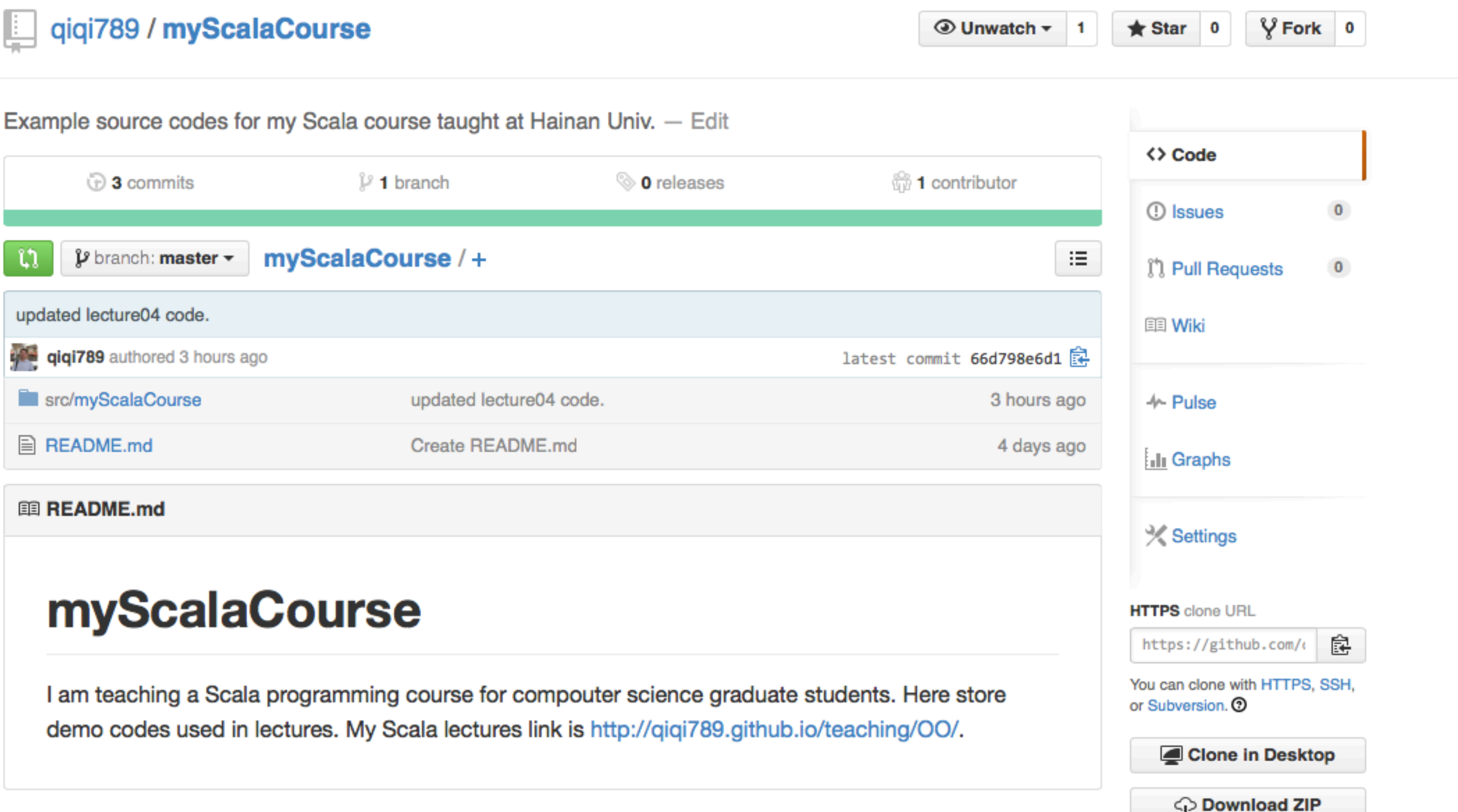
演示代码已发布到Github

- https://github.com/qiqi789/myScalaCourse



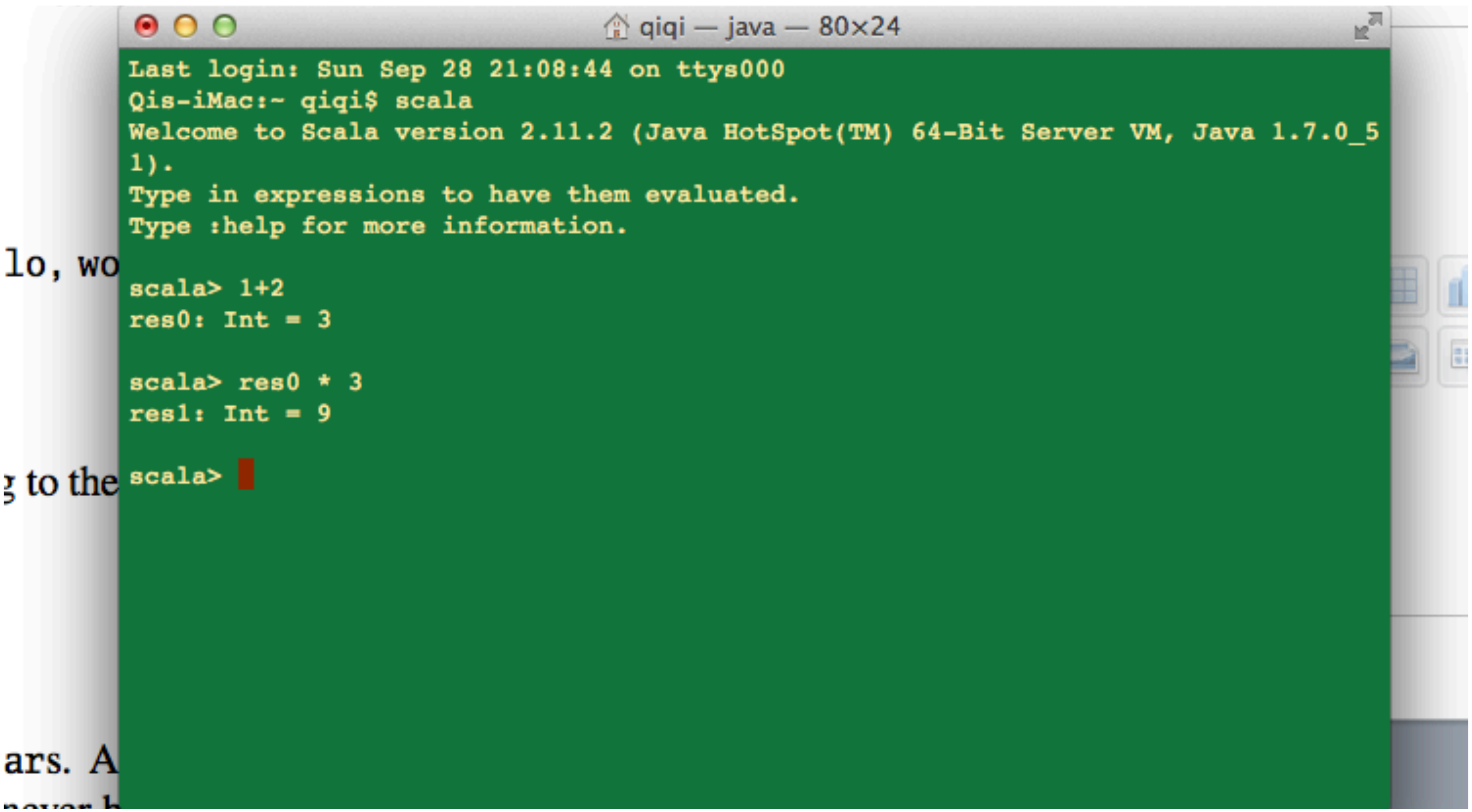
今天的内容

- **Scala** 基础（继续）
- Scala类和对象
- **Scala** 实例类和模式匹配
- 一般的类型和一般的方法

Scala 基础

齐琦
海南大学

Scala interpreter(解释器)



Sbt console (Scala 命令行解释器)

qiqi — java — 80×24

```
Qis-iMac:~ qiqi$ sbt console
[info] Set current project to qiqi (in build file:/Users/qiqi/)
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_5
1).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 1 + 1
res0: Int = 2

scala> 
```

定义变量，函数

- val:　定义的绑定不能改变
  - Val x = 6
- var：能够改变绑定值
  - var y = 6
  - y　= 9
- def : 定义函数
  - scala> def max(x:Int, y:Int): Int = {
  - 　　| if (x>y) x else y }
  - max: (x: Int, y: Int)Int

If 语句表达式 返回一个整型值

比较 val 和 def

- Val
  - Val x = e
  - 定义时，即刻算值；调用时，则用预先计算好的值替代
- Def
  - Def x = e
  - 定义时不计算值，调用**x**时才计算；每次调用，重新计算
  - 可以定义有参数的函数

