CSE250 Week 3, Continued

Case, Inheritance, and Generic Polymorphism

Kenneth W. Regan

February 18, 2022

Preface About Submissions

It is supposed to be possible to do any of these with `MaxWords.scala`:

- Have it in your project root folder (where `output.txt` appears?). Not required to call the root folder itself `MaxWords`; e.g. `CSE250assgt1` is an equally fine name in your IDE area.
- OR in `src/`
- OR in `src/main/scala/` (which is where I get it unless...)
- OR in `src/main/scala/MaxWords/` (...I give IntelliJ a subname)
- OR anywhere else in your project hierarchy—we will find it... (but after that is what I'm having unexpected difficulty with)

Not OK: ignoring "`words.txt`" and reading from, say, `data/a.txt`. My autograding scripts cannot be expected to find that. **Unless maybe** they can find your `output.txt` and figure your program reads there...but going to another folder relative to that is still bad.

Variable names should not begin with capital letters—except for special values like `Pi`. Scala and Linux/UNIX/Mac are case-sensitive

First, a note about Setters (ch. 3, § 3.2)

- Idea: Give syntax of field assignment but make interceptable by methods.
- IMPHO, if a field `foo` is to be (re-)settable from outside then it is not derived data.
- It is primary data, so should be a class argument
  `private var _foo`
- You can write a public setter
  `def setFoo(newFoo: Bar): Unit = _foo = newFoo`
- Scala Special Syntax:
  `def foo_=(newFoo: Bar): Unit = _foo = newFoo`
- Instead of `obj.setFoo(bar)` can now write simply `obj.foo = bar`
- The `foo_=` method name is "magic syntax" for any field name in place of `foo`.
- Can rewrite the method body (or override it in a subclass) to do further checks and updates, as may become needed, while never breaking client syntax.

Scala Symbolic Names

- Scala not only allows overloading built-in operators, it allows defining new operators with symbolic names.
- Legal symbols:

  `+ - * / % | & ^ ~ ! < > = ? $ \ :` Also `@ #`
- Can make identifiers with a legal alphanumeric name, then `_`, then symbols. The name `foo_=` was an example.
- More implicit magic: if a symbolic name ends in = and is not the name of an already-defined method, then Scala looks for a function named the symbols before the =. So `x !~*= y` becomes `x = x !~* y`.
- We've already used the triple-char operators `::=` and `+:=` and `:+=` for prepending and appending. The text covers related ones on pages 189–199.

Three Dimensions of Polymorphism (§§ 3.5,4.2,4.3

- Think of Inheritance as "up/down."
- Case classes are horizontal—in the sense of not inheriting from each other.
- Type parameters in `[...]` are like the foreground/background dimension. E.g. `Array[String]`.
- Case classes are not allowed to inherit from each other.
- Should inherit only from a `trait` that is `sealed` so that only case classes in the same file can inherit from it.
- Unlike with Java `instanceof` and C++ `dynamic_cast`, the Scala compiler can then know that the cases cover all possibilities. Comes up on p423 in the "Linked Lists" ch. 12—we will get there sooner than you may expect.
- Case classes provide **matchable structure**.
- Example code: `RealOrComplex.scala` (for all these slides).

Multiple Inheritance and the Diamond Problem

- A `UB_Person` is someone with a `UB_ID`. And a `name`.
- A `UB_Student` is a `UB_Person`. It (the class) inherits the `UB_ID` and `name` fields. The `UB_ID` is immutable, but the `name` is mutable, so a `UB_Student` instance has its own copy of `name`. It also defines a method `updateAccount`.
- A `UB_Employee` is a `UB_Person`. Also inherits own copy of `name`. It defines a method `updateAccount` with different body.
- A `UB_TA` is both a `UB_Student` and a `UB_Employee`. This class inherits `name` and `updateAccount` from both `UB_Student` and `UB_Employee`.
- **But which version of the field and method is inherited?**
- Of course, *good code* would avoid clashes, but in order to troubleshoot *bad code*, we need to specify rules and behavior subject to those rules.

Diamond-Avoidance Policies

- C++ recycles the `virtual` keyword to require all but one of the inherited classes to not be primary lookup. Else compile error.
- Java allows multiple inheritance only of `interfaces`, which disallow non-constant data and implemented methods altogether.
- That is, a Java `interface` may only have *abstract* methods.
- Whereas, an `abstract class` in both Java and Scala is a class that merely has at least one unimplemented method.
- In both Java and Scala, a subclass can inherit only one *class*.
- A Scala `trait` is like a Java interface but allowed to have method bodies and some kinds of data.
- Disallows class arguments...except **Scala 3** will allow them.
- Uses **linearization** to disambiguate.
- This constructs a *linear order* of inheritance for any *concrete class*. Goes in reverse order. Example `NewCar.scala`

Problem of Deep Object Hierarchies (not in text?)

- The `NewCar` example shows that the runtime system will look "down" for a superclass method, but then "bounce up" to re-poll the actual class of the invoking object, in this case `newCar`, then look down again...
- Sometimes called the "pogo-stick problem" or the "ping-pong problem."
- Case classes give ways to avoid this problem when hierarchy is really not needed.

Rectangle Versus Square, Part Deux (text end of §4.3)

- An *immutable* Square "Is-A" immutable Rectangle
- How about Array[Square] "Is-A" Array[Rectangle]?
- Problem: the array itself is mutable:

```
val as = //create a new Array[Square]
var ar: Array[Rectangle] = as          //compile error!
ar = ar :+ new Rectangle(3.0,4.0)    //that's why
```

- So Array[Square] is not a subtype of Array[Rectangle]
- Can work around by writing generic methods or classes with type parameter [A <: Rectangle], so that we can instantiate Array[A] as either Array[Rectangle] or Array[Square].