Inheritance and Abstract Classes

Nancy Hitschfeld
Matías Toro

Java Lists in Scala

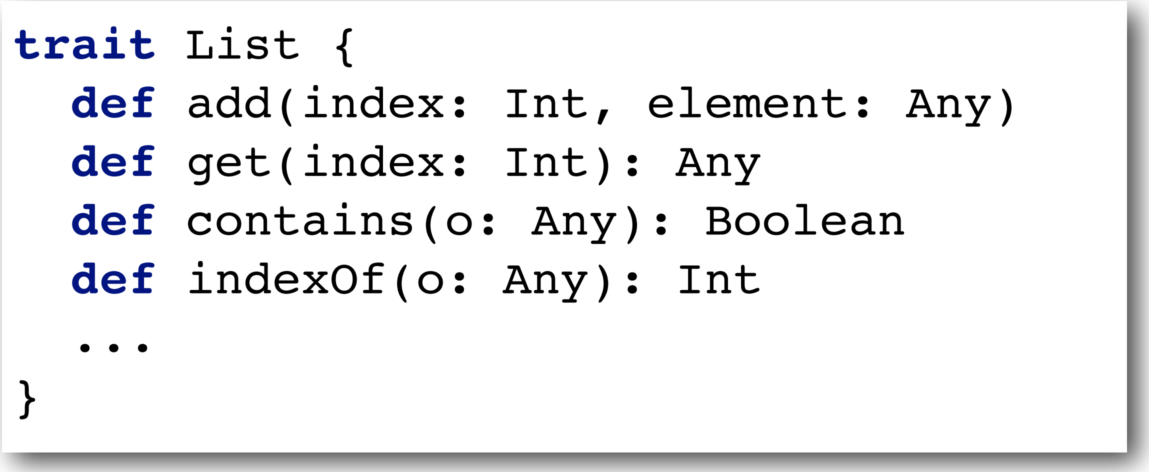
A list is an ordered collection / sequence

- precise control over where an element is inserted
- access elements by position
- search for elements in the list

Should this be specified in a class,
or an interface?

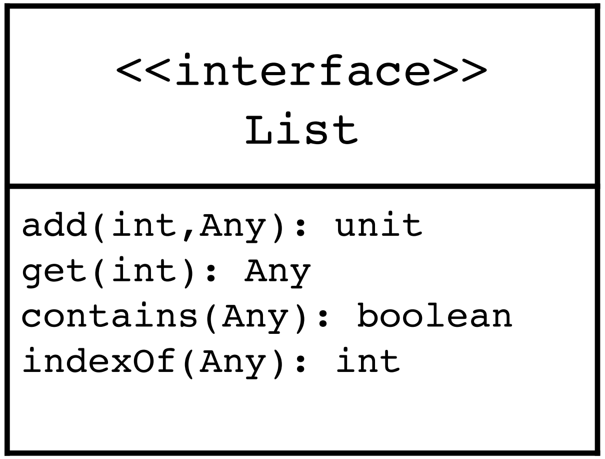
Do it!

Scala



```
trait List {
  def add(index: Int, element: Any)
  def get(index: Int): Any
  def contains(o: Any): Boolean
  def indexOf(o: Any): Int
  ...
}
```

UML



```
<<interface>>
List
```

```
add(int,Any): unit
get(int): Any
contains(Any): boolean
indexOf(Any): int
```

This is the specification of the interface of a list object

how do we create lists?
we need classes(factories)!

Creating Lists

Different implementation strategies

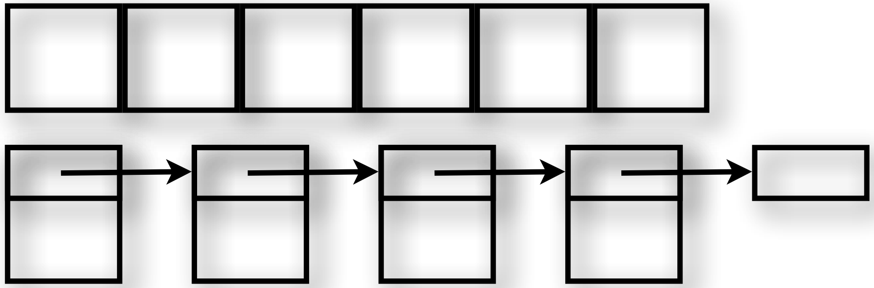
an array list

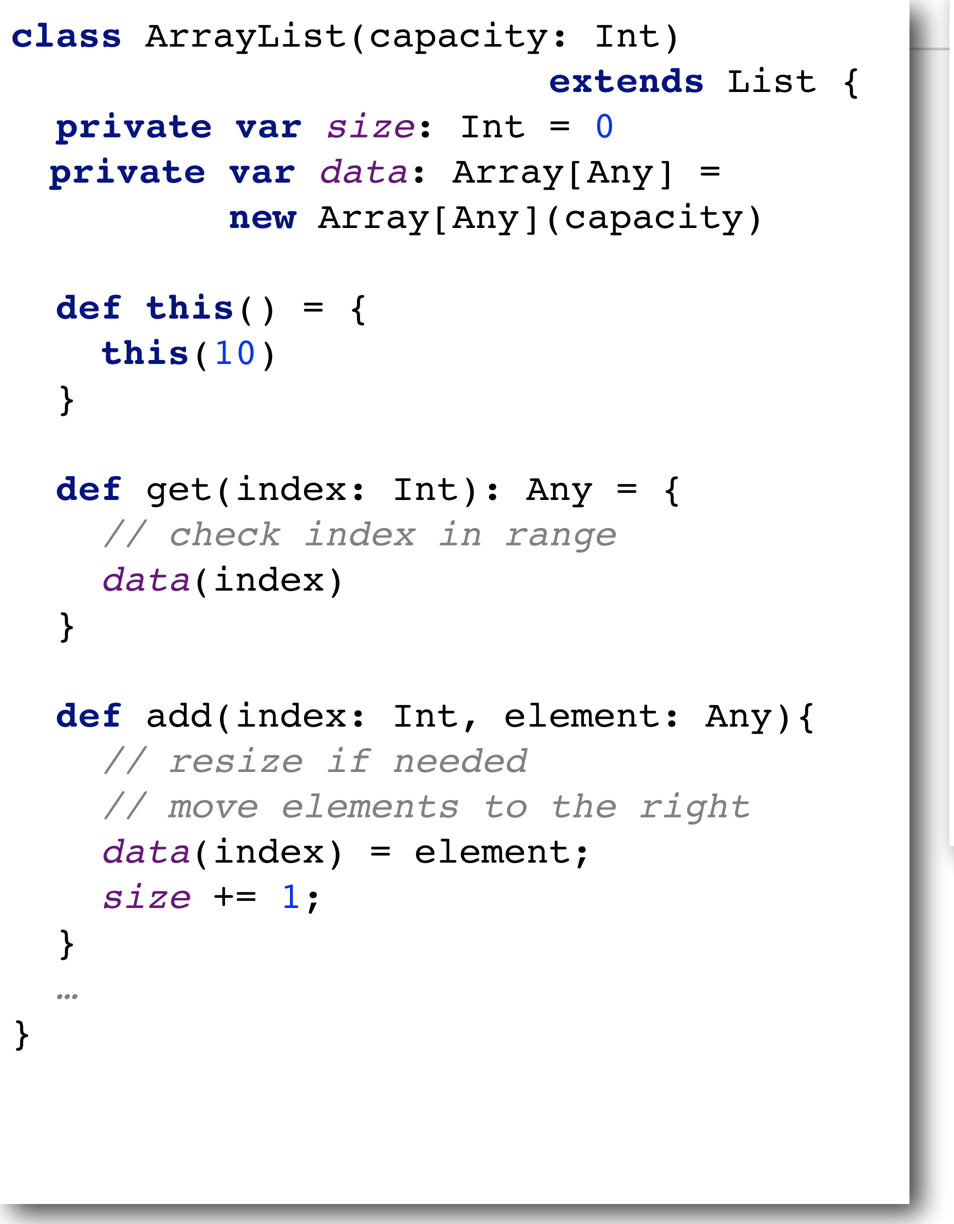
a linked list

a double-linked list

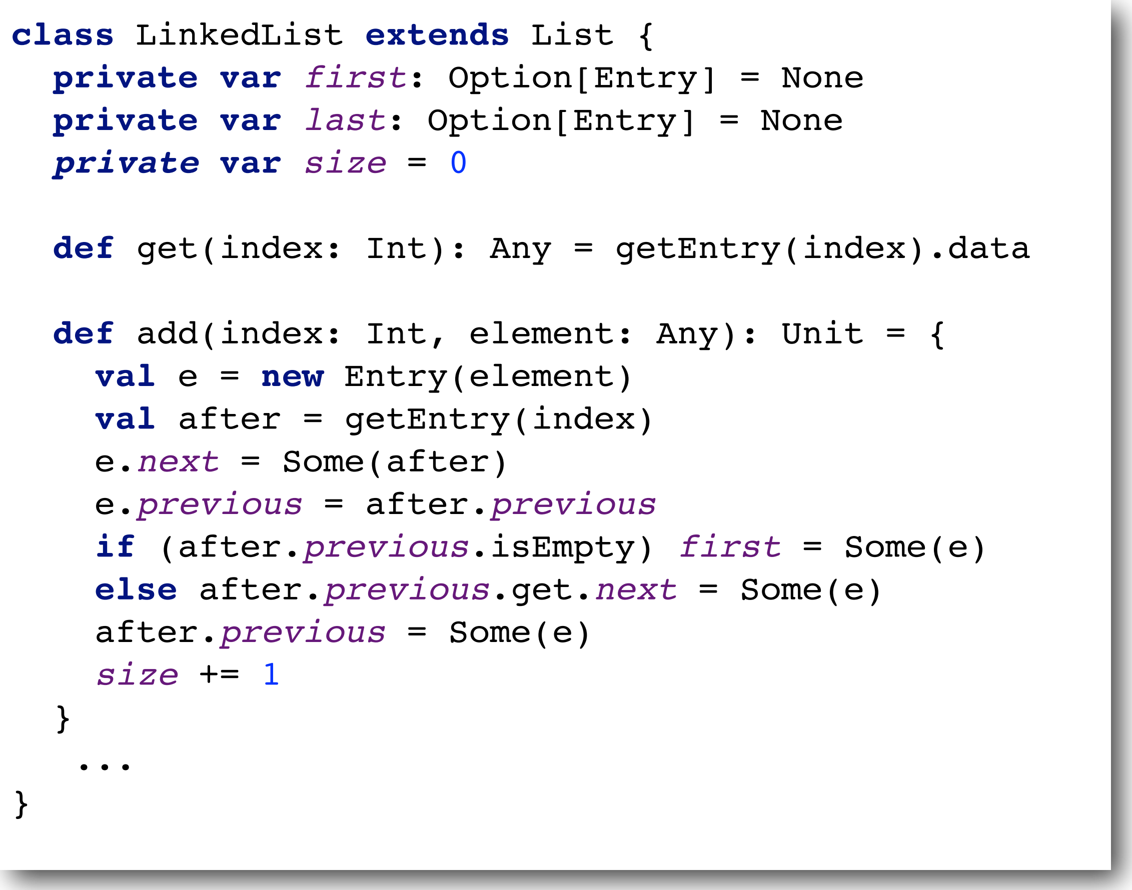
a copy-on-write array list

etc.... (10 implementations in the JDK)

