PART 3

Type Classes for Interfaces

Problems with OOP

Subtype Polymorphism

```scala
trait Ord {
  // this cmp that < 0  iff this < that
  // this cmp that > 0  iff this > that
  // this cmp that == 0 iff this == that
  def cmp(that: Ord): Int

  def ===(that: Ord): Boolean = (this.cmp(that)) == 0
  def <  (that: Ord): Boolean = (this cmp that) < 0
  def >  (that: Ord): Boolean = (this cmp that) > 0
  def <= (that: Ord): Boolean = (this cmp that) <= 0
  def >= (that: Ord): Boolean = (this cmp that) >= 0
}
def max3(a: Ord, b: Ord, c: Ord) : Ord =
  if (a <= b) { if (b <= c) c else b }
  else        { if (a <= c) c else a }
```

* Problem: hard (almost impossible) to implement Ord (e.g., using Int)

Interface over Parameter Types

```
trait Ord[A] {
  def cmp(that: A): Int

  def ===(that: A): Boolean = (this.cmp(that)) == 0
  def <  (that: A): Boolean = (this cmp that) < 0
  def >  (that: A): Boolean = (this cmp that) > 0
  def <= (that: A): Boolean = (this cmp that) <= 0
  def >= (that: A): Boolean = (this cmp that) >= 0
}
def max3[A <: Ord[A]](a: A, b: A, c: A) : A =
  if (a <= b) {if (b <= c) c else b }
  else        {if (a <= c) c else a }

class OInt(val value : Int) extends Ord[OInt] {
  def cmp(that: OInt) = value - that.value
}
max3(new OInt(3), new OInt(2), new OInt(10)).value
```

Further example: Ordered Bag

```
class Bag[U <: Ord[U]] protected (val toList: List[U]) {
  def this() = this(Nil)
  def add(x: U) : Bag[U] = {
    def go(elmts: List[U]): List[U] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (x < e) => x :: elmts
        case e :: _ if (x === e) => elmts
        case e :: rest => e :: go(rest)
      }
    new Bag(go(toList))
  }
}
val emp = new Bag[OInt]()
val b = emp.add(new OInt(3)).add(new OInt(2)).
            add(new OInt(10)).add(new OInt(2))
b.toList.map((x)=>x.value)
```

Problems with OOP

1. Needs "subtyping" like "OInt <: Ord[OInt]", which is quite complex as we have seen (and moreover, involves more complex concepts like variance).

2. Needs a wrapper class like "OInt" in order to add a new interface to an existing type like "Int".

3. Interface only contains only "elimination" functions, not "introduction" functions.

4. No canonical operator

5. …

Type Classes

Separating Functions from Data

```
trait Ord[A] {
  def cmp(self: A)(a: A): Int

  def ===(self: A)(a: A) = cmp(self)(a) == 0
  def <  (self: A)(a: A) = cmp(self)(a) < 0
  def >  (self: A)(a: A) = cmp(self)(a) > 0
  def <= (self: A)(a: A) = cmp(self)(a) <= 0
  def >= (self: A)(a: A) = cmp(self)(a) >= 0
}

def max3[A](a: A, b: A, c: A)(implicit ORD: Ord[A]) : A =
  if (ORD.<=(a)(b)) {if (ORD.<=(b)(c)) c else b }
  else              {if (ORD.<=(a)(c)) c else a }

// behaves like Int <: Ord in OOP
implicit val intOrd : Ord[Int] = new {
  def cmp(self: Int)(a: Int) = self - a }
max3(3,2,10) // 10
```

Implicit

- Implicit
  - An argument is given “implicitly”

```
def foo(s: String)(implicit t: String) = s + t

implicit val exclamation : String = "!!!!!!"

foo("Hi")
foo("Hi")("???") // can give it explicitly
```

Syntax for type class: syntactic sugar

```
trait Ord[A]:
  extension (self: A)
    def cmp(a: A): Int
    def ===(a: A) = self.cmp(a) == 0
    def <  (a: A) = self.cmp(a) < 0
    def >  (a: A) = self.cmp(a) > 0
    def <= (a: A) = self.cmp(a) <= 0
    def >= (a: A) = self.cmp(a) >= 0

def max3[A: Ord](a: A, b: A, c: A) : A =
  if (a <= b) { if (b <= c) c else b }
  else        { if (a <= c) c else a }

given intOrd : Ord[Int] with
  extension (self: Int)
    def cmp(a: Int) = self - a


max3(3,2,10) // 10
```

Syntax for type class: syntactic sugar

```
trait Ord[A]:

  def cmp(self: A)(a: A): Int
  def ===(self: A)(a: A) = cmp(self)(a) == 0
  def < (self: A)(a: A) = cmp(self)(a) < 0
  def > (self: A)(a: A) = cmp(self)(a) > 0
  def <= (self: A)(a: A) = cmp(self)(a) <= 0
  def >= (self: A)(a: A) = cmp(self)(a) >= 0

def max3[A](a: A, b: A, c: A)(implicit ORD: Ord[A]) : A =
  if (ORD.<=(a)(b)) { if (ORD.<=(b)(c)) c else b }
  else                  { if (ORD.<=(a)(c)) c else a }

implicit def intOrd : Ord[Int] = new {
  def cmp(self:Int)(a: Int) = self - a
}


max3(3,2,10) // 10
```

