Scala

a short introduction

based on M.Odersky, L.Spoon, B.Venners
Programming in Scala

Scala – Yet another programming language

- developed at the
  Swiss Federal Institute of Technology Lausanne
- by M. Odersy, M. Zenger and others
- statically typed programming language
- runs on the .Net- and Java-platform
- Conventional syntax similar to Java and C#
- unifies object-oriented and functional programming
- is a compiled and a scripting language

Martin Odersky at LinkedIn Tech Talk speaking on SCALA June 5, 2009;
© LindaPoengPhotography

"I can honestly say if someone had shown me the Programming in Scala book by by Martin Odersky, Lex Spoon & Bill Venners back in 2003 I'd probably have never created Groovy."
James Strachan

Programming and Scripting

```
package package1

object Hallo {
        def main(args: Array[String]): Unit = {
                println("Hello World");
        }
}
```

Scala programming

```
scala> var dict = Map("Hugo"->4711, "Karla"->4712)

dict: scala.collection.immutable.Map[java.lang.String,Int] = Map(Hugo -> 4711, Karla -> 4712)

scala> dict += ("Klausi"->2211)

scala> dict("Karla")

res2: Int = 4712

scala>
```

Scala scripting

Scripting

Script: a file ending in an expression

```
#!/bin/bash
exec ~/Scala/bin/scala "$0" "$@"
!#

println("Hello this is scala !\n Your arguments: ")

args.foreach(println)
```

./scala-script.scala blub blubber

→

```
Hello this is scala !
 Your arguments:
blub
blubber
```

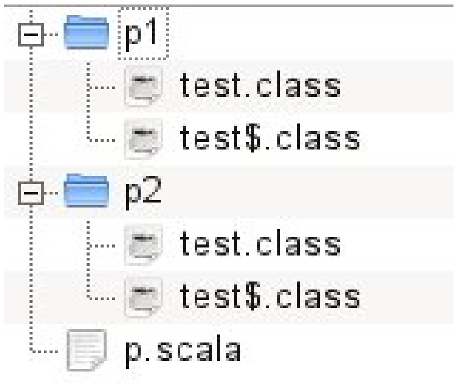
Compiling

- Runnable a scala application: Object (singleton) with main-method
- Scala compilation unit: file containing definitions
- Packages: packages are nested name spaces (like in C++, C#)

```
package p1 {
  object test extends Application {
      println("p1.test")
    }
  }
package p2 {
  object test extends Application {
      println("p2.test")
  }
}
```

p.scala

scalac p.scala



scala p1.test

p1.test

Compiling

- **Application trait**
  mixin of main-Method
- **Java package notation**
  as syntactic sugar

```
object MyApp extends Application {
  for (i <- 1 to 10) {
    println(i)
  }
}
```

Application trait

```
package package1

object App extends Application {
  println("this is App")
}
```

syntactic ⇓ sugar for

```
package package1 {
  object App extends Application {
    println("this is App")
  }
}
```

Compiling

- **Nested packages are really nested**

```
package package1

object UseAClass extends Application{
  (new AClass()).f()
}
```

import statements relate to their actual position in the package hierarchy

```
package package1

import package11.helpers_of_p1.HelperClass

class AClass {
  def f() {
    new helpers_of_p1.HelperClass().helper();
  }
}
```

```
package package11 {
  object O {
    def g() = {
      println("with a litte help of my friends");
    }
  }

  package helpers_of_p1 {
    class HelperClass {
      def helper () = {
           O.g();
      }
    }
  }
}
```

Objects and classes

- Scala is a pure OO language
- Object definition: singleton class

```
class Vector(xp:Int, yp:Int) {
  def x = xp
  def y = yp
  override def toString() = {
        "(" + x + ", " + y + ")"
  }
  def +(v:Vector) : Vector = {
       new Vector(x +  v.x, y + v.y)
  }
}
```

a simple class
and its usage:

```
println(
    new Vector(1,2) + (new Vector(3,4))
);
```

```
import scala.collection.mutable.HashMap

object Dict {
  var entries = new HashMap[String, Int]

  def enter(name : String, number : Int) : Unit = {
    entries += (name -> number)
  }

  def search(name : String) : Int = {
    return entries(name)
  }
}
```

a simple object
and its usage:

```
def main(args: Array[String])
         : Unit = {
    Dict.enter("Karla", 4711)
    println(Dict.search("Karla"))
}
```

Class and Companion Object

All dictionaries have the same emergency number:
Java: static class-member
Scala: Companion Object

– **Modeling classes with static and dynamic attributes**

```
import scala.collection.mutable.HashMap

class Dict {
  private var entries = new HashMap[String, Int]

  def enter(name : String, number : Int) : Unit = {
    entries += (name -> number)
  }

  def search(name : String) : Int = {
    if (name == "Emergency") {
         return Dict.emenergencyNumber
    } else {
         return entries(name)
     }
  }
}

object Dict {
  private var emenergencyNumber = 1100

  def setEmergencyNumber(number : Int) : Unit = {
    emenergencyNumber = number
  }
}
```

```
println(myDict2.search("Emergency"))
Dict.setEmergencyNumber(1234)
println(myDict1.search("Emergency"))
```

1100
1234

"static" members in ***companion object***

Objects and classes

- **Class / Object Members:**
  - **def** : method
  - **var**: field (- reference)
  - **val**: constant field (- reference)
- **All values are objects, all operations are method calls**

  x + x ~ x.+(y)
- **Paramerless methods**

```
class Celsius {
  private var d: Int = 0;
  def degree: Int = d;
  def warm: Boolean = if (d > 25) true else false
}
```

uniform access principle
(use only for reading the mutable state of an object)

