PART 2

Object-Oriented Programming

Sub Type Polymorphism (Concept)

Motivation

We want:

```
object tom {
  val name = "Tom"
  val home = "02-880-1234"
}
object bob {
  val name = "Bob"
  val mobile = "010-1111-2222"
}
```

```
def greeting(r: ???) = "Hi " + r.name + ", How are you?"
greeting(tom)
greeting(bob)
```

Note that we have

```
tom: {val name: String; val home: String}
bob: {val name: String; val mobile: String}
```

Sub Types to the Rescue!

```
import reflect.Selectable.reflectiveSelectable

type NameHome = { val name: String; val home: String }
type NameMobile = { val name: String; val mobile: String}
type Name = { val name: String }
```

`NameHome <: Name` (`NameHome` is a sub type of `Name`)

`NameMobile <: Name` (`NameMobile` is a sub type of `Name`)

```
def greeting(r: Name) = "Hi " + r.name + ", How are you?"
greeting(tom)
greeting(bob)
```

Sub Types

- The sub type relation is kind of the subset relation.
- But they are NOT the same.

- T <: S
  Every element of T can be used as that of S.

- *Cf.* T is a subset of S.
  Every element of T is that of S.

- Why polymorphism?
  A function of type S=>R can be used as T=>R for many sub types T of S.
  Note that S=>R <: T=>R when T <: S.

Summary: Subtype Polymorphism

- Subtype Polymorphism
  - Program against known datatypes with common structures
  - How is it possible?

Two Kinds of Sub Types

- Structural Sub Types (a.k.a. Duck Typing)
  - The system implicitly determines the sub type relation by the structures of data types.
  - Structurally equivalent types are treated the same.

- Nominal Sub Types (a.k.a. Ad hoc Polymorphism)
  - The user explicitly specify the sub type relation using the names of data types.
  - Structurally equivalent types with different names may be treated differently.

Structural Sub Types

General Sub Type Rules

- Reflexivity:
  For any type T, we have:

$$T <: T$$

- Transitivity:
  For any types T, S, R, we have:

$$\frac{T <: R \qquad R <: S}{T <: S}$$

Sub Types for Special Types

- Nothing: The empty set
- Any: The set of all values

- For any type T, we have:

$$\text{Nothing} <: T <: \text{Any}$$

- Example

```
val a : Int = 3
val b : Any = a
def f(a: Nothing) : Int = a
```

Sub Types for Records

- Permutation

$$\frac{}{\{\ldots;\ x\!: T1;\ y\!: T2;\ \ldots\} <: \{\ldots;\ y\!: T2,\ x\!: T1;\ \ldots\}}$$

- Width

$$\frac{}{\{\ldots;\ x\!: T;\ \ldots\} <: \{\ldots;\ \ldots\}}$$

- Depth

$$\frac{T <: S}{\{\ldots;\ x\!: T;\ \ldots\} <: \{\ldots;\ x\!: S;\ \ldots\}}$$

Sub Types for Records

- Example

  {val x: { val y: Int; val z: String}, val w: Int}

  <:   (by permutation)

  {val w: Int; val x: { val y: Int; val z: String}}

  <:   (by depth & width)

  {val w: Int; val x: {val z: String}}

Sub Types for Tuples

- Depth

$$\frac{T <: S}{(\ldots, T, \ldots) <: (\ldots, S, \ldots)}$$

Sub Types for Functions

- Function Sub Type

$$\frac{T <: T' \qquad S <: S'}{(T' => S) <: (T => S')}$$

- Example

```
import reflect.Selectable.reflectiveSelectable
def foo(s: {val a: Int; val b: Int}) : {val x: Int; val y: Int} = {
  object tmp {
    val x = s.b
    val y = s.a
  }
  tmp
}
val gee: {val a: Int; val b: Int; val c: Int} => {val x: Int} =
  foo _
```

Classes

Class: Parameterized Record

```
import reflect.Selectable.reflectiveSelectable

type gee_type = {val name:String; val age: Int; def getPP(): String}
def gee_fun(_name: String, _age: Int) : gee_type = {
  if (!(_age >= 0 && _age < 200)) throw new Exception("Out of range")
  object tmp {
    val name : String = _name
    val age : Int = _age
    def getPP() : String = name + " of age " + age.toString() }
  tmp }
val gee : gee_type = gee_fun("David Jones",25)

gee.getPP()
```

Class: Parameterized Record

```
class foo_type(_name: String, _age: Int) {
  if (!(_age >= 0 && _age < 200)) throw new Exception("Out of range")
  val name : String = _name
  val age : Int = _age
  def getPP() : String = name + " of age " + age.toString() }
val foo : foo_type = new foo_type("David Jones",25)
foo.getPP()
```

use: `foo.name` `foo.age` `foo.getPP`

- `foo` is a value of `foo_type`
- `gee` is a value of `gee_type`

Class: No Structural Sub Typing

- Records: Structural sub-typing

```
foo_type <: gee_type
```

- Classes: Nominal sub-typing

$$\text{gee\_type} \not<: \text{foo\_type}$$

```
val v1 : gee_type = foo
val v2 : foo_type = gee  // type error

def greeting(r:{val name:String}) =
  "Hi " + r.name + ", How are you?"
greeting(foo)
```

Structural Types vs. Nominal Types

- Structural Types
  - Includes arbitrary values with the required structures as elements
  - Allows arbitrary types with the required structures as sub types
  - Cannot assume any properties on their elements

- Nominal Types
  - Includes only specific values as elements
  - Allows only specific types as sub types
  - Can assume specific properties on their elements

Class: Can be Recursive!

```
class MyList[A](v: A, nxt: Option[MyList[A]]) {
  val value : A = v
  val next : Option[MyList[A]] = nxt
}
type YourList[A] = Option[MyList[A]]

val t : YourList[Int] =
  Some(new MyList(3,Some (new MyList(4,None))))
val s : YourList[Int] =
  None
```

Note on Null value

- `null`: The special element of every class & structural type

- `null` is often used to represent None instead of using an Option type (Efficient but Not Safe)

- It is discouraged to use `null` in Scala although Scala supports `null` for compatibility with Java.