Bag Example using type class

```
class Bag[A: Ord] protected (val toList: List[A])
{ def this() = this(Nil)
  def add(x: A) : Bag[A] = {
    def loop(elmts: List[A]) : List[A] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (x < e) => x :: elmts
        case e :: _ if (x === e) => elmts
        case e :: rest => e :: loop(rest)
      }
    new Bag(loop(toList))
  }
}

(new Bag[Int]()).add(3).add(2).add(3).add(10).toList
```

Bag Example using type class

```
class Bag[A] protected (val toList: List[A])(implicit ORD: Ord[A])
{ def this()(implicit ORD: Ord[A]) = this(Nil)
  def add(x: A) : Bag[A] = {
    def loop(elmts: List[A]) : List[A] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (ORD.<(x)(e)) => x :: elmts
        case e :: _ if (ORD.===(x)(e)) => elmts
        case e :: rest => e :: loop(rest)
      }
    new Bag(loop(toList))
  }
}

(new Bag[Int]()).add(3).add(2).add(3).add(10).toList
```

Bootstrapping Implicits

```
// lexicographic order
given tupOrd[A, B](using Ord[A], Ord[B]): Ord[(A,B)] with
  extension (self: (A,B))
    def cmp(a: (A, B)) : Int = {
      val c1 = self._1.cmp(a._1)
      if (c1 != 0) c1
      else { self._2.cmp(a._2) }
    }

val b = new Bag[(Int,(Int,Int))]
b.add((3,(3,4))).add((3,(2,7))).add((4,(0,0))).toList
```

Bootstrapping Implicits

```
// lexicographic order
implicit def tupOrd[A, B](implicit ORDA: Ord[A], ORDB: Ord[B]): Ord[(A,B)] =
new {
  def cmp(self:(A,B))(a: (A, B)) : Int = {
    val c1 = ORDA.cmp(self._1)(a._1)
    if (c1 != 0) c1
    else { ORDB.cmp(self._2)(a._2) }
  }
}

val b = new Bag[(Int,(Int,Int))]
b.add((3,(3,4))).add((3,(2,7))).add((4,(0,0))).toList
```

With Different Orders

```
def intOrdRev : Ord[Int] = new {
  extension (self: Int)
    def cmp(a: Int) = a - self
}

(new Bag[Int]()).add(3).add(2).add(10).toList
(new Bag[Int]()(intOrdRev)).add(3).add(2).add(10).toList
```

With Different Orders

```
def intOrdRev : Ord[Int] = new {
  def cmp(self: Int)(a: Int) = a - self
}

(new Bag[Int]()).add(3).add(2).add(10).toList
(new Bag[Int]()(intOrdRev)).add(3).add(2).add(10).toList
```

Type Classes: Abstraction

Interfaces I: elimination

```
trait Iter[I,A]:
  extension (self: I)
    def getValue: Option[A]
    def getNext: I

trait Iterable[I,A]:
  type Itr
  given ITR: Iter[Itr,A]
  extension (self: I)
    def iter: Itr

// behaves like Iter[A] <: Iterable[A] in OOP
given iter2iterable[I,A](using _ITR: Iter[I,A]): Iterable[I,A] with
  type Itr = I
  def ITR = _ITR
  extension (self: I)
    def iter = self
```

Interfaces I: elimination

```
trait Iter[I,A]:
  def getValue(self: I): Option[A]
  def getNext(self: I): I

trait Iterable[I,A]:
  type Itr
  implicit def ITR: Iter[Itr,A]
  def iter(self: I): Itr

// behaves like Iter[A] <: Iterable[A] in OOP
implicit def iter2iterable[I,A](implicit _ITR: Iter[I,A]): Iterable[I,A] = new {
  type Itr = I
  def ITR = _ITR
  def iter(self: I) = self
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I](xs: I)(implicit ITRA:Iterable[I,Int]) = {
  def loop(i: ITRA.Itr): Int =
    i.getValue match {
      case None => 0
      case Some(n) => n + loop(i.getNext)
    }
  loop(xs.iter)
}

def printElements[I,A](xs: I)(implicit ITRA: Iterable[I,A]) = {
  def loop(i: ITRA.Itr): Unit =
    i.getValue match {
      case None =>
      case Some(a) => {println(a); loop(i.getNext)}
    }
  loop(xs.iter)
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I](xs: I)(implicit ITRA:Iterable[I,Int]) = {
 def loop(i: ITRA.Itr): Int =
  ITRA.ITR.getValue(i) match {
   case None => 0
   case Some(n) => n + loop(ITRA.ITR.getNext(i))
  }
 loop(ITRA.iter(xs))
}

def printElements[I,A](xs: I)(implicit ITRA: Iterable[I,A]) = {
 def loop(i: ITRA.Itr): Unit =
  ITRA.ITR.getValue(i) match {
   case None =>
   case Some(a) => {println(a); loop(ITRA.ITR.getNext(i))}
  }
 loop(ITRA.iter(xs))
}
```

