Principles of Programming

(4190.306)

Chung-Kil Hur (허충길)

Department of Computer Science and Engineering
Seoul National University

Syllabus

- Lecture & Practice Session
  - Tue & Thu, 9:00 ~ 10:50 (302-208)
  - https://github.com/snu-sf-class/pp201701
  - Bring your laptop to lectures & practice sessions
- Instructor
  - Chung-Kil Hur
  - http://sf.snu.ac.kr/gil.hur/
- Teaching Assistant
  - Yoonseung Kim
  - http://sf.snu.ac.kr/yoonseung.kim/
- Grading
  - 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.), U of Cambridge (Ph.D.)
  - Software Foundations Lab
    http://sf.snu.ac.kr
  - Research Topics
    - Software Verification
    - Low-level Language Semantics (C/C++/LLVM/Rust)
    - Concurrency Models
  - Our collaborators
    - [UK] U of Cambridge, Microsoft Research Cambridge
    - [Germany] Max Planck Institute for Software Systems
    - [France] INRIA
    - [USA] Princeton, UPenn, Utah, State U of New York at Oswego, Google, IBM, Mozilla, Azul Systems.
  - Publications
    - 8 top conference papers (last 4 years at SNU). PLDI(4),POPL(2),ICFP(1),AAAI(1)

Introduction

Overview

- Part 1
  Functional Programming with Function Applications

- Part 2
  Object-Oriented Programming with Subtypes

- Part 3
  Type-Oriented Programming with Typecasts

- Part 4
  Imperative Programming with Memory Updates

Imperative vs. Functional Programming

- Imperative Programming
  - Computation by memory reads/writes
  - Sequence of read/write operations
  - Repetition by loop
  - More procedural
  - 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
  - Easier to write safe code

```
def sum(n) =
  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, …
  - More functional: OCaml, SML, Lisp, Scheme, …
  - Middle: Scala
  - Purely functional: Haskell

Object-Oriented Programming

- Object-Oriented Programming
  - Classes/Objects: data + methods
  - Subtyping (Inheritance): hierarchy among classes
  - Can separate Specification & Implementation

```
class Point(x: Int, y: Int) {
  // data
  val px : Int = x
  val py : Int = y
  // methods
  def plus(q : Point) : Point =
    new Point(px + q.px, py + q.py)
}
val p1 : Point = new Point(3,4)
val p2 : Point = new Point(1,10)
val p3 : Point = p1.plus(p2)
```

Type-Oriented Programming

- Type-Oriented Programming
  - Type-Classes/Instances: type + methods
  - Typecasting: hierarchy among type classes
  - Can separate Specification & Implementation

```scala
class Point(x: Int, y: Int) {
  // data
  val px : Int = x
  val py : Int = y
}
val Point = new {
  // methods
  def plus(p : Point, q: Point) : Point =
    new Point(p.px + q.px, p.py + q.py)
}
val p1 : Point = new Point(3,4)
val p2 : Point = new Point(1,10)
val p3 : Point = Point.plus(p1,p2)
```

Object-Oriented vs. Type-Oriented

(My opinion) TOP is better than OOP.

Why Scala?

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

PART 1

Functional Programming with Function Applications

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 (= Programming)
  - 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) ~ … ~ 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 “=>”

```scala
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

```scala
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` $e_2 \rightarrow e_1$
  - `if (false)` e_1 `else` $e_2 \rightarrow e_2$

```
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 …
```

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)
def g(x: Int) = x * x
val x = f(5)
val r = {
  val t = 10
  val s = f(5)
  t + s
}
val y = 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 inside a block is accessible unless it is shadowed
- A function is evaluated under the environment where it is defined, not the environment where it is invoked.

Rewriting for Blocks

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

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

- Evaluation with Environment

[],1 ~ [t=0],2 ~ […,f=(x)t+g(x)],3 ~ […,g=(x)x*x],4 ~ […,x=25],5 ~ […]:[],6 ~ […]:[t=10],7 ~ […]:[…,s=25],8 ~ […,r=35],9 ~ […,y=35],10

4: [t=0,f=…,g=…]:[x=5],t+g(x) ~ 0+g(5) ~ 25

g(5): [t=0,f=…,g=…]:[x=5],x*x ~ 5*5 ~ 25

7: [t=0,f=…,g=…,x=25]:[x=5],t+g(x) ~ 0+g(5) ~ 25

g(5): [t=0,f=…,g=…,x=25]:[x=5],x*x ~ 5*5 ~ 25

Example: def with no arguments

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

➢Evaluation with Environment

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

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)
}
```