- **Field access = getter / setter call (implicitly defined, may be redefined)**

```
class Celsius {
  private var d: Int = 0;
  def degree: Int = d;
  def degree_=(x: Int)  =  {
      if (x >= -273) d = x
  }
}
```

```
object Degree {
def main(args: Array[String]): Unit = {
       val c = new Celsius();
       c.degree = -5000
       println (c.degree)
    }
}
```

→ 0

Class example : Rational numbers

```
package rational

class Rational(n: Int, d: Int) {
  require(d != 0)

  private val g = gcd(n, d)
  val numer = n/g
  val denom = d/g

  def this(n: Int) = this(n, 1)

  def + (that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )

  def * (that: Rational): Rational =
    new Rational(numer * that.numer, denom * that.denom)

  override def toString = numer + "/" + denom

  private def gcd(a: Int, b:Int): Int =
    if (a == b) a else gcd(max(a,b) - min(a,b),  min(a,b))

  private def max(a: Int, b:Int): Int =
    if (a > b) a else b

  private def min(a: Int, b:Int): Int =
    if (a > b) b else a
}
```

```
package rational

object RatApp extends Application {
  val x = new Rational(1)
  val y = new Rational(1, 4)
  println(x+y)
}
```

```
5/4
```

Classes

- Classes may be **abstract**
  abstract classes may declare (instead of define) methods
- No **interface**s
  but traits
- **Inheritance** roughly as in Java
  - however: *methods <u>and</u> fields may be overridden*
  - however: *fields and methods with same name in the same scope are <u>not</u> allowed*
  - however: override *is a modifier that <u>has</u> to be used (except when overriding abstract methods)*
  - <u>final</u> *methods and classes*
  - *polymorphism / dynamic binding*
  - *inheritance induced subtype hierarchy*
- **Overloading** as in Java
  however: *error message if there is no best matching method for given arguments*

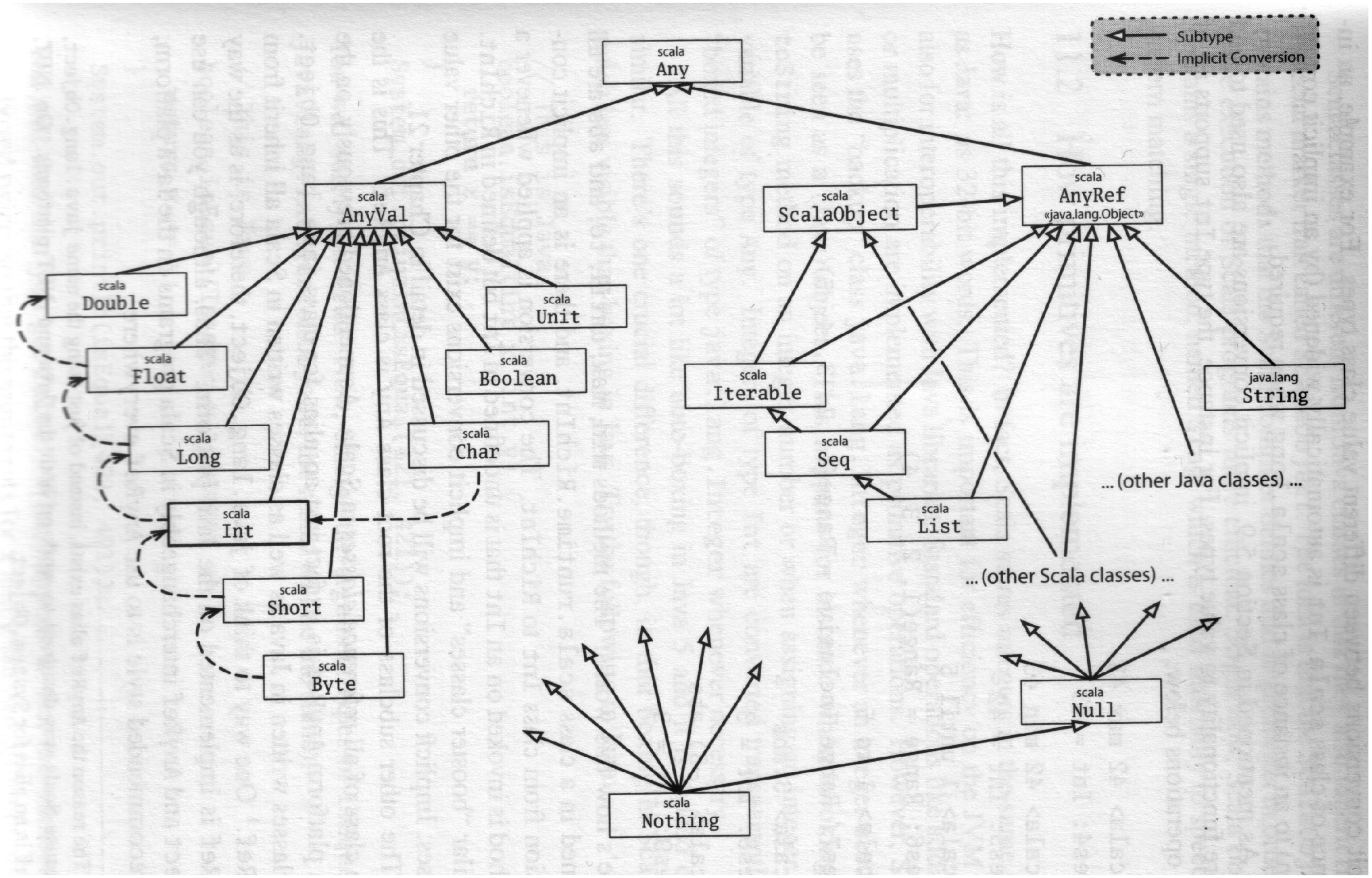
Classes

– **Transparent autoboxing**
  - == *is an alias for equals (* ==.equals(equals) *)*

– **Bottom Types**
  - **Null :** **type of the null-pointer**
  - **Nothing:** **subtype of any other type**
    - **type without values**
    - **example:**

```
def error(msg: String): Nothing = {
    throw new RuntimeException(msg);
}
```