Interfaces II: introduction + elimination

```
trait Listlike[L,A]:
  extension(u:Unit)
    def unary_! : L
  extension(elem:A)
    def ::(l: =>L): L
  extension(l: L)
    def head: Option[A]
    def tail: L
    def ++(l2: L): L

trait Treelike[T,A]:
  extension(u:Unit)
    def unary_! : T
  extension(a:A)
    def has(lt: T, rt: T): T
  extension(t: T)
    def root : Option[A]
    def left : T
    def right : T
```

Interfaces II: introduction + elimination

```
trait Listlike[L,A]:
  def ! : L
  def ::(elem:A)(l: =>L): L
  def head(l: L): Option[A]
  def tail(l: L): L
  def ++(l: L)(l2: L): L

trait Treelike[T,A]:
  def ! : T
  def has(a:A)(lt: T, rt: T): T
  def root(t: T) : Option[A]
  def left(t: T) : T
  def right(t: T) : T
```

Programs for Testing: use All

```
def testList[L](implicit LL: Listlike[L,Int], ITRA: Iterable[L,Int]) = {
  val l = (3 :: !()) ++ (1 :: 2 :: !())
  println(sumElements(l))
  printElements(l)
}

def testTree[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  val t = 3.has(4.has(!(), !()), 2.has(!(),!()))
  println(sumElements(t))
  printElements(t)
}
```

Programs for Testing: use All

```
def testList[L](implicit LL: Listlike[L,Int], ITRA: Iterable[L,Int]) = {
  val l = LL.++(LL.::(3)(LL.!))(LL.::(1)(LL.::(2)(LL.!)))
  println(sumElements(l))
  printElements(l)
}

def testTree[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  val t = TL.has(3)(TL.has(4)(TL.!, TL.!), TL.has(2)(TL.!,TL.!))
  println(sumElements(t))
  printElements(t)
}
```

Implement Iter and Listlike for List

```
// behaves like Listlike[A] <: Iter[A] in OOP
given listIter[L,A](using LL: Listlike[L,A]): Iter[L,A] with
  extension (l: L)
    def getValue = l.head
    def getNext = l.tail

// behaves like List[A] <: Listlike[A] in OOP
given listListlike[A]: Listlike[List[A],A] with
  extension (u: Unit)
    def unary_! = Nil
  extension (a: A)
    def ::(l: =>List[A]) = a::l
  extension (l: List[A])
    def head = l.headOption
    def tail = l.tail
    def ++(l2: List[A]) = l ::: l2
```

Implement Iter and Listlike for List

```
// behaves like Listlike[A] <: Iter[A] in OOP
implicit def listIter[L,A](implicit LL: Listlike[L,A]): Iter[L,A] = new {
  def getValue(l: L) = LL.head(l)
  def getNext(l: L) = LL.tail(l)
}

// behaves like List[A] <: Listlike[A] in OOP
implicit def listListlike[A]: Listlike[List[A],A] = new {
  def ! = Nil
  def ::(a: A)(l: => List[A]) = a :: l
  def head(l: List[A]) = l.headOption
  def tail(l: List[A]) = l.tail
  def ++(l: List[A])(l2: List[A]) = l ::: l2
}
```

Implement Iterable for MyTree using Listlike,Iter

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
given treeIterable[L,A](using LL: Listlike[L,A], _ITR: Iter[L,A])
  : Iterable[MyTree[A], A] with
  type Itr = L
  def ITR = _ITR
  extension (t: MyTree[A])
    def iter: L = t match {
      case Leaf => !()
      case Node(v, lt, rt) => v :: (lt.iter ++ rt.iter)
    }
```

Implement Iterable for MyTree using Listlike,Iter

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
implicit def treeIterable[L,A](implicit LL: Listlike[L,A], _ITR: Iter[L,A])
  : Iterable[MyTree[A], A] = new {
  type Itr = L
  def ITR = _ITR
  def iter(t: MyTree[A]): L = t match {
    case Leaf => LL.!
    case Node(v, lt, rt) => LL.::(v)(LL.++(iter(lt))(iter(rt)))
  }
}
```

Implement Treelike for MyTree

```
// behaves like MyTree[A] <: Treelike[A] in OOP
given mytreeTreelike[A] : Treelike[MyTree[A],A] with
  extension (u: Unit)
    def unary_! = Leaf
  extension (a: A)
    def has(l: MyTree[A], r: MyTree[A]) = Node(a,l,r)
  extension (t: MyTree[A])
    def root = t match {
      case Leaf => None
      case Node(v,_,_) => Some(v)
    }
    def left = t match {
      case Leaf => t
      case Node(_,lt,_) => lt
    }
    def right = t match {
      case Leaf => t
      case Node(_,_,rt) => rt }
```

