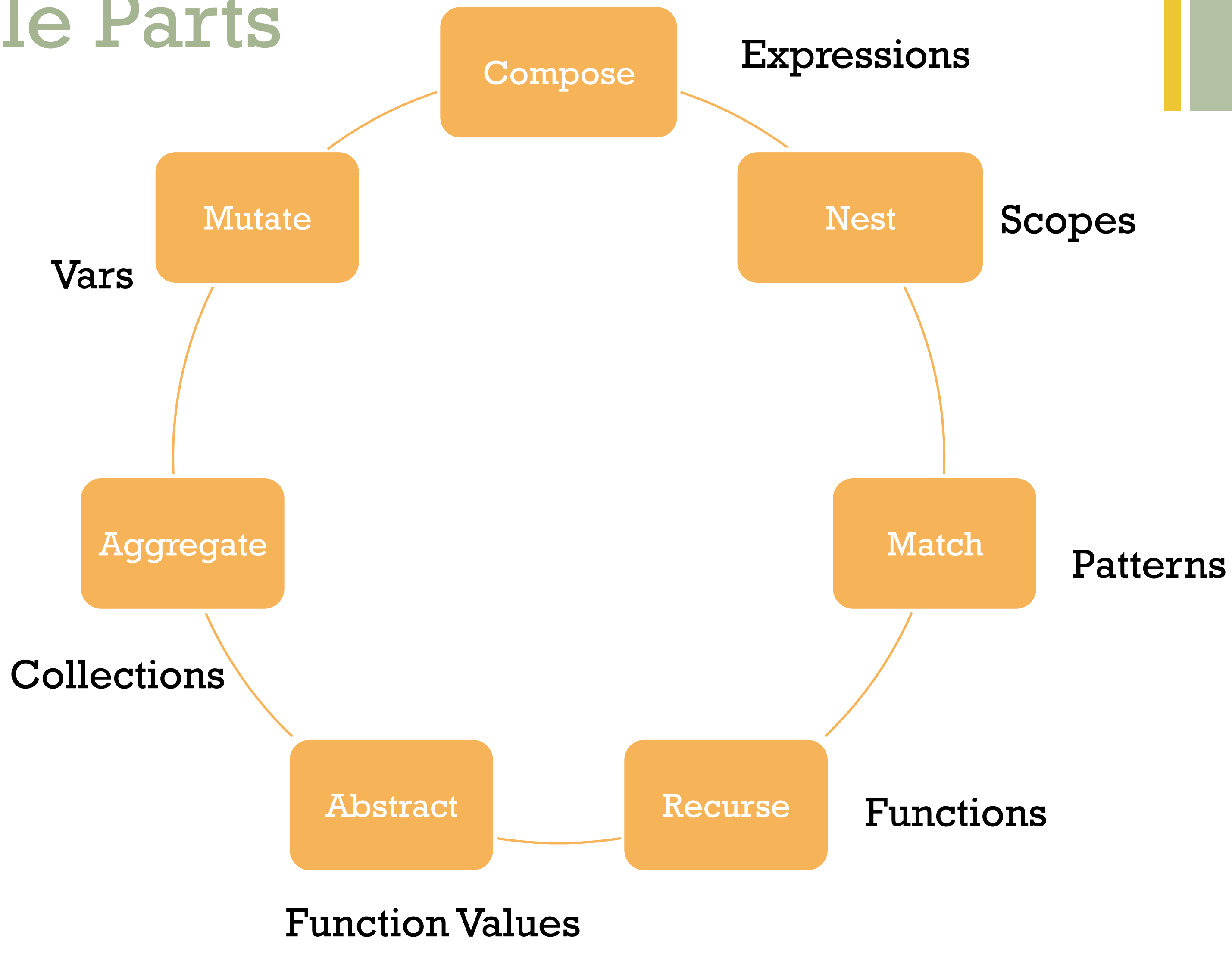
今天的内容

- Scala 语言的基本构件（继续）
- Scala 语言基础
- Scala 举例
  - 函数的应用

Scala 语言的基本构件（继续）

齐琦
海南大学

From Fundamental Actions to Simple Parts



Modules（模块）

Modules can take a large number of forms

- A function
- An object
- A class
- An actor
- A stream transform
- A microservice

Modular programming is putting the focus on how modules can be combined, not so much what they do.

In Scala, modules talk about values as well as types.

