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:

Nothing <: T <: 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

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

- 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

$$\texttt{foo\_type} <: \texttt{gee\_type}$$

- Classes: Nominal sub-typing

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

```
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

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

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

```
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)))
```

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]`

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

```
// 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)
```

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

```
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

```
// 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)) }
```