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

  $(x_1: T_1,\ldots,x_n:T_n) \Rightarrow e$

  or

  $(x_1,\ldots,x_n) \Rightarrow e$

```
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(combine: (Int,Int)=>Int,inival: Int,
              f: Int=>Int, a: Int, b: Int): Int = {
  if (a <= b) combine(f(a),mapReduce(combine,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 = {
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

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

Parameterized expression vs. values

- Functions defined using "def" are not values but parameterized expressions.
- Anonymous functions are values.
- But, parameterized expressions are implicitly converted to values.
- Explicit conversion: f _
- Anonymous functions can be seen as syntactic sugar:

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

  is equivalent to

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

  as long as `__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: functions 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  // x is treated as x()
3: val r = {
4:    val t = 10
5:    f(t*t) }              // t*t is treated as ()=>t*t
```

- Evaluation with Closures

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

  5: [t=0,f=…]:[x={[t=0,f=…]:[t=10],()t*t}],t+x
     ~ 0 + x ~ 0 + 100 ~ 100
    x: [t=0,f=…]:[t=10]:[],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 linear(n: Int) = n
def square(n: Int) = n * n
def cube(n: Int) = n * n * n
def sumLinear(a: Int, b: Int) = sum(linear, a, b)
def sumSquare(a: Int, b: Int) = sum(square, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, 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
}

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

Benefits

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

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

We don't need to define the wrapper functions.

```
sum(square)(3,10) + sum(cube)(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
}
```

We can also write as follows.

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

Or more simply:

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

Currying and Uncurrying

- A function of type

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

  can be turned into one of type

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

- 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.

Solution

```
def mapReduce(combine:(Int,Int)=>Int,inival: Int)
              (f: Int=>Int) (a: Int, b: Int): Int = {
  if (a <= b) combine(f(a),mapReduce(combine,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

```
val foo = new //or, object foo
{
  val a = 3
  def b = a + 1
  def f(x: Int) = b + x
  def f(x: String) = "hello" + x
}


foo.b
foo.f(3)
foo.f("gil")

def g(x: {val a: Int; def b: Int;
          def f(x:Int): Int; def f(x:String): String}) =
  x.f(3)

g(foo)
```

Structural Types: Scope and 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)
```

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(1,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
   …
  case Cn(…) => en
}
```

Pattern Matching: An Example

```
def length(xs: IList) : Int =
  xs match {
    case INil() => 0
    case ICons(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.

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()
  }
```

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) that checks whether x is in t.

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 t: BTree = Node(5,Node(4,Node(2,Leaf(),Leaf()),Leaf()),
  Node(7,Node(6,Leaf(),Leaf()),Leaf()))

find(t,7)
```

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.

- Why 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 DRY?
  - 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](x:A)A`
  - `id` is a parametric expression.
  - `id[T] _` is a value of type `T=>T` for any type `T` .

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

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())
```

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 with Subtypes

Sub Type Polymorphism (Concept)

Motivation

We want:

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

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

Note that we have

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

Sub Types to the Rescue!

```
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 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 different.

Structural Sub Types

General Sub Type Rules

- Reflexivity:
  For any type T, we have:

$$T <: T$$

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

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

Sub Types for Special Types

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

- For any type T, we have:

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

- Example

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

Sub Types for Records

- Permutation

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

- Width

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

- Depth

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

Sub Types for Records

- Example

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

  <: (by permutation)

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

  <: (by depth & width)

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

Sub Types for 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

```
object gee {
  val a : Int = 10
  def b : Int = a + 20
  def f(z: Int) : Int = b + 20 + z
}
type gee_type = {val a:Int; def b: Int; def f(z:Int): Int}

class foo_type(x: Int, y: Int) {
  val a : Int = x
  def b : Int = a + y
  def f(z: Int) : Int = b + y + z
}
val foo : foo_type = new foo_type(10,20)
```

- 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

foo_type <: gee_type

- Classes: Nominal sub-typing

gee_type ~~<:~~ foo_type

```
val v1 : gee_type = foo

val v2 : foo_type = goo  // type error
```

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 = v
  val next = 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 {
  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
```

Q: Can we override with a different type?