Implement Treelike for MyTree

```
// behaves like MyTree[A] <: Treelike[A] in OOP
implicit def mytreeTreelike[A] : Treelike[MyTree[A],A] = new {
  def ! = Leaf
  def has(a: A)(l: MyTree[A], r: MyTree[A]) = Node(a, l, r)
  def root(t: MyTree[A]) = t match {
    case Leaf => None
    case Node(v, _, _) => Some(v)
  }
  def left(t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, lt, _) => lt
  }
  def right(t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, _, rt) => rt
  }
}
```

Linking Modules

```
testList[List[Int]]

testTree[MyTree[Int]]
```

Test for Lazy List

```
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block    // call-by-name
  val t1 = System.nanoTime()
  println("Elapsed time: " + ((t1 - t0)/1000000) + "ms"); result
}
def sumN[I](n: Int, t: I)(implicit ITRA: Iterable[I,Int]): Int = {
  def go(res: Int, n: Int, itr: ITRA.Itr): Int =
    if (n <= 0) res
    else itr.getValue match {
      case None => res
      case Some(v) => go(v + res, n - 1, itr.getNext)
    }
  go(0, n, t.iter)
}
```

Test for Lazy List

```
def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block    // call-by-name
  val t1 = System.nanoTime()
  println("Elapsed time: " + ((t1 - t0)/1000000) + "ms"); result
}
def sumN[I](n: Int, t: I)(implicit ITRA: Iterable[I,Int]): Int = {
  def go(res: Int, n: Int, itr: ITRA.Itr): Int =
    if (n <= 0) res
    else ITRA.ITR.getValue(itr) match {
      case None => res
      case Some(v) => go(v + res, n - 1, ITRA.ITR.getNext(itr))
    }
  go(0, n, ITRA.iter(t))
}
```

Test for Lazy List

```
def testTree2[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  def generateTree(n: Int): T = {
    def gen(lo: Int, hi: Int): T = {
      if (lo > hi) !()
      else {
        val mid = (lo + hi) / 2
        mid.has(gen(lo, mid - 1), gen(mid + 1, hi))
      }
    }
    gen(1, n)
  }

  // Problem: takes a few seconds to get a single value
  { val t = generateTree(200000)
    time (sumN(2, t)) }
}
```

Test for Lazy List

```
def testTree2[T](implicit TL: Treelike[T,Int], ITRA: Iterable[T,Int]) = {
  def generateTree(n: Int): T = {
    def gen(lo: Int, hi: Int): T = {
      if (lo > hi) TL.!
      else {
        val mid = (lo + hi) / 2
        TL.has(mid)(gen(lo, mid - 1), gen(mid + 1, hi))
      }
    }
    gen(1, n)
  }

  // Problem: takes a few seconds to get a single value
  { val t = generateTree(200000)
    time (sumN(2, t)) }
}
```

Lazy List

```
sealed abstract class LazyList[+A] {
  def matches[R](caseNil: =>R, caseCons: (A,LazyList[A])=>R) : R
}
case object LNil extends LazyList[Nothing] {
  def matches[R](caseNil: =>R, _u: (Nothing,LazyList[Nothing])=>R) =
    caseNil
}
class LCons[A](hd: A, _tl: =>LazyList[A]) extends LazyList[A] {
  lazy val tl = _tl
  def matches[R](_u: =>R, caseCons: (A, LazyList[A])=>R) =
    caseCons(hd, tl)
}
object LazyList {
  extension [A](l: LazyList[A])
    def append(l2: LazyList[A]) : LazyList[A] =
      l.matches(l2, (hd,tl) => LCons(hd, tl.append(l2)))
}
import LazyList.*
```

Lazy List

```
sealed abstract class LazyList[+A] {
  def matches[R](caseNil: =>R, caseCons: (A,LazyList[A])=>R) : R
}
case object LNil extends LazyList[Nothing] {
  def matches[R](caseNil: =>R, _u: (Nothing,LazyList[Nothing])=>R) =
    caseNil
}
class LCons[A](hd: A, _tl: =>LazyList[A]) extends LazyList[A] {
  lazy val tl = _tl
  def matches[R](_u: =>R, caseCons: (A, LazyList[A])=>R) =
    caseCons(hd, tl)
}
object LazyList {
  def append[A](l: LazyList[A])(l2: LazyList[A]) : LazyList[A] =
    l.matches(l2, (hd,tl) => LCons(hd, append(tl)(l2)))
}
import LazyList.*
```

Lazy List

```
given lazylistListlike[A]: Listlike[LazyList[A],A] with
  extension (u: Unit)
    def unary_! = LNil
  extension (a: A)
    def ::(l: =>LazyList[A]) = LCons(a,l)
  extension (l: LazyList[A])
    def head = l.matches(None, (hd,tl) => Some(hd))
    def tail = l.matches(LNil, (hd,tl)=>tl)
    def ++(l2: LazyList[A]) = l.append(l2)


testList[LazyList[Int]]
testTree[MyTree[Int]]
testTree2[MyTree[Int]]
```

