call verifyType(y,Number)
- Return type Number

**Evaluation**

MultiplicativeExpression : UnaryExpression

- Return the result of evaluating *UnaryExpression*

MultiplicativeExpression : MultiplicativeExpression ***** UnaryExpression

- Let *ref* be the result of evaluating *MultiplicitiveExpression*
- Let *x* be the result of calling readReference(*ref*)
- Let *ref* be the result of evaluating *UnaryExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of calling multiply(*x,y*)

MultiplicativeExpression: MultiplicativeExpression **/** UnaryExpression

- Let *ref* be the result of evaluating *MultiplicitiveExpression*
- Let *x* be the result of calling readReference(*ref*)
- Let *ref* be the result of evaluating *UnaryExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of calling divide(*x,y*)

MultiplicativeExpression: MultiplicativeExpression **%** UnaryExpression

- Let *ref* be the result of evaluating *MultiplicitiveExpression*
- Let *x* be the result of calling readReference(*ref*)
- Let *ref* be the result of evaluating *UnaryExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of calling remainder(*x,y*)

### 14.16.2 Additive expressions

#### Syntax

AdditiveExpression
MultiplicativeExpression
AdditiveExpression **+** MultiplicativeExpression
AdditiveExpression **–** MultiplicativeExpression

#### Verification

AdditiveExpression: MultiplicativeExpression

- Return the result of evaluating *MultiplicativeExpression*

AdditiveExpression: MultiplicativeExpression **+** UnaryExpression

- Let *x* be the result of verifying *MultiplicativeExpression*
- Let *y* be the result of verifying *UnaryExpression*
- Return type *

AdditiveExpression: MultiplicativeExpression **-** UnaryExpression

- Let *x* be the result of verifying *MultiplicativeExpression*
- Call verifyType(*x*,Number)

- Let *y* be the result of verifying *UnaryExpression*
- Call verifyType(*y*,Number)
- Return type Number

**Evaluation**

AdditiveExpression: MultiplicativeExpression

- Return the result of evaluating *MultiplicativeExpression*

AdditiveExpression: MultiplicativeExpression **+** UnaryExpression

- Let *ref* be the result of evaluating *MultiplicativeExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *UnaryExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of plus(*x*,*y*)

AdditiveExpression: MultiplicativeExpression **-** UnaryExpression

- Let *ref* be the result of evaluating *MultiplicativeExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *UnaryExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of minus(*x*,*y*)

### 14.16.3 Shift expressions

**Syntax**

ShiftExpression
AdditiveExpression
ShiftExpression **<<** AdditiveExpression
ShiftExpression **>>** AdditiveExpression
ShiftExpression **>>>** AdditiveExpression

**Verification**

ShiftExpression : AdditiveExpression

- Return the result of verifying *AdditiveExpression*

ShiftExpression : ShiftExpression **<<** AdditiveExpression
ShiftExpression : ShiftExpression **>>** AdditiveExpression
ShiftExpression : ShiftExpression **>>>** AdditiveExpression

- Let *x* be the result of verifying *ShiftExpression*
- Call verifyType(*x*,Number)
- Let *y* be the result of verifying *AdditiveExpression*
- Call verifyType(*y*,Number)
- Return the type Number

**Evaluation**

ShiftExpression : AdditiveExpression

- Return the result of evaluating *AdditiveExpression*

ShiftExpression : ShiftExpression **<<** AdditiveExpression

- Let *ref* be the result of evaluating *ShiftExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *AdditiveExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of shiftLeft(*x*,*y*)

ShiftExpression : ShiftExpression **>>** AdditiveExpression

- Let *ref* be the result of evaluating *ShiftExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *AdditiveExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of shiftRight(*x*,*y*)

ShiftExpression : ShiftExpression **>>>** AdditiveExpression

- Let *ref* be the result of evaluating *ShiftExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *AdditiveExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of shiftRightUnsigned(*x*,*y*)

### 14.16.4 Relational expressions

#### Syntax

RelationalExpression$^{allowIn}$
ShiftExpression
RelationalExpression$^{allowIn}$ **<** ShiftExpression
RelationalExpression$^{allowIn}$ **>** ShiftExpression
RelationalExpression$^{allowIn}$ **<=** ShiftExpression
RelationalExpression$^{allowIn}$ **>=** ShiftExpression
RelationalExpression$^{allowIn}$ **in** ShiftExpression
RelationalExpression$^{allowIn}$ **instanceof** ShiftExpression
RelationalExpression$^{allowIn}$ **is** ShiftExpression
RelationalExpression$^{allowIn}$ **as** ShiftExpression

RelationalExpression$^{noIn}$
ShiftExpression
RelationalExpression$^{noIn}$ **<** ShiftExpression
RelationalExpression$^{noIn}$ **>** ShiftExpression
RelationalExpression$^{noIn}$ **<=** ShiftExpression
RelationalExpression$^{noIn}$ **>=** ShiftExpression
RelationalExpression$^{noIn}$ **instanceof** ShiftExpression
RelationalExpression$^{noIn}$ **is** ShiftExpression

RelationalExpression$^{noIn}$ **as** ShiftExpression

### Verification

RelationalExpression **:** ShiftExpression

- Return the result of verifying *ShiftExpression*

RelationalExpression **:** RelationalExpression **<** ShiftExpression
RelationalExpression **:** RelationalExpression **>** ShiftExpression
RelationalExpression **:** RelationalExpression **<=** ShiftExpression
RelationalExpression **:** RelationalExpression **>=** ShiftExpression
RelationalExpression **:** RelationalExpression **in** ShiftExpression
RelationalExpression **:** RelationalExpression **instanceof** ShiftExpression

- Let *x* be the result of verifying *RelationalExpression*
- Let *y* be the result of verifying *ShiftExpression*
- Return the type Boolean

RelationalExpression : RelationalExpression **is** ShiftExpression

- Verify *RelationalExpression*
- Let *type* be the result of verifying *ShiftExpression*
- Call verifyType(*type,*Type)
- Return the type Boolean

RelationalExpression : RelationalExpression **as** ShiftExpression

- Verify *RelationalExpression*
- Let *type* be the result of verifying *ShiftExpression*
- Call verifyType(*type,*Type)
- Return the *type*

### Evaluation

RelationalExpression **:** ShiftExpression

- Return the result of evaluating *ShiftExpression*

RelationalExpression **:** RelationalExpression **<** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of lessThan(*x,y*)

RelationalExpression **:** RelationalExpression **>** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)

- Return the result of lessThan(*y*,*x*)

RelationalExpression **:** RelationalExpression **<=** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of lessThanOrEquals(*x*,*y*)

RelationalExpression **:** RelationalExpression **>=** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of lessThanOrEquals(*y*,*x*)

RelationalExpression **:** RelationalExpression **in** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of hasProperty(*x*,*y*)

RelationalExpression **:** RelationalExpression **instanceof** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of instanceof(*x*,*y*)

RelationalExpression : RelationalExpression **is** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of isType(*x*,*y*)

RelationalExpression : RelationalExpression **as** ShiftExpression

- Let *ref* be the result of evaluating *RelationalExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *ShiftExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of asType(*x*,*y*)

### 14.16.5 Equality expressions

### Syntax

EqualityExpression
RelationalExpression
EqualityExpression **==** RelationalExpression
EqualityExpression **!=** RelationalExpression
EqualityExpression **===** RelationalExpression
EqualityExpression **!==** RelationalExpression

### Verification

EqualityExpression : RelationalExpression

- Return the result of verifying *RelationalExpression*

EqualityExpression : EqualityExpression **==** RelationalExpression
EqualityExpression : EqualityExpression **!=** RelationalExpression
EqualityExpression : EqualityExpression **===** RelationalExpression
EqualityExpression : EqualityExpression **!==** RelationalExpression

- Let *x* be the result of verifying *EqualityExpression*
- Let *y* be the result of verifying *RelationalExpression*
- If isStrict() and x is not a subtype of y and y is not a subtype of x
  - If x is final or y is not an interface, then throw a type error
  - If y is final or x is not an interface, then throw a type error
- Return type Boolean