输入参数的类型必须指定

```
scala> def addOne(m) = m+1
<console>:1: error: ':' expected but ')' found.
       def addOne(m) = m+1
                   ^

scala> def addOne(m):Int = m+1
<console>:1: error: ':' expected but ')' found.
       def addOne(m):Int = m+1
                   ^

scala> def addOne(m:Int) = m+1
addOne: (m: Int)Int

scala> addOne(2)
res1: Int = 3
```

匿名函数定义，存储

```
scala> def addOne(m:Int) = m+1
addOne: (m: Int)Int

scala> addOne(2)
res1: Int = 3

scala> (x: Int) => x + 1
res2: Int => Int = <function1>

scala> res2(1)
res3: Int = 2

scala> val addOne = (x: Int) => x + 1
addOne: Int => Int = <function1>

scala> addOne(1)
res4: Int = 2
```

多表达式函数定义

```
scala> def timesTwo(i: Int): Int = {
     | println("This is a timesTwo function.")
     | i * 2
     | }
timesTwo: (i: Int)Int

scala> timesTwo(2)
This is a timesTwo function.
res5: Int = 4

scala> { i : Int =>
     | println("another timeTwo function")
     | i * 2
     | }
res6: Int => Int = <function1>

scala> res6(3)
another timeTwo function
res7: Int = 6

scala> (i : Int) => {
     | println("another another timeTwo function")
     | i * 2
     | }
res8: Int => Int = <function1>

scala> res8(3)
another another timeTwo function
res9: Int = 6
```

部分函数应用（偏函数应用）

```
scala> def adder(m: Int, n: Int) = m + n
adder: (m: Int, n: Int)Int

scala> val add3 = adder(_ , 3)
<console>:8: error: missing parameter type for expanded function ((x$1) => adder
(x$1, 3))
       val add3 = adder(_ , 3)
                        ^


scala> val add3 = adder(_: Int , 3)
add3: Int => Int = <function1>

scala> add3(3)
res10: Int = 6
```

Curried 函数

```
scala> def multiply (m: Int) (n: Int): Int = m * n
multiply: (m: Int)(n: Int)Int

scala> multiply(2)(3)
res11: Int = 6

scala> val timesSix = multiply _ (6)
<console>:1: error: ';' expected but '(' found.
       val timesSix = multiply _ (6)
                                 ^

scala> val timesSix = multiply (6) _
timesSix: Int => Int = <function1>

scala> timesSix 3
<console>:1: error: ';' expected but integer literal found.
       timesSix 3
                ^

scala> timesSix(3)
res12: Int = 18
```

可变长的参数

```
scala> def capitalizeAll(args: String *) = {
     |   args.map{ arg =>
     |     arg.capitalize
     |   }
     | }
capitalizeAll: (args: String*)Seq[String]

scala> capitalizeAll("hainan", "university")
res14: Seq[String] = ArrayBuffer(Hainan, University)
```

Scala 举例

齐琦

海南大学

Quicksort: imperative version

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

- Def , var , val
- I : Int // type declaration
- Array[T] rather than T[]
  - a(i) than a[i]
- Nested functions; nested functions can access parameter & locals

QuickSort: functional style

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```

- 快排的本质
- 递归调用
- 返回新数组

- 时间复杂度（平均）都是 $O(N\ log(N))$

  最差情况：$O(N^2)$

- 更多空间

函数的应用解释

```
def filter(p: T => Boolean): Array[T]
```

- 断言函数（predicate function）
- 高阶函数 (higher-order functions)

- Scala 不区分标志符(identifiers) 和操作符名字（operator names）
  - 字母数字或特殊操作符的序列
  - Xs filter (pivot >)
    - Xs.filter(pivot >)
- 偏函数 pivot >
  - X => pivot > x
  - X的类型省略；自动从函数运行环境中推断

函数说明，继续

```
scala> def While (p: => Boolean) (s: => Unit) {
     |  if (p) {s; While(p)(s) }
     |  }
While: (p: => Boolean)(s: => Unit)Unit
```

- P，测试函数
  - 没有输入参数；返回一个布尔值
- S : => Unit
  - 命令执行函数
  - 没有输入参数；返回类型Unit的值（像是Java里的void）
  - 返回unit的函数也叫过程（procedure）
- 函数的返回值
  - 它里面的最后一个表达式的值
  - Return 不需要指明
  - 需要“=”在定义之前，如果返回一个显示的值

