Object Oriented Scala

Object Oriented Scala

- Scala supports the OO paradigm
- Every value is an object
  - Compiler uses JVM primitive types when appropriate
- Scala follows the Java approach to OO
  - Classes and traits define object types
  - Singleton objects available
  - Single inheritance of classes
  - Mixin inheritance provided through traits

```
scala> 350.toString
res0: String = 350

scala> :type 350
Int

scala> 350 + 1
res1: Int = 351

scala> (350).+(1)
res2: Int = 351
```

Defining a Class

- Easy to define a new class
  - Instantiate using new
  - REPL provides :javap command to examine bytecodes

```
scala> class MyClass
defined class MyClass

scala> val myObj = new MyClass
myObj: MyClass = MyClass@14bf9759

scala> myObj.toString
res3: String = MyClass@14bf9759
```

```
scala> :javap -c MyClass
Compiled from "<console>"
public class MyClass {
  public MyClass();
    Code:
       0: aload_0
       1: invokespecial #9                  // Method java/lang/Object."<init>":()V
       4: return
}
```


Constructor

- Every class has a "primary" constructor
  - Code embedded between { … }

```
class MyClass {
  println("Building a MyClass")
}

scala> val myObj = new MyClass
Building a MyClass
myObj: MyClass = MyClass@5db45159
```

- Class parameters may be defined
  - Accessible in constructor code

```
class Message( head: String, body: String ) {
  println(s"$head $body")
}

scala> val myMsg = new Message("Hello", "world")
Hello world
myMsg: Message = Message@5c669da8
```

Constructor

- Other constructors may be defined
  - Specify as method called this
  - Allows chaining, as per Java
  - Must eventually execute primary constructor

```
class Message( head: String, body: String ) {
  def this( bd: String ) = this( "Hello", bd )
  println( s"Primary: $head $body")
 }

scala> val myMsg = new Message("Class")
Primary: Hello Class
myMsg: Message = Message@ff6077
```


Class Properties

- Class may have immutable or mutable data properties
  - Use val or var as appropriate

```
class Message ( head: String, body: String ) {
  val msg = s"$head $body"
}


scala> val myMsg = new Message("Hello", "world")
myMsg: Message = Message@6f9ad11c

scala> myMsg.msg
res9: String = Hello world

scala> myMsg.msg = "Goodnight everyone"
<console>:13: error: reassignment to val
       myMsg.msg = "Goodnight everyone"
                 ^
```

Class Parameters and Properties

- Class parameters are not stored as properties
  - Only available in primary constructor

```
class Message ( head: String, body: String ) {
  val msg = s"$head $body"
}


scala> val myMsg = new Message("Hello", "world")
myMsg: Message = Message@6f9ad11c

scala> myMsg.head
<console>:14: error: value head is not a member of Message
       myMsg.head
             ^
```


Class Parameters and Properties

- Use val or var to promote parameters to properties

```
class Message( val head: String, val body: String ) {
  val msg = s"$head $body"
}


scala> val myMsg = new Message("Hello", "world")
myMsg: Message = Message@5f59ea8c

scala> myMsg.head
res12: String = Hello

scala> myMsg.msg
res14: String = Hello world
```

Adding Methods

- Methods define calculations on instances of the class
  - Use def to define methods

```
class Arith ( val i: Int, val j: Int ) {
  def add = i + j
  def sub: Int = i - j
  def addAndMult( by: Int ) = (i + j) * by
}


scala> val arith = new Arith( 5, 2 )
arith: Arith = Arith@6ac4c3f7

scala> arith.add
res15: Int = 7

scala> arith.addAndMult( by = 2 )
res16: Int = 14
```

Compiler infers return type

Explicit return type (recommended for public interface)

Parameter types must be specified

Parameters may be passed using their name

val and def – the Uniform Access Principle

- val and def are similar syntactically
  - If def has no parameters

```
class VClass {
  val now = new java.util.Date().toString
}


scala> val vObj = new VClass
vObj: VClass = VClass@16d07cf3

scala> vObj.now
res17: String = Wed Aug 10 11:45:46 CST 2016

scala> vObj.now
res18: String = Wed Aug 10 11:45:46 CST 2016
```

val and def – the Uniform Access Principle

- val causes expression to be evaluated *eagerly* (or *strictly*)
  - def causes expression to be evaluated *lazily*
  - Caller should not notice different syntax

```
class DClass {
  def now = new java.util.Date().toString
}


scala> val dObj = new DClass
dObj: DClass = DClass@7b53b1ad

scala> dObj.now
res20: String = Wed Aug 10 11:46:09 CST 2016

scala> dObj.now
res21: String = Wed Aug 10 11:46:16 CST 2016
```