Class hierarchy



M. Odersky, L. Spoon, B. Venners: Programming in Scala, p.207

Traits

- mixins as traits in Scala
- Example:

```java
public interface Hooter {
    void hoot();
}
```

```java
public class RubberElephant
            extends Toy
            implements Hooter {
   @Override
   public void hoot() {
        System.out.println("hoot hoot");
   }
}
```

```java
public class RubberDog
            extends Toy
            implements Hooter {

  @Override
  public void hoot() {
        System.out.println("hoot hoot");
  }
}
```

Java

```scala
trait Hooter {
   def hoot() : Unit = {
          println("hoot hoot")
   }
}
```

```scala
class RubberDog extends Toy
                with Hooter {

}
```

```scala
class RubberElephant extends Toy
                     with Hooter {

}
```

Scala

Traits

- Traits may declare abstract and concrete methods
- Traits may define fields
- super-call is dynamically bound
- Mixed-in methods from traits may be overridden
- Example:

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
  private val g = gcd(n, d)
  val numer = n/g
  val denom = d/g

  def compare(that: Rational) =
    (this.numer * that.denom) - (that.numer * this.denom)

  // etc ....

}
```

```
trait Ordered[A] {
  def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0
  def >  (that: A): Boolean = (this compare that) >  0
  def <= (that: A): Boolean = (this compare that) <= 0
  def >= (that: A): Boolean = (this compare that) >= 0
  def compareTo(that: A): Int = compare(that)
}
```

concrete
abstract

Ordered-trait from the scala library

Multiple inheritance

- Classes may extend one class and several traits
- Example:

```
trait Animal {
  def eat()
}
class Milk

trait MilkGiver {
  def milk() : Milk
}

class Cow extends Animal with MilkGiver {
  override def eat() { println ("Cow eating")}
  override def milk() : Milk = new Milk
}
```

```
trait Animal {
  def eat() { println ("Animal eating")}
}

class Milk

trait MilkGiver {
  def eat() { println ("Milk animal eating")}
}

class Cow extends Animal with MilkGiver
```

conflicting definitions lead to compiler errors

Case classes

- **case class: class defined with modifier case**
- **Example:**

  Trait that defines 4 case classes each with an empty body

```
trait Expr {
  case class Var(name: String) extends Expr
  case class Number(num: Int) extends Expr
  case class UnOp(operator: String,
                  arg: Expr) extends Expr
  case class BinOp(operator: String,
                   left: Expr,
                   right: Expr) extends Expr
}
```

- **conveniences of cases classes:**
  - implicitly defined: factory method, fields, equals, hashCode , toString

```
package caseClass
case class Person(name: String)
object Example extends Application {
  val hugo = Person("Hugo")
  if (hugo == Person("Hugo")) { println(hugo) }
}
```

Pattern matching

- match expressions: generalization of switch statements
- Example:

```
trait Expr {
  case class Var(name: String) extends Expr
  case class Number(num: Int) extends Expr
  case class UnOp(operator: String, arg: Expr) extends Expr
  case class BinOp(operator: String, left: Expr, right: Expr) extends Expr

  def simplify(expr: Expr) : Expr =
    expr match {
      case UnOp("-", UnOp("-", e))          => e
      case BinOp("+", e, Number(0))         => e
      case BinOp("*", e, Number(1))         => e
      case _                                => expr
    }
}
```

```
object Main extends Application {
  class AnExpr extends Expr;
  val e = new AnExpr
  println(e.simplify(
       e.UnOp("-",
          e.UnOp("-",
              e.BinOp("+",
                  e.Number(1), e.Number(0))))))
}
```

→

```
BinOp(+,Number(1),Number(0))
```

Pattern matching

- match expressions in variable definitions
- Example:

```
object Main extends Application {
class AnExpr extends Expr;
    val e = new AnExpr
    val binExpr = e.BinOp("+", e.Number(1), e.Number(0))

    val e.BinOp(op, l, r) = binExpr

    println(op)
    println(l)
    println(r)
}
```

defines **op**, **l** and **r**

```
+
Number(1)
Number(0)
```

Enumeration

- Enumerations are part of the standard library, not part of the language
- Scala enumerations do not support the semantic richness of Java enums
- simple Example:

```
object Day extends Enumeration {
  val MON = Value("Monday")
  val TUE = Value("Tuesday")
  val WED = Value("Wednesday")
  val THU = Value("Thursday")
  val FRI = Value("Friday")
  val SAT = Value("Saturday")
  val SUN = Value("Sunday")
}

for (day <- Day)
  println(day)
```

Value is a method of Enumeration that produces unique values

Functions

- Scala supports local functions
  - Functions may
    be defined anywhere even in in methods or functions
- Parameters may be passed by-name
  - Example

indicates lazy evaluation

```
object Lazy extends Application {

  def cond (ifE: Boolean, thenE: => Int, elseE: => Int) : Int = {
    if (ifE) {
      thenE
    } else {
      elseE
    }
  }

  def fac (n : Int) : Int = {
    cond( n==0, 1, fac(n-1) * n)
  }

  println(fac(5))

}
```

infinite recursion avoided

Functions

– **Functions are first-class**

  ▪ **Example**

f is a Int → Int function

```
object Functions extends Application {
  val f = (x: Int) => x + 1
  def map(f: (Int) => Int, l : List[Int]): List[Int] = l match {
    case List()             => List()
    case first :: rest      => f(first) :: map(f, rest)
  }
  val list = List(1,2,3,4,5,6,7,8,9,0);
  println(map(f, list))
}
```

```
List(2, 3, 4, 5, 6, 7, 8, 9, 10, 1)
```

```
println(list.map(x => x+1))
```

Note: lists have a predefined map method

Partially applied Functions

– **Functions may be supplied with partial argument lists thus creating a new a function**
  - **Example**

