The ActionScript 3 Language Specification
Names and Lexical Environments

Avik Chaudhuri, Jeff Dyer, Lars Hansen, and Basil Hosmer

Adobe Systems Inc.
`{achaudhu,jodyer,lhansen,bhosmer}@adobe.com`

May 12, 2011

Contents

1 Introduction

1(1) This chapter describes how definitions in a program are organized and referenced using names and lexical environments. A definition is associated with a scope and denoted by a name. The scope is determined by the context in which the definition appears, and the name, which consists of a namespace and an identifier, is determined by the definition itself. The contexts in which the definition can be referenced are controlled by its namespace as well as by its scope. A lexical environment is a structured collection of named definitions ordered by scopes.

1(2) Syntactically, definitions are referenced by *Name*s. A *Name* is desugared to a set of possible names based on context. The process of narrowing the set of names to the name of a definition by looking up in the lexical environment is called name resolution. Name resolution occurs throughout the interpretation of programs, including as early as when the lexical environments are being built at compile time, and as late as when expressions are being evaluated at run time. The meaning of a name may be fixed at compile time, or may vary at run time.

1(3) In certain syntactic contexts names have a restricted form, *RestrictedName*. Unlike names that appear in other contexts, whose evaluation may involve the evaluation of arbitrary expressions at run time, *RestrictedName*s are evaluated at compile time. *RestrictedName*s are important for lexical-environment building and static verification, and may denote namespace attributes of definitions, namespace initializers of namespace definitions, namespaces that are opened by use directives and import directives, class types and interface types that are inherited, as well as type annotations.

1(4) The various sections of this chapter describe the composition of names, the process of building lexical environments, and the process of resolving names of references to names of definitions in lexical environments.

1(5) Sections 2, 3 and 4 develop the required concepts around names, while Sections 5 and 6 rely on those concepts to present algorithms for the construction of lexical environments and the resolution of names. (The reader should note that the contents of the sections in this chapter have some cyclic dependencies: the resolution of names depends on the state of lexical environments, and the building of the lexical environments depends on the resolution of names. Therefore, the reader may not get the full picture of how names and lexical environments work until the whole chapter is read.)

1.1 Structure and Meaning of Names

1(6) *Namespace values*, which are parts of names, are either system-defined or user-defined. System-defined namespace values correspond to *ReservedNamespace*s, which are the `public`, `internal`, `protected` and `private` namespaces associated with programs, packages, classes, and interfaces. User-defined namespace values are constructed by *PackageDirective*s, *NamespaceDefinition*s, and *InterfaceDefinition*s.

1(7) Namespace values are *opened* by *UseDirective*s and *ImportDirective*s.

1(8) A definition is denoted by a *QName*, which consists of the Identifier of the definition qualified by a namespace value. The namespace value is obtained by evaluating the *NamespaceAttribute* of the definition.

1(9) A reference appears as either an *UnqualifiedName* or a *QualifiedName*. The reference is denoted by a *multiname*, which consists of an Identifier qualified by a set of namespace values (thereby representing a set of QNames with the same Identifier). For a reference that appears as an *UnqualifiedName*, the associated set of namespace values contains those namespace values that are open in the scope of that reference. For a reference that appears as a *QualifiedName*, the associated set of namespace values contains the namespace value obtained by evaluating the qualifier of the *QualifiedName*.

1(10) Names that appear in certain syntactic contexts are evaluated during lexical environment building and static verification at compile time. Such names form a subset of *Name*s called *RestrictedName*s. Notably, a *RestrictedName* cannot be a *QualifiedName* whose qualifier is an arbitrary *NameExpression*. *RestrictedName*s are transformed to restricted names by desugaring *ReservedNamespace*s.

1(11) Evaluation of names involves the steps of *expansion* and *resolution*. Expansion maps *UnqualifiedName*s to multinames. In particular, restricted names are transformed to *expanded names*, which are either multinames or identifiers qualified by expanded names. Resolution looks up multinames by mapping them to QNames of definitions in a lexical environment. In particular, expanded names are resolved by successive resolution of their qualifiers to namespace values, resulting in multinames that are finally looked up in the lexical environment.

1.2 Glossary

Listed below are some technical terms that appear in this chapter. We define them informally here to give the reader an intuitive understanding of what they mean and how they relate to each other. Rigorous definitions of these terms appear in later sections as appropriate.

Namespace value. A *namespace value* is a value of an abstract data type **Namespace**. The various constructors of **Namespace** give rise to distinct kinds of namespace values. (See Definition 2.1.)

QName. A *QName* is a name that consists of a namespace value and an identifier. A definition is denoted by a QName. (See Definition 2.2.)

Multiname. A *multiname* is name that consists of a set of namespace values and an identifier. A reference is represented by a multiname. (See Definition 2.3.)

Restricted name. A *restricted name* is a name derived from a *RestrictedName*, by desugaring reserved namespaces and package names to their namespace values. A restricted name is a multiname, an identifier, or an identifier that is qualified by a restricted name. The evaluation of restricted names depends only on the name expansion and name resolution algorithms defined in this chapter. Specifically, it does not depend on general expression evaluation. (See Definition 4.1.)

Expanded name. An *expanded name* is a name that is a multiname or an identifier qualified by an expanded name. Expanded names are intermediate forms that occur during the evaluation of restricted names, after expansion and before resolution. (See Definition 4.2.)

Slot. A *slot* is a record of static information on a definition. Distinct kinds of slots, corresponding to distinct kinds of definitions, are constructed with various constructors of an abstract data type **Slot**. (See Definition 3.5.)

Lexical binding. A *lexical binding* maps a QName to a slot, and is the result of interpreting a definition at compile time. (See Definition 3.5.)

Environment table. An *environment table* is a map from QNames to slots, represented as a set of lexical bindings. (See Definition 3.6.)

Lexical environment. A *lexical environment* is an ordered list of environment tables. Names are looked up in those environment tables in the order in which they appear in the lexical environment. (See Definition 3.7.)

Binding scope. A *binding scope* is a region of program text within which a set of lexical bindings are visible and shadow lexical bindings with the same QNames in outer binding scopes. (See Definition 3.3.)

Defining scope. A *defining scope* is a binding scope within which definitions may appear. Not all binding scopes are defining scopes. (See Definition 3.1.)

Qualifier association. A *qualifier association*, introduced by a use directive, import directive, package directive, class definition, or program, is a mapping of a particular identifier or all identifiers to a namespace value. Qualifier associations are used to map unqualified names to multinames. (See Definition 3.4.)

Opening scope. An *opening scope* is a region of program text within which any unqualified names are implicitly qualified by some set of namespace values, as determined by qualifier associations, to yield multinames. (See Definition 3.2.)

Expansion. *Expansion* is the process of deriving an expanded name from a restricted name, by successively mapping the unqualified names within the restricted name to multinames. (See Definition 4.3.)

Resolution. *Resolution* is the process of successively mapping multinames in the qualifiers of an expanded name to namespace values, yielding a multiname that is finally looked up to yield a namespace value or lexical binding. (See Definition 4.4.)

Lookup. *Lookup* is the process of narrowing a multiname to a QName in the lexical environment and finding the namespace value or lexical binding corresponding to that QName in the lexical environment. (See Definition 4.5.)

Type annotation. A *type annotation* is a restricted name that must denote a *type* that constrains the set of values that may be stored in the slot it annotates.

Static verification. *Static verification* is the compile time phase in which certain semantic properties are checked based on lexical environments. Static verification occurs after lexical environment building and before run time.