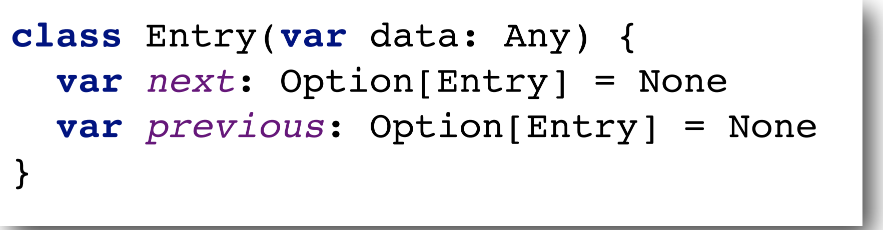




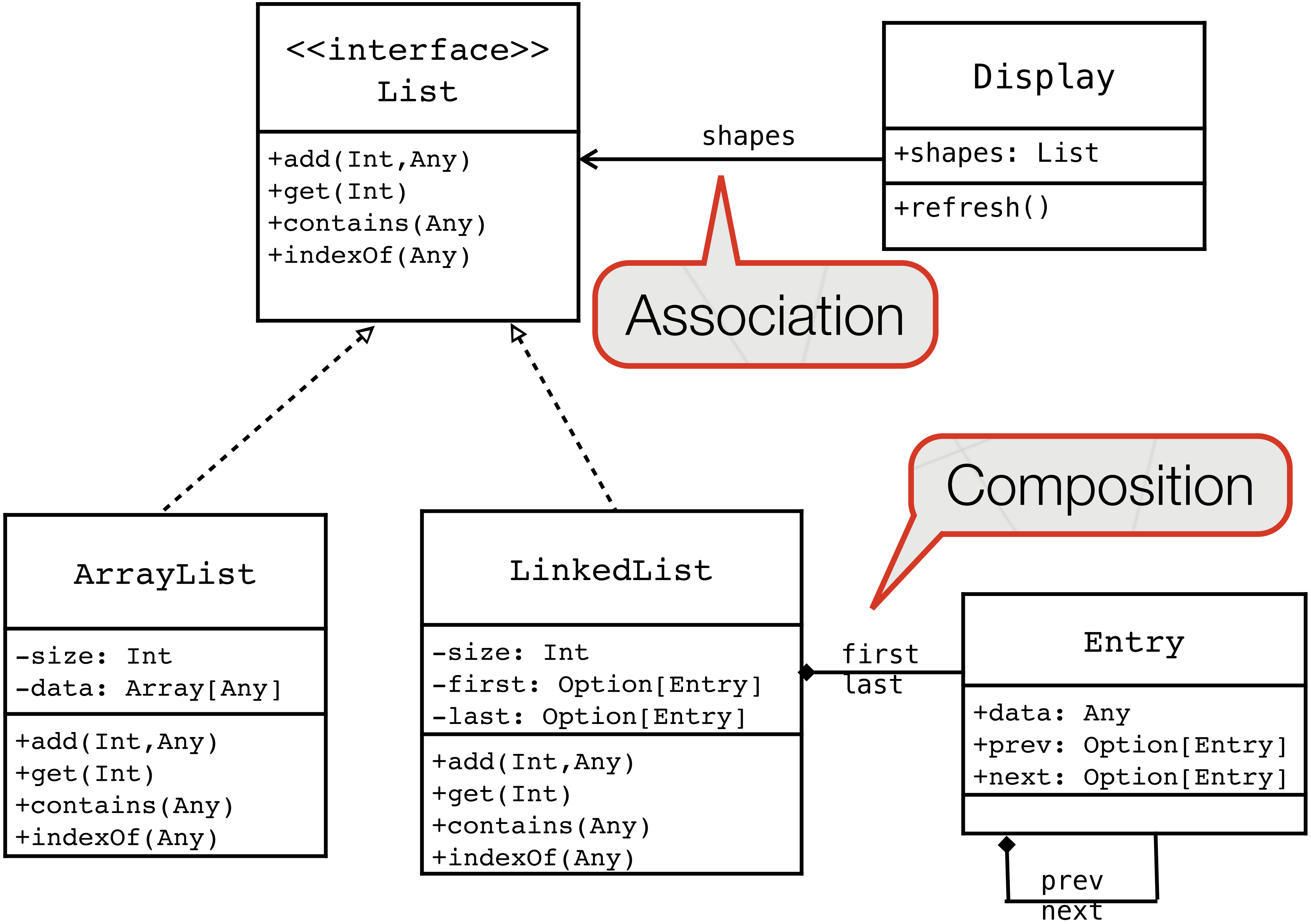
```
class ArrayList(capacity: Int)
                        extends List {
  private var size: Int = 0
  private var data: Array[Any] =
          new Array[Any](capacity)

  def this() = {
    this(10)
  }

  def get(index: Int): Any = {
    // check index in range
    data(index)
  }

  def add(index: Int, element: Any){
    // resize if needed
    // move elements to the right
    data(index) = element;
    size += 1;
  }
  …
}
```



```
class LinkedList extends List {
  private var first: Option[Entry] = None
  private var last: Option[Entry] = None
  private var size = 0

  def get(index: Int): Any = getEntry(index).data

  def add(index: Int, element: Any): Unit = {
    val e = new Entry(element)
    val after = getEntry(index)
    e.next = Some(after)
    e.previous = after.previous
    if (after.previous.isEmpty) first = Some(e)
    else after.previous.get.next = Some(e)
    after.previous = Some(e)
    size += 1
  }
   ...
}
```



```
class Entry(var data: Any) {
  var next: Option[Entry] = None
  var previous: Option[Entry] = None
}
```



<<interface>>
List

+add(Int,Any)
+get(Int)
+contains(Any)
+indexOf(Any)

Display

+shapes: List

+refresh()

Association

ArrayList

-size: Int
-data: Array[Any]

+add(Int,Any)
+get(Int)
+contains(Any)
+indexOf(Any)

LinkedList

-size: Int
-first: Option[Entry]
-last: Option[Entry]

+add(Int,Any)
+get(Int)
+contains(Any)
+indexOf(Any)

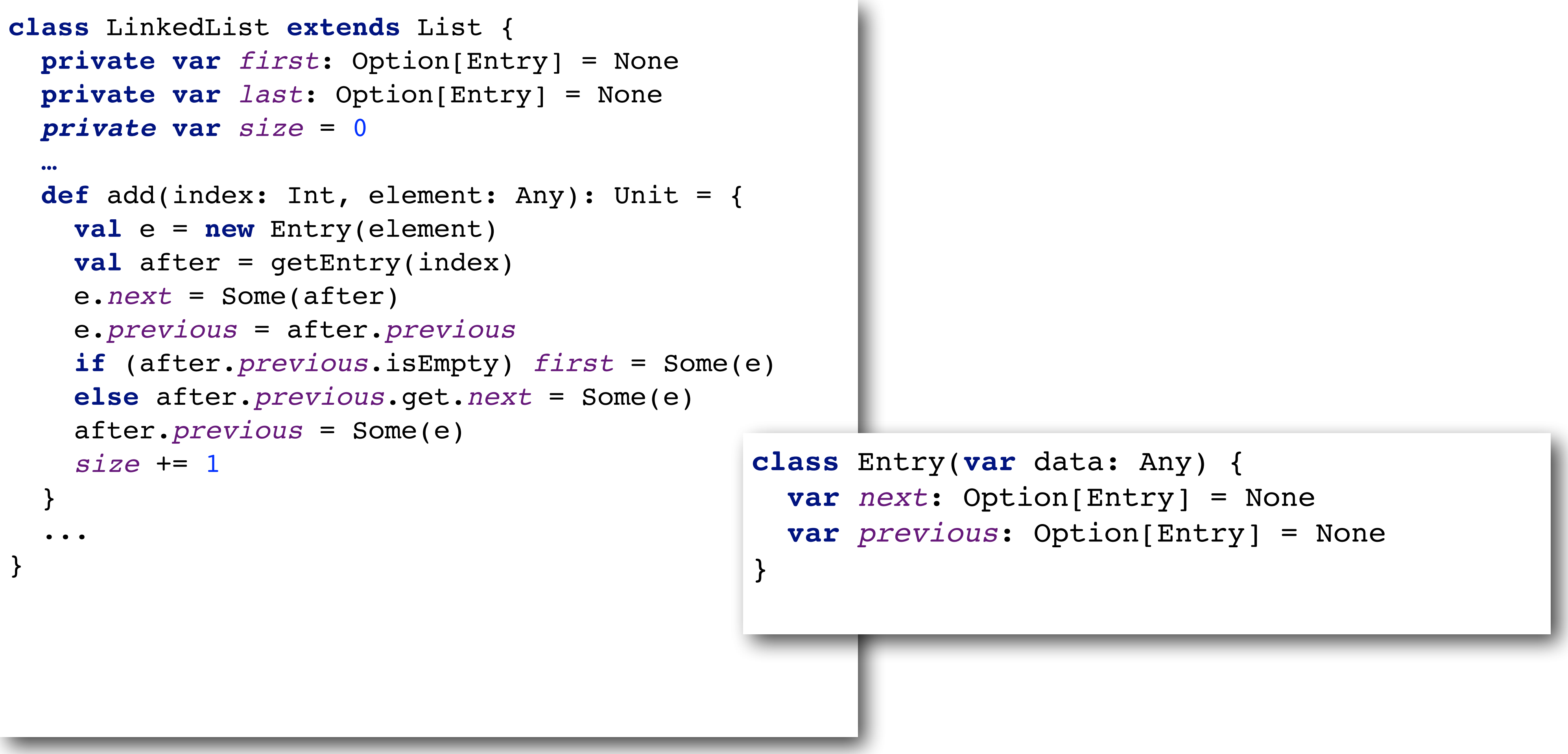
Composition

Entry

+data: Any
+prev: Option[Entry]
+next: Option[Entry]

What do you mean OOP?

What do you think about the definition and use of Entry?

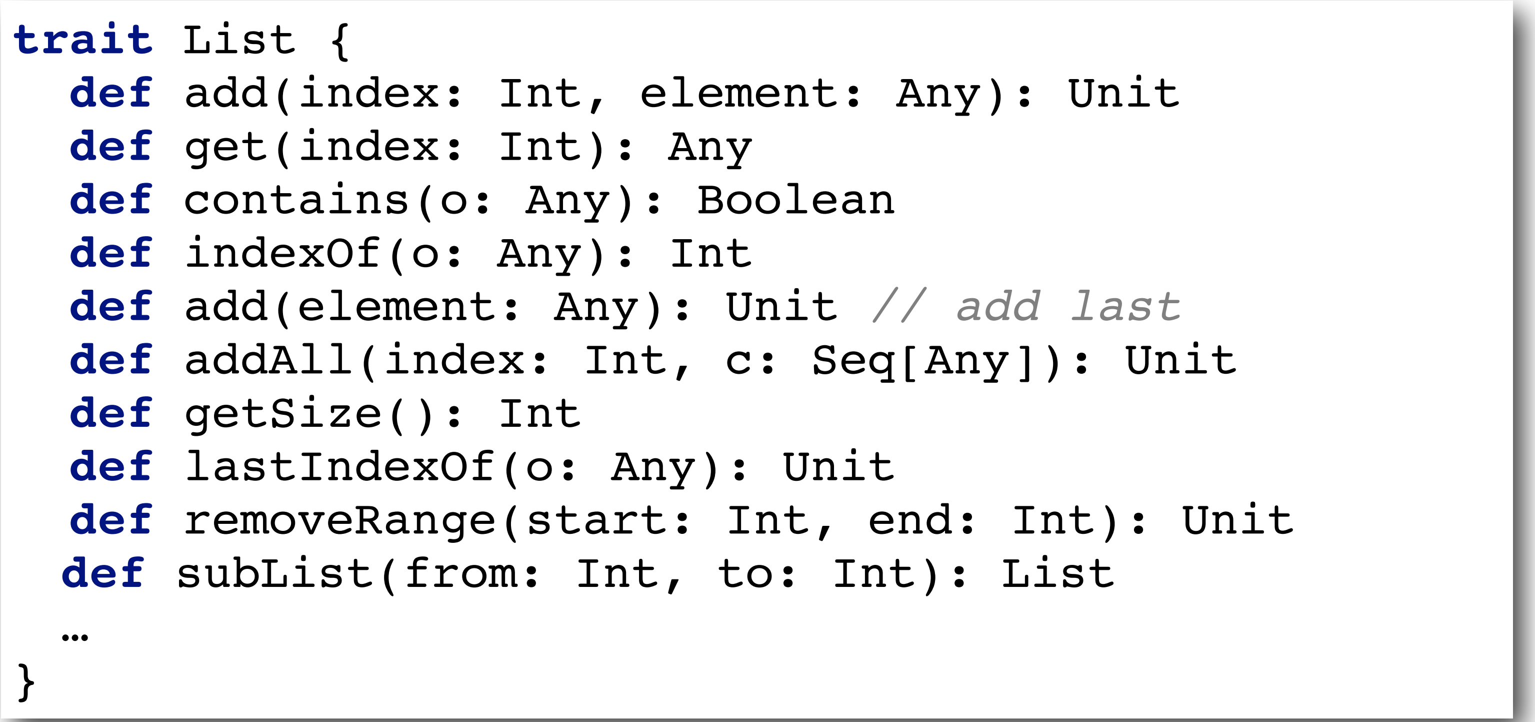


```
class LinkedList extends List {
  private var first: Option[Entry] = None
  private var last: Option[Entry] = None
  private var size = 0
  …
  def add(index: Int, element: Any): Unit = {
    val e = new Entry(element)
    val after = getEntry(index)
    e.next = Some(after)
    e.previous = after.previous
    if (after.previous.isEmpty) first = Some(e)
    else after.previous.get.next = Some(e)
    after.previous = Some(e)
    size += 1
  }
  ...
}
```

```
class Entry(var data: Any) {
  var next: Option[Entry] = None
  var previous: Option[Entry] = None
}
```

Lists in Scala

The List interface has many more methods



```
trait List {
  def add(index: Int, element: Any): Unit
  def get(index: Int): Any
  def contains(o: Any): Boolean
  def indexOf(o: Any): Int
  def add(element: Any): Unit // add last
  def addAll(index: Int, c: Seq[Any]): Unit
  def getSize(): Int
  def lastIndexOf(o: Any): Unit
  def removeRange(start: Int, end: Int): Unit
  def subList(from: Int, to: Int): List
  …
}
```

how would you implement add and `addAll` in ArrayList?
in LinkedList?

