Scala Immutable Collections

```
import scala.collection.immutable._
```

Immutable Lists

- much like the lists we have defined in class
- Lists are covariant
- The empty list is written `Nil`
- Nil extends `List[Nothing]`

Immutable Lists

- constructor takes a variable number of arguments:

```
List(1,2,3,4,5,6)
```

Immutable Lists

- Non-empty lists are built from Nil and Cons
- written as the right-associative operator `::`

```
1 :: 2 :: 3 :: 4 :: Nil
```

↦

```
(1 :: (2 :: (3 :: (4 :: Nil))))
```

List Operations

- `head` returns the first element
- `tail` returns a list of elements but the first
- `isEmpty` returns true if the list is empty
- Many of the methods we have defined are available on the built-in lists

FoldLeft and FoldRight as Operators

- foldLeft:

```
(zero /: xs) op
```

- foldRight:

```
(xs :\ zero) op
```

Sort

```
List(1,2,3,4,5,6) sortWith (_ < _)
```

Range

```
List.range(1,5)
```

Using Fill for Uniform Lists

```
List.fill(10)(0) ↦
List(0,0,0,0,0,0,0,0,0,0)
```

Using Fill for Uniform Lists

```
List.fill(3,3)(0) ↦
List(List(0,0,0),
     List(0,0,0),
     List(0,0,0))
```

Tabulating Lists

```
List.tabulate(3,3) (
  (m,n) => if (m == n) 1 else 0)
)
↦
List(List(1,0,0),
     List(0,1,0),
     List(0,0,1))
```

Immutable Sets

Immutable Sets

- unordered, unrepeated collections of elements

- parametric and covariant in their element type

Immutable Sets

```
Set(1,2,3,4,5)
```

Immutable Sets

```
Set(1,2,3) + 4 ↦
Set(1,2,3,4)
```

Immutable Sets

```
Set(1,2,3) - 2 ↦
Set(1,3)
```

Immutable Sets

```
Set(1,2,3) - 4 ↦
Set(1,2,3)
```

Immutable Sets

```
Set(1,2,3) ++ Set(2,4,5) ↦
Set(1,2,3,4,5)
```

Immutable Sets

```
Set(1,2,3) -- Set(2,4,5,3) ↦
Set(1)
```

Immutable Sets

```
Set(1,2,3) & Set(2,4,5,3) ↦
Set(2,3)
```

Immutable Sets

```
Set(1,2,3).size ↦
```

3

Immutable Sets

```
Set(1,2,3).contains(2) ↦
true
```

Immutable Maps

Immutable Maps

- collections of key/value pairs
- parametric in both the key and value type
  - Invariant in their key type
  - Covariant in their value type

The -> Operator

- The infix operator `->` returns a pair of its arguments:

```
1 -> 2
```

↦

```
(1,2)
```

-> is Left Associative

```
1 -> 2 -> 3 -> 4
```

↦

```
(((1,2),3),4)
```

The Map Constructor

```
Map(“a” -> 1, “b” -> 2, “c” -> 3)
```

↦

```
Map(a -> 1, b -> 2, c -> 3)
```

Map Addition

```
Map(“a” -> 1, “b” -> 2, “c” -> 3) + (“d” -> 4)
```

↦

```
Map(a -> 1, b -> 2, c -> 3, d -> 4)
```

Map Operations

- The operators `-, ++, --, size` are defined in the expected way

Map Search

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).contains(“b”)
```

↦

```
true
```

Map Access

```
Map(“a” -> 1, “b” -> 2, “c” -> 3)(“c”)
```

↦

3

Map keys

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).keys
```

↦

```
Set(a, b, c):Iterable[String]
```

Map values

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).values
```

↦

```
MapLike(1,2,3):Iterable[Int]
```

Map emptiness

```
Map(“a” -> 1, “b” -> 2, “c” -> 3).isEmpty
```

↦

```
false
```

Traits

Traits

- Traits provide a way to factor out common behavior among multiple classes and mix it in where appropriate

Trait Definitions

- Syntactically, a trait definition looks like a class definition but with the keyword “trait”

```
trait Echo {
  def echo(message: String) =
    message
}
```

Trait Definitions

- Traits can declare fields and full method definitions
- They must not include constructors