### Evaluation

EqualityExpression : RelationalExpression

- Return the result of evaluating *RelationalExpression*

EqualityExpression : EqualityExpression **==** RelationalExpression

- Let *ref* be the result of evaluating *EqualityExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *RelationalExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of equals(*x*,*y*)

EqualityExpression : EqualityExpression **!=** RelationalExpression

- Let *ref* be the result of evaluating *EqualityExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *RelationalExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of not equals(*x*,*y*)

EqualityExpression : EqualityExpression **===** RelationalExpression

- Let *ref* be the result of evaluating *EqualityExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *RelationalExpression*

- Let *y* be the result of readReference(*ref*)
- Return the result of strictEquals(*x*,*y*)

EqualityExpression : EqualityExpression **!==** RelationalExpression

- Let *ref* be the result of evaluating *EqualityExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *RelationalExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of not strictEquals(*x*,*y*)

### 14.16.6 Bitwise expressions

#### Syntax

BitwiseAndExpression
    EqualityExpression
    BitwiseAndExpressionr & EqualityExpression

BitwiseXorExpression
    BitwiseAndExpression
    BitwiseXorExpression ^ BitwiseAndExpression

BitwiseOrExpression
    BitwiseXorExpression
    BitwiseOrExpression | BitwiseXorExpression

#### Verification

BitwiseAndExpression : EqualityExpression

- Return the result of verifying *EqualityExpression*

BitwiseAndExpression : BitwiseAndExpressionr & EqualityExpression

- Let *x* be the result of evaluating *BitwiseAndExpression*
- Call verifyType(*x*,Number)
- Let *y* be the result of evaluating *EqualityExpression*
- Call verifyType(*y*,Number)
- Return the type Number

BitwiseXorExpression : BitwiseAndExpression

- Return the result of evaluating *BitwiseAndExpression*

BitwiseXorExpression : BitwiseXorExpression **^** BitwiseAndExpression

- Let *x* be the result of evaluating *BitwiseXorExpression*
- Call verifyType(*x*,Number)
- Let *y* be the result of evaluating *BitwiseAndExpression*
- Call verifyType(*y*,Number)
- Return the type Number

BitwiseOrExpression : BitwiseXorExpression

- Return the result of evaluating *BitwiseXorExpression*

BitwiseOrExpression : BitwiseOrExpression | BitwiseXorExpression

- Let *x* be the result of evaluating *BitwiseOrExpression*
- Call verifyType(*x*,Number)
- Let *y* be the result of evaluating *BitwiseXorExpression*
- Call verifyType(*y*,Number)
- Return the type Number

**Evaluation**

BitwiseAndExpression : EqualityExpression

- Return the result of evaluating *EqualityExpression*

BitwiseAndExpression : BitwiseAndExpressionr & EqualityExpression

- Let *ref* be the result of evaluating *BitwiseAndExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *EqualityExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of bitwiseAnd(*x*,*y*)

BitwiseXorExpression : BitwiseAndExpression

- Return the result of evaluating *BitwiseAndExpression*

BitwiseXorExpression : BitwiseXorExpression ^ BitwiseAndExpression

- Let *ref* be the result of evaluating *BitwiseXorExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *BitwiseAndExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of bitwiseXor(*x*,*y*)

BitwiseOrExpression : BitwiseXorExpression

- Return the result of evaluating *BitwiseXorExpression*

BitwiseOrExpression : BitwiseOrExpression | BitwiseXorExpression

- Let *ref* be the result of evaluating *BitwiseOrExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *BitwiseXorExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of bitwiseOr(*x*,*y*)

### 14.16.7 Logical expressions

**Syntax**

LogicalAndExpression
    BitwiseOrExpression
    LogicalAndExpression **&&** BitwiseOrExpression

LogicalXorExpression
    LogicalAndExpression
    LogicalXorExpression **^^** LogicalAndExpression

LogicalOrExpression
    LogicalXorExpression
    LogicalOrExpression **||** LogicalXorExpression

### Verification

LogicalAndExpression: BitwiseOrExpression

- Return the result of verifying *BitwiseOrExpression*

LogicalAndExpression: LogicalAndExpression **&&** BitwiseOrExpression

- Let *x* be the result of evaluating *LogicalAndExpression*
- Let *y* be the result of evaluating *BitwiseOrExpression*
- Return the type *

LogicalXorExpression : LogicalAndExpression

- Return the result of evaluating *LogicalAndExpression*

LogicalXorExpression : LogicalXorExpression **^^** LogicalAndExpression

- Let *x* be the result of evaluating *LogicalXorExpression*
- Let *y* be the result of evaluating *LogicalAndExpression*
- Return the type *

LogicalOrExpression : LogicalXorExpression

- Return the result of evaluating *LogicalXorExpression*

LogicalOrExpression : LogicalOrExpression **||** LogicalXorExpression

- Let *x* be the result of evaluating *LogicalOrExpression*
- Let *y* be the result of evaluating *LogicalXorExpression*
- Return the type *

### Evaluation

LogicalAndExpression: BitwiseOrExpression

- Return the result of evaluating *BitwiseOrExpression*

LogicalAndExpression: LogicalAndExpression **&&** BitwiseOrExpression

- Let *ref* be the result of evaluating *LogicalAndExpression*
- Let *x* be the result of readReference(*ref*)

- Let *ref* be the result of evaluating *BitwiseOrExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of logicalAnd(*x,y*)

LogicalXorExpression : LogicalAndExpression

- Return the result of evaluating *LogicalAndExpression*

LogicalXorExpression : LogicalXorExpression **^^** LogicalAndExpression

- Let *ref* be the result of evaluating *LogicalXorExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *LogicalAndExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of logicalXor(*x,y*)

LogicalOrExpression : LogicalXorExpression

- Return the result of evaluating *LogicalXorExpression*

LogicalOrExpression : LogicalOrExpression **||** LogicalXorExpression

- Let *ref* be the result of evaluating *LogicalOrExpression*
- Let *x* be the result of readReference(*ref*)
- Let *ref* be the result of evaluating *LogicalXorExpression*
- Let *y* be the result of readReference(*ref*)
- Return the result of logicalOr(*x,y*)

## 14.17 Conditional expressions

### Syntax

ConditionalExpression
  LogicalOrExpression
  LogicalOrExpression **?** AssignmentExpression **:** AssignmentExpression

A ConditionalExpression may be used where ever an AssignmentExpression may be used.

```
y = true ? x = true : x = false
```

### Verification

ConditionalExpression : LogicalOrExpression

- Return the result of verifying *LogicalOrExpression*

ConditionalExpression : LogicalOrExpression **?** AssignmentExpression **:** AssignmentExpression

- Verify all non-terminal symbols on the right side of the production
- Return type *

### Evaluation

ConditionalExpression : LogicalOrExpression

- Return the result of evaluating *LogicalOrExpression*

ConditionalExpression : LogicalOrExpression **?** AssignmentExpression : AssignmentExpression

- Let *ref* be the result of evaluating *LogicalOrExpression*
- Let *val* be the result of readReference(*ref*)
- If Boolean(*val*) is equal to true
  - Return the result of evaluating the first *AssignmentExpression*
- Else
  - Return the result of evaluating the second *AssignmentExpression*

## 14.18 Non-assignment expressions

### Syntax

NonAssignmentExpression[β]
  LogicalOrExpression[β]
  LogicalOrExpression[β] **?** NonAssignmentExpression[β] **:** NonAssignmentExpression[β]

A NonAssignmentExpression may be used where ever a TypeExpression may be used.

```
var x : hintString ? String : Number
```

### Verification

NonAssignmentExpression : LogicalOrExpression

- Return the result of verifying *LogicalOrExpression*

ConditionalExpression : LogicalOrExpression **?** AssignmentExpression : AssignmentExpression

- Verify all non-terminal symbols on the right side of the production
- Return type *

### Evaluation

NonAssignmentExpression : LogicalOrExpression

- Return the result of evaluating *LogicalOrExpression*

NonAssignmentExpression : LogicalOrExpression **?** AssignmentExpression : AssignmentExpression

- Let *ref* be the result of evaluating *LogicalOrExpression*
- Let *val* be the result of readReference(*ref*)
- If Boolean(*val*) is equal to true
  - Return the result of evaluating the first *AssignmentExpression*
- Else
  - Return the result of evaluating the second *AssignmentExpression*