2 Names and Namespaces

2.1 Namespace Values

2(1) There are four distinct kinds of namespace values: **public**, **internal**, **static**, and **unique**. These kinds are characterized by how namespace values of those kinds are constructed. For each kind, we define a constructor which returns a namespace value when called with arguments. All constructors other than **unique** return the same namespace value when called with the same arguments. In contrast, the **unique** constructor returns a fresh namespace value every time it is called.

2(2) Namespace values are constructed in the following circumstances:

1. Namespace definitions that appear in the program are explicitly evaluated to namespace values, based on their initializers.
2. Reserved namespaces and package names are implicitly desugared to namespace values in a context-sensitive manner.

2(3) Reserved namespaces usually correspond to different namespace values in different contexts. For example, `private` is desugared to different namespace values inside different classes.

2(4) Some reserved namespaces may correspond to the same namespace values in different contexts. For example, the same **public** namespace values may be explicitly constructed by initializing namespace definitions, and also implicitly constructed by desugaring `public` inside packages and classes.

2(5) Conversely, some reserved namespaces may correspond to different namespace values in the same context. For example, `protected` may be implicitly desugared to both **unique** namespace values and **static** namespace values inside classes.

Definition 2.1 (Namespace value)**.** A *namespace value* nsval is a value of the abstract data type **Namespace**. A namespace value must be constructed in any of the following ways (with the corresponding function and arguments):

1. **public**(str), for some string str
2. **internal**(str), for some string str
3. **static**(nsval::id), for some namespace value nsval and Identifier id
4. **unique**()

2(6) Namespace values (constructed in the above ways) must satisfy the following equality constraints:

1. **public**(str) = **public**(str), for all strings str
2. **internal**(str) = **internal**(str), for all strings str

3. **static**(nsval::id) = **static**(nsval::id), for all namespace values nsval and Identifiers id

4. **unique**() $\neq$ nsval, for all namespace values nsval constructed elsewhere.

5. nsval $\neq$ nsval′ for all namespace values nsval, nsval′ that are not constructed with the same function and arguments.

2(7) The evaluation of any *NamespaceName* or *NamespaceExpression* must result in a namespace value at compile time and runtime, respectively.

2.2 QNames and Multinames

Definition 2.2 (QName)**.** A *QName* qname is of the form nsval::id, where nsval is a namespace value and id is an Identifier.[1]

2(8) Every *Definition* introduces a QName, which consists of the namespace value denoted by its *NamespaceAttribute* and its Identifier. This QName is the definition's key in its environment table.

Definition 2.3 (Multiname)**.** A *multiname* is of the form $\{\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k\}$::id, where $\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k$ are namespace values and id is an Identifier.

2(9) A multiname $\{\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k\}$::id denotes a set of possible QNames that an *UnqualifiedName* id may be interpreted as, where the set of namespace values $\{\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k\}$ denote qualifiers that are associated with id by *UseDirective*s, *ImportDirective*s, *Package Directive*s, *ClassDefinition*s, or the *Program*.

2(10) A QName nsval::id can be viewed as a multiname {nsval}::id. Every reference that appears as a *QualifiedName*, when evaluated, is transformed to a QName, which is then viewed as a multiname for lookup.

2(11) Every multiname, when looked up, must resolve to one and only one QName. If it resolves to none, the reference is not found. If it resolves to more than one, the name is ambiguous.

3 Data Structures for Name Evaluation

3.1 Scopes

3(1) Scopes are identified with specific forms of syntax trees, and can be nested (like syntax trees).

Definition 3.1 (Defining scope)**.** A *defining scope* is a *Program*, or a *ClassBody*, or an *InterfaceBody*, or a *FunctionBody*.

Definition 3.2 (Opening scope)**.** An *opening scope* is a defining scope, or a *PackageDirective*.

Definition 3.3 (Binding scope)**.** A *binding scope* is a defining scope, or a *FunctionExpression* that has an Identifier, or the *Block* of a *CatchClause*.

3.2 Qualifier Associations

Definition 3.4 (Qualifier association)**.** A *qualifier association* is of the form (key, nsval), where key is an Identifier or * and nsval is a namespace value.

3(2) A qualifier association, introduced by a *UseDirective*, *ImportDirective*, *PackageDirective*, *ClassDefinition*, or

[1] A mnemonic for QName may be "qualified name," but it should not to be confused with *QualifiedName*. Whereas a *QualifiedName* may have any *NamespaceExpression* on the left of ::, a QName always has a namespace value on the left of ::.

Program, pairs a namespace value with either a specific Identifier or all Identifiers (*). The namespace value may be used to qualify any *Identifier*s paired with it by the qualifier association.

3(3) Every opening scope is associated with a set of qualifier associations, which are used to qualify any *UnqualifiedName*s that are nested by that opening scope without crossing another opening scope.

3.3 Lexical Bindings and Lexical Environments

Definition 3.5 (Lexical binding)**.** A *lexical binding* is of the form (qname, slot), where qname is a QName and slot is a value of the abstract data type **Slot**, constructed in either of the following ways:

1. **namespace**(nsval), where nsval is a namespace value
2. **class**(final, dynamic, qnameBaseClass, qnamesBaseInterfaces), where final and dynamic are booleans, qnameBaseClass is a QName and qnamesBaseInterfaces is a set of QNames
3. **interface**(qnamesBaseInterfaces), where qnamesBaseInterfaces is a set of QNames
4. **var**(const, type), where const is a boolean, type is a type (defined elsewhere, as a QName, a primitive type, or of the form `Vector.<`T`>` where T is a type)
5. **function**(accessor, final, override, native, argtypes, returntype), where accessor, final, override, and native are booleans, argtypes is an ordered list of types and returntype is a type

Definition 3.6 (Environment table)**.** An *environment table* is a set of lexical bindings.

Definition 3.7 (Lexical environment)**.** A *lexical environment* is an ordered list of environment tables.

3(4) Every defining scope is associated with an environment table, which is used to store lexical bindings for *Definition*s that are nested by that defining scope without crossing another defining scope, unless otherwise specified.

3(5) Every binding scope is associated with a lexical environment, which is used to look up names that are nested by that binding scope without crossing another binding scope.

4 Evaluation of Names

4.1 Statically Evaluable Names

4(1) A *RestrictedName* must be evaluated statically (at compile time), whereas any other *Name* does not need to be evaluated until the expression it appears in is evaluated at runtime.

4(2) *RestrictedName*s appear in the following syntactic contexts (as *NamespaceName*s or *TypeName*s):

Namespace attributes e.g., the name N in the syntax tree N `var x`, N `function f() {...}`, or N `namespace L`.

Namespace initializers e.g., the name N in the syntax tree `namespace L =` N

Inheritance clauses e.g., the name T in the syntax tree `class C extends T`, `class C implements T`, or `interface I extends T`

Type annotations e.g., the name T in the syntax tree `var x:` T or `function f():` T `{...}` or `function (x:` T`) {...}`

4(3) A *RestrictedName* is transformed to a *restricted name*, by desugaring *ReservedNamespace*s and *PackageName*s in a context-sensitive manner as shown in Section 5.4, and dropping parentheses. A restricted name must be evaluated during lexical environment building or static verification as a namespace value or a type definition. Evaluation involves expanding the *UnqualifiedName*s in the restricted name to multinames, yielding an *expanded name* that is then resolved to the lexical binding of a definition.