```
object Functions extends Application {
  val f = (x: Int, y: Int, z: Int) => x + y + z
  val g = f(1,2, _: Int)
  println(g(5))
}
```

↳ 8

Closures

- Closure: Function that refers to a context (via its free variables)
- In Scala closures capture variables (not their value) in the defining context
  - Example

```
object Functions extends Application {
  var v = 2;
  val f = (x: Int) => x + v

  class C {
    val v = 100
    def g(f: (Int)=>Int) = {
        println(f(2))
    }
  }

  val c = new C
  c.g(f)
  v = 3;
  c.g(f)
}
```

```
4
5
```

Closures

– Example

```
object Functions extends Application {

  def kringel(
        f: (Int) => Int,
        g: (Int) => Int) : (Int) => Int =
    (x: Int) => f(g(x))

  val id = (x: Int) => x

  var F = id
  List(1, 2, 3, 4, 5) foreach( i =>
    F = kringel((x: Int) => x*i, F)
  )
  println("first version:\t"+F(2))

  F = id
  var i = 1
  while (i<6) {
    F = kringel((x: Int) => x*i, F)
    i = i+1
  }
  println("second version:\t"+F(2))
}
```

```
first version:    240
second version:   15552
```

Currying

– **Currying :** λx,y.··x··y·· **=>** λx.λy.··x··y··

– **Example 1:**

```
object Functions extends Application {

  def curryfiedAdd(x: Int)(y: Int) : Int = x + y

  println(curryfiedAdd(5)(6))

  val f = curryfiedAdd(5)_

  println(f(6))
}
```

Currying

– **Example 2: Define Your own “control structure”**

Unit ~ void

```
object BoundedWhile extends Application {

  def boundedWhile(bound : Int)(ifE: => Boolean)(op : => Unit) {
    var counter = 0
    while (ifE && counter < bound) {
      op
      counter = counter + 1
    }
  }

  val loop10 = boundedWhile(10) _

  var i = 0
  loop10(i<500) {
    println("loop i = " + i)
    i = i+1
  }

}
```

```
loop i = 0
loop i = 1
loop i = 2
loop i = 3
loop i = 4
loop i = 5
loop i = 6
loop i = 7
loop i = 8
loop i = 9
```

10 loops at most

Generics

- Generic functions, classes and traits
- No raw types
- Example:

```
object Generics extends Application {

  class Pair[T](first: T, second: T) {
    val f = first
    val s = second
    override def toString(): String = "(" + f + ", " + s + ")"
  }

  def swap[T](p: Pair[T]): Pair[T] = new Pair(p.s, p.f)

  println(swap(new Pair[Int](2,3)))

}
```

Upper bounded generic Parameters

– **Restriction to upper bound** `<:`

– **Example**

```
trait Animal {
  def eat()
}

class Cow extends Animal {
  override def eat() { println ("Cow eating") }
}

class Tiger extends Animal {
  override def eat() { println ("Tiger eating") }
}

class Stable[T<: Animal](a1: T, a2: T) {
  var box1: T = a1
  var box2: T = a2
}
```

T: *bounded type parameter*

Lower bounded generic Parameters

– **Restriction to lower bound: “>:”**
– **Example 1**

generic function with two parameters, one bounded by the other

```
def cons[TE, TL >:TE] (e: TE, l: List[TL]) : List[TL] = e :: l

trait Animal {
  def eat()
}

class Cow extends Animal {
  override def eat() { println ("Cow eating")}
}

cons(new Cow, List[Animal]())
```

put a cow into an empty list of animals

```
static <TE> void cons(TE e, List<? super TE> l) {
   l.add(0, e);
}
```

```
static <TL, TE extends TL> List<TL> cons(TE e, List<TL> l) {
  l.add(0, e);
  return l;
}
```

Roughly similar Java versions, there is no “`TL super TE`” in Java so either wildcard (with capture problems) or reverse order of parameters

Lower bounded generic Parameters / Example 2

```
trait Animal {
   def eat()
 }

 class Cow extends Animal {
   override def eat() { println ("Cow eating")}
 }

 class Tiger extends Animal {
   override def eat() { println ("Tiger eating")}
 }

 case class Pair[T](x:T, y:T) {
   var f : T = x
   var s : T = y
 }

 def count[TE, TL >: TE](x: TE, p: Pair[TL]): Int = {
   if (p.f == x) (if (p.s == x) 2 else 1) else (if (p.s == x) 1 else 0)
 }

 val berta = new Cow
 val hugo  = new Tiger

 val p1: Pair[Animal]= Pair[Animal](berta, hugo)
 val p2: Pair[Tiger] = Pair[Tiger](hugo, hugo)
 val p3: Pair[Cow] = Pair[Cow](berta, berta)

 println(count(berta, p1))
 println(~~count(berta, p2)~~)
 println(count(berta, p3))
```

***Type Error:** f,s in Pair are* vars *so they may be modified (write-access) – maybe by inserting a Tiger into a pair of cows.*

Variances

– Java reminder

- Covariant subtype relation on arrays:

  `T < T'  =>  array[T] < array[T']`

  type errors are detected <u>at</u> <u>runtime</u> via `ArrayStoreException`s

- No variance on Collection-Types

  `T < T' ≠> List(T) < List(T')`

  `       ≠> List(T') < List(T)`

  induced unnecessary restrictions are removed in Java

  - at <u>program</u> <u>creation</u> <u>time</u>
  - by the <u>client-code</u> programmer
  - via wildcard-extends / wildcard-super

Variances

– **Java reminder: array example**

```
class Animal {
  void eat() { System.out.println("eat"); }
}

class Milk {}

class Cow extends Animal {
   Milk milk() { return new Milk(); }
}

class Tiger extends Animal {}

static <T> void feed(Animal[] animals) {
  for(Animal a : animals) {
    a.eat();
  }
}