Lazy List

```
implicit def lazylistListlike[A]: Listlike[LazyList[A],A] = new {
  def ! = LNil
  def ::(a: A)(l: => LazyList[A]) = LCons(a, l)
  def head(l: LazyList[A]) = l.matches(None, (hd, tl) => Some(hd))
  def tail(l: LazyList[A]) = l.matches(LNil, (hd, tl) => tl)
  def ++(l: LazyList[A])(l2: LazyList[A]) = LazyList.append(l)(l2)
}

testList[LazyList[Int]]
testTree[MyTree[Int]]
testTree2[MyTree[Int]]
```

Type class: Code Reuse

IntStack Spec

```
trait Stack[S,A]:
  extension (u: Unit)
    def empty : S
  extension (s: S)
    def get: (A,S)
    def put(a: A): S

def testStack[S](implicit STK: Stack[S,Int]) = {
  val s = ().empty.put(3).put(-2).put(4)
  val (v1,s1) = s.get
  val (v2,s2) = s1.get
  (v1,v2)
}
```

IntStack Spec

```
trait Stack[S,A]:
  def empty : S
  def get(s: S): (A,S)
  def put(s: S)(a: A): S

def testStack[S](implicit STK: Stack[S,Int]) = {
  val s = STK.put(STK.put(STK.put(STK.empty)(3))(-2))(4)
  val (v1,s1) = STK.get(s)
  val (v2,s2) = STK.get(s1)
  (v1,v2)
}
```

Implementation using List

```
given BasicStack[A] : Stack[List[A],A] with
  extension (u: Unit)
    def empty = List()
  extension (s: List[A])
    def get = (s.head, s.tail)
    def put(a: A) = a :: s
```

Implementation using List

```
implicit def BasicStack[A] : Stack[List[A],A] = new {
  def empty = List()
  def get(s: List[A]) = (s.head, s.tail)
  def put(s: List[A])(a: A) = a :: s
}
```

Modifying Traits

```
def StackOverridePut[S,A](newPut: (S,A)=>S)(implicit STK: Stack[S,A])
: Stack[S,A] = new {
  extension (u: Unit)
    def empty = STK.empty(u)
  extension (s: S)
    def get = STK.get(s)
    def put(a: A) = newPut(s,a)
}

def Doubling[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => s.put(2 * a))

def Incrementing[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => s.put(a + 1))

def Filtering[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => if (a >= 0) s.put(a) else s)
```

Modifying Traits

```
def StackOverridePut[S,A](newPut: (S,A)=>S)(implicit STK: Stack[S,A])
: Stack[S,A] = new {
  def empty = STK.empty
  def get(s: S) = STK.get(s)
  def put(s: S)(a: A) = newPut(s,a)
}

def Doubling[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => STK.put(s)(2 * a))

def Incrementing[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => STK.put(s)(a + 1))

def Filtering[S](implicit STK: Stack[S,Int]) : Stack[S,Int] =
  StackOverridePut((s,a) => if (a >= 0) STK.put(s)(a) else s)
```

Linking

```
// testStack(BasicStack)
testStack

// testStack(Filtering(Incrementing (Doubling(BasicStack))))
testStack(Filtering (Incrementing (Doubling)))

// testStack(Filtering(Incrementing(Incrementing(Doubling(BasicStack)))))
testStack(Filtering (Incrementing (Incrementing (Doubling))))
```

Implementation: Sorted Stack

```
def SortedStack : Stack[List[Int],Int] = new {
  extension (u: Unit)
    def empty = List()
  extension (s: List[Int])
    def get = (s.head, s.tail)
    def put(a: Int) : List[Int] = {
      def loop(l: List[Int]) : List[Int] = l match {
        case Nil => a :: Nil
        case hd :: tl => if (a <= hd) a :: l else hd :: loop(tl)
      }
      loop(s)
    }
  }

testStack(Filtering(Incrementing(Doubling(SortedStack))))
```

Implementation: Sorted Stack

```
def SortedStack : Stack[List[Int],Int] = new {
  def empty = List()
  def get(s: List[Int]) = (s.head, s.tail)
  def put(s: List[Int])(a: Int) : List[Int] = {
    def loop(l: List[Int]) : List[Int] = l match {
      case Nil => a :: Nil
      case hd :: tl => if (a <= hd) a :: l else hd :: loop(tl)
    }
    loop(s)
  }
}

testStack(Filtering(Incrementing(Doubling(SortedStack))))
```

Higher Type Classes

Interfaces I

```
// eg. Iter[List]
trait Iter[I[_]]:
  extension [A](i: I[A])
    def getValue: Option[A]
    def getNext: I[A]

// eg. Iterable[MyTree]
trait Iterable[I[_]]:
  type Itr[_]
  given ITR: Iter[Itr]
  extension [A](i: I[A])
    def iter: Itr[A]

given iter2iterable[I[_]](using _ITR: Iter[I]): Iterable[I] with
  type Itr[A] = I[A]
  def ITR = _ITR
  extension [A](i:I[A])
    def iter = i
```

```
// trait Iter[I,A]:
//   extension (i: I)
//     def getValue: Option[A]
//     def getNext: I

// trait Iterable[I,A]:
//   type Itr
//   given ItrI: Iter[Itr,A]
//   extension (i: I)
//     def iter: Itr
```

