Principles of Programming

(4190.210)

Chung-Kil Hur (허충길)

Department of Computer Science and Engineering

Seoul National University

Syllabus

- Lecture
  - Tue & Thu: 11:00 ~ 12:30 (@302-106)
  - https://github.com/snu-sf-class/pp202202
  - Bring your laptop to lectures
- Instructor
  - Chung-Kil Hur
  - http://sf.snu.ac.kr/gil.hur/
- Teaching Assistant
  - Hoyoung Joo & Dongjae Lee
  - Contact Tas at pp@sf.snu.ac.kr
- Grading (tentative)
  - Attendance: 5%
  - Assignments: 25%
  - Midterm exam: 30%
  - Final exam: 40%

Free Laptop Rental

You can rent a laptop from the department for free.

Who am I?

- Prof. Chung-Kil (Gil) Hur [허충길]
  - Education: KAIST (B.S.), Univ of Cambridge (Ph.D.)
  - (High school) Bronze medal in IMO 1994 @Hongkong.
  - Software Foundations Lab
    http://sf.snu.ac.kr
  - Research Topics
    - Software Verification
    - Low-level Language Semantics (C/C++/LLVM/Rust)
    - Relaxed-Memory Concurrency
  - Our collaborators
    - Many in USA, UK, Germany, France, Israel.
  - Publications in Programming Languages (PL) area
    - POPL and PLDI are the two most prestigious conferences in PL area (see https://csrankings.org)
    - Published 10 PLDI and 5 POPL papers for the last 10 years (6th in the world)
    - Published either POPL or PLDI paper for 12 years in a row (2nd in the world)
    - Received distinguished paper awards from PLDI 2017, 2021 & POPL 2020

Introduction

Overview

- Part 1
  Functional Programming

- Part 2
  Object-Oriented Programming

- Part 3
  Type Class (Type-Oriented) Programming

- Part 4
  ~~Imperative Programming~~

Imperative vs. Functional Programming

- Imperative Programming
  - Computation by memory reads/writes
  - Sequence of read/write operations
  - Repetition by loop
  - More procedural (ie, describe how to do)
  - Easier to write efficient code

```
sum = 0;
i = n;
while (i > 0) {
   sum = sum + i;
   i = i - 1;
}
```

- Functional Programming
  - Computation by function application
  - Composition of function applications
  - Repetition by recursion
  - More declarative (ie, describe what to do)
  - Easier to write safe code

```
def sum(n: Int): Int
  if (n <= 0)
     0
  else
     n + sum(n-1)
```

Both Imperative & Functional Style Supported

- Many languages support both imperative & functional style
  - More imperative: Java, Javascript, C++, Python, Rust, …
  - More functional: OCaml, SML, Lisp, Scheme, …
  - Middle: Scala
  - Purely functional: Haskell (monadic programming)

Object-Oriented vs. Type-Oriented Programming

- Goal
  - To support module systems for writing large software
  - Specifically, to support code abstraction & reuse

- Object-Oriented Programming
  - Class Instance (ie, Object): data + methods (ie, functions)
  - Code abstraction & reuse together via inheritance

- Type Class (Type-Oriented) Programming
  - Type Class Instance: type + methods (ie, functions)
  - Abstraction and Reuse are separated
    - Code abstraction via type class instantiation
    - Code reuse via composition

Why Scala?

- Why Scala?
  - Equally well support both imperative & functional style
  - Many advanced features (both OOP & Type class supported)
  - Compatible with Java

PART 1
Functional Programming with Function Applications

Values, Names, Functions and Evaluations

Values, Expressions, Names

- Types and Values
  - A type is a set of values
  - Int: {-2147483648,…,-1,0,1, …,2147483647} //32-bit integers
  - Double: 64-bit floating point numbers // real numbers in practice
  - Boolean: {true, false}
  - …
- Expressions
  - Composition of
    values, names, primitive operations, …
- Name Binding
  - Binding expressions to names
- Examples

```
def a = 1 + (2 + 3)
def b = 3 + a * 4
```

Evaluation

- Evaluation
  - Reducing an expression into a value
  - Strategy
  1. Take a name or an operator (outer to inner)
  2. (name) Replace the name with its associated expression
  3. (name) Evaluate the expression
  4. (operator) Evaluate its operands (left to right)
  5. (operator) Apply the operator to its operands

- Examples

```
5+b ~ 5+(3+a*4) ~ 5+(3+(1+(2+3))*4) ~ … ~ 32
```

Functions and Substitution

- Functions
  - Expressions with Parameters
  - Binding functions to names

```
def f(x: Int): Int = x + a
```

- Evaluation by substitution
  - …
  - (function) Evaluate its operands (left to right)
  - (function)
    Replace the function application by the expression of the function
    Replace its parameters with the operands

```
5+f(f(3)+1) ~ 5+f((3+a)+1) ~ … ~ 5+f(10) ~ 5+(10+a)
~ … ~ 21
```

Simple Recursion

- Recursion
  - Use X in the definition of X
  - Powerful mechanism for repetition
  - Nothing special but just rewriting

```
def sum(n: Int) : Int =
  if (n <= 0)
    0
  else
    n + sum(n-1)
```

```
sum(2) ~ if (2<=0) 0 else (2+sum(2-1)) ~
2+sum(1) ~ 2+(if (1<=0) 0 else (1+sum(1-1))) ~
2+(1+sum(0)) ~ 2+(1+(if (0<=0) 0 else (0+sum(0-1))))
~ 2+(1+0) ~ 3
```

Termination/Divergence

Evaluation may not terminate

- Termination
  - An expression may reduce to a value

- Divergence
  - An expression may reduce forever

```
def loop: Int = loop
```

```
loop ~ loop ~ loop ~ …
```

Evaluation strategy: Call-by-value, Call-by-name

```
f(e1,e2)
```

- Call-by-value
  - Evaluate the arguments first, then apply the function to them
- Call-by-name
  - Just apply the function to its arguments, without evaluating them.

```
def square (x: Int) = x * x

[cbv]square(1+1) ~ square(2) ~ 2*2 ~ 4

[cbn]square(1+1) ~ (1+1)*(1+1) ~ 2*(1+1) ~ 2*2 ~ 4
```

CBV, CBN: Differences

- Call-by-value
  - Evaluates arguments once
- Call-by-name
  - Do not evaluate unused arguments

- Question
  - Do both always result in the same value?

Scala's evaluation strategy

- Call-by-value
  - By default
- Call-by-name
  - Use "=>"

```
def one(x: Int, y: =>Int) = 1

one(1+2, loop)

one(loop, 1+2)
```

Scala's name binding strategy

- Call-by-value
  - Use "val" (also called "field") e.g. `val x = e`
  - Evaluate the expression first, then bind the name to it
- Call-by-name
  - Use "def" (also called "method") e.g. `def x = e`
  - Just bind the name to the expression, without evaluating it
  - Mostly used to define functions

```
def a = 1 + 2 + 3
val a = 1 + 2 + 3 // 6
def b = loop
val b = loop

def f(a: Int, b: Int): Int = a*b - 2
```

Conditional Expressions

- If-else
  - `if (`b`)` e_1 `else` e_2
  - b : Boolean expression
  - e_1, e_2: expressions of the same type
- Rewrite rules:
  - `if (true)` e_1 `else` e_2 → e_1
  - `if (false)` e_1 `else` e_2 → e_2

```
def abs(x: Int) = if (x >= 0) x else -x
```

Boolean Expressions

- Boolean expression
  - `true, false`
  - `!b`
  - `b && b`
  - `b || b`
  - `e <= e, e >= e, e < e, e > e, e == e, e != e`
- Rewrite rules:
  - `!true → false`
  - `!false → true`
  - `true && b → b`
  - `false && b → false`
  - `true || b → true`
  - `false || b → b`

`true && (loop == 1) ~ loop == 1 ~ loop == 1`

Exercise: and, or

- Write two functions
  - `and(x,y) == x && y`
  - `or(x,y) == x || y`
  - Do not use `&&,||`

```
and(false,loop==1)
~ if (false) loop==1 else false
~ false

and(true,loop==1)
~ if (true) loop==1 else false
~ loop==1 ~ loop==1 …
```

Solution

```
def and(x: Boolean, y: =>Boolean) =
  if (x) y else false

def or(x: Boolean, y: =>Boolean) =
  if (x) true else y
```