Simplification using Argument Members

```scala
class MyList[A](v: A, nxt: Option[MyList[A]]) {
  val value : A = v
  val next : Option[MyList[A]] = nxt
}
```

```scala
class MyList[A](val value:A, val next:Option[MyList[A]]) {
}
```

```scala
class MyList[A](val value:A, val next:Option[MyList[A]])
```

Simplification using Companion Object

```
class MyList[A](val value:A, val next:Option[MyList[A]])
object MyList
{ def apply[A](v: A, nxt: Option[MyList[A]]) =
    new MyList(v,nxt)
}

type YourList[A] = Option[MyList[A]]
object YourList
{ def apply[A](v: A, nxt: Option[MyList[A]]) =
    Some(new MyList(v,nxt))
}
val t0 = None
val t1 = Some(new MyList(3,Some(MyList(4,None))))
val t2 = YourList(3,(YourList(4,None)))
```

Exercise

Define a class "`MyTree[A]`" for binary trees:

MyTree[A] =
    (value: A) *
    (left: Option[MyTree[A]]) *
    (right: Option[MyTree[A]])

Solution

```scala
class MyTree[A](v: A,
                lt: Option[MyTree[A]],
                rt: Option[MyTree[A]]) {
  val value = v
  val left = lt
  val right = rt
}

type YourTree[A] = Option[MyTree[A]]

val t0 : YourTree[Int] = None
val t1 : YourTree[Int] = Some(new MyTree(3, None, None))
val t2 : YourTree[Int] =
  Some(new MyTree(3, Some (new MyTree(4,None,None)), None))
```

Simplified Solution

```
class MyTree[A](val value : A,
                val left : Option[MyTree[A]],
                val right : Option[MyTree[A]])

type YourTree[A] = Option[MyTree[A]]

object YourTree
{ def apply[A](v:A, lt:Option[MyTree[A]], rt:Option[MyTree[A]]) =
    Some(new MyTree(v,lt,rt))
}

val t0: YourTree[Int] = None
val t1: YourTree[Int] = YourTree(3,None,None)
val t2: YourTree[Int] = YourTree(3,YourTree(4,None,None),None)
```

Nominal Sub Typing for Classes

Nominal Sub Typing, a.k.a. Inheritance

```
class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = a + y
  def f(z: Int) : Int = b + y + z
}

class gee_type(x: Int) extends foo_type(x+1,x+2) {
  val c : Int = f(x) + b
}
```

gee_type <: foo_type

```
(new gee_type(30)).c
def test(f: foo_type) = f.a + f.b
test(new foo_type(10,20))
test(new gee_type(30))
```

Overriding

```
class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = 0
  def f(z: Int) : Int = b * z
}
class gee_type(x: Int) extends foo_type(x+1,x+2) {
 override def b = 10
 // or, override def b = super.b + 10
 val c : Int = f(x) + b
}

(new gee_type(30)).c
def test(v: foo_type) =
 println(v.f(42))
test(new foo_type(1,2))
test(new gee_type(0))
```

Overriding vs. Overloading

```
class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = 0
  def f(z: Int) : Int = b * z
}
class gee_type(x: Int) extends foo_type(x+1,x+2) {
  def f(z: String) : Int = 77
}
```

Q: Can we override with a different type?

```
override def f(z: String): Int = 77   //No,  arg: diff type
def f(z: String): Int = 77   // Overloading, arg: diff type
override def f(z: Int): Int = 77 //Yes, arg: same type
```

Example: MyList using Inheritance

```
class MyList[A](v: A, nxt: Option[MyList[A]]) {
  val value : A = v
  val next : Option[MyList[A]] = nxt
}
type YourList[A] = Option[MyList[A]]
val t : YourList[Int] =
  Some(new MyList(3,Some (new MyList(4,None))))
```

```
class MyList[A]()
class MyNil[A]() extends MyList[A]
class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A]
val t: MyList[Int] =
    new MyCons(3, new MyCons(4, new MyNil()))
```

Simplification: MyList

```
class MyList[A]

class MyNil[A]() extends MyList[A]
object MyNil { def apply[A]() = new MyNil[A]() }

class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A]
object MyCons {
  def apply[A](hd:A, tl:MyList[A]) = new MyCons[A](hd, tl)}

val t: MyList[Int] = MyCons(3, MyNil())

def length(x: MyList[Int]) = ???
```

Example: MyList with match

```
abstract class MyList[A]() {
  def matches[R](nilE: =>R, consE: (A,MyList[A]) => R) : R
}
class MyNil[A]() extends MyList[A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A]) => R) : R =
    nilE
}
class MyCons[A](val hd: A, val tl: MyList[A]) extends MyList[A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A]) => R) : R =
    consE(hd,tl)
}
def length[A](l: MyList[A]) : Int =
  l.matches(0,
                (hd, tl) => 1 + length(tl))

length(new MyCons(10, new MyCons(5, new MyNil())))
```