举例：牛顿方法求解平方根

1	2/1 = 2	1.5
1.5	2/1.5 = 1.3333	1.4167
1.4167	2/1.4167 = 1.4118	1.4142
1.4142	...	...
y	x/y	$(y+x/y)/2$

- 近似计算方法
  - X：输入参数
  - Y: 猜测值
- 循环可以被递归替代
- 见lecture03.scala 代码(用Scala-IDE打开)

嵌套函数

- 许多小函数（帮助函数）
  - sqrtIter, improve, isGoodEnough
  - 只用于 sqrt
  - 引发name-space pollution
  - 不希望用户直接访问它们
- 可在 sqrt 内定义
- Lecture03.scala 里的 newtonMethod.impl2
- 外围定义名字，在里层可见（除非定义同名）

尾递归（Tail Recursion）

```
def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
```

```
        gcd(14, 21)
→       if (21 == 0) 14 else gcd(21, 14 % 21)
→       if (false) 14 else gcd(21, 14 % 21)
→       gcd(21, 14 % 21)
→       gcd(21, 14)
→       if (14 == 0) 21 else gcd(14, 21 % 14)
→ →     gcd(14, 21 % 14)
→       gcd(14, 7)
→       if (7 == 0) 14 else gcd(7, 14 % 7)
→ →     gcd(7, 14 % 7)
→       gcd(7, 0)
→       if (0 == 0) 7 else gcd(0, 7 % 0)
→ →     7
```

- 最大公约数

- 替代模式
- 同一样式
- 固定堆栈空间

尾递归（Tail Recursion），继续

```
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
```

```
          factorial(5)
→         if (5 == 0) 1 else 5 * factorial(5 - 1)
→         5 * factorial(5 - 1)
→         5 * factorial(4)
→ ... →   5 * (4 * factorial(3))
→ ... →   5 * (4 * (3 * factorial(2)))
→ ... →   5 * (4 * (3 * (2 * factorial(1))))
→ ... →   5 * (4 * (3 * (2 * (1 * factorial(0)))))
→ ... →   5 * (4 * (3 * (2 * (1 * 1))))
→ ... →   120
```

- 累积乘

- 替代模式
- 变长
- 堆栈空间增长

函数是头等（First-Class）值

- 函数是值
  - 可以是输入参数
  - 也可是返回值
- 高阶(higher-order)函数

高阶函数举例

$$\Sigma_a^b f(n)$$

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

def id(x: Int): Int = x
def square(x: Int): Int = x * x
def powerOfTwo(x: Int): Int = if (x == 0) 1 else 2 * powerOfTwo(x - 1)

def sumInts(a: Int, b: Int): Int = sum(id, a, b)
def sumSquares(a: Int, b: Int): Int = sum(square, a, b)
def sumPowersOfTwo(a: Int, b: Int): Int = sum(powerOfTwo, a, b)
```

好像a,b有点累赘

高阶函数举例，继续 Currying

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

```scala
def sumInts   =  sum(x => x)
def sumSquares   =  sum(x => x * x)
def sumPowersOfTwo   =  sum(powerOfTwo)
```

```
scala> sumSquares(1, 10) + sumPowersOfTwo(10, 20)
unnamed0: Int = 2096513
```

$f(args_1)(args_2)$ is equivalent to $(f(args_1))(args_2)$

函数应用结合满足：左结合律

Currying，继续

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

$\textbf{def}\ f\ (args_1)\ \ldots\ (args_n) = E$

一个Curried 函数定义

$\textbf{def}\ f\ (args_1)\ \ldots\ (args_{n-1}) = \{\ \textbf{def}\ g\ (args_n) = E\ ;\ g\ \}$

$\textbf{def}\ f\ (args_1)\ \ldots\ (args_{n-1}) = (\ args_n\ ) \Rightarrow E$

$\textbf{def}\ f = (args_1) \Rightarrow \ldots \Rightarrow (args_n) \Rightarrow E$

函数类型：右结合律

$T_1 \Rightarrow T_2 \Rightarrow T_3$ is equivalent to $T_1 \Rightarrow (T_2 \Rightarrow T_3)$

另一个例子：寻找函数的定点

- X 是 一个函数f 的定点
  - F(x) = x
  - 收敛

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

