CS 520 Advanced Programming Languages
Fall Semester, 2009
Doc 18 Scala Inheritance, Traits, Packages
Nov 19, 2009

Copyright ©, All rights reserved. 2009 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

References

Programming in Scala, Odersky, Spoon, Venners, Artima Press, 2008

Reading

Programming in Scala, Odersky, Spoon, Venners, Artima Press, 2008
Chapters 10, 11, 12, 13

Nesting

```
class Outer(val a:String) {
    class Inner(val a:Int){
        override def toString = "Inner " + a
    }

    def bar(x: Int) = new Inner(x)

    def foo = {
        class ReallyNested {
            override def toString =
                { new Inner(2).toString + " " + a}
        }
        new ReallyNested
    }
}
```

```
val test = new Outer("why")
val whatIsIt = test.foo
println(whatIsIt)
val x = new test.Inner(4)
```

Output

Inner 2 why

Inheritance

Inheritance

```
class Parent {
    val a = "aa"
    var b = "ab"
    override def toString = "Parent " + a + " " + b

    def foo = {print("Parent foo - "); bar}

    def bar = println("Parent bar")
}

class Child extends Parent {
    override val a = "Child-a"
    override def toString = {super.toString + " " + a}
    override def bar =  println("Child bar")
}
```

```
var test: Parent = new Parent
println(test)
test = new Child
println(test)
test.foo
```

Output

```
Parent aa bb
Parent Child-a bb
Child-a
Parent foo - Child bar
```


this & super have the same meaning as in Java

Can't override var fields

```
class Parent {
      val a = "a"
      var b = "b"
}

class Child extends Parent {
      override val a = "Child-a"
      override var b = 4              //Compile error
}
```

```
class Child extends Parent {
      override val a = "Child-a"
      b = "OK"
}
```

Private & override

```
class Parent {
    private val a = "aa"
    override def toString = "Parent " + a
    }

class Child extends Parent {
    val a = "Child-a"
    override def toString = {super.toString + " " + a}
}
```

```
var test: Parent = new Child
println(test)
```

Output

Parent aa Child-a

Access Level is statically determined

```
class Parent {
      private val a = "aa"
      override def toString = "Parent " + a
      }

class Child extends Parent {
      val a = "Child-a"
      override def toString = {super.toString + " " + a}
}
```

```
var test: Parent = new Child
test.a                  //Compile error

var child = new Child
child.a
```

Calling Parent Constructor

```
class Parent(val a:Int, var b:Int ) {
    override def toString = "Parent " + a + " " + b
}

class Child(a:Int) extends Parent(a + 1,3) {
    override def toString = {super.toString + " C " + a}
}
```

```
var test = new Child(1)
println(test)
```

Output

```
Parent 2 3 C 1
```

Final

```
class Top {
    final val a = 1
    final def foo = println("foo")
    def bar = println("bar")
}

final class Middle extends Top {
    override val a = 2                        //Compile Error
    override def foo = println("middle") //Compile Error
    override def bar = println("bar2")
}

class Bottom extends Middle {}          //Compile Error
```

Object & Inheritance

```
class Parent {
    val a = "aa"
    var b = "bb"
    override def toString =
            "Parent " + a + " " + b

    def foo = {print("Parent foo - "); bar}

    def bar = println("Parent bar")
}

object Child extends Parent {
    override val a = "Child-a"
    override def toString = {super.toString + " C " + a}
    override def bar =  println("Child bar")
}
```

```
println(Child.a)
println(Child.b)
println(Child)
Child.foo
```

Output

```
Child-a
bb
Parent Child-a bb C Child-a
Parent foo - Child bar
```

With Partent Constructor

```
class Parent(var b: String) {
}

object Child extends Parent("a") {
}
```

No object parent classes

```
object Parent {
	val a = "a"
	var b = "b"
	override def toString = "Parent " + a + " " + b
}

object Child extends Parent {			//Compile Error
	override val a = "Child-a"
	override def toString = {super.toString + " C " + a}
}
```

