Advance in Object-Orientation
Overrides, companion objects, case classes, ...

Walter Cazzola
Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano
e-mail: cazzola@dico.unimi.it

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.

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

Details on the Linearization Algorithm

class C1 { def n = List("C1") }
trait T1 extends C1 { override def n = { "T1" :: super.n } }
trait T2 extends C1 { override def n = { "T2" :: super.n } }
trait T3 extends C1 { override def n = { "T3" :: super.n } }
trait C2A extends C1 with T2 with T3 { override def n = { "C2A" :: super.n } }
def linearization(obj: C1, name: String) = {
  val lin = obj.n :: List("ScalaObject", "AnyRef", "Any")
  println(name + " : " + lin)
}
scala> linearization(new C2, "C2 ")
C2 : List(C2, T3, T1, C2A, T2, C1, ScalaObject, AnyRef, Any)

Linearization

1 C2
2 C2, T3, C1
3 C2, T3, C1, T2, C1
4 C2, T3, C1, T2, C1, T1, C1
5 C2, T3, C1, T2, C1, T1, C1, T2, C1
6 C2, T3, T2, C1, T2, C1
7 C2, T3, T1, C2A, T2, C1
8 C2, T3, T1, C2A, T2, C1, ...

Description
- add the type of the instance
- add the linearization for T3
- add the linearization for T2
- add the linearization for T1
- add the linearization for T2A
- remove duplicates of C1, all but the last C1
- remove duplicates of T2, all but the last T2
- done!
Overriding Fields in Traits

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

class ClassExtendsT1 extends T1 {
  override val name = "ClassExtendsT1"
}
class ClassWithT1 extends T1 {
  override val name = "ClassWithT1"
  val c2 = new ClassExtendsT1()
  println(c2.name)
}

Overriding Fields in Classes

class C1 {
  val name = "C1" ;
  var count = 0
}
class ClassWithC1 extends C1 {
  override val name = "ClassWithC1";
  count = 1
}
class TypeC1 {
  val C1 = new C1()
  println(C1.name, C1.count)
}

class ClassWithC1 extends C1 {
  override val name = "ClassWithC1";
  count = 1
}
class ClassC1 {
  val c = new ClassWithC1()
  println(c.name, c.count)
}

Companion Objects

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

This permits to create a Pair as

val p = Pair(1, "one")

Companion Objects (Cont'd)

Unapply

- it is used to extract the constituent parts of an instance
- these methods are also called extractors

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 => stuff.toString()
}

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

E.g., the lambda terms

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

• a copy method is generated as well

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

Case classes are particularly useful with pattern matching.

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

References