```
val tolerance = 0.0001
def isCloseEnough(x: Double, y: Double) = abs((x - y) / x) < tolerance
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

另一个例子：寻找函数的定点

- 应用到平方根的求解上

```
sqrt(x)  =  the y such that  y * y = x
         =  the y such that  y = x / y
```

- Sqrt(x) 是函数 y=x/y 的定点，可以用定点迭代来估算
- 但是，

```
def sqrt(x: double) = fixedPoint(y => x / y)(1.0)
```

Then, `sqrt(2)` yields:

```
2.0
1.0
2.0
1.0
2.0
```

平方根估算

- 避免震荡
  - 平均化连续的值

```
scala> def sqrt(x: Double) = fixedPoint(y => (y + x/y) / 2)(1.0)
sqrt: (Double)Double

scala> sqrt(2.0)
  1.5
  1.4166666666666665
  1.4142156862745097
  1.414213562374689
  1.414213562374689
```

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2

def sqrt(x: Double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

Scala 类和对象

齐琦
海南大学

+实例：有理数类

```
class Rational(n: Int, d: Int) {
  private def gcd(x: Int, y: Int): Int = {
    if (x == 0) y
    else if (x < 0) gcd(-x, y)
    else if (y < 0) -gcd(x, -y)
    else gcd(y % x, x)
  }
  private val g = gcd(n, d)

  val numer: Int = n/g
  val denom: Int = d/g
  def +(that: Rational) =
    new Rational(numer * that.denom + that.numer * denom,
                 denom * that.denom)
  def -(that: Rational) =
    new Rational(numer * that.denom - that.numer * denom,
                 denom * that.denom)
  def *(that: Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that: Rational) =
    new Rational(numer * that.denom, denom * that.numer)
}
```

类和对象

- 私有成员 private
- 建立和访问对象

```
var i = 1
var x = new Rational(0, 1)
while (i <= 10) {
  x += new Rational(1, i)
  i += 1
}
println("" + x.numer + "/" + x.denom)
```

- 继承(inheritance)和重写(overriding)
  - 每个类都有一个父类
  - 根类: scala.AnyRef
  - 子类继承父类所有成员；可以重写

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
  override def toString = "" + numer + "/" + denom
}
```

类和对象：继承

- 子类对象可以被赋值给父类变量
  - 子类类型和父类类型一致

```scala
class Fruit
case class Apple(color: String) extends Fruit

def main(args: Array[String]): Unit = {
  val apple = Apple("red")
  val fruit = new Fruit
  val fruitList:List[Fruit] = List(apple)
  val appleList:List[Apple] = List(fruit)
```

类和对象：无参数方法

- 无参方法类似于值域，访问形式统一
- 但也有区别
- 提升类实现的灵活性
  - 类的版本不同（固定域值，需计算的值）

```
class Rational(n: Int, d: Int) extends AnyRef {
  ... // as before
  def square = new Rational(numer*numer, denom*denom)
}
val r = new Rational(3, 4)
println(r.square)            // prints''9/16''*
```

抽象类 和 特征类

- 可能有『推迟』的成员
  - 声明但没有实现
- 不能用**new**创建其对象
- 作为基类，或单独使用

```
abstract class IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

- 特征类(trait)
  - 为了加入到其他类；方法或域值
  - 抽象的功能（不同类所共有的，Java interface）

```
trait IntSet {
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean
}
```

实现抽象类

- 二叉树实现整数集合类

EmptySet 和 NonEmptySet 与 IntSet相一致

```
class EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}

class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
```

动态绑定

- 用于方法调用
  - 依赖于对象的实时类型

```
(new EmptySet).contains(7)
```

```
new NonEmptySet(7, new EmptySet, new EmptySet).contains(1)
```

  - 用相应类的方法实现代码来替代contains调用

对象

- 定义对象
  - 单一对象

```
object EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)
}
```

  - 没有构建函数参数；不能用new
  - 顶级(top-level)实体
  - 创建和初始化，是在其成员函数第一次被访问的时候（lazy evaluation）

标准类

- Scala是纯面向对象式语言
  - 每一个值(value)都是一个对象(object)
  - 基本类型是类的类型别名

```
type boolean = scala.Boolean
type int = scala.Int
type long = scala.Long
...
```