Exercise: square root calculation

➢Calculate square roots with Newton's method

```
def isGoodEnough(guess: Double, x: Double) =
  ??? // guess*guess is 99.9% close to x

def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2

def sqrtIter(guess: Double, x: Double): Double =
  ??? // repeat improving guess until it is good enough

def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Solution

➢Calculate square roots with Newton's method

```
def isGoodEnough(guess: Double, x: Double) =
  guess*guess/x > 0.999 && guess*guess/x < 1.001

def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2

def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess,x)) guess
  else sqrtIter(improve(guess,x),x)

def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Blocks and Name Scoping

Blocks in Scala

- Block
  - ```
    { val x1 = e1
      def x2 = e2
      e
    }
    ```
  - Is an expression
  - Allow nested name binding
  - Allow arbitrary order of "`def`"s, but not "`val`"s (think about why)
    ```

Scope of names

➢Block

```
val t = 0
def f(x: Int) = t + g(x+1)
def g(y: Int) = y * y
val x = f(5)
val r = {
  val t = 10
  val s = f(5)
  s - t }
t + r
```

- A definition inside a block is only accessible within the block
- A definition inside a block shadows definitions of the same name outside the block
- A definition is accessible inside a nested block unless it is shadowed
- The same name cannot be defined twice in the same block
- A function is evaluated under the environment where it is defined, not the environment where it is invoked.

Problem

```
// should be allowed
{
  def f(x:Int) = g(x)
  def g(x: Int) = 10
  val x = f(10)
  x
}

// should not be allowed
{
  def f(x:Int) = g(x)
  val x = f(10)
  def g(x: Int) = 10
  x
}
```

Safety Checking

Safety checking for ruling out accessing undefined values is simple.

- For “val x = e”,
  all names in “e” should be defined before this definition.
- For “def x = e”,
  all names in “e” should be defined before the next “val” definition.

/* The following code passes the safety checking */

```
{ def f(x:Int) = g(x)
  def g(x: Int) = 10
  val a = 10
  f(10) }
```

/* The following code fails at the safety checking */

```
{ def f(x:Int) = g(x)
  val a = 10
  def g(x: Int) = 10
  f(10) }
```

Rewriting for Blocks

```
1: val t = 0
2: def f(x: Int) = t + g(x+1)
3: def g(y: Int) = y*y
4: val x = f(5)
```

```
5: val r = {
6:   val t = 10
7:   val s = f(5)
8:   s - t }
9: t + r
```

➢Evaluation with Environment

```
[],1 ~ [t=0],2 ~ […,f=(x)t+g(x)],3 ~ […,g=(y)y*y],4
~ […,x=36],5 ~ […]:[],6 ~ […]:[t=10],7
~ […]:[…,s=36],8 ~ […,r=26],9 ~ […],26
```

```
f(5): [t=0,f=…,g=…]:[x=5],t+g(x+1) ~ 0+g(6) ~ 36
   g(6): [t=0,f=…,g=…]:[y=6],y*y ~ 6*6 ~ 36
```

```
f(5): [t=0,f=…,g=…,x=36]:[x=5],t+g(x+1) ~ 0+g(6) ~ 36
   g(6): [t=0,f=…,g=…,x=36]:[y=6],y*y ~ 6*6 ~ 36
```

Example: def with no arguments

```
1: val t = 0
2: def x = t+t //essentially treated same as def x() = t+t
3: val r = {
4:    val t = 10
5:    x+t }
```

➢Evaluation with Environment

```
[],1 ~ [t=0],2 ~ […,x=t+t],3 ~ […]:[],4
~ […]:[t=10],5 ~ […,r=10],6

  5: [t=0,x=t+t],t+t ~ 0
```

Semi-colons and Parenthesis

- Block
  - Can write two definitions/expressions in a single line using ;
  - Can write one definition/expression in two lines using (), but can omit () when clear

```
// ok
val r = {
  val t = 10; val s = square(5); t +
  s }
// Not ok
val r = {
  val t = 10; val s = square(5); t
  + s }
// ok
val r = {
  val t = 10; val s = square(5); (t
  + s) }
```

Exercise: Writing Better Code using Blocks

➢Make the following code better

```
def isGoodEnough(guess: Double, x: Double) =
  guess*guess/x > 0.999 && guess*guess/x < 1.001

def improve(guess: Double, x: Double) =
  (guess + x/guess) / 2

def sqrtIter(guess: Double, x: Double): Double = {
  if (isGoodEnough(guess,x)) guess
  else sqrtIter(improve(guess,x),x)
}

def sqrt(x: Double) =
  sqrtIter(1, x)

sqrt(2)
```

Solution

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double = {
    if (isGoodEnough(guess,x)) guess
    else sqrtIter(improve(guess,x),x)
  }
  def isGoodEnough(guess: Double, x: Double) = {
    val ratio = guess * guess / x
    ratio > 0.999 && ratio < 1.001
  }
  def improve(guess: Double, x: Double) =
    (guess + x/guess) / 2

  sqrtIter(1, x)
}

sqrt(2)
```

Lazy Call-By-Value

Lazy call-by-value

- Lazy call-by-value
  - Use “lazy val” e.g. `lazy val x = e`
  - Evaluate the expression first time it is used, then bind the name to it

```
def f(c: Boolean, i: =>Int): Int = {
  lazy val iv = i
  if (c) 0
  else   iv * iv * iv
}

f(true, {println("ok"); 100+100+100+100})
f(false, {println("ok"); 100+100+100+100})
```

Tail Recursion & Tail Call

Recursion needs care

- Summation function
  - Write a summation function `sum` such that
    `sum(n) = 1+2+…+n`
  - Test
    `sum(10),sum(100),sum(1000),sum(10000),`
    `sum(100000), sum(1000000)`
  - What's wrong? (Think about evaluation)

Recursion: Try

```
def sum(n: Int): Int =
  if (n <= 0) 0 else (n+sum(n-1))
```

Recursion: Tail Recursion

```
import scala.annotation.tailrec

def sum(n: Int): Int = {
  @tailrec def sumItr(res: Int, m: Int): Int =
    if (m <= 0) res else sumItr(m+res,m-1)
  sumItr(0,n)
}
```

Mutual Recursion: Try

```
{
  def sum(acc: Int, n: Int): Int =
    if (n <= 0) acc else sum2(n + acc, n-1)

  def sum2(acc: Int, n: Int): Int =
    if (n <= 0) acc else sum(2*n + acc, n-1)

  sum(0, 20000)  // stack overflow
}
```

Mutual Recursion: Tail Call Optimization

```
import scala.util.control.TailCalls._

{
  def sum(acc: Int, n: Int): TailRec[Int] =
    if (n <= 0) done(acc) else tailcall(sum2(n + acc, n-1))

  def sum2(acc: Int, n: Int): TailRec[Int] =
    if (n <= 0) done(acc) else tailcall(sum(2*n + acc, n-1))

  sum(0, 20000).result
}
```

Higher-Order Functions

Functions as Values

- Functions
  - Functions are normal values of function types ($A_1,\ldots,A_n=>B$).
  - They can be copied, passed and returned.
  - Functions that take functions as arguments or return functions are called higher-order functions.
  - Higher-order functions increase code reusability.

Examples

```
def sumLinear(n: Int): Int =
  if (n <= 0) 0 else n + sumLinear(n-1)

def sumSquare(n: Int): Int =
  if (n <= 0) 0 else n*n + sumSquare(n-1)

def sumCubes(n: Int): Int =
  if (n <= 0) 0 else n*n*n + sumCubes(n-1)
```

Q: How to write reusable code?