Interfaces I

```
// eg. Iter[List]
trait Iter[I[_]]:
  def getValue[A](i: I[A]): Option[A]
  def getNext[A](i: I[A]): I[A]

// eg. Iterable[MyTree]
trait Iterable[I[_]]:
  type Itr[_]
  implicit def ITR: Iter[Itr]
  def iter[A](i: I[A]): Itr[A]

implicit def iter2iterable[I[_]](using _ITR: Iter[I]): Iterable[I] = new {
  type Itr[A] = I[A]
  def ITR = _ITR
  def iter[A](i:I[A]) = i
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I[_]](xs: I[Int])(implicit ITRA:Iterable[I]) = {
  def loop(i: ITRA.Itr[Int]): Int =
    i.getValue match {
      case None => 0
      case Some(n) => n + loop(i.getNext)
    }
  loop(xs.iter)
}

def printElements[I[_],A](xs: I[A])(implicit ITRA: Iterable[I]) = {
  def loop(i: ITRA.Itr[A]): Unit =
    i.getValue match {
      case None =>
      case Some(a) => {println(a); loop(i.getNext)}
    }
  loop(xs.iter)
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I[_]](xs: I[Int])(implicit ITRA:Iterable[I]) = {
  def loop(i: ITRA.Itr[Int]): Int =
    ITRA.ITR.getValue(i) match {
      case None => 0
      case Some(n) => n + loop(ITRA.ITR.getNext(i))
    }
  loop(ITRA.iter(xs))
}

def printElements[I[_],A](xs: I[A])(implicit ITRA: Iterable[I]) = {
  def loop(i: ITRA.Itr[A]): Unit =
    ITRA.ITR.getValue(i) match {
      case None =>
      case Some(a) => {println(a); loop(ITRA.ITR.getNext(i))}
    }
  loop(ITRA.iter(xs))
}
```

Interfaces II

```
trait Listlike[L[_]]:
  extension[A](u:Unit)
    def unary_! : L[A]
  extension[A](elem:A)
    def ::(l: =>L[A]): L[A]
  extension[A](l: L[A])
    def head: Option[A]
    def tail: L[A]
    def ++(l2: L[A]): L[A]

trait Treelike[T[_]]:
  extension[A](u:Unit)
    def unary_! : T[A]
  extension[A](a:A)
    def has(lt: T[A], rt: T[A]): T[A]
  extension[A](t: T[A])
    def root : Option[A]
    def left : T[A]
    def right : T[A]
```

Interfaces II

```
trait Listlike[L[_]]:
  def ![A] : L[A]
  def ::[A](elem:A)(l: =>L[A]): L[A]
  def head[A](l: L[A]): Option[A]
  def tail[A](l: L[A]): L[A]
  def ++[A](l: L[A])(l2: L[A]): L[A]

trait Treelike[T[_]]:
  def ![A] : T[A]
  def has[A](a:A)(lt: T[A], rt: T[A]): T[A]
  def root[A](t: T[A]) : Option[A]
  def left[A](t: T[A]) : T[A]
  def right[A](t: T[A]) : T[A]
```

Programs for Testing: use All

```
def testList[L[_]](implicit LL: Listlike[L], ITRA: Iterable[L]) = {
  val l = (3 :: !()) ++ (1 :: 2 :: !())
  println(sumElements(l))
  printElements(l)
}

def testTree[T[_]](implicit TL: Treelike[T], ITRA: Iterable[T]) = {
  val t = 3.has(4.has(!(), !()), 2.has(!(),!()))
  println(sumElements(t))
  printElements(t)
}
```

Programs for Testing: use All

```
def testList[L[_]](implicit LL: Listlike[L], ITRA: Iterable[L]) = {
  val l = LL.++(LL.::(3)(LL.!))(LL.::(1)(LL.::(2)(LL.!)))
  println(sumElements(l))
  printElements(l)
}

def testTree[T[_]](implicit TL: Treelike[T], ITRA: Iterable[T]) = {
  val t = TL.has(3)(TL.has(4)(TL.!, TL.!), TL.has(2)(TL.!,TL.!))
  println(sumElements(t))
  printElements(t)
}
```

List: provide Iter, ListIF

```
// behaves like List[A] <: Iter[A] in OOP
given listIter: Iter[List] with
  extension [A](l: List[A])
    def getValue = l.headOption
    def getNext = l.tail

// behaves like List[A] <: Listlike[A] in OOP
given listListlike: Listlike[List] with
  extension [A](u: Unit)
    def unary_! = Nil
  extension [A](a: A)
    def ::(l: =>List[A]) = a::l
  extension [A](l: List[A])
    def head = l.headOption
    def tail = l.tail
    def ++(l2: List[A]) = l ::: l2
```