A Complete Example

- Complex numbers:
  - A simple (sic) implementation

Default value for property

```
class Complex ( val re: Int, val im: Int = 0 ) {

  def + ( that: Complex ) =
              new Complex( this.re + that.re, this.im + that.im )

  def - ( that: Complex ) =
              new Complex ( this.re - that.re, this.im - that.im )

  override def toString = s"${re} + ${im}i"

}
```

Keyword is mandatory when overriding method from base class

A Complete Example

- Single argument methods can be invoked as operators
  - "Normal" precedence applies for methods named using operator characters

```
scala> val c1 = new Complex(1,2)
c1: Complex = 1 + 2i

scala> val c2 = new Complex(3)
c2: Complex = 3 + 0i

scala> c1.+(c2)
res23: Complex = 4 + 2i

scala> c1 + c2
res24: Complex = 4 + 2i
```

REPL uses toString method for display

A Complete Example

- Method with no arguments can be written as postfix

```
scala> val c1 = new Complex(1,2)
c1: Complex = 1 + 2i

scala> c1 toString
res25: String = 1 + 2i
```

- Define unary_... methods for prefix +, -, !, ~

```
...
  def unary_- = new Complex( -re, -im )
...

scala> -c1
res26: Complex = -1 + -2i
```

Equality

- Scala == and != operators check state equality
  - Compiler translates to call equals() method for compatibility with Java
- Use eq and ne methods to check identity equality
  - For reference types (see later)

```
scala> 3 == 3
res27: Boolean = true

scala> new String("foobar") == new String("foobar")
res28: Boolean = true

scala> new String("foobar") eq new String("foobar")
<console>:12: warning: comparing a fresh object using `eq' will always yield false
       new String("foobar") eq new String("foobar")
                            ^
res29: Boolean = false
```


Singleton Objects

- Class and single automatically created instance
  - Cannot create further instances

```
object MessageObj {
  val hd = "Hello"
  val bd = "World"
  def showMessage = s"$hd $bd"
}


scala> MessageObj.hd
res34: String = Hello

scala> MessageObj.showMessage
res35: String = Hello World
```

Objects may be used to gather together related but not tightly bound data and functions, they are sometimes known as "modules"

```
scala> new MessageObj
<console>:12: error: not found: type MessageObj
       new MessageObj
           ^
```

Companion Objects

- Singleton object compiled in same unit as a class is called a Companion Object
  - Has access to all private items in the class
  - Can be viewed as a container for static data and methods

```
class Message ( val hd: String, val bd: String ) {
  Message.count += 1
}
object Message {
  var count = 0
}
```

```
scala> val m1 = new Message ( "hello", "world" )
m1: Message = Message@3292eff7

scala> val m2 = new Message("I see", "no ships")
m2: Message = Message@59014efe

scala> Message.count
res37: Int = 2
```

- Use :paste mode to have REPL compile both together

Object Factories

- Uses companion object and associated apply() method
  - Compiler rewrites () "operator" to call this method

```
class Message private ( val hd: String, bd: String )

object Message {
  var count = 0
  def apply( h: String, b: String ): Message = {
    count += 1
    new Message(h, b)
  }
}
```

```
scala> val m1 = new Message("from", "the constructor")
<console>:13: error: constructor Message in class Message
                                        cannot be accessed in object $iw
       val m1 = new Message("from", "the constructor")
                ^

scala> val m1 = Message("from", "the factory")
m1: Message = Message@18b45500
```

Case Classes

- Useful for representing types that present "value" semantics
- Compiler provides boilerplate implementation of methods
  - equals, hashcode, toString, copy
- Class parameters are automatically promoted to properties
- Companion object and apply method provided for construction

```
scala> case class Complex ( re: Int, im: Int )
defined class Complex

scala> val c1 = Complex(1,2)
c1: Complex = Complex(1,2)

scala> c1 == Complex(1,2)
res39: Boolean = true
```