Examples

```
def sum(f: Int=>Int, n: Int): Int =
  if (n <= 0) 0 else f(n) + sum(f, n-1)

def linear(n: Int) = n
def square(n: Int) = n * n
def cube(n: Int) = n * n * n

def sumLinear(n: Int) = sum(linear, n)
def sumSquare(n: Int) = sum(square, n)
def sumCubes(n: Int) = sum(cube, n)
```

Anonymous Functions

- Anonymous Functions
  - Syntax

    $(x_1: T_1,...,x_n:T_n) => e$

    or

    $(x_1,...,x_n) => e$

```
def sumLinear(n: Int) = sum((x:Int)=>x, n)
def sumSquare(n: Int) = sum((x:Int)=>x*x, n)
def sumCubes(n: Int) = sum((x:Int)=>x*x*x, n)
```

Or simply

```
def sumLinear(n: Int) = sum((x)=>x, n)
def sumSquare(n: Int) = sum((x)=>x*x, n)
def sumCubes(n: Int) = sum((x)=>x*x*x, n)
```

Exercise

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f, a+1, b) else 0

def product(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) * product(f, a+1, b) else 1
```

DRY (Do not Repeat Yourself) using a higher-order function, called “mapReduce”.

Exercise

```
def mapReduce(reduce: (Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) reduce(f(a),mapReduce(reduce,inival,f,a+1,b))
  else inival
}

def sum(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x+y,0,f,a,b)

def product(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x*y,1,f,a,b)
```

Closures for functional values

```
1: val t = 0
2: val f : Int=>Int = {        // Note: What if using "def f" ?
3:     val t = 10
4:     def g(x: Int) : Int = x + t
5:     g _ }
6: f(20)
```

* Try: Evaluation without Closures

```
[],1 ~ [t=0],2 ~ […]:[],3 ~ […]:[t=10],4
~ […]:[…,g=(x)x+t],5 ~ [t=0,f=(x)x+t],6 ~ […],20
  6: [t=0,f=(x)x+t]:[x=20],x+t ~ 20+0 ~ 20
```

* Evaluation with Closures

```
E1[],1 ~ E1[t=0],2 ~ E1[…]:E2[],3 ~ E1[…]:E2[t=10],4
~ E1[…]:E2[t=10,g=(x)x+t],5
~ E1[t=0,f=((x)x+t,E2)],6 ~ E1[…],30
  6: E1[…]:E2[…]:E3[x=20],x+t ~ 20+10 ~ 30
```

(Parameterized) expression vs. (closure) value

- Functions defined using "def" are not values but parameterized expressions.
- Parameterized expression `f` can be converted to a value `(f _)`.
- The compiler often infers and inserts missing conversions automatically.
- Anonymous functions are values.
- Anonymous functions can be seen as syntactic sugar:

  ```
  (x:T)=>e
  ```

  is equivalent to

  ```
  { def noname(x:T) = e; (noname _) }
  ```

  where `noname` is not used in `e`.
- One can even write a recursive anonymous function in this way.
- Q: what's the difference between param. exps and function values?
  A: function values are "closures" (ie, param. exp. + env.)
- Q: how to implement call-by-name?
  A: The argument expression is converted to a closure.

Example: call by name with closures

```
1: val t = 0
2: def f(x: =>Int) = t + x
3: val r = {
4:   val t = 10
5:   f(t*t) }                    // t*t is treated as ()=>t*t
```

➢Evaluation with Closures

```
E1[],1 ~ E1[t=0],2 ~ E1[t=0,f=(x:=>Int)t+x],3
~ E1[…]:E2[],4 ~ E1[…]:E2[t=10],5 ~ E1[…,r=100],6

  5: E1[…]:E3[x=(t*t,E2)],t+x
     ~ 0 + x ~ 0 + 100 ~ 100
  x: E1[…]:E2[…],t*t ~ 10*10 ~ 100
```

Currying

Motivation

```
def sum(f: Int=>Int, a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f, a+1, b) else 0
def sumLinear(a: Int, b: Int) = sum((n)=>n, a, b)
def sumSquare(a: Int, b: Int) = sum((n)=>n*n, a, b)
def sumCubes(a: Int, b: Int) = sum((n)=>n*n*n, a, b)
```

We want the following. How?

```
def sumLinear = sum(linear)
def sumSquare = sum(square)
def sumCubes = sum(cube)
```

Solution

```
def sum(f: Int=>Int): (Int,Int)=>Int = {
  def sumF(a: Int, b: Int): Int =
    if (a <= b) f(a) + sumF(a+1, b) else 0
  sumF  // sumF _
}

def sumLinear = sum((n)=>n)
def sumSquare = sum((n)=>n*n)
def sumCubes = sum((n)=>n*n*n)
```

Benefits

```
def sumLinear = sum((n)=>n)
def sumSquare = sum((n)=>n*n)
def sumCubes = sum((n)=>n*n*n)

sumSquare(3,10) + sumCubes(5,20)
```

We don't need to define the wrapper functions.

```
sum((n)=>n*n)(3,10) + sum((n)=>n*n*n)(5,20)
```

Multiple Parameter List

```
def sum(f: Int=>Int): (Int,Int)=>Int = {
  def sumF(a: Int, b: Int): Int =
    if (a <= b) f(a) + sumF(a+1, b) else 0
  sumF _
}
```

Or simply:

```
def sum(f: Int=>Int)(a: Int, b: Int): Int =
  if (a <= b) f(a) + sum(f)(a+1, b) else 0
```

Note that `sum(f)` is just a parameterized expression.
`(sum(f) _)` creates a closure like `(sumF _)`

The following code is slightly inefficient. Think about why.

```
def sum(f: Int=>Int): (Int,Int)=>Int =
  (a,b) => if (a <= b) f(a) + sum(f)(a+1, b) else 0
```

Comparison between Param. Exp vs. Closure

```
{
  def sum(f: Int=>Int)(a: Int, b: Int): Int =
    if (a <= b) f(a) + sum(f)(a+1, b) else 0
  def sumLinear = sum((n)=>n) _ // sum((n)=>n) incorrect

  sumLinear(0,100)
}
{
  def sum(f: Int => Int): (Int, Int) => Int = {
    def sumF(a: Int, b: Int): Int =
      if (a <= b) f(a) + sumF(a + 1, b) else 0
    sumF _
  }
  def sumLinear = sum((n)=>n) // sum((n)=>n) _ incorrect

  sumLinear(0,100)
}
```

Currying and Uncurrying

- A function of type

$$(T_1, T_2, \ldots, T_n) \Rightarrow T$$

can be turned into that of type

$$T_1 \Rightarrow (T_2 \Rightarrow (\ldots \Rightarrow (T_n \Rightarrow T) \ldots))$$

- This is called “currying” named after Haskell Brooks Curry.
- The opposite direction is called “uncurrying”.

Currying using Anonymous Functions

```
def foo(x: Int, y: Int, z: Int)(a: Int, b: Int) =
  x + y + z + a + b

val f1 = (x: Int, z: Int, b: Int) => foo(x,1,z)(2,b)

val f2 = foo(_:Int,1,_:Int)(2, _:Int)

val f3 = (x: Int, z: Int) => ((b: Int) => foo(x,1,z)(2,b))

f1(1,2,3)
f2(1,2,3)
f3(1,2)(3)
```

Exercise

Curry the mapReduce function.

```
def mapReduce(reduce: (Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) reduce(f(a),mapReduce(reduce,inival,f,a+1,b))
  else inival
}

def sum(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x+y,0,f,a,b)

def product(f: Int=>Int, a: Int, b: Int): Int =
  mapReduce((x,y)=>x*y,1,f,a,b)
```

Solution

```
def mapReduce(reduce:(Int,Int)=>Int,inival: Int)
              (f: Int=>Int) (a: Int, b: Int): Int = {
  if (a <= b) reduce(f(a),mapReduce(reduce,inival)(f)(a+1,b))
  else inival
}

// need to make a closure since mapReduce is param. code.
def sum = mapReduce((x,y)=>x+y,0) _

// val is better than def. Think about why.
val product = mapReduce((x,y)=>x*y,1) _
```

Exceptions

Exception & Handling

```
class factRangeException(val arg: Int) extends Exception

def fact(n : Int): Int =
  if (n < 0) throw new factRangeException(n)
  else if (n == 0) 1
  else n * fact(n-1)

def foo(n: Int) = fact(n + 10)

try {
  println (fact(3))
  println (foo(-100))
} catch {
  case e : factRangeException => {
    println("fact range error: " + e.arg)
  }
}
```

Datatypes

Types so far

Types have introduction operations and elimination ones.

- Introduction: how to construct elements of the type
- Elimination: how to use elements of the type

➢Primitive types

- Int, Boolean, Double, String, …
- Intro for Int: …,-2,-1,0,1,2,
- Elim for Int: +,-,*,/,<,<=,…

➢Function types

- Int=>Int, (Int=>Int)=>(Int=>Int), …
- Intro: (x:T)=>e
- Elim: f(v)

Tuples

- Tuples

  Intro:

  - (1,2,3) : (Int, Int, Int)
  - (1,“a”) : (Int, String)

  Elim:

  - (1, “a”,10)._1 = 1
  - (1, “a”,10)._2 = “a”
  - (1, “a”,10)._3 = 10

  Only up to length 22

Structural Types (a.k.a. Record Types): Examples