Case Class

```
sealed abstract class MyList[A] { … }
case class MyNil[A]() extends MyList[A] { … }
~~object MyNil { def apply[A]() = new MyNil[A]() }~~
case class MyCons[A](~~val~~ hd: A, ~~val~~ tl: MyList[A])
  extends MyList[A] { … }
~~object MyCons {~~
~~  def apply[A](hd:A, tl:MyList[A]) = new MyCons[A](hd, tl)}~~
val t: MyList[Int] = MyCons(3, MyNil())
```

Allow Pattern Matching

```
def length(x: MyList[Int]): Int =
  x match {
    case MyNil() => 0
    case MyCons(hd, tl) => 1 + length(tl)
  }
```

Cf. `sealed abstract class MyList[A]`

Exercise

Define "MyTree[A]" using sub class.

```
class MyTree[A](v: A,
                lt: Option[MyTree[A]],
                rt: Option[MyTree[A]]) {
  val value = v
  val left = lt
  val right = rt
}

type YourTree[A] = Option[MyTree[A]]
```

Solution

```scala
sealed abstract class MyTree[A]
case class Empty[A]() extends MyTree[A]
case class Node[A](value:A, left:MyTree[A], right:MyTree[A])
  extends MyTree[A]

val t : MyTree[Int] =
  Node(3, Node(4,Empty(),Empty()), Empty())

t match {
  case Empty() => 0
  case Node(v,l,r) => v
}
```

Solution with Monotonicity

```
// sealed abstract class MyTree[A]
// case class Empty[A]() extends MyTree[A]
// case class Node[A](value:A, left:MyTree[A], right:MyTree[A])
//   extends MyTree[A]

// MyTree[+A]: A <: B  ⟹  MyTree[A] <: MyTree[B]
// MyTree[-A]: A <: B  ⟹  MyTree[B] <: MyTree[A]
sealed abstract class MyTree[+A]
case object Empty extends MyTree[Nothing]
case class Node[A](value:A, left:MyTree[A], right:MyTree[A])
  extends MyTree[A]

val t : MyTree[Int] = Node(3, Node(4,Empty,Empty), Empty)
t match {
  case Empty => 0
  case Node(v,l,r) => v
}
```

Solution with enum

```
// sealed abstract class MyTree[+A]
// case object Empty extends MyTree[Nothing]
// case class Node[A](value:A, left:MyTree[A], right:MyTree[A])
//   extends MyTree[A]

enum MyTree[+A]:
  case Empty  //: MyTree[Nothing]
  case Node(value: A, left: MyTree[A], right: MyTree[A])
import MyTree._

val t : MyTree[Int] = Node(3, Node(4,Empty,Empty), Empty)
t match {
  case Empty => 0
  case Node(v,l,r) => v
}
```

Encoding ADT using classes: Monotonicity

```
sealed abstract class MyList[+A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A])=>R) : R
  def append[B>:A](l: MyList[B]) : MyList[B]
}
object MyNil extends MyList[Nothing] {
  def matches[R](nilE: =>R, consE: (Nothing,MyList[Nothing])=>R) = nilE
  def append[B](l: MyList[B]) = l
}
class MyCons[A](val hd: A, val tl: MyList[A]) extends MyList[A] {
  def matches[R](nilE: =>R, consE: (A,MyList[A])=>R) = consE(hd,tl)
  def append[B>:A](l: MyList[B]) = new MyCons[B](hd, tl.append(l))
}
object MyCons{ def apply[A](hd:A, tl:MyList[A]) = new MyCons[A](hd, tl) }
def length[A](l: MyList[A]) : Int =
  l.matches(
    0,
    (_,tl) => 1 + length(tl) )
length(MyCons(3, MyCons(2, MyNil)).append(MyCons(1,MyNil)))
```

Abstract Classes for Interface

Abstract Class: Interface

- Abstract Classes
  - Can be used to abstract away the implementation details.

Abstract classes for Interface
Concrete sub-classes for Implementation

Abstract Class: Interface

➢ Example Interface

```
// Written by Alice
// if getValue(i) returns None, you should not use i.getNext()
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.getValue match {
    case None => 0
    case Some(n) => f(n) + sumElements(f)(xs.getNext)
  }
def sumElementsId(xs:Iter[Int]) =
  sumElements((x:Int)=>x)(xs)
```

Concrete Class: Implementation

```scala
// Written by Bob
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...")
}
case class MyCons[A](hd: A, tl: MyList[A])
  extends MyList[A]
{
  def getValue = Some(hd)
  def getNext = tl
}

val t1 = MyCons(3, MyCons(5, MyCons(7, MyNil())))

sumElementsId(t1)
```

Exercise

Define `IntCounter(n)` that implements the interface `Iter[A]`.

```
// Written by Catherine
class IntCounter(n: Int) extends Iter[Int] {
  def getValue = ???
  def getNext = ???
}

sumElementsId(new IntCounter(100))
```

Solution

Define `IntCounter(n)` that implements the interface `Iter[A]`.

```
// Written by Catherine
class IntCounter(n: Int) extends Iter[Int] {
  def getValue = if (n >= 0) Some(n) else None
  def getNext = new IntCounter(n-1)
}

sumElementsId(new IntCounter(100))
```

A Better Interface

```
abstract class Iter[A] {
  def get: Option[(A,Iter[A])]
}
def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.get match {
    case None => 0
    case Some(n,nxt) => f(n) + sumElements(f)(nxt)
  }
def sumElementsId(xs:Iter[Int]) = sumElements((x:Int)=>x)(xs)
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def get = None }
case class MyCons[A](hd: A, tl: MyList[A]) extends MyList[A] {
  def get = Some(hd,tl) }
class IntCounter(n: Int) extends Iter[Int] {
  def get = if (n >= 0) Some(n, new IntCounter(n-1)) else None }
```

More on Abstract Classes

Problem: Iter for MyTree

```scala
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

// Written by David
sealed abstract class MyTree[A]
case class Empty[A]() extends MyTree[A]
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A]
```

Q: Can MyTree[A] implement Iter[A]?

