Traits;
Packages and Imports

Traits (1)

- A trait encapsulates method and field definitions, which can then be reused by **mixing them into** classes
- Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix in any number of traits

```
trait Philosophical {
   def philosophize() = {
     println("I consume memory, therefore I am!" )
   }
 }
```

- `Philosophical` does not declare a superclass, so like a class, it has the default superclass of `AnyRef`

Traits (2)

- Once a trait is defined, it can be **mixed in** to a class using either the `extends` or `with` keywords.

```
class Frog extends Philosophical {
    override def toString = "green"
  }
```

- Class `Frog` subclasses `AnyRef` (the superclass of `Philosophical`) and mixes in `Philosophical`

```
val frog = new Frog

frog.philosophize()

  I consume memory, therefore I am!
```

Traits (3)

- A trait also defines a type

```
val phil: Philosophical = frog

phil.philosophize()

 I consume memory, therefore I am!
```

- If you wish to mix a trait into a class that explicitly extends a superclass, you use `extends` to indicate the superclass and `with` to mix in the trait.

```
class Animal

class Frog extends Animal with Philosophical {
   override def toString = "green"
 }
```

Traits (4)

- You can mix in multiple traits

```
class Animal
trait HasLegs

 class Frog extends Animal with Philosophical with HasLegs {
    override def toString = "green"
  }
```

- You can override concrete methods inherited from a trait

```
class Animal

 class Frog extends Animal with Philosophical {
    override def toString = "green"
    override def philosophize() = {
      println("It ain't easy being "  + toString + "!")
    }
  }
```

Traits (5)

- You can do anything in a trait definition that you can do in a class definition, and the syntax looks exactly the same, with only two exceptions
  - A trait cannot have any "class" parameters

```
class Point(x: Int, y: Int)
```

vs

```
trait NoPoint(x: Int, y: Int) // Does not compile
```

  - In classes, `super.someMethod` calls are statically bound, in traits, they are dynamically bound. The implementation to invoke will be determined anew each time the trait is mixed into a concrete class => *Stackable modifications*

Thin versus rich interfaces

- A *rich* interface has many methods => Good for the caller, bad for the implementer. A *thin* interface has fewer methods => Good for the implementer, bad for the caller
- One major use of traits is to automatically add methods to a class in terms of methods the class already has. That is, traits can *enrich* a thin interface, making it into a rich interface
- To enrich an interface using traits, simply define a trait with a small number of abstract methods, and a potentially large number of concrete methods, **all implemented in terms of the abstract methods**
- Then you can mix the enrichment trait into a class, implement the thin portion of the interface, and end up with a class that has all of the rich interface available

Example: Rectangular objects (1)

```
class Point(val x: Int, val y: Int)

 class Rectangle(val topLeft: Point, val bottomRight: Point) {
    def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
  }

abstract class Component {
    def topLeft: Point
    def bottomRight: Point

   def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
  }
```

- Notice that the definitions of `left`, `right`, and `width` are exactly the same in the two classes.

Example: Rectangular objects (2)

```
trait Rectangular {
    def topLeft: Point
    def bottomRight: Point

   def left = topLeft.x
    def right = bottomRight.x
    def width = right - left
    // and many more geometric methods...
  }
```

- Classes then can mix in this trait to get all the geometric methods provided by Rectangular

```
abstract class Component extends Rectangular {
     // other methods...
}
class Rectangle(val topLeft: Point, val bottomRight: Point)
     extends Rectangular {
     // other methods…
}
```

Example: Rectangular objects (3)

- Given these definitions, you can create a `Rectangle` and call geometric methods such as width and left on it

```
scala> val rect = new Rectangle(new Point(1, 1),
             new Point(10, 10))
 rect: Rectangle = Rectangle@5f5da68c

scala> rect.left
 res2: Int = 1

scala> rect.right
 res3: Int = 10

scala> rect.width
 res4: Int = 9
```

Example: The Ordered trait

```
trait Ordered[T] {
  def compare(that: T): Int

 def <(that: T): Boolean = (this compare that) < 0
  def >(that: T): Boolean = (this compare that) > 0
  def <=(that: T): Boolean = (this compare that) <= 0
  def >=(that: T): Boolean = (this compare that) >= 0
}
```

- `T` here is a type parameter, it will be detailed in Chapter 19

Traits as stackable modifications (1)

- Traits let you *modify* the methods of a class, and they do so in a way that allows you to *stack* those modifications with each other
- E.g. We have a queue of integers with a `put` and a `get` method (FIFO)
- Given a class that implements such a queue, you could define traits to perform modifications such as these:
  - `Doubling`: double all integers that are put in the queue
  - `Incrementing`: increment all integers that are put in the queue
  - `Filtering`: filter out negative integers from a queue
- These three traits represent *modifications*, because they modify the behavior of an underlying queue class rather than defining a full queue class themselves. The three are also *stackable*. You can select any of the three you like, mix them into a class, and obtain a new class that has all of the modifications you chose.