```
trait Echo {
  val language = “Portuguese"
  def echo(message: String) =
    message
}
```

Using Traits

- Classes “mix in” traits using either the **`extends`** or **`with`** keywords

```
class Parrot extends Echo {
  def fly() = {
    // forget to fly and talk instead
    echo(“Polly wants a cracker”)
  }
}
```

Using Traits

- Classes “mix in” traits using either the **`extends`** or **`with`** keywords

```
class Parrot extends Bird with Echo {
  def fly() = {
    // forget to fly and talk instead
    echo(“Polly wants a cracker”)
  }
}
```

Using Traits

- Classes “mix in” traits using either the **`extends`** or **`with`** keywords

```
trait Smart {
  def somethingClever() =
    “better a witty fool than a foolish wit”
}
```

Using Traits

- Classes can mix in multiple traits using either the `with` keywords

```
class Parrot extends Bird with Echo
with Smart {
  def fly() = {
    // forget to fly and talk instead
    echo(somethingClever())
  }
}
```

Thin vs Rich Interfaces

- Traits provide a way to resolve the tension between “thin” and “rich” interfaces:
    - Thin interface: Include only essential methods in an interface
        - Good for implementors
- Rich interface: Include a rich set of methods in an interface
    - Good for clients

Thin vs Rich Interfaces

- With traits, we can define an interface to include only a small number of essential methods, but then include traits to build rich functionality based on the essential methods

- Implementors win

- Clients win

Thin vs Rich Interfaces

- Consider our implementations of Interval, Rational, Measurement
- We want to include all comparison operators on them:

  `<` `<=` `>=` `>`

  - With traits, we could define just one operator `<` and mix in a trait to define the rest in terms of `<`

Thin vs Rich Interfaces

```
case class Measurement(magnitude: BigDecimal,
                       unit: PhysicalUnit)
extends Ordered[Measurement]
  def compare(that: Measurement) =
    val (u,m1,m2) = this.unit commonUnits that.unit
    (m1 * magnitude) - (m2 * that.magnitude)
  }
  …
}
```

Traits as Stackable Modifiers

```
abstract class IntMap {
  def insert(s: String, n: Int): IntMap
  def retrieve(s: String): Int
}
```

Traits as Stackable Modifiers

```
case class IntListMap(elements: List[(String,Int)] = Nil)
extends IntMap {

  def insert(s: String, n: Int): IntMap =
    IntListMap((s -> n) :: elements)

  def retrieve(s: String) = {
    def retrieve(xs: List[(String, Int)]): Int = {
      xs match {
        case Nil => throw new IllegalArgumentException(s)
        case (t, n) :: ys if (s == t) => n
        case y :: ys => retrieve(ys)
      }
    }
    retrieve(elements)
  }
}
```

Traits as Stackable Modifiers

```
trait Incrementing extends IntMap {
  abstract override def insert(s: String, n: Int) =
    super.insert(s, n + 1)
}
```

This super call depends on how the trait is mixed into a particular class

Traits as Stackable Modifiers

```
trait Filtering extends IntMap {
  abstract override def insert(s: String, n: Int) = {
    if (n >= 0) super.insert(s, n)
    else this
  }
}
```

As does this one

Traits as Stackable Modifiers

```
> val m = new IntListMap() with Incrementing with Filtering
m: IntListMap with Incrementing with Filtering = IntListMap(List())
```

The order in which the traits are listed is important.
The trait furthest to the right is called first

Traits as Stackable Modifiers

```
> m.insert("a", -1)
res0: IntMap = IntListMap(List())
```

Traits as Stackable Modifiers

```
> res0.retrieve("a")
java.lang.IllegalArgumentException: a
```

Traits as Stackable Modifiers

```
> val m = new IntListMap() with Filtering with Incrementing
m: IntListMap with Filtering with Incrementing = IntListMap(List())
```

Now we have reversed the order

Traits as Stackable Modifiers

```
> m.insert("a", 1)
res2: IntMap = IntListMap(List((a,2)))
```

Traits as Stackable Modifiers

```
> res2.retrieve("a")
res3: Int = 2
```

Traits as Stackable Modifiers

```
> m.insert("a", -1)
res0: IntMap = IntListMap(List((a,0)))
```

Now the integer is incremented before filtering,
and so it passes the filter

Traits as Stackable Modifiers

