Basic Declarations & Definitions

Basic Declarations and Definitions

```
Dcl         ::=  ‘val’ ValDcl
              |  ‘var’ VarDcl
              |  ‘def’ FunDcl
              |  ‘type’ {nl} TypeDcl
PatVarDef   ::=  ‘val’ PatDef
              |  ‘var’ VarDef
Def         ::=  PatVarDef
              |  ‘def’ FunDef
              |  ‘type’ {nl} TypeDef
              |  TmplDef
```

A *declaration* introduces names and assigns them types. It can form part of a class definition or of a refinement in a compound type.

A *definition* introduces names that denote terms or types. It can form part of an object or class definition or it can be local to a block. Both declarations and definitions produce *bindings* that associate type names with type definitions or bounds, and that associate term names with types.

The scope of a name introduced by a declaration or definition is the whole statement sequence containing the binding. However, there is a restriction on forward references in blocks: In a statement sequence $s_1 \ldots s_n$ making up a block, if a simple name in s_i refers to an entity defined by s_j where $j \geq i$, then for all s_k between and including s_i and s_j,

- s_k cannot be a variable definition.
- If s_k is a value definition, it must be lazy.

Value Declarations and Definitions

```
Dcl          ::=  ‘val’ ValDcl
ValDcl       ::=  ids ‘:’ Type
PatVarDef    ::=  ‘val’ PatDef
PatDef       ::=  Pattern2 {‘,’ Pattern2} [‘:’ Type] ‘=’ Expr
ids          ::=  id {‘,’ id}
```

A value declaration `val $x$: $T$` introduces x as a name of a value of type T.

A value definition `val $x$: $T$ = $e$` defines x as a name of the value that results from the evaluation of e. If the value definition is not recursive, the type T may be omitted, in which case the packed type of expression e is assumed. If a type T is given, then e is expected to conform to it.

Evaluation of the value definition implies evaluation of its right-hand side e, unless it has the modifier `lazy`. The effect of the value definition is to bind x to the value of e converted to type T. A `lazy` value definition evaluates its right hand side e the first time the value is accessed.

A *constant value definition* is of the form

```
final val x = e
```

where `e` is a constant expression. The `final` modifier must be present and no type annotation may be given. References to the constant value `x` are themselves treated as constant expressions; in the generated code they are replaced by the definition's right-hand side `e`.

Value definitions can alternatively have a pattern as left-hand side. If p is some pattern other than a simple name or a name followed by a colon and a type, then the value definition `val $p$ = $e$` is expanded as follows:

1. If the pattern p has bound variables $x_1, \ldots, x_n$, where $n > 1$:

```
val $\$ x$ = $e$ match {case $p$ => ($x_1 , \ldots , x_n$)}
val $x_1$ = $\$ x$._1
$\ldots$
val $x_n$ = $\$ x$._n
```

Here, $\$x$ is a fresh name.

2. If p has a unique bound variable x:

```
val $x$ = $e$ match { case $p$ => $x$ }
```

3. If p has no bound variables:

```
$e$ match { case $p$ => ()}
```

Example

The following are examples of value definitions

```
val pi = 3.1415
val pi: Double = 3.1415   // equivalent to first definition
val Some(x) = f()         // a pattern definition
val x :: xs = mylist      // an infix pattern definition
```

The last two definitions have the following expansions.

```
val x = f() match { case Some(x) => x }

val x$\$$ = mylist match { case x :: xs => (x, xs) }
val x = x$\$$._1
val xs = x$\$$._2
```

The name of any declared or defined value may not end in `_=`.

A value declaration `val $x_1 , \ldots , x_n$: $T$` is a shorthand for the sequence of value declarations `val $x_1$: $T$; ...; val $x_n$: $T$`. A value definition `val $p_1 , \ldots , p_n$ = $e$` is a shorthand for the sequence of value definitions `val $p_1$ = $e$; ...; val $p_n$ = $e$`. A value definition `val $p_1 , \ldots , p_n: T$ = $e$` is a shorthand for the sequence of value definitions `val $p_1: T$ = $e$; ...; val $p_n: T$ = $e$`.

