Advance in Object-Orientation

Overrides, companion objects, case classes, ...

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Advance in Object-Orientation
Override Keyword

Classes and traits

- can declare **abstract** members: fields, methods and types;
- abstract member must be defined by a derived class or trait before an instance could be created.

To override a member Scala requires the **override** keyword

- optional for overriding abstract members
- it can't be used when you are not overriding a member.

Some benefits

1. it catches misspelled members that were intended to be overrides;
2. it avoids undesired overrides, i.e., member clashes in derived classes/-traits;

Java's `@Override` helps with the former but it is useless with the latter.

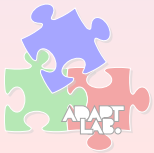
Advance in Object-Orientation
Override Keyword (Cont'd)

Overriding Concrete & Abstract Methods

- it behaves as expected
- `super` keyword permits to access to the parent, which is the aggregation of the parent class and any mixed-in traits.

Linearization Algorithm

1. put the actual type of the instance as the first element
2. right to left, compute the linearization of each type, appending its linearization to the cumulative linearization
3. left to right, remove any type that re-appears to the right
4. append **ScalaObject**, **AnyRef**, and **Any**.



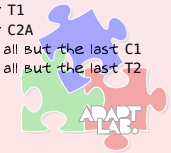
Advance in Object-Orientation
Details on the Linearization Algorithm

```
class C1 { def m = List("C1") }
trait T1 extends C1 { override def m = { "T1" :: super.m } }
trait T2 extends C1 { override def m = { "T2" :: super.m } }
trait T3 extends C1 { override def m = { "T3" :: super.m } }
class C2A extends T2 { override def m = { "C2A" :: super.m } }
class C2 extends C2A with T1 with T2 with T3 { override def m = { "C2" :: super.m } }

def linearization(obj: C1, name: String) = {
  val lin = obj.m ::: List("ScalaObject", "AnyRef", "Any")
  println(name + ": " + lin)
}
```

```
scala> linearization(new C2,"C2 ")
C2 : List(C2,T3,T1,C2A,T2,C1,ScalaObject,AnyRef,Any)
```

#	Linearization	Description
1	C2	add the type of the instance
2	C2, T3, C1	Add the linearization for T3
3	C2, T3, C1, T2, C1	Add the linearization for T2
4	C2, T3, C1, T2, C1, T1, C1	Add the linearization for T1
5	C2, T3, C1, T2, C1, T1, C1, C2A, T2, C1	Add the linearization for C2A
6	C2, T3, T2, T1, C2A, T2, C1	Remove duplicates of C1; all but the last C1
7	C2, T3, T1, C2A, T2, C1	Remove duplicates of T2; all but the last T2
8	C2, T3, T1, C2A, T2, C1, ...	Finish!



Overriding Fields in Traits

```
trait T1 { val name = "T1" }
class Base
class ClassWithT1 extends Base with T1 { override val name = "ClassWithT1" }
val c = new ClassWithT1()
println(c.name)
class ClassExtendsT1 extends T1 { override val name = "ClassExtendsT1" }
val c2 = new ClassExtendsT1()
println(c2.name)
```

```
[DING!]cazzola@surtur:~/lp/scala>scala val-override.scala
ClassWithT1
ClassExtendsT1
```

Overriding Fields in Classes

```
class C1 { val name = "C1"; var count = 0 }
class ClassWithC1 extends C1 { override val name = "ClassWithC1"; count = 1 }
val c = new ClassWithC1()
println(c.name, c.count)
```

```
[15:10]cazzola@surtur:~/lp/scala>scala val-override-inclass.scala
(ClassWithC1,1)
```



```
import java.io._
abstract class BulkReader {
  type In
  val source: In
  def read: String
}
class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read = source
}
class FileBulkReader(val source: File) extends BulkReader {
  type In = File
  def read = {
     val in = new BufferedInputStream(new FileInputStream(source))
     val numBytes = in.available()
     val bytes = new Array[Byte](numBytes)
     in.read(bytes, 0, numBytes)
     new String(bytes)
  }
}
println( new StringBulkReader("Hello Scala!").read )
println( new FileBulkReader(new File("BulkReader.scala")).read )
```

```
[15:34]cazzola@surtur:~/lp/scala>scala BulkReader.scala
Hello Scala!
import java.io._
     ...
```


A Class (or type) and an object declared in the same package with the same name are called companion class (or type) and object respectively.