```
> res0.retrieve("a")
res5: Int = 0
```

Traits vs Multiple Inheritance

Traits vs Multiple Inheritance

- The key property of traits that distinguishes them from multiple inheritance is *linearization*

- With traditional multiple inheritance, which implementation of insert would be called:

```
class MyMap() extends IntListMap() with Filtering
                      with Incrementing

                  new MyMap().insert("b",2)
```

Traits vs Multiple Inheritance

- With traits, the effect of a super call is determined by the linearization of traits, which enables:
  - Multiple trait implementation of the same method to be called
  - Multiple ways to compose the traits depending on circumstances

Trait Linearization

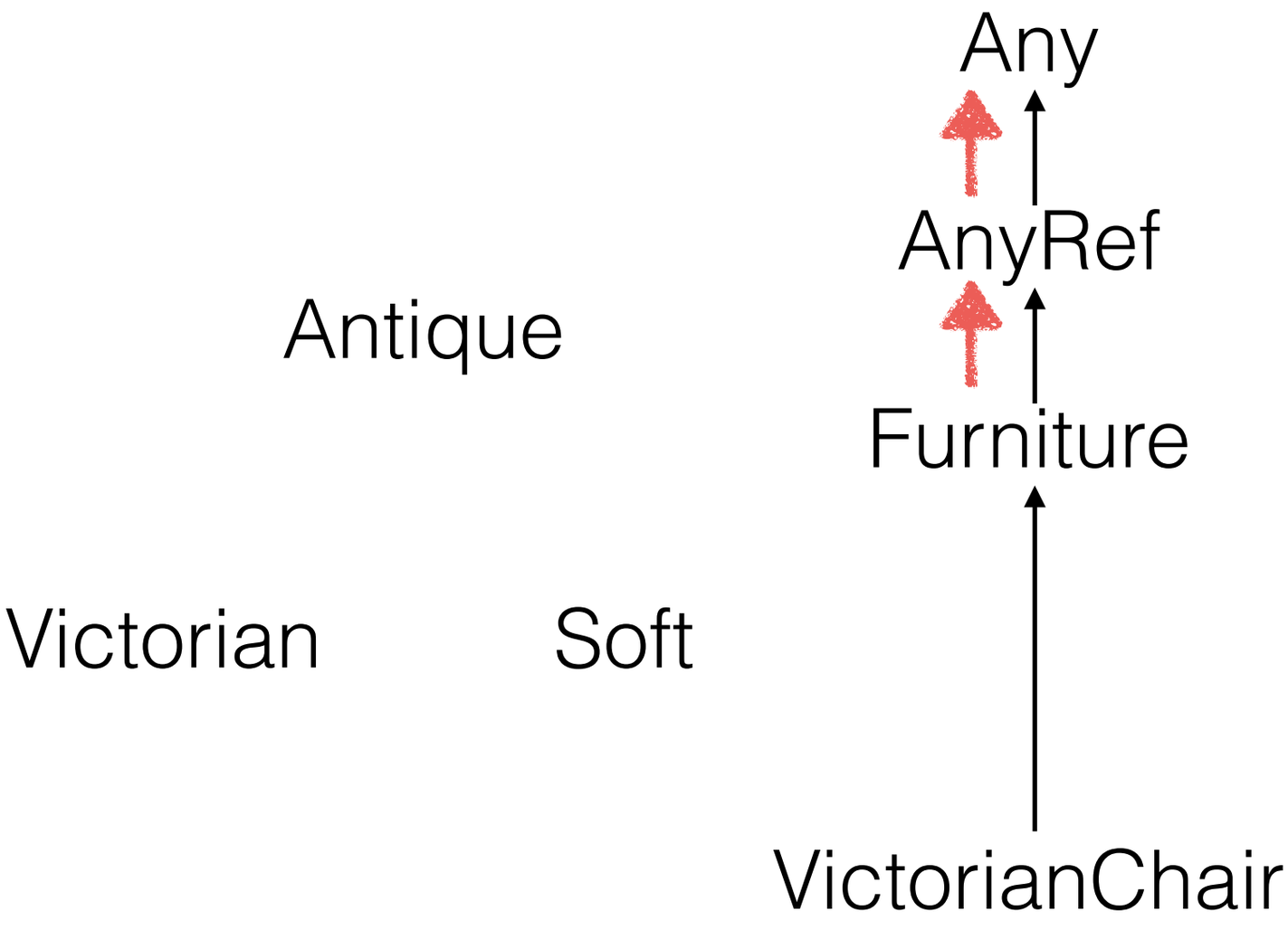
```
class C() extends D() with T1… with TN {
    …
}
```

- To linearize class C
- Linearize class D
- Extend with the linearization of T1, leaving out classes already linearized
- Continue until extending with the linearization of TN, leaving out classes already linearized
- Finally, extend with the body of class C

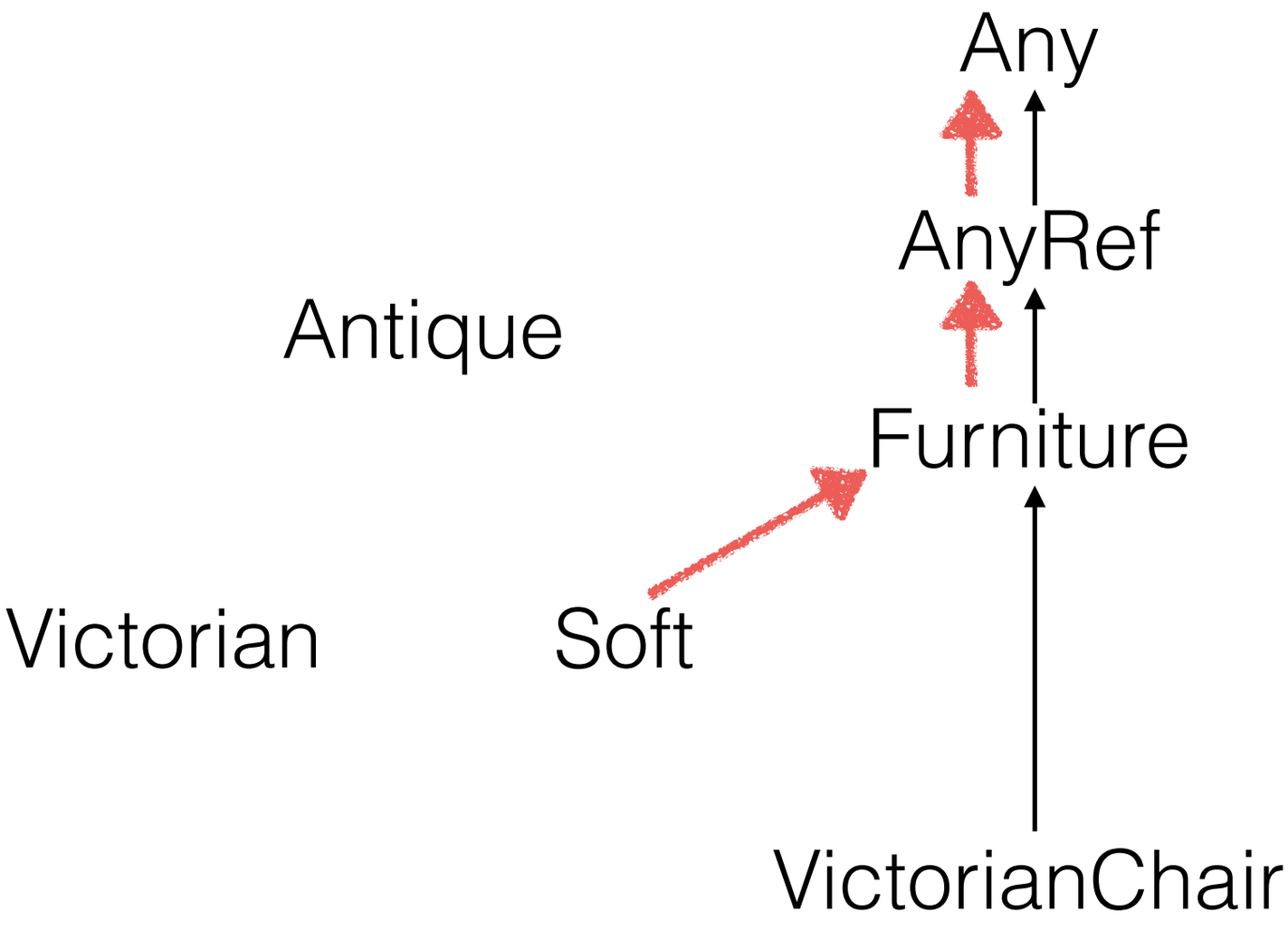
Trait Linearization