```
override def f(z: Any): Int = 77        //No,  arg: diff type
def f(z: Any): Int = 77                 //Yes, arg: diff type
override def f(z: Int): Nothing = ??? //Yes, ret:  sub type
```

Overriding 2

```
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 b = 10
}

(new gee_type(30)).c
```

Example: My List

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

Let’s use sub typing.

```
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 foo(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())
```

+ Pattern Matching

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 Specification

Abstract Class: Specification

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

Abstract classes for Specification
Concrete sub-classes for Implementation

Abstract Class: Specification

➢ Example Specification

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

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

Concrete Class: Implementation

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

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

sumElements(t1)
```

Exercise

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

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

sumElements(new IntCounter(100))
```

Solution

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

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

sumElements(new IntCounter(100))
```

More on Abstract Classes

Problem: Iter for MyTree

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

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]?

Solution: Better Specification

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

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

Let's Use MyList

```
sealed abstract class MyList[A] extends Iter[A]
case class MyNil[A]() extends MyList[A] {
  def getValue = None
  def getNext = this
}
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] {
  def 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 = this
  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] {
  override def iter : MyList[A]
  /* Note:
  override def iter : Int // Type Error (no bug in Scala)
                          // because not (Int <: Iter[A])
  */
}
case class Empty[A]() extends MyTree[A] {
  def iter = MyNil()
}
case class Node[A](value: A,
                   left: MyTree[A],
                   right: MyTree[A])
  extends MyTree[A] {
  val iter = MyCons(value, left.iter.append(right.iter))
}
```

Test

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

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

sumElements(t.iter)
```

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())))

sumElements(lst.iter)
```

Wrapper for Inheritance

Using a Wrapper Class

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

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

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

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

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

sumElements(t.iter)
```

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

MyTree Using ListIter

```scala
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(t)
```

Abstract Classes with Arguments

Abstract Class with Arguments

```
abstract class Iterable[A](eq: (A,A) => Boolean) {
  type iter_t
  def iter: iter_t
  def getValue(i: iter_t) : Option[A]
  def getNext(i: iter_t) : iter_t
  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 Iterable[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(t)
```

More on Classes

Motivating Example

```scala
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

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

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

Traits for Specification

Motivation

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

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

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

Q: How can we extend ListIter and implement Dict?

Problems

- 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.
```

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

Traits to the rescue!

- Traits
  - Are the same as abstract classes
  - But, have only one constructor with no arguments

Specification using Traits

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


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

trait Dict[K,V] {
  def equals(k1: K, k2: K): Boolean
  def add(k: K, v: V): Dict[K,V]
  def find(k: K): Option[V]
}
```

Implementing Traits

```
class ListIter[A](list: List[A]) extends Iter[A]
{ def getValue = list.headOption
  def getNext = new ListIter(list.tail) }
class ListIterDict[K,V]
      (eq: (K,K)=>Boolean, list: List[(K,V)])
      extends ListIter[(K,V)](list)
         with Dict[K,V] {
  def equals(k1:K, k2:K) = eq(k1,k2)
  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 (equals(k, k1)) Some(v1) else go(tl) }
    go(list) }
}
```

Mixin with Traits

Motivation: Mixin Functionality

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

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

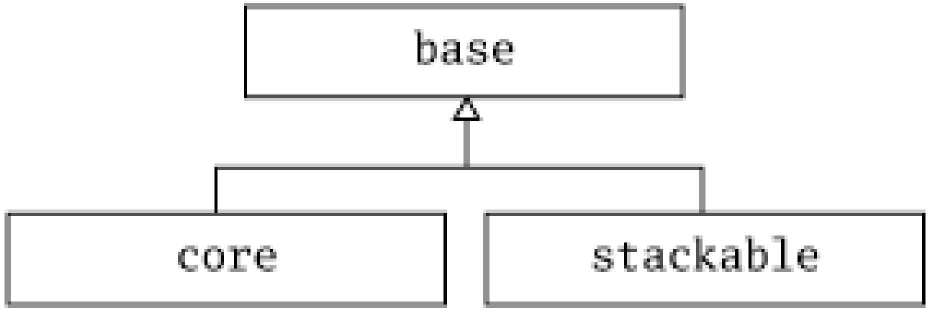
```
trait MRIter[A] extends Iter[A] {
  override def getNext: MRIter[A]
  def mapReduce[B,C](combine: (B,C)=>C, ival: C, f: A=>B): C =
    getValue match {
      case None => ival
      case Some(v) =>
        combine(f(v), getNext.mapReduce(combine, ival, f))
    }
}

class MRListIter[A](list: List[A])
  extends ListIter (list) with MRIter[A]
{
  override def getNext: MRIter[A] = 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)
```