...
Cow[] cows = new Cow[]{ new Cow() };
feed(cows);
```

OK

```
class Animal {
  void eat() { System.out.println("eat"); }
}

class Milk {}

class Cow extends Animal {
   Milk milk() { return new Milk(); }
}

class Tiger extends Animal {}

static <T> void feed(Animal[] animals) {
  for(Animal a : animals) {
    a.eat();
  }
  animals[0] = new Tiger();
}

...
Cow[] cows = new Cow[]{ new Cow() };
feed(cows);
```

ArrayStoreException

Variances

– Java reminder: Collection example

```
class Animal {
   void eat() { System.out.println("eat"); }
}

class Milk {}

class Cow extends Animal {
  Milk milk() { return new Milk(); }
}

class Tiger extends Animal {}

static <T> void feed(List<Animal> animals) {
   for(Animal a : animals) {
     a.eat();
   }
}

...
List<Cow> cows = new LinkedList<Cow>();
cows.add(new Cow());
feed(cows);
...
```

does not compile

```
class Animal {
   void eat() { System.out.println("eat"); }
}

class Milk {}

class Cow extends Animal {
  Milk milk() { return new Milk(); }
}

class Tiger extends Animal {}

static <T> void feed(
           List<? extends Animal> animals) {
   for(Animal a : animals) {
     a.eat();
   }
  // Type error: animals.add(new Tiger());
}

...
List<Cow> cows = new LinkedList<Cow>();
cows.add(new Cow());
feed(cows);
...
```

OK

The user of the collection solves the problem

Variances

– **The scala way: Variance annotations for generic types**

```
class Animal {
  def eat() { System.out.println("eat"); }
}

class Milk;

class Cow extends Animal {
  def milk(): Milk = new Milk()
}

class Tiger extends Animal

class MyCollection[T](ee: T) {
  val e: T = ee
}

def feed(c: MyCollection[Animal]) {
  c.e.eat()
}

val cows : MyCollection[Cow] = new MyCollection[Cow](new Cow)

feed(cows)
```

no variance

Type error:

only Animal-collections allowed here!

Variances

– **Variance annotations for generic types**

```
class Animal {
  def eat() { System.out.println("eat"); }
}

class Milk;

class Cow extends Animal {
  def milk(): Milk = new Milk()
}

class Tiger extends Animal

class MyCollection[+T](ee: T) {
  val e: T = ee
}

def feed(c: MyCollection[Animal]) {
  c.e.eat()
}

val cows : MyCollection[Cow] = new MyCollection[Cow](new Cow)

feed(cows)
```

allow covariance (annotating `class MyCollection[+T](ee: T) {`)

OK! (annotating `feed(cows)`)

MyCollection[Cow] < MyCollection[Animal]

Variances

– **Covariant types must not have modifiable components of type T**

```
class MyCollection[+T](ee: T) {
  var e: T = ee
}
```

by this, the “tiger among cows” problem is avoided

*Error: **var** not allowed here*

Either:

+:
allow covariance,
forbid write-access

```
class MyCollection[+T](ee: T) {
  val e: T = ee
}

def feed(c: MyCollection[Animal]) {
  c.e = new Tiger
}

val cows : MyCollection[Cow]
   = new MyCollection[Cow](new Cow)

feed(cows)
```

ERROR

Or:

no +:
forbid covariance,
allow write-access

```
class MyCollection[T](ee: T) {
  var e: T = ee
}

def feed(c: MyCollection[Animal]) {
  c.e = new Tiger
}

val cows : MyCollection[Cow]
   = new MyCollection[Cow](new Cow)

feed(cows)
```

ERROR

Variances

- Contra-variance annotation
- Example

contra-variance annotation

so contravariant Counters will be accepted

```
class Counter[-T](l: List[T]) {
  def count(x: T): Int = l.count(_ == x)
}

def count(c: Counter[Cow], x: Cow) {
  println(c.count(x))
}

val berta : Cow = new Cow

val animalList: List[Animal] = List(new Animal, berta, new Cow, berta)

val counter: Counter[Animal] = new Counter[Animal](animalList)

count(counter, berta);
```

in Java you would say:
`count(Counter<? super Cow> c, Cow x)`

a contravariant argument

You may look for objects of type T in lists of any supertype of T

Abstract members

- Abstract member:
  - a member that is just declared but not implemented
  - Subtypes should define them
  - E.g.: abstract methods in Java

- Abstract members in Scala
  - method declared but undefined method
  - val declared but undefined value
  - var declared but undefined variable
  - type declared but undefined type

Abstract types

assume `Grass` to be a subtype of `Food`

– Example

does not compile (in any typed oo language)

Cow is abstract and eat does not override

```
class Food

abstract class Animal {
    def eat(food: Food)
}

class Grass extends Food

class Cow extends Animal {
    override def eat(grass: Grass) { println ("Cow eats grass")}
}
```

OK with abstract type in Scala

```
class Food

abstract class Animal {
    type SuitableFood <: Food
    def eat(food: SuitableFood)
}

class Grass extends Food

class Cow extends Animal {
    type SuitableFood = Grass
    override def eat(grass: Grass) { println ("Cow eats grass")}
}
```

Abstract types / path dependable types

– path: path of object references that identify a type

```
class Food

abstract class Animal {
    type SuitableFood <: Food
    def eat(food: SuitableFood)
}

class Grass extends Food
class DogFood extends Food

class Cow extends Animal {
    type SuitableFood = Grass
    override def eat(grass: Grass) { println ("Cow eats grass")}
}

class Dog extends Animal {
    type SuitableFood = DogFood
    override def eat(dogFood: DogFood) { println ("Dog eats dog food")}
}

val berta = new Cow
val fifi = new Dog

berta.eat(new berta.SuitableFood)
 fifi.eat(new  fifi.SuitableFood)
```

path — path dependable type

Implicit conversion

- function call that the compiler inserts to avoid type errors
- `x + y` does not type check

  => compiler tries `convert(x) + y` for some function `convert`
- Conversion functions to be used by the compiler may be provided by programmers
- typical usage: wrapper for library functions

Implicit conversion

– **Example**