```
class Furniture
trait Soft extends Furniture
trait Antique extends Furniture
trait Victorian extends Antique
class VictorianChair extends Furniture with Soft with Victorian
```

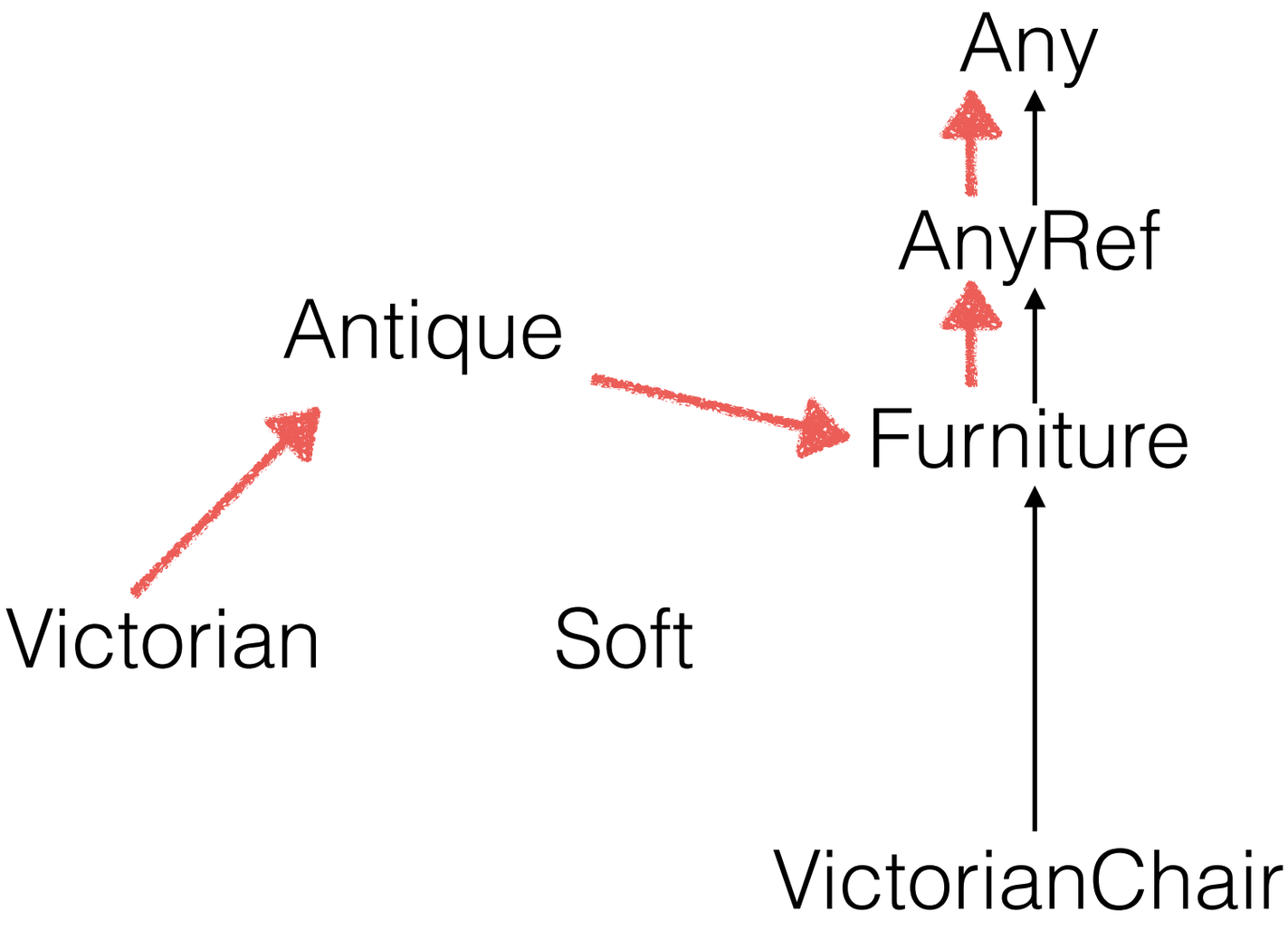
Linearization of Furniture



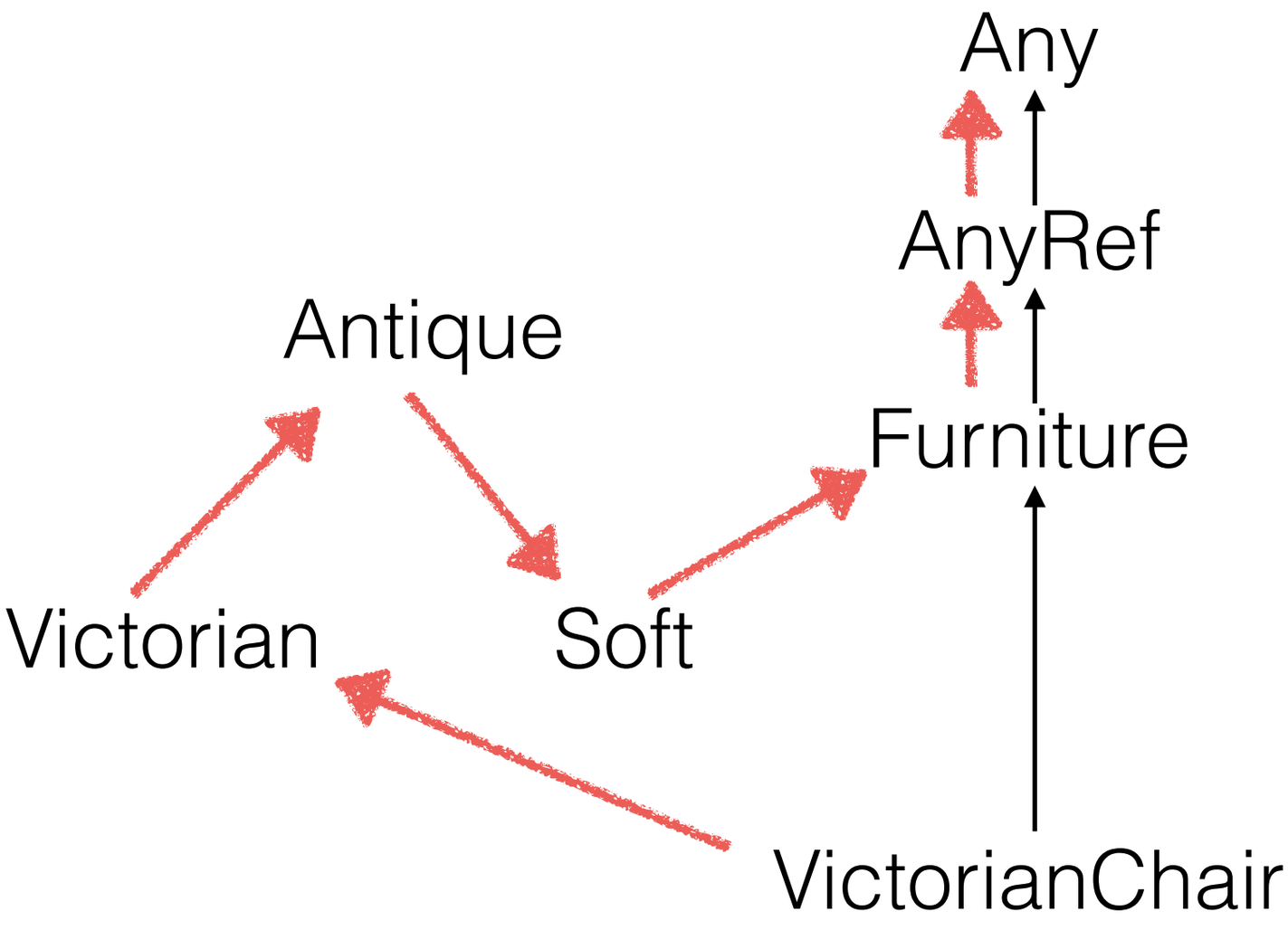
Linearization of Soft



Linearization of Victorian



Linearization of VictorianChair



Guidelines on Using Traits

- Use concrete classes when the behavior is not reused

- Use traits to capture behavior that is reused in multiple, unrelated classes

- If clients will inherit the behavior, try to make it an abstract class