Case Classes

- Copy method provides safe copying
  - Possibly altering properties
- Case classes not universally applicable
  - Additional bytecode generated
  - Case class cannot inherit from another case class
  - Normally used for value objects, not service objects

```
scala> c1.copy() == c1
res41: Boolean = true

scala> c1.copy() eq c1
res42: Boolean = false

scala> val c3 = c1.copy( re = 4 )
c3: Complex = Complex(4,2)
```

Case Classes and Pattern Matching

- Case classes are essentially structured values
- Pattern matching allows us to decompose the structure into its components
- Implemented using compiler provided unapply method
  - "deconstruction"

```
scala> val c1 = Complex(2,3)
c1: Complex = Complex(2,3)

scala> c1 match {
     |   case Complex(re, im) => s"Real: $re, Im: $im"
     | }
res7: String = Real: 2, Im: 3
```


The Extractor Pattern

- Can we extract components from a non-case class?

```
class Person ( val first: String,
               val last: String,
               val age: Int) {
  override def toString = s"$first $last: $age"
}

scala> val p1 = new Person("John", "Doe", 21)
p1: Person = John Doe: 21

scala> p1 match {
     |   case Person(f, l, a) => s"$f $l is $a years old"
     | }
<console>:17: error: not found: value Person
         case Person(f, l, a) => s"$f $l is $a years old"
              ^
```

The Extractor Pattern

- Use companion object for class
  - Define unapply method

```
class Person ( val first: String, val last: String, val age: Int) {
  override def toString = s"$first $last: $age"
}
object Person {
  def unapply ( p: Person ): Option[(String, String, Int)] =
    Some(p.first, p.last, p.age)
}

scala> val p1 = new Person("John", "Doe", 21)
p1: Person = John Doe: 21

scala> p1 match {
     |   case Person(f, l, a) => s"$f $l is $a years old"
     | }
res9: String = John Doe is 21 years old
```


The Extractor Pattern

- Unapply method can be used for generalised extraction of components from a structured value
  - E.g. Regular Expression Capture Groups

```
scala> val Person(f, l, a) = p1
f: String = John
l: String = Doe
a: Int = 21

scala> f
res10: String = John
```

```
scala> val dateMatcher = """^(\d\d)/(\d\d)/(\d\d)$""".r
dateMatcher: scala.util.matching.Regex =
                                    ^(\d\d)/(\d\d)/(\d\d)$

scala> val dateMatcher(d, m, y) = "01/02/16"
d: String = 01
m: String = 02
y: String = 16
```

```
scala> val Complex( realPart, imaginaryPart ) = c1
realPart: Int = 2
imaginaryPart: Int = 3
```

Packages

- Define namespaces, as in Java
- More flexible
  - Can be nested
  - Many packages per source file

```
package myPackage {
  class AA

  package mySubPackage {
    class AB1
    class AB2
  }
}

class C

object foo extends App {
  val a = new myPackage.AA
  val b = new myPackage.mySubPackage.AB1
  val c = new C
}
```


Package Objects

- Allow definition of package-global data and methods

package.scala

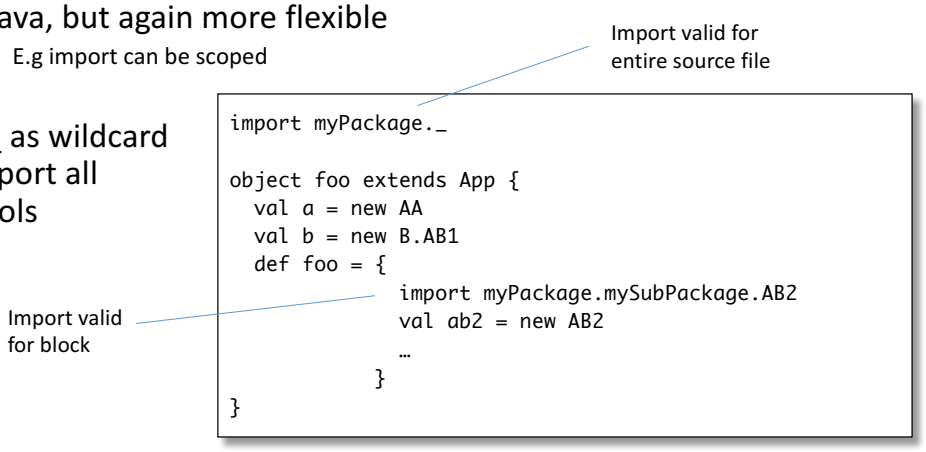
```
package object myPackage {
  val packageGlobalVal = 100;
  def packageGlobalDef = "Hello There"
}
```