List: provide Iter, ListIF

```
// behaves like List[A] <: Iter[A] in OOP
implicit def listIter: Iter[List] = new {
  def getValue[A] (l: List[A]) = l.headOption
  def getNext[A] (l: List[A]) = l.tail
}
// behaves like List[A] <: Listlike[A] in OOP
implicit def listListlike: Listlike[List] = new {
  def ![A] = Nil
  def ::[A](a: A)(l: => List[A]) = a :: l
  def head[A](l: List[A]) = l.headOption
  def tail[A](l: List[A]) = l.tail
  def ++[A](l: List[A])(l2: List[A]) = l ::: l2
}
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
given treeIterable[L[_]](using LL: Listlike[L], _ITR: Iter[L]): Iterable[MyTree]
with
  type Itr[A] = L[A]
  def ITR = _ITR
  extension [A](t: MyTree[A])
    def iter: L[A] = t match {
      case Leaf => !()
      case Node(v, lt, rt) => v :: (lt.iter ++ rt.iter)
    }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
enum MyTree[+A]:
  case Leaf
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

// behaves like MyTree[A] <: Iterable[A], but clumsy in OOP
implicit def treeIterable[L[_]](using LL: Listlike[L], _ITR: Iter[L]):
Iterable[MyTree] = new {
  type Itr[A] = L[A]
  def ITR = _ITR
  def iter[A] (t: MyTree[A]): L[A] = t match {
    case Leaf => LL.!
    case Node(v, lt, rt) => LL.::(v)(LL.++(iter(lt))(iter(rt)))
  }
}
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
// behaves like MyTree[A] <: Treelike[A] in OOP
given mytreeTreelike: Treelike[MyTree] with
  extension [A](u: Unit)
    def unary_! = Leaf
  extension [A](a: A)
    def has(l: MyTree[A], r: MyTree[A]) = Node(a,l,r)
  extension [A](t: MyTree[A])
    def root = t match {
      case Leaf => None
      case Node(v,_,_) => Some(v)
    }
    def left = t match {
      case Leaf => t
      case Node(_,lt,_) => lt
    }
    def right = t match {
      case Leaf => t
      case Node(_,_,rt) => rt }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```scala
// behaves like MyTree[A] <: Treelike[A] in OOP
implicit def mytreeTreelike: Treelike[MyTree] = new {
  def ![A] = Leaf
  def has[A] (a: A)(l: MyTree[A], r: MyTree[A]) = Node(a, l, r)
  def root[A](t: MyTree[A]) = t match {
    case Leaf => None
    case Node(v, _, _) => Some(v)
  }
  def left[A](t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, lt, _) => lt
  }
  def right[A](t: MyTree[A]) = t match {
    case Leaf => t
    case Node(_, _, rt) => rt
  }
}
```

Linking Modules

```
testList[List]

testTree[MyTree]
```

List with Map

```
trait Maplike[L[_]]:
  extension[A](l: L[A])
    def map[B](f: A => B): L[B]

def testMapList[L[_]](implicit LL: Listlike[L], ML: Maplike[L], ITR: Iter[L]) = {
  val l1 = 3.3 :: 2.2 :: 1.5 :: !()
  val l2 = l1.map((n:Double)=>n.toInt)
  val l3 = l2.map((n:Int)=>n.toString)
  printElements(l3)
}
```

List with Map

```
trait Maplike[L[_]]:
  def map[A,B](l: L[A])(f: A => B): L[B]

def testMapList[L[_]] (implicit LL: Listlike[L], ML: Maplike[L], ITR: Iter[L]) = {
  val l1 = LL.::(3.3)(LL.::(2.2)(LL.::(1.5)(LL.!)))
  val l2 = ML.map(l1)((n:Double)=>n.toInt)
  val l3 = ML.map(l2)((n:Int)=>n.toString)
  printElements(l3)
}
```

List with Map

```
given listMaplike: Maplike[List] with
extension [A](l: List[A])
   def map[B](f: A => B) = l.map(f)

testMapList[List]
```

List with Map

```
implicit def listMaplike: Maplike[List] = new {
  def map[A,B](l: List[A])(f: A => B) = l.map(f)
}

testMapList[List]
```

Turning Type Classes into OO Classes

Interfaces

```
trait DataProcessor[D]:
  extension (d: D)
    def input(s: String) : D
    def output : String

trait DPFactory:
  extension (u: Unit)
    def getTypes: List[String]
    def makeDP(dptype: String) : ???

def run(implicit factory: DPFactory) : Unit
```

How to return data with associated functions like OOP?

Turning Type Classes into OO Classes

```
import scala.language.implicitConversions
type curry1[F[_,_],A1] = ([X] =>> F[X,A1])
type curry2[F[_,_,_],A1,A2] = ([X] =>> F[X,A1,A2])
type curry3[F[_,_,_,_],A1,A2,A3] = ([X] =>> F[X,A1,A2,A3])

trait dyn[S[_]]:
  type Data
  val * : Data
  given DI: S[Data]

object dyn {
  implicit // needed for implicit conversion of D into dyn[S]
  def apply[S[_],D](d: D)(implicit i: S[D]): dyn[S] = new {
      type Data = D
      val * = d
      val DI = i
    }
}
```