  - 几乎所有类型都是用类和对象来实现
    - 嵌入式类型为了优化和运行效率

```
package scala
abstract class Boolean {
  def ifThenElse(thenpart: => Boolean, elsepart: => Boolean)

  def && (x: => Boolean): Boolean  =  ifThenElse(x, false)
  def || (x: => Boolean): Boolean  =  ifThenElse(true, x)
  def !                 : Boolean  =  ifThenElse(false, true)

  def == (x: Boolean)   : Boolean  =  ifThenElse(x, x.!)
  def != (x: Boolean)   : Boolean  =  ifThenElse(x.!, x)
  def <  (x: Boolean)   : Boolean  =  ifThenElse(false, x)
  def >  (x: Boolean)   : Boolean  =  ifThenElse(x.!, false)
  def <= (x: Boolean)   : Boolean  =  ifThenElse(x, true)
  def >= (x: Boolean)   : Boolean  =  ifThenElse(true, x.!)
}
case object True extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = t
}
case object False extends Boolean {
  def ifThenElse(t: => Boolean, e: => Boolean) = e
}
```

+ MVC 例子

- 见lecture04.scala。

实例类和模式匹配

+ 实例：算术表达式，（方法1）

```
new Sum(new Number(1), new Sum(new Number(3), new Number(7)))
```

```
abstract class Expr {
  def isNumber: Boolean
  def isSum: Boolean
  def numValue: Int
  def leftOp: Expr
  def rightOp: Expr
}
class Number(n: Int) extends Expr {
  def isNumber: Boolean = true
  def isSum: Boolean = false
  def numValue: Int = n
  def leftOp: Expr = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
  def numValue: Int = error("Sum.numValue")
  def leftOp: Expr = e1
  def rightOp: Expr = e2
}
```

```
def eval(e: Expr): Int = {
  if (e.isNumber) e.numValue
  else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
  else error("unrecognized expression kind")
}
```

每个类内容繁冗

如果新加一个乘法类（Prod）？
扩展性差

实例：算术表达式，方法2

```
abstract class Expr {
  def eval: Int
}
class Number(n: Int) extends Expr {
  def eval: Int = n
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
}


class Prod(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval * e2.eval
}
```

- 方法2：面向对象编程
  - 使eval变成每个类的方法
  - 简化
  - 加Prod（新***数据类型***）不改变其他类

实例：算术表达式

- 如果，新加表达式操作？（打印一个表达式）

方法1：只加一个独立函数

```
def print(e: Expr) {
  if (e.isNumber) Console.print(e.numValue)
  else if (e.isSum) {
    Console.print("(")
    print(e.leftOp)
    Console.print("+")
    print(e.rightOp)
    Console.print(")")
  } else error("unrecognized expression kind"
}
```

似乎回到了原点

方法2：每个类都要改

```
abstract class Expr {
  def eval: Int
  def print
}
class Number(n: Int) extends Expr {
  def eval: Int = n
  def print { Console.print(n) }
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
  def print {
    Console.print("(")
    print(e1)
    Console.print("+")
    print(e2)
    Console.print(")")
  }
}
```

实例类和实例对象

```
abstract class Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

  def Number(n: Int) = new Number(n)
  def Sum(e1: Expr, e2: Expr) = new Sum(e1, e2)

  Sum(Sum(Number(1), Number(2)), Number(3))


 def n: Int          def e1: Expr, e2: Expr
```

- 使用case前缀
  - 隐式构建函数
  - 隐式实现方法：toString, equals, and hashCode
  - 隐式实现输入参数访问方法
  - 模式识别，根据构建函数的形式

模式匹配

```
def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(l, r) => eval(l) + eval(r)
}
```

- Switch 语句的通用化
- Match方法对所有对象可见
- 模式变量，绑定值
- MatchError

模式匹配，替换过程

```
eval(Sum(Number(1), Number(2)))

->                  (by rewriting the application)

Sum(Number(1), Number(2)) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}

->                  (by rewriting the pattern match)

eval(Number(1)) + eval(Number(2))

->                  (by rewriting the first application)

Number(1) match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
} + eval(Number(2))

->                  (by rewriting the pattern match)

1 + eval(Number(2))

->* 1 + 2 -> 3
```

模式匹配，匿名函数

$$\{\ \texttt{case}\ P_1\ \texttt{=>}\ E_1\ \ldots\ \texttt{case}\ P_n\ \texttt{=>}\ E_n\ \}$$

$$\downarrow$$

$$(\texttt{x}\ \texttt{=>}\ \texttt{x}\ \texttt{match}\ \{\ \texttt{case}\ P_1\ \texttt{=>}\ E_1\ \ldots\ \texttt{case}\ P_n\ \texttt{=>}\ E_n\ \})$$