4.1.1 Restricted Names

Definition 4.1 (Restricted name)**.** A *restricted name* rname is either a multiname, or an Identifier, or a *restricted qualified name* of the form rname′::id where rname′ is a restricted name and id is an Identifier.

4(4) Syntax trees that match *RestrictedName* are transformed to restricted names by desugaring reserved namespaces and package names in a context-sensitive manner, as shown in Section 5.4. In particular, identifiers qualified by package names and identifiers qualified by reserved namespaces are transformed to multinames.

4(5) **Example.** In the following code, restricted names on various lines are shown in corresponding comments on those lines. (Some of the namespace values in these restricted names rely on the contextual expansion of *ReservedNamespace*s such as `public` and `internal`, described in Section 5.4.)

```
package p.q {
    public namespace N;
    public class A {
        N namespace O;              // N is an UnqualifiedName
        N::O var x;                 // N::O is a restricted qualified name
        public::N var y;            // public::N is (desugared to) a multiname
        function f() {
            return public::N::y     // public::N::y is (desugared to) a restricted qualified name
        }
    }
}

class B extends p.q.A {             // p.q.A is (desugared to) a multiname
}
```

4.1.2 Expanded Names

4(6) Evaluation of restricted names involves two steps: expansion and resolution.

1. restricted names are expanded to eliminate unqualified names, as shown in Section 4.2.
2. The resulting expanded names, which are either multinames, or identifiers qualified by expanded names, are resolved to namespace values or lexical bindings, as shown in Section 4.3.

4(7) Expanded names may also arise by partial evaluation of names in other contexts, as described in Section 6.3. In general, expanded names are resolved when the values they denote are needed, which might be during lexical-environment building, static verification, or even dynamic execution. Until then, names remain in the form of expanded names.

Definition 4.2 (Expanded name)**.** An *expanded name* is either a multiname, or an *expanded qualified name* of the form xname::id, where xname is a expanded name and id is an Identifier.

4.2 Expansion of Restricted Names

4(8) A restricted name is expanded by a set of qualifier associations to an expanded name.

Definition 4.3 (Expansion of restricted names)**.** Let XQuals be a set of qualifier associations and rname be a restricted name. Then *expand*(rname, XQuals) is an expanded name that denotes the *expansion* of rname with XQuals, and is defined recursively as follows.

1. If rname is a multiname mname, return mname.
2. If rname is an Identifier id, then:
   (a) Let $\{\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k\}$ be the set of all namespace values nsval such that $(\mathsf{id}, \mathsf{nsval}) \in \mathsf{XQuals}$ or $(\ast, \mathsf{nsval}) \in \mathsf{XQuals}$.
   (b) Return $\{\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k\}$::id.
3. Otherwise, rname is a restricted qualified name of the form rname′::id.
   (a) Let xname = *expand*(rname′, XQuals).
   (b) Return xname::id.

4(9) **Example.** In the following code we show how the restricted names of the previous example are expanded with qualifier associations. (Only the expansions that are significant to the uses are shown.)

```
package p.q {
    public namespace N;
    public class A {
        N namespace O;            // N is expanded to a multiname of the form {...}::N
        N::O var x;               // N::O is expanded to an expanded qualified name of the form {...}::N::O
        public::N var y;          // public::N expands to itself as a multiname
        function f() {
            return public::N::y   // public::N::y expands to itself as an expanded qualified name
        }
    }
}

class B extends p.q.A {           // p.q.A expands to itself as a multiname
}
```

4.3 Resolution of Expanded Names

4(10) An expanded name is resolved by a lexical environment to a namespace value or a lexical binding. Resolution always ends in the lookup of a multiname.

Definition 4.4 (Expanded name resolution)**.** Resolving an expanded name xname in a lexical environment LexEnv, denoted *resolve*(xname, LexEnv), returns a namespace value or lexical binding (or reports an error), as follows:

1. If xname is a multiname mname, return *lookup*(mname, LexEnv).
2. Otherwise xname is an expanded qualified name xname′::id.
   (a) If *resolve*(xname′, LexEnv) returns a namespace value nsval, return *lookup*({nsval}::id, LexEnv).
   (b) Otherwise, report an error.

Definition 4.5 (Multiname lookup)**.** The lookup of a multiname mname in a lexical environment LexEnv, denoted $\mathit{lookup}(\mathsf{mname}, \mathsf{LexEnv})$, returns a namespace value or lexical binding (or reports an error), as follows:

1. If LexEnv is the empty list, report an error.
2. Otherwise, LexEnv contains an environment table $\{(\mathsf{qname}_1, \mathsf{slot}_1), \ldots, (\mathsf{qname}_l, \mathsf{slot}_l)\}$ followed by a lexical environment LexEnv'. Suppose that mname is $\{\mathsf{nsval}_1, \ldots, \mathsf{nsval}_k\}\mathsf{::id}$.
   (a) Let $\mathcal{J}$ be the set of all $j \in \{1, \ldots, l\}$ such that $\mathsf{qname}_j = \mathsf{nsval}_i\mathsf{::id}$ for some $i \in \{1, \ldots, k\}$.
   (b) If $\mathcal{J}$ is the empty set, return $\mathit{lookup}(\mathsf{mname}, \mathsf{LexEnv}')$.
   (c) If $\mathcal{J}$ contains a single j:
      i. If slot_j is of the form **namespace**(nsval), return nsval.
      ii. Otherwise, return $(\mathsf{qname}_j, \mathsf{slot}_j)$.
   (d) Otherwise, report an error.

Example. The following code shows how multiname resolution is sensitive to scopes. Lexical environments, which are set up as described in Section 5.3.2, ensure that slots for definitions in inner scopes shadow slots for definitions in outer scopes.

```
package p {
    public var x;           // package variable
    class A {
        private var x;      // instance variable
        function f(x) {     // local variable
            return x;       // resolves to internal::x (local variable)
        }
    }
}
```

```
package p {
    public var x;           // package variable
    class A {
        private var x;      // instance variable
        function f() {
            return x;       // resolves to private::x (instance variable)
        }
    }
}
```

```
package p {
    public var x;           // package variable
    class A {
        function f() {
            return x;       // resolves to p.x (package variable)
        }
    }
}
```

5 Building Lexical Environments

This section describes the algorithm for building lexical environments. Some implicit assumptions on syntax trees are outlined in Section 5.1, and the algorithm itself is outlined in Section 5.2.

5.1 Preliminaries

5(1) A *NamespaceDefinition* without a *NamespaceInitialization* is treated as if it had a *NamespaceInitialization* that is = nsvalInit, where nsvalInit is a namespace value constructed by **unique**(). A *NamespaceInitialization* that is = str, where str is a string literal, is treated as if it were = **public**(str).

5(2) A *Definition* that does not have a *NamespaceAttribute* is treated as if it had `internal` as its *NamespaceAttribute*, unless the *Definition* is nested by a *InterfaceDefinition* whose Identifier is I, in which case:

1. If the *InterfaceDefinition* is nested by a *PackageDirective* whose *PackageName* is pkg, the *Definition* is treated as if it had **public**("pkg:I") as its *NamespaceAttribute*.
2. If the *InterfaceDefinition* is nested by a *PackageDirective* that does not have a *PackageName*, the *Definition* is treated as if it had **public**("I") as its *NamespaceAttribute*.
3. If the *InterfaceDefinition* is not nested by a *PackageDirective*, the *Definition* is treated as if it had **public**("I") as its *NamespaceAttribute*.

