Imperative Programming 2:
Traits

Hongseok Yang
University of Oxford

Experience with traits so far

- In IP1, you have used traits as interfaces, in order to specify available methods and fields.

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
}

import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

Experience with traits so far

- In IP1, you have used traits as interfaces, in order to specify available methods and fields.

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
}
```

```
class ListQueue extends IntQueue {
  private var l: List[Int] = List()
  def get() = ...
  def put(x: Int) { ... }
}
```

```
import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

Experience with traits so far

- In IP1, you have used traits as interfaces, in order to specify available methods and fields.

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
}
```

```
class ListQueue extends IntQueue {
  private var l: List[Int] = List()
  def get() = ...
  def put(x: Int) { ... }
}
```

```
import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

- But traits are more powerful. This power is frequently used in practice.

Today's lecture

- We will study powerful features of traits, and discuss about idioms of using them.

Features of traits

1. We can define methods and fields in a trait.
2. Multiple traits can be inherited simultaneously.
3. They support so called stackable modification, a powerful mechanism for composing traits.

1. We can define methods and fields in a trait.

1. We can define methods and fields in a trait.

- [Q] Add "`putL(xs:List[Int]){..}`".

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
}

import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

class ListQueue extends IntQueue {
  private var l: List[Int] = List()
  def get() = { val res = l.head; l = l.tail; res }
  def put(x: Int) { l = l :+ x }
}
```

1. We can define methods and fields in a trait.

- [Q] Add "`putL(xs:List[Int]){..}`".

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
  def putL(xs: List[Int]) { for(x<-xs) put(x) }
}

import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}

class ListQueue extends IntQueue {
  private var l: List[Int] = List()
  def get() = { val res = l.head; l = l.tail; res }
  def put(x: Int) { l = l :+ x }
}
```

1. We can define methods and fields in a trait.

- [Q] Add "`putL(xs:List[Int]){..}`".
- [Q] What's the problem of this alternative?

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
  def putL(xs: List[Int]): Unit
}
import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
  def putL(xs: List[Int]) { but ++= xs }
}
class ListQueue extends IntQueue {
  private var l: List[Int] = List()
  def get() = { val res = l.head; l = l.tail; res }
  def put(x: Int) { l += x }
  def putL(xs: List[Int]) { l ++= xs }
}
```

1. We can define methods and fields in a trait.

- [Q] Add "`putL(xs:List[Int]){..}`".
- [Q] What's the problem of this alternative?
- By defining methods in a trait, we can easily support rich interfaces without duplicating code.

By defining N functions, we can add 2xN methods

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
  def putL(xs: List[Int]) { for(x<-xs) put(x) }
  def putUpto(upper: Int) { for(x<- 1 to upper) put(x) }
  def remove(n: Int) { for(i<-1 to n) get() }
  ...
}
class ArrayQueue extends IntQueue ...

class ListQueue extends IntQueue ...
```

1. We can define methods and fields in a trait.
2. Multiple traits can be inherited simultaneously.
   - Syntax: `... extends T0 with T1 ... with Tn`