Traits as stackable modifications (2)

```
abstract class IntQueue {
   def get(): Int
   def put(x: Int)
}

import scala.collection.mutable.ArrayBuffer

class BasicIntQueue extends IntQueue {
   private val buf = new ArrayBuffer[Int]
   def get() = buf.remove(0)
   def put(x: Int) = { buf += x }
}
```

Traits as stackable modifications (3)

```
scala> val queue = new BasicIntQueue
  queue: BasicIntQueue = BasicIntQueue@23164256

 scala> queue.put(10)

 scala> queue.put(20)

 scala> queue.get()
  res9: Int = 10

 scala> queue.get()
  res10: Int = 20
```

Traits as stackable modifications (4)

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) = { super.put(2 * x) }
}
```

- `Doubling` declares a superclass, `IntQueue`. This declaration means that the trait can only be mixed into a class that also extends `IntQueue`
- It has a `super` call on a method declared abstract
- Since `super` calls in a trait are dynamically bound, the `super` call in trait `Doubling` will work so long as the trait is mixed in *after* another trait or class that gives a concrete definition to the method
- To tell the compiler you are doing this on purpose, you must mark such methods as `abstract override`
- It is only allowed for members of traits, and it means that the trait must be mixed into some class that has a concrete definition of the method in question

Traits as stackable modifications (5)

```
scala> val queue = new BasicIntQueue with Doubling
  queue: BasicIntQueue with Doubling = $anon$1@141f05bf

 scala> queue.put( 10)

 scala> queue.get()
  res14: Int = 20
```

- We put a 10 in the queue, but because `Doubling` has been mixed in, the 10 is doubled. When we get an integer from the queue, it is a 20

Traits as stackable modifications (6)

- To see how to stack modifications, we need to define the other two modification traits, `Incrementing` and `Filtering`

```
trait Incrementing extends IntQueue {
    abstract override def put(x: Int) = { super.put(x + 1) }
  }

  trait Filtering extends IntQueue {
    abstract override def put(x: Int) = {
      if (x >= 0) super.put(x)
    }
  }
```

- Given these modifications, you can now pick and choose which ones you want for a particular queue

Traits as stackable modifications (7)

```
scala> val queue = (new BasicIntQueue with Incrementing with Filtering)
  queue: BasicIntQueue with Incrementing with Filtering...

 scala> queue.put(-1); queue.put(0); queue.put(1)

 scala> queue.get()
  res16: Int = 1

 scala> queue.get()
  res17: Int = 2
```

- The order of mixins is significant. **The method in the trait furthest to the right is called first**. If that method calls `super`, it invokes the method in the next trait to its left, and so on. In the previous example, `Filtering`'s `put` is invoked first, so it removes integers that were negative to begin with. `Incrementing`'s `put` is invoked second, so it adds one to those integers that remain.

Traits as stackable modifications (8)

```
scala> val queue = (new BasicIntQueue
             with Filtering with Incrementing)
  queue: BasicIntQueue with Filtering with Incrementing...

 scala> queue.put(-1); queue.put(0); queue.put(1)

 scala> queue.get()
  res19: Int = ?

 scala> queue.get()
  res20: Int = ?

 scala> queue.get()
  res21: Int = ?
```

Traits as stackable modifications (9)

```
scala> val queue = (new BasicIntQueue
             with Filtering with Incrementing)
 queue: BasicIntQueue with Filtering with Incrementing...

scala> queue.put(-1); queue.put(0); queue.put(1)

scala> queue.get()
 res19: Int = 0

scala> queue.get()
 res20: Int = 1

scala> queue.get()
 res21: Int = 2
```

- Overall, code written in this style gives you a great deal of flexibility. You can define sixteen different classes by mixing in these three traits in different combinations and orders

Why not multiple inheritance?

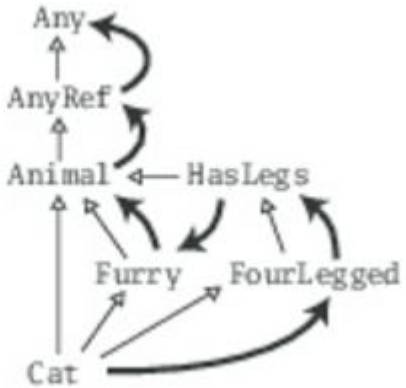
- With multiple inheritance, the method called by a `super` call can be determined right where the call appears
- With traits, the method called is determined by a *linearization* of the classes and traits that are mixed into a class