Do it!

Lists in Scala

These methods have the SAME implementation in any concrete list class!

```
def add(element: Any) = {
  add(getSize(), element)
}
```

```
def addAll(index: Int, c: Seq[Any]) = {
  for(o <- c.reverse){
    add(index, o)
  }
}
```

Reusing Implementation

A class is a factory of objects

- it holds the “template” out of which its instances are made
- definitions of fields and methods

Inheritance is a mechanism for reusing and extending factories

- produce similar kinds of objects, with variations/extensions

Avoiding code duplication: Inheritance

A class can INHERIT from another class

- it inherits all methods and fields
- (therefore, it is a subtype)

The superclass

- factors common state and behavior of its subclasses

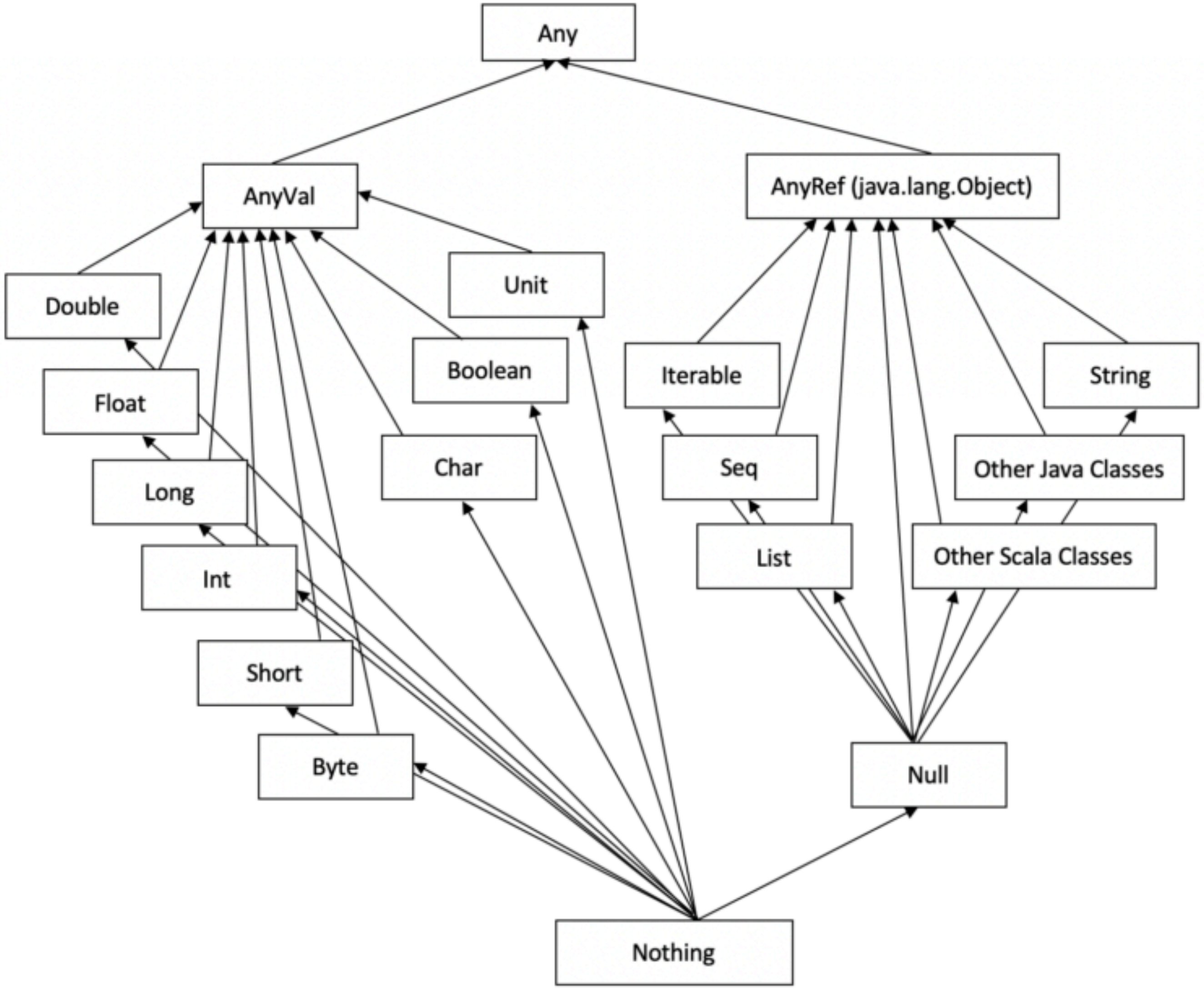
Class Hierarchies

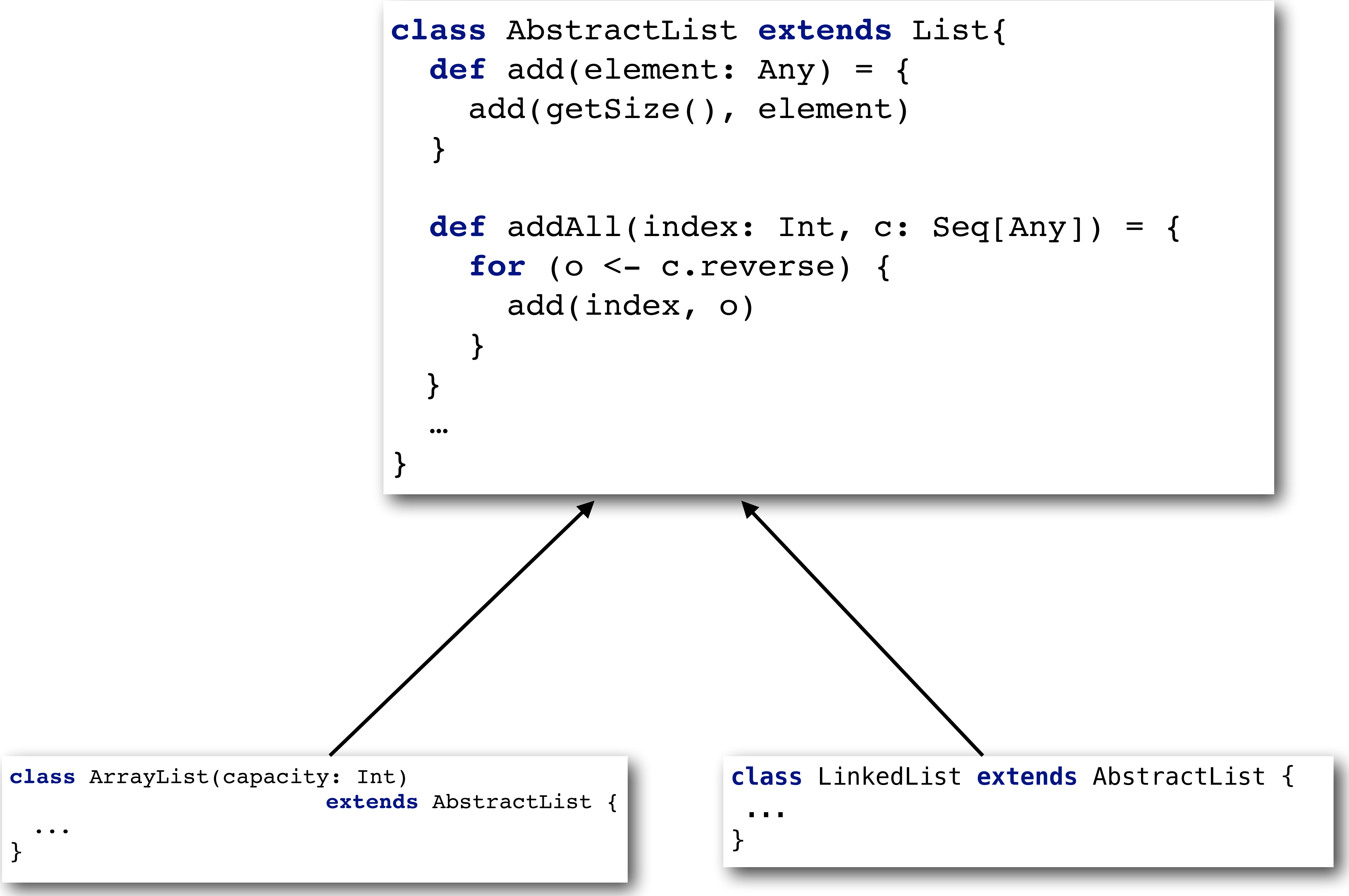
Inheritance

all classes inherit from a root class (Any in Scala)

with single class inheritance, hierarchy is a tree

Not an UML Diagram!!