## 14.19 Assignment expressions

### Syntax

AssignmentExpression

ConditionalExpression

PostfixExpression **=** AssignmentExpression

PostfixExpression CompoundAssignment AssignmentExpression

PostfixExpression LogicalAssignment AssignmentExpression

CompoundAssignment

***=**

**/=**

**%=**

**+=**

**-=**

**<<=**

**>>=**

**>>>=**

**&=**

**^=**

**|=**

LogicalAssignment

**&&=**

**^^=**

**||=**

### Verification

AssignmentExpression : PostfixExpression **=** AssignmentExpression
AssignmentExpression : PostfixExpression CompoundAssignment AssignmentExpression
AssignmentExpression : PostfixExpression LogicalAssignment AssignmentExpression

- Let *lhstype* be the result of verifying *PostfixExpression*
- Let *rhstype* be the result of verifying *AssignmentExpression*
- Call verifyType(*rhstype*,*lhstype*)
- Return *rhstype*

### Evaluation

AssignmentExpression : PostfixExpression **=** AssignmentExpression

- Let *ref1* be the result of verifying *PostfixExpression*
- Let *ref*2 be the result of verifying *AssignmentExpression*
- Let *val* be the result of calling readReference(*ref*2)
- Call writeReference(*ref1*,*val*)
- Return *val*

AssignmentExpression : PostfixExpression CompoundAssignment AssignmentExpression
AssignmentExpression : PostfixExpression LogicalAssignment AssignmentExpression

- Let *ref1* be the result of verifying *PostfixExpression*
- Let *ref*2 be the result of verifying *AssignmentExpression*
- Let *val1* be the result of calling readReference(*ref1*)
- Let *val2* be the result of calling readReference(*ref*2)

- Let *val* be the result of calling the operator method that corresponds to *CompoundAssignment* or *LogicalAssignment* with arguments *val1* and *val2*
- Call writeReference(*ref1*,*val*)
- Return *val*

## 14.20 List expressions

### Syntax

ListExpression[β]
    AssignmentExpression[β]
    ListExpression[β] , AssignmentExpression[β]

ListExpression may be used as an ExpressionStatement, after the `case` keyword in a CaseLabel, after the `in` keyword in a ForInStatement, as a ForInitializer, as an OptionalExpression, after the `return` keyword in a ReturnStatement, after the `throw` keyword in a ThrowStatement, in a ParenthesizedListExpression, in a Brackets, or in an Arguments.

### Verification

ListExpression : AssignmentExpression

- Return the result of verifying *AssignmentExpression*

ListExpression : ListExpression , AssignmentExpression

- Verify *ListExpression*
- Return the result of verifying *AssignmentExpression*

### Evaluation

ListExpression : AssignmentExpression

- Let *ref* be the result of evaluating *AssignmentExpression*
- Return the result of readReference(*ref*)

ListExpression : ListExpression , AssignmentExpression

- Evaluate *ListExpression*
- Let *ref* be the result of evaluating *AssignmentExpression*
- Return the result of readReference(*ref*)

## 14.21 Type expressions

### Syntax

TypeExpression[β]
    NonAssignmentExpression[β]

TypeExpression is used in a typed identifier definition, result type definition, and extends and implements declarations of classes and interfaces.

```
var x : String
```

```
function f() : Number { return y }
class A extends B implements C, D {}
```

**Verification**

TypeExpression : AssignmentExpression

- If *AssignmentExpression* consists of the identifier *****
  - Return type *****
- Return the result of verifying *AssignmentExpression*

**Evaluation**

TypeExpression : AssignmentExpression

- Let *ref* be the result of evaluating *AssignmentExpression*
- Let *val* be the result of readReference(*ref*)
- If isType(*val*,Type) equals false
  - Throw TypeError
- Return *val*

# 15 Statements

ω = {abbrev, noShortIf, full}

## Syntax

Statement[ω]
    SuperStatement Semicolon[ω]
    Block
    IfStatement[ω]
    SwitchStatement
    DoStatement Semicolon[ω]
    WhileStatement[ω]
    ForStatement[ω]
    WithStatement[ω]
    ContinueStatement Semicolon[ω]
    BreakStatement Semicolon[ω]
    ReturnStatement Semicolon[ω]
    ThrowStatement Semicolon[ω]
    TryStatement
    ExpressionStatement Semicolon[ω]
    LabeledStatement[ω]
    DefaultXMLNamespaceStatement

Substatement[ω]
    EmptyStatement
    Statement[ω]
    SimpleVariableDefinition Semicolon[ω]

Substatements
    «empty»
    SubstatementsPrefix Substatement[abbrev]

SubstatementsPrefix
    «empty»
    SubstatementsPrefix Substatement[full]

Semicolon[abbrev]
    ;
    **VirtualSemicolon**
    «empty»

Semicolon[noShortIf]
    ;
    **VirtualSemicolon**
    «empty»

Semicolon[full]
    ;
    **VirtualSemicolon**

## 15.1 Empty statement

### Syntax

EmptyStatement
    ;

### Verification

EmptyStatment : ;

- Do nothing

### Evaluation

- Return the value of named argument *cv*

## 15.2 Expression statement

### Syntax

ExpressionStatement
    [lookahead !{ **function, {** }] ListExpression$^{allowIn}$

### Verification

ExpressionStatement : ListExpression

- Verify *ListExpression*

### Evaluation

ExpressionStatement : ListExpression

- Let *ref* be the result of evaluating *ListExpression*
- Return the result of readReference(*ref*)

## 15.3 Super statement

A SuperStatement causes the constructor of the immediate base class to be called. If no SuperStatement is specified, the default constructor of the base class is called. Unlike in Java, a SuperStatement may be used anywhere in the body of the constructor before an instance property is accessed. It is a compile error to use more than one SuperStatement in a constructor.

### Syntax

SuperStatement
    **super** Arguments

A SuperStatement may only be used inside a constructor. It is a syntax error to use a SuperStatement anywhere else in a program.

```
class B extends A {
    function B(x,y,z) {
        super(x,y)
        // other constructor code here
    }
}
```

**Semantics**

**Compatibility**

In As2 a SuperStatement may be used anywhere in a program, except in a static method of a class. It is equivalent to the statement

```
this.constructor.prototype.constructor.apply(this,arguments)
```

If used in a class instance function it will call the class' constructor function using the current value of `this` as the first argument. If used in global code, it will call the global object's class' super constructor.

In As3 a SuperStatement may only be used in an instance constructor. All other uses will result in a syntax error. Also, if the numbetht sr and types of Arguments is not compatible with Parameters of the super constructor, the result is a runtime error.

## 15.4 Block statement

**Syntax**

Block
**{** Directives **}**

## 15.5 Labeled statement

**Syntax**

LabeledStatement
Identifier **:** Substatement

**Verification**

LabeledStatement : Substatement

- Let *breakTargets* be the current set of possible targets of *BreakStatements*
- Let *target* be the sequence of characters of *Identifier*
- If *target* is a member of *breakTargets*, throw a SyntaxError
- Add *target* to *breakTargets* by calling *breakTargets*.push(*target*)
- Verify *Substatement*

**Evaluation**

LabeledStatement : Substatement

- Try
  - Return the result of evaluating *Substatement*
- Catch exception *x* if *x* is of type Break
  - Let *label* be a string value consisting of the same sequence of characters as *Identifier*
  - If *x*.target equals *label*, then return *x*.value
  - Throw *x*

**Compatibility**

As2 does not allow LabeledStatements. This is a compatible change to the language.