Features For Modular Programming

1. Our Vocabulary: Rich types with static checking and functional semantics
   - gives us the domains of discourse,
   - gives us the means to guarantee encapsulation,
   - see: "On the Criteria for Decomposing Systems into Modules" (David Parnas, 1972).

2. Start with Objects — atomic modules

3. Parameterize with Classes — templates to create modules dynamically

4. Mix it up with Traits — mixable slices of behavior

5. Abstract By Name

Members of a class or trait can be concrete or abstract.

Example: A Graph Library

```
trait Graphs {
  type Node
  type Edge
  def pred(e: Edge): Node
  def succ(e: Edge): Node
  type Graph <: GraphSig
  def newGraph(nodes: Set[Node], edges: Set[Edge]): Graph

  trait GraphSig {
    def nodes: Set[Node]
    def edges: Set[Edge]
    def outgoing(n: Node): Set[Edge]
    def incoming(n: Node): Set[Edge]
    def sources: Set[Node]
  }
}
```

Where To Use Abstraction?

Simple rule:

- Define what you know, leave abstract what you don't.
- Works universally for values, methods, and types.

```
trait AbstractModel extends Graphs {
  class Graph(val nodes: Set[Node],
              val edges: Set[Edge]) extends GraphSig {
    def outgoing(n: Node) = edges filter (pred(_) == n)
    def incoming(n: Node) = edges filter (succ(_) == n)
    lazy val sources = nodes filter (incoming(_).isEmpty)
  }
  def newGraph(nodes: Set[Node], edges: Set[Edge]) =
    new Graph(nodes, edges)
}
```

Encapsulation(封装包囊) = Parameterization（参数设定）

Two sides of the coin:

1. Hide an implementation
2. Parameterize an abstraction

```
trait ConcreteModel extends Graphs {
  type Node = Person
  type Edge = (Person, Person)
  def succ(e: Edge) = e._1
  def pred(e: Edge) = e._2
}

class MyGraph extends AbstractModel with ConcreteModel
```

在Scala-IDE里运行这行程序。

6. Abstract By Position

Parameterize classes and traits.

- `class List[+T]  (apple is a fruit; a list of apples is a list of fruits too.)`
- `class Set[T]`
- `class Function1[-T, +R]`

- **List[Number]**
- **Set[String]**
- **Function1[String, Int]**

Variance expressed by +/- annotations

A good way to explain variance is by mapping to abstract types.

Modelling Parameterized Types

```
class Set[T] { ... }        →  class Set { type $T }

Set[String]                 →  Set { type $T = String }


class List[+T] { ... }      →  class List { type $T }
List[Number]                →  List { type $T <: Number }
```

Parameters(参数) → Abstract members（抽象成员）

Arguments（参数实体化）→ Refinements （进一步明确）

7. Keep Boilerplate Implicit

Implicit parameters are a rather simple concept

But they are surprisingly versatile(多用途的)!

Can represent a *typeclass*:

- `def min(x: A, b: A)(implicit cmp: Ordering[A]): A`

Implicit Parameters

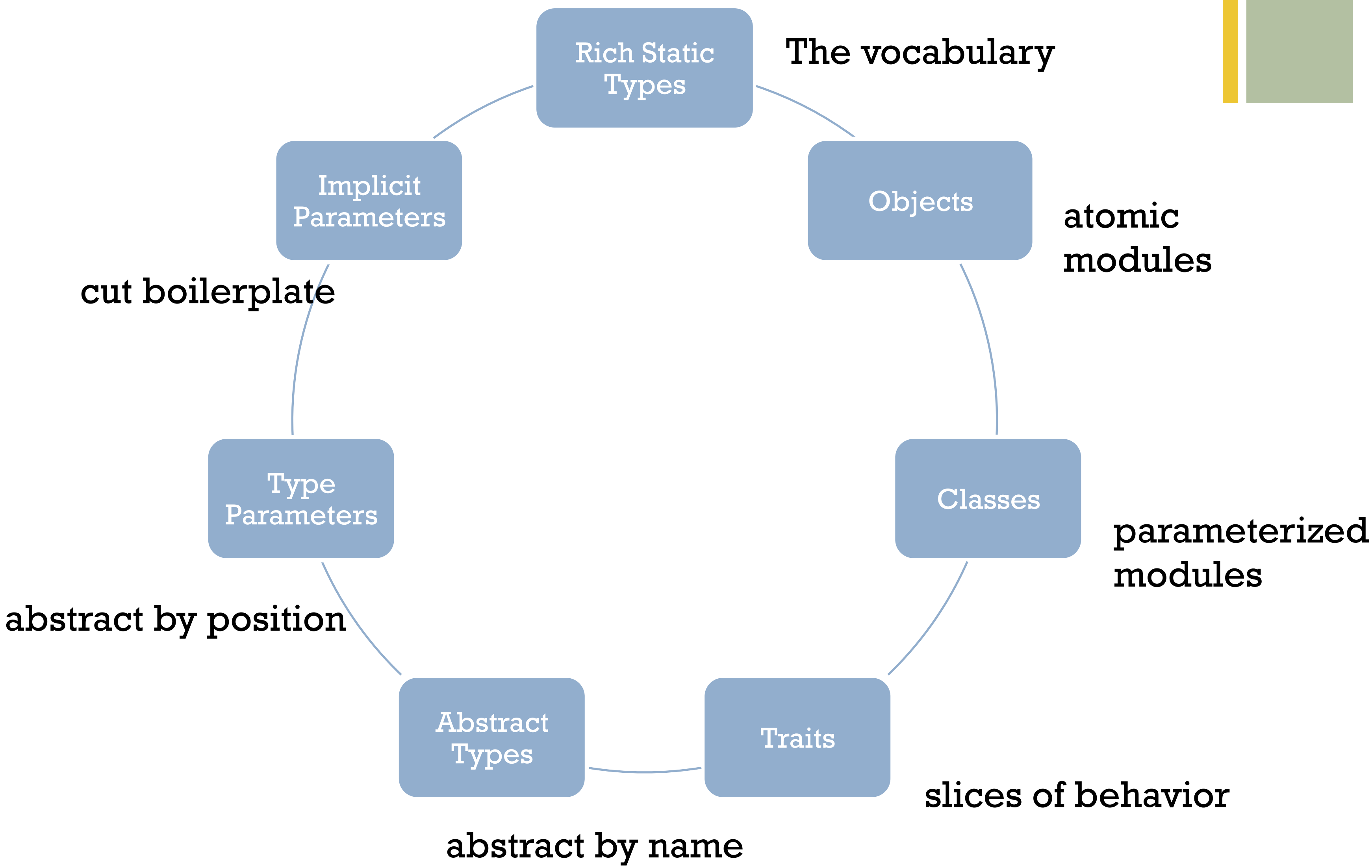
Can represent a *context*:

- **def typed(tree: untpd.Tree, expected: Type)(implicit ctx: Context): Type**
- **def compile(cmdLine: String) (implicit defaultOptions: List[String]): Unit**

Can represent a *capability*:

- **def accessProfile(id: CustomerId) (implicit admin: AdminRights): Info**

Module Parts



程序展示

Scala List

```
val nums = (1 to 10).toList

val total = nums.
  filter(x => x % 2 == 0).
  map(x => x * x).
  foldLeft(0)((a, b) => a + b)

val allByAllSum = nums.
  flatMap (n1 => nums.map (n2 => n1 * n2)).
  foldLeft(0)((a, b) => a + b)
```

程序展示

Scala List Using For

```
val nums = (1 to 10).toList

val totalf = (for {
   n <- nums if n % 2 == 0
} yield n * n).sum

val allByAllSumf = (for {
   n1 <- nums
   n2 <- nums
} yield n1 * n2).sum
```

程序展示

Scala Future Using For

```scala
val usdQuote = future { connection.getCurrentValue(USD) }
val chfQuote = future { connection.getCurrentValue(CHF) }

val purchase = for {
    usd <- usdQuote
    chf <- chfQuote
    if isProfitable(usd, chf)
} yield connection.buy(amount, chf)

purchase onSuccess {
    case _ => println("Purchased " + amount + " CHF")
}
```