```
class Person(name: String) {
  val n = name
  override def toString() : String = n
}

class Pair(partner1: Person, partner2: Person) {
   override def toString() : String = partner1 + " and " + partner2
}

class Woman(name: String) extends Person(name) {
   def +(husband: Man) : Pair = new Pair(this, husband)
}

class Man(name: String) extends Person(name) {
  def +(wife: Woman) : Pair = new Pair(this, wife)
}

implicit def ManToWoman(man: Man) : Woman = new Woman(man.n+"-chen")

val klaus = new Man("Klaus")
val guido = new Man("Guido")

val karla = new Woman("Karla")

val pair1 = karla + klaus
val pair2 = klaus + guido
```

Implicit parameters

– **Parameters added by the compiler to incomplete calls**

For Expressions

- for expressions are powerful tools to solve combinatorial problems
- General form:

  for(seq) yield expression

  where

  seq is a sequence of *generators, definitions* and *filters*
- Example

```
object ForExample extends Application {
  case class Person(name: String, isFemale: Boolean, children: Person*)
  val lara =      Person("Lara",  true)
  val hugo =      Person("Hugo",  false)
  val nadja =     Person("Nadja", true,  lara, hugo)
  val karla =     Person("Karla", true,  nadja)
  val emil =      Person("Emil",  false, lara, hugo)
  val persons = List(lara, hugo, nadja, karla, emil)

  val motherAndChild = for(p <- persons;
                           if p.isFemale;
                           c <- p.children ) yield (p.name, c.name)

  println(motherAndChild)
}
```

```
List((Nadja,Lara), (Nadja,Hugo), (Karla,Nadja))
```

For Expressions

- Generator
  - generates values
  - first element in a sequence has to be a generator
  - there may be several generators
  - form: `pat <- expression`
- Definition
  - binds a value to one or more names
  - form: `pat = expression`
- Filter
  - drops from iteration all values
    for which *`expression`* evaluates to false
  - form `if expression`

Divide and conquer algorithm with for

- Example with two generators: generate all permutations of a list
- Note: generators have <u>nothing</u> in common with generators and the keyword yield in e.g. Python or C#

```
def ins[T](x: T, l: List[T]) : List[List[T]] = l match {
  case List()       => List(List(x))
  case head :: tail => (x :: l) :: (for(p <- ins(x, tail)) yield head :: p)
}

def perm[T](l: List[T]) : List[List[T]] = l match {
  case  List()       => List(List())
  case  head :: tail => for(p <- perm(tail); i <- ins(head, p)) yield i
}
```

Query with for

– print affiliations of researchers who wrote books on “Concurrency”

```
case class Book(title: String, authors: String*)
case class Researcher(name: String, affiliation: String)

val books = List(
        Book("Java Concurrency in Practice",
             "B. Goetz", "T. Peierls", "J. Bloch", "J. Bowbeer", "D. Holmes", "D. Lea"),
        Book("Concurrent Programming", "G. Andrews"),
        Book("Groovy in Action", "D. König"),
        Book("Introduction to Distributed Algorithms", "G. Tel"),
        Book("The Art of Multiprocessor Programming", "M. Herlihy", "N. Shavit"),
        Book("Programming in Scala", "M. Odersky", "L. Spoon", "B. Venners"))
val researchers = List(
        Researcher("D. Lea", "New York University at Oswego"),
        Researcher("J. Bloch", "Google"),
        Researcher("T. Peierls", "BoxPop.biz"),
        Researcher("B. Goetz", "Sun Microsystems"),
        Researcher("D. Holmes", "Sun Microsystems"),
        Researcher("J. Bowbeer", "MIT"),
        Researcher("G. Tel", "Utrecht University"),
        Researcher("M. Herlihy", "Brown University"),
        Researcher("N. Shavit", "Tel-Aviv University"),
        Researcher("D. König", "Canoo Engineering AG"),
        Researcher("M. Odersky", "University of Lausanne"),
        Researcher("L. Spoon", "Google"),
        Researcher("B. Venners", "Artima Inc."))

(for(b <- books;
     t = b.title; if (t.indexOf("Concurr") >=0);
     a <- b.authors;
     r <- researchers;
     if (a == r.name) yield r.affiliation
 ).foreach( println )
```

XML literals

- Scala supports xml literals
- xml literals have type `Elem` a may contain embedded expressions

```
val xmlCode = <a> blubber  <b> bla bla</b>  </a>
println(xmlCode)
```

xml-val with infered type

```
import scala.xml.Elem

object SomeXml extends Application {
  val xmlCode : Elem = <a> blubber  <b> bla bla</b>  </a>
  println(xmlCode)
}
```

xml-val with explicit type

```
object SomeXml extends Application {
  def f(s: String) : List[String] = List(s, s, s)
  val xmlCode : Elem = <a> blubber  <b> {f("bla")} </b>  </a>
  println(xmlCode)
}
```

include embedded expression

Serialization

– **Transform data to XML**

```
object Gender extends Enumeration {
    val male, female = Value
}

case class Person(firstName: String,
                  surName: String,
                  gender: Gender.Value) {
    def toXML =
      <person male={if (gender==Gender.male) "true" else "false" }>
         <name first={firstName} sur={surName}/>
      </person>
  }

  val p = Person("John", "Doe", Gender.male)

  println(p.toXML)
```

```
<person male="true">
         <name sur="Doe" first="John"></name>
      </person>
```

XPATH support

– **extract parts using XPath similar expressions**