But we have import

Parent.scala

```
object Parent {
    val a = "a"
    var b = "b"
    override def toString =
        "Parent " + a + " " + b

    def foo = {print("Parent foo - "); bar}

    def bar = println("Parent bar")
}
```

Child.scala

```
import Parent._

object Child {
    val a = "Child-a"
    override def toString =
            {Parent.toString + " C " + a}
    def bar =  println("Parent " + b)
}
```

```
println(Child)
Child.bar
```

Output,

```
Parent a b C Child-a
Parent b
```

Abstract classes

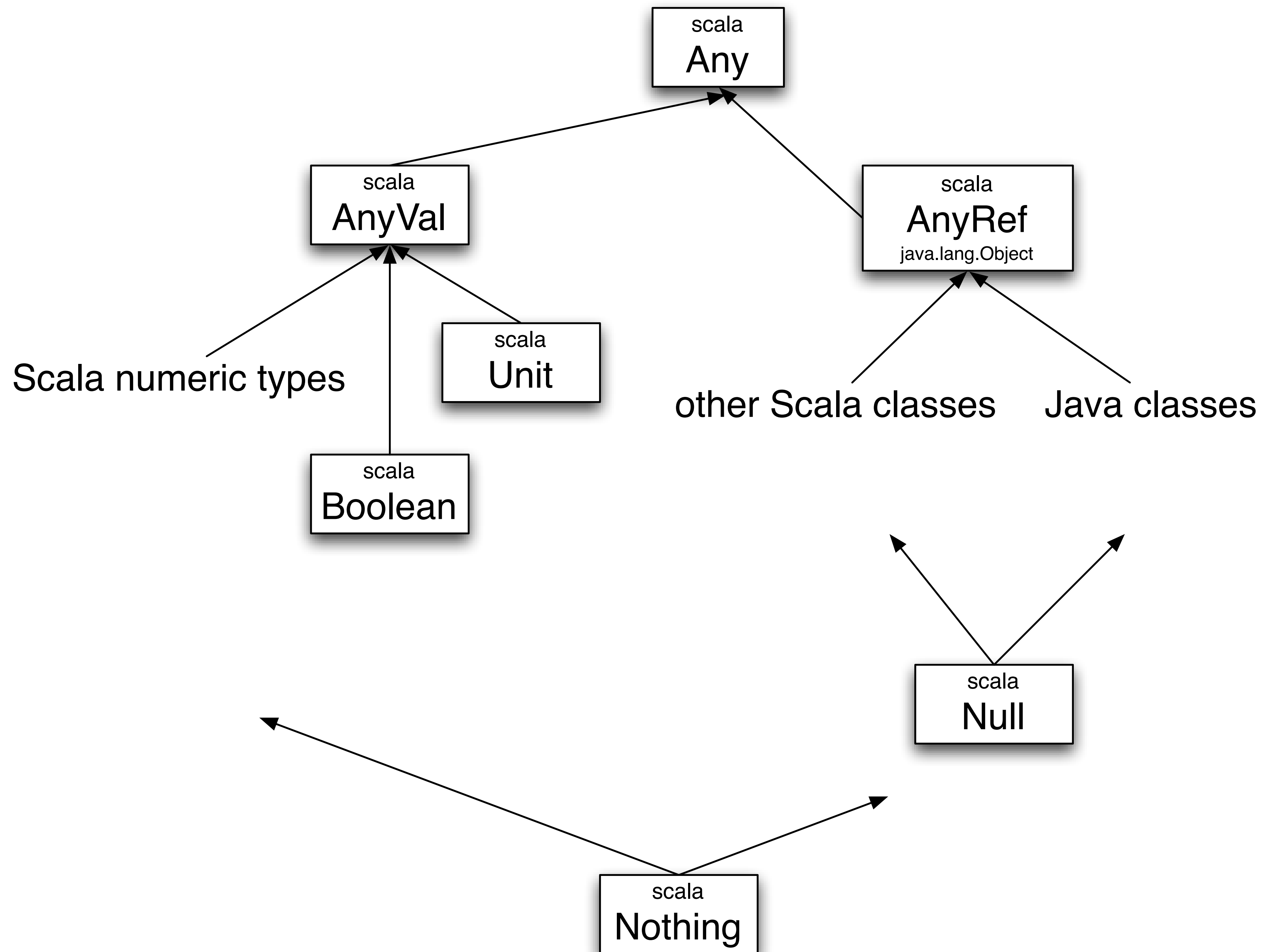
```
abstract class Parent {
    def foo: String
    def bar: String = foo + " bar"
    val a: String
    val b = "bb"
}

class Child extends Parent {
    def foo = "foo " + a + " " + b
    val a = "aa"
}
```

```
val test : Parent = new Child
test.bar
```

