Packages, Imports and Scope

Controlling Visibility and Access in Scala

Agenda

1. Public, Protected and Private
2. Packages
3. Package Visibility
4. Imports from Packages
5. Package Objects
6. Imports from Objects and Instances
7. Importing Fu
8. Companion Objects

Public, Protected and Private

- Scala has three visibility scopes, `private`, `protected` and public.
- There is no public keyword: classes, traits, objects, vals and defs not marked `private` or `protected` are public by default.
- `private` means only available to the current class, companion object, or inner classes/objects in the current class.
- `protected` means `private` access plus availability to any sub-class of the current class.
- public means anyone can see and access the item. This is the default when no other visibility is specified.
- Java's *package protected* default is not the default in Scala, and package access is handled through an orthogonal mechanism.
- Scala also has a `private[this]` visibility which means private to this specific instance only, companions, nested classes and even other instances of the same class cannot access a `private[this]`.
- `private[this]` is the default visibility for constructor parameters that are not parametric-fields (i.e. not in a case class or marked with a val).

Packages

- Packages are a way of organizing classes and objects into different logical access units
- The standard way packages are used in Scala is the same way as Java, a `package` declaration at the start of a source file (and everything in that source file is then in that package):

```
package demo.food.domain.api

trait Dessert

case class IceCream(flavor: String) extends Dessert
case class Jello(color: String) extends Dessert
```

Package Structure Alternatives

- Scala also supports C# namespace style

```
package demo {
  package food {
    package domain {
      package api {
        trait Dessert
        case class IceCream(flavor: String) extends Dessert
        case class Jello(color: String) extends Dessert
      }
    }
  }
}
```

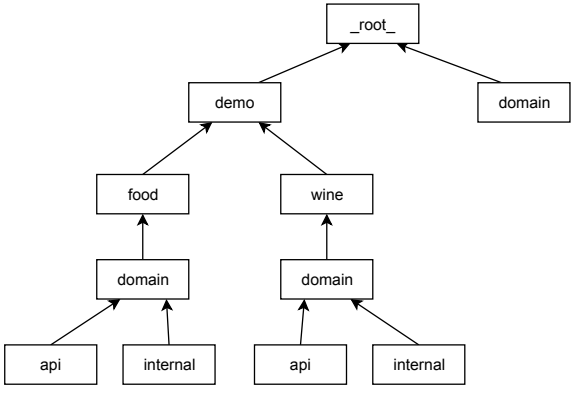
- or in namespace shorthand:

```
package demo.food.domain.api {
  trait Dessert
  case class IceCream(flavor: String) extends Dessert
  case class Jello(color: String) extends Dessert
}
```

- In practice, very few developers use this form, but you may occasionally see it.

Namespace Notation

- Although not widely used, namespace notation does illustrate better the scoping
- It also looks like another `object` or `class` top level definition, and it is close, though only `traits`, `classes` and `objects` can go directly under a `package`
- Namespace notation also allows multiple different packages in a single source file (again, uncommon in Scala)
- Seeing the structures above will help explain some of the remainder of this module



More Parts of the Model

```
package demo.food.domain.api

import demo.food.domain.allFoods  // more on this in a moment

object FindFood {
  def lookupFood(name: String): Option[Dessert] = allFoods.lookupDessert(name)
}
```

```
package demo.food.domain.internal

import demo.food.domain.api.Dessert

import scala.collection.mutable

private[domain] class FoodDB {
  private[this] val desserts = mutable.Map.empty[String, Dessert]

  private[domain] def addDessert(name: String, dessert: Dessert): Unit =
    desserts.put(name, dessert)

  def lookupDessert(name: String): Option[Dessert] = desserts.get(name)
}
```

- `private[domain]` makes this private but accessible up to `demo.food.domain`
- `private[this]` on `desserts` means only this specific instance can access it.

Package Visibility

- `private` and `protected` cover the visibility within the instance and sub-classes
- Package scoping is handled with `[super-package]` after one of those keywords
- The super-package must be one of the parent packages or you will get a compile error
- This allows you to control visibility but still have freedom in organizing sub-packages
- Note that this is a Scala language specific feature, until compatibility with Java super-packages is implemented, package-scoped items are public when accessed from Java!
- `private[this]` is private to the specific instance. It cannot be accessed even from the companion object or another instance of this class. It is the only visibility modifier in Scala that does not have accessor methods created.

Logger in the top level `domain` package:

```
package domain

object Logger {
  def log(msg: String): Unit = println(msg)
}
```

- Normally we would attempt to not have three different `domain` packages located at different parts of the hierarchy because it can make things confusing, but that is the point in this case - to demonstrate disambiguation.