Try it, but it is not easy.

Possible Solution

```scala
// Written by David
sealed abstract class MyTree[A] extends Iter[A]
case class Empty[A]() extends MyTree[A] {
  def getValue = None
  def getNext = this  }
case class Node[A](value: A, left: MyTree[A], right: MyTree[A])
     extends MyTree[A] {
  def getValue = Some(value)
  def getNext: MyTree[A] = {
    def merge_right(l : MyTree[A]): MyTree[A] = l match {
      case Empty() => right
      case Node(v, lt, rt) => Node(v, lt, merge_right(rt))  }
    merge_right(left)  } }
val t1 = Node(3, Node(7, Node(2, Empty(), Empty()), Empty()),
                  Node(8, Empty(), Empty()))
sumElements[Int]((x)=>x*x)(t1)
```

Solution: Better Interface

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}
abstract class Iterable[A] {
  def iter : Iter[A]
}

def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.getValue match {
    case None => 0
    case Some(n) => f(n) + sumElements(f)(xs.getNext)
  }
def sumElementsGen[A](f: A=>Int)(xs: Iterable[A]) : Int =
  sumElements(f)(xs.iter)
```

Let’s Use MyList

```
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...")
}
case class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A] {
  def getValue = Some(hd)
  def getNext = tl
}
```

MyTree <: Iterable (Try)

```
sealed abstract class MyTree[A] extends Iterable[A]

case class Empty[A]() extends MyTree[A] {
  val iter = MyNil()
}

case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A] {
  // "val iter" is more specific than "def iter",
  // so it can be used in a sub type.
  // In this example, "val iter" is also
  // more efficient than "def iter".
  val iter = MyCons(value, ???(left.iter,right.iter))
}
```

Extend MyList with append

```
sealed abstract class MyList[A] extends Iter[A] {
  def append(lst: MyList[A]) : MyList[A]
}
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...")
  def append(lst: MyList[A]) = lst
}
case class MyCons[A](val hd: A, val tl: MyList[A])
  extends MyList[A]
{
  def getValue = Some(hd)
  def getNext = tl
  def append(lst: MyList[A]) = MyCons(hd,tl.append(lst))
}
```

MyTree <: Iterable

```
sealed abstract class MyTree[A] extends Iterable[A] {
  def iter : MyList[A]
  // Note:
  // def iter : Int // Type Error because not (Int <: Iter[A])
}
case class Empty[A]() extends MyTree[A] {
  val iter = MyNil()
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A] {
  def iter = MyCons(value, left.iter.append(right.iter))
  // def iter = left.iter.append(MyCons(value,right.iter))
  // def iter = left.iter.append(right.iter.append(
  //               MyCons(value,MyNil())))
}
```

Test

```
def generateTree(n: Int) : MyTree[Int] = {
  def gen(lo:Int, hi: Int) : MyTree[Int] =
    if (lo > hi) Empty()
    else {
      val mid = (lo+hi)/2
      Node(mid, gen(lo,mid-1), gen(mid+1,hi))
    }
  gen(1,n)
}

sumElementsGen((x:Int)=>x)(generateTree(100))
```

Iter <: Iterable

```
abstract class Iterable[A] {
  def iter : Iter[A]
}

abstract class Iter[A] extends Iterable[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
  def iter = this
}

val lst : MyList[Int] =
  MyCons(3, MyCons(4, MyCons(2,MyNil())))

sumElementsGen ((x:Int)=>x)(lst)
```

Note: tail-recursive “append”

```
sealed abstract class MyList[A] extends Iter[A] {
  def append(lst: MyList[A]) : MyList[A] =
    MyList.revAppend(MyList.revAppend(this, MyNil()), lst)
}
object MyList { //  Mutual references are allowed between class T and object T
  //  Tail-recursive functions should be written in “object”, or as final methods
  def revAppend[A](lst1: MyList[A], lst2: MyList[A]): MyList[A] =
    lst1 match {
      case MyNil() => lst2
      case MyCons(hd, tl) => revAppend(tl, MyCons(hd, lst2))
    }
}
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = throw new Exception("...") }
case class MyCons[A](val hd:A, val tl:MyList[A]) extends MyList[A] {
  def getValue = Some(hd)
  def getNext = tl }
```

Lazy List

Problem: Inefficiency

```
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[A](f: A=>Int)(n: Int, xs: Iterable[A]) : Int = {
  def sumIter(res : Int, n: Int, xs: Iter[A]) : Int =
    if (n <= 0) res
    else xs.getValue match {
      case None => res
      case Some(v) => sumIter(f(v) + res, n-1, xs.getNext)
    }
  sumIter(0,n,xs.iter)
}
// Problem: takes a few seconds to get a single value
{ val t: MyTree[Int] = generateTree(200000)
  time (sumN((x:Int) => x)(1, t)) }
```

Solution 1: Using Lists of Trees