Returned
foo aa bb bar

When one overrides an abstract method or field "override" is optional

Scala Class Hierarchy



Null

Subtype of every AnyRef type

```
var test: Array[Int] = null
```

```
var bad: Int = null       //Compile error
```

Nothing

Subtype of every type

No Nothing objects

```
def error(message: String): Nothing =
    throw new RuntimeException(message)

def divide(x: Int, y: Int) :Int =
    if (y != 0 ) x/y
    else error("zero divide")
```

Traits

```
trait Example {
    val a: String
    val b = "bb"
    def bar(x:Int) = x + 1
    def foo(x:String): String
}
```

```
class A extends Example {
    val a = "aa"
    def foo(x:String) = b + x
}

class Parent {
    override def toString = "Parent"
}

class Childs extends Parent with Example {
    val a = "aa"
    def foo(x:String) = b + x
}

object Test extends Example {
    val a = "aa"
    def foo(x:String) = b + x
}
```

Traits & Single inheritance

Class can

- extend one class or trait
- use many traits

Traits are like abstract classes

- No class parameters
- super acts differently

Why Traits?

```
trait Sequence {
    def foreach(f:Int =>Unit):Unit

    def exists(p:Int => Boolean): Boolean = {
        var test = (x: Int) => {if (p(x)) return true}
        foreach(test)
        false
    }

    def map(p:Int => Any): List[Any] = {
        var result = List[Any]()
        var test = (x: Int) => {result = p(x) :: result}
        foreach(test)
        result.reverse
    }
```

```
    def filter(p:Int => Boolean): List[Any] =
{
        var result = List[Any]()
        var test = (x: Int) =>
            {if (p(x)) result = x :: result}
        foreach(test)
        result.reverse
    }
}
```

Multiple Inheritance?

```
class Top {
	println("Top")
	val a = "aa"
	override def toString = "T" + a
}

trait Left extends Top {
	println("Left")
	override val a = "ll"
	override def toString = super.toString + " L " + a
}

trait Right extends Top {
	println("Right")
	override def toString = super.toString + " R " + a
}

class Bottom extends Top with Left with Right {
	println("Bottom")
	override val a = "cc"
	override def toString = super.toString + " B " + a
}
```

```
val test = new Bottom
println(test)
```

Output

```
Top
Left
Right
Bottom
Tcc L cc R cc B cc
```


Top is only inherited once.

Super Order

```
class Top {
	println("Top")
	val a = "aa"
	override def toString = "T " + a
}

trait Side extends Top {
	println("Side")
	override val a = "bb"
	override def toString = super.toString + " S " + a
}

class Bottom extends Top with Side {
	println("Bottom")
	override val a = "cc"
	override def toString = super.toString + " B " + a
}
```

Bottom super calls Side

Side super calls Top

Top is only inherited once.