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 IntStack {
  def get(): (Int,IntStack)
  def put(x: Int): IntStack
}
```

IntStack: Core

➢ **CORE**

```
class BasicIntStack protected (xs: List[Int]) extends IntStack
{
  def this() = this(Nil)
  protected def mkStack(xs: List[Int]): IntStack =
    new BasicIntStack(xs)
  def get(): (Int,IntStack) = (xs.head, mkStack(xs.tail))
  def put(x: Int): IntStack = 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 IntStack {
  abstract override def put(x: Int): IntStack = super.put(2 * x)
}

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

trait Filtering extends IntStack {
  abstract override def put(x: Int): IntStack =
    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]): IntStack =
    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()
```

Subtypes for Traits

Examples

```
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]): IntStack =
    new DIFIntStack(xs)
}

val s0 = new DIFIntStack
val t0 : Incrementing with Doubling = s0
val t1 : Incrementing with BasicIntStack = s0
```

Subtype Relation for “with”

The subtype relation for “with” is structural, not nominal, because we cannot not use “with” recursively.

- 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

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

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-Oriented Programming with Typecasts

Subtype & Parametric Polymorphism

Subtype & Parametric Polymorphism

- Subtype Polymorphism
  - Specification & Implementation

- Parametric Polymorphism
  - DRY (Generic Programming)

- Specification over Parameter Types
  - Precise Spec

- Parameter Types over Specification
  - More DRY

*** We will see a better way of doing Spec & Impl later.

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 c }
  else        { if (b <= c) c else B }
```

* Problem: hard (almost impossible) to define OrdInt <: Ord

Specification over Parameter Types

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

  def ===(that: Ord[A]): Boolean = (this.cmp(that)) == 0
  def <  (that: Ord[A]): Boolean = (this cmp that) < 0
  def >  (that: Ord[A]): Boolean = (this cmp that) > 0
  def <= (that: Ord[A]): Boolean = (this cmp that) <= 0
  def >= (that: Ord[A]): Boolean = (this cmp that) >= 0
}
def max3[A](a: Ord[A], b: Ord[A], c: Ord[A]) : Ord[A] =
  if (a <= b) {if (b <= c) c else b }
  else        {if (a <= c) c else a }
class OInt(val getInt : Int) extends Ord[OInt] {
  def cmp(that: Ord[OInt]) = getInt.compare(that.getValue.getInt)
  def getValue = this
}
max3(new OInt(3), new OInt(2), new OInt(10)).getValue.getInt
```

Further example: Ordered Bag

```
class Bag[A <: Ord[A]] 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 (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))
b.toList.map((x)=>x.getInt)
```

Works, but Very Awkward!!!

Type-Oriented Programming
(No Object-Oriented Programming!)

Type-Oriented Programming using Implicit

- Ideas from Type Classes
  - Object-Oriented: Associate functionality with data (Bad)
  - Type-Oriented: Associate functionality with types (Good)

- How to do Type-Oriented Programming
  - Separate functionality from data
  - Find the associated functionality for a given type using “implicit”

- Implicit
  - An argument is given “implicitly”

```
def foo(s: String)(implicit t: String) = s + t
implicit val exclamation : String = "!!!!!!"
foo("Hi")
foo("Hi")("???") // possible to give it explicitly
```

View Bounds (Better, but Not So Good)

Separating Ord from Int (up to some degree)

```
abstract class 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
}
```

Bag Example

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

implicit def intProxy(x:Int) =
  new Ord[Int] { def cmp(that: Int) = x.compare(that) }

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

View Bound: Syntactic Sugar

```
// class Bag[A] protected (val toList: List[A])
//                         (implicit proxy: A => Ord[A])
class Bag[A <% Ord[A]]protected (val toList: List[A])
{
  // def this()(implicit proxy: A => Ord[A]) = this(Nil)(proxy)
  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 (x < e) => x :: elmts
        case e :: _ if (x === e) => elmts
        case e :: rest => e :: go(rest)
      }
    new Bag(go(toList))
  }
}
```

Bootstrapping Proxies