5(3) The base class of a *ClassDefinition* that has a *ClassInheritence* with `extends` *TypeName* is that *TypeName*. The base class of any other *ClassDefinition* is the distinguished *root class*.

5(4) A *TypedBinding* that does not have a *Type* is treated as if it had * as its *Type*.

5.2 Overview of the Algorithm

5(5) Building lexical environments is the process of synthesizing lexical bindings for namespaces, classes, interfaces, variables, and functions by evaluating restricted names.

5(6) lexical environments are built by processing each defining scope in the program in breadth-first order.

5(7) For each defining scope, we process opening scopes, binding scopes, *UseDirective*s and *ImportDirective*s, and *Definition*s in depth-first order. Note that a syntax tree may be an opening scope as well as a binding scope. If so, it must be processed both as an opening scope and as a binding scope upon visiting it.

5(8) Reserved namespaces and package names are desugared as sets of namespace values in a context-sensitive manner, as defined in Section 5.4.

5.3 Processing of Opening Scopes and Binding Scopes

5(9) A defining scope is itself an opening scope, and in addition, may nest other opening scopes without crossing another defining scope.

5(10) Upon visiting an opening scope, a set of qualifier associations is computed by processing the *UseDirective*s and *ImportDirective*s in that opening scope, and including the qualifier associations associated with the immediately enclosing opening scope, as described in Section 5.3.1. This set of qualifier associations is used to expand any restricted names that appear as part of definitions in the opening scope. The resulting expanded names are subsequently resolved when those definitions are visited.

5(11) A defining scope is itself a binding scope, and in addition, may nest other binding scopes without crossing another defining scope.

5(12) Upon visiting a binding scope, environment tables are introduced in a context-sensitive manner to store information on *Definition*s that appear in this binding scope, and a lexical environment is set up to retrieve such information, by chaining the environment tables to the lexical environment of the immediately enclosing binding scope. This processing step is described in Section 5.3.2.

5.3.1 Initialization of Qualifier Associations

5(13) The following are aliases of namespace values that are implicitly constructed upon visiting the opening scope.

1. **Program** is an alias for the namespace value constructed by **unique**() upon visiting the *Program*.
2. $\mathbf{Public}_{\mathsf{pkg}}$ is an alias for the namespace value constructed by **public**("pkg") upon visiting a *PackageDirective* whose *PackageName* is pkg.
3. **Public** is an alias for the namespace value constructed by **public**("") upon visiting a *PackageDirective* without a *PackageName*.
4. $\mathbf{Internal}_{\mathsf{pkg}}$ is an alias for the namespace value constructed by **internal**("pkg") upon visiting a *PackageDirective* whose *PackageName* is pkg.
5. **Internal** is an alias for the namespace value constructed by **internal**("") upon visiting a *PackageDirective* without a *PackageName*.
6. $\mathbf{Private}_{\mathsf{qnameClass}}$ is an alias for the namespace value constructed by **unique**() upon visiting a *ClassDefinition* whose QName is qnameClass.
7. $\mathbf{Protected}_{\mathsf{qnameClass}}$ is an alias for the namespace value constructed by **unique**() upon visiting a *ClassDefinition* whose QName is qnameClass.
8. $\mathbf{StaticProtected}_{\mathsf{qnameClass}}$ is an alias for the namespace value constructed by **static**(qnameClass) upon visiting a *ClassDefinition* whose QName is qnameClass.

Definition 5.1 (Implicit qualifier associations)**.** For every opening scope, there is a set of *implicit qualifier associations* as follows:

1. In a global context not nested by a *PackageDirective*, the set of implicit qualifier associations is $\{(*, \mathbf{Program}), (*, \mathbf{Public})\}$.
2. In a global context nested by a *PackageDirective* whose *PackageName* is pkg, the set of implicit qualifier associations is $\{(*, \mathbf{Public}_{\mathsf{pkg}}), (*, \mathbf{Internal}_{\mathsf{pkg}})\}$.
3. In a global context nested by a *PackageDirective* without a *PackageName*, the set of implicit qualifier associations is $\{(*, \mathbf{Public}), (*, \mathbf{Internal})\}$.
4. In a class context for a *ClassDefinition* whose QName is qname, the set of implicit qualifier associations is $\{(*, \mathbf{Private}_{\mathsf{qnameClass}}), (*, \mathbf{Protected}_{\mathsf{qnameClass}})\} \cup \mathit{StaticQuals}(\mathsf{qnameClass})$, where $\mathit{StaticQuals}(\mathsf{qnameClass})$ is defined (recursively) as follows:
   (a) If qnameClass denotes the root class, then $\mathit{StaticQuals}(\mathsf{qnameClass}) = \{(*, \mathbf{StaticProtected}_{\mathsf{qnameClass}})\}$.
   (b) Otherwise, let qnameBaseClass denote the class that this *ClassDefinition* extends (which is the root class if elided in the *ClassDefinition*). Then $\mathit{StaticQuals}(\mathsf{qnameClass}) = \{(*, \mathbf{StaticProtected}_{\mathsf{qnameClass}})\} \cup \mathit{StaticQuals}(\mathsf{qnameBaseClass})$.
5. In any other opening scope, the set of implicit qualifier associations is $\varnothing$.

5(14) The set of qualifier associations XQuals associated with an opening scope is initialized to be the union of the set of implicit qualifier associations for the opening scope and:

1. $\varnothing$ if the opening scope is the *Program*;
2. otherwise, the set of qualifier associations associated with the immediately enclosing opening scope.

5.3.2 Initialization of Environment Tables and Setup of Lexical Environment

5(15) Upon visiting a binding scope, environment tables are introduced and initialized, and a lexical environment is set up, as follows:

1. Upon visiting a *Program*:
   (a) One environment table is introduced, which is initialized to $\varnothing$. The *Program* is associated with this environment table.
   (b) The lexical environment that the *Program* is associated with is a list containing just that environment table.
2. Upon visiting a *ClassBody* whose class is denoted by qnameClass:
   (a) Let $\mathit{resolve}(\{\mathsf{qnameClass}\}, \mathsf{LexEnv})$ be of the form $\mathbf{class}(\ldots, \mathsf{qnameBaseClass}, \ldots)$ where LexEnv is the lexical environment of the *Program*. Two environment tables, called the *static environment table* of the class and the *instance environment table* of the class, are introduced and initialized as follows:
      i. the static environment table is initialized to $\varnothing$;
      ii. the instance environment table is initialized to the instance environment table of the base class qnameBaseClass, with $\mathbf{Protected}_{\mathsf{qnameBaseClass}}$ renamed to $\mathbf{Protected}_{\mathsf{qnameClass}}$.