+

一般的类型和一般的方法（Generic Types and Methods）

齐琦

为什么需要一般类型？

```
abstract class IntStack {
  def push(x: Int): IntStack = new IntNonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: Int
  def pop: IntStack
}
class IntEmptyStack extends IntStack {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class IntNonEmptyStack(elem: Int, rest: IntStack) extends IntStack {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

如何再定义一个字符串堆栈？

一般类型(generic types)含有类型参数

```
abstract class Stack[A] {
  def push(x: A): Stack[A] = new NonEmptyStack[A](x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
class EmptyStack[A] extends Stack[A] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

- 类型参数名字任意，在[ ]中

如何使用

```
val x = new EmptyStack[Int]
val y = x.push(1).push(2)
println(y.pop.top)
```

一般的方法（generic methods）

类型参数

值参数

```
def isPrefix[A](p: Stack[A], s: Stack[A]): Boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[A](p.pop, s.pop)
}
```

- 有类型参数的方法
- 多形的(polymorphic = “having many forms”)
- 局部类型推理，在使用时可省略类型参数
  - isPrefix(s1,s2)

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s1.top)
println(isPrefix[String](s1, s2))
```

类型参数界限：为什么需要？

```
class EmptySet extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}
```

```
class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
```

```
abstract class Set[A] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

- 加入类型参数[A]
- A 有定义 > 或 < 方法吗？

类型参数界限

```scala
trait Set[A <: Ordered[A]] {
  def incl(x: A): Set[A]
  def contains(x: A): Boolean
}
```

```scala
class EmptySet[A <: Ordered[A]] extends Set[A] {
  def contains(x: A): Boolean = false
  def incl(x: A): Set[A] = new NonEmptySet(x, new EmptySet[A], new EmptySet[A])
}
```

```scala
class NonEmptySet[A <: Ordered[A]]
        (elem: A, left: Set[A], right: Set[A]) extends Set[A] {
  def contains(x: A): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: A): Set[A] =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

- 输入类型需要限制
- Trait Ordered[A]{…}
  - 本身类的值可相互比较
- 规定输入类型的值必须可比较（它是Ordered的子类型）

类型参数界限，继续

```
case class Num(value: Double) extends Ordered[Num] {
  def compare(that: Num): Int =
    if (this.value < that.value) -1
    else if (this.value > that.value) 1
    else 0
}
```

```
val s = new EmptySet[Num].incl(Num(1.0)).incl(Num(2.0))
s.contains(Num(1.5))
```

```
val s = new EmptySet[java.io.File]
                      ^ java.io.File does not conform to type
                        parameter bound Ordered[java.io.File].
```

- 使用之前的类定义值

类型参数界限，view bounds

```
trait Set[A <% Ordered[A]] ...
class EmptySet[A <% Ordered[A]] ...
class NonEmptySet[A <% Ordered[A]] ...
```

- 如果还不是Ordered的子类？
- Int, Double, String，从Java继承来的
- View bounds，<%
- 隐式转换存在就行

一般类型的变化标识(generic types' variance)

- 问题：如何避免程序中类型不匹配？

潜在的问题？

```
val x = new Array[String](1)
val y: Array[Any] = x
y(0) = new Rational(1, 2)  // this is syntactic sugar for
                           // y.update(0, new Rational(1, 2))
```

Java：在#3行，运行时检查(run-time check)类型

Scala：在#2行，静态时编译检查；Array 在Scala里不允许变体子类（non-variant subtyping）

同变（子类化）（Co-variant subtyping）

- If T 是 S的子类, then Stack[T] 是 Stack[S]的子类
- Scala里的一般类型(generic types)缺省是无变体子类化(non-variant subtyping)
- 强制同变的符号： +
  - Class Stack[+A]
- 纯函数世界，所有类型都可能成为同变的，但是当引入可变数据时，情况就不一样了。

反变子类化 (contra-variant subtyping)

- 表示：Class Stack[-A]
- 如果T 是 S 的子类，那么Stack[S] 是 Stack[T] 的子类

问题：如何验证变化标识的合理性？

- 如果array被定义成同变的(co-variant)，怎么及时检测出这个潜在的问题？
- Scala采用了一种保守的估计方法
- 同变类型参数只应出现在同变的位置上
  - 同变的位置有：
    - 类中值的类型
    - 类中方法的返回类型
    - 其他同变类型的类型输入参数
  - 非同变位置有：
    - 类中正式方法的参数类型

问题：如何验证变化标识的合理性？