```
// lexicographic order
implicit def tup2Proxy[A <% Ord[A], B <% Ord[B]](x: (A, B)) =
  new Ord[(A, B)] {
    def cmp(that: (A, B)) : Int = {
      val c1 = x._1 cmp that._1
               // implicitly[A](x._1).cmp(that._1)
      if (c1 != 0) c1 else { x._2 cmp that._2 }
    }
}

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

Type Classes (Good)

Completely Separating Ord from Int

```
abstract class 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
}
```

Bag Example

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

  def add(x: A) : Bag[A] = {
    def go(elmts: List[A]) : List[A] =
      elmts match {
        case Nil => x :: Nil
        case e :: _ if (proxy.<(x,e)) => x :: elmts
        case e :: _ if (proxy.===(x,e)) => elmts
        case e :: rest => e :: go(rest)
      }
    new Bag(go(toList))
  }
}
implicit def intProxy : Ord[Int] =
  new Ord[Int] { def cmp(me: Int, you: Int) = me.compare(you) }

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

Context Bound: Syntactic Sugar

```
//class Bag[A] protected(val toList:List[A])(implicit proxy:Ord[A])
class Bag[A : Ord] protected (val toList: List[A])
{ val proxy = implicitly[Ord[A]]
  // def this()(implicit proxy: Ord[A]) = this(Nil)(proxy)
  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 (proxy.<(x,e)) => x :: elmts
        case e :: _ if (proxy.===(x,e)) => elmts
        case e :: rest => e :: go(rest)
      }
    new Bag(go(toList))
  }
}
```

Bootstrapping Proxies

```
// lexicographic order
implicit def tup2Proxy[A : Ord, B : Ord] = {
  val proxyA = implicitly[Ord[A]]
  val proxyB = implicitly[Ord[B]]
  new Ord[(A, B)] {
    def cmp(me: (A,B), you: (A, B)) : Int = {
      val c1 = proxyA.cmp(me._1, you._1)
      if (c1 != 0) c1
      else { proxyB.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 Order

```
def intProxyRev : Ord[Int] =
  new Ord[Int] { def cmp(me: Int, you: Int) = you.compare(me) }

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

Type Classes With Multiple Parameters

Iter

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

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

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

List

```scala
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
  }

val l = List(3,5,2,1)

sumElements(l) //sumElements(l)(listIter[Int])

printElements(l) //printElements(l)(listIter[Int])
```

Iterable

```
// trait Iterable[A] {
//   def iter : Iter[A]
// }
abstract class Iterable[R,A] {
  type iterT
  def iter(a: R): iterT
  def iterProxy: Iter[iterT, A]
}

def sumElements2[R](xs: R)(implicit proxy: Iterable[R,Int]) =
  sumElements(proxy.iter(xs))(proxy.iterProxy)
  //sumElements[proxy.iterT](proxy.iter(xs))(proxy.iterProxy)

def printElements2[R,A](xs: R)(implicit proxy: Iterable[R,A]) =
  printElements(proxy.iter(xs))(proxy.iterProxy)
 //printElements[proxy.iterT,A](proxy.iter(xs))(proxy.iterProxy)
```

Iterable: Bad Designs

```
// Too much information is specified
abstract class Iterable2[R,I,A] {
  def iter(a: R): I
  def iterProxy: Iter[I,A]
}

// Too little information is specified
abstract class Iterable3[R] {
  type elmtT
  type iterT
  def iter(a: R): iterT
  def iterProxy: Iter[iterT, elmtT]
}
```

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] {
    type iterT = List[A]
    def iter(a: MyTree[A]): List[A] = a match {
      case Empty() => Nil
      case Node(v, left, right) => v :: (iter(left) ++ iter(right))
    }
    val iterProxy = implicitly[Iter[List[A], 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 proxy: Iter[I,A]) : Iterable[I,A] =
  new Iterable[I,A] {
    type iterT = I
    def iter(a: I) = a
    val iterProxy = proxy
  }