   The *ClassBody* is associated with the static environment table.
   (b) The lexical environment that the *ClassBody* is associated with is a list containing the instance environment table, followed by the environment tables in $\mathit{StaticEnvs}(\mathsf{qnameClass})$, followed by the environment tables in the lexical environment of the *Program*, where $\mathit{StaticEnvs}(\mathsf{qnameClass})$ is defined (recursively) as follows:
      i. If qnameClass denotes the root class, then $\mathit{StaticEnvs}(\mathsf{qnameClass})$ is a list containing just the static environment table for the root class.
      ii. Otherwise, $\mathit{StaticEnvs}(\mathsf{qnameClass})$ is a list containing the static environment table for this class, followed by $\mathit{StaticEnvs}(\mathsf{qnameBaseClass})$.
3. Upon visiting an *InterfaceBody*:
   (a) One environment table is introduced, which is initialized to $\varnothing$. The *InterfaceBody* is associated with this environment table.
   (b) The lexical environment that the *InterfaceBody* is associated with is the lexical environment of the *Program*.
4. Upon visiting a *FunctionBody*:
   (a) Let the *FunctionSignature* of the *FunctionExpression* or the *FunctionDefinition* that the *FunctionBody* belongs to have *Parameter*s whose Identifiers are $\mathsf{id}_1, \ldots, \mathsf{id}_k$. Let $\mathsf{type}_1, \ldots, \mathsf{type}_k$ be the types of those *Parameter*s (which are evaluated later, as shown in Section 6.1). One environment table is introduced, which is initialized to $\{(\mathsf{nsval}::\mathsf{id}_1, \mathbf{var}(\text{false}, \mathsf{type}_1)), \ldots, (\mathsf{nsval}::\mathsf{id}_k, \mathbf{var}(\text{false}, \mathsf{type}_k))\}$,

where nsval is whatever namespace value `internal` is expanded to in that context (see Section 5.4). The *FunctionBody* is associated with this environment table.

   (b) The lexical environment that the *FunctionBody* is associated with is a list containing that environment table followed by the environment tables in the lexical environment of the immediately enclosing binding scope.

5. Upon visiting a *FunctionExpression* that has an Identifier:

   (a) Let that Identifier be id. Let argtypes be the ordered list of argument types of that *FunctionExpression*, and let returntype be the return type of that *FunctionExpression* (which are evaluated later, as shown in Section 6.1). One (fixed) environment table is introduced, which is initialized to $\{(\mathsf{nsval}::\mathsf{id}, \textbf{function}(\text{false}, \text{false}, \text{false}, \text{false}, \mathsf{argtypes}, \mathsf{returntype}))\}$, where nsval is whatever namespace value `internal` is expanded to in that context (see Section 5.4).

   (b) The lexical environment that this *FunctionExpression* is associated with is a list containing that environment table followed by the environment tables in the lexical environment of the immediately enclosing binding scope.

6. Upon visiting a *Block* of a *CatchClause*:

   (a) Let the *CatchClause* it belongs to have a *TypedBinding* with Identifier id. Let type be the type of that *TypedBinding* (which is evaluated later, as shown in Section 6.1). One (fixed) environment table is introduced, which is initialized to $\{(\mathsf{nsval}::\mathsf{id}, \textbf{var}(\text{false}, \mathsf{type}))\}$, where nsval is whatever namespace value `internal` is expanded to in that context (see Section 5.4).

   (b) The lexical environment that this binding scope is associated with is a list containing that environment table followed by the environment tables in the lexical environment of the immediately enclosing binding scope.

5.4 Desugaring of Reserved Namespaces and Package Names

5(16) *ReservedNamespace*s, which appear either as *NamespaceAttribute*s or in *QualifiedName*s, are desugared based on the opening scope in which they appear, as follows:

1. In a global context that is not nested by a *PackageDirective*:

   (a) `internal` is expanded to **Program**.

2. In a global context nested by a *PackageDirective* whose *PackageName* is pkg:

   (a) `public` is expanded to $\textbf{Public}_{\mathsf{pkg}}$.

   (b) `internal` is expanded to $\textbf{Internal}_{\mathsf{pkg}}$.

3. In a global context nested by a *PackageDirective* without a *PackageName*:

   (a) `public` is expanded to **Public**.

   (b) `internal` is expanded to **Internal**.

4. In a class context for a *ClassDefinition* whose QName is qnameClass:

   (a) `public` is expanded to **Public**.

   (b) `internal` is expanded to whatever it would be expanded to in the global context outside the *ClassDefinition*.

   (c) `private` is expanded to $\textbf{Private}_{\mathsf{qnameClass}}$.

(d) A *NamespaceAttribute* that is `protected` is expanded to $\mathbf{StaticProtected}_{\mathsf{qnameClass}}$ if the definition it marks is a *NamespaceDefinition* or is marked `static`, and to $\mathbf{Protected}_{\mathsf{qnameClass}}$ otherwise.

(e) A *QualifiedName* of the form `protected::id` is expanded to the multiname $\{\mathbf{StaticProtected}_{\mathsf{qnameClass}}, \mathbf{Protected}_{\mathsf{qnameClass}}\}$`::id`.

5. In any other opening scope, a *NamespaceAttribute* or a *QualifiedName* in which a *ReservedNamespace* appears is expanded to whatever it would be expanded to in the immediately enclosing opening scope.

5(17) Any reference of the form `pkg.id`, where id is an Identifier and pkg is a *PackageName* that appears earlier than that reference in the program, is replaced by the reference $\mathbf{Public}_{\mathsf{pkg}}$`::id`.

5(18) **Example.** The following code shows the translation of reserved namespaces to their corresponding namespace values in various syntactic contexts.

```
package P {
    public class A {
        private var a;              // Private_A::a, where A is Public_P::A
        protected var b;            // Protected_A::b, where A is Public_P::A
        protected static var b;     // StaticProtected_A::b, where A is Public_P::A
        var x;                      // Internal_P::x
        public var y;               // Public::y
        function f(z) {             // Internal_P::z
            private::a              // Private_A::a, where A is Public_P::A
            protected::b            // {StaticProtected_A, Protected_A}::b, where A is Public_P::A
            internal::x;            // Internal_P::x
            public::y;              // Public::y
            P.z;                    // Public_P::z
        }
    }

    var x;                          // Internal_P::x
    public var z;                   // Public_P::z
    function f(z) {                 // Internal_P::z
        internal::x;                // Internal_P::x
        internal::z;                // Internal_P::z
        public::z;                  // Public_P::z
    }
}

var c;                              // Program::c
internal::c;                        // Program::c
```

5.5 Processing of Imports and Uses of Namespaces, and Definitions of Names

5(19) In this section, we specify how the set of qualifier associations associated with an opening scope is built by considering the *UseDirective*s and *ImportDirective*s in the opening scope in the order in which they appear. As the set of qualifier associations is built, it can be used to expand restricted names that appear in that opening scope. Furthermore, we specify how the lexical environment associated with a binding scope is built by considering the *Definition*s in the binding scope in the order in which they appear. As the lexical environment is built, it can be used to resolve expanded names that appear in that binding scope. By the end of this section, for every opening scope in the *Program*, we can finalize the set of qualifier associations initialized in Section 5.3.1, which can then be used to qualify *UnqualifiedName*s in that opening scope. Furthermore, for every binding scope in the *Program*, we can finalize the lexical environment initialized in Section 5.3.2, which can then be used to look up multinames in that binding scope.

5(20) For any *ImportDirective* or *UseDirective*, let the immediately enclosing opening scope be associated with the set of qualifier associations XQuals. The *ImportDirective* or *UseDirective* is processed in the following two steps, as elaborated below:

1. The namespace name that appears in the directive is evaluated.
2. Based on the evaluation of the namespace name, a qualifier association is synthesized and added to XQuals.

5(21) For any *Definition*, let the immediately enclosing binding scope be associated with the lexical environment LexEnv, and the immediately enclosing opening scope be associated with the set of qualifier associations XQuals. The *Definition* is processed in the following two steps, as elaborated below:

1. Namespace names (namespace attributes or namespace initializers) or inherited type names (type names for inherited classes and interfaces) that appear in the definition are evaluated.
2. Based on the evaluations of the namespace names and inherited type names, a lexical binding for the Identifier in the definition is synthesized and added to an environment table in LexEnv, unless doing so makes the definition *duplicate* or *ambiguous*. Roughly, a definition is considered duplicate if there is another definition in the current binding scope with the same Identifier and the same namespace value. A definition is considered ambiguous if there is another definition in the current binding scope with the same Identifier, and the namespace values of both definitions are open in the current opening scope.