```
case class Person(firstName: String,
                  surName: String,
                  gender: Gender.Value) {
    def toXML =
      <person male={if (gender==Gender.male) "true" else "false" }>
         <name first={firstName} sur={surName}/>
      </person>
  }

  val p = Person("John", "Doe", Gender.male)

  val x = p.toXML

  println(x)
  println(x \"name"\"@first")
```

```
<person male="true">
         <name sur="Doe" first="John"></name>
      </person>
John
```

Matching support

match whitespaces (that appear as text elements)

```
val p = Person("Duck", "Daisy", Gender.female)

val x = p.toXML

println(x)

println(
  x match {
    case <person>{_}{n}{_}</person>
         => "it s a " +
                 (if ((x \ "@male") == "true") "male" else "female") +
                 " person"
    case _ => "?"
  }
)
```

```
<person male="false">
          <name sur="Daisy" first="Duck"></name>
       </person>
it s a female person
```

Matching + For

```
val catalog = <catalog>
       <cd>
          <artist>Ludwig van Beethoven</artist>
           <title>Sinfonie nr.5 c-Moll op. 67</title>
       </cd>
       <book>
           <author>Edward Gibbon</author>
           <title>The History of the Decline and Fall of the Roman Empire</title>
       </book>
       <cd>
           <artist>Richard Wagner</artist>
           <title>Götterdämmerung</title>
       </cd>
       <book>
           <author>Oswald Spengler</author>
           <title>Der Untergang des Abendlandes</title>
       </book>
    </catalog>

catalog match {
   case <catalog>{items @ _*}</catalog> =>
      for (book @ <book>{_*}</book>  <- items)
          println("processing: " + (book \ "title").text)
}
```

```
processing: The History of the Decline and Fall of the Roman Empire
processing: Der Untergang des Abendlandes
```

Threads

– **Scala supports common Java threads**

```
object ThreadEx extends Application {

  val threads  =
    for(i <- 1 to 3) yield new Thread {
                  override def run () = {
                           println("Hello there. I'm thread "+i)
                           Thread.sleep(1000)
                           println("Thread "+i+" says good bye")
                  }
    }

  threads.foreach(_.start())
}
```

```
Hello there. I'm thread 1
Hello there. I'm thread 2
Hello there. I'm thread 3
Thread 1 says good bye
Thread 2 says good bye
Thread 3 says good bye
```

Actors

- **Actors:**
  - **Event-driven concurrency**
  - **Objects interact by message exchange**
- **Actors in Scala**
  - **Library implementation on top of threads**
  - **Scala follows actor concept in Erlang**
    - Functional language
    - For real-time applications
    - Developed at Ericsson
  - **Scala:**
    - Prefer actors, avoid Java threads and synchronization
    - However: combine all mechanisms if you need to (and know what you are doing)

Actor Example

- each actor runs as Java thread
- send an receive are connected via mail boxes
- receive with case expressions (messages that do not match are ignored)
- send as `actor ! msg`

```
import scala.actors._
import scala.actors.Actor._

object ThreadEx extends Application {
  val a = actor {
    while(true) {
      receive {
        case msg =>
           println("actor received message " + msg)
      }
    }
  }

  a ! "hello actor!"
}
```

React

– **react : receive that does not return**

– **save threads: no need to preserve current stack**

```
import scala.actors._
import scala.actors.Actor._

object ThreadEx extends Application {
  val a = actor {
    while(true) {
      receive {
        case msg => println("actor a received message " + msg)
      }
      println("actor a has processed a message")
    }
  }

  val b = actor {
    react {
      case msg => println("actor b received message " + msg)
    }
    // never reached:
    println("actor b has processed a message")
  }

  a ! "hello actor a!"
  b ! "hello actor b!"

}
```

```
actor a received message hello actor a!
actor a has processed a message
actor b received message hello actor b!
```

Combinator Parsing

- **A Grammar:**

  expr ::= term { "+" term | "-" term }

  term ::= factor { "*" factor | "/" factor }

  factor :: floatingPointNumber | "(" expr ")"

- **A parser:**

```
import scala.util.parsing.combinator._

object SimpleArithmeticExpression extends Application {

  class Arith extends JavaTokenParsers {
    def expr:   Parser[Any] = term~rep("+"~term | "-"~term)
    def term:   Parser[Any] = factor~rep("*"~factor | "/"~factor)
    def factor: Parser[Any] = floatingPointNumber | "("~expr~")"
  }

  object ExprParser extends Arith {
    def parse(s: String) = parseAll(expr, s)
  }

  println(ExprParser.parse("2 * (3 + 4)"))
}
```

```
[1.12] parsed: ((2~List((*~(((~((3~List())~List((+~(4~List())))))~)))))~List())
```

Combinator Parsing

- Trait `JavaTokenParsers`
  - Machinery for writing parsers
  - some pre-build parsers for common cases
    - identifiers
    - number literals
    - string literals
- Trait `RegexParsers`
  - Machinery for parsing ('scanning') regular expressions
  - More low level
  - Example

```
object MyId extends RegexParsers {
    val myIdent: Parser[Any] = """[a-zA-Z]+""".r
    def parse(s: String) = parseAll(myIdent, s)
}

println(MyId.parse("hallo"))
```

```
[1.6] parsed: hallo
```

- Trait `JavaTokenParsers`
  - Machinery for parsing ('scanning') Java Tokens
  - ~ Java class Scanner

Combinator Parsing

- **Example: “scanner” + “parser”**

```
import scala.util.parsing.combinator._

object SimpleArithmeticExpression extends Application {

  object Arith extends JavaTokenParsers {
    val MyId = """[a-zA-Z]+""".r
    def expr: Parser[Any] = term~rep("+"~term | "-"~term)
    def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
    def factor: Parser[Any] = floatingPointNumber | MyId | "("~expr~")"

    def parse(s: String) = parseAll(expr, s)
  }


  val str = "hugo * (3 + 4) - 12.4 * carla"

  println(Arith.parse(str))

