Identifiers, Names & Scopes

Identifiers, Names and Scopes

Names in Scala identify types, values, methods, and classes which are collectively called *entities*. Names are introduced by local definitions and declarations, inheritance, import clauses, or package clauses which are collectively called *bindings*.

Bindings of different kinds have a precedence defined on them:

1. Definitions and declarations that are local, inherited, or made available by a package clause and also defined in the same compilation unit as the reference to them, have highest precedence.
2. Explicit imports have next highest precedence.
3. Wildcard imports have next highest precedence.
4. Definitions made available by a package clause, but not also defined in the same compilation unit as the reference to them, as well as imports which are supplied by the compiler but not explicitly written in source code, have lowest precedence.

There are two different name spaces, one for types and one for terms. The same name may designate a type and a term, depending on the context where the name is used.

A binding has a *scope* in which the entity defined by a single name can be accessed using a simple name. Scopes are nested. A binding in some inner scope *shadows* bindings of lower precedence in the same scope as well as bindings of the same or lower precedence in outer scopes.

Note that shadowing is only a partial order. In the following example, neither binding of `x` shadows the other. Consequently, the reference to `x` in the last line of the block is ambiguous.

```
val x = 1
locally {
  import p.X.x
  x
}
```

A reference to an unqualified (type- or term-) identifier x is bound by the unique binding, which

- defines an entity with name x in the same namespace as the identifier, and
- shadows all other bindings that define entities with name x in that namespace.

It is an error if no such binding exists. If x is bound by an import clause, then the simple name x is taken to be equivalent to the qualified name to which x is mapped by the import clause. If x is bound by a definition or declaration, then x refers to the entity introduced by that binding. In that case, the type of x is the type of the referenced entity.

A reference to a qualified (type- or term-) identifier $e.x$ refers to the member of the type T of e which has the name x in the same namespace as the identifier. It is an error if T is not a value type. The type of $e.x$ is the member type of the referenced entity in T.

Binding precedence implies that the way source is bundled in files affects name resolution. In particular, imported names have higher precedence than names, defined in other files, that might otherwise be visible because they are defined in either the current package or an enclosing package.

Note that a package definition is taken as lowest precedence, since packages are open and can be defined across arbitrary compilation units.

```
package util {
  import scala.util
  class Random
  object Test extends App {
    println(new util.Random)  // scala.util.Random
  }
}
```

The compiler supplies imports in a preamble to every source file. This preamble conceptually has the following form, where braces indicate nested scopes:

```
import java.lang._
{
  import scala._
  {
    import Predef._
    { /* source */ }
  }
}
```

These imports are taken as lowest precedence, so that they are always shadowed by user code, which may contain competing imports and definitions. They also increase the nesting depth as shown, so that later imports shadow earlier ones.

As a convenience, multiple bindings of a type identifier to the same underlying type is permitted. This is possible when import clauses introduce a binding of a member type alias with the same binding precedence, typically through

wildcard imports. This allows redundant type aliases to be imported without introducing an ambiguity.

```
object X { type T = annotation.tailrec }
object Y { type T = annotation.tailrec }
object Z {
  import X._, Y._, annotation.{tailrec => T}  // OK, all T mean tailrec
  @T def f: Int = { f ; 42 }                  // error, f is not tail recursive
}
```

Similarly, imported aliases of names introduced by package statements are allowed, even though the names are strictly ambiguous:

```
// c.scala
package p { class C }

// xy.scala
import p._
package p { class X extends C }
package q { class Y extends C }
```

The reference to `C` in the definition of `X` is strictly ambiguous because `C` is available by virtue of the package clause in a different file, and can't shadow the imported name. But because the references are the same, the definition is taken as though it did shadow the import.

Example

Assume the following two definitions of objects named `X` in packages `p` and `q` in separate compilation units.

```
package p {
  object X { val x = 1; val y = 2 }
}

package q {
  object X { val x = true; val y = false }
}
```

The following program illustrates different kinds of bindings and precedences between them.

```
package p {                   // `X' bound by package clause
import Console._              // `println' bound by wildcard import
object Y {
  println(s"L4: \$X")          // `X' refers to `p.X' here
  locally {
    import q._                // `X' bound by wildcard import
    println(s"L7: \$X")        // `X' refers to `q.X' here
    import X._                // `x' and `y' bound by wildcard import
    println(s"L9: \$x")        // `x' refers to `q.X.x' here
```

```
    locally {
      val x = 3               // `x' bound by local definition
      println(s"L12: \$x")     // `x' refers to constant `3' here
      locally {
        import q.X._          // `x' and `y' bound by wildcard import
//      println(s"L15: \$x")   // reference to `x' is ambiguous here
        import X.y            // `y' bound by explicit import
        println(s"L17: \$y")   // `y' refers to `q.X.y' here
        locally {
          val x = "abc"       // `x' bound by local definition
          import p.X._        // `x' and `y' bound by wildcard import
//        println(s"L21: \$y") // reference to `y' is ambiguous here
          println(s"L22: \$x") // `x' refers to string "abc" here
}}}}}}
```