1. We can define methods and fields in a trait.

2. Multiple traits can be inherited simultaneously.

   - Syntax: `... extends T0 with T1 ... with Tn`

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
  def putL(xs: List[Int]) { for(x<-xs) put(x) }
}
trait Ordered[A] {
  def compare(that: A): Int
  def <(that: A): Boolean = (this compare that) <  0
  def >(that: A): Boolean = (this compare that) >  0
  def <=(that: A): Boolean = (this compare that) <= 0
}
import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue with Ordered[ArrayQueue]{
  private val buf = new ArrayBuffer[int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
  def compare(that: ArrayQueue) = buf.length - that.buf.length
}
```

1. We can define methods and fields in a trait.
2. Multiple traits can be inherited simultaneously.
   - Syntax: `... extends T0 with T1 ... with Tn`
   - The same syntax for trait definition.
   - Terminology: We mix in `Ti`'s.
   - Methods in `T`, `T1`, ... and `Tn` are combined in an intricate way.
   - In particular, `super.meth(..)` in a trait has a special semantics that affects this combination.
   - This is exploited in the stackable modification pattern.

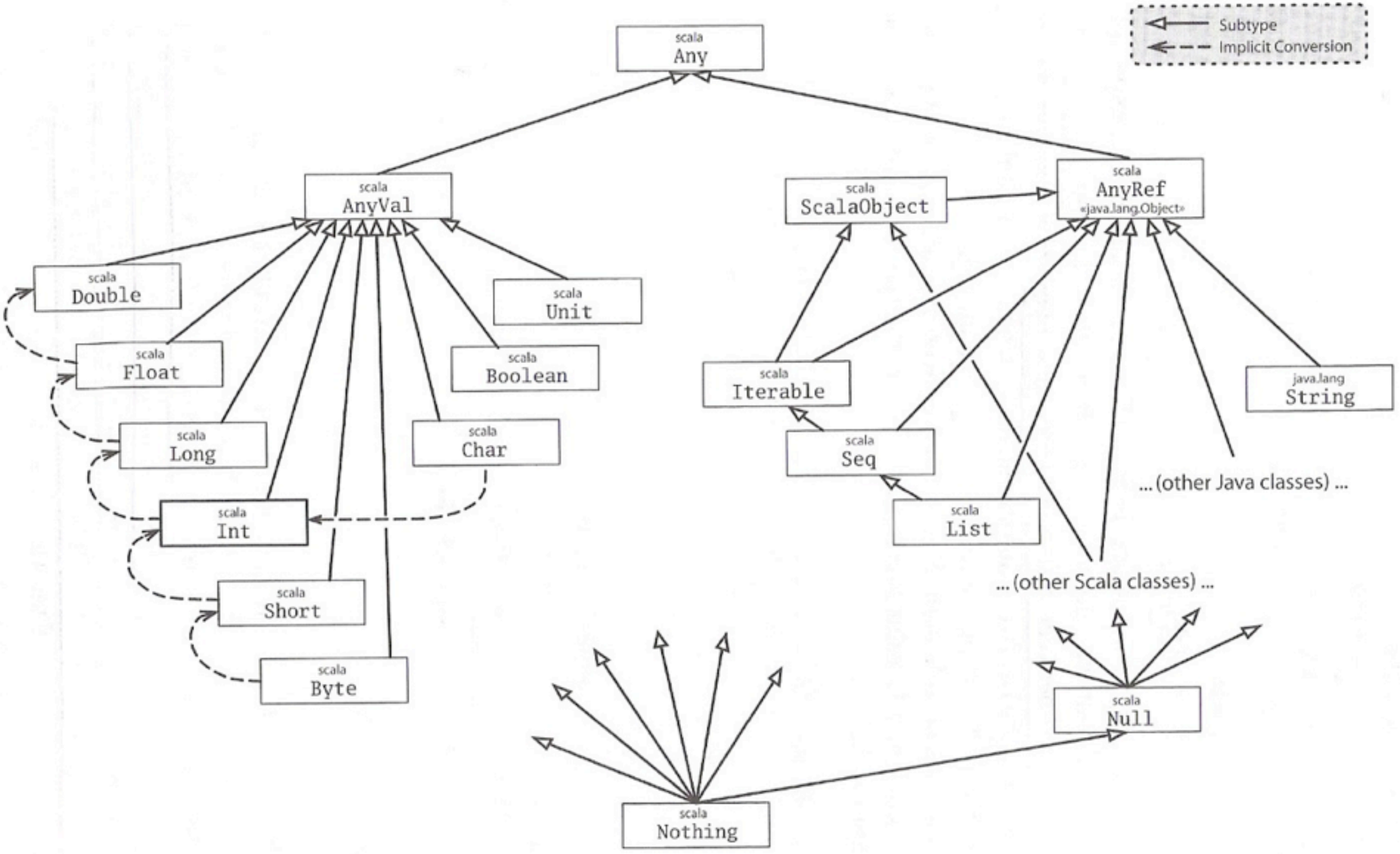
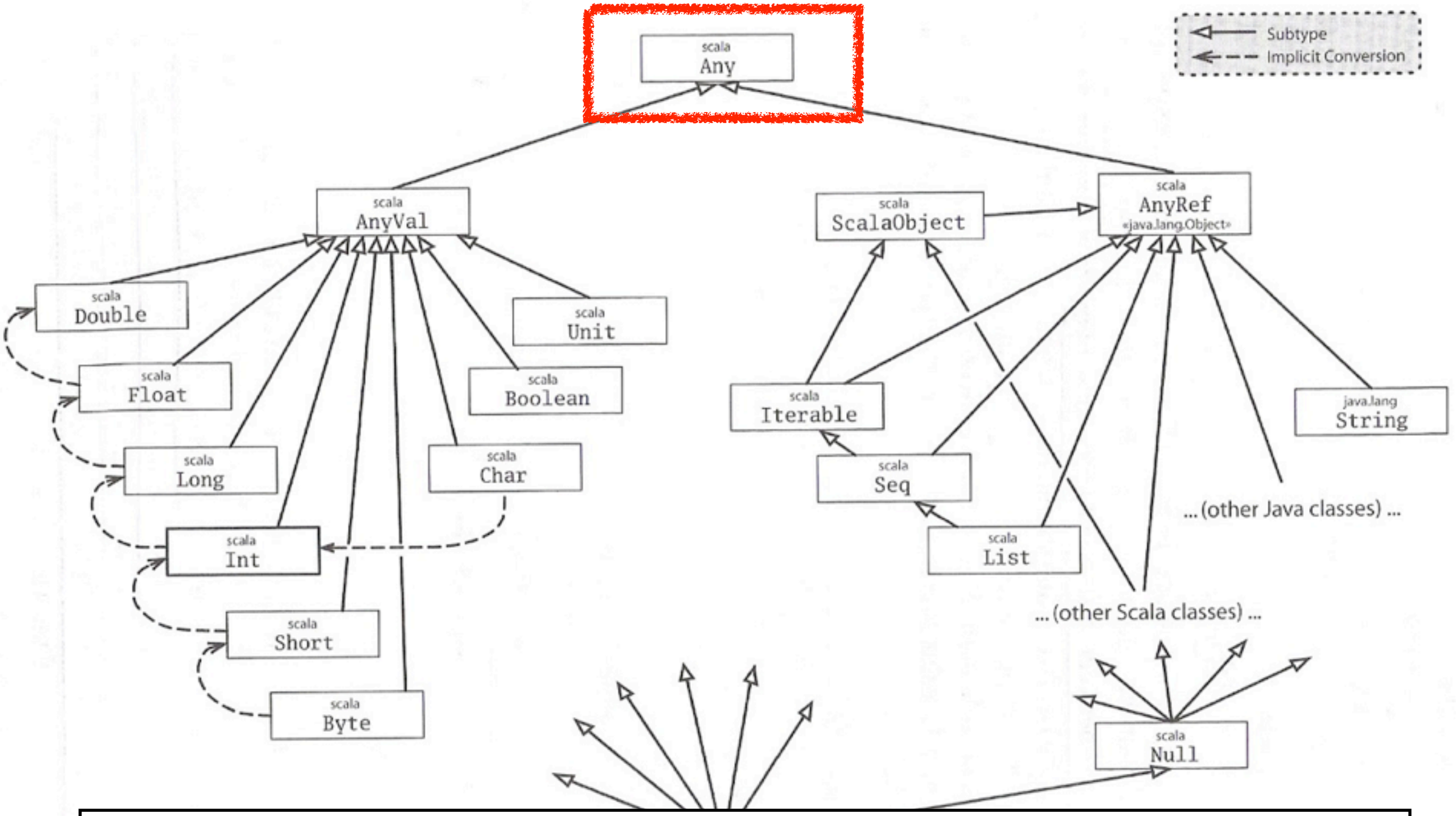
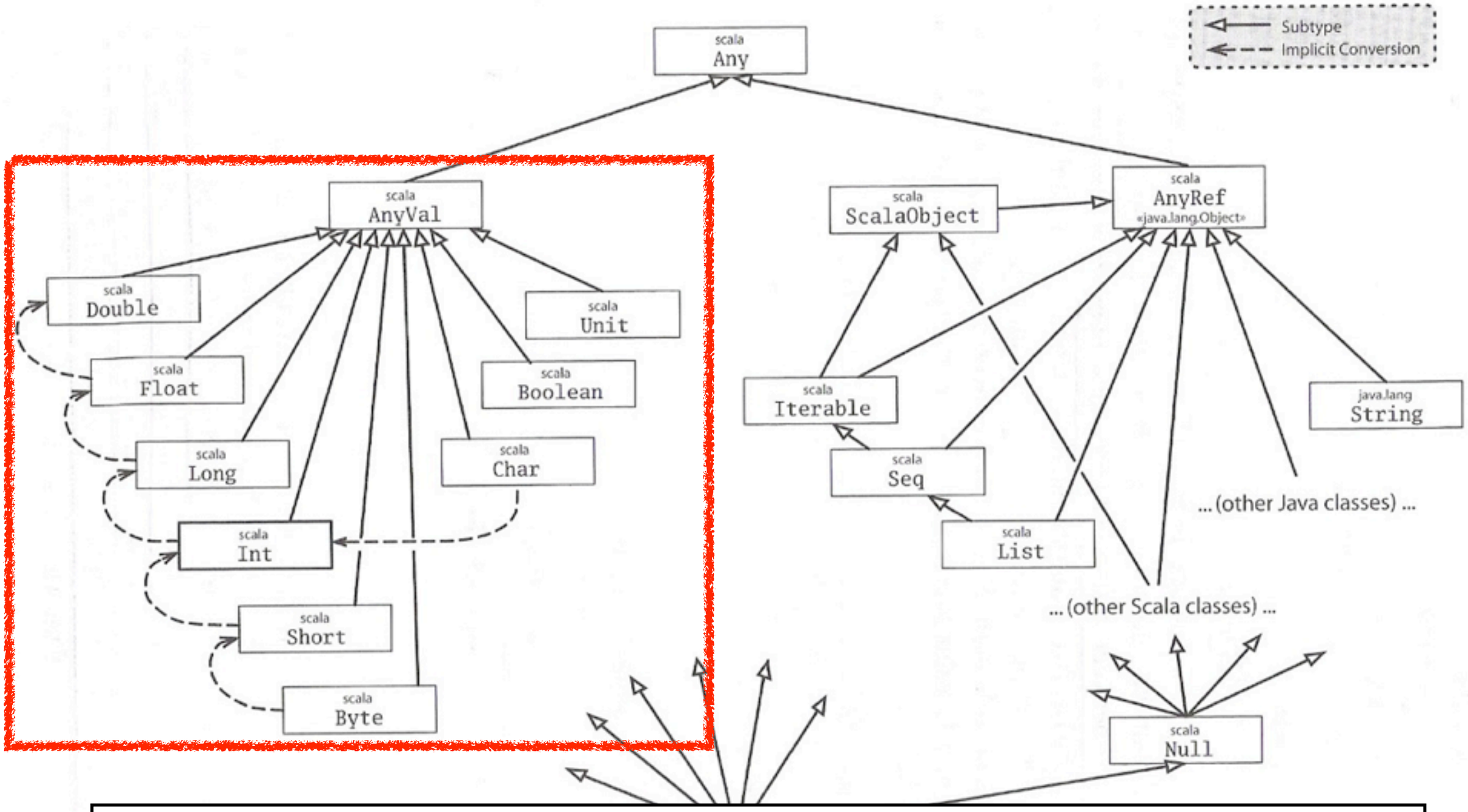


Figure 11.1 · Class hierarchy of Scala.

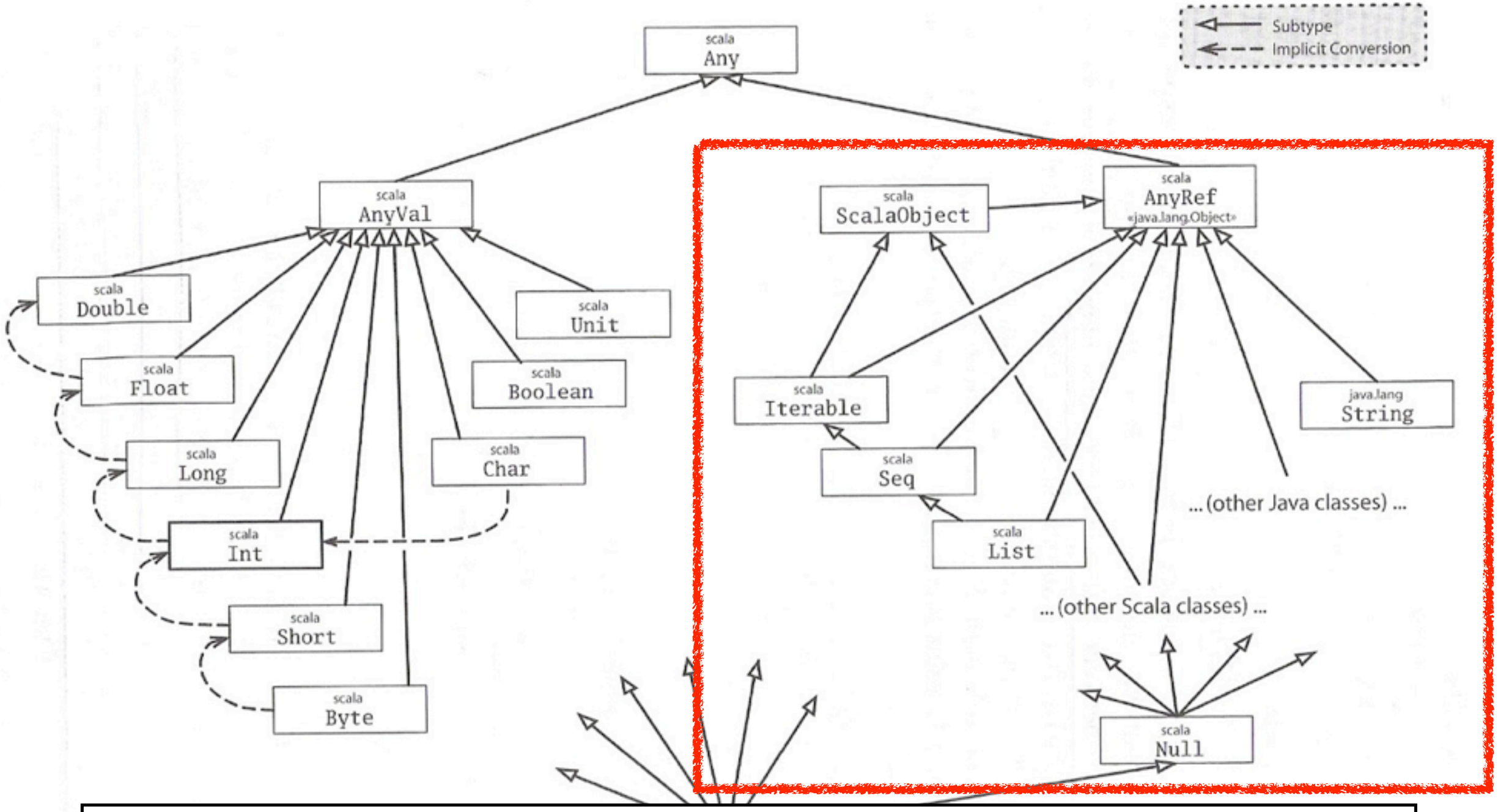


Any is a superclass of everything. It defines basic methods, like ==, !=, equals, hashCode, etc.



Any is a superclass of everything. It defines basic methods, like ==, !=, equals, hashCode, etc.

AnyVal is a superclass of all values.



Any is a superclass of everything. It defines basic methods, like ==, !=, equals, hashCode, etc.

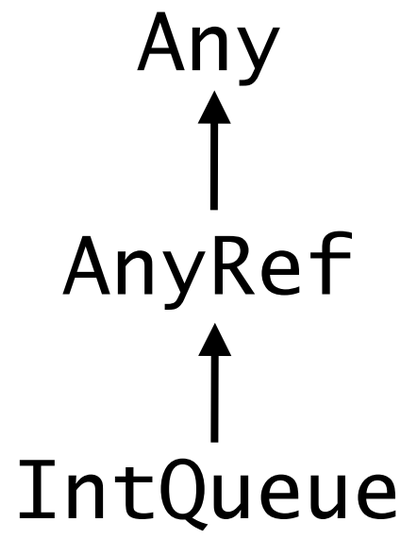
AnyVal is a superclass of all values.

AnyRef is a superclass of all user-defined classes.

Default trait inheritance

- If the "extend' clause is missing, a trait inherits from the Scala class `AnyRef`.

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
  def putL(xs: List[Int]) { for(x<-xs) put(x) }
}
```