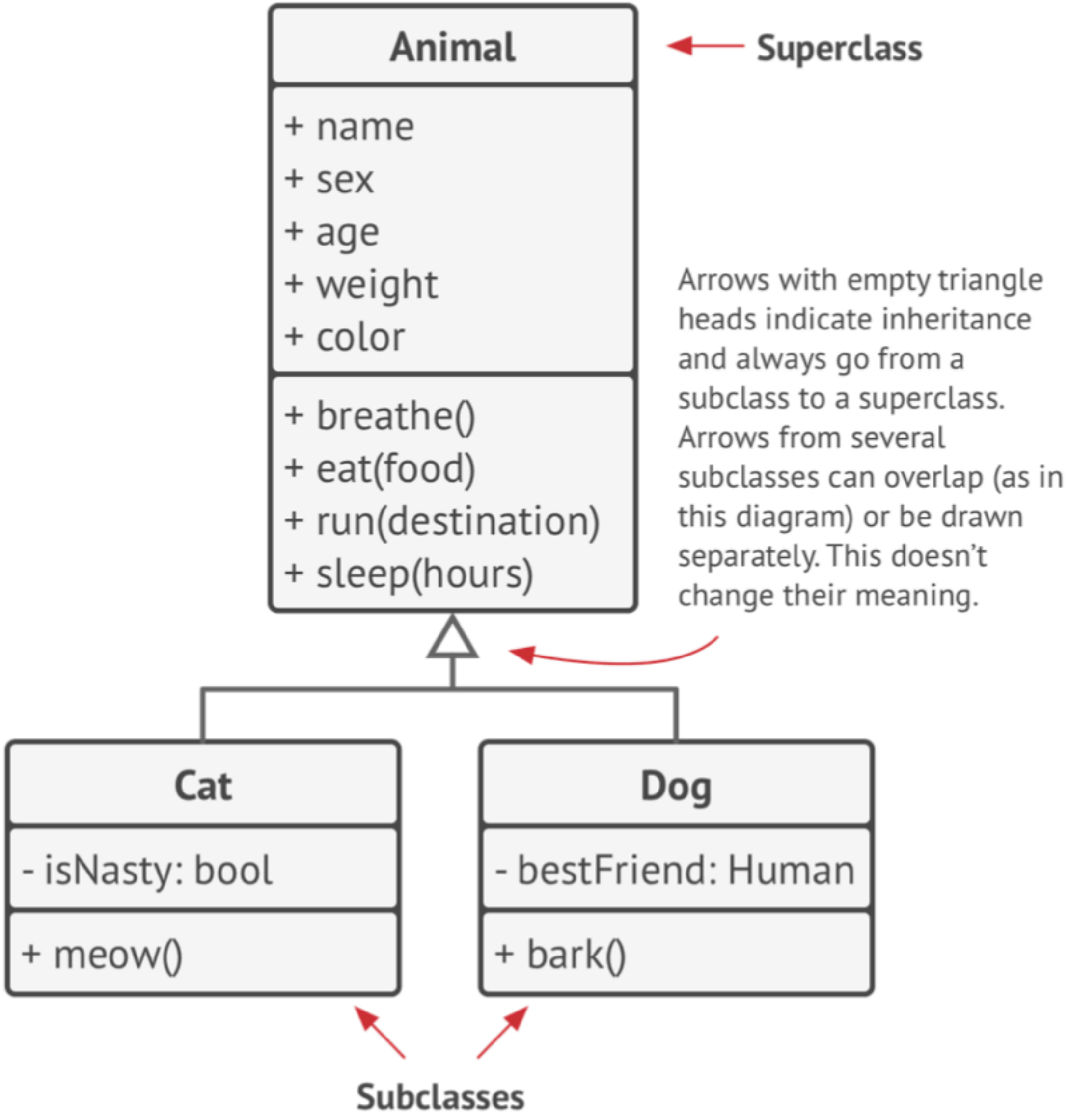


```
class AbstractList extends List{
  def add(element: Any) = {
    add(getSize(), element)
  }

  def addAll(index: Int, c: Seq[Any]) = {
    for (o <- c.reverse) {
      add(index, o)
    }
  }
  …
}
```

```
class ArrayList(capacity: Int)
                        extends AbstractList {
  ...
}
```

```
class LinkedList extends AbstractList {
 ...
}
```

UML diagram of a class hierarchy



Animal

+ name
+ sex
+ age
+ weight
+ color

+ breathe()
+ eat(food)
+ run(destination)
+ sleep(hours)

Superclass

Arrows with empty triangle heads indicate inheritance and always go from a subclass to a superclass. Arrows from several subclasses can overlap (as in this diagram) or be drawn separately. This doesn't change their meaning.

Cat

- isNasty: bool

+ meow()

Dog

- bestFriend: Human

+ bark()

Subclasses

Abstract class

An abstract class is an incomplete class.

```
abstract class AbstractList extends List{
 ...
}
```

Abstract classes are not meant to be used as a type. Use interfaces/traits for that purpose!

You should **not** write:

```
val button: AbstractList = new LinkedList()
```

Abstract class

Does it make sense to create an AbstractList object?

```
abstract class AbstractList extends List{
  ...
}
```

an abstract class cannot be instantiated

What about all the other methods of List?

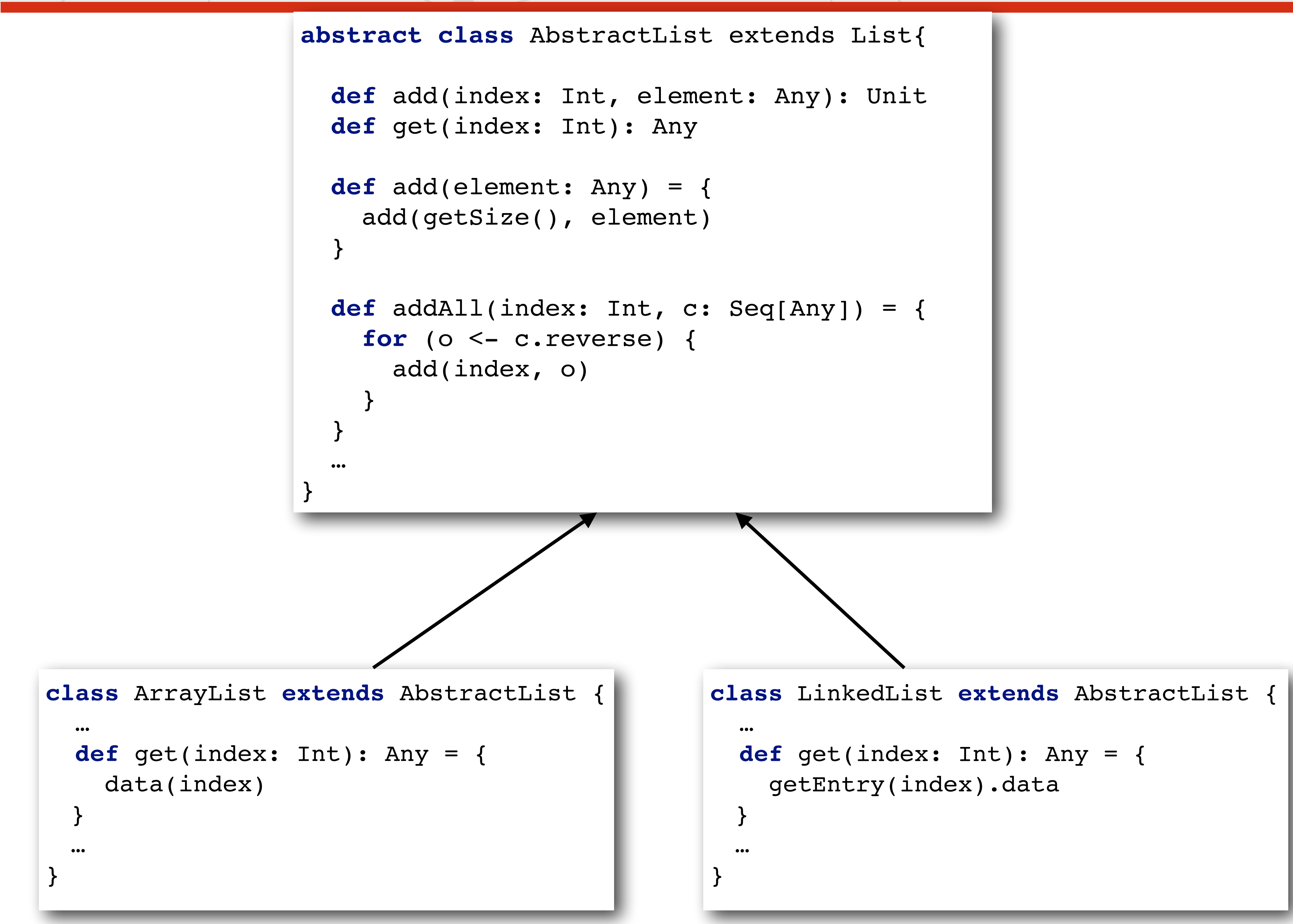
```
abstract class AbstractList extends List{
  def add(index: Int, element: Any): Unit
  def get(index: Int): Any
 ...
}
```

abstract methods MUST be implemented (or inherited) by all concrete subclasses

Transitivity of Requirements

A concrete class must implement (or inherit implementations for):

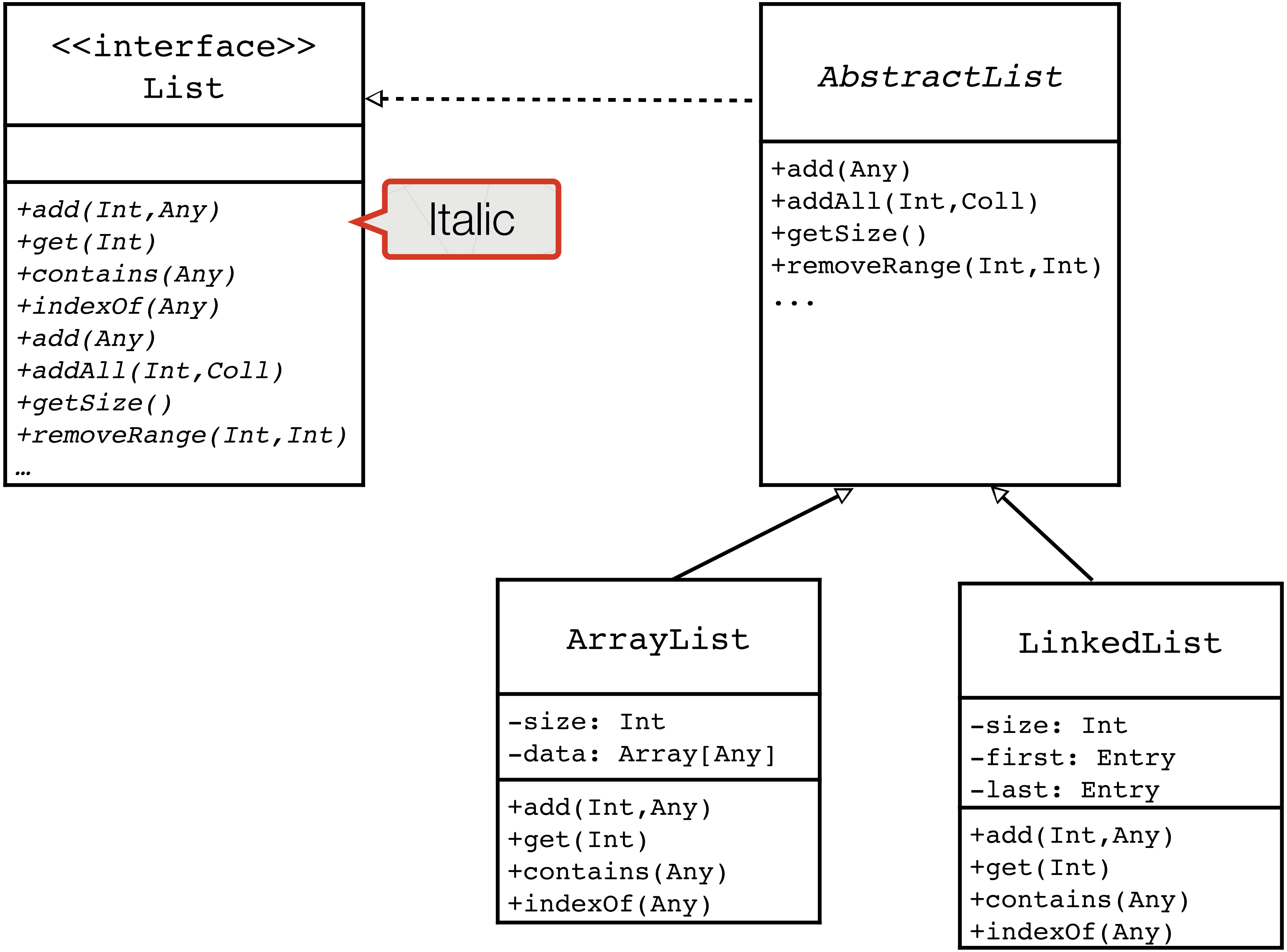
- the methods of the interfaces it implements
- the methods of the interfaces from which these interfaces extend (transitively)
- all the abstract methods it inherits, for which no superclass provides an implementation



```
abstract class AbstractList extends List{

  def add(index: Int, element: Any): Unit
  def get(index: Int): Any

  def add(element: Any) = {
    add(getSize(), element)
  }

  def addAll(index: Int, c: Seq[Any]) = {
    for (o <- c.reverse) {
      add(index, o)
    }
  }
  …
}
```

```
class ArrayList extends AbstractList {
  …
  def get(index: Int): Any = {
    data(index)
  }
  …
}
```

```
class LinkedList extends AbstractList {
  …
  def get(index: Int): Any = {
    getEntry(index).data
  }
  …
}
```



<<interface>>
List

+add(Int,Any)
+get(Int)
+contains(Any)
+indexOf(Any)
+add(Any)
+addAll(Int,Coll)
+getSize()
+removeRange(Int,Int)
…

Italic

AbstractList

+add(Any)
+addAll(Int,Coll)
+getSize()
+removeRange(Int,Int)
...