- Use like any other package members

```
object MyProg extends App {
  val myVal = myPackage.packageGlobalVal
  println(myPackage.packageGlobalDef)
}
```

Import

- Allows symbols to be used without package qualification
- Like Java, but again more flexible
  - E.g import can be scoped
- Use _ as wildcard to import all symbols



Import

- Import multiple symbols at one time

```
import java.io.{ InputReader, OutputWriter }
```

- Import symbol and create alias
  - Helps deal with name clashes

```
import java.util.Date
import java.sql.{ Date => SqlDate }
```

- Import allowed from entities other than packages
  - package objects
  - singleton objects

Import

- Default imports performed by the compiler

```
import java.lang._
import scala._
import scala.Predef._
```

- scala.Predef defines many useful functions
  - println
  - printf
  - Various read… functions
  - assert
  - require

```
scala> require ( 1 == 0, "WTF???" )
java.lang.IllegalArgumentException: requirement
failed: WTF???
        at
scala.Predef$.require(Predef.scala:233)
        at .<init>(<console>:8)
        at .<clinit>(<console>)
…
```


Visibility

- By default all class members (properties and methods) are public
- Can be restricted to private
  - Visible in containing type only
- …or protected
  - Visible in containing type and subtypes

```
class Person ( val firstName: String,
               val lastName: String,
               private val age: Int ) {

  protected val name = s"${firstName} ${lastName}"
}
```

Visibility

- Visibility restriction can be qualified
- Restricted to package

```
package myPackage

class Person ( val firstName: String,
               val lastName: String,
               private [myPackage] val age: Int ) {

   protected val name = s"${firstName} ${lastName}"
}
```

- Restricted to instance

```
class Person ( val firstName: String,
               val lastName: String,
               private [this] val age: Int ) {

   protected val name = s"${firstName} ${lastName}"
}
```


Inheritance

- Scala classes may inherit from one other class
  - Standard JVM model
  - Non-private members inherited
  - Non-final members may be overridden

```
class Animal ( val name: String ) {
  def eat = println("Hungry....")
  private def mySecret = println("This is my secret")
}

class Lion (n: String ) extends Animal( n ) {
  def roar = println(name + " says Grrr")
  override def eat = println("Rare steak please")
}
```

Invoke superclass constructor with arg

Preventing Subclassing or Overriding

- Use final to prevent a class from being extended

```
scala> final class Animal
defined class Animal

scala> class Lion extends Animal
<console>:12: error: illegal inheritance from final class Animal
       class Lion extends Animal
```