```
class MyTreeIter[A](val lst: MyList[MyTree[A]]) extends Iter[A] {
  val getValue = lst match {
    case MyCons(Node(v,_,_), _) => Some(v)
    case _ => None
  }
  def getNext = {
    val remainingTrees : MyList[MyTree[A]] = lst match {
      case MyNil() => throw new Exception("...")
      case MyCons(hd,tl) => hd match {
        case Empty() => throw new Exception("...")
        case Node(_,Empty(),Empty()) => tl
        case Node(_,lt,Empty()) => MyCons(lt,tl)
        case Node(_,Empty(),rt) => MyCons(rt,tl)
        case Node(_,lt,rt) => MyCons(lt,MyCons(rt,tl))
      }
    }
    new MyTreeIter(remainingTrees)
  } }
```

Lazy Iteration using Lists of Trees

```
sealed abstract class MyTree[A] extends Iterable[A]
case class Empty[A]() extends MyTree[A] {
  val iter = new MyTreeIter(MyNil())
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A]
{
  val iter = new MyTreeIter(MyCons(this,MyNil()))
}

{ val t: MyTree[Int] = generateTree(200000)
  time (sumN((x:Int) => x)(100, t))
  time (sumN((x:Int) => x)(100000, t))
}
```

Solution 2: Lazy List

```
sealed abstract class LazyList[A] extends Iter[A] {
  def append(lst: LazyList[A]) : LazyList[A]
}

case class LNil[A]() extends LazyList[A] {
  def getValue = None
  def getNext = throw new Exception("")
  def append(lst: LazyList[A]) = lst
}

class LCons[A](hd: A, _tl: =>LazyList[A]) extends LazyList[A] {
  lazy val tl = _tl
  def getValue = Some(hd)
  def getNext = tl
  def append(lst: LazyList[A]) = LCons(hd, tl.append(lst)) }
object LCons {
  def apply[A](hd: A, tl: =>LazyList[A]) = new LCons(hd, tl)
}
```

Note: “append” is not recursive!!!

Lazy Iteration using LazyList

```scala
sealed abstract class MyTree[A] extends Iterable[A] {
  def iter : LazyList[A]
}
case class Empty[A]() extends MyTree[A] {
  val iter = LNil()
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A]) extends MyTree[A] {
  lazy val iter = LCons(value, left.iter.append(right.iter))
  // lazy val iter = left.iter.append(LCons(value,right.iter))
  // lazy val iter = left.iter.append(right.iter.append(
  //                         LCons(value,LNil())))
}
{ val t: MyTree[Int] = generateTree(200000)
  time (sumN((x:Int) => x)(100, t))
  time (sumN((x:Int) => x)(100000, t))
}
```

Note: "iter" is not recursive!!!

Wrapper for Inheritance

Using a Wrapper Class

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

class ListIter[A](val list: List[A]) extends Iter[A] {
  def getValue = list.headOption
  def getNext = new ListIter(list.tail)
}

sumElements((x:Int)=>x)(new ListIter(List(1,2,3,4)))
```

MyTree Using ListIter

```
abstract class Iterable[A] {
  def iter : Iter[A]
}
sealed abstract class MyTree[A] extends Iterable[A] {
  def iter : ListIter[A]
}
case class Empty[A]() extends MyTree[A] {
  val iter : ListIter[A] = new ListIter(Nil)
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A])
  extends MyTree[A] {
  val iter : ListIter[A] = new ListIter(
    value::(left.iter.list ++ right.iter.list))
}
```

Test

```
val t : MyTree[Int] =
  Node(3, Node(4,Node(2,Empty(),Empty()),
     Node(3,Empty(),Empty()))),
     Node(5,Empty(),Empty()))

sumElementsGen((x:Int)=>x)(t)
```

Abstract Class With Associate Types

Using an Associate Type

```
abstract class Iterable[A] {
  type iter_t
  def iter: iter_t
  def getValue(i: iter_t) : Option[A]
  def getNext(i: iter_t) : iter_t
}

def sumElements[A](f:A=>Int)(xs: Iterable[A]) : Int = {
  def sumElementsIter(i: xs.iter_t) : Int =
    xs.getValue(i) match {
      case None => 0
      case Some(n) => f(n) + sumElementsIter(xs.getNext(i))
    }
  sumElementsIter(xs.iter)
}
```

MyTree Using List

```
sealed abstract class MyTree[A] extends Iterable[A] {
  type iter_t = List[A]
  def getValue(i: List[A]): Option[A] = i.headOption
  def getNext(i: List[A]): List[A] = i.tail
}
case class Empty[A]() extends MyTree[A] {
  val iter : List[A] = Nil
}
case class Node[A](value: A,
                   left: MyTree[A], right: MyTree[A])
  extends MyTree[A] {
  val iter = value :: (left.iter ++ right.iter) //Pre-order
//val iter = left.iter ++ (value :: right.iter) // In-order
//val iter = left.iter ++ (right.iter ++ List(value))
                                              //Post-order
}
```

Test

```
val t : MyTree[Int] =
  Node(3, Node(4,Node(2,Empty(),Empty()),
    Node(3,Empty(),Empty())),
    Node(5,Empty(),Empty()))

sumElements((x:Int)=>x)(t)
```