```
def bar (x: Int) = x+1
val foo = new
//object foo
{ val a = 1+2
  def b = a + 1
  def f(x: Int) = b + x
  def f(x: String) = "hello" + x
  val g : Int => Int = bar _
}
foo.b; foo.f(3); foo.f("gil")
foo.f _ : Int=>Int
def g(x: {val a: Int; def b: Int;
          def f(x:Int): Int; def f(x:String): String;
          val g: Int => Int}) =
  x.f(3)
g(foo)
```

Type Alias

```
val gn = 0
object foo {
  val a = 3
  def b = a + 1
  def f(x: Int) = b + x + gn
}

foo.f(3)

type Foo = {val a: Int; def b: Int; def f(x:Int):Int}

def g(x: Foo) = {
  val gn = 10
  x.f(3)
}

g(foo)
```

Structural Types: Evaluation

```
1: def bar (x: Int) = x+1
2: val foo = new //or, object foo
3: { val a = 2 + 1
4:   def f(x: Int) = a + x
5:   val g : Int => Int = bar _
6: }
7: val b = foo.f(1)
8: foo.g(b)
```

➢Evaluation with Closures

```
E1[],1 ~ E1[bar=(x)x+1],2 ~ E1[…]:E2[],3 ~
E1[…]:E2[a=3],4 ~
E1[…]:E2[a=3,f=(x)a+x],5 ~
E1[…]:E2[a=3,f=(x)a+x,g=((x)x+1,E1)],6 ~
E1[bar=(x)x+1,foo=(E2)],7 ~
E1[bar=(x)x+1,foo=(E2),b=4],8 ~ 5
7: E1:E2:E3[x=1],a+x ~ 3+1 ~ 4
8: E1:E4[x=4],x+1 ~ 4+1 ~ 5
```

Algebraic Datatypes

- Ideas
  - T = C of T * … * T
    | C of T * … * T
    | …
    | C of T * … * T

  - E.g.
    Attr = Name of String
    | Age of Int
    | DOB of Int * Int * Int
    | Height of Double

    Intro:
    Name(“Chulsoo Kim”), Name(“Younghee Lee”), Age(16),
    DOB(2000,3,10), Height(171.5), …

Algebraic Datatypes: Recursion

- Recursive ADT
  - E.g.

```
IList = INil
      | ICons of Int * IList
```

Intro:

```
INil(),
ICons(3, INil()),
ICons(2, ICons(3, INil())),
ICons(1, ICons(2, ICons(3, INil()))),
…
```

Algebraic Datatypes In Scala

- Attr

```
sealed abstract class Attr
case class Name(name: String) extends Attr
case class Age(age: Int) extends Attr
case class DOB(year: Int, month: Int, day: Int) extends Attr
case class Height(height: Double) extends Attr

val a : Attr = Name("Chulsoo Kim")
val b : Attr = DOB(2000,3,10)
```

- IList

```
sealed abstract class IList
case class INil() extends IList
case class ICons(hd: Int, tl: IList) extends IList

val x : IList = ICons(2, ICons(1, INil()))
def gen(n: Int) : IList =
  if (n <= 0) INil()
  else ICons(n, gen(n-1))
```

Exercise

IOption = INone
| ISome of Int

BTree = Leaf
| Node of Int * BTree * BTree

```
sealed abstract class IList
case class INil() extends IList
case class ICons(hd: Int, tl: IList) extends IList

def x : IList = ICons(2, ICons(1, INil()))
```

Solution

```
sealed abstract class IOption
case class INone() extends IOption
case class ISome(some: Int) extends IOption

sealed abstract class BTree
case class Leaf() extends BTree
case class Node(value: Int, left: BTree, right: BTree)
  extends BTree
```

Pattern Matching

- Pattern Matching
  - A way to use algebraic datatypes

```
e match {
  case C1(…) => e1 : T
   …
  case Cn(…) => en : T
} : T
```

Pattern Matching: An Example

```
def length(xs: IList) : Int =
  xs match {
    case /Nil/() => 0
    case /Cons(x, tl) => 1 + length(tl)
  }

length(x)
```

Advanced Pattern Matching

- Advanced Pattern Matching

```
e match {
  case P1 => e1
   …
  case Pn => en
}
```

- One can combine constructors and use _ and | in a pattern.
  (E.g) case ICons(x, INil()) | ICons(x, ICons(_, INil())) => …
- The given value `e` is matched against the first pattern `P1`.
  If succeeds, evaluate `e1`.
  If fails, `e` is matched against `P2`.
  If succeeds, evaluate `e2`.
  If fails, …
- The compiler checks exhaustiveness (ie, whether there is a missing case).

Advanced Pattern Matching: An Example

```
def secondElmt(xs: IList) : IOption =
  xs match {
    case INil() | ICons(_,INil()) => INone()
    case ICons(_, ICons(x, _)) => ISome(x)
  }
```

Vs.

```
def secondElmt2(xs: IList) : IOption =
  xs match {
    case INil() | ICons(_,INil()) => INone()
    case ICons(_, ICons(x, INil())) => ISome(x)
    case _ => INone()
  }
```

Vs.

```
def secondElmt2(xs: IList) : IOption =
  xs match {
    case ICons(_, ICons(x, INil())) => ISome(x)
    case _ => INone()  }
```

Pattern Matching on Int

```
def factorial(n: Int) : Int =
  n match {
    case 0 => 1
    case _ => n * factorial(n-1)
  }

def fib(n: Int) : Int =
  n match {
    case 0 | 1 => 1
    case _ => fib(n-1) + fib(n-2)
  }
```

Pattern Matching with If

```
def f(n: Int) : Int =
  n match {
    case 0 | 1 => 1
    case _ if (n <= 5) => 2
    case _ => 3
  }


def f(t: BTree) : Int =
  t match {
    case Leaf() => 0
    case Node(n,_,_) if (n <= 10) => 1
    case Node(_,_,_) => 2
  }
```

Exercise

Write a function “find(t: BTree, x: Int) : Boolean” that checks whether x is in t.

```
sealed abstract class BTree
case class Leaf() extends BTree
case class Node(value: Int, left: BTree, right: BTree)
  extends BTree
```

Solution

```
def find(t: BTree, i: Int) : Boolean =
  t match {
    case Leaf() => false
    case Node(n,lt,rt) =>
      if (i == n) true
      else find(lt, i) || find(rt, i)
  }
def find(t: BTree, i: Int) : Boolean = {
 t match {
   case Leaf() => false
   case Node(v, _, _) if v == i => true
   case Node(_, lt, rt) => find(lt, i) || find(rt, i)
 }
}
def t: BTree = Node(5,Node(4,Node(2,Leaf(),Leaf()),Leaf()),
  Node(7,Node(6,Leaf(),Leaf()),Leaf()))
find(t,7),find(t,1)
```

Solution with Tail Recursion

```
sealed abstract class BTList
case class BTNil() extends BTList
case class BTCons(hd: BTree, tl: BTList) extends BTList
def find(t: BTree, x: Int) : Boolean = {
  def findIter(ts: BTList) : Boolean =
    ts match {
      case BTNil() => false
      case BTCons(Leaf(), tl) => findIter(tl)
      case BTCons(Node(v, _, _), _) if v == x => true
      case BTCons(Node(_,l,r), tl) =>
        findIter(BTCons(l, BTCons(r, tl)))
    }
  findIter(BTCons(t, BTNil()))
}
def genTree(v: Int, n: Int) : BTree = {
  def genTreeIter(t: BTree, m : Int) : BTree =
    if (m == 0) t
    else genTreeIter(Node(v, t, Leaf()), m-1)
  genTreeIter(Leaf(), n)
}
find(genTree(0,10000), 1)
```

Type Checking & Inference (Concept)

What Are Types For?

- Typed Programming

```
def id1(x: Int): Int = x
def id2(x: Double): Double = x
```

  - At run time, type information is erased (ie, `id1` = `id2`)

- Untyped Programming