## 15.6 Conditional statements

### 15.6.1 If statement

**Syntax**

IfStatement$^{abbrev}$
  **if** ParenListExpression Substatement$^{abbrev}$
  **if** ParenListExpression Substatement$^{noShortIf}$ **else** Substatement$^{abbrev}$

IfStatement$^{full}$
  **if** ParenListExpression Substatement$^{full}$
  **if** ParenListExpression Substatement$^{noShortIf}$ **else** Substatement$^{full}$

IfStatement$^{noShortIf}$
  **if** ParenListExpression Substatement$^{noShortIf}$ **else** Substatement$^{noShortIf}$

**Verification**

IfStatement : **if** ParenListExpression Substatement
IfStatement : **if** ParenListExpression Substatement **else** Substatement

- Verify the non-terminal symbols other right side of the production

**Evaluation**

IfStatement : **if** ParenListExpression Substatement

- Let *cv* be a named argument passed to this evaluator
- Let *ref* be the result of evaluating *ParenListExpression*
- Let *obj* be the result of readReference(*ref*)
- If toBoolean(*obj*) has the value true
  - Return the result of evaluating *Substatement*
- Return *cv*

IfStatement : **if** ParenListExpression Substatement$_1$ **else** Substatement$_2$

- Let *cv* be a named argument passed to this evaluator
- Let *ref* be the result of evaluating ParenListExpression
- Let *obj* be the result of readReference(ref)
- If toBoolean(*obj*) has the value true
  - Return the result of evaluating $Substatement_1$ passing the argument *cv*
- Return the result of evaluating *Substatement2* passing the argument *cv*

### 15.6.2 Switch statement

#### Syntax

SwitchStatement
  **switch** ParenListExpression **{** CaseElements **}**

CaseElements
  «empty»
  CaseLabel
  CaseLabel CaseElementsPrefix CaseElement$^{abbrev}$

CaseElementsPrefix
  «empty»
  CaseElementsPrefix CaseElement$^{full}$

CaseElement$^{\omega}$
  Directive$^{\omega}$
  CaseLabel

CaseLabel
  **case** ListExpression$^{allowIn}$ **:**
  **default :**

#### Semantics

Switch statements have the same syntax and semantics as defined in Ecma-262 edition 3.

## 15.7 Iteration statements

### 15.7.1 Do-while statement

#### Syntax

DoStatement
  **do** Substatement$^{abbrev}$ **while** ParenListExpression

#### Verification

DoStatement : **do** Substatement **while** ParenListExpression

- Let *continueTargets* be the current set of possible targets of continue targets
- Let *breakTargets* be the current set of possible targets of break targets

- Add the label default to *continueTargets* by calling *continueTargets*.push(**default**)
- Add the label default to *breakTargets* by calling *breakTargets*.push(**default**)
- Verify *Substatement*
- Verify *ParenListExpression*

**Evaluation**

DoStatement : **do** Substatement **while** ParenListExpression

- Let *cv* be a named argument passed to this evaluator
- Try
- Loop
  - Try
    - Let *cv* be the result of evaluating *Substatement* with argument *cv*
  - Catch if exception *x* is of type Continue
    - If *x*.label is a member of the current loop's continueTargets, then *cv* = *x*.value
    - Throw *x*
  - Let *ref* be the result of evaluating *ParenListExpression*
  - Let *obj* be the result of readReference(*ref*)
  - If toBoolean(*obj*) is not true, then return *cv*
- Catch if exception *x* is of type Break
  - If *x*.label equals **default** then return *x*.value
  - Throw *x*

### 15.7.2 While statement

**Syntax**

WhileStatement
  **While** ParenListExpression Substatement

**Verification**

WhileStatement : **while** ParenLIstExpression Substatement

- Let *continueTargets* be the current set of possible targets of continue targets
- Let *breakTargets* be the current set of possible targets of break targets
- Add the label default to *continueTargets* by calling *continueTargets*.push(**default**)
- Add the label default to *breakTargets* by calling *breakTargets*.push(**default**)
- Verify *ParenListExpression*
- Verify *Substatement*

**Evaluation**

DoStatement : **do** Substatement **while** ParenListExpression

- Let *cv* be a named argument passed to this evaluator
- Try
- Loop
  - Let *ref* be the result of evaluating *ParenListExpression*
  - Let *obj* be the result of readReference(*ref*)
  - If toBoolean(*obj*) is not true, then return *cv*

- Try
  - Let *cv* be the result of evaluating *Substatement* with argument *cv*
- Catch if exception *x* is of type Continue
  - If *x*.label is a member of the current loop's continueTargets, then *cv* = *x*.value
  - Throw *x*

- Catch if exception *x* is of type Break
  - If *x*.label equals **default** then return *x*.value
  - Throw *x*

### 15.7.3 For statements

#### Syntax

ForStatement
**For (** ForInitializer **;** OptionalExpression **;** OptionalExpression **)** Substatement
**For (** ForInBinding **in** ListExpression$^{allowIn}$ **)** Substatement
**For** [no line break] **each (** ForInBinding **in** ListExpression$^{allowIn}$ **)** Substatement

ForInitializer
«empty»
ListExpression$^{noIn}$
VariableDefinition$^{noIn}$

ForInBinding
PostfixExpression
VariableDefinitionKind VariableBinding$^{noIn}$

OptionalExpression
ListExpression$^{allowIn}$
«empty»

#### Semantics

For statements in edition 4 have the same syntax and semantics as defined in edition 3 and E4X

## 15.8 Continue statement

#### Syntax

ContinueStatement
**Continue**
**continue** [no line break] Identifier

#### Verification

ContinueStatement : **continue**

- Let *continueTargets* be the current set of possible continue targets
- If **default** is not a member of *continueTargets*, throw a SyntaxError

ContinueStatement : **continue** Identifier

- Let *continueTargets* be the current set of possible continue targets
- Let *label* be the sequence of characters of *Identifier*
- If *label* is not a member of *continueTargets*, throw a SyntaxError

**Evaluation**

ContinueStatement : **continue**

- Let *cv* be a named argument passed to this evaluator
- Throw the exception Continue(*cv*,**default**)

ContinueStatement : **continue** Identifier

- Let *cv* be a named argument passed to this evaluator
- Let *label* be the sequence of characters of *Identifier*
- Throw the exception Continue(*cv*,*label*)

**Compatibility**

As2 does not allow the second form of ContinueStatement. This is a compatible change.

## 15.9 Break statement

**Syntax**

BreakStatement
**break**
**break** [no line break] Identifier

**Verification**

BreakStatement : **break**

- Let *breakTargets* be the current set of possible break targets
- If **default** is not a member of *breakTargets*, throw a SyntaxError

BreakStatement: **break** Identifier

- Let *breakTargets* be the current set of possible continue targets
- Let *label* be the sequence of characters of *Identifier*
- If *label* is not a member of *breakTargets*, throw a SyntaxError

**Evaluation**

BreakStatement: **break**

- Let *cv* be a named argument passed to this evaluator
- Throw the exception Break(*cv*,**default**)

BreakStatement: **break** Identifier

- Let *cv* be a named argument passed to this evaluator

- Let *label* be the sequence of characters of *Identifier*
- Throw the exception Break(*cv*,*label*)

### Compatibility

As2 does not allow the second form of BreakStatement. This is a compatible change.

## 15.10 With statement

### Syntax

WithStatement$_{\omega}$
**with** ParenListExpression Substatement$_{\omega}$

### Semantics

With statements have the same syntax and semantics as defined in Ecma-262 edition 3.

## 15.11 Return statement

### Syntax

ReturnStatement
**Return**
**return** [no line break] ListExpression$^{allowIn}$

### Verification

ReturnStatement : **return**

- Let *env* be the lexical environment
- If *env* does not contain a parameter frame
  - Throw a SyntaxError exception

ReturnStatement : **return** ListExpression

- Let *env* be the lexical environment
- If *env* does not contain a parameter frame
  - Throw a SyntaxError exception
- Let *frame* be the enclosing parameter frame
- If frame does not allow a return value
  - Throw a SyntaxError exception
- Verify *ListExpression*

### Evaluation

BreakStatement: **return**

- Throw the exception Return(**undefined**)

BreakStatement: **return** ListExpression

- Let *ref* be the result of evaluating *ListExpression*
- Let *obj* be the result of readReference(*ref*)
- Throw the exception Return(*obj*)

## 15.12 Throw statement

### Syntax

ThrowStatement
    **throw** [no line break] ListExpression$^{allowIn}$

### Verification

ThrowStatement : **throw** ListExpression

- Verify *ListExpression*

### Evaluation

ThrowStatement : **throw** ListExpression

- Let *ref* be the result of evaluating *ListExpression*
- Let *obj* be the result of readReference(*ref*)
- Throw the value *obj*

