CSE250 Week 3, Continued

Case, Inheritance, and Generic Polymorphism

Kenneth W. Regan

February 21, 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?).
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.
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/

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

Variable names should not begin with capital letters—except for special values like Pi. Scala and Linux/UNIX/Mac are case-sensitive.
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)
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...
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)
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.
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.
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.
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.
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.
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
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
  
  ```scala
  private var _foo
  ```
- You can write a public setter
  
  ```scala
  def setFoo(newFoo: Bar): Unit = _foo = newFoo
  ```
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
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
  ```scala
class MyClass {
  private var _foo
}
```
- You can write a public setter
  ```scala
def setFoo(newFoo: Bar): Unit = _foo = newFoo
```
- Scala Special Syntax:
  ```scala
def foo_=(newFoo: Bar): Unit = _foo = newFoo
```
- Instead of `obj.setFoo(bar)` can now write simply `obj.foo = bar`
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
  ```scala
  private var _foo
  ```
- You can write a public setter
  ```scala
def setFoo(newFoo: Bar): Unit = _foo = newFoo
  ```
- Scala Special Syntax:
  ```scala
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`. 
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 not only allows overloading built-in operators, it allows defining new operators with symbolic names.
Scala Symbolic Names

- Scala not only allows overloading built-in operators, it allows defining new operators with symbolic names.
- Legal symbols:
  \[ + - * \ / \ \% \ \| \ \& \ ^ \ ~ \ ! \ < \ > \ = \ ? \$ \ \backslash \ : \Also \ \@ \# \]
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.
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.
Scala Symbolic Names

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

- Legal symbols:
  
  $+$ $-$ $\ast$ $/$ $\%$ $|$ $\&$ $\sim$ $\sim$ $!$ $<$ $>$ $=$ $?$ $\$$ $\backslash$ $:$ 

  Also $\circ$ $@$ $\#$

- 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 $x$ $!\sim*$ $y$ becomes $x = x$ $!\sim*$ $y$.

- We’ve already used the triple-char operators $::=$ and $+:= $ and $+:=$ for prepending and appending. The text covers related ones on pages 189–199.
Think of Inheritance as “up/down.”
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.
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].
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.
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.
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.
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.
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.
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.
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_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.
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.
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?**
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.
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.
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.
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.
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.
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.
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.
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...
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.
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.
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
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
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...
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.”
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 Vesus Square, Part 1-1/2

- The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.

You cannot override the foo_ setter for a var field. You still can have a base-class method reset it when holding your derived-class object. This is legal but bad:

```java
MutableRectangle r = new MutableSquare(5.0)
r.width = 20
```

Just don’t have `MutableSquare` inherit from `MutableRectangle`. A subclass argument `class Square(var width)` will shadow the superclass. Making a new name via `class Square(var side)` is still code duplication. But `class Square(side)` is OK and avoids this.
Rectangle Vesus Square, Part 1-1/2

- The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.
- You cannot override the \texttt{foo_=} setter for a \texttt{var} field \texttt{foo}.
The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.

You cannot override the foo_= setter for a var field foo.

You still can have a base-class method reset it when holding your derived-class object. This is legal but bad:

```java
MutableRectangle r = new MutableSquare(5.0)
r.width = 20
```

Just don't have `MutableSquare` inherit from `MutableRectangle`.

A subclass argument `class Square(var width)` will shadow the superclass. Making a new name via `class Square(var side)` is still code duplication.

But `class Square(side)` is OK and avoids this.
The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.

You cannot override the foo\_= setter for a var field foo.

You still can have a base-class method reset it when holding your derived-class object. This is legal but bad:

```
MutableRectangle r = new MutableSquare(5.0)
r.width = 20
```

Just don’t have MutableSquare inherit from MutableRectangle.
Rectangle Versus Square, Part 1-1/2

- The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.
- You cannot override the foo_= setter for a var field foo.
- You still can have a base-class method reset it when holding your derived-class object. This is legal but bad:

```java
MutableRectangle r = new MutableSquare(5.0)
r.width = 20
```

- Just don’t have MutableSquare inherit from MutableRectangle.
- A subclass argument class Square(var width) will shadow the superclass.
Rectangle Versus Square, Part 1-1/2

- The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.
- You cannot override the foo_= setter for a var field foo.
- You still can have a base-class method reset it when holding your derived-class object. This is legal but bad:

```java
MutableRectangle r = new MutableSquare(5.0)
r.width = 20
```

- Just don’t have MutableSquare inherit from MutableRectangle.
- A subclass argument class Square(var width) will shadow the superclass.
- Making a new name via class Square(var side) is still code duplication.
The test goes thru Rectangle versus Square on pp123-125 with some extra points over what I said before.

You cannot override the foo_= setter for a var field foo.

You still can have a base-class method reset it when holding your derived-class object. This is legal but bad:

```
MutableRectangle r = new MutableSquare(5.0)
r.width = 20
```

Just don’t have MutableSquare inherit from MutableRectangle.

A subclass argument class Square(var width) will shadow the superclass.

Making a new name via class Square(var side) is still code duplication.

But class Square(side) is OK and avoids this.
An immutable Square “Is-A” immutable Rectangle
An immutable Square “Is-A” immutable Rectangle

How about Array[Square] “Is-A” Array[Rectangle]?
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:

```scala
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
```
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:

```scala
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]
An immutable Square “Is-A” immutable Rectangle

How about Array[Square] “Is-A” Array[Rectangle]?

Problem: the array itself is mutable:

```scala
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].
List Matching Examples

(show code FilterLists.scala)