Top-Level Definitions

Top-Level Definitions

Compilation Units

```
CompilationUnit  ::=  {‘package’ QualId semi} TopStatSeq
TopStatSeq       ::=  TopStat {semi TopStat}
TopStat          ::=  {Annotation} {Modifier} TmplDef
                   |  Import
                   |  Packaging
                   |  PackageObject
                   |
QualId           ::=  id {‘.’ id}
```

A compilation unit consists of a sequence of packagings, import clauses, and class and object definitions, which may be preceded by a package clause.

A *compilation unit*

```
package $p_1$;
$\ldots$
package $p_n$;
$\mathit{stats}$
```

starting with one or more package clauses is equivalent to a compilation unit consisting of the packaging

```
package $p_1$ { $\ldots$
  package $p_n$ {
    $\mathit{stats}$
  } $\ldots$
}
```

Every compilation unit implicitly imports the following packages, in the given order: 1. the package `java.lang`, 2. the package `scala`, and 3. the object `scala.Predef`, unless there is an explicit top-level import that references `scala.Predef`.

Members of a later import in that order hide members of an earlier import.

The exception to the implicit import of `scala.Predef` can be useful to hide, e.g., predefined implicit conversions.

Packagings

```
Packaging       ::=  ‘package’ QualId [nl] ‘{’ TopStatSeq ‘}’
```

A *package* is a special object which defines a set of member classes, objects and packages. Unlike other objects, packages are not introduced by a definition. Instead, the set of members of a package is determined by packagings.

A packaging `package $p$ { $\mathit{ds}$ }` injects all definitions in *ds* as members into the package whose qualified name is p. Members of a package are called *top-level* definitions. If a definition in *ds* is labeled `private`, it is visible only for other members in the package.

Inside the packaging, all members of package p are visible under their simple names. However this rule does not extend to members of enclosing packages of p that are designated by a prefix of the path p.

```
package org.net.prj {
  ...
}
```

all members of package `org.net.prj` are visible under their simple names, but members of packages `org` or `org.net` require explicit qualification or imports.

Selections $p.m$ from p as well as imports from p work as for objects. However, unlike other objects, packages may not be used as values. It is illegal to have a package with the same fully qualified name as a module or a class.

Top-level definitions outside a packaging are assumed to be injected into a special empty package. That package cannot be named and therefore cannot be imported. However, members of the empty package are visible to each other without qualification.

Package Objects

```
PackageObject   ::=  ‘package’ ‘object’ ObjectDef
```

A *package object* `package object $p$ extends $t$` adds the members of template t to the package p. There can be only one package object per package. The standard naming convention is to place the definition above in a file named `package.scala` that's located in the directory corresponding to package p.

The package object should not define a member with the same name as one of the top-level objects or classes defined in package p. If there is a name conflict, the behavior of the program is currently undefined. It is expected that this restriction will be lifted in a future version of Scala.

Package References

```
QualId           ::=  id {‘.’ id}
```

A reference to a package takes the form of a qualified identifier. Like all other references, package references are relative. That is, a package reference starting in a name *p* will be looked up in the closest enclosing scope that defines a member named *p*.

If a package name is shadowed, it's possible to refer to its fully-qualified name by prefixing it with the special predefined name `_root_`, which refers to the outermost root package that contains all top-level packages.

The name `_root_` has this special denotation only when used as the first element of a qualifier; it is an ordinary identifier otherwise.

Example

Consider the following program:

```
package b {
  class B
}

package a {
  package b {
    class A {
      val x = new _root_.b.B
    }
    class C {
      import _root_.b._
      def y = new B
    }
  }
}
```

Here, the reference `_root_.b.B` refers to class `B` in the toplevel package `b`. If the `_root_` prefix had been omitted, the name `b` would instead resolve to the package `a.b`, and, provided that package does not also contain a class `B`, a compiler-time error would result.

Programs

A *program* is a top-level object that has a member method *main* of type `(Array[String])Unit`. Programs can be executed from a command shell. The program's command arguments are passed to the `main` method as a parameter of type `Array[String]`.

The `main` method of a program can be directly defined in the object, or it can be inherited. The scala library defines a special class `scala.App` whose body acts as a `main` method. An objects m inheriting from this class is thus a program, which executes the initialization code of the object m.

Example

The following example will create a hello world program by defining a method `main` in module `test.HelloWorld`.

```
package test
object HelloWorld {
  def main(args: Array[String]) { println("Hello World") }
}
```

This program can be started by the command

```
scala test.HelloWorld
```

In a Java environment, the command

```
java test.HelloWorld
```

would work as well.

`HelloWorld` can also be defined without a `main` method by inheriting from `App` instead:

```
package test
object HelloWorld extends App {
  println("Hello World")
}
```