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

Higher-kind Type Classes

Iter

```scala
import scala.language.higherKinds
//trait Iter[I,A] {
//  def getValue(a: I): Option[A]
//  def getNext(a: I): I }
abstract class Iter[F[_]] {
  def getValue[A](a: F[A]) : Option[A]
  def getNext[A](a: F[A]) : F[A]
}
def sumElements[F[_] : Iter](xs: F[Int]) : Int = {
  val proxy = implicitly[Iter[F]]
  proxy.getValue(xs) match {
    case None => 0
    case Some(n) => n + sumElements(proxy.getNext(xs)) }}
def printElements[F[_] : Iter, A](xs: F[A]) : Unit = {
  val proxy = implicitly[Iter[F]]
  proxy.getValue(xs) match {
    case None =>
    case Some(n) => {println(n); printElements(proxy.getNext(xs))}}}
```

List

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

val l = List(3,5,2,1)

sumElements(l) //sumElements(l)(listIter)

printElements(l) //printElements(l)(listIter)
```

Iterable

```scala
//trait Iterable[R,A] {
//  type iterT
//  def iter(a: R): iterT
//  def iterProxy: Iter[iterT, A]
//}
abstract class Iterable[F[_]] {
  type iterT[_]
  def iter[A](a: F[A]): iterT[A]
  def iterProxy: Iter[iterT]
}

def sumElements2[F[_]](xs: F[Int])(implicit proxy: Iterable[F]) =
  sumElements(proxy.iter(xs))(proxy.iterProxy)
  //sumElements[proxy.iterT](proxy.iter(xs))(proxy.iterProxy)

def printElements2[F[_],A](xs: F[A])(implicit proxy: Iterable[F]) =
  printElements(proxy.iter(xs))(proxy.iterProxy)
  //printElements[proxy.iterT,A](proxy.iter(xs))(proxy.iterProxy)
```

MyTree

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

implicit val treeIterable : Iterable[MyTree] =
  new Iterable[MyTree] {
    type iterT[A] = List[A]
    def iter[A](a: MyTree[A]) : List[A] = a match {
      case Empty() => Nil
      case Node(v,left,right) => v :: (iter(left) ++ iter(right)) }
    val iterProxy = implicitly[Iter[List]]
  }

val t : MyTree[Int] =
        Node(3,Node(4,Empty(),Empty()),Node(2,Empty(),Empty()))
sumElements2(t) //sumElements2(t)(treeIterable)
printElements2(t) //printElements2(t)(treeIterable)
```

Iter being Iterable

```
implicit def iterIterable[F[_]]
              (implicit proxy: Iter[F]) : Iterable[F] =
  new Iterable[F] {
    type iterT[A] = F[A]
    def iter[A](a: F[A]) = a
    def iterProxy = proxy
  }

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

Example: Functor Specification

```
trait Functor[F[_]] {
  def map[A,B](f: A=>B)(x: F[A]) : F[B]
}

def compose[F[_]:Functor,A,B,C](f: A=>B)(g: B=>C)(a: F[A]): F[C] = {
  val proxy = implicitly[Functor[F]]
  proxy.map(g)(proxy.map(f)(a))
}
```

Example: Functor Implementation

```
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 val ListFunctor : Functor[List] = new Functor[List] {
  def map[A,B](f: A=>B)(x: List[A]) = x.map(f)
}
implicit val MyTreeFunctor : Functor[MyTree] = new Functor[MyTree] {
  def map[A,B](f: A=>B)(x: MyTree[A]) : MyTree[B] = x match {
    case Empty() => Empty()
    case Node(v,l,r) => Node(f(v),map(f)(l),map(f)(r)) }
}

compose((x:Int)=>x+x)((x:Int)=>x*x)(List(1,2,3))
val t : MyTree[Int] =
        Node(3,Node(4,Empty(),Empty()),Node(2,Empty(),Empty()))
compose((x:Int)=>x+x)((x:Int)=>x*x)(t)
```

Even Higher Kinds

```
// Iter: (* -> *) -> *
abstract class Iter[F[_]] {
  def getValue[A](a: F[A]) : Option[A]
  def getNext[A](a: F[A]) : F[A]
}

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

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

Typecasts

Typecasts

```
class A(val v: Int)
class B(val v: Int)

implicit def A2B(x: A) : B = new B(x.v + 1)

def foo(x: B) = x.v

foo(new B(0)) // 0
foo(new A(0)) // 1 because foo(A2B(new A(0))) = 1
```