Abstract Class with Arguments

Abstract Class with Arguments

```
abstract class IterableH[A] extends Iterable[A] {
  def hasElement(a: A) : Boolean
}
abstract class IterableHE[A](eq: (A,A) => Boolean)
  extends IterableH[A]
{
  def hasElement(a: A) : Boolean = {
    def hasElementIter(i: iter_t) : Boolean =
      getValue(i) match {
        case None => false
        case Some(n) =>
          if (eq(a,n)) true
          else hasElementIter(getNext(i))
      }
    hasElementIter(iter)
  }
}
```

MyTree

```
sealed abstract class MyTree[A](eq:(A,A)=>Boolean)
  extends IterableHE[A](eq) {
  type iter_t = List[A]
  def getValue(i : List[A]) : Option[A] = i.headOption
  def getNext(i: List[A]) : List[A] = i.tail
}
case class Empty[A](eq: (A,A)=>Boolean)
  extends MyTree[A](eq) {
  val iter : List[A] = Nil
}
case class Node[A](eq: (A,A)=>Boolean,
               value: A, left: MyTree[A], right: MyTree[A])
  extends MyTree[A](eq) {
  val iter : List[A] = value :: (left.iter ++ right.iter)
}
```

Test

```
val leq = (x:Int,y:Int) => x == y
val lEmpty = Empty(leq)
def lNode(n: Int, t1: MyTree[Int], t2: MyTree[Int]) =
  Node(leq,n,t1,t2)

val t : MyTree[Int] =
  lNode(3, lNode(4,lNode(2,lEmpty,lEmpty),
                   lNode(3,lEmpty,lEmpty)),
           lNode(5,lEmpty,lEmpty))

sumElements((x:Int)=>x)(t)
t.hasElement(5)
t.hasElement(10)
```

Alternatively, Argument Elimination

```
abstract class IterableHE[A]
  extends Iterable[A]
{
  def eq(a:A, b:A) : Boolean
  def hasElement(a: A) : Boolean = {
    def hasElementIter(i: iter_t) : Boolean =
      getValue(i) match {
        case None => false
        case Some(n) =>
          if (eq(a,n)) true
          else hasElementIter(getNext(i))
      }
    hasElementIter(iter)
  }
}
```

MyTree

```
sealed abstract class MyTree[A] extends IterableHE[A] {
  type iter_t = List[A]
  def getValue(i : List[A]) : Option[A] = i.headOption
  def getNext(i: List[A]) : List[A] = i.tail
}
case class Empty[A](_eq:(A,A)=>Boolean) extends MyTree[A] {
  def eq(a:A, b:A) = _eq(a,b)
  val iter : List[A] = Nil
}
case class Node[A](_eq: (A,A)=>Boolean,
               value: A, left: MyTree[A], right: MyTree[A])
  extends MyTree[A] {
  def eq(a:A, b:A) = _eq(a,b)
  val iter : List[A] = value :: (left.iter ++ right.iter)
}
```

Test

```
val leq = (x:Int,y:Int) => x == y
val lEmpty = Empty(leq)
def lNode(n: Int, t1: MyTree[Int], t2: MyTree[Int]) =
  Node(leq,n,t1,t2)

val t : MyTree[Int] =
  lNode(3, lNode(4,lNode(2,lEmpty,lEmpty),
                   lNode(3,lEmpty,lEmpty)),
           lNode(5,lEmpty,lEmpty))

sumElements((x:Int)=>x)(t)
t.hasElement(5)
t.hasElement(10)
```

More on Classes

Motivating Example

```
class Primes(val prime: Int, val primes: List[Int]) {
  def getNext: Primes = {
    val p = computeNextPrime(prime + 2)
    new Primes(p, primes ++ (p :: Nil))
  }
  def computeNextPrime(n: Int) : Int =
    if (primes.forall((p:Int) => n%p != 0)) n
    else computeNextPrime(n+2)
}

def nthPrime(n: Int): Int = {
  def go(primes: Primes, k: Int): Int =
    if (k <= 1) primes.prime
    else go(primes.getNext, k - 1)
  if (n <= 0) 2 else go(new Primes(3, List(3)), n)
}
nthPrime(10000)
```

Multiple Constructors

```
class Primes(val prime: Int, val primes: List[Int]) {
  def this() = this(3, List(3))
  def getNext: Primes = {
    val p = computeNextPrime(prime + 2)
    new Primes(p, primes ++ (p :: Nil))
  }
  def computeNextPrime(n: Int) : Int =
    if (primes.forall((p:Int) => n%p != 0)) n
    else computeNextPrime(n+2)
}

def nthPrime(n: Int): Int = {
  def go(primes: Primes, k: Int): Int =
    if (k <= 1) primes.prime
    else go(primes.getNext, k - 1)
  if (n == 0) 2 else go(new Primes, n)
}
nthPrime(10000)
```

Access Modifiers

- Access Modifiers
  - Private: Only the class can access the member.
  - Protected: Only the class and its sub classes can access the member.

Using Access Modifiers

```
class Primes private (val prime:Int, protected val primes:List[Int])
{ def this() = this(3, List(3))
  def getNext: Primes = {
    val p = computeNextPrime(prime + 2)
    new Primes(p, primes ++ (p :: Nil))
  }
  private def computeNextPrime(n: Int) : Int =
    if (primes.forall((p:Int) => n%p != 0)) n
    else computeNextPrime(n+2)
}

def nthPrime(n: Int): Int = {
  def go(primes: Primes, k: Int): Int =
    if (k <= 1) primes.prime
    else go(primes.getNext, k - 1)
  if (n == 0) 2 else go(new Primes, n)
}
nthPrime(10000)
```

Traits for Multiple Inheritance

Multiple Inheritance Problem

- Multiple Inheritance
  - The famous "diamond problem"

```
class A(val a: Int)
class B extends A(10)
class C extends A(20)
class D extends B, C.
```

Problem 1: What is the value of (new D).a ?