## 15.13 Try statement

### Syntax

TryStatement
    **try** Block CatchClauses
    **try** Block CatchClausesOpt **finally** Block

CatchClausesOpt
    «empty»
    CatchClauses

CatchClauses
    CatchClause
    CatchClauses CatchClause

CatchClause
    **catch (** Parameter **)** Block

### Verification

TryStatement : **try** Block CatchClauses
TryStatement : **try** $Block_1$ CatchClausesOpt **finally** $Block_2$
CatchClausesOpt : CatchClauses
CatchClauses : CatchClause

CatchClauses : CatchClauses CatchClause

- Verify each of the non-terminal symbols on the right side of the production

CatchClause : **catch (** Parameter **)** Block

- Let *frame* be an empty activation frame
- Let *env* be a copy of the current environment with *frame* added
- Verify *Parameter* with the arguments *env* and *frame*
- Verify *Block* with the argument *env*

**Evaluation**

TryStatement : **try** Block CatchClauses

- Let *cv* be a named argument passed to this evaluator
- Try
  - Let *cv* be the result of evaluating *Block* with argument *cv*
- Catch if exception *x* is of type Object (note: excludes Return, Break and Continue exceptions)
  - Let *val* be the result evaluating *CatchClauses*
  - If *val* is not **none**, then return *val*
  - Throw *x*

TryStatement : **try** $Block_1$ CatchClausesOpt **finally** $Block_2$

- Let *cv* be a named argument passed to this evaluator
- Try
  - Let *cv* be the result of evaluating $Block_1$ with argument *cv*
- Catch if exception *x* is of type Object (note: excludes Return, Break and Continue exceptions)
  - Try
    - Let *val* be the result evaluating *CatchClauses*
    - If *val* is not **none**, then let *e* be have the value of *x*
    - Else let *e* be **none**
  - Catch if exception x
    - Let *e* have the value of *x*
- Evaluate $Block_2$
- If *e* is not equal to **none**, then throw *e*, else return *val*

CatchClausesOpt : *empty*

- Return **none**

CatchClausesOpt : CatchClauses

- Return the result of evaluating *CatchClauses*

CatchClauses : CatchClause

- Return the result of evaluating *CatchClause*

CatchClauses : CatchClauses CatchClause

- Let *val* be the result of evaluating *CatchClauses*
- If *val* is not equal to **none**, then return *val*

- Return the result of evaluating *CatchClause*

CatchClause : **catch (** Parameter **)** Block

- Let *env* be a copy of the current lexical environment
- Let *x* be the named argument of this evaluator
- Let *T* be the type of *Parameter*
- Let *name* be the name of *Parameter*
- If *x* is of type *T*
  - Let *scope* be instance of the activation frame of *CatchClause*
  - Add *scope* to the lexical environment *env*
  - Call writeProperty(*scope*,*name*,*x*)
  - Return the result of evaluating *Block*
- Return **none**

# 16 Directives

### Syntax

Directive
- EmptyStatement
- Statement
- AnnotatableDirective
- Attributes [no line break] AnnotatableDirective
- IncludeDirective Semicolon
- NamespaceDefinition Semicolon
- ImportDirective Semicolon
- UseDirective Semicolon
- DefaultXMLNamespaceDirective Semicolon

AnnotatableDirective
- VariableDefinition$^{allowIn}$ Semicolon
- FunctionDefinition
- ClassDefinition
- InterfaceDefinition

Directives
- «empty»
- DirectivesPrefix Directive$^{abbrev}$

DirectivesPrefix
- «empty»
- DirectivesPrefix Directive$^{full}$

## 16.1 Attributes

### Syntax

Attributes

Attribute
AttributeCombination

AttributeCombination
Attribute [no line break] Attributes

Attribute
AttributeExpression
ReservedNamespace
**[** AssignmentExpression$^{allowIn}$ **]**

AttributeExpression
Identifier
AttributeExpression PropertyOperator

An AttributeExpression may be used as an Attribute.

An Attribute of one kind or another may be used before any AnnotatableDirective. AnnotatableDirectives include variable, function, class, interface definitions.

Here is a complete list of reserved attribute names:

```
public
private
internal
protected
override
final
dynamic
native
static
```

**Semantics**

The meaning of an Attribute depends on its compile-time value and its usage. See the description of the definitions being modified by the attribute.

## 16.2 Import directive

**Syntax**

ImportDirective
**import** PackageName **.** ***
**import** PackageName **.** Identifier
**import** Identifier **=** PackageName **.** Identifier

ImportDirective may be used where ever a *Directive* or AnnotatableDirective can be used.

```
import a.b.*
import a.b.x
import y = a.b.x
```

**Semantics**

An *ImportDirective* causes the simple and fully qualified names of one or more public definitions of the specified package to be introduced into the current package. Simple names will be shadowed by identical locally defined names. Ambiguous references to imported names result in runtime errors.

The wildcard form (`import a.b.*`) imports all public names in a package. The single name form (`import a.b.x`) imports only the specified name. The alias form of import directive (`import y = a.b.x`) imports the name on the right hand side of the assignment expression and introduces the identifier on the lefthand side as an alias for that name.

The mechanism for locating and loading imported packages is implementation defined.

**Compatibility**

The As2 behavior of raising an error if there are two classes with the name simple name being imported is deprecated. As3 will import both classes, but references to the shared simple class name will result a compile-time error. Such references must be disambiguated by using a fully qualified class name.

The As2 behavior of implicit import feature is also deprecated and will result in a compile time error in As3. To work around such errors, an explicit import directive must be added to the current package, which imports the referenced class.

## 16.3 Include directive

**Syntax**

IncludeDirective
    **include** [no line break] **String**

An *IncludeDirective* may be used where ever a *Directive* may be used.

```
include "reusable.as"
```

**Semantics**

An *IncludeDirective* results at compile-time in the replacement of the text of the *IncludeDirective* with the content of the stream specified by **String**.

**Compatibility**

In As2 the include keyword is spelled `#include`. This form is deprecated and results in a compile warning in As3.

## 16.4 Use directive

**Syntax**

UseDirective

**use namespace** ListExpression$^{allowIn}$

A UseDirective may be used where ever a Directive or AnnotatableDirective may be used. This includes the top-level of a Program, PackageDefinition and *ClassDefinition*.

```
use namespace ns1, ns2
```

**Semantics**

A UseDirective causes the specified namespaces to be added to the open namespaces and removed when the current block scope is exited. Each sub expression of *ListExpression* must have a compile-time constant Namespace value.

**Compatibility**

*UseDirective* is an extension to As2.

### 16.5 Default XML namespace directive

**Syntax**

DefaultXMLNamespaceDirective
**default** [no line break] **xml** [no line break] **namespace =** NonAssignmentExpression$^{allowIn}$

**Semantics**

*DefaultXMLNamespaceDirective* sets the internal DefaultXMLNamespace property to the value of *NonAssignmentExpression*. If a *DefaultXMLNamespaceDirective* appears in a function definition, the default xml namespace associated with the corresponding function object in initially set to the unnamed namespace.

## 17 Definitions

### 17.1 Variable definition

**Syntax**

VariableDefinition$^{allowIn}$
VariableDefinitionKind VariableBindingList$^{allowIn}$

VariableDefinitionKind
**var**
**const**

VariableBindingList$^{allowIn}$
VariableBinding$^{allowIn}$
VariableBindingList$^{allowIn}$ **,** VariableBinding$^{allowIn}$

VariableBinding
    TypedIdentifier VariableInitialisation

VariableInitialisation
    «empty»
    = VariableInitialiser

VariableInitialiser
    AssignmentExpression
    AttributeCombination

TypedIdentifier
    Identifier
    Identifier : TypeExpression

TypedIdentifier may be used in a VariableBinding or Parameter definition.

```
var x : String = "initial String value of var x"
function plusOne( n : Number ) { return n + 1 }
```

**Semantics**

TypedIdentifer results at compile-time in a variable or parameter that is optionally typed. The TypeExpression, if given, results at compile-time to a Type value. It is used to specify the set of values that are compatible with the variable or parameter being declared.