5.5.1 Import Directives

5(22) An *ImportDirective* of the form `import pkg.key`, where key is an Identifier or `*` and pkg is a *PackageName*, is processed as follows:

1. The qualifier association $(\mathsf{key}, \mathbf{Public}_{\mathsf{pkg}})$ is added to XQuals.

5(23) **Example.** In the following examples we show the correspondence between namespace imports and qualifier associations. Qualifier associations are built and unqualified names are expanded with matching associations in a single phase.

```
package P {
    public class A {
    }
}

package Q {             // initial qualifier associations: {(*, Program), (*, Public), (*, Public_Q), (*, Internal_Q)}
    import P.A;         // the set of qualifier associations becomes {..., (A, Public_P)}
    class B extends A   // the UnqualifiedName A is expanded to the multiname {..., Public_P}::A
    {
    }
}
```

5.5.2 Use Directives

5(24) A *UseDirective* of the form `use namespace rnameUse`, where rnameUse is a restricted name, is processed as follows:

1. If *resolve*(*expand*(rnameUse, XQuals), LexEnv) returns a namespace value nsvalUse, the qualifier association (*, nsvalUse) is added to XQuals.

5(25) **Example.** In the following example we show the correspondence between namespace uses and qualifier associations. Qualifier associations are built and unqualified names are expanded with matching associations in a single phase.

```
class A {
    namespace M = "M";
    M namespace N = "N";      // M expands to the multiname {...}::M
    use namespace M;          // the set of qualifier associations becomes {..., (*, public("M"))}
    N namespace P = "P";      // N expands to the multiname {..., public("M")}::N
    use namespace N;          // the set of qualifier associations becomes {..., (*, public("M")), (*, public("N"))}
    P namespace Q;            // P expands to the multiname {..., public("M"), public("N")}::P
}
```

5.5.3 Namespace Definitions

5(26) A *NamespaceDefinition* with Identifier id is processed as follows:

1. The namespace attribute is evaluated to a namespace value nsvalAttr as follows:
   (a) If the namespace attribute is already a namespace value, let nsvalAttr be that namespace value.
   (b) Otherwise, the namespace attribute is a restricted name rnameAttr.
      i. If *resolve*(*expand*(rnameAttr, XQuals), LexEnv) returns a namespace value, let nsvalAttr be that namespace value.
      ii. Otherwise, report an error.
2. Similarly, the namespace initializer is evaluated to a namespace value nsvalInit as follows:
   (a) If the namespace initializer is already a namespace value, let nsvalInit be that namespace value.
   (b) Otherwise, the namespace initializer is a restricted name rnameInit.
      i. If *resolve*(*expand*(rnameInit, XQuals), LexEnv) returns a namespace value, let nsvalInit be that namespace value.
      ii. Otherwise, report an error.
3. Finally, a lexical binding is synthesized as follows:
   (a) If there is already a lexical binding of the form (nsvalAttr::id, ...) in the environment table that the defining scope is associated with, an error is reported.
   (b) Otherwise:
      i. If $(\mathsf{id}, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ and there is some namespace value $\mathsf{nsval} \neq \mathsf{nsvalAttr}$ such that $(\mathsf{id}, \mathsf{nsval}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsval}) \in \mathsf{XQuals}$, and there is already a lexical binding of the form (nsval::id, ...) in the environment table that the defining scope is associated with, then an error is reported.
      ii. Otherwise, (nsvalAttr::id, **namespace**(nsvalInit)) is added to the environment table that the defining scope is associated with.

5(27) **Example.** In the following example we show how namespace definitions are processed.

```
package p {
    public namespace M = "M";          // (Public_p::M, namespace(public("M")))
    class A {
        private namespace N = "N";     // (Private_A::N, namespace(public("N"))), where A is Internal_p::A
        N namespace O = M;             // (public("N")::O, namespace(public("M")))
    }
}
```

5.5.4 Interface Definitions

5(28) An *InterfaceDefinition* with Identifier id and base interfaces $\mathsf{rnameBaseInterface}_1, \ldots, \mathsf{rnameBaseInterface}_k$ is processed as follows.

1. The namespace attribute is evaluated to a namespace value nsvalAttr as follows:
   (a) If the namespace attribute is already a namespace value, let nsvalAttr be that namespace value.
   (b) Otherwise, the namespace attribute is a restricted name rnameAttr.
      i. If $\mathit{resolve}(\mathit{expand}(\mathsf{rnameAttr}, \mathsf{XQuals}), \mathsf{LexEnv})$ returns a namespace value, let nsvalAttr be that namespace value.
      ii. Otherwise, report an error.
2. The base interfaces $\mathsf{rnameBaseInterface}_1, \ldots, \mathsf{rnameBaseInterface}_k$ are respectively evaluated to QNames $\mathsf{qnameBaseInterface}_1, \ldots, \mathsf{qnameBaseInterface}_k$ as follows.
   (a) If $\mathit{resolve}(\mathit{expand}(\mathsf{rnameBaseInterface}_i, \mathsf{XQuals}), \mathsf{LexEnv})$ returns a lexical binding of the form $(\mathsf{qname}_i, \mathbf{interface}(\ldots))$ for each $i \in \{1, \ldots, k\}$:
      i. If $\mathsf{qname}_1, \ldots, \mathsf{qname}_k$ are not distinct, report an error.
      ii. Otherwise, for each $i \in \{1, \ldots, k\}$, let $\mathsf{qnameBaseInterface}_i$ be qname_i.
   (b) Otherwise, report an error.
3. Finally, a lexical binding is synthesized as follows:
   (a) If there is already a lexical binding of the form $(\mathsf{nsvalAttr}::\mathsf{id}, \ldots)$ in the environment table that the defining scope is associated with, an error is reported.
   (b) Otherwise:
      i. If $(\mathsf{id}, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ and there is some namespace value $\mathsf{nsval} \neq \mathsf{nsvalAttr}$ such that $(\mathsf{id}, \mathsf{nsval}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsval}) \in \mathsf{XQuals}$, and there is already a lexical binding of the form $(\mathsf{nsval}::\mathsf{id}, \ldots)$ in the environment table that the defining scope is associated with, then an error is reported.
      ii. Otherwise, the lexical binding $(\mathsf{nsvalAttr}::\mathsf{id}, \mathbf{interface}(\mathsf{qnamesBaseInterfaces}))$ is added to the environment table that the defining scope is associated with, where $\mathsf{qnamesBaseInterfaces} = \{\mathsf{qnameBaseInterface}_1, \ldots, \mathsf{qnameBaseInterface}_k\}$.

5(29) **Example.** In the following example we show how interface definitions are processed.

```
package {
    public namespace N;
    interface I { }                          // (Internal::I, interface({}))
    public interface J extends I { }         // (Public::J, interface({Internal::I}))
    // public interface J extends I,N { }    // error! N is not an interface
}
```

```
    // public interface J extends I,I { }      // error! interfaces are not distinct
    // public interface N { }                  // error! duplicate definition
    // public interface I { }                  // error! ambiguous definition
}
```

5.5.5 Class Definitions

5(30) A *ClassDefinition* with Identifier id, base class qnameBaseClass, and base interfaces $\mathsf{rnameBaseInterface}_1, \ldots, \mathsf{rnameBaseInterface}_k$ is processed as follows.