```
def id(x) = x
```

  - Do not care about types at compile time.
  - But, many such languages check types at run time paying cost.
  - Without run-time type check, errors can be badly propagated.

- What is compile-time type checking for?
  - Can detect type errors at compile time.
  - Increase Readability (Give a good abstraction).
  - Soundness: Well-typed programs raise no type errors at run time.

Type Checking and Inference

- Type Checking

$$x_1:T_1,\ x_2:T_2,\ \ldots,\ x_n:T_n \vdash e : T$$

  - `def f(x: Boolean): Boolean = x > 3`
    => Type error
  - `def f(x: Int): Boolean = x > 3`
    => OK. `f: (x: Int)Boolean`

- Type Inference

$$x_1:T_1,\ x_2:T_2,\ \ldots,\ x_n:T_n \vdash e : ?$$

  - `def f(x: Int) = x > 3`
    => OK by type inference. `f: (x: Int)Boolean`
  - Too much type inference is not good. Why?

You can learn how type checking & inference work in
4190.310 Programming Languages

Parametric Polymorphism

Parametric Polymorphism: Functions

- Problem

```
def id1(x: Int): Int = x
def id2(x: Double): Double = x
```

  - Can we avoid copy and paste?
  - Polymorphism to the rescue!

- Parametric Polymorphism (a.k.a. For-all Types)

```
def id[A](x: A) : A = x
```

  - The type of `id` is `[A](val x:A)=>A`
  - `id` is a parametric expression.
  - `id[T] _` is a value of type `T=>T` for any type `T`.
  - Function types do not directly support polymorphism.

  [We will learn other kinds of polymorphism later.]

Examples

```
def id[A](x:A) = x
id(3)
id("abc")

def applyn[A](f: A => A, n: Int, x: A): A =
  n match {
    case 0 => x
    case _ => f(applyn(f, n - 1, x))
  }
applyn((x:Int)=>x+1,100,3)
applyn((x:String)=>x+"!", 10, "gil")
applyn(id[String], 10, "hur")

def foo[A,B](f: A=>A, x: (A,B)) : (A,B) =
  (applyn[A](f, 10, x._1), x._2)

foo[String,Int]((x:String)=>x+"!",("abc",10))
```

Full Polymorphism using Scala's trick

```
type Applyn = {def apply[A](f: A=>A, n: Int, x: A): A}
val applyn = new {          // object applyn {
  def apply[A](f: A=>A, n: Int, x: A): A =
    n match {
      case 0 => x
      case _ => f(apply(f, n-1, x))
    }
}
// applyn.apply[String]((x:String)=>x+"!", 10, "gil")
applyn((x:String)=>x+"!", 10, "gil")


def foo(f: Applyn): String = {
  val a:String = f[String]((x:String)=> x + "!",10,"gil")
  val b:Int = f[Int]((x:Int)=> x + 2, 10, 5)
  a + b.toString()
}
foo(applyn)
```

Summary

- Main points
  - Definitions allow type parameters at the beginning.
    (E.g.) `def id[A](x: A) : A = x`
  - Values does not allow type parameters.
    (E.g.) `[A](A=>A)` is not a valid value type.
  - Record types allow type parameters in their "def" fields.
    (E.g.) `{def apply[A](x: A): A}` is a valid value type.
- Minor points
  - The compiler automatically inserts "`.apply`" when a record is applied to arguments.
  - The function types are encoded as "class types", which we will learn in Part II.

Parametric Polymorphism: Datatypes

```
sealed abstract class MyOption[A]
case class MyNone[A]() extends MyOption[A]
case class MySome[A](some: A) extends MyOption[A]

sealed abstract class MyList[A]
case class MyNil[A]() extends MyList[A]
case class MyCons[A](hd: A, tl: MyList[A]) extends MyList[A]

sealed abstract class BTree[A]
case class Leaf[A]() extends BTree[A]
case class Node[A](value: A, left: BTree[A], right: BTree[A])
extends BTree[A]

def x: MyList[Int] = MyCons(3, MyNil())
def y: MyList[String] = MyCons("abc", MyNil())
```

Revisit: Solution with Tail Recursion

```
def find[A](t: BTree[A], x: A) : Boolean = {
  def findIter(ts: MyList[BTree[A]]) : Boolean =
    ts match {
      case MyNil() => false
      case MyCons(Leaf(), tl) => findIter(tl)
      case MyCons(Node(v, _, _), _) if v == x => true
      case MyCons(Node(_,l,r), tl) =>
        findIter(MyCons(l, MyCons(r, tl)))    }
  findIter(MyCons(t, MyNil()))
}

def genTree(v: Int, n: Int) : BTree[Int] = {
  def genTreeIter(t: BTree[Int], m : Int) : BTree[Int] =
    if (m == 0) t
    else genTreeIter(Node(v, t, Leaf()), m-1)
  genTreeIter(Leaf(), n)
}

find(genTree(0,10000), 1)
```

Exercise

BSTree[A] = Leaf
| Node of Int * A * BSTree[A] * BSTree[A]

```
def lookup[A](t: BSTree[A], k: Int) : MyOption[A] =
  ???

def t : BSTree[String] =
  Node(5,"My5",
    Node(4,"My4",Node(2,"My2",Leaf(),Leaf()),Leaf()),
    Node(7,"My7",Node(6,"My6",Leaf(),Leaf()),Leaf()))

lookup(t, 7)
lookup(t, 3)
```

Solution

```
sealed abstract class BSTree[A]
case class Leaf[A]() extends BSTree[A]
case class Node[A](key: Int, value: A, left: BSTree[A], right:
BSTree[A]) extends BSTree[A]
def lookup[A](t: BSTree[A], key: Int) : MyOption[A] =
  t match {
    case Leaf() => MyNone()
    case Node(k,v,lt,rt) =>
      k match {
        case _ if key == k => MySome(v)
        case _ if key < k => lookup(lt,key)
        case _ => lookup(rt, key)
      }
  }
def t : BSTree[String] =
  Node(5,"My5",
    Node(4,"My4",Node(2,"My2",Leaf(),Leaf()),Leaf()),
    Node(7,"My7",Node(6,"My6",Leaf(),Leaf()),Leaf()))
lookup(t, 7)
lookup(t, 3)
```

A Better Way

```
sealed abstract class BTree[A]
case class Leaf[A]() extends BTree[A]
case class Node[A](value: A, left: BTree[A], right: BTree[A])
  extends BTree[A]

type BSTree[A] = BTree[(Int,A)]

def lookup[A](t: BSTree[A], k: Int) : MyOption[A] =
  ???

def t : BSTree[String] =
  Node((5,"My5"),
    Node((4,"My4"),Node((2,"My2"),Leaf(),Leaf()),Leaf()),
    Node((7,"My7"),Node((6,"My6"),Leaf(),Leaf()),Leaf()))

lookup(t, 7)
```

Solution

```
type BSTree[A] = BTree[(Int,A)]

def lookup[A](t: BSTree[A], key: Int) : MyOption[A] =
  t match {
    case Leaf() => MyNone()
    case Node((k,v),lt,rt) =>
      k match {
        case _ if key == k => MySome(v)
        case _ if key < k => lookup(lt,key)
        case _ => lookup(rt, key)
      }
  }

def t : BSTree[String] =
  Node((5,"My5"),
    Node((4,"My4"),Node((2,"My2"),Leaf(),Leaf()),Leaf()),
    Node((7,"My7"),Node((6,"My6"),Leaf(),Leaf()),Leaf()))

lookup(t, 7)
lookup(t, 3)
```

Polymorphic Option (Library)

- Option[T]

  Intro:
  - None
  - Some(x)
  - Library functions

  Elim:
  - Pattern matching
  - Library functions

  Some(3) : Option[Int]
  Some("abc"): Option[String]
  None: Option[Int]
  None: Option[String]

Polymorphic List (Library)

- List[T]

  Intro:
  - Nil
  - x :: L
  - Library functions

  Elim:
  - Pattern matching
  - Library functions

  “abc”::Nil : List[String]

  List(1,3,4,2,5) = 1::3::4::2::5::Nil : List[Int]

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"
}
```

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

Note that we have

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

Sub Types to the Rescue!

```
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.

Two Kinds of Sub Types

- Structural Sub Types
  - The system implicitly determines the sub type relation by the structures of data types.
  - Structurally equivalent types are treated the same.

- Nominal Sub Types
  - 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

```
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

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

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)
```

use: foo.a foo.b foo.f

- 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))))
```

Note on Null value

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

- This value is needed to construct disjoint union types using classes in Java, which, however, is not as elegant and type safe as algebraic data types (ADTs):
  - Such disjoint union types can contain junk values (not elegant).
  - Null-point exception can be raised at run time (not type safe).

- For this reason, it is discouraged to use `null` in Scala although Scala supports `null` for compatibility with Java.