ArrayList

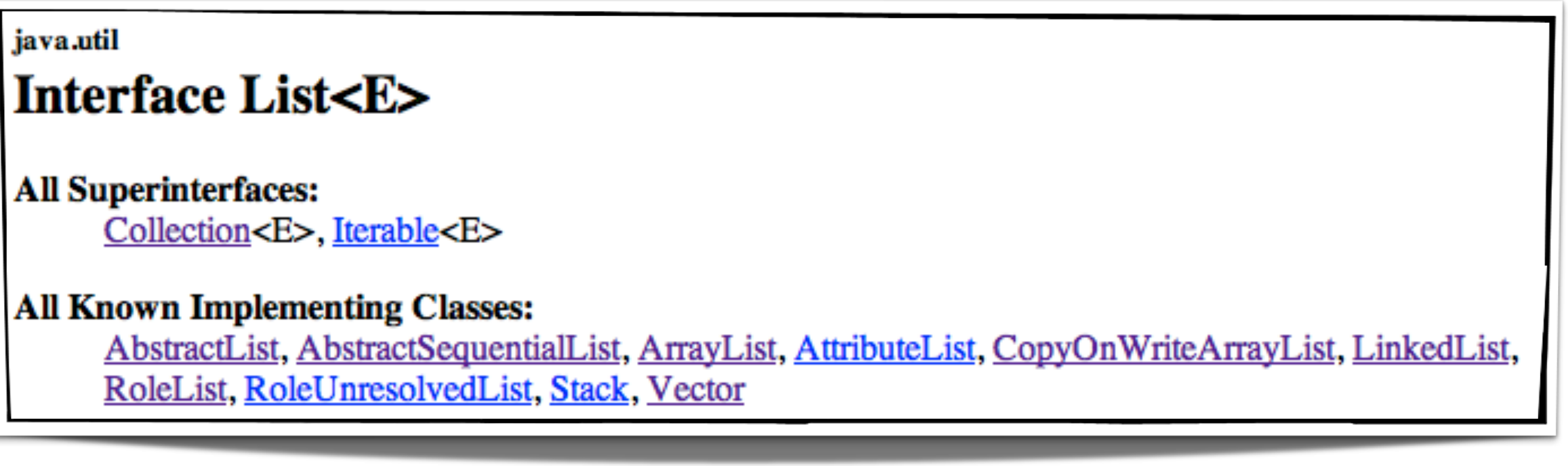
-size: Int
-data: Array[Any]

+add(Int,Any)
+get(Int)
+contains(Any)
+indexOf(Any)

LinkedList

-size: Int
-first: Entry
-last: Entry

+add(Int,Any)
+get(Int)
+contains(Any)
+indexOf(Any)



java.util

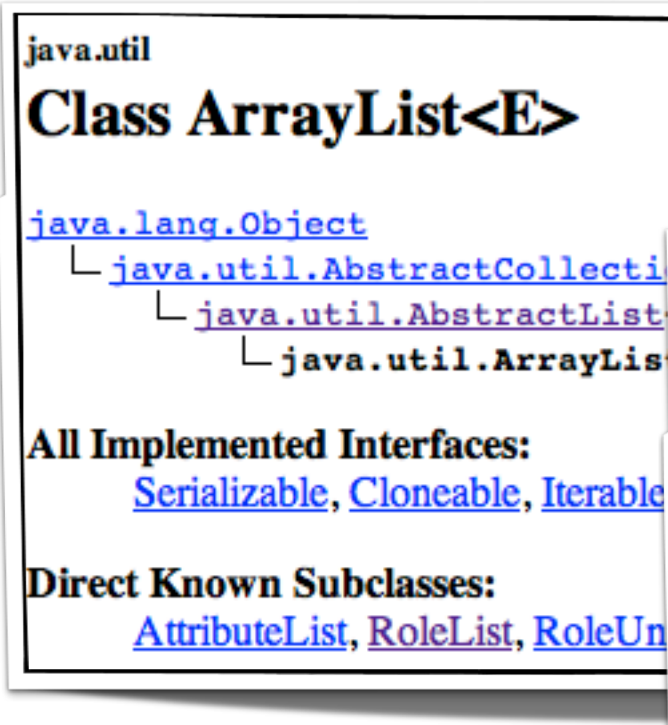
Interface List<E>

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector



java.util

Class ArrayList<E>

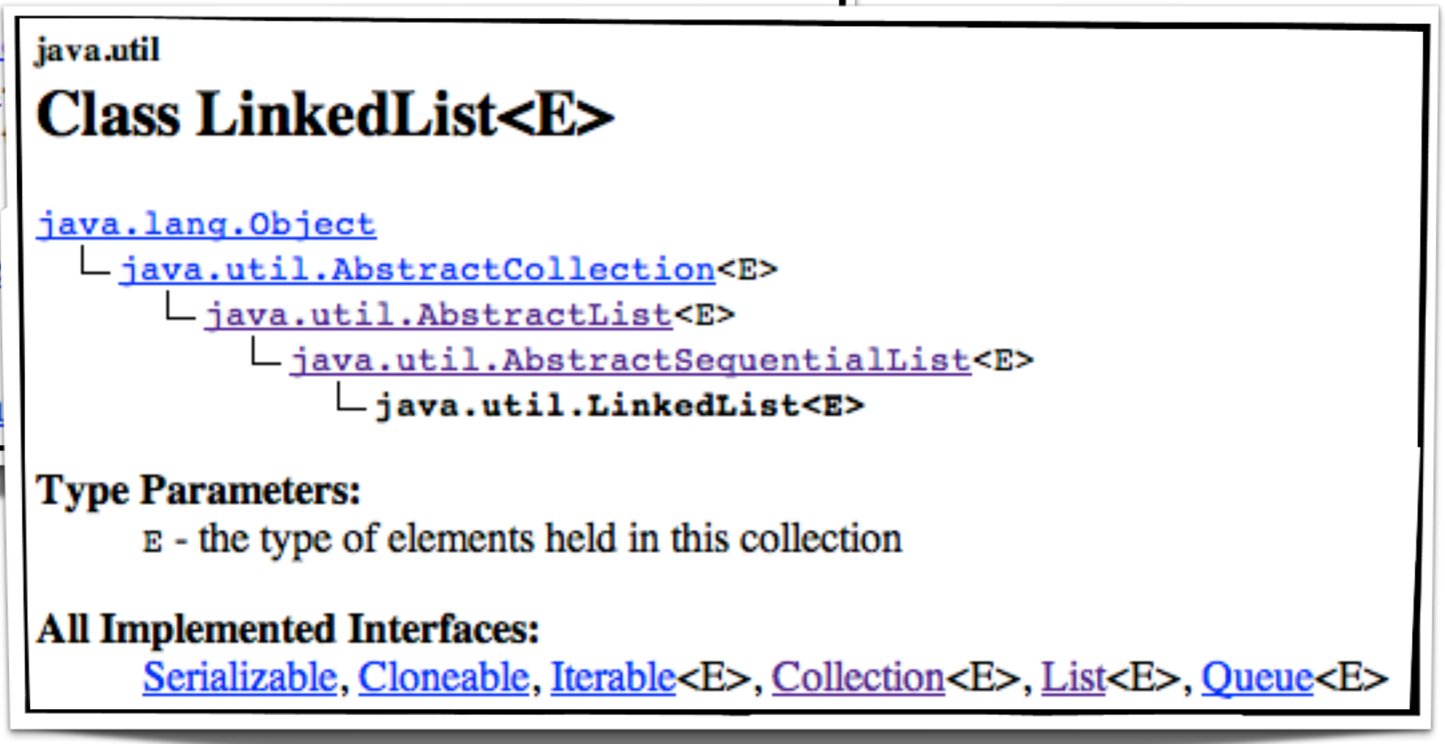
```
java.lang.Object
  └ java.util.AbstractCollecti
      └ java.util.AbstractList
          └ java.util.ArrayLis
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable

Direct Known Subclasses:

AttributeList, RoleList, RoleUn



java.util

Class LinkedList<E>

```
java.lang.Object
  └ java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.AbstractSequentialList<E>
              └ java.util.LinkedList<E>
```

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, Queue<E>

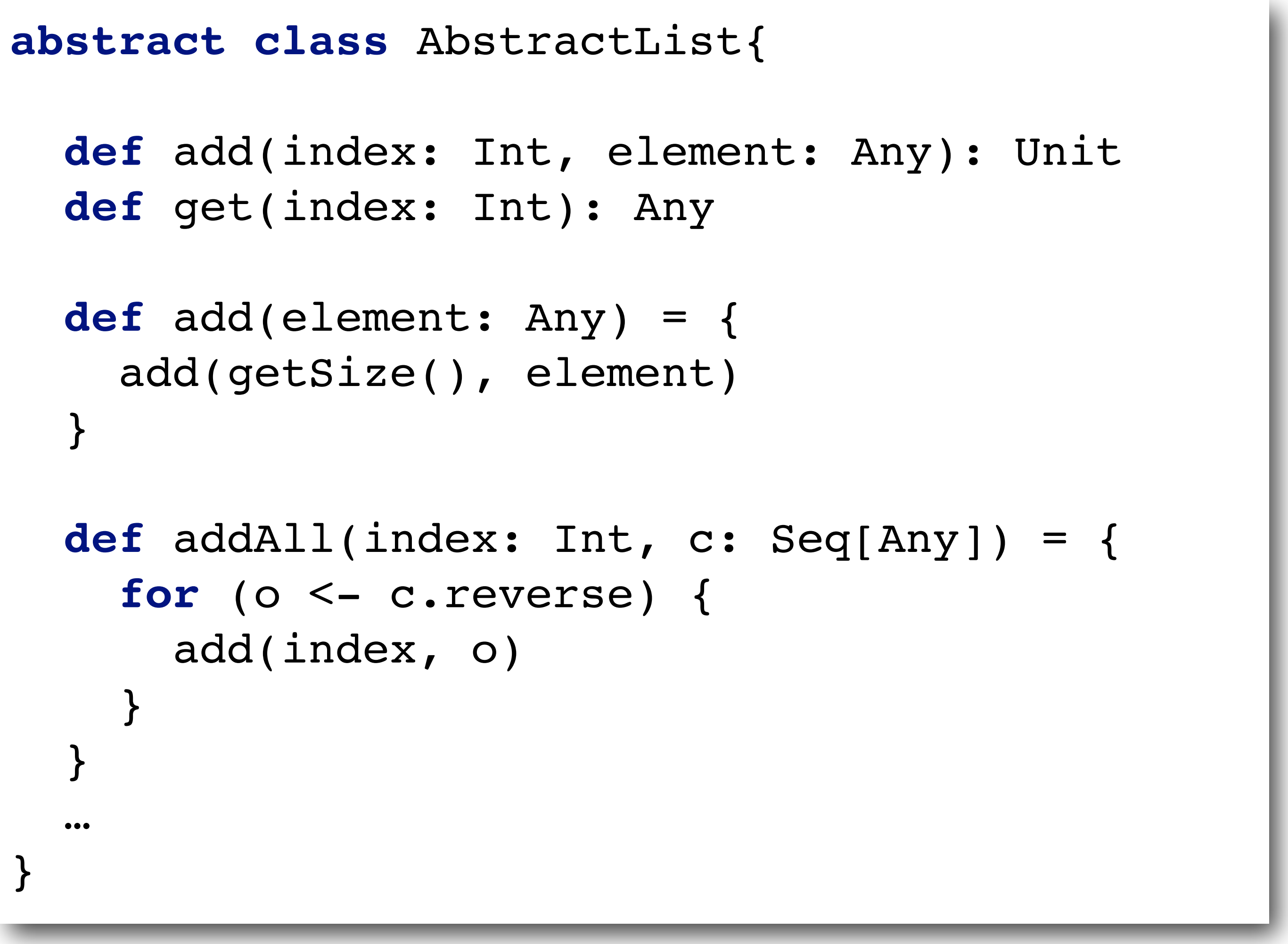
Abstract Classes

No instances -- just useful because of inheritance

As superclasses, they gather common implementation of their subclasses

Different kinds of methods

- abstract
- concrete (self-contained)
- hybrid: implementation that relies on abstract methods