1. The namespace attribute is evaluated to a namespace value nsvalAttr as follows:
   (a) If the namespace attribute is already a namespace value, let nsvalAttr be that namespace value.
   (b) Otherwise, the namespace attribute is a restricted name rnameAttr.
      i. If $\mathit{resolve}(\mathit{expand}(\mathsf{rnameAttr}, \mathsf{XQuals}), \mathsf{LexEnv})$ returns a namespace value, let nsvalAttr be that namespace value.
      ii. Otherwise, report an error.
2. The base class qnameBaseClass is evaluated to a QName qnameBaseClass as follows.
   (a) If $\mathit{resolve}(\mathit{expand}(\mathsf{qnameBaseClass}, \mathsf{XQuals}), \mathsf{LexEnv})$ does not return a lexical binding of the form $(\mathsf{qnameBaseClass}, \mathbf{qnameClass}(\ldots))$, report an error.
   (b) Otherwise, return qnameBaseClass.
3. The base interfaces $\mathsf{rnameBaseInterface}_1, \ldots, \mathsf{rnameBaseInterface}_k$ are evaluated respectively to QNames $\mathsf{qnameBaseInterface}_1, \ldots, \mathsf{qnameBaseInterface}_k$ as follows.
   (a) If $\mathit{resolve}(\mathit{expand}(\mathsf{rnameBaseInterface}_i, \mathsf{XQuals}), \mathsf{LexEnv})$ returns a lexical binding of the form $(\mathsf{qname}_i, \mathbf{interface}(\ldots))$ for each $i \in \{1, \ldots, k\}$:
      i. If $\mathsf{qname}_1, \ldots, \mathsf{qname}_k$ are not distinct, report an error.
      ii. Otherwise, for each $i \in \{1, \ldots, k\}$, let $\mathsf{qnameBaseInterface}_i$ be qname_i.
   (b) Otherwise, report an error.
4. Finally, a lexical binding is synthesized as follows:
   (a) If there is already a lexical binding of the form $(\mathsf{nsvalAttr}::\mathsf{id}, \ldots)$ in the environment table that the defining scope is associated with, an error is reported.
   (b) Otherwise:
      i. If $(\mathsf{id}, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ and there is some namespace value $\mathsf{nsval} \neq \mathsf{nsval}$ such that $(\mathsf{id}, \mathsf{nsval}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsval}) \in \mathsf{XQuals}$, and there is already a lexical binding of the form $(\mathsf{nsval}::\mathsf{id}, \ldots)$ in the environment table that the defining scope is associated with, then an error is reported.
      ii. Otherwise, let dynamic be true if the *ClassDefinition* is marked `dynamic`, false otherwise, and let final be true if the *ClassDefinition* is marked `final`, false otherwise. The lexical binding $(\mathsf{nsvalAttr}::\mathsf{id}, \mathbf{class}(\mathsf{final}, \mathsf{dynamic}, \mathsf{qnameBaseClass}, \mathsf{qnamesBaseInterfaces}))$ is added to the environment table that the defining scope is associated with, where $\mathsf{qnamesBaseInterfaces} = \{\mathsf{qnameBaseInterface}_1, \ldots, \mathsf{qnameBaseInterface}_k\}$.

5(31) **Example.** In the following example we show how class definitions are processed.

```
package {
    public interface C { }
}

package p {
    public dynamic class B { }                 // (Public_p::B, class(false, true, Public::Object, {}))
    final class A extends B implements C { } // (Internal_p::A, class(true, false, Public_p::B, {Public::C}))
}
```

5.5.6 Function Definitions

TODO: We need to talk about lexical bindings for `arguments`.

5(32) A *FunctionDefinition* with Identifier id that is marked `static` if it is in a class context is processed as follows.

1. The namespace attribute is evaluated to a namespace value nsvalAttr as follows:
   (a) If the namespace attribute is already a namespace value, let nsvalAttr be that namespace value.
   (b) Otherwise, the namespace attribute is a restricted name rnameAttr.
      i. If $resolve(expand(\mathsf{rnameAttr}, \mathsf{XQuals}), \mathsf{LexEnv})$ returns a namespace value, let nsvalAttr be that namespace value.
      ii. Otherwise, report an error.
2. Finally, a lexical binding is synthesized as follows:
   (a) If there is already a lexical binding of the form $(\mathsf{nsvalAttr}::\mathsf{id}, \mathsf{slot})$ in the environment table that the defining scope is associated with, then an error is reported unless slot is of the form **function**(true, ...) and the *FunctionDefinition* has an *AccessorKind*.
   (b) Otherwise:
      i. If $(\mathsf{id}, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ and there is some namespace value $\mathsf{nsval} \neq \mathsf{nsval}$ such that $(\mathsf{id}, \mathsf{nsval}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsval}) \in \mathsf{XQuals}$, and there is already a lexical binding of the form $(\mathsf{nsval}::\mathsf{id}, \mathsf{slot}'')$ in the environment table that the defining scope is associated with, and slot'' is of the form **qnameFunction**(...) or **namespace**(...) if the *Definition* is in a class context, then an error is reported.
      ii. Otherwise, let accessor be true if the *FunctionDefinition* has an *AccessorKind*, false otherwise, let final be true if the *FunctionDefinition* is marked `final`, false otherwise, let override be true if the *FunctionDefinition* is marked `override`, false otherwise, and let native be true if the *FunctionDefinition* is marked `native`, false otherwise. Let the ordered list of argument types of the *FunctionDefinition* be argtypes, and the return type of the *FunctionDefinition* be returntype (evaluated later, as shown in Section 6.1.) The lexical binding $(\mathsf{nsvalAttr}::\mathsf{id}, \mathbf{function}(\mathsf{accessor}, \mathsf{final}, \mathsf{override}, \mathsf{native}, \mathsf{argtypes}, \mathsf{returntype}))$ is added to the environment table that the defining scope is associated with.

5(33) A *FunctionDefinition* with Identifier id that is in a class context but not marked `static` is processed as follows.

1. The namespace attribute is evaluated to a namespace value nsvalAttr as follows:
   (a) If the namespace attribute is already a namespace value, let nsvalAttr be that namespace value.
   (b) Otherwise, the namespace attribute is a restricted name rnameAttr.

i. If *resolve*(*expand*(rnameAttr, XQuals), LexEnv) returns a namespace value, let nsvalAttr be that namespace value.

ii. Otherwise, report an error.

2. Finally, a lexical binding is synthesized as follows:

(a) If there is already a lexical binding of the form (nsvalAttr::id, slot) in the environment table that the defining scope is associated with, then an error is reported unless slot is of the form **function**(true, ...) and the *FunctionDefinition* has an *AccessorKind*.

(b) Otherwise, let accessor be true if the *FunctionDefinition* has an *AccessorKind*, false otherwise, let final be true if the *FunctionDefinition* is marked `final`, false otherwise, let override be true if the *FunctionDefinition* is marked `override`, false otherwise, and let native be true if the *FunctionDefinition* is marked `native`, false otherwise. Let the ordered list of argument types of the *FunctionDefinition* be argtypes, and the return type of the *FunctionDefinition* be returntype (evaluated later, as shown in Section 6.1.) The lexical binding (nsvalAttr::id, **function**(accessor, final, override, native, argtypes, returntype)) is added to the instance environment table introduced by the enclosing *ClassBody*.