+ 程序展示

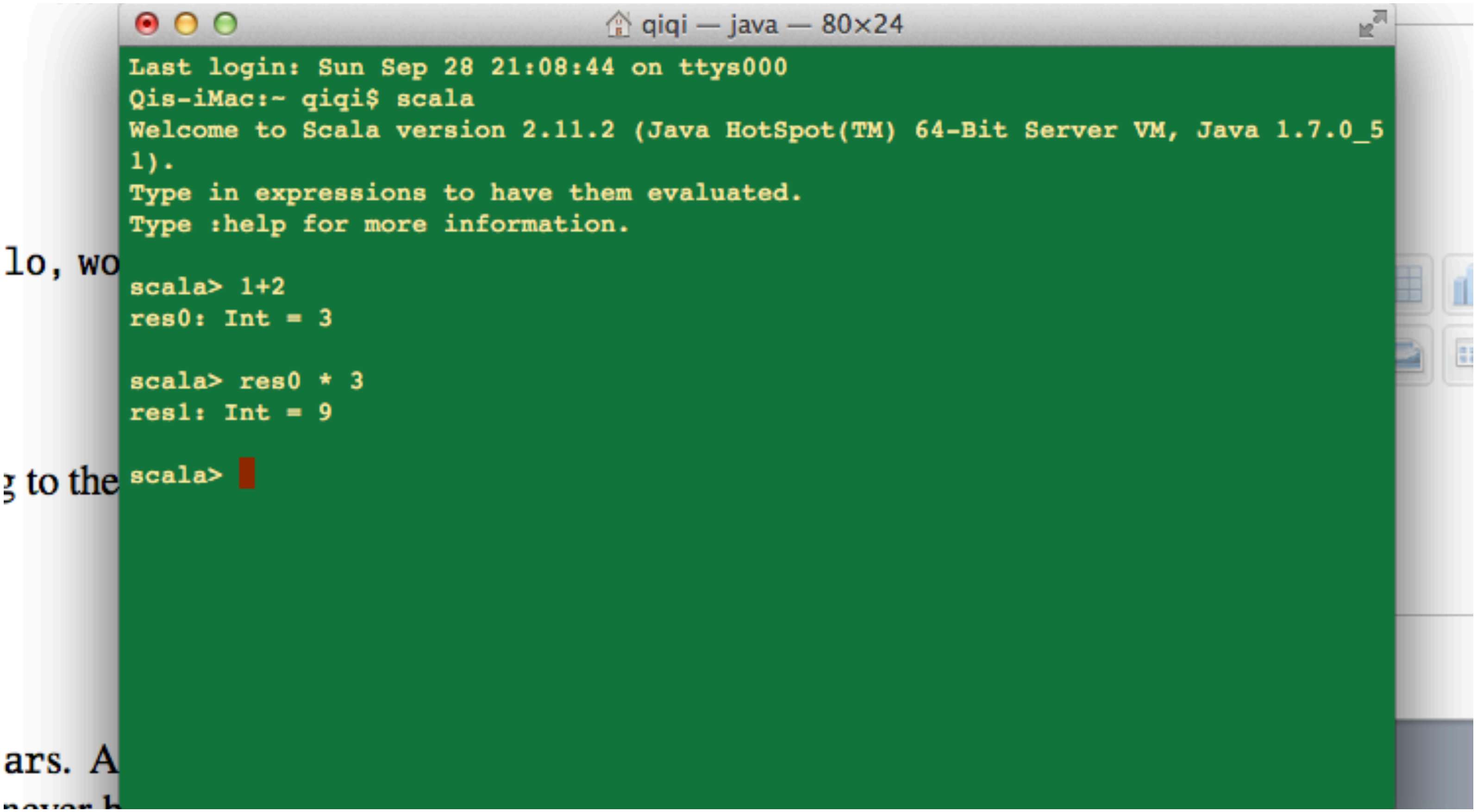
Scala Try Using For

```
import scala.util.{Try, Success, Failure}

 def divide: Try[Int] = {
    val dividend = Try(Console.readLine("Enter an Int that you'd like to divide:\n").toInt)
    val divisor = Try(Console.readLine("Enter an Int that you'd like to divide by:\n").toInt)

    val problem = for {
       x <- dividend
       y <- divisor
    } yield x/y

    problem match {
       case Success(v) =>
          println("Result of " + dividend.get + "/"+ divisor.get +" is: " + v)
          Success(v)
       case Failure(e) =>
          println("You must've divided by zero or entered something that's not an Int. Try again!")
          println("Info from the exception: " + e.getMessage)
          divide
     }
}
```

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

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

```
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)
```

好像a,b有点累赘

```
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)
```

高阶函数举例，继续 Currying

```
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
}
```

```
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)
```

def f (args$_1$) ... (args$_n$) = E

一个Curried 函数定义

def f (args$_1$) ... (args$_{n-1}$) = { **def** g (args$_n$) = E ; g }

def f (args$_1$) ... (args$_{n-1}$) = (args$_n$) => E

def f = (args$_1$) => ... => (args$_n$) => E

函数类型：右结合律

T_1 => T_2 => T_3 is equivalent to T_1 => (T_2 => 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
  1.4142135623746899
  1.4142135623746899
```

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

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