Now for the `wine` package

```
package demo.wine.domain.api

case class Wine(name: String, vintage: Int, varietal: String)
```

```
package demo.wine.domain.internal

import demo.food.domain.api.{Dessert, IceCream}
import demo.wine.domain.api.Wine

private[domain] object PairingDB {
  private[this] val winePairs = Map[Dessert, Wine](
    IceCream("Vanilla") -> Wine("Solis Old Vine Zin", 2014, "Zinfandel"))

  def pairWineWithDessert(dessert: Dessert): Option[Wine] = winePairs.get(dessert)
}
```

- Note the `demo.food.domain.api.{Dessert, IceCream}` to bring in both classes

The `PairWine` object

```
package demo.wine.domain.api

import demo.food._
import domain.api.Dessert
import demo.wine.domain.internal.PairingDB
import _root_.domain.Logger

object PairWine {
  def pairWineWithDessert(dessert: Dessert): Option[Wine] = {
    Logger.log(s"Attempting to pair $dessert")
    PairingDB.pairWineWithDessert(dessert)
  }
}
```

- `import demo.food._` brings in the `domain` package from there
- `import domain.api.Dessert` references that `domain` from `demo.food`
- `import domain.internal.PairingDB` will not work, since that thinks you now mean starting from `domain` in `demo.food`. We have to use `import demo.wine.domain.internal.PairingDB`
- To import `domain.Logger`, we need to use `import _root_.` to force Scala to go back to the root of the package hierarchy.

Package Objects

- Remember `import demo.food.domain.allFoods` ?
- In the JVM, methods and fields must be inside classes
- Unlike the REPL, defs and vals can't just be defined without an enclosing `class`, `trait` or `object`, unless you use a `package object`

```
package demo.food

import demo.food.domain.api.IceCream
import demo.food.domain.internal.FoodDB

package object domain {
  lazy val allFoods: FoodDB = {
    val foodDB = new FoodDB
    foodDB.addDessert("vanilla ice cream", IceCream("Vanilla"))
    foodDB
  }
}
```

- Can now `import demo.food.domain.allFoods` **or** `import demo.food.domain._` will bring the `allFoods` instance in as well
- `package objects` can be a useful place to put `implicits` as well.

Importing from an `object`

- Scala can import anything, from anywhere, at any point in your code
- Importing from a package is the most common, but import from objects and instances is also useful
- E.g. to import the `log` method from Logger directly:

```
import demo.food._
import domain.api.Dessert
import demo.wine.domain.internal.PairingDB

object PairWine {
  import _root_.domain.Logger.log
  def pairWineWithDessert(dessert: Dessert): Option[Wine] = {
    log(s"Attempting to pair $dessert")
    PairingDB.pairWineWithDessert(dessert)
  }
}
```

- Note that the `import` is within the `PairWine` object body, `log` is only in scope inside `PairWine`
- You can import and scope to a single method, or even an arbitrary code block in `{}`s if desired

Importing from an instance

```
import demo.food.domain.api._

val iceCream = IceCream("Vanilla")

def thirdLetterOfDessert(dessert: IceCream): Char = {
  import dessert._
  import flavor._

  charAt(3)
}

thirdLetterOfDessert(iceCream) // Char = i
```

- We import the members of `dessert`, then the members of the `flavor` string
- `charAt(3)` then refers to the `"Vanilla"` string, resulting in `i`
- These imports are only in scope within the `thirdLetterOfDessert` method

Importing Fu: Renaming

```
// import the demo.food.domain package and rename it fooddomain
import demo.food.{domain => fooddomain}

// import the demo.wine.domain package and rename it winedomain
import demo.wine.{domain => winedomain}

// Now there is no confusion referring to the top domain for Logger
import domain.Logger

Logger.log("happy")

import fooddomain.api.{Jello => Jelly, _}
import winedomain.api._
IceCream("Vanilla")
Jelly("green")
Jello("red") // compile error - not found: value Jello
Wine("Foo", 1987, "Cab Sav")
```

- `Jello` is renamed to `Jelly` on the way in, everything else imported from food domain with its regular name

Selective Importing

- Import just `HashMap` and `HashSet` from `immutable`

```
import scala.collection.immutable.{HashMap, HashSet}

HashSet(1,2,3)
HashMap(1 -> "one", 2 -> "two")
```

- Import all of `java.util` except from `Date` and `Deque`, import `ArrayDeque` as `Deque` instead:

```
import java.util.{Date => _, Deque => _, ArrayDeque => Deque, _}

new ArrayList[Int](10) // java.util.ArrayList[Int] = []
new Deque[Int](10) // java.util.ArrayDeque[Int] = []
new Date // compile error
```

- Standard imports for all source files:

```
import java.lang._   // everything in the java.lang package
import scala._         // everything in the scala package
import Predef._       // everything in the Predef object
```

Companion Objects

- Companions can share `private` state, but not `private[this]`:

```
class ShippingContainer[T] private (val items: Seq[T]) {
  private[this] val maxCount = 10
  private def isFull: Boolean = items.length >= maxCount
  override def toString: String = s"${ShippingContainer.containerColor} Container"
}

object ShippingContainer {
  def apply[T](items: T*) = new ShippingContainer[T](items)
  private def containerColor: String = "Green"

  def maxItems(container: ShippingContainer[_]): Int =
    container.maxCount  // compile error, maxCount is private[this]

  def containerFull(container: ShippingContainer[_]): Boolean =
    container.isFull    // fine
}

val sc = ShippingContainer("a", "b", "c") // Green Container
sc.items  // Seq(a,b,c)
sc.maxCount  // compile error
sc.isFull    // compile error
ShippingContainer.containerFull(sc)  // false
```

Exercises for Module 10

- Find the `Module10` class and follow the instructions to make the tests pass
- `Module10` is under `module10/src/test/scala/koans`
- The `Laser` scala file is there for your implementation of the "Shoot a LASER beam" test since packages cannot go in your test class.