No Transitive Casting in Scala

```
class A(val v: Int)
class B(val v: Int)
class C(val v: Int)

implicit def A2B(x: A) : B = new B(x.v + 1)
implicit def B2C(x: B) : C = new C(x.v + 1)

def foo(x: C) = x.v

foo(new C(0)) // 0
foo(new B(0)) // 1 because bar(B2C(new B(0))) = 1
foo(new A(0)) // error, because there is no function: A => C
              // That is, Scala does not cast transitively
              // like bar(B2C(A2B(new A(0))))
```

A Trick to Implement Transitive Casting

```
class A(val v: Int)
class B(val v: Int)
class C(val v: Int)

implicit def A2B[T<%A](x:T): B = new B(x.v + 1)
//implicit def A2B[T](x:T)(implicit T2A:T=>A):B = new B(T2A(x).v+1)

implicit def B2C[T<%B](x:T): C = new C(x.v + 1)
//implicit def B2C[T](x:T)(implicit T2B:T=>B):C = new C(T2B(x).v+1)

def foo(x: C) = x.v

foo(new C(0)) // 0
foo(new B(0)) // 1 because bar(B2C[B](new B(0))((x)=>x)) = 1
foo(new A(0)) // 2 because bar(B2C[A](new A(0))
              //                  ((x)=>A2B[A](x)((x)=>x))) = 2
```

Typecasts vs. Subtypes

Typecasting is Better than Subtyping

- Subtypes
  - Structural Subtypes
    - ✓Subtype relation is too implicit. (Bad)
    - ✓Usually applies to non-recursive types only because typechecking is hard for recursive types. (Bad)
  - Nominal Subtypes
    - ✓Subtype relation is explicit. (Good)
    - ✓Subtype relation should be given at datatype definition time. (Bad)
- Typecasts using "implicit"
  - ✓Subtype relation is explicit (Good)
  - ✓Subtype relation can be given any time. (Good)

We will compare between

- Object-Oriented Programming (OOP) with Subtypes

- Type-Oriented Programming (TOP) with Typecasts

OOP with Subtypes

```
class Main[A](val a: A, val b: A) {
  def f(x: A) : Main[A] = new Main(x,x)
  def g(x: A) : Main[A] = new Main(a,x)
  def h : Main[A] = new Main(b,a)
}
class Sub[A](val c: A, val d: A) extends Main[A](d,c) {
  override def f(x: A) = new Sub(super.f(x).a,c)
  def r(x: A) : Sub[A] = new Sub(c,x)
}
def foo(m: Main[Int]) =
  m.f(10).a + m.g(20).b + m.h.a

val y : Sub[Int] = new Sub(100,200)
foo(y)  // 220

val z : Main[Int] = y
foo(z)  // 220, follows the run-time type information (Bad!)
        //      This is called "dynamic dispatch".
```

TOP with Typecasts: Main

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

class Main[A](val a: A, val b: A)

abstract class MainSig[M[_]] {
  def f[A](m: M[A])(x: A) : Main[A]
  def g[A](m: M[A])(x: A) : Main[A]
  def h[A](m: M[A]) : Main[A]
}

implicit val Main: MainSig[Main] = new MainSig[Main] {
  def f[A](m: Main[A])(x: A) = new Main(x,x)
  def g[A](m: Main[A])(x: A) = new Main(m.a,x)
  def h[A](m: Main[A]) = new Main(m.b,m.a)
}
```

TOP with Typecasts: Sub

```
class Sub[A](val c: A, val d: A) {
  val parent : Main[A] = new Main(d,c) }
implicit def SubMain[A,T<%Sub[A]](m:T) : Main[A] = m.parent

abstract class SubSig[M[_]] {
  val parent: MainSig[M]
  def r[A](m: M[A])(x: A) : Sub[A] }
implicit def SubSigMainSig[M[_]]
              (implicit m: SubSig[M]) : MainSig[M] = m.parent

implicit val Sub : SubSig[Sub] = new SubSig[Sub] {
  val parent : MainSig[Sub] = new MainSig[Sub] {
  def f[A](m:Sub[A])(x:A) = new Sub(Main.f(m)(x).a,m.c) //override
  def g[A](m:Sub[A])(x:A) = Main.g(m)(x)
  def h[A](m:Sub[A])       = Main.h(m)
  }
  def r[A](m:Sub[A])(x:A) = new Sub(m.c,x)
}
```

TOP with Typecasts: Test