```
abstract class AbstractList{

  def add(index: Int, element: Any): Unit
  def get(index: Int): Any

  def add(element: Any) = {
    add(getSize(), element)
  }

  def addAll(index: Int, c: Seq[Any]) = {
    for (o <- c.reverse) {
      add(index, o)
    }
  }
  …
}
```

abstract?

concrete?

hybrid?

Design Process

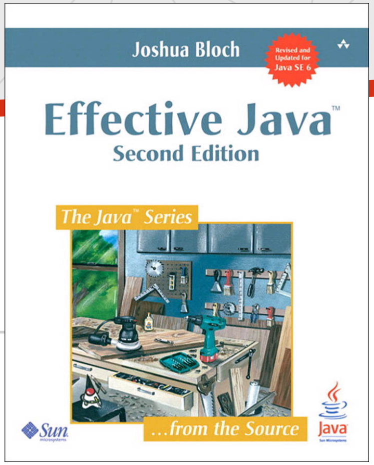
Incremental, learn through experience

- start with interface
- develop the implementation that is needed
- if other implementations needed, code them and factor out commonalities in an abstract class
- refine... and possibly introduce intermediate abstract classes

NOTE:

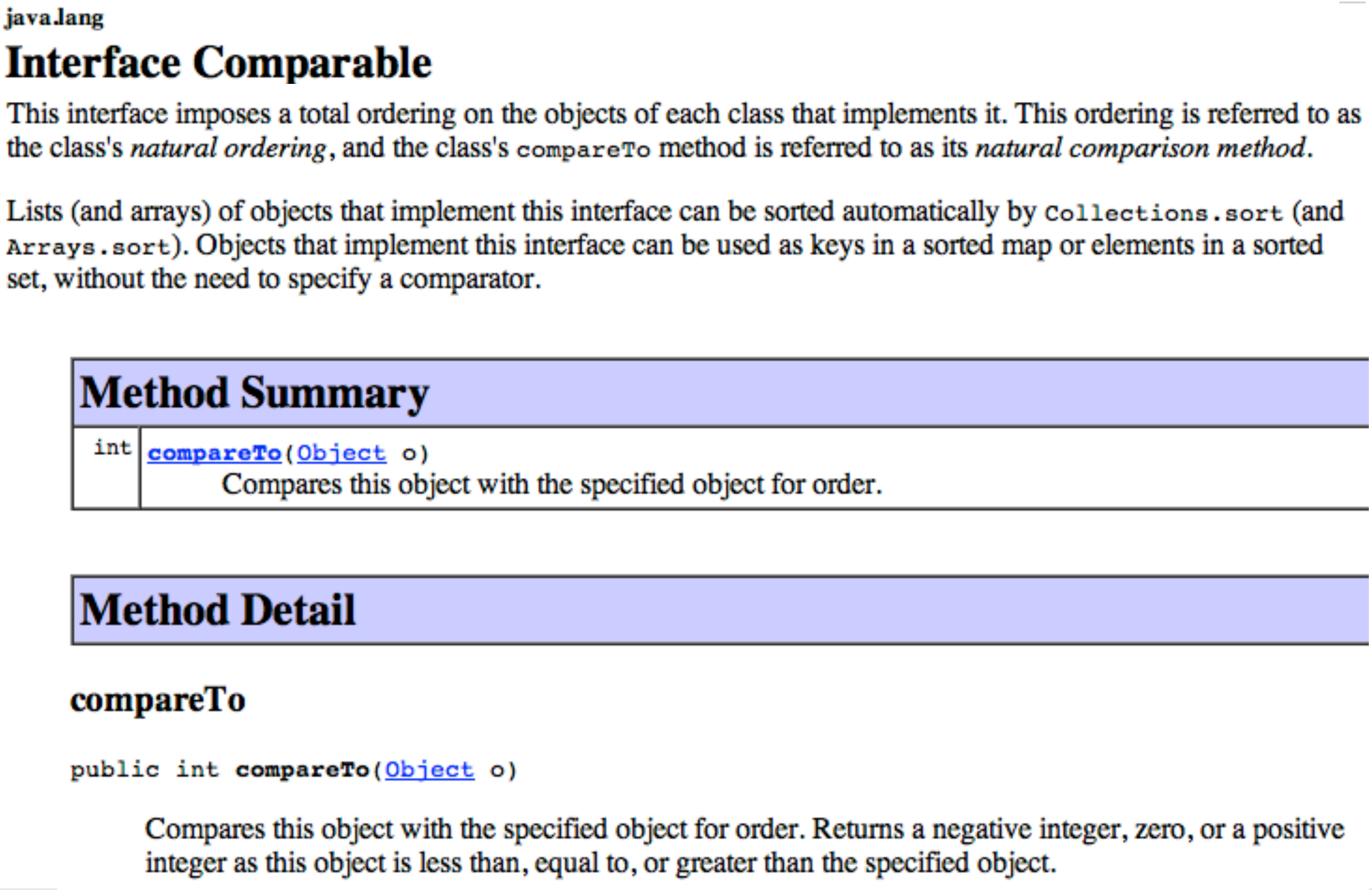
A purely abstract class
(no fields, only abstract methods)
is “similar” to an interface

EffectiveJava #18



- Prefer interfaces to abstract classes
  - easy to retrofit a class to implement a new interface
  - interfaces are ideal for mixins (additional, optional behavior, like Comparable, Serializable...)
  - interfaces support non-hierarchical frameworks
- Best of both worlds
  - interface
  - skeletal implementation class (abstract)

The Comparable “Mixin”



java.lang

Interface Comparable

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

Method Summary

`int`	**`compareTo`**`(Object o)` Compares this object with the specified object for order.

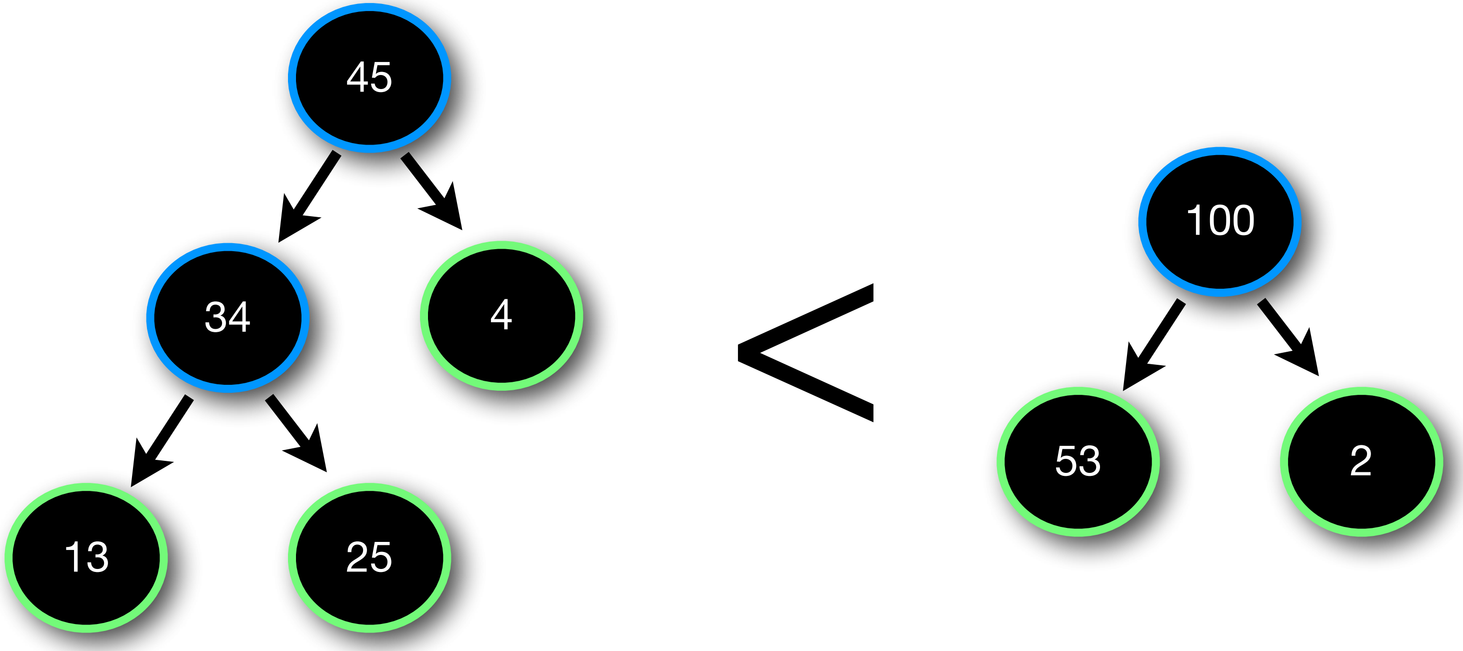
Method Detail

compareTo

```
public int compareTo(Object o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Comparable