Variable Declarations and Definitions

```
Dcl            ::=  ‘var’ VarDcl
PatVarDef      ::=  ‘var’ VarDef
VarDcl         ::=  ids ‘:’ Type
VarDef         ::=  PatDef
                 |  ids ‘:’ Type ‘=’ ‘_’
```

A variable declaration `var $x$: $T$` is equivalent to the declarations of both a *getter function x and* a *setter function* `$x$_=`:

```
def $x$: $T$
def $x$_= ($y$: $T$): Unit
```

An implementation of a class may *define* a declared variable using a variable definition, or by defining the corresponding setter and getter methods.

A variable definition `var $x$: $T$ = $e$` introduces a mutable variable with type T and initial value as given by the expression e. The type T can be omitted, in which case the type of e is assumed. If T is given, then e is expected to conform to it.

Variable definitions can alternatively have a pattern as left-hand side. A variable definition `var $p$ = $e$` where p is a pattern other than a simple name or a name followed by a colon and a type is expanded in the same way as a value definition `val $p$ = $e$`, except that the free names in p are introduced as mutable variables, not values.

The name of any declared or defined variable may not end in `_=`.

A variable definition `var $x$: $T$ = _` can appear only as a member of a template. It introduces a mutable field with type T and a default initial value. The default value depends on the type T as follows:

default	type T
`0`	`Int` or one of its subrange types
`0L`	`Long`

default	type T
`0.0f`	`Float`
`0.0d`	`Double`
`false`	`Boolean`
`()`	`Unit`
`null`	all other types

When they occur as members of a template, both forms of variable definition also introduce a getter function x which returns the value currently assigned to the variable, as well as a setter function `$x$_=` which changes the value currently assigned to the variable. The functions have the same signatures as for a variable declaration. The template then has these getter and setter functions as members, whereas the original variable cannot be accessed directly as a template member.

Example

The following example shows how *properties* can be simulated in Scala. It defines a class `TimeOfDayVar` of time values with updatable integer fields representing hours, minutes, and seconds. Its implementation contains tests that allow only legal values to be assigned to these fields. The user code, on the other hand, accesses these fields just like normal variables.

```
class TimeOfDayVar {
  private var h: Int = 0
  private var m: Int = 0
  private var s: Int = 0

  def hours              =  h
  def hours_= (h: Int)   =  if (0 <= h && h < 24) this.h = h
                            else throw new DateError()

  def minutes            =  m
  def minutes_= (m: Int) =  if (0 <= m && m < 60) this.m = m
                            else throw new DateError()

  def seconds            =  s
  def seconds_= (s: Int) =  if (0 <= s && s < 60) this.s = s
                            else throw new DateError()
}
val d = new TimeOfDayVar
d.hours = 8; d.minutes = 30; d.seconds = 0
d.hours = 25                    // throws a DateError exception
```

A variable declaration `var $x_1 , \ldots , x_n$: $T$` is a shorthand for the sequence of variable declarations `var $x_1$: $T$; ...; var $x_n$: $T$`. A variable definition `var $x_1 , \ldots , x_n$ = $e$` is a shorthand for the sequence of variable definitions `var $x_1$ = $e$; ...; var $x_n$ = $e$`. A variable definition `var $x_1 , \ldots , x_n: T$ = $e$` is a shorthand for the sequence of variable definitions `var $x_1: T$ = $e$; ...; var $x_n: T$ = $e$`.

Type Declarations and Type Aliases

```
Dcl         ::=  ‘type’ {nl} TypeDcl
TypeDcl     ::=  id [TypeParamClause] [‘>:’ Type] [‘<:’ Type]
Def         ::=  ‘type’ {nl} TypeDef
TypeDef     ::=  id [TypeParamClause] ‘=’ Type
```

A *type declaration* `type $t$[$\mathit{tps}\,$] >: $L$ <: $U$` declares t to be an abstract type with lower bound type L and upper bound type U. If the type parameter clause `[$\mathit{tps}\,$]` is omitted, t abstracts over a first-order type, otherwise t stands for a type constructor that accepts type arguments as described by the type parameter clause.