- Instead, it is encouraged to use ADTs, which themselves are classes and thus take advantages of both ADT and class.

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

object MyList // val MyList = new
{ def apply[A](v: A, nxt: Option[MyList[A]]) =
    new MyList(v,nxt)
}
type YourList[A] = Option[MyList[A]]

val t0 = None
val t1 = Some(new MyList(3,Some (new MyList(4,None))))
val t2 = Some(MyList(3,Some (MyList(4,None))))
```

Exercise

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

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

Solution

```
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))
```

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 1

```
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) {
  override def f(z: Int) = b + z
  // or, override def f(z: Int) = super.f(z) * 2
  val c : Int = f(x) + b
}
(new gee_type(30)).c
```

Overriding 2

```
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
}

(new gee_type(30)).f(10)
```

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): Nothing = ??? //Yes, ret:  sub type
```

Example: My List

```
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()))
```

Example: 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]) = ???
```

Case Class

```
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

```
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
}
```

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)
```

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))
```

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)) }
```

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

```
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] {
  override 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 Abstract Types

Using an Abstract 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 IterableHE[A](eq: (A,A) => Boolean)
  extends Iterable[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)
```

Code Reuse without Inheritance

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

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)
```

Using Interface

- OOP-style Interface

```
abstract class PrimesSig {
  // Problem: need an instance of PrimesSig to make a new instance
  def makeNew : PrimesSig
  def prime : Int
  def getNext : PrimesSig
}
```

- Type class-style Specification

```
abstract class PrimesSig[P] {
  def makeNew : P
  def prime(p: P) : Int
  def getNext(p: P) : P
}
def nthPrime[P](n: Int)(implicit tc: PrimesSig[P]): Int = {
  def go(p: P, k: Int): Int =
    if (k <= 1) tc.prime(p) else go(tc.getNext(p), k - 1)
  if (n == 0) 2 else go(tc.makeNew, n)
}
```

Implementation

```
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)
}
class PrimesImpl extends PrimesSig[Primes] {
  def makeNew = new Primes
  def prime(p: Primes) = p.prime
  def getNext(p: Primes) = p.getNext
}
implicit val primesTC = new PrimesImpl

nthPrime(10000) // (primesTC)
```

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
  - Among the ancestors of a class, for the same class,
    - A constructor with arguments can appear at most once
    - A 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) = 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)
```

Without Using Inheritance

```
class ListIterDict[K,V]
    (eq: (K,K)=>Boolean, list: List[(K,V)])
    extends Iter[(K,V)] with Dict[K,V]
{
  val listIter = new ListIter(list)
  def getValue = listIter.getValue
  def getNext = listIter.getNext

  def add(k:K,v: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) }
}
```

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 go(c: Iter[A]): C = c.getValue match {
      case None => ival
      case Some(v) => combine(f(v), go(c.getNext))
    }
    go(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

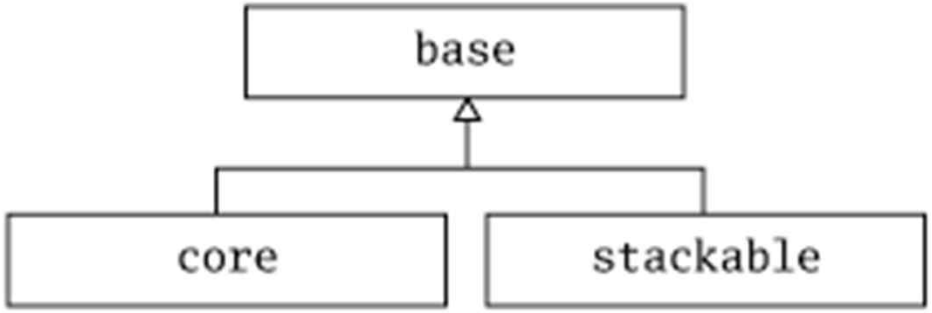
$$\frac{}{\ldots \text{ with } T \ldots <: \ldots \ldots}$$

- 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: Int): Stack[A]
}
```

IntStack: Core

➢**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: 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)
  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/lectures/scala/02-classes.html

PART 3
Type Classes for Interfaces

Interfaces over Parameter Types

Subtype Polymorphism

```
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.

Type Classes

Completely Separating Ord from Int

```
trait Ord[A] {
  def cmp(me: A, you: A): Int

  def ===(me: A, you: A): Boolean = cmp(me,you) == 0
  def <  (me: A, you: A): Boolean = cmp(me,you) < 0
  def >  (me: A, you: A): Boolean = cmp(me,you) > 0
  def <= (me: A, you: A): Boolean = cmp(me,you) <= 0
  def >= (me: A, you: A): Boolean = cmp(me,you) >= 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 val intOrd : Ord[Int] = new Ord[Int] {
  def cmp(me: Int, you: Int) = me - you }

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
```

Bag Example

```
class Bag[A] protected (val toList: List[A])(implicit ord: Ord[A])
{ def this()(implicit ord: Ord[A]) = this(Nil)(ord)

  def add(x: A) : Bag[A] = {
    def go(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 :: go(rest)
      }
    new Bag(go(toList))
  }
}

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

Implicitly

➢Definition

```
def implicitly[A](implicit a: A) : A = a
```

➢Example

```
class Bag[A] protected (val toList: List[A])(implicit ord: Ord[A])
{ def this()(implicit ord: Ord[A]) = this(Nil)(ord)
  …
        case e :: _ if (ord.===(x,e)) => elmts
  …
}
```

is equivalent to

```
class Bag[A : Ord] protected (val toList: List[A])
{ def this() = this(Nil)
  …
        case e :: _ if (implicitly[Ord[A]].===(x,e)) => elmts
  …
}
```

Bag Example Implicitly

```
class Bag[A : Ord] protected (val toList: List[A])
{ def this() = this(Nil)

  def add(x: A) : Bag[A] = {
    def go(elmts: List[A]) : List[A] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (implicitly[Ord[A]].<(x,e)) => x :: elmts
        case e :: _ if (implicitly[Ord[A]].===(x,e)) => elmts
        case e :: rest => e :: go(rest)
      }
    new Bag(go(toList))
  }
}

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

Bootstrapping Implicits

```
// lexicographic order
implicit def tupOrd[A, B](implicit ordA: Ord[A], ordB: Ord[B])
  : Ord[(A, B)] =
  new Ord[(A, B)] {
    def cmp(me: (A, B), you: (A, B)) : Int = {
      val c1 = ordA.cmp(me._1, you._1)
      if (c1 != 0) c1
      else { ordB.cmp(me._2, you._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

```
val intOrdRev : Ord[Int] =
  new Ord[Int] { def cmp(me: Int, you: Int) = you - me }

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

Type Classes With Multiple Parameters

Interfaces I

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

// trait Iterable[A] {
//   def iter : Iter[A]
// }
trait Iterable[R,A] {
  type Itr
  def iterIF: Iter[Itr, A]
  def iter(a: R): Itr
}
```

Programs for Testing: use Iter, Iterable

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

def printElements[I,A](xs: I)(implicit IT: Iter[I,A]) : Unit =
  IT.getValue(xs) match {
    case None =>
    case Some(n) => {println(n); printElements(IT.getNext(xs))}}

def sumElements2[R](xs: R)(implicit ITR: Iterable[R,Int]) =
  sumElements(ITR.iter(xs))(ITR.iterIF)
//sumElements[I](ITR.iter(xs))(ITR.iterIF)

def printElements2[R,A](xs: R)(implicit ITR: Iterable[R,A]) =
  printElements(ITR.iter(xs))(ITR.iterIF)
//printElements[I,A](ITR.iter(xs))(ITR.iterIF)
```

Interfaces II

```
trait ListIF[L,A] {
  def empty : L
  def head(l: L) : Option[A]
  def tail(l: L) : L
  def cons(a: A, l: L) : L
  def append(l1: L, l2: L) : L
}

trait TreeIF[T,A] {
  def empty : T
  def node(a: A, l: T, r: T) : T
  def head(t: T) : Option[A]
  def left(t: T) : T
  def right(t: T) : T
}
```

Programs for Testing: use All