Linearization

```
class C extends T0 with T1 ... with Tn
trait T extends T0 with T1 ... with Tn
```

- Linearization determines a linear order of all inherited classes and traits by `C` and `T`.
- Recursive algorithm : Handle `T0`, ... then `Tn`, finally `C/T`.
- Recursive step : Extend the current linearization by adding classes and traits that are inherited by `Ti` (including `Ti`) but not linearized so far.

```
trait Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Linearization

```
class C extends T0 with T1 ... with Tn
trait T extends T0 with T1 ... with Tn
```

- Linearization determines a linear order of all inherited classes and traits by `C` and `T`.
- It decides methods to be called.

```
trait Animal { def f(){ println("A") } }
trait Furry extends Animal { override def f(){ println("F") } }
trait HasLegs extends Animal { override def f(){ println("H") } }
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
(new Cat).f()
```

super.m in a trait

- `super.m` in a trait `T` is bound not when the trait `T` is defined, but when it is mixed in.
- `super.m` is resolved according to linearization.
- It refers to the most recent `m` defined in a class or trait before `T` in the linearization.

```
trait Animal { def f(){ println("A") } }
trait Furry extends Animal {
  override def f(){ super.f(); println("F") } }
trait HasLegs extends Animal {
  override def f(){ super.f(); println("H") } }
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
(new Cat).f()
```

super.m in a trait

- Use the keyword "abstract override" when `m` is not defined in a superclass or supertrait of `T`.

```
trait Animal { def f() }
trait NormalAnimal extends Animal { def f() { println("NA") } }
trait Furry extends Animal {
  abstract override def f(){ super.f(); println("F") } }
trait HasLegs extends Animal {
  abstract override def f(){ super.f(); println("H") } }
trait FourLegged extends HasLegs
class Cat extends NormalAnimal with Furry with FourLegged
(new Cat).f()
```

super.m in a trait

Stackable modification:
Implement a new functionality by mixing in traits with super calls in a particular order.

- Use the keyword "abstract override" when `m` is not defined in a superclass or supertrait of `T`.

```
trait Animal { def f() }
trait NormalAnimal extends Animal { def f() { println("NA") } }
trait Furry extends Animal {
  abstract override def f(){ super.f(); println("F") } }
trait HasLegs extends Animal {
  abstract override def f(){ super.f(); println("H") } }
trait FourLegged extends HasLegs
class Cat extends NormalAnimal with Furry with FourLegged
class Cat2 extends NormalAnimal with FourLegged with Furry
(new Cat).f(); (new Cat2).f()
```

Exercise: Fill in the boxes

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
}
import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
trait Filtering extends IntQueue {
  abstract override def put(x:Int) { if (x > 0) super.put(x) }
}
trait Doubling extends IntQueue {
  
}
trait Increasing extends IntQueue {
  
}
class IncFilteringArrayQueue extends ArrayQueue
  with Filtering with Increasing
class IncFilterDoubleArrayQueue extends ArrayQueue
  
```

Exercise: Fill in the boxes

```
trait IntQueue {
  def get(): Int
  def put(x: Int): Unit
}
import scala.collection.mutable.ArrayBuffer
class ArrayQueue extends IntQueue {
  private val buf = new ArrayBuffer[int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
trait Filtering extends IntQueue {
  abstract override def put(x:Int) { if (x > 0) super.put(x) }
}
trait Doubling extends IntQueue {
  abstract override def put(x:Int) { super.put(2*x) }
}
trait Increasing extends IntQueue {
  abstract override def put(x:Int) { super.put(1+x) }
}
class IncFilteringArrayQueue extends ArrayQueue
  with Filtering with Increasing
class IncFilterDoubleArrayQueue extends ArrayQueue
  with Doubling with Filtering with Increasing
```

Trait vs class

- When do you want to use traits?
- When do you want to use classes or abstract classes?

Summary

- Traits can include method and field definitions.
- Multiple traits can be inherited simultaneously. Linearization determines their inheritance hierarchy.
- Traits support stackable modification via super call.
- Read Chap 12.