```
class Array[+A] {
  def apply(index: Int): A
  def update(index: Int, elem: A)
                                  ^ covariant type parameter A
                                    appears in contravariant position.
}
```

Update 改变状态，影响了同变子类化的合理性

```
class Stack[+A] {
  def push(x: A): Stack[A] =
                ^ covariant type parameter A
                  appears in contravariant position.
```

Stacks 是纯函数数据类型
Push 没有改变状态，但仍会被挑错，怎么办？

一般类型参数的：底界

```
class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
```

- T >: S , 类型参数T 只能是 类型S 的父类(supertypes)
- T >: S <: U
- Push in Stack，A不出现在push的参数类型位置；一个方法的类型参数的底界（这个位置是同变（co-variant）位置）。
- 不仅解决了技术问题，也通用化了push的定义。
- Push是一个多态方法(polymorphic method)
- What if push[B >: A] changed to [B<:A] ? What would happen?
- Can I push an Int value onto a String Stack?

EmptyStack的定义

```
object EmptyStack extends Stack[Nothing] { ... }

val s = EmptyStack.push("abc").push(new AnyRef())
```

- 对象不能有类型参数
- Nothing 是所有其他类型的子类；For co-variant stacks, Stack[Nothing] is a subtype of Stack[T]

Stack 的完整定义

```
abstract class Stack[+A] {
  def push[B >: A](x: B): Stack[B] = new NonEmptyStack(x, this)
  def isEmpty: Boolean
  def top: A
  def pop: Stack[A]
}
object EmptyStack extends Stack[Nothing] {
  def isEmpty = true
  def top = error("EmptyStack.top")
  def pop = error("EmptyStack.pop")
}
class NonEmptyStack[+A](elem: A, rest: Stack[A]) extends Stack[A] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

+ 一般类型实例
generic class: tuples and functions

Tuples

```
case class TwoInts(first: Int, second: Int)
def divmod(x: Int, y: Int): TwoInts = new TwoInts(x / y, x % y)

package scala
case class Tuple2[A, B](_1: A, _2: B)

def divmod(x: Int, y: Int) = new Tuple2[Int, Int](x / y, x % y)
```

类型参数可忽略

```
val xy = divmod(x, y)
println("quotient: " + xy._1 + ", rest: " + xy._2)

divmod(x, y) match {
  case Tuple2(n, d) =>
    println("quotient: " + n + ", rest: " + d)
}
```

- Tuple2: 含有两个值； Tuplen(): n个值
- 可直接用（x/y, x % y）

Tuples

```
def divmod(x: Int, y: Int): (Int, Int) = (x / y, x % y)
```

```
divmod(x, y) match {
  case (n, d) => println("quotient: " + n + ", rest: " + d)
}
```

函数(Functions)

- Scala, a functional language; functions are first-class **values**
- Also a object-oriented language; every value is an **object**.
- Functions are objects.

```
package scala
trait Function1[-A, +B] {
  def apply(x: A): B
}
```

- (T1, …, Tn) => S 缩写 Functionn[T1, …, Tn, S]
- f(x) shorthand for f.apply(x)

函数(Functions)

```scala
    val f: (AnyRef => Int) = x => x.hashCode()
    val g: (String => Int) = f

//     val g: (String => Int) = x => x.length()
//     val f: (AnyRef => Int) = g
```

- 反变类型参数的应用(contra-variant type parameter)
- Function subtyping is contra-variant in its argument type whereas co-variant in its result type.
- S’ => T’
- S => T

函数的对象本质

```scala
val plus1: (Int => Int)  =  (x: Int) => x + 1
plus1(2)
```

通常的函数使用

```scala
val plus1: Function1[Int, Int] = new Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
plus1.apply(2)
```

实体化一个抽象类？行吗？

面向对象代码；**New Function1** 构建了匿名类，实现了**apply**方法；Function1是抽象类

```scala
val plus1: Function1[Int, Int] = {
  class Local extends Function1[Int, Int] {
    def apply(x: Int): Int = x + 1
  }
  new Local: Function1[Int, Int]
}
plus1.apply(2)
```

使用命名的类扩展**Function1**

类型参数的子类变化控制的本质用意

- 子类值可以被赋给父类(变量); 反之不行。