A *VariableDefinition* may be modified by the following attributes

| | |
|---|---|
| `static` | adds property to the class object |
| `prototype` | adds property to the prototype object |
| `private` | accessible from within the current class |
| `public` | accessible outside the current package |
| `protected` | accessible from within an instance of the current class or a derived classes |
| `internal` | accessible from within the current package |

**Compatibility**

Typed identifier behavior differs between As3 and As2 in two ways. As2 checks for type compatibility using compile-time types at compile-time, while As3 checks for type compatibility using runtime types at runtime. The difference can be seen in the following examples.

```
var s : String = o
function f( s : String ) {}
var o = 10
f(o)    // OK in As2, error in As3
```

In As2 the variable o does not have an explicit compile-time type that can be compared to the type String of the parameter s in the call to function f, so no error is reported. In As3 the value of argument o is compared to the type of the parameter s at runtime, resulting in an error.

```
class A {}
class B extends A { var x }
var a : A = new B
```

```
a.x = 20   // Error in As2, OK in As3 (since instance of B has an x
property)
```

In As2, the compiler uses A, the declared type of a, to conservatively check for valid uses of a, excluding completely safe and reasonable uses of a. In As3 the compiler uses the type of `a` to optimize its use, but does not report type errors. It leaves that task to the runtime.

## 17.2 Function definition

### Syntax

FunctionDefinition
  **function** FunctionName FunctionCommon

### Semantics

A *FunctionDefinition* introduces a new name and binds that name to a new created function object specified by *FunctionCommon*. The implementation of the function object depends on whether the function is static or virtual as indicated by its context and attributes.

A *FunctionDefinition* may be modified by the following attributes

| | |
|---|---|
| `static` | adds property to the class object |
| `prototype` | adds property to the prototype object |
| `final` | adds non-overridable property to each instance |
| `override` | overrides a method of the base class |
| `private` | accessible from within the current class |
| `public` | accessible outside the current package |
| `protected` | accessible from within an instance of the current or a derived classes |
| `internal` | accessible from within the current package |
| `native` | generates a native stub (implementation defined) |

### Syntax

FunctionName
  Identifier
  **get** [no line break] Identifier
  **set** [no line break] Identifier

FunctionName is used inside a FunctionDefinition.

```
function f() {}
function get x () { return impl.x }
function set x (x) { impl.x = x }
```

### Semantics

FunctionName specifies at compile-time the name and kind of function being defined. A name that includes a `get` or `set` modifier specifies that the function being defined is a property accessor.

### 17.2.1 Function body

#### Syntax

```
FunctionCommon
    FunctionSignature
    FunctionSignature Block
```

#### Verification

*FunctionCommon* that is a *FunctionSignature* without a *Block* introduces an abstract method trait. *FunctionCommon* with a *FunctionSignature* followed by a *Block* defines a concrete function. The result of verifying a *FunctionCommon* node is the addition of a method trait to a set of traits associated with an object at runtime.

#### Evaluation

During evaluation a FunctionCommon node is instantiated and activated. Function instantiation is when a lexical environment is associated with a function object. This captured environment is used to activate the function. Activation is when the function is called with a specific receiver (this) and set of arguments.

### 17.2.2 Function signature

#### Syntax

```
FunctionSignature
    ( ) ResultType
    ( Parameters ) ResultType
```

#### Semantics

The function signature defines the set of traits associated with the activation of a function object.

### 17.2.3 Parameter list

In the strict dialect the *Arguments* assigned to *Parameters* must have compatible number and types. In the standard dialect the handling of arguments is the same as edition 3.

#### Syntax

```
Parameters
    «empty»
    NonemptyParameters

NonemptyParameters
    Parameter
```

Parameter , NonemptyParameters
RestParameter

Parameter
TypedIdentifier$^{allowIn}$
TypedIdentifier$^{allowIn}$ = AssignmentExpression$^{allowIn}$

RestParameter
...
... Identifier

### Verification

Parameters : empty

- Do nothing

Parameters : NonemptyParameters

- Verify *NonemptyParameters*

NonemptyParameters : Parameter

- Let *frame* be the named argument passed into this verifier
- Verify *Parameter*
- Let *name* be the *name* of *Parameter*
- Let *type* be the *type* of *Parameter*
- Call defineSlotTrait(*frame*,*name*,*type*,false)

NonemptyParameters : Parameter , NonemptyParameters

- Let *frame* be the named argument passed into this verifier
- Verify *Parameter*
- Verify *NonemptyParameters* with the argument *frame*
- Let *name* be the *name* of *Parameter*
- Let *type* be the *type* of *Parameter*
- Call defineSlotTrait(*frame*,*name*,*type*,false)

NonemptyParameters : RestParameter

- Verify *RestParameter*

Parameter : TypedIdentifier

- Verify *TypedIdentifier*
- Let *name* be the name of *TypedIdentifier*
- Let *type* be the type of *TypedIdentifier*
- Call defineSlotTrait(*frame*,*name*,*type*,**undefined**,**false**)

Parameter : TypedIdentifier **=** AssignmentExpression

- Verify *TypedIdentifier*
- Verify *AssignmentExpression*
- Let *name* be the name of *TypedIdentifier*

- Let *type* be the type of *TypedIdentifier*
- Let *val* be the value of *AssignmentExpression*
- If *val* is equal to **none**, then throw a VerifyError: must be a compile-time constant
- Call defineSlotTrait(*frame*,*name*,*type*,*val*,**false**)

RestParameter : …

- Do nothing

RestParameter : … Identifier

- Let *frame* be a named argument passed into this verifier
- Verify *Identifier*
- Let *name* be the *name* of *Identifier*
- Call defineSlotTrait(*frame*,*name*,Array,false)

### 17.2.4 Result type

**Syntax**

ResultType
«empty»
: TypeExpression$^{allowIn}$

ResultType may be used in a FunctionSignature.

```
function f(x) : Number { return x }
```

**Semantics**

ResultType guarantees the type of the value returned from a function. It is a verify error if the return value does not implicitly convert to the *ResultType* of the function.

**Compatibility**

The As2 behavior of checking types only at compile-time is more permissive than in As3. This will result in new runtime errors in cases such as calling the method shown above with an argument of type String.

## 17.3 Class definition

**Syntax**

ClassDefinition
**Class** ClassName Inheritance Block

ClassDefinition may be used where ever an AnnotatableDirective may be used, which includes where ever a *Directive* can be used and following a list of attributes, except inside of another *ClassDefinition* or *InterfaceDefinition*.

```
class A extends B implements C {}
```

```
dynamic public final class D {}
```

### 17.3.1 Class attributes

Class definitions may be modified by these attributes

| | |
|---|---|
| internal | Visible to references inside the current package (default) |
| public | Visible to references everywhere |
| final | Prohibit extension by sub-classing |
| dynamic | Allow the addition of dynamic properties |

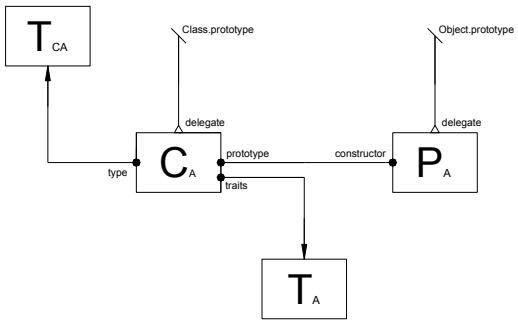
The default attributes for a class definition are internal, non-dynamic, and non-final.

**Semantics**

A class definition adds a new class name into the current scope.

```
class A {}
```

In this example, the class name A refers to a class object with the structure shown in the drawing below:




A class definition causes a class object and prototype instance to be created. The default delegate of the instance prototype is the Object prototype. The default super class of the class object is the Object class. Static members are added to the class object as fixed properties, and non-static members are added to the instance prototype as fixed properties. The internal references (traits, prototype, constructor, and delegate) between these objects are read-only.

### 17.3.2 Class name

**Syntax**

ClassName
    ClassIdentifiers

ClassIdentifiers

Identifier
ClassIdentifiers . Identifier

*ClassName* can be used in *ClassDefinition*.

```
class A {}
```

**Semantics**

ClassName evaluates at compile-time to a type name.