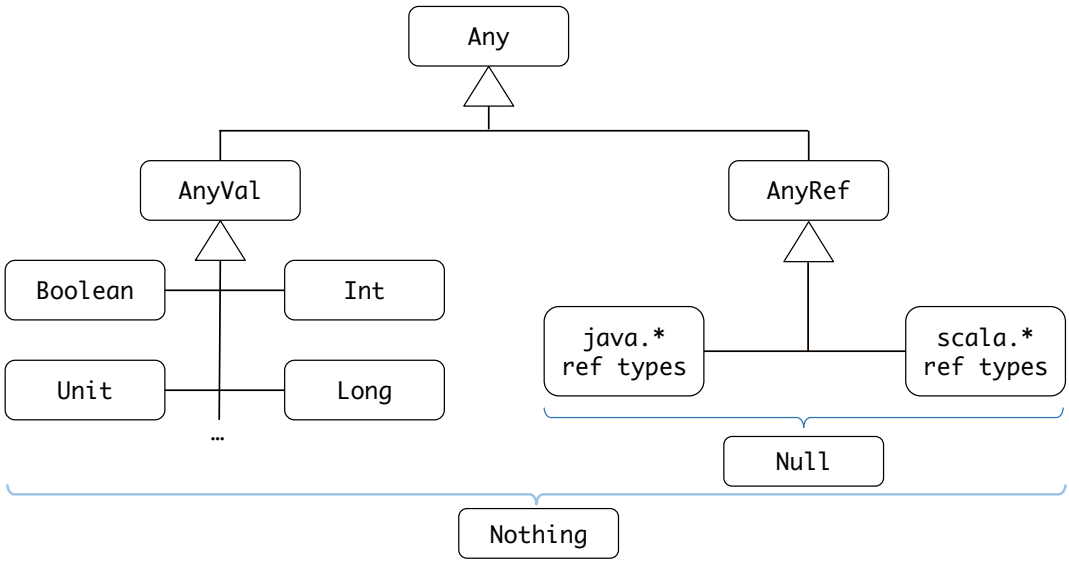

Preventing Subclassing or Overriding

- …or to prevent a member being overridden

```
scala> class Animal { final def eat = println("Hungry") }
defined class Animal

scala> class Lion extends Animal {
     |   override def eat = println("WTF??")
     | }
<console>:13: error: overriding method eat in class Animal of type => Unit;
 method eat cannot override final member
         override def eat = println("WTF??")
```

Scala Class Inheritance Hierarchy



Value Types

- Subtypes of AnyVal
- Present as objects
  - Act as wrappers around JVM primitive types
  - Compiler removes object wrapper when possible
  - Balances efficiency with object capabilities
- New value types may be defined
  - Must wrap a single JVM primitive type
  - Must extend AnyVal

```
class Foo ( i: Int ) {

  def bar = i + 1

}
```

```
scala> :javap -c Foo
Compiled from "<console>"
public class Foo {
  public int bar();
    Code:
       0: aload_0
       1: getfield      #11  // Field i:I
       4: iconst_1
       5: iadd
       6: ireturn

  …
}
```

Abstract Classes

- Class need not provide full implementation of all properties
- Such a class must be marked as abstract
  - Cannot be instantiated

```
abstract class Animal {
  def eat: String
}

scala> new Animal
<console>:13: error: class Animal is abstract; cannot be instantiated
       new Animal
       ^
```

Argument types and result type should be specified

```
class Lion extends Animal {
  override def eat = "Yum"
}

scala> new Lion().eat
res48: String = Yum
```

Keyword not mandatory but advised

Overriding and the Uniform Access Principle

- def may be overridden with a val
  - Subclass causes the property value to become stable
- val may *not* be overridden by a def

```
class Person {
  def age: Int = // Some calculation
}

scala> val p = new Person
p: Person = Person@1c59d3ae

scala> p.age
res23: Int = 22
```

```
class DorianGray extends Person {
  override val age: Int = 21
}
```

Traits

- Trait is similar to Java 8 interface
  - May include implementation
  - May not define construction parameters
  - Use trait to interact with Java interfaces
- Scala class may "inherit" or "mix in" several traits
- Use trait to
  - Advertise public interface
  - Add behaviour to existing type

Why Traits?

- The case for multiple inheritance is normally based on mixins
  - Can help produce a more accurate model

```
scala> abstract class Animal
defined class Animal

scala> class Bird extends Animal { def fly = "wheee" }
defined class Bird

scala> class Fish extends Animal { def swim = "splash" }
defined class Fish

scala> class Seagull extends ?????????

scala> class Penguin extends ?????????
```

Why Traits?

- Start with (abstract) base class
  - Mix in required functionality

```
scala> abstract class Animal
defined class Animal

scala> trait CanSwim { def swim = "Splash" }
defined trait CanSwim

scala> trait CanFly { def fly = "Wheeee" }
defined trait CanFly

scala> class Fish extends Animal with CanSwim
defined class Fish

scala> class Seabird extends Animal with CanFly with CanSwim
```

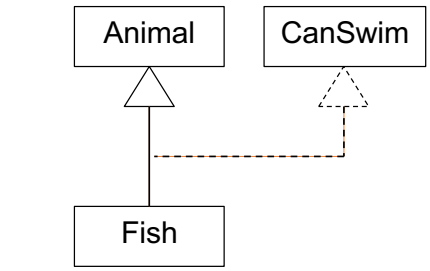
First supertype can be class or trait

Use "with" for subsequent supertypes

Mixins and the super Reference

- Multiple inheritance must resolve the meaning of "super" in the derived class
- Scala's mixin inheritance based on traits forms a single chain of supertypes:
- `Fish`, `CanSwim`, `Animal`, `AnyRef`, `Any`

```
class Fish extends Animal with CanSwim
```