```
// Multiple inheritance thought experiment
  val q = new BasicIntQueue with Incrementing with Doubling
  q.put(42)  // which put would be called?
```

- When you instantiate a class with `new`, Scala takes the class, and all of its inherited classes and traits, and puts them in a single, *linear* order
- Then, whenever you call `super` inside one of those classes, the invoked method is the next one up the chain

Linearization

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```



Packages and Imports

- Packages: they help you program in a modular style
- The constructs are similar in spirit to constructs in Java, but there are some differences
- Without package definition: *unnamed* package
- You can place code into named packages in Scala in two ways
  - A `package` clause at the top of the file

```
package bobsrockets.navigation
class Navigator
```

  - Curly braces contain the definitions that go into the package (*packaging*)

```
package bobsrockets.navigation {
  class Navigator
}
```

Packages (1)

- You can have different parts of a file in different packages

```
package bobsrockets {
   package navigation {

    // In package bobsrockets.navigation
     class Navigator

    package tests {

      // In package bobsrockets.navigation.tests
       class NavigatorSuite
     }
   }
 }
```

Packages (2)

- A class can be accessed from within its own package without needing a prefix (`StarMap`)
- A package itself can be accessed from its containing package without needing a prefix (`Navigator`)
- When using the curly-braces packaging syntax, all names accessible in scopes outside the packaging are also available inside it (`addShip()`)

Packages (3)

```
package bobsrockets {
   package navigation {
     class Navigator {
       // No need to say bobsrockets.navigation.StarMap
       val map = new StarMap
     }
     class StarMap
   }
   class Ship {
     // No need to say bobsrockets.navigation.Navigator
     val nav = new navigation.Navigator
   }
   package fleets {
     class Fleet {
       // No need to say bobsrockets.Ship
       def addShip() = { new Ship }
     }
   }
 }
```

Packages (4)

- The last kind of access is only available if you explicitly nest the packagings

```
package bobsrockets {
      class Ship
}
package bobsrockets.fleets {
      class Fleet {
        // Doesn't compile! Ship is not in scope.
        def addShip() = { new Ship }
      }
    }
```

- You can also use multiple package clauses without the braces

```
package bobsrockets
package fleets
class Fleet {
  // No need to say bobsrockets.Ship
  def addShip() = { new Ship }
}
```

Packages (5)

- Sometimes, you end up coding in a heavily crowded scope where package names are hiding each other
- Scala provides a package named `_root_` that is outside any package a user can write
- Every top-level package you can write is treated as a member of package `_root_`

Packages (6)

```
// In file launch.scala
 package launch {
   class Booster3
 }

// In file bobsrockets.scala
 package bobsrockets {
   package navigation {
     package launch {
       class Booster1
     }
     class MissionControl {
       val booster1 = new launch.Booster1
       val booster2 = new bobsrockets.launch.Booster2
       val booster3 = new _root_.launch.Booster3
     }
   }
   package launch {
     class Booster2
   }
 }
```

Imports (1)

```
package bobsdelights

abstract class Fruit(
   val name: String,
   val color: String
 )
object Fruits {
   object Apple extends Fruit("apple", "red")
   object Orange extends Fruit("orange", "orange")
   object Pear extends Fruit("pear", "yellowish")
   val menu = List(Apple, Orange, Pear)
 }
----------------------------
// easy access to Fruit
  import bobsdelights.Fruit

 // easy access to all members of bobsdelights
  import bobsdelights._

 // easy access to all members of Fruits
  import bobsdelights.Fruits._
```

Imports (2)

- Imports in Scala can appear anywhere
- They can refer to arbitrary values

```
def showFruit(fruit: Fruit) = {
  import fruit._
  println(name + "s are " + color)
}
```

- The above references are equivalent to `fruit.name` and `fruit.color`

- Scala's `import` clauses are quite a bit more flexible than Java's
- In Scala, imports
  - may appear anywhere
  - may refer to objects (singleton or regular) in addition to packages
  - let you rename and hide some of the imported members

Imports (3)

- You can import packages themselves, not just their non-package members