```
def foo[M[_]](m: M[Int])(implicit proxy: MainSig[M]) =
  proxy.f(m)(10).a + proxy.g(m)(20).b + proxy.h(m).a

val y : Sub[Int] = new Sub(100,200)
foo(y) // 220

val z : Main[Int] = y
foo(z) // 130, follows the static type information (Good!)
       //      no need for run-time type information
```

TOP with Typecasts & Mixin: Main

```
class Main[A](val a: A, val b: A)

abstract class MainSig[M[_]] {
  def f[A](m: M[A])(x: A) : Main[A]
  def g[A](m: M[A])(x: A) : Main[A]
  def h[A](m: M[A]) : Main[A]
}

trait MainDef[M[_]] {
  implicit def castMain[A](m: M[A]) : Main[A]
  def f[A](m: M[A])(x: A) = new Main(x,x)
  def g[A](m: M[A])(x: A) = new Main(m.a,x)
  def h[A](m: M[A]) = new Main(m.b,m.a)
}

implicit val Main : MainSig[Main] =
  new MainSig[Main] with MainDef[Main] {
    def castMain[A](m: Main[A]) = m }
```

TOP with Typecasts & Mixin: Sub

```
class Sub[A](val c: A, val d: A) {
  val parent : Main[A] = new Main(d,c) }
implicit def SubMain[A,T<%Sub[A]](m:T) : Main[A] = m.parent

abstract class SubSig[M[_]] {
  val parent: MainSig[M]
  def r[A](m: M[A])(x: A) : Sub[A] }
implicit def SubSigMainSig[M[_]]
             (implicit m: SubSig[M]) : MainSig[M] = m.parent

trait SubDef[M[_]] {
  implicit def castSub[A](m: M[A]) : Sub[A]
  val parent : MainSig[M] = new MainSig[M] with MainDef[M] {
  implicit def castMain[A](m: M[A]) = castSub(m)
  override def f[A](m:M[A])(x:A) = new Sub(Main.f(m)(x).a,m.c) }
  def r[A](m: M[A])(x: A) = new Sub(m.c,x) }
implicit val Sub : SubSig[Sub] = new SubSig[Sub] with SubDef[Sub] {
  implicit def castSub[A](m: Sub[A]) = m }
```

TOP with Typecasts & Mixin: Test

```
def foo[M[_]](m: M[Int])(implicit proxy: MainSig[M]) =
  proxy.f(m)(10).a + proxy.g(m)(20).b + proxy.h(m).a

val y : Sub[Int] = new Sub(100,200)
foo(y) // 220

val z : Main[Int] = y
foo(z) // 130, follows the static type information (Good!)
       //      no need for run-time type information
```

Dynamic Dispatch with Typecasts (No Subtypes!)

Dynamic Dispatch in OOP

Dynamic Dispatch (value-dependent method call) might be sometimes needed.
Heterogeneous list is a good example.

```
def gee(l: List[Main[Int]]) : Int = l match {
  case Nil => 0
  case m :: tl => foo(m) + gee(tl)
}

foo(new Sub(100,200))  // 220
foo(new Main(100,200)) // 230

gee(List(new Sub(100,200),new Main(100,200))) // 450 = 220 + 230
```

Dynamic Dispatch in TOP (1)

```
abstract class MainDyn[A] {
  type MainT[_]
  val data : MainT[A]
  val proxy : MainSig[MainT]
}

implicit def toMainDyn[M[_]:MainSig,A](m:M[A]): MainDyn[A] =
  new MainDyn[A] {
    type MainT[A] = M[A]
    val data = m
    val proxy = implicitly[MainSig[M]]
  }
```

Dynamic Dispatch in TOP (2)

```
// Static Dispatch
def gee(l: List[Main[Int]]) : Int = l match {
  case Nil => 0
  case m :: tl => foo(m) + gee(tl)
}

// Dynamic Dispatch
def geeDyn(l: List[MainDyn[Int]]) : Int = l match {
  case Nil => 0
  case m :: tl => foo(m.data)(m.proxy) + geeDyn(tl)
}

foo(new Sub(100,200): Main[Int]) // 130
foo(new Sub(100,200))  // 220
foo(new Main(100,200)) // 230

gee(List(new Sub(100,200),new Main(100,200)))    // 360 = 130 + 230
geeDyn(List(new Sub(100,200),new Main(100,200))) // 450 = 220 + 230
```

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(c)(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