**Compatibility**

The form ClassIdentifiers : ClassIdentifiers . Identifier is deprecated. It is equivalent to declaring the class name Identifier in the package ClassIdentifiers.

```
class P.A {}        // deprecated
package P {         // preferred
    class A {}
}
```

### 17.3.3 Class inheritance

**Syntax**

Inheritance
«empty»
**extends** TypeExpression$^{allowIn}$
**implements** TypeExpressionList
**extends** TypeExpression$^{allowIn}$ **implements** TypeExpressionList

TypeExpressionList
TypeExpression$^{allowIn}$
TypeExpressionList , TypeExpression$^{allowIn}$

**Semantics**

A ClassDefinition may extend another class definition and implement one or more interfaces. We say that a class inherits the properties of its base class and the abstract methods of its interfaces. When a class extends another class it is inherits the base class' instance properties, but not its static properties. When a class implements one or more interfaces it is required to define each inherited interface method.

The *TypeExpressions* that occur in the `extends` and `implements` clauses must be compile-time constant expressions without forward references.

### 17.3.4 Class block

**Syntax**

The body of a class definition is syntactically a *Block*. The class block must come immediately after the *ClassName* or *Inheritance* constituents, if present. The class block must not contain a *ClassDefinition* or *InterfaceDefinition*.

**Semantics**

Declarations modified by the static attribute contribute properties to the class object; declarations without the static attribute contribute properties to the instance traits object. Statements that are not declarations are evaluated normally when the class object is instantiated.

#### 17.3.4.1 Variables

Attributes allowed in variable definitions in a class block are:

| | |
|---|---|
| static | Defines a property of the class object |
| private | Visible to references inside the current class |
| internal | Visible to references inside the current package |
| protected | Visible to references inside an instance of the current class and derived classes |
| prototype | Defines a property of the class prototype object |
| public | Visible to references everywhere |
| *AttributeExpression* | Namespace value is the qualifier for the name of the definition |

The default attributes for variable definitions are non-`static` and `internal`.

#### 17.3.4.2 Methods

Attributes allowed in function definitions in a class block are:

| | |
|---|---|
| static | Defines a property of the class object |
| final | Must not be overridden |
| override | Must redefine an inherited non-final method |
| native | Implementation defined |
| private | Visible to references inside the current class |
| internal | Visible to references inside the current package |
| protected | Visible to references inside instances of the current class and derived classes |
| public | Visible to references everywhere |
| prototype | Defines a property of the class prototype object |
| *AttributeExpression* | Namespace value is the qualifier for the name of the definition |

The default attributes for function definitions in a class are non-`static` , non-`final` , non-`native` and `internal`.

Methods that implement interface methods must be instance methods defined with attributes that include `public`. Interface methods may be overridden in a derived class as long as the overriding method is also has the `public` attribute.

A constructor method is a method with the same name as the class it is defined in. It is a syntax error for the constructor method to have a different namespace attribute than its class.

It is a verifier error for `override` to appear as an attribute of a class method that does not override another method.

## 17.4 Interface definition

### Syntax

InterfaceDefinition
    **interface** ClassName ExtendsList Block

An InterfaceDefinition may be used where ever a Directive or AnnotatableDirective may be used, which includes where ever a *Directive* can be used and following a list of attributes, except inside of another *ClassDefinition* or *InterfaceDefinition*.

```
interface T { function m() }
```

### Semantics

An InterfaceDefinition constrains the structure of any ClassDefinition that implements it. These constraints are enforced when the *ClassDefinition* is being compiled. An InterfaceDefinition also introduces a new type name into the current scope. When evaluated in a context that expects a type value, a reference to that name results the type of all instances of all classes that implement the interface.

### Compatibility

In As2, user defined types only exist at compile-time. Therefore any use of an interface name that cannot be enforced at compile-time will have no effect on the program. See descriptions of *ResultType* and *TypeIdentifier*.

### 17.4.1 Interface attributes

Interface definitions may be modified by these attributes

| | |
|---|---|
| internal | Visible to references inside the current package (default) |
| public | Visible to references everywhere |

The default modifiers for an interface definition are internal.

### 17.4.2 Interface name

The name of an interface definition has the syntax and semantics of a *ClassName* (section 16.3.1).

### 17.4.3 Interface inheritance

### Syntax

ExtendsList
    «empty»
    **extends** TypeExpressionList

An *ExtendsList* may be used after the *ClassName* and before *Block* in an *InterfaceDefinition*.

```
interface U extends T { function n() }
```

**Semantics**

An *ExtendsList* specifies the interfaces that include instances of the current *InterfaceDefinition* in their value set. It also specifies that the current *InterfaceDefinition* inherits the structure of each of the interfaces named in the *ExtendsList*.

An interface definition must not introduce a method with a name that has the same identifier as an inherited method.

An interface definition must not inherit itself directly or indirectly.

### 17.4.4 Interface block

The body of an interface definition is syntactically a *Block*, but must only contain *FunctionDefinitions* with no *Block* and no attribute.

#### 17.4.4.1 Interface methods

Interface methods must be defined with no attribute. An interface method is given the name that has its interface as its qualifier and the identifier as the string.

Interface methods have the syntax of a *FunctionDefinition* without the *Block* of *FunctionCommon*. Class methods that implement interface methods must match the name and signature, including parameter count, types and result type, exactly. The name of the implementing method must have a name that is qualified by the public namespace.

## 17.5 Package definition

**Syntax**

PackageDefinition
    **package** PackageNameOpt Block

A PackageDefinition may be used in a Program before any Directive that is not a PackageDefinition is used.

```
package p {
    public class A {}
    public interface I {}
}
package q {
    public var x = 10
}
import p.*
import q.f
import y = q.x

class B extends A implements I {}
q.f()
print(x)
```

**Semantics**

A PackageDefinition introduces a new package name into the current scope. A package definition causes the public members of that package to be qualified by the package name, and the internal members of that package definition to be qualified by an anonymous namespace that is only accessible to code inside the package.

The statements of a package body are executed in the global scope of the *Program*.

**Compatibility**

*PackageDefinition* is an extension to As2. It is added to As3 to replace the deprecated form of *ClassDefinition* that uses a *ClassName* qualified by a package name.

### 17.5.1 Package name

**Syntax**

PackageName
    Identifier
    PackageName **.** Identifier

## 17.6 Namespace definition

**Syntax**

NamespaceDefinition
    **namespace** NamespaceBinding

NamespaceBinding
    Identifier NamespaceInitialisation

NamespaceInitialisation
    «empty»
    **=** AssignmentExpression$^{allowIn}$

A NamespaceDefinition may be used where ever a Directive or AnnotatableDirective may be used. This includes the top-level of a Program, PackageDefinition and *ClassDefinition*.

```
namespace NS1
namespace NS2 = NS1
namespace NS3 = "http://www.macromedia.com/flash/2005"
```

**Semantics**

A NamespaceDefinition introduces a new namespace constant into the current block scope and assigns to it either an anonymous namespace value, or the value of the *AssignmentExpression* in the *NamespaceInitialisation* implicitly coerced to type Namespace. The value of *NamespaceInitialisation* must be a compile-time constant with a value of type String or type Namespace.

*NamespaceDefinitions* can be annotated by an access specifier (`private, internal, protected` or `public`), the `static` modifier inside a *ClassDefinition*.

## 17.7 Program definition

### Syntax

Program
    Directives
    PackageDefinition Program

```
package P {
    function f() {}
}
package Q {
    function f() {}
}
P.f()
Q.f()
```

# 18 Errors

## 18.1 Class errors

The following errors may occur while parsing or verifying a class definition:

- Defining a class with the name of another definition in the same scope
- Defining a class that extends itself directly or indirectly
- Defining a constructor with a namespace attribute that is different than the namespace attribute of its class
- Defining a constructor with a result type
- Defining a constructor that calls its super constructor more than once
- Defining a constructor that calls its super constructor accessing a non local property
- Introducing a method or variable with the same name as an inherited method or variable
- Overriding a variable
- Overriding a final method
- Overriding a method that is not defined in a base class
- Overriding a method with a method that has a different number, types of parameters, or result type

## 18.2 Interface errors

The following errors may occur while parsing or verifying an interface definition:

- Defining an interface with the name of another definition in the same scope
- Defining an interface that extends itself directly or indirectly
- Defining an interface with a body that contain a definition or statement other than a function definition with no block
- Defining an interface method with the same identifier as an inherited interface method
- Defining an interface method with a attribute

## 18.3 Package errors

- It is a parser error to define a package inside a package
- It is a parser error to use attributes on a package definition
- It is a parser error to import a packages names into itself
- It is a strict error to import the same name more than once into the same package
- It is a strict error to import a package that cannot be found
- It is a strict error to reference a package property that cannot be found in an imported package

## 18.4 Namespace errors

- It is a verifier error to use an expression that does not have a compile-time constant namespace value in a use namespace directive
- It is a verifier error to use an attribute expression that is not a compile-time constant namespace value as an definition attribute
- It is a verifier error to use a user defined namespace as an attribute except to define a class or instance property

# 19 Native objects

The form and function of the native objects is the same as Ecmascript ed. 3 except that all prototype properties are also implemented as class methods. Prototype properties that are functions are implemented as regular methods. Prototype properties that are variables are implemented as a pair of get and set methods that forward state to the prototype property.