```
def testList[L](implicit LI: ListIF[L,Int], IT: Iter[L,Int]) {
  val l = LI.cons(3, LI.cons(5, LI.cons(2, LI.cons(1, LI.empty))))
  println(sumElements(l)) //sumElements(l)(listIter[Int])
  printElements(l) //printElements(l)(listIter[Int])
}

def testTree[T](implicit TI: TreeIF[T,Int],
                         ITR: Iterable[T,Int]) {
  val t: T = TI.node(3, TI.node(4, TI.empty, TI.empty),
                        TI.node(2, TI.empty, TI.empty))
  println(sumElements2(t))
  printElements2(t)
}
```

List: provide Iter, ListIF

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

implicit def listIF[A] : ListIF[List[A],A] =
  new ListIF[List[A],A] {
    def empty: List[A] = Nil
    def head(l: List[A]) = l.headOption
    def tail(l: List[A]) = l.tail
    def cons(a: A, l: List[A]) = a :: l
    def append(l1: List[A], l2: List[A]) = l1 ::: l2
  }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
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]


implicit def treeIF[A] : TreeIF[MyTree[A],A] =
  new TreeIF[MyTree[A],A] {
    def empty = Empty()
    def node(a: A, l: MyTree[A], r: MyTree[A]) = Node(a,l,r)
    def head(t: MyTree[A]) = t match {
      case Empty() => None
      case Node(v,_,_) => Some(v)    }
    def left(t: MyTree[A]) = t match {
      case Empty() => t
      case Node(_,lt,_) => lt    }
    def right(t: MyTree[A]) = t match {
      case Empty() => t
      case Node(_,_,rt) => rt    }  }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
def treeIterable[L,A](implicit IF: ListIF[L,A], IT: Iter[L,A])
    : Iterable[MyTree[A], A]
 =
  new Iterable[MyTree[A], A] {
    type Itr = L
    def iter(a: MyTree[A]): L = a match {
      case Empty() => IF.empty
      case Node(v, left, right) =>
        IF.cons (v, IF.append(iter(left), iter(right)))    }
    val iterIF = IT }

implicit def treeIterableList[A] = treeIterable[List[A],A]
```

Linking Modules

```
testList[List[Int]]
```

```
testTree[MyTree[Int]]
```

Refined Interfaces

```
trait ListProdIF[L,A] {
  def empty : L
  def cons(a: A, l: L) : L
  def append(l1: L, l2: L) : L
}

def treeIterable[L,A]
  (implicit IF: ListProdIF[L,A], IT: Iter[L,A]) =
  new Iterable[MyTree[A], A] {
    type Itr = L
    def iter(a: MyTree[A]): L = a match {
      case Empty() => IF.empty
      case Node(v, left, right) =>
        IF.cons (v, IF.append(iter(left), iter(right)))    }
    val iterIF = IT }
implicit def treeIterableList[A] = treeIterable[List[A],A]
```

Linking Modules

```
implicit def List2ListProdIF[L,A]
    (implicit lF: ListIF[L,A]) : ListProdIF[L,A] =
  new ListProdIF[L,A] {
    def empty = lF.empty
    def cons(a: A, l: L) = lF.cons(a,l)
    def append(l1: L, l2: L) = lF.append(l1, l2)
  }

testList[List[Int]]

testTree[MyTree[Int]]
```

Iter being Iterable

```
implicit def iterIterable[I,A](implicit IT: Iter[I,A])
             : Iterable[I,A] =
  new Iterable[I, A] {
    type Itr = I
    val iterIF = IT
    def iter(a: I) = a
  }

val l = List(3,5,2,1)
sumElements2(l) //sumElements2(iterIterable(listIter[Int]))
printElements2(l) //printElements2(iterIterable(listIter[Int]))
```

Higher-kind Type Classes

Interfaces I

```
import scala.language.higherKinds
//trait Iter[I,A] {
//  def getValue(a: I): Option[A]
//  del getNext(a: I): I }
trait Iter[I[_]] {
  def getValue[A](a: I[A]) : Option[A]
  def getNext[A](a: I[A]) : I[A]
}
//trait Iterable[R,A] {
//  type Itr
//  def iterIF: Iter[Itr, A]
//  def iter(a: R): Itr
//}
trait Iterable[R[_]] {
  type Itr[_]
  def iter[A](a: R[A]): Itr[A]
  def iterIF: Iter[Itr]
}
```

Programs for Testing: use Iter, Iterable

```
def sumElements[I[_]](xs: I[Int])(implicit IT: Iter[I]): Int = {
  IT.getValue(xs) match {
    case None => 0
    case Some(n) => n + sumElements(IT.getNext(xs)) }
}
def printElements[I[_],A](xs:I[A])(implicit IT: Iter[I]): Unit = {
  IT.getValue(xs) match {
    case None =>
    case Some(n) => {println(n); printElements(IT.getNext(xs))}}
}
def sumElements2[R[_]](xs: R[Int])(implicit ITR: Iterable[R]) =
  sumElements(ITR.iter(xs))(ITR.iterIF)

def printElements2[R[_],A](xs: R[A])(implicit ITR: Iterable[R]) =
  printElements(ITR.iter(xs))(ITR.iterIF)
```

Interfaces II

```
trait ListIF[L[_]] {
  def empty[A] : L[A]
  def head[A](l: L[A]) : Option[A]
  def tail[A](l: L[A]) : L[A]
  def cons[A](a: A, l: L[A]) : L[A]
  def append[A](l1: L[A], l2: L[A]) : L[A]
}

trait TreeIF[T[_]] {
  def empty[A] : T[A]
  def node[A](a: A, l: T[A], r: T[A]) : T[A]
  def head[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 LI: ListIF[L], IT: Iter[L]) {
  val l = LI.cons(3, LI.cons(5, LI.cons(2, LI.cons(1, LI.empty))))
  println(sumElements(l)) //sumElements[L](l)(IT)
  printElements(l) //printElements[L](l)(IT)
}

def testTree[T[_]](implicit TI: TreeIF[T], ITR: Iterable[T]) {
  val t = TI.node(3, TI.node(4, TI.empty, TI.empty),
                     TI.node(2, TI.empty, TI.empty))
  println(sumElements2(t))
  printElements2(t)
}
```

List: provide Iter, ListIF

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

implicit val listIF : ListIF[List] =
  new ListIF[List] {
    def empty[A]: List[A] = Nil
    def head[A](l: List[A]) : Option[A] = l.headOption
    def tail[A](l: List[A]) : List[A] = l.tail
    def cons[A](a: A, l: List[A]) = a :: l
    def append[A](l1: List[A], l2: List[A]) = l1 ::: l2
  }
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
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]

def treeIterable[L[_]](implicit IF: ListIF[L], IT: Iter[L]): Iterable[MyTree] =
 new Iterable[MyTree] {
   type Itr[A] = L[A]
   def iter[A](a: MyTree[A]): L[A] = a match {
     case Empty() => IF.empty
     case Node(v, left, right) =>
       IF.cons (v, IF.append(iter(left), iter(right)))    }
   val iterIF : Iter[L] = IT }

implicit val treeIterableList : Iterable[MyTree] = treeIterable[List]
```

MyTree: use Iter, ListIF, provide Iterable, TreeIF

```
implicit val treeIF : TreeIF[MyTree] =
  new TreeIF[MyTree] {
    def empty[A] = Empty()
    def node[A](a: A, l: MyTree[A], r: MyTree[A]) = Node(a,l,r)
    def head[A](t: MyTree[A]) = t match {
      case Empty() => None
      case Node(v,_,_) => Some(v)    }
    def left[A](t: MyTree[A]) = t match {
      case Empty() => t
      case Node(_,lt,_) => lt    }
    def right[A](t: MyTree[A]) = t match {
      case Empty() => t
      case Node(_,_,rt) => rt    }
  }
```

Linking Modules

```
testList[List]


testTree[MyTree]
```

Iter being Iterable

```
implicit def iterIterable[I[_]](implicit IT: Iter[I]): Iterable[I]
  = new Iterable[I] {
    type Itr[A] = I[A]
    def iter[A](a: I[A]): I[A] = a
    def iterIF: Iter[I] = IT
  }

val l = List(3,5,2,1)
sumElements2(l)
printElements2(l)
```

List with Map

```
trait ListIF[L[_]] {
  def empty[A] : L[A]
  def head[A](l: L[A]) : Option[A]
  def tail[A](l: L[A]) : L[A]
  def cons[A](a: A, l: L[A]) : L[A]
  def append[A](l1: L[A], l2: L[A]) : L[A]
  def map[A,B](f: A=>B)(l: L[A]): L[B]     // Added
}

def testList[L[_]](implicit LI: ListIF[L], IT: Iter[L]) = {
  val l1 = LI.cons(3.3, LI.cons(2.2, LI.cons(1.5, LI.empty)))
  val l2 = LI.map((n:Double)=>n.toInt)(l1)
  val l3 = LI.map((n:Int)=>n.toString)(l2)
  printElements(l3)
}
```

List with Map