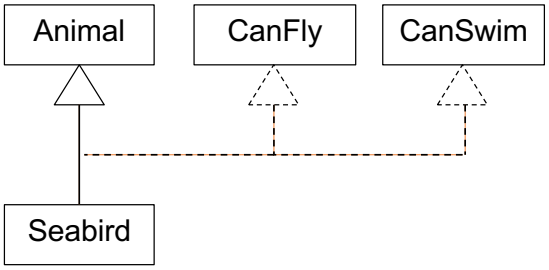


Mixins and the super Reference

- Multiple inheritance must resolve the meaning of "super" in the derived class
- Scala's mixin inheritance based on traits forms a single chain of supertypes:

- `Seabird`, `CanSwim`, `CanFly`, `Animal`, `AnyRef`, `Any`

```
class Seagull extends Animal with CanFly with CanSwim
```

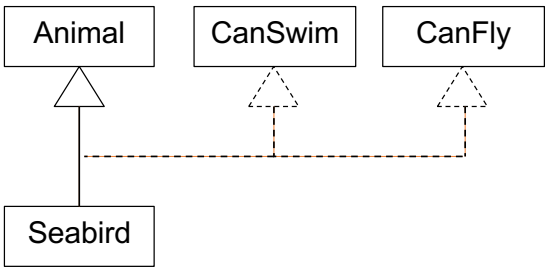


Mixins and the super Reference

- Multiple inheritance must resolve the meaning of "super" in the derived class
- Scala's mixin inheritance based on traits forms a single chain of supertypes:

- `Seabird`, `CanFly`, `CanSwim`, `Animal`, `AnyRef`, `Any`

```
class Seagull extends Animal with CanSwim with CanFly
```



Multiple Implementations in Supertypes

- Ambiguity must be removed

(new C).bar???

```
scala> trait Foo { def bar = "FooBar" }
defined trait Foo

scala> trait Bar { def bar = "BarBar" }
defined trait Bar

scala> class C extends Foo with Bar
<console>:13: error: class C inherits conflicting members:
  method bar in trait Foo of type => String  and
  method bar in trait Bar of type => String
(Note: this can be resolved by declaring an override in class C.)
       class C extends Foo with Bar

scala> class C extends Foo with Bar {
     |   override def bar = super[Foo].bar
     | }
defined class C

scala> (new C).bar
res49: String = FooBar
```


Sealed Types

- Sealed types are types that can only be extended in the same compilation unit (source file)
  - Normally abstract
  - Allows control over subtypes
  - Subtypes normally final
  - Used to create Algebraic Sum Data Types

```
sealed trait Expression

final case class Const(v: Int) extends Expression
final case class Neg(e: Expression) extends Expression
final case class Add ( l: Expression, r: Expression ) extends Expression
```

```
10 + ( - ( 3 + 4 ) )
```

```
scala> val expr = Add ( Const(10), Neg ( Add( Const(3), Const(4) ) )  )
```

Sealed Types

- Often used to define DSLs
  - Pattern Matching can be used to build an interpreter for the DSL

```
object ExpressionInterpreter {

  def eval ( e: Expression ): Int = e match {
    case Const(c) => c
    case Neg(e) => - eval(e)
    case Add(l, r) => eval(l) + eval(r)
  }

}
```

```
scala> val e1 = Add( Const(10), Neg( Add( Const(3), Const(4) ) ) )
e1: Add = Add(Const(10),Neg(Add(Const(3),Const(4))))

scala> ExpressionInterpreter.eval(e1)
res13: Int = 3
```


Sealed Types

- Sealed type hierarchy allows compiler to perform "exhaustiveness checking" in pattern match
  - Compiler knows all possible subtypes
  - Error not to include all possibilities in match
  - Alternative is MatchError exception at runtime

```
object ExpressionInterpreter {

  def eval ( e: Expression ): Int = e match {
    case Const(c) => c
    case Add(l, r) => eval(l) + eval(r)
  }

}
```

Match must include the Neg case otherwise MatchError exception may occur