## 19.1 Global object

Global object
- NaN
- Infinity
- undefined
- eval
- parseInt
- parseFloat
- isNaN
- isFinite
- decodeURI
- decodeURIComponent
- encodeURI
- encodeURIComponent

## 19.2 Object objects

Object object
- Object
- Object.prototype
- Object.prototype.constructor
- Object.prototype.toString
- Object.prototype.toLocaleString
- Object.prototype.valueOf
- Object.prototype.hasOwnProperty
- Object.prototype.isPrototypeOf
- Object.prototype.propertyIsEnumberable

## 19.3 Function objects

Function object
- Function
- Function.prototype
- Function.prototype.constructor
- Function.prototype.toString
- Function.prototype.apply
- Function.prototype.call
- Function.length

Function.prototype

## 19.4 Array objects

Array object
- Array
- Array.prototype
- Array.prototype.constructor
- Array.prototype.toString
- Array.prototype.toLocaleString
- Array.prototype.concat
- Array.prototype.join
- Array.prototype.pop
- Array.prototype.push
- Array.prototype.reverse
- Array.prototype.shift
- Array.prototype.slice
- Array.prototype.sort
- Array.prototype.splice
- Array.prototype.unshift
- Array.[[Put]]
- Array.length

## 19.5 String objects

String object
- String
- String.prototype
- String.fromCharCode
- String.prototype.constructor
- String.prototype.toString
- String.prototype.valueOf
- String.prototype.charAt
- String.prototype.charCodeAt
- String.prototype.concat
- String.prototype.indexOf
- String.prototype.lastIndexOf
- String.prototype.localeCompare
- String.prototype.match
- String.prototype.replace
- String.prototype.search
- String.prototype.slice
- String.prototype.split
- String.prototype.substring
- String.prototype.toLowerCase
- String.prototype.toLocaleLowerCase
- String.protoype.toUpperCase
- String.protoype.toLocaleUpperCase

String.[[Value]]
String.length

## 19.6 Boolean objects

Boolean object
Boolean
Boolean.prototype
Boolean.prototype.constructor
Boolean.prototype.toString
Boolean.prototype.valueOf

## 19.7 Number objects

Number object
Number
Number.prototype
Number.MAX_VALUE
Number.MIN_VALUE
Number.NaN
Number.NEGATIVE_INFINITY
Number.POSITIVE_INFINITY
Number.protoype.constructor
Number.protoype.toString
Number.prototype.toLocaleString
Number.prototype.valueOf
Number.prototype.toFixed
Number.prototype.toExponential
Number.prototype.toPrecision

## 19.8 Math object

Math object
Math.E
Math.LN10
Math.LN2
Math.LOG2E
Math.LOG10E
Math.PI
Math.SQRT1_2
Math.SQRT2
Math.abs
Math.acos
Math.asin
Math.atan
Math.atan2
Math.ceil

Math.cos
Math.exp
Math.floor
Math.log
Math.max
Math.min
Math.pow
Math.random
Math.round
Math.sin
Math.sqrt
Math.tan

## 19.9 Date objects

Date object
Date
Date.protoype
Date.parse
Date.UTC
Date.prototype.constructor
Date.prototype.toString
Date.prototype.toDateString
Date.prototype.toTimeString
Date.prototype.toLocaleString
Date.prototype.toLocaleDateString
Date.prototype.toLocaletimeString
Date.prototype.valueOf
Date.prototype.getTime
Date.prototype.getFullYear
Date.prototype.getUTCFullYear
Date.prototype.getMonth
Date.prototype.getUTCMonth
Date.prototype.getDate
Date.prototype.getUTCDate
Date.prototype.getDay
Date.prototype.getUTCDay
Date.prototype.getHours
Date.prototype.getUTCHours
Date.prototype.getMinutes
Date.prototype.getUTCMinutes
Date.prototype.getSeconds
Date.prototype.getUTCSeconds
Date.prototype.getMilliseconds
Date.prototype.getUTCMilliseconds
Date.prototype.getTimezoneOffset
Date.prototype.setTime
Date.prototype.setMilliseconds
Date.prototype.setUTCMilliseconds

Date.prototype.setSeconds
Date.prototype.setUTCSeconds
Date.prototype.setMinutes
Date.prototype.setUTCMinutes
Date.prototype.setHours
Date.prototype.setUTCHours
Date.prototype.setDate
Date.prototype.setUTCDate
Date.prototype.setMonth
Date.prototype.setUTCMonth
Date.prototype.setFullYear
Date.prototype.setUTCFullYear
Date.prototype.toUTCString

## 19.10 Error objects

Error object
Error
Error.prototype
Error.prototype.constructor
Error.prototype.name
Error.prototype.message
Error.prototype.toString

# 20 Compatibility with the static profile

The static profile defines a dialect that is a subset of the 4th edition. It allows for the static interpretation of type names and the reporting of verifier errors ahead-of-time.

## 20.1 Static types

## 20.2 Ahead-of-time verification

# 21 Compatibility with the 3rd edition

While we have made this edition as compatible as possible with the 3rd edition, there are certain behaviors for which there is no clear use case and keeping them as-is would have been placed an unneeded heavy burden on the new features of the language. In such cases, we have made small and calculated changes to allow the new definition to be simpler and easier to use.

## 21.1 'this' inside of nested function

In ES3, when 'this' appears in a nested function, it is bound to the global object if the function is called lexically, without an explicit receiver object. In ES3/AS4 'this' is bound to the innermost nested 'this' when the function is called lexically.

## 21.2 No boxing of primitives

In ES3 primitive values (Boolean, Number, String) are boxed in Object values in various contexts. In ES4 primitives are permanently sealed Objects. Unlike boxed objects, attempts to dynamically extend a sealed object results in an run time exception.

## 21.3 Assignment to 'const' is a run time exception

In ES3 primitive assignment to read only properties failed silently. In ES4 such assignment cause a runtime error to be thrown.

## 21.4 Class names are const

In ES3 constructor functions were write-able. In ES4 we implement these propeties with class definitions, which are read only.

## 21.5 Array 'arguments' object

In ES3 the function 'arguments' property is a generic Object. In ES4 'arguments' is an Array.

# 22 Compatibility with E4X

While we have made this edition as compatible as possible with the E4X specification, there are certain behaviors which are either bugs in the original specification of E4X or for which the motivation to revise the behavior outweighs the motivation to keep it the same.

# 23 Compatibility with the Netscape proposal of the 4th edition

The current draft is based on the Netscape proposal dated June 23, 2003.

## 23.1 Removed features

The following language features have been removed.

### 23.1.1 ParenExpression as FieldName in ObjectInitialiser expressions

### 23.1.2 Rest expressions

### 23.1.3 Annotated blocks

### 23.1.4 Pragma directives

### 23.1.5 Built-in types other than `int` and `uint`

### 23.1.6 Type `Never`

### 23.1.7 Local block scope

## 23.2 Modified features

The following language features have been modified.

### 23.2.1 Instance property lookup

## 23.3 Added features

The following language features have been added.

### 23.3.1 Interfaces

### 23.3.2 E4X

# 24 Open Issues

Here are the issues that need to be discussed

## 24.1 Enum like construct

## 24.2 Class initialization order

- What is it?