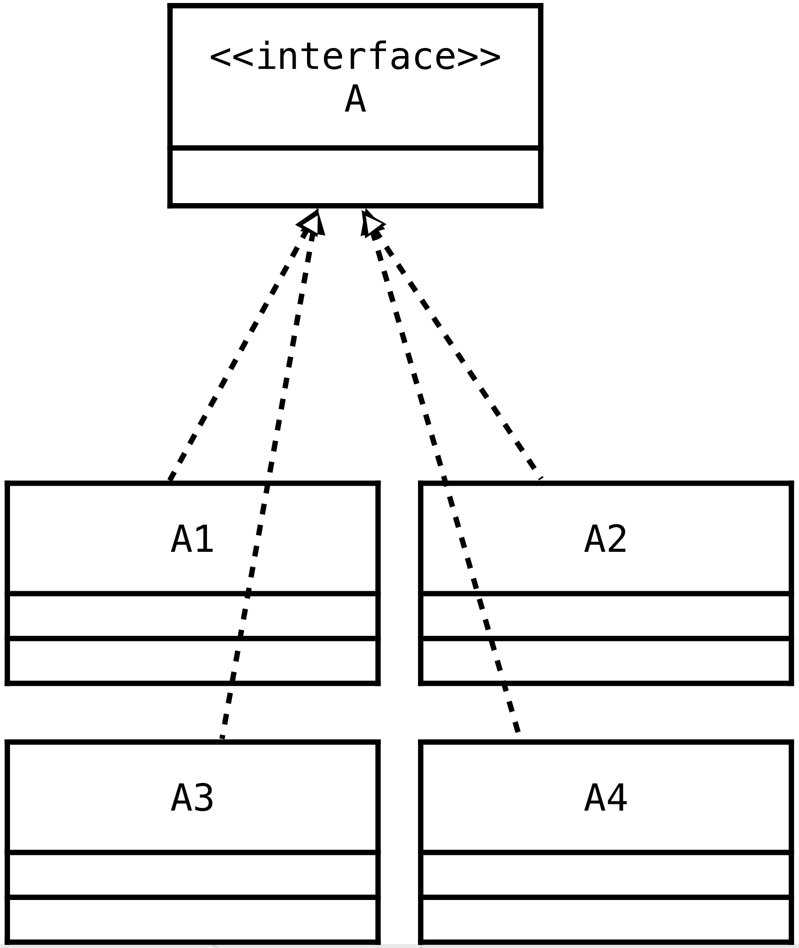


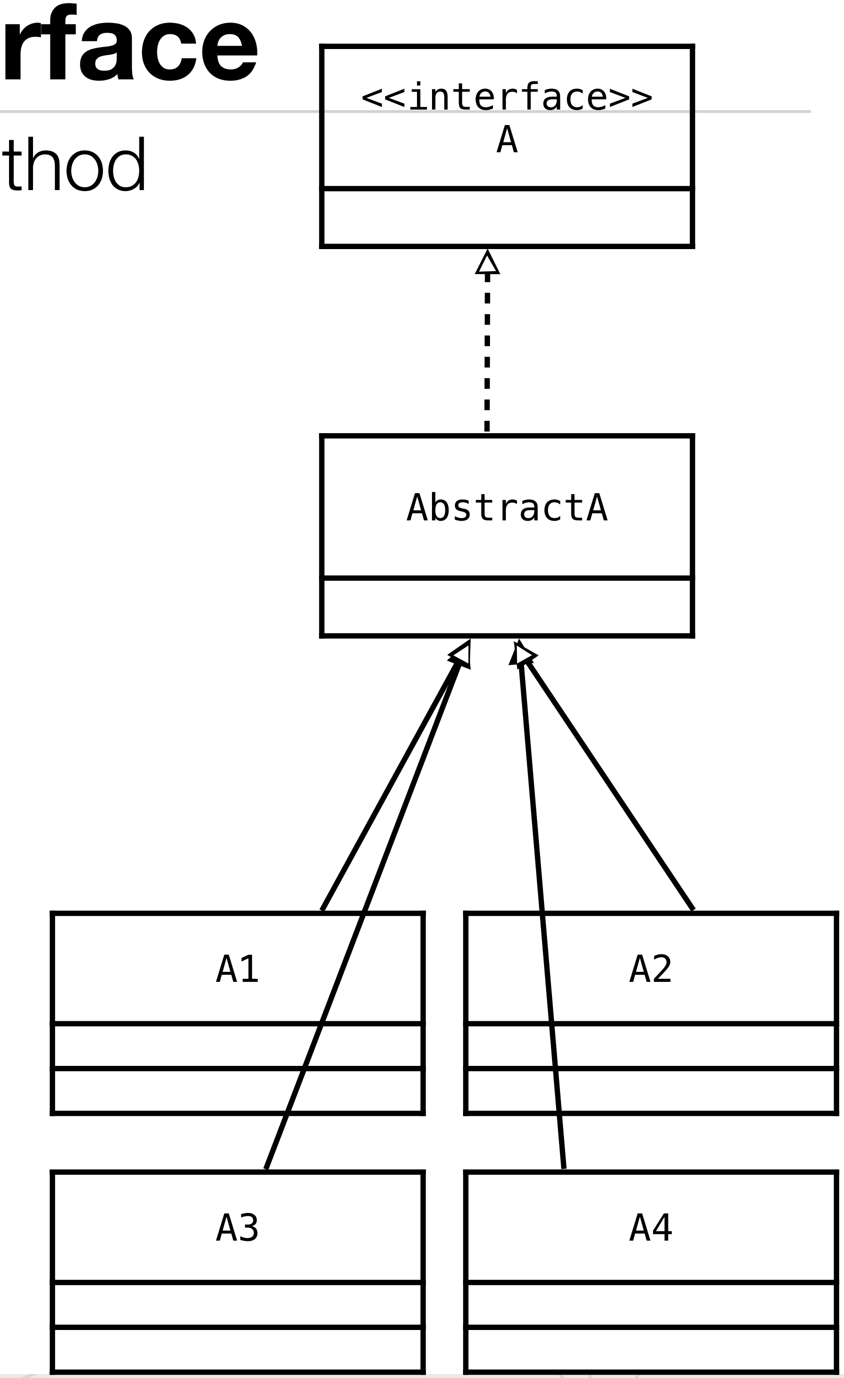
Do it!

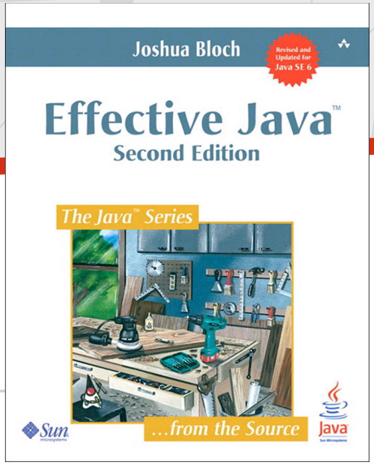
Abstract Class vs. Interface

what if we need to add a new method to the interface?

```
interface TreeNode  extends Comparable
```







EffectiveJava #18 continued

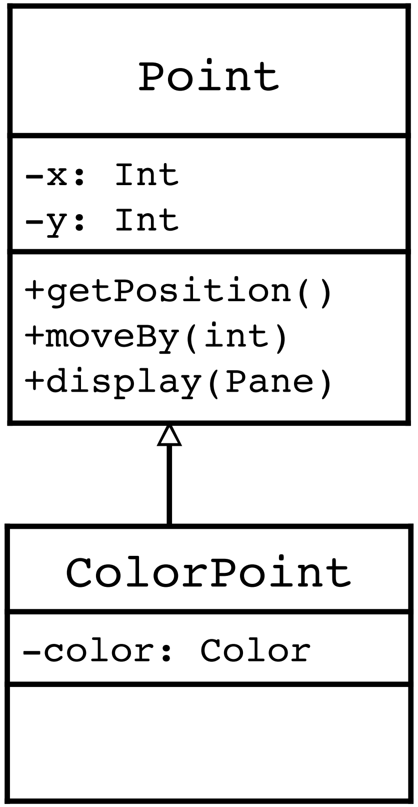
There is one advantage of abstract classes

- can add a new method without breaking existing subclasses
- evolving an interface is much more troublesome

Extension and Refinement

A class can not only extend traits or abstract classes. It can also extends other classes.

If class A extends class B, then we say that A is a subclass and B is a superclass



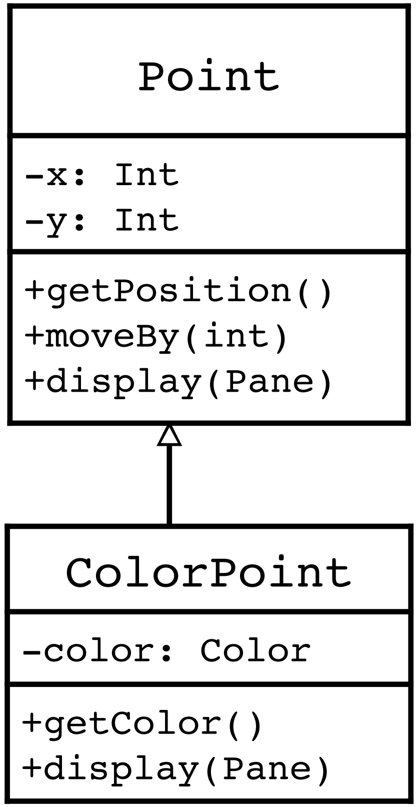
Extension and Refinement

A subclass can do more than implement abstract methods!

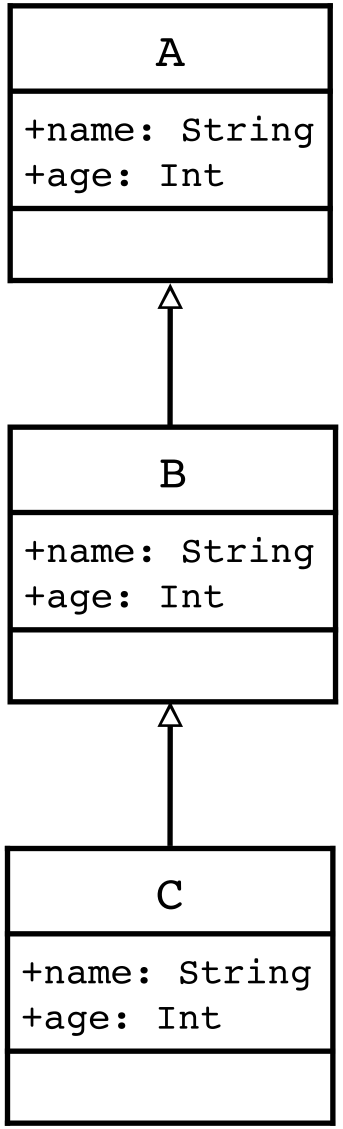
it can define new methods

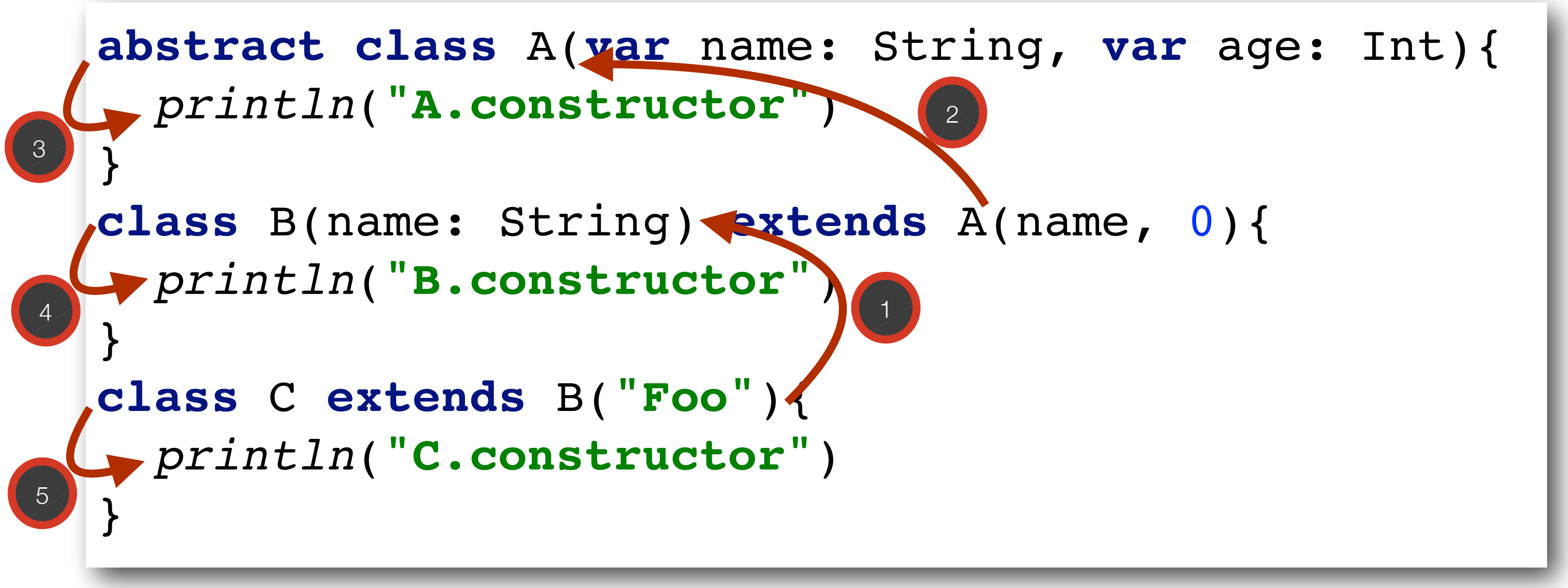
and refine existing behavior (more on this next class)

```
val p = new Point(10, 10)
p.display(thisSlide)

val cp = new ColorPoint(20, 10, RED)
cp.display(thisSlide)
```