```
implicit val listIF : ListIF[List] =
  new ListIF[List] {
    def empty[A]: List[A] = Nil
    def head[A](l: List[A]) : Option[A] = l.headOption
    def tail[A](l: List[A]) : List[A] = l.tail
    def cons[A](a: A, l: List[A]) = a :: l
    def append[A](l1: List[A], l2: List[A]) = l1 ::: l2
    def map[A,B](f: A=>B)(l: List[A]): List[B] = l.map(f)  // Added
  }

testList[List]
```

Even Higher Kinds

```scala
// Iter: (* -> *) -> *
trait Iter[I[_]] {
  def getValue[A](a: I[A]) : Option[A]
  def getNext[A](a: I[A]) : I[A]
}

// I: (* -> *) -> *
// Foo: ((* -> *) -> *) -> *
trait Foo[I[_[_]]] {
  def get : I[List]
}

def f(x: Foo[Iter]) : Iter[List] = x.get
```

Turning Type Classes into OO Classes

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) : ???
}

trait UserInteraction {
  def run(factory: DPFactory) : Unit
}
```

How to return data with associated functions like OOP?

Turning Type Classes into OO Classes

```
import scala.language.higherKinds
import scala.language.implicitConversions

trait Box[S[_]] {
  type Data
  val d: Data
  val i: S[Data]
}

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

Interfaces

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

trait DPFactory {
  def getTypes: String
  def makeDP(dptype: String) : Box[DataProcessor]
}

trait UserInteraction {
  def run(implicit factory: DPFactory) : Unit
}
```

User Intraction

```
val userInteraction = new UserInteraction {
  def run(factory: DPFactory) = {
    val dptype = scala.io.StdIn.readLine(
      "Input a processor type " + factory.getTypes.toString + ": ")
    val dp = factory.makeDP(dptype)
    val d_done = getInputs(dp.d)(dp.i)
    printOutputs(d_done)(dp.i)
  }
  def getInputs[D](d: D)(implicit DP: DataProcessor[D]): D = {
    val d2 = DP.input(d, scala.io.StdIn.readLine("Input Data: "))
    val done = scala.io.StdIn.readLine("More inputs? [Y/N]: ")
    if (done.toLowerCase() == "n") d2
    else getInputs(d2)
  }
  def printOutputs[D](d: D)(implicit DP: DataProcessor[D]) = {
    println("The result of processing your inputs is:")
    println(DP.output(d))
  } }
```

Data Processor

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

Linking

userInteraction.run(dpfactory)

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

def sumElementsList(xs: List[Box2[Iter,Int]]) : Int =
  xs match {
    case Nil => 0
    case hd :: tl => sumElements(hd.d)(hd.i) + sumElementsList(tl)
  }
```

Turning Type Classes into OO Classes

```
import scala.language.higherKinds
import scala.language.implicitConversions

trait Box2[S[_,_],A] {
  type Data
  val d: Data
  val i: S[Data,A]
}

object Box2 {
  implicit def apply[S[_, _],D,A](dd: D)(implicit ii: S[D,A]):
  Box2[S,A] = new Box2[S,A] {
    type Data = D
    val d = dd
    val i = ii
  }
}
```

Test

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

implicit def decIter : Iter[Int,Int] = new Iter[Int,Int]
{
  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))
```

Iterable

```
// trait Iterable[R,A] {
//   type Itr
//   def iter(a: R): Itr
//   def iterIF: Iter[Itr, A]
// }
trait Iterable[R,A] {
  def iter(a: R): Box2[Iter,A]
}

def sumElements2[R](xs: R)(implicit ITR: Iterable[R,Int]) = {
  val cs = ITR.iter(xs)
  sumElements(cs.d)(cs.i)
}

def printElements2[R,A](xs: R)(implicit ITR: Iterable[R,A]) = {
val cs = ITR.iter(xs)
  printElements(cs.d)(cs.i)
}
```

MyTree

```
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]

implicit def treeIterable[A] :
  Iterable[MyTree[A], A] = new Iterable[MyTree[A], A] {
  def iter(a: MyTree[A]) : Box2[Iter,A] = {
    def go(l: MyTree[A]) : List[A] =
      l match {
        case Empty() => Nil
        case Node(v, left, right) => v :: (go(left) ++ go(right)) }
    go(a) // Box2(go(a))(listIter[A])
  } }
val t : MyTree[Int] =
  Node(3,Node(4,Empty(),Empty()),Node(2,Empty(),Empty()))
sumElements2(t) //sumElements2(t)(treeIterable[Int])
printElements2(t) //printElements2(t)(treeIterable[Int])
```

Iter being Iterable

```
implicit def iterIterable[I,A](implicit IT: Iter[I,A]) :
  Iterable[I,A] = new Iterable[I,A] {
    def iter(a: I) = a // Box2(a)(IT)
  }

val l = List(3,5,2,1)
sumElements2(l)
printElements2(l)
```

Stacking with Type Classes

IntStack Spec

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

def testStack[S](implicit stk: Stack[S,Int]) = {
  val s0 = stk.empty
  val s1 = stk.put(s0)(3)
  val s2 = stk.put(s1)(-2)
  val s3 = stk.put(s2)(4)
  val (v1,s4) = stk.get(s3)
  val (v2,s5) = stk.get(s4)
  (v1,v2)
}
```

Implementation using List

```
implicit def StackListInt : Stack[List[Int],Int] =
  new Stack[List[Int],Int] {
    val empty = List()
    def get(s: List[Int]) = (s.head, s.tail)
    def put(s: List[Int])(x: Int) = x :: s
  }
```

Modifying Traits

```
def IntStackWithPut[S](parent: Stack[S,Int],
                       newPut: (S,Int) => S) : Stack[S,Int] =
  new Stack[S,Int] {
    def empty = parent.empty
    def get(s: S) = parent.get(s)
    def put(s: S)(x: Int) = newPut(s,x)
  }

def Doubling[S](parent: Stack[S,Int]) : Stack[S,Int] =
  IntStackWithPut(parent, (s,x) => parent.put(s)(2 * x))

def Incrementing[S](parent: Stack[S,Int]) : Stack[S,Int] =
  IntStackWithPut(parent, (s,x) => parent.put(s)(x + 1))

def Filtering[S](parent: Stack[S,Int]) : Stack[S,Int] =
  IntStackWithPut(parent,
                  (s,x) => if (x >= 0) parent.put(s)(x) else s)
```

Linking

```
testStack(Filtering(Incrementing(Doubling(StackListInt))))
```

Implementation: Sorted Stack

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

testStack(Filtering(Incrementing(Doubling(SortedStackListInt))))
```

PART 4
Imperative Programming with Memory Updates

Mutable Variables

- Mutable Variables
  - Use “var” instead of “val” and “def”
  - We can update the value stored in a variable.

```
class Main(i: Int) {
  var a = i
}

val m = new Main(10)
m.a  // 10
m.a = 20
m.a  // 20
m.a += 5  // m.a = m.a + 5
m.a  // 25
```

While loop

➢While loop

- Syntax: `while` (*cond*) *body*
  Executes *body* while *cond* holds.
- It is equivalent to:

```
def mywhile(cond: =>Boolean)(body: =>Unit) : Unit =
  if (cond) { body; mywhile(cond)(body) } else ()
```

➢Example

```
var i = 0
var sum = 0
while (i <= 100) { // mywhile (i <= 100) {
  sum += i
  i += 2
}
sum // 2550
```

For loop

➢For loop

- Syntax: `for (i <- collection) body`
  Executes *body* for each `i` in *collection*.
- It is equivalent to:

```
def myfor[A](xs: Traversable[A])(f: A => Unit) : Unit =
  xs.foreach(f)
```

➢Example

```
var sum = 0
for (i <- 0 to 100 by 2) { // myfor (0 to 100 by 2) { i =>
  sum += i
}
sum // 2550
```

Additional Resources

- UCSD CSE 130
  - http://cseweb.ucsd.edu/classes/wi14/cse130-a/lectures/scala/00-crash.html
  - http://cseweb.ucsd.edu/classes/wi14/cse130-a/lectures/scala/01-iterators.html

Thanks for your hard work!