If a type declaration appears as a member declaration of a type, implementations of the type may implement t with any type T for which $L <: T <: U$. It is a compile-time error if L does not conform to U. Either or both bounds may be omitted. If the lower bound L is absent, the bottom type `scala.Nothing` is assumed. If the upper bound U is absent, the top type `scala.Any` is assumed.

A type constructor declaration imposes additional restrictions on the concrete types for which t may stand. Besides the bounds L and U, the type parameter clause may impose higher-order bounds and variances, as governed by the conformance of type constructors.

The scope of a type parameter extends over the bounds `>: $L$ <: $U$` and the type parameter clause *tps* itself. A higher-order type parameter clause (of an abstract type constructor *tc*) has the same kind of scope, restricted to the declaration of the type parameter *tc*.

To illustrate nested scoping, these declarations are all equivalent: `type t[m[x] <: Bound[x], Bound[x]]`, `type t[m[x] <: Bound[x], Bound[y]]` and `type t[m[x] <: Bound[x], Bound[_]]`, as the scope of, e.g., the type parameter of m is limited to the declaration of m. In all of them, t is an abstract type member that abstracts over two type constructors: m stands for a type constructor that takes one type parameter and that must be a subtype of *Bound*, t's second type constructor parameter. `t[MutableList, Iterable]` is a valid use of t.

A *type alias* `type $t$ = $T$` defines t to be an alias name for the type T. The left hand side of a type alias may have a type parameter clause, e.g. `type $t$[$\mathit{tps}\,$] = $T$`. The scope of a type parameter extends over the right hand side T and the type parameter clause *tps* itself.

The scope rules for definitions and type parameters make it possible that a type name appears in its own bound or in its right-hand side. However, it is a static error if a type alias refers recursively to the defined type constructor itself. That is, the type T in a type alias `type $t$[$\mathit{tps}\,$] = $T$` may not refer directly or indirectly to the name t. It is also an error if an abstract type is directly or indirectly its own upper or lower bound.

Example

The following are legal type declarations and definitions:

```
type IntList = List[Integer]
type T <: Comparable[T]
type Two[A] = Tuple2[A, A]
type MyCollection[+X] <: Iterable[X]
```

The following are illegal:

```
type Abs = Comparable[Abs]      // recursive type alias

type S <: T                     // S, T are bounded by themselves.
type T <: S

type T >: Comparable[T.That]    // Cannot select from T.
                                // T is a type, not a value
type MyCollection <: Iterable   // Type constructor members must explicitly
                                // state their type parameters.
```

If a type alias `type $t$[$\mathit{tps}\,$] = $S$` refers to a class type S, the name t can also be used as a constructor for objects of type S.

Example

Suppose we make `Pair` an alias of the parameterized class `Tuple2`, as follows:

```
type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

As a consequence, for any two types S and T, the type `Pair[$S$, $T\,$]` is equivalent to the type `Tuple2[$S$, $T\,$]`. `Pair` can also be used as a constructor instead of `Tuple2`, as in:

```
val x: Pair[Int, String] = new Pair(1, "abc")
```

Type Parameters

```
TypeParamClause  ::= ‘[’ VariantTypeParam {‘,’ VariantTypeParam} ‘]’
VariantTypeParam ::= {Annotation} [‘+’ | ‘-’] TypeParam
TypeParam        ::= (id | ‘_’) [TypeParamClause] [‘>:’ Type] [‘<:’ Type] [‘:’ Type]
```

Type parameters appear in type definitions, class definitions, and function definitions. In this section we consider only type parameter definitions with lower bounds `>: $L$` and upper bounds `<: $U$` whereas a discussion of context bounds `: $U$` and view bounds `<% $U$` is deferred to here.

The most general form of a first-order type parameter is `$@a_1 \ldots @a_n$ $\pm$ $t$ >: $L$ <: $U$`. Here, L, and U are lower and upper bounds that constrain possible type arguments for the parameter. It is a compile-time error if L does not conform to U. $\pm$ is a *variance*, i.e. an optional prefix of either `+`, or `-`. One or more annotations may precede the type parameter.

The names of all type parameters must be pairwise different in their enclosing type parameter clause. The scope of a type parameter includes in each case the whole type parameter clause. Therefore it is possible that a type parameter appears as part of its own bounds or the bounds of other type parameters in the same clause. However, a type parameter may not be bounded directly or indirectly by itself.

A type constructor parameter adds a nested type parameter clause to the type parameter. The most general form of a type constructor parameter is `$@a_1\ldots@a_n$ $\pm$ $t[\mathit{tps}\,]$ >: $L$ <: $U$`.

The above scoping restrictions are generalized to the case of nested type parameter clauses, which declare higher-order type parameters. Higher-order type parameters (the type parameters of a type parameter t) are only visible in their immediately surrounding parameter clause (possibly including clauses at a deeper nesting level) and in the bounds of t. Therefore, their names must only be pairwise different from the names of other visible parameters. Since the names of higher-order type parameters are thus often irrelevant, they may be denoted with a '`_`', which is nowhere visible.

Example

Here are some well-formed type parameter clauses:

```
[S, T]
[@specialized T, U]
[Ex <: Throwable]
[A <: Comparable[B], B <: A]
[A, B >: A, C >: A <: B]
[M[X], N[X]]
[M[_], N[_]] // equivalent to previous clause
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
```

The following type parameter clauses are illegal:

```
[A >: A]                  // illegal, `A' has itself as bound
[A <: B, B <: C, C <: A]  // illegal, `A' has itself as bound
[A, B, C >: A <: B]       // illegal lower bound `A' of `C' does
                          // not conform to upper bound `B'.
```

Variance Annotations

Variance annotations indicate how instances of parameterized types vary with respect to subtyping. A '+' variance indicates a covariant dependency, a '-' variance indicates a contravariant dependency, and a missing variance indication indicates an invariant dependency.

A variance annotation constrains the way the annotated type variable may appear in the type or class which binds the type parameter. In a type definition `type $T$[$\mathit{tps}\,$] = $S$`, or a type declaration `type $T$[$\mathit{tps}\,$] >: $L$ <: $U$` type parameters labeled '+' must only appear in covariant position whereas type parameters labeled '-' must only appear in contravariant position. Analogously, for a class definition `class $C$[$\mathit{tps}\,$]($\mathit{ps}\,$) extends $T$ { $x$: $S$ => ...}`, type parameters labeled '+' must only appear in covariant position in the self type S and the template T, whereas type parameters labeled '-' must only appear in contravariant position.

The variance position of a type parameter in a type or template is defined as follows. Let the opposite of covariance be contravariance, and the opposite of invariance be itself. The top-level of the type or template is always in covariant position. The variance position changes at the following constructs.

- The variance position of a method parameter is the opposite of the variance position of the enclosing parameter clause.
- The variance position of a type parameter is the opposite of the variance position of the enclosing type parameter clause.
- The variance position of the lower bound of a type declaration or type parameter is the opposite of the variance position of the type declaration or parameter.
- The type of a mutable variable is always in invariant position.
- The right-hand side of a type alias is always in invariant position.
- The prefix S of a type selection `$S$#$T$` is always in invariant position.
- For a type argument T of a type `$S$[$\ldots T \ldots$ ]`: If the corresponding type parameter is invariant, then T is in invariant position. If the corresponding type parameter is contravariant, the variance position of T is the opposite of the variance position of the enclosing type `$S$[$\ldots T \ldots$ ]`.

References to the type parameters in object-private or object-protected values, types, variables, or methods of the class are not checked for their variance position. In these members the type parameter may appear anywhere without restricting its legal variance annotations.

Example

The following variance annotation is legal.

```
abstract class P[+A, +B] {
  def fst: A; def snd: B
}
```

With this variance annotation, type instances of P subtype covariantly with respect to their arguments. For instance,

```
P[IOException, String] <: P[Throwable, AnyRef]
```

If the members of P are mutable variables, the same variance annotation becomes illegal.