Super Order

```
class Top {
      override def toString = "T"
}

trait A extends Top { override def toString = super.toString + "A" }
trait B extends A  { override def toString = super.toString + "B"}
trait C extends B  { override def toString = super.toString + "C" }

trait X  extends Top { override def toString = super.toString + "X" }
trait Y extends X  { override def toString = super.toString + "Y"}
trait Z extends Y  { override def toString = super.toString + "Z"}

trait Single {override def toString = "S"}

class Bottom extends Top with Single with C with Z {
      override def toString = super.toString + "B"
}
```

```
val test = new Bottom
println(test)
```

Output
SABCXYZB

Constructor Order

```
class Top {
    println("Top")
}

trait A extends Top { println("A") }
trait B extends A  { println("B") }
trait C extends B  { println("C") }

trait X  extends Top { println("X") }
trait Y extends X  { println("Y") }
trait Z extends Y  { println("Z") }

trait Single {println("Single")}

class Bottom extends Top with Single with C with Z {
    println("Bottom")
}
```

```
val test = new Bottom
```

Output

```
Top
Single
A
B
C
X
Y
Z
Bottom
```

Stackable Traits

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

class BasicPutter extends Putter {
	def put(x:Int) {println(x)}
}

trait Squaring extends Putter {
	abstract override def put(x:Int) {super.put(x * x)}
}

trait Increasing extends Putter {
	abstract override def put(x:Int) {super.put(x + 1)}
}
```

Using Stackable

```
var test = new BasicPutter with Squaring
test.put(2)
```

Output

4

```
var filterFirst = new BasicPutter with Increasing with Squaring
filterFirst.put(2)
```

Output

5

```
var filterLast = new BasicPutter with Squaring with Increasing
filterLast.put(2)
```

Output

9

Traits & Types

```
trait Foo {
    def bar = println("bar")
}

class Bar extends Foo {
    def test = println("test")
}
```

```
var a : Foo = new Bar
a.bar
a.test                //Compile Error

var b : Bar = new Bar
b.bar
b.test
```

Anonymous Classes

```
trait Foo {
    val x: String
    def bar:Unit
}
```

```
abstract class Foo {
    val x: String
    def bar:Unit
}
```

```
val test = new Foo{val x = "a"; def bar = println(x)}
test.bar
test.getClass()
```

When to use

If behavior will not be reused
- Class

If it might be reused in multiple unrelated classes
- Trait

If you want to inherit it in Java Code
- Abstract class

If you plan to distribute it as compiled library
- Abstract class

If efficiency is important
- Class

Otherwise
- Trait

Packages & Imports

Java Like Package Syntax

File example.scala

```
package roger.sample

object test extends Application {
    println("roger.sample.test")
}

class Foo {
        val x = 1
}
```


The interpreter does not process packages

Scala General Syntax

```
package roger {
    object Example extends Application {
      println("roger.example")
    }

      class Foo {
            val x = 1
      }
  package test {
    object Example extends Application {
      println("roger.test.Example")
    }
  }
}
```

Accessing without import

roger.Example

new roger.Foo

roger.test.Example

import syntax

	makes available without qualification
import p._	all members of p (like import p.* in Java)
import p.x	the member x of p
import p.{x => a}	the member x of p renamed as a
import p.{x, y}	the members x and y of p
import p1.p2.z	the member z of p2, itself member of p1

Always imported

the package java.lang,
the package scala,
and the object scala.Predef.

Food package

```
package Food

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", "red")
    val menu = List(Apple, Orange, Pear)
}
```

```
No import
Food.Fruits.menu
```

```
import Food.Fruits.menu
menu(1)
```

```
import Food.Fruits._
Apple
menu(1)
```

```
import Food._
Fruits.Apple
Fruits.menu(1)
```

More imports

```
import Food.Fruits.{Apple,menu=>select}
Apple
select(1)
Food.Fruits.menu(1)
```

```
import Food.Fruits.{Apple => _,  _}
Apple           //Compile error
Orange
menu(1)
```

```
import Food._
import Fruits.Apple
Apple
Fruits.Orange
```

Imports can go anywhere

```
import Food.Fruit

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

showFruit(Food.Fruits.Apple)
```