- no namespace collision since
- class name is stored in the type namespace and
- object name is stored in the term namespace

Apply

- when an instance is followed by a list of zero or more parameters between parentheses the compiler invokes apply
- this is true either for an **object** or an instance of a **class** defining apply.

```
type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

This permits to create a Pair as

```
val p = Pair(1, "one")
```


Unapply

- it is used to extract the constituent parts of an instance

```
object Twice { def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None }
object TwiceTest extends Application {
  val x = 42; x match { case Twice(n) => Console.println(n) }
}
```

```
scala> TwiceTest
21
res8: TwiceTest.type = TwiceTest$@3b2601c
```

Apply & UnapplySeq for collections

- they can be used to build a collection from a variable argument list or to extract the first few elements from a collection

```
object L2 {
   def unapplySeq(s: String) : Option[List[String]] = Some(s.split(",").toList)
   def apply(stuff: String*) = stuff.mkString(",")
}
```

```
scala> val x2 = L2("4","5","6")
x2: String = 4,5,6
scala> val L2(d,e,f) = x2
d: String = 4
e: String = 5
f: String = 6
```

Advance in Object-Orientation
Case Classes

Case classes are

- classes which export their constructor parameters and which provide a recursive decomposition mechanism via pattern matching.

E.g., the lambda terms

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

- its constructor parameters are treated as public values and can be accessed directly

```
scala> val x = Var("x")
scala> Console.println(x.name)
x
```

- equals, hashCode and toString methods based on the constructor arguments are generated (note == delegates to equals)

```
scala> val x1 = Var("x")
scala> val x2 = Var("x")
scala> val y1 = Var("y")
scala> println("" + x1 + " == " + x2 + " => " + (x1 == x2))
Var(x) == Var(x) => true
scala> println("" + x1 + " == " + y1 + " => " + (x1 == y1))
Var(x) == Var(y) => false
```

- a copy method is generated as well.

Advance in Object-Orientation
Case Classes (Cont'd)

Case classes are particularly useful with pattern matching.

```
object TermTest extends Application {
  def printTerm(term: Term) {
    term match {
      case Var(n) => print(n)
      case Fun(x, b) => print("λ" + x + "."); printTerm(b)
      case App(f, v) => Console.print("("); printTerm(f)
        print(" "); printTerm(v); print(")")
    }
  }

  def isIdentityFun(term: Term): Boolean = term match {
    case Fun(x, Var(y)) if x == y => true
    case _ => false
  }

  val id = Fun("x", Var("x"))
  val t = Fun("x", Fun("y", App(Var("x"), Var("y"))))
  printTerm(t); println; println(isIdentityFun(t))
  printTerm(id); println; println(isIdentityFun(id))
}
```

```
[15:16]cazzola@surtur:~/lp/scala>scala TermTest.scala
λx.λy.(x y)
false
λx.x
true
```

References

- Martin Odersky and Matthias Zenger.
  Scalable Component Abstractions.
  In Richard P. Gabriel, editor, Proceedings of 19th ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05), pages 41–57, San Diego, CA, USA, October 2005. ACM Press.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable Units of Behaviour.
  In Luca Cardelli, editor, Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03), Lecture Notes in Computer Science 2743, pages 248–274, Darmstadt, Germany, July 2003. Springer.
- Venkat Subramaniam.
  Programming Scala.
  The Pragmatic Bookshelf, June 2009.
- Dean Wampler and Alex Payne.
  Programming Scala.
  O'Reilly, September 2009.