```
import java.util.regex

class AStarB {
   // Accesses java.util.regex.Pattern
   val pat = regex.Pattern.compile( "a*b")
 }
```

- Imports in Scala can also rename or hide members
- This is done with an *import selector clause*

```
import Fruits.{Apple, Orange}
```

- This imports just members `Apple` and `Orange` from object `Fruits`

Imports (4)

```
import Fruits.{Apple => McIntosh, Orange}
```

- This also renames the `Apple` object to `McIntosh` (can be accessed with either `Fruits.Apple` or `McIntosh`)

```
import java.sql.{Date => SDate}
```

- This imports the SQL date class as SDate

```
import java.{sql => S}
```

- This imports the `java.sql` package as `S`, so that you can write things like `S.Date`

Imports (5)

```
import Fruits.{_}
```

- This imports all members from object `Fruits`. It means the same thing as import `Fruits._`

```
import Fruits.{Apple => McIntosh, _}
```

- This imports all members from object `Fruits` but renames `Apple` to `McIntosh`

```
import Fruits.{Pear => _, _}
```

- This imports all members of `Fruits` *except* `Pear`

Implicit imports

- Scala adds some imports implicitly to every program

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

- The `java.lang` package contains standard Java classes
- The `scala` package contains the standard Scala library
- The `Predef` object contains many definitions of types, methods, and implicit conversions that are commonly used on Scala programs

Access modifiers (1)

- Members of packages, classes, or objects can be labeled with the access modifiers `private` and `protected`
- Scala has no explicit modifier for public members
- A member labeled `private` is visible only inside the class or object that contains the member definition
- In Scala, this rule applies also for inner classes

```
class Outer {
  class Inner {
    private def f() = { println("f") }
    class InnerMost {
      f() // OK
    }
  }
  (new Inner).f() // error: f is not accessible
}
```

Access modifiers (2)

- Access to `protected` members in Scala is also a bit more restrictive than in Java
- In Scala, a `protected` member is only accessible from subclasses of the class in which the member is defined

```
package p {
  class Super {
    protected def f() = { println("f") }
  }
  class Sub extends Super {
    f()
  }
  class Other {
    (new Super).f()  // error: f is not accessible
  }
}
```

Scope of protection

- Access modifiers in Scala can be augmented with qualifiers
- `private[X]` or `protected[X]` means that access is private or protected "up to" `X`
- `X` designates some enclosing package, class or singleton object
- Qualified access modifiers give you very fine-grained control over visibility
- They enable you to express Java's accessibility notions
- They also let you express accessibility rules that cannot be expressed in Java

Scope of protection (1)

```
package bobsrockets

package navigation {
   private[bobsrockets] class Navigator {
     protected[navigation] def useStarChart() = {}
     class LegOfJourney {
       private[Navigator] val distance = 100
     }
     private[this] var speed = 200
   }
 }
 package launch {
   import navigation._
   object Vehicle {
     private[launch] val guide = new Navigator
   }
 }
```

Scope of protection (2)

- A definition labeled `private[this]` is accessible only from within the same object that contains the definition (*object-private*)
- Any access must not only be within the class, it must also be made from the very same instance

Visibility and companion objects

- Companion object instead of static members
- A class shares all its access rights with its companion object and *vice versa*

```
class Rocket {
   import Rocket.fuel
   private def canGoHomeAgain = fuel > 20
 }

object Rocket {
   private def fuel = 10
   def chooseStrategy(rocket: Rocket) = {
     if (rocket.canGoHomeAgain)
       goHome()
     else
       pickAStar()
   }
   def goHome() = {}
   def pickAStar() = {}
 }
```

Package objects (1)

- Any kind of definition that you can put inside a class can also be at the top level of a package
- Each package is allowed to have one *package object*
- Any definitions placed in a package object are considered members of the package itself
- Package objects are compiled to class files named `package.class` that are located in the directory of the package that they augment. It's useful to keep the same convention for source files (`package.scala`)
- Package objects are frequently used to hold package-wide type aliases (Chapter 20) and implicit conversions (Chapter 21)

Package objects (2)

```
// In file bobsdelights/package.scala
 package object bobsdelights {
   def showFruit(fruit: Fruit) = {
     import fruit._
     println(name + "s are " + color)
   }
 }

// In file PrintMenu.scala
 package printmenu
 import bobsdelights.Fruits
 import bobsdelights.showFruit

object PrintMenu {
   def main(args: Array[String]) = {
     for (fruit <- Fruits.menu) {
       showFruit(fruit)
     }
   }
 }
```