Problem 2: The constructor of A must be executed once
because A may contain side effects such as
sending messages over the network.

Java's Solution: Interface

- Interface
  - An interface cannot contain any implementation
    but only types of its methods.
  - A class can inherit implementations from only one parent class
    but implement multiple interfaces.

Scala's Solution: Trait

- Traits
  - A trait can implement any of its methods,
    but should have only one constructor with no arguments.
  - An [abstract] class (resp. trait) X can
    "extends" one trait or [abstract] class with any (resp. no) arguments
    "with" multiple traits T_1 , … , T_n
    such that, for each i,
    the least superclass of T_i, if exists, should be a superclass of X
    where
    C is a superclass of T if C is an (abstract) class and T transitively "extends" C.
  - No cyclic inheritance is allowed.
- Property
  - For any ancestor class in the inheritance tree of a class:
    - Its constructor with arguments can appear at most once
    - Its constructor with no argument can appear multiple times

Example

```
class A(val a : Int) {
  def this () = this(0)
}
trait B {
  def f(x: Int): Int = x
}
trait C extends A with B {
  def g(x: Int): Int = x + a
}
trait D extends B {
  def h(x: Int): Int = f(x + 50)
}
class E extends A(10) with C with D {
  override def f(x: Int) = x * a
}

val e = new E
```

Algorithm for Multiple Inheritance

- Algorithm
  - Give a linear order among all ancestors by "post-order" traversing without revisiting the same node.
  - Invoke the constructors once in that order.

    Note. Post-order traversal of a class C means
    - Recursively post-order traverse C's first parent; …;
    - Recursively post-order traverse C's last parent; and
    - Visit C.

    By post-order traversing from "E" in the previous example, we have the order: `A(10) -> B -> C -> D -> E`

```
val e = new E
e.a // 10                      e.f(100) // 100*10
e.g(100) // 100 + 10           e.h(100) // (100 + 50) * 10
```

  - A constructor with arguments is always visited before the same constructor with no arguments.
  - Compile error if the same field is implemented by multiple classes

A Simple Example With Traits

Motivation

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

class ListIter[A](val list: List[A]) extends Iter[A] {
  def getValue = list.headOption
  def getNext = new ListIter(list.tail)
}

abstract class Dict[K,V] {
  def add(k: K, v: V): Dict[K,V]
  def find(k: K): Option[V]
}
```

Q: How can we extend ListIter and implement Dict?

Interface using Traits

```
// abstract class Dict[K,V] {
//   def add(k: K, v: V): Dict[K,V]
//   def find(k: K): Option[V] }

trait Dict[K,V] {
  def add(k: K, v: V): Dict[K,V]
  def find(k: K): Option[V]
}
```

Implementing Traits

```
class ListIterDict[K,V]
      (eq: (K,K)=>Boolean, list: List[(K,V)])
      extends ListIter[(K,V)](list)
         with Dict[K,V]
{
  def add(k:K,v:V): ListIterDict[K,V] =
    new ListIterDict(eq,(k,v)::list)
  def find(k: K) : Option[V] = {
    def go(l: List[(K, V)]): Option[V] = l match {
        case Nil => None
        case (k1, v1) :: tl =>
          if (eq(k, k1)) Some(v1) else go(tl) }
    go(list) }
}
```

Test

```
def sumElements[A](f: A=>Int)(xs: Iter[A]) : Int =
  xs.getValue match {
    case None => 0
    case Some(n) => f(n) + sumElements(f)(xs.getNext)
  }

def find3(d: Dict[Int,String]) = {
  d.find(3)
}

val d0 = new ListIterDict[Int,String]((x,y)=>x==y,Nil)
val d = d0.add(4,"four").add(3,"three")

sumElements[(Int,String)](x=>x._1)(d)
find3(d)
```

Mixin with Traits

Motivation: Mixin Functionality

```
abstract class Iter[A] {
  def getValue: Option[A]
  def getNext: Iter[A]
}

class ListIter[A](val list: List[A]) extends Iter[A]
{
  def getValue = list.headOption
  def getNext: ListIter[A] = new ListIter(list.tail)
}

trait MRIter[A] extends Iter[A] {
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C = ???
}
```

Mixin Composition

```
trait MRIter[A] extends Iter[A] {
  override def getNext: MRIter[A]
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C =
    getValue match {
      case None => ival
      case Some(v) =>
        combine(f(v), getNext.mapReduce(combine, ival, f))
    }
}

class MRListIter[A](list: List[A])
  extends ListIter (list) with MRIter[A]
{
 override def getNext = new MRListIter(super.getNext.list)
                      // new MRListIter(list.tail)
}
val mr = new MRListIter[Int](List(3,4,5))
mr.mapReduce[Int,Int]((b,c)=>b+c,0,(a)=>a*a)
```

Mixin Composition: A Better Way

```
trait MRIter[A] extends Iter[A] {
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C = {
    def loop(c: Iter[A]): C = c.getValue match {
      case None => ival
      case Some(v) => combine(f(v), loop(c.getNext))
    }
    loop(this)
  }
}

class MRListIter[A](list: List[A])
  extends ListIter (list) with MRIter[A]

val mr = new MRListIter[Int](List(3,4,5))