Superclass constructors are sequentially executed

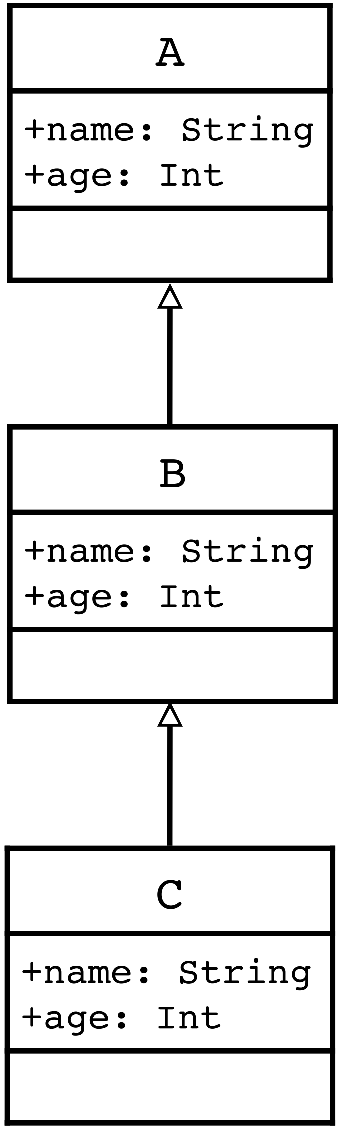


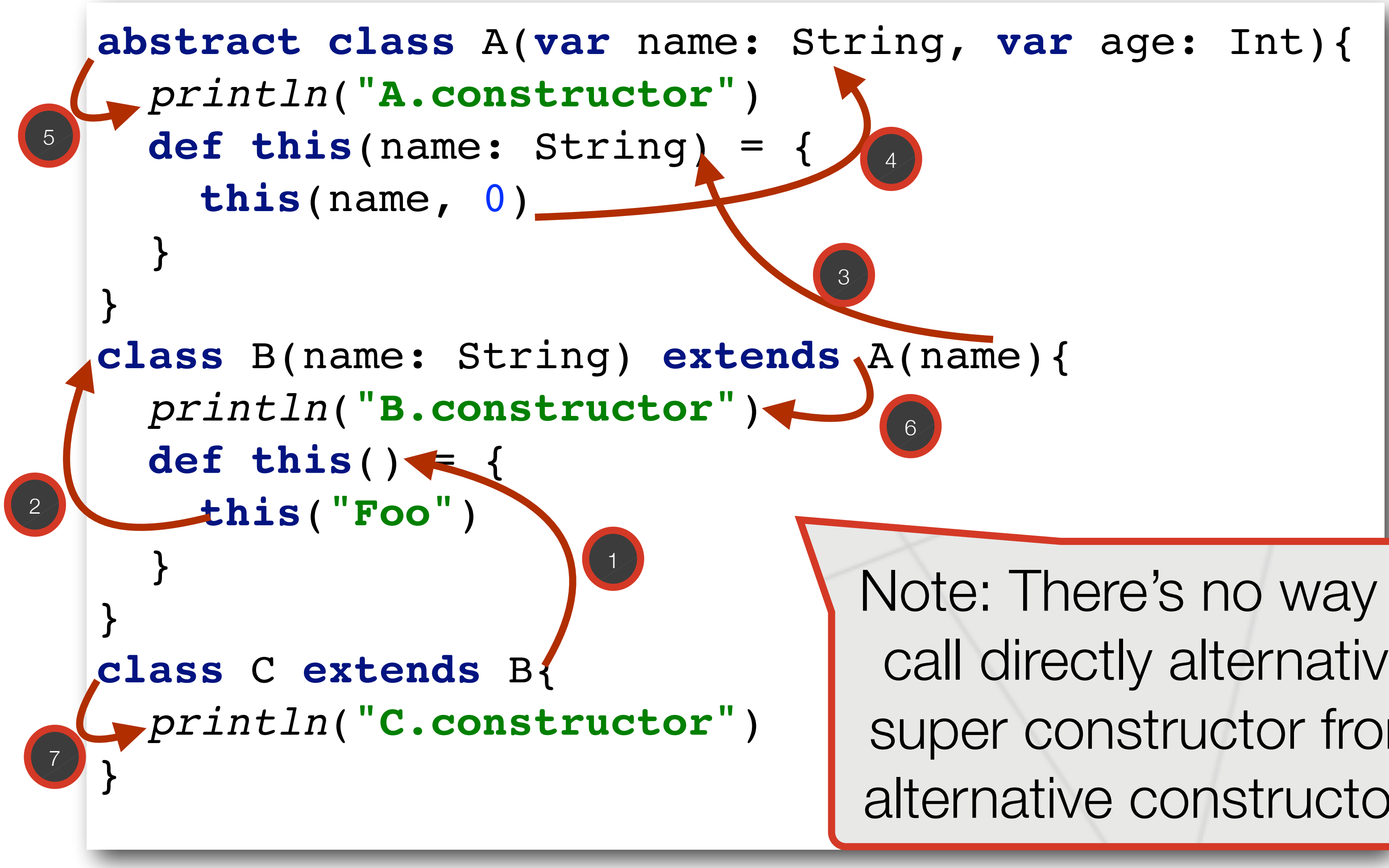


```
abstract class A(var name: String, var age: Int){
  println("A.constructor")
}
class B(name: String) extends A(name, 0){
  println("B.constructor")
}
class C extends B("Foo"){
  println("C.constructor")
}
```

```
new C()
```

Superclass constructors are sequentially executed

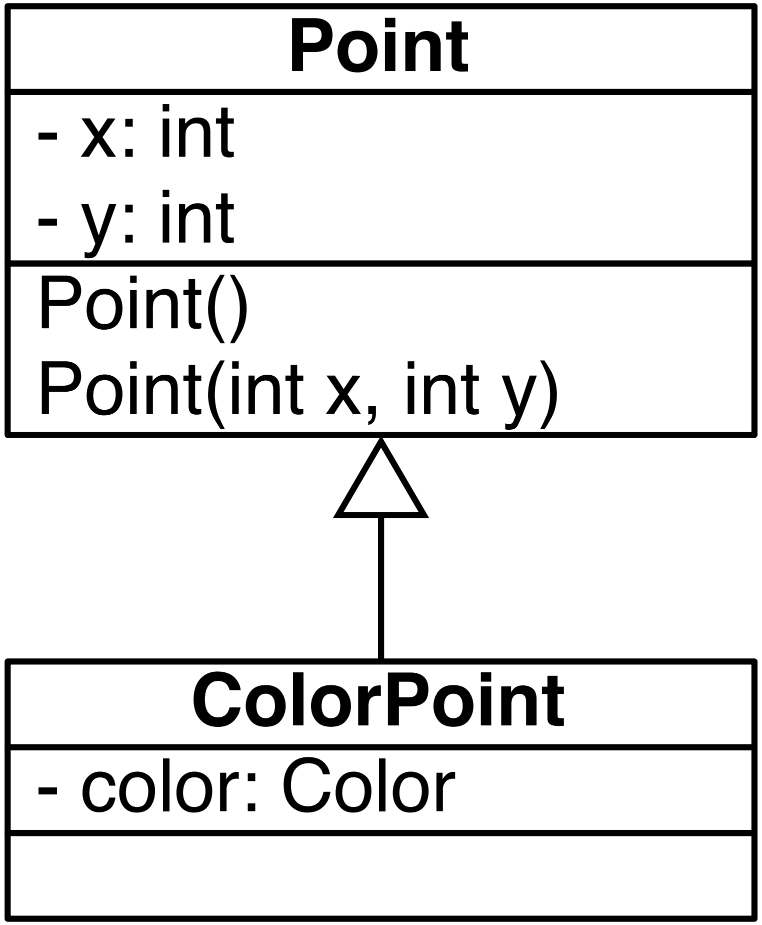




```
abstract class A(var name: String, var age: Int){
  println("A.constructor")
  def this(name: String) = {
    this(name, 0)
  }
}
class B(name: String) extends A(name){
  println("B.constructor")
  def this() = {
    this("Foo")
  }
}
class C extends B{
  println("C.constructor")
}
```

Note: There's no way to call directly alternative super constructor from alternative constructors

Constructor are not inherited!



Point

- x: int
- y: int

Point()
Point(int x, int y)

ColorPoint

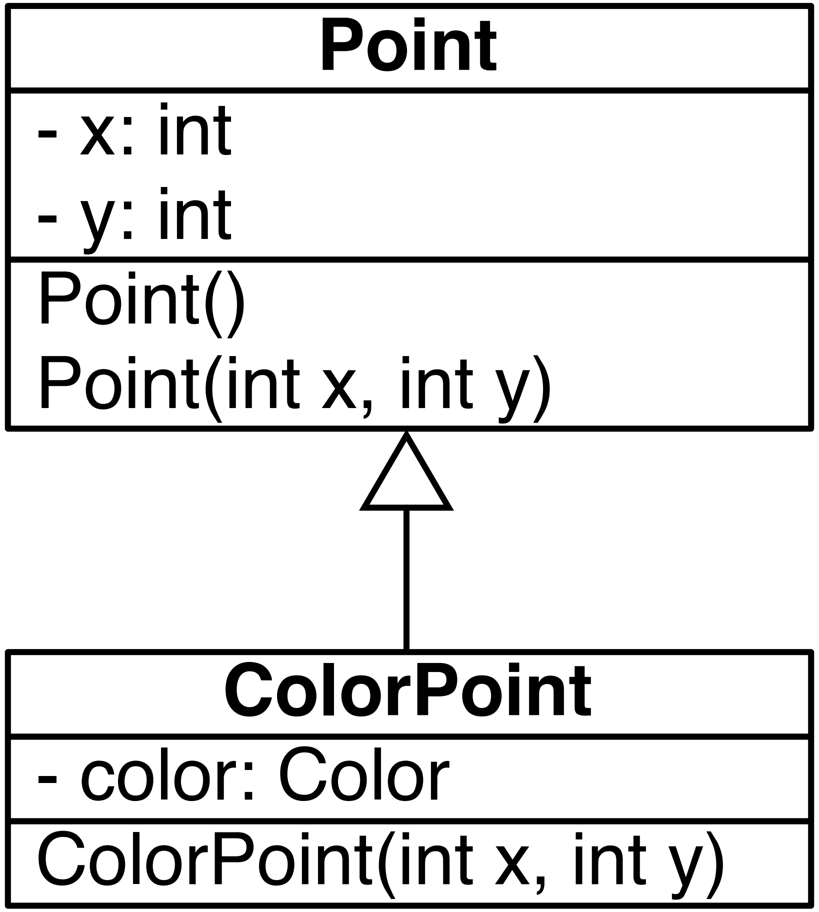
- color: Color

```
new Point()
new Point(2,3)
=> Okay
```

```
new ColorPoint()
=> Okay (because of the
default constructor)
```

```
new ColorPoint(2,3)
=> Does not compile
```

Missing default constructor



Point

- x: int
- y: int

Point()
Point(int x, int y)

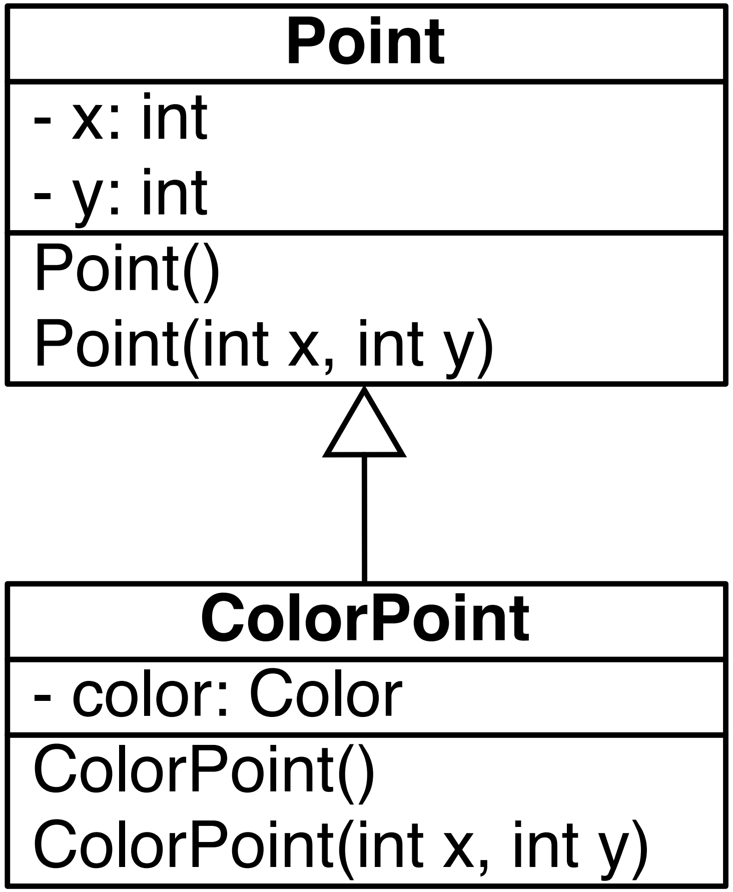
ColorPoint

- color: Color

ColorPoint(int x, int y)

```
new Point()
new Point(2,3)
=> Okay


new ColorPoint()
=> Does not compile
(because there is no
default constructor)

new ColorPoint(2,3)
=> Okay
```

Missing default constructor



Point

- x: int
- y: int

Point()
Point(int x, int y)

ColorPoint

- color: Color

ColorPoint()
ColorPoint(int x, int y)

```
new Point()
new Point(2,3)
=> Okay
```

```
new ColorPoint()
=> Okay
```

```
new ColorPoint(2,3)
=> Okay
```

License

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)
You are free to:
- Share: copy and redistribute the material in any medium or format
- Adapt: remix, transform, and build upon the material for any purpose, even commercially

The licensor cannot revoke these freedoms as long as you follow the license terms

Attribution: you must give appropriate credit

ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original

Complete license: https://creativecommons.org/licenses/by-sa/4.0/

dcc

CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

www.dcc.uchile.cl

/ DCCUCHILE