5(34) **Example.** In the following example we show how function definitions are processed.

```
package {
    public function f() { }                 // (Public::f, function(false, false, false, false, (), *))
    class A {
        native set function g(x:int);       // (Internal::g, function(true, false, false, true, [int], *))
        get function g():int;               // (Internal::g, function(true, false, false, false, [], int))
        function h(x):void { }              // (Internal::h, function(false, false, false, false, [*], void))
    }
    class B extends A {
        final override function h(x):void { }  // (Internal::h, function(false, true, true, false, [*], void))
    }
}
```

5.5.7 Variable Definitions

5(35) A *VariableDefinition* with Identifier id that is marked `static` if it is in a class context is processed as follows.

1. The namespace attribute is evaluated to a namespace value nsvalAttr as follows:

(a) If the namespace attribute is already a namespace value, let nsvalAttr be that namespace value.

(b) Otherwise, the namespace attribute is a restricted name rnameAttr.

i. If *resolve*(*expand*(rnameAttr, XQuals), LexEnv) returns a namespace value, let nsvalAttr be that namespace value.

ii. Otherwise, report an error.

2. Finally, a lexical binding is synthesized as follows:

(a) If there is already a lexical binding of the form (nsvalAttr::id, slot) in the environment table that the defining scope is associated, then an error is reported unless slot is of the form **var**(...) and the *VariableDefinition* is not in a global context nested by a *PackageDirective*, or a class context, or an interface context.

(b) Otherwise:

i. If $(\mathsf{id}, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsvalAttr}) \in \mathsf{XQuals}$ and there is some namespace value $\mathsf{nsval} \neq \mathsf{nsval}$ such that $(\mathsf{id}, \mathsf{nsval}) \in \mathsf{XQuals}$ or $(*, \mathsf{nsval}) \in \mathsf{XQuals}$, and there is already a lexical binding of the form $(\mathsf{nsval}::\mathsf{id}, \mathsf{slot}'')$ in the environment table that the defining scope is associated with, and slot'' is of the form $\mathbf{qnameVar}(\ldots)$ or $\mathbf{namespace}(\ldots)$ if the *Definition* is in a class context, then an error is reported.

ii. Otherwise, let const be true if the *VariableDefinition* is of kind `const`, false otherwise. Let the type of the *VariableDefinition* be type (evaluated later, as shown in Section 6.1.) The lexical binding $(\mathsf{nsvalAttr}::\mathsf{id}, \mathbf{var}(\mathsf{const}, \mathsf{type}))$ is added to the environment table that the defining scope is associated with.

5(36) A *VariableDefinition* with Identifier id that is in a class context but not marked `static` is processed as follows.

1. The namespace attribute is evaluated to a namespace value $\mathsf{nsvalAttr}$ as follows:

   (a) If the namespace attribute is already a namespace value, let $\mathsf{nsvalAttr}$ be that namespace value.

   (b) Otherwise, the namespace attribute is a restricted name $\mathsf{rnameAttr}$.

   i. If $\mathit{resolve}(\mathit{expand}(\mathsf{rnameAttr}, \mathsf{XQuals}), \mathsf{LexEnv})$ returns a namespace value, let $\mathsf{nsvalAttr}$ be that namespace value.

   ii. Otherwise, report an error.

2. Finally, a lexical binding is synthesized as follows:

   (a) If there is already a lexical binding of the form $(\mathsf{nsvalAttr}::\mathsf{id}, \ldots)$ in the environment table that the defining scope is associated with, an error is reported.

   (b) Otherwise, let const be true if the *VariableDefinition* is of kind `const`, false otherwise. Let the type of the *VariableDefinition* be type (evaluated later, as shown in Section 6.1.) The lexical binding $(\mathsf{nsvalAttr}::\mathsf{id}, \mathbf{var}(\mathsf{const}, \mathsf{type}))$ is added to the instance environment table introduced by the enclosing *ClassBody*.

5(37) **Example.** In the following example we show how variable definitions are processed.

```
var x;                                // (Program::x, var(false, *))
var x = 0;                            // (Program::x, var(false, *))
package {
    var y : A;                        // (Internal::y, var(false, Internal::A))
    // var y : A = new A;             // error! duplicate definition
    class A {
        public var z;                 // (Public::z, var(false, *))
        // public var z;              // error! duplicate definition
        // internal var z;            // error! ambiguous definition
        public static const var z = 1;  // (Public::z, var(true, *))
        namespace N;
        use namespace N;
        // N namespace z              // error! ambiguous definition
        N static function z(cond) {
            if (cond) var w = 2;      // (Internal::w, var(false, *))
            else var w = 3;           // (Internal::w, var(false, *))
        }
    }
}
```

6 Preparing for Static Verification and Dynamic Execution

6.1 Evaluation of Type Annotations

6(1) Any type annotation (which may appear as the type of a variable definition or an argument type of a function or the return type of a function definition) must be a primitive type, or a restricted name, or `Vector` of a type annotation. Any restricted name rname in a type annotation is evaluated to $\mathit{resolve}(\mathit{expand}(\mathsf{rname}, \mathsf{XQuals}), \mathsf{LexEnv})$, where XQuals is the set of qualifier associations of the immediately enclosing opening scope, and LexEnv is the lexical environment of the immediately enclosing binding scope.

6(2) **Example.** This code shows how type annotations are evaluated after all definitions in the program have been processed. In particular they can denote class and interface definitions that occur later in the program.

```
package {
  import P.T;
  var x : A;
  class A {
    function f(x: P.T) { }
  }
}

package P {
  public interface T { }
}
```

6.2 Detection of Inconsistent Definitions

6(3) If an environment table has more than one lexical binding corresponding to *VariableDefinition*s for the same QName qnameVar, say $(\mathsf{qnameVar}, \mathsf{slot})$ and $(\mathsf{qnameVar}, \mathsf{slot}_1), \ldots, (\mathsf{qnameVar}, \mathsf{slot}_k)$ where slot and $\mathsf{slot}_1, \ldots, \mathsf{slot}_k$ are each of the form $\mathbf{var}(\ldots)$, then:

1. Unless $\mathsf{slot} = \mathsf{slot}_1 = \ldots = \mathsf{slot}_k$, report an error.
2. Remove the lexical bindings $(\mathsf{qnameVar}, \mathsf{slot}_1), \ldots, (\mathsf{qnameVar}, \mathsf{slot}_k)$ from the environment table.

6(4) If an environment table has more than one lexical binding corresponding to *FunctionDefinition*s for the same QName qnameFunction, then report an error unless there are exactly two such lexical bindings, one of the form $(\mathsf{qnameFunction}, \mathbf{function}(\mathrm{true}, \ldots, [], \mathsf{type}))$ and the other of the form $(\mathsf{qnameFunction}, \mathbf{function}(\mathrm{true}, \ldots, [\mathsf{type}], \ldots))$.

6.3 Partial Evaluation of Other Names

6(5) Any *UnqualifiedName* id whose immediately enclosing opening scope is associated with the set of qualifier associations XQuals, is replaced by the multiname $\mathit{expand}(\mathsf{id}, \mathsf{XQuals})$.

6(6) Any expanded name xname whose immediately enclosing binding scope is associated with the lexical environment LexEnv, and $\mathit{resolve}(\mathsf{xname}, \mathsf{LexEnv}))$ returns a namespace value nsval, is replaced by nsval.