// or, val mr = new ListIter(List(3,4,5)) with MRIter[Int]

mr.mapReduce[Int,Int]((b,c)=>b+c,0,(a)=>a*a)
```

Syntactic Sugar: new A with B with C { … }

```
new A(...) with B1 ··· with Bm {
  code
}
```

is equivalent to

```
{
  class _tmp_(args) extends A(args) with B1 ··· with Bm {
    code
  }
  new _tmp_(...)
}
```

Intersection Types

Intersection Types

- Typing Rule

$$\frac{t : T1 \qquad t : T2}{t : T1 \text{ with } T2}$$

- Example

```
trait A { val a: Int = 0 }
trait B { val b: Int = 0 }
class C extends A with B {
  override val a = 10
  override val b = 20
  val c = 30
}
val x = new C
val y: A with B = x
y.a // 10
y.b // 20
y.c // type error
```

Subtype Relation for “with”

The subtype relation for “with” is structural.

- Permutation

$$\frac{}{\ldots \text{ with } T1 \text{ with } T2 \ldots <: \ldots \text{ with } T2 \text{ with } T1 \ldots}$$

- Width

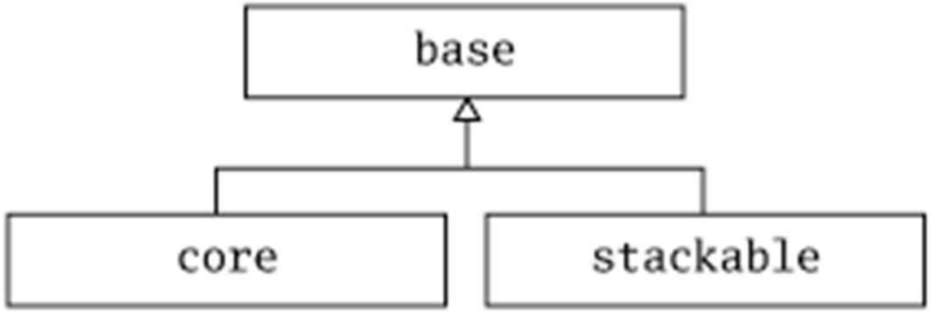
$$\frac{}{\ldots \text{ with } T \ldots <: \ldots \ldots}$$

- Depth

$$\frac{T <: S}{\ldots \text{ with } T \ldots <: \ldots \text{ with } S \ldots}$$

Stacking with Traits

Typical Hierarchy in Scala



- **BASE**
  Interface (trait or abstract class)

- **CORE**
  Functionality (trait or concrete class)

- **CUSTOM**
  Modifications (each in a separate, composable trait)

IntStack: Base

➢ **BASE**

```
trait Stack[A] {
  def get(): (A, Stack[A])
  def put(x: A): Stack[A]
}
```

IntStack: Core

- **CORE**

```
class BasicIntStack protected (xs: List[Int]) extends Stack[Int]
{
  override val toString = "Stack:" + xs.toString
  def this() = this(Nil)

  def get():(Int,Stack[Int]) = (xs.head,new BasicIntStack(xs.tail))
  def put(x:Int): Stack[Int] = new BasicIntStack(x :: xs)
}

val s0 = new BasicIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
```

IntStack: Custom Modifications

- **CUSOM**

```
trait Doubling extends Stack[Int] {
  abstract override def put(x: Int): Stack[Int] = super.put(2 * x)
}

trait Incrementing extends Stack[Int] {
  abstract override def put(x: Int): Stack[Int] = super.put(x + 1)
}

trait Filtering extends Stack[Int] {
  abstract override def put(x: Int): Stack[Int] =
    if (x >= 0) super.put(x) else this
}
```

IntStack: Stacking

- **Stacking**

```
class DIFIntStack protected (xs: List[Int])
  extends BasicIntStack(xs)
  with Doubling with Incrementing with Filtering
{
  def this() = this(Nil)
}

val s0 = new DIFIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
val (v2,s6) = s5.get()
```

IntStack: Core (Correct)

- **CORE**

```
class BasicIntStack protected (xs: List[Int]) extends Stack[Int]
{
  override val toString = "Stack:" + xs.toString
  def this() = this(Nil)
  protected def mkStack(xs: List[Int]): Stack[Int] =
    new BasicIntStack(xs)
  def get(): (Int,Stack[Int]) = (xs.head, mkStack(xs.tail))
  def put(x: Int): Stack[Int] = mkStack(x :: xs)
}

val s0 = new BasicIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
```

IntStack: Stacking (Correct)

➤ **Stacking**

```
class DIFIntStack protected (xs: List[Int])
  extends BasicIntStack(xs)
  with Doubling with Incrementing with Filtering
{
  def this() = this(Nil)
  override def mkStack(xs: List[Int]): Stack[Int] =
    new DIFIntStack(xs)
}

val s0 = new DIFIntStack
val s1 = s0.put(3)
val s2 = s1.put(-2)
val s3 = s2.put(4)
val (v1,s4) = s3.get()
val (v2,s5) = s4.get()
```

Additional Resources

- Traits
  - http://www.scala-lang.org/old/node/126
- Mixin Composition
  - http://www.scala-lang.org/old/node/117
- Stackable Trait Pattern
  - http://www.artima.com/scalazine/articles/stackable_trait_pattern.html
- Multiple Inheritance via Traits
  - https://www.safaribooksonline.com/blog/2013/05/30/traits-how-scala-tames-multiple-inheritance/
- UCSD CSE 130
  - http://cseweb.ucsd.edu/classes/wi14/cse130-a/