```
abstract class Q[+A, +B](x: A, y: B) {
  var fst: A = x           // **** error: illegal variance:
  var snd: B = y           // `A', `B' occur in invariant position.
}
```

If the mutable variables are object-private, the class definition becomes legal again:

```
abstract class R[+A, +B](x: A, y: B) {
  private[this] var fst: A = x        // OK
  private[this] var snd: B = y        // OK
}
```

Example

The following variance annotation is illegal, since a appears in contravariant position in the parameter of `append`:

```
abstract class Sequence[+A] {
  def append(x: Sequence[A]): Sequence[A]
                  // **** error: illegal variance:
                  // `A' occurs in contravariant position.
}
```

The problem can be avoided by generalizing the type of `append` by means of a lower bound:

```
abstract class Sequence[+A] {
  def append[B >: A](x: Sequence[B]): Sequence[B]
}
```

Example

```
abstract class OutputChannel[-A] {
  def write(x: A): Unit
}
```

With that annotation, we have that `OutputChannel[AnyRef]` conforms to `OutputChannel[String]`. That is, a channel on which one can write any object can substitute for a channel on which one can write only strings.

Function Declarations and Definitions

```
Dcl                ::=  ‘def’ FunDcl
FunDcl             ::=  FunSig ‘:’ Type
Def                ::=  ‘def’ FunDef
FunDef             ::=  FunSig [‘:’ Type] ‘=’ Expr
FunSig             ::=  id [FunTypeParamClause] ParamClauses
FunTypeParamClause ::=  ‘[’ TypeParam {‘,’ TypeParam} ‘]’
ParamClauses     ::= {ParamClause} [[nl] ‘(’ ‘implicit’ Params ‘)’]
ParamClause        ::=  [nl] ‘(’ [Params] ‘)’
Params             ::=  Param {‘,’ Param}
Param              ::=  {Annotation} id [‘:’ ParamType] [‘=’ Expr]
ParamType          ::=  Type
                     |  ‘=>’ Type
                     |  Type ‘*’
```

A *function declaration* has the form `def $f\,\mathit{psig}$: $T$`, where f is the function's name, *psig* is its parameter signature and T is its result type. A *function definition* `def $f\,\mathit{psig}$: $T$ = $e$` also includes a *function body* e, i.e. an expression which defines the function's result. A parameter signature consists of an optional type parameter clause `[$\mathit{tps}\,$]`, followed by zero or more value parameter clauses `($\mathit{ps}_1$)$\ldots$($\mathit{ps}_n$)`. Such a declaration or definition introduces a value with a (possibly polymorphic) method type whose parameter types and result type are as given.

The type of the function body is expected to conform to the function's declared result type, if one is given. If the function definition is not recursive, the result type may be omitted, in which case it is determined from the packed type of the function body.

A *type parameter clause tps* consists of one or more type declarations, which introduce type parameters, possibly with bounds. The scope of a type parameter includes the whole signature, including any of the type parameter bounds as well as the function body, if it is present.

A *value parameter clause ps* consists of zero or more formal parameter bindings such as `$x$: $T$` or `$x: T = e$`, which bind value parameters and associate them with their types.

Default Arguments

Each value parameter declaration may optionally define a default argument. The default argument expression e is type-checked with an expected type T' obtained by replacing all occurrences of the function's type parameters in T by the undefined type.

For every parameter $p_{i,j}$ with a default argument a method named `$f\$$default$\$$n` is generated which computes the default argument expression. Here, n denotes the parameter's position in the method declaration. These methods are parametrized by the type parameter clause `[$\mathit{tps}\,$]` and all value parameter clauses `($\mathit{ps}_1$)$\ldots$($\mathit{ps}_{i-1}$)` preceding $p_{i,j}$. The `$f\$$default$\$$n` methods are inaccessible for user programs.

Example

In the method

```
def compare[T](a: T = 0)(b: T = a) = (a == b)
```

the default expression `0` is type-checked with an undefined expected type. When applying `compare()`, the default value `0` is inserted and `T` is instantiated to `Int`. The methods computing the default arguments have the form:

```
def compare$\$$default$\$$1[T]: Int = 0
def compare$\$$default$\$$2[T](a: T): T = a
```

The scope of a formal value parameter name x comprises all subsequent parameter clauses, as well as the method return type and the function body, if they are given. Both type parameter names and value parameter names must be pairwise distinct.

A default value which depends on earlier parameters uses the actual arguments if they are provided, not the default arguments.