Turning Type Classes into OO Classes

```
import scala.language.implicitConversions
type curry1[F[_,_],A1] = ([X] =>> F[X,A1])
type curry2[F[_,_,_],A1,A2] = ([X] =>> F[X,A1,A2])
type curry3[F[_,_,_,_],A1,A2,A3] = ([X] =>> F[X,A1,A2,A3])

trait dyn[S[_]]:
  type Data
  val * : Data
  implicit def DI: S[Data]

object dyn {
  implicit // needed for implicit conversion of D into dyn[S]
  def apply[S[_],D](d: D)(implicit i: S[D]): dyn[S] = new {
    type Data = D
    val * = d
    val DI = i
  }
}
```

Interfaces

```
trait DataProcessor[D]:
  extension (d: D)
    def input(s: String): D
    def output: String

trait DPFactory:
  extension (u: Unit)
    def getTypes: List[String]
    def makeDP(dptype: String): dyn[DataProcessor]
```

Interfaces

```
trait DataProcessor[D]:
  def input(d: D)(s: String): D
  def output(d: D): String

trait DPFactory:
  def getTypes: List[String]
  def makeDP(dptype: String): dyn[DataProcessor]
```

Test

```
def test(implicit DF: DPFactory) = {
  def go(types: List[String]) : Unit =
    types match {
      case Nil => ()
      case ty :: rest => {
        val dp = ().makeDP(ty)
        println(dp.*.input("10").input("20").output)
        go(rest)
      }
    }
  val types = ().getTypes
  println(types)
  go(types)
}
```

Test

```
def test(implicit DF: DPFactory) = {
  def go(types: List[String]) : Unit =
    types match {
      case Nil => ()
      case ty :: rest => {
        val dp : dyn = DF.makeDP(ty)
        println(dp.DI.output(dp.DI.input(dp.DI.input(dp.*)("10"))("20")))
        go(rest)
      }
    }
  val types = DF.getTypes
  println(types)
  go(types)
}
```

Data Processor

```
given dpfactory: DPFactory with
  extension (u: Unit)
    def getTypes = List("sum", "mult")
    def makeDP(dptype: String) = {
      if (dptype == "sum")
        makeProc(0, (x, y) => x + y)
      else
        makeProc(1, (x, y) => x * y)
    }

    def makeProc(init: Int, op: (Int, Int) => Int): dyn[DataProcessor] = {
      given dp: DataProcessor[Int] with
        extension (d: Int)
          def input(s: String) = op(d, s.toInt)
          def output = d.toString()
      init      // dyn(init) // dyn.apply[Int,DataProcessor](init)(dp)
    }
```

Data Processor

```
implicit val dpfactory: DPFactory = new {
  def getTypes = List("sum", "mult")
  def makeDP(dptype: String) = {
    if (dptype == "sum")
      makeProc(0, (x, y) => x + y)
    else
      makeProc(1, (x, y) => x * y)
  }

  def makeProc(init: Int, op: (Int, Int) => Int): dyn[DataProcessor] = {
    implicit def dp: DataProcessor[Int] = new {
      def input(d: Int)(s: String) = op(d, s.toInt)
      def output(d: Int) = d.toString()
    }
    init // dyn(init)(dp) // dyn.apply[Int,DataProcessor](init)(dp)
  }
}
```

Linking

test

Heterogeneous List of Iter

```
trait Iter[I,A]:
  extension (i: I)
    def getValue: Option[A]
    def getNext: I

def sumElements[I](xs: I)(implicit ITR:Iter[I,Int]) : Int = {
  xs.getValue match {
    case None => 0
    case Some(n) => n + sumElements(xs.getNext)
  }
}

def sumElementsList(xs: List[dyn[curry1[Iter,Int]]]) : Int =
  xs match {
    case Nil => 0
    case hd :: tl => sumElements(hd.*) + sumElementsList(tl)
  }
```

Heterogeneous List of Iter

```
trait Iter[I,A]:
  def getValue(i: I): Option[A]
  def getNext(i: I): I

def sumElements[I](xs: I)(implicit ITR:Iter[I,Int]) : Int = {
  ITR.getValue(xs) match {
    case None => 0
    case Some(n) => n + sumElements(ITR.getNext(xs))
  }
}

def sumElementsList(xs: List[dyn[curry1[Iter,Int]]]) : Int =
  xs match {
    case Nil => 0
    case hd :: tl => sumElements(hd.*) + sumElementsList(tl)
  }
```

Test

```scala
given listIter[A]: Iter[List[A],A] with
  extension (l: List[A])
    def getValue = l.headOption
    def getNext = l.tail

given decIter : Iter[Int,Int] with
  extension (i: Int)
    def getValue = if (i >= 0) Some(i) else None
    def getNext = i - 1

sumElementsList(List(
  100,
  List(1,2,3),
  10))
```

Test

```scala
implicit def listIter[A]: Iter[List[A],A] = new {
  def getValue(l: List[A]) = l.headOption
  def getNext(l: List[A]) = l.tail
}

implicit val decIter : Iter[Int,Int] = new {
  def getValue(i: Int) = if (i >= 0) Some(i) else None
  def getNext(i: Int) = i - 1
}

sumElementsList(List(
  100,
  List(1,2,3),
  10))
```