```
def f(a: Int = 0)(b: Int = a + 1) = b // OK
// def f(a: Int = 0, b: Int = a + 1)  // "error: not found: value a"
f(10)()                               // returns 11 (not 1)
```

If an implicit argument is not found by implicit search, it may be supplied using a default argument.

```
implicit val i: Int = 2
def f(implicit x: Int, s: String = "hi") = s * x
f                                     // "hihi"
```

By-Name Parameters

```
ParamType          ::=  ‘=>’ Type
```

The type of a value parameter may be prefixed by `=>`, e.g. `$x$: => $T$`. The type of such a parameter is then the parameterless method type `=> $T$`. This indicates that the corresponding argument is not evaluated at the point of function application, but instead is evaluated at each use within the function. That is, the argument is evaluated using *call-by-name*.

The by-name modifier is disallowed for parameters of classes that carry a `val` or `var` prefix, including parameters of case classes for which a `val` prefix is implicitly generated.

Example

The declaration

```
def whileLoop (cond: => Boolean) (stat: => Unit): Unit
```

indicates that both parameters of `whileLoop` are evaluated using call-by-name.

Repeated Parameters

```
ParamType          ::=  Type ‘*’
```

The last value parameter of a parameter section may be suffixed by `'*'`, e.g. `(..., $x$:$T$*)`. The type of such a *repeated* parameter inside the method is then the sequence type `scala.Seq[$T$]`. Methods with repeated parameters `$T$*` take a variable number of arguments of type $T$. That is, if a method m with type `($p_1:T_1 , \ldots , p_n:T_n, p_s:S$*)$U$` is applied to arguments $(e_1, \ldots, e_k)$ where $k \geq n$, then m is taken in that application to have type $(p_1 : T_1, \ldots, p_n : T_n, p_s : S, \ldots, p_{s'} S)U$, with $k - n$ occurrences of type $S$ where any parameter names beyond p_s are fresh. The only exception to this rule is if the last argument is marked to be a *sequence argument* via a `_*` type annotation. If m above is applied to arguments `($e_1 , \ldots , e_n, e'$: _*)`, then the type of m in that application is taken to be `($p_1:T_1, \ldots , p_n:T_n,p_{s}:$scala.Seq[$S$])`.

It is not allowed to define any default arguments in a parameter section with a repeated parameter.

Example

The following method definition computes the sum of the squares of a variable number of integer arguments.

```
def sum(args: Int*) = {
  var result = 0
  for (arg <- args) result += arg
  result
}
```

The following applications of this method yield `0`, `1`, `6`, in that order.

```
sum()
sum(1)
sum(1, 2, 3)
```

Furthermore, assume the definition:

```
val xs = List(1, 2, 3)
```

The following application of method `sum` is ill-formed:

```
sum(xs)       // ***** error: expected: Int, found: List[Int]
```

By contrast, the following application is well formed and yields again the result `6`:

```
sum(xs: _*)
```

Procedures

```
FunDcl   ::=  FunSig
FunDef   ::=  FunSig [nl] ‘{’ Block ‘}’
```

Special syntax exists for procedures, i.e. functions that return the `Unit` value `()`. A *procedure declaration* is a function declaration where the result type is omitted. The result type is then implicitly completed to the `Unit` type. E.g., `def $f$($\mathit{ps}$)` is equivalent to `def $f$($\mathit{ps}$): Unit`.

A *procedure definition* is a function definition where the result type and the equals sign are omitted; its defining expression must be a block. E.g., `def $f$($\mathit{ps}$) {$\mathit{stats}$}` is equivalent to `def $f$($\mathit{ps}$): Unit = {$\mathit{stats}$}`.

Example

Here is a declaration and a definition of a procedure named `write`:

```
trait Writer {
  def write(str: String)
}
object Terminal extends Writer {
  def write(str: String) { System.out.println(str) }
}
```

The code above is implicitly completed to the following code:

```
trait Writer {
  def write(str: String): Unit
}
object Terminal extends Writer {
  def write(str: String): Unit = { System.out.println(str) }
}
```

Method Return Type Inference

A class member definition m that overrides some other function m' in a base class of C may leave out the return type, even if it is recursive. In this case, the return type R' of the overridden function m', seen as a member of C, is taken as the return type of m for each recursive invocation of m. That way, a type

R for the right-hand side of m can be determined, which is then taken as the return type of m. Note that R may be different from R', as long as R conforms to R'.

Example

Assume the following definitions:

```
trait I {
  def factorial(x: Int): Int
}
class C extends I {
  def factorial(x: Int) = if (x == 0) 1 else x * factorial(x - 1)
}
```

Here, it is OK to leave out the result type of `factorial` in `C`, even though the method is recursive.

Import Clauses

```
Import          ::= ‘import’ ImportExpr {‘,’ ImportExpr}
ImportExpr      ::= StableId ‘.’ (id | ‘_’ | ImportSelectors)
ImportSelectors ::= ‘{’ {ImportSelector ‘,’}
                    (ImportSelector | ‘_’) ‘}’
ImportSelector  ::= id [‘=>’ id | ‘=>’ ‘_’]
```

An import clause has the form `import $p$.$I$` where p is a stable identifier and I is an import expression. The import expression determines a set of names of importable members of p which are made available without qualification. A member m of p is *importable* if it is accessible. The most general form of an import expression is a list of *import selectors*

```
{ $x_1$ => $y_1 , \ldots , x_n$ => $y_n$, _ }
```

for $n \geq 0$, where the final wildcard ‘`_`’ may be absent. It makes available each importable member `$p$.$x_i$` under the unqualified name y_i. I.e. every import selector `$x_i$ => $y_i$` renames `$p$.$x_i$` to y_i. If a final wildcard is present, all importable members z of p other than `$x_1 , \ldots , x_n,y_1 , \ldots , y_n$` are also made available under their own unqualified names.

Import selectors work in the same way for type and term members. For instance, an import clause `import $p$.{$x$ => $y\,$}` renames the term name `$p$.$x$` to the term name y and the type name `$p$.$x$` to the type name y. At least one of these two names must reference an importable member of p.

If the target in an import selector is a wildcard, the import selector hides access to the source member. For instance, the import selector `$x$ => _` “renames” x to the wildcard symbol (which is unaccessible as a name in user programs), and thereby effectively prevents unqualified access to x. This is useful if there is a final wildcard in the same import selector list, which imports all members not mentioned in previous import selectors.

The scope of a binding introduced by an import-clause starts immediately after the import clause and extends to the end of the enclosing block, template, package clause, or compilation unit, whichever comes first.

Several shorthands exist. An import selector may be just a simple name x. In this case, x is imported without renaming, so the import selector is equivalent to `$x$ => $x$`. Furthermore, it is possible to replace the whole import selector list by a single identifier or wildcard. The import clause `import $p$.$x$` is equivalent to `import $p$.{$x\,$}`, i.e. it makes available without qualification the member x of p. The import clause `import $p$._` is equivalent to `import $p$.{_}`, i.e. it makes available without qualification all members of p (this is analogous to `import $p$.*` in Java).

An import clause with multiple import expressions `import $p_1$.$I_1 , \ldots , p_n$.$I_n$` is interpreted as a sequence of import clauses `import $p_1$.$I_1$; $\ldots$; import $p_n$.$I_n$`.

Example

Consider the object definition:

```
object M {
  def z = 0, one = 1
  def add(x: Int, y: Int): Int = x + y
}
```

Then the block

```
{ import M.{one, z => zero, _}; add(zero, one) }
```

is equivalent to the block

```
{ M.add(M.z, M.one) }
```