}
```

AST Construction and Evaluation - 1

```
import scala.util.parsing.combinator._

object SimpleArithmeticExpression extends Application {
  abstract class Tree() {
    def eval() : Double
  }

  case class Literal(value: String) extends Tree {
    val doubleVal: Double = value.toDouble
    def eval() : Double = doubleVal
  }

  case class Id(spelling: String) extends Tree {
    def eval() : Double = if (spelling == "carla") 10 else 1
  }

  case class BinOp(op: String, left: Tree, right: Tree) extends Tree {
    def eval() : Double = op match {
      case "+" => left.eval + right.eval
      case "-" => left.eval - right.eval
      case "*" => left.eval * right.eval
      case "/" => left.eval / right.eval
    }
  }
```

AST Construction and Evaluation -2

```
object Arith extends JavaTokenParsers {

  def processTF(first: Tree, rest: List[~[String, Tree]]): Tree =
      rest match {
        case List() => first
            case op~t :: List() => BinOp(op, first, t)
            case op~t :: op2~t2 :: tail
                => BinOp(op, first,  BinOp(op2, t, processTF(t2, tail)))
      }

  val MyId = """[a-zA-Z]+""".r

  def expr: Parser[Tree] = term~rep("+"~term | "-"~term) ^^ {
      case first~rest => processTF(first, rest)
                                          }
  def term: Parser[Tree] = factor~rep("*"~factor | "/"~factor) ^^ {
      case first~rest => processTF(first, rest)
                                          }
  def factor: Parser[Tree] = floatingPointNumber ^^ Literal |
      MyId ^^ Id |
      "("~expr~")" ^^ {case "("~t~")"  => t}
  def parse(s: String) : Tree = parse(expr, s).get

}
val str = "hugo * (3 + 4) - 12.4 * carla"
println(Arith.parse(str).eval)
}
```

-117.0

Scala GUIs

– **based on Swing, with some simplification**

```
import scala.swing._
import scala.swing.event._

object SimpleGuiAp extends SimpleGUIApplication {
  def top = new MainFrame {
    title = "GUI"
    val button = new Button { text = "Click me!"}
    val label = new Label { text = "No Clicks yet" }

    contents = new BoxPanel(Orientation.Vertical) {
      contents += button
      contents += label
      border = Swing.EmptyBorder(30, 30, 10, 30)
    }

    listenTo(button)

    var nClicks = 0

    reactions += {
      case ButtonClicked(b) =>
        nClicks = nClicks+1
        label.text = "Clicks: " + nClicks
    }
  }
}
```

Packages

- Based on Java package system with some additional features
  - packages are name-spaces
  - packages may be nested
- Notion 1: Java Version
  - syntactical sugar for the more general notation
- Notion 2: Name-space Version ~ C++/C#
  - more general basic notation
- Example
  - Three ways to say

    class *C belongs to package* `p11`

    and `p11` *is nested within* `p1`

```
package p1 {
  package p11 {
    class C {}
  }
}
```

basic notation

```
package p1

package p11 {
    class C {}
}
```

mixed notation

```
package p1.p11

class C {}
```

Java-style notation

Packages and Imports

Packages "truly nest":

- import ***relative to the actual position*** in package hierarchy (instead always relative to root as in Java)
- packages in a inner scope ***hide*** packages of the same name in an outer scope

```
package outerP {
  class C
  package inner1P {
    class C
    package inner2P {
      class D {
        val c1 : C = new C
        val c2 : outerP.C = new outerP.C
      }
    }
  }
}
```

inner and outer packages

```
package outerP {
  import inner1P.D
  import inner2P.E

  class C {
    val d: D = new D
    val e: E = new E
  }
  package inner1P {
    import inner2P.E
    class D {
      val c: C = new C
      val e: E = new E
    }
  }
  package inner2P {
    import inner1P.D
    class E {
      val c: C = new C
      val d: D = new D
    }
  }
}
```

import from actual position

Imports

- **imports as in Java (with '_' instead of '*')**
- **local imports**
  imports may appear anywhere
- **import of objects**
  objects may be imported
- **import of packages**
  packages themselves may be imported
- **renaming and hiding of imported members**
  imported names may be changed or omitted

```
package pack1 {
   class C
}

package pack2 {
    package inner2P {
      class D {
        import pack1.C
        val c : C = new C
      }
    }
}
```

local import

```
package pack1 {
  package pack1_1 {
      class C
  }
}

package pack2 {
    import pack1.pack1_1

    package inner2P {
    class D {
      val c : pack1_1.C = new pack1_1.C
    }
   }
}
```

package import

```
package pack1 {
  class C
  class D
}

package pack2 {
   import pack1.{C => C1, D => _}

    package inner2P {
    class D {
      val c : C1 = new C1
    }
   }
}
```

package import with renaming and hiding

Access Modifiers

- **private** as in Java
- **protected** only visible in subclasses (no package-local visibility)
- **public** members without access modifier are public
- **Defined scope of protection**
  - **labeled private** visibility is restricted to some package scope
  - **labeled protected** visibility is restricted to subclasses in some package scope
  - **Object private** visibility is restricted to the object

```
package pack1 {
  import pack2.pack2_1.E
  class C {
    val ~~e:E = new E~~
  }
}

package pack2 {
  class D {
    import pack2_1.E
    val e: E = new E
  }
  package pack2_1 {
    private[pack2] class E
  }
}
```

Visibility of E *is restricted to package* pack2 *and its sub-packages*

```
class C {
  private[this] var my_i = 0;
  def inc() { my_i = my_i+1 }
  def accessOther(other: C) {
      ~~other.my_i = 0~~
  }
}
```

Visibility of my_i *is restricted to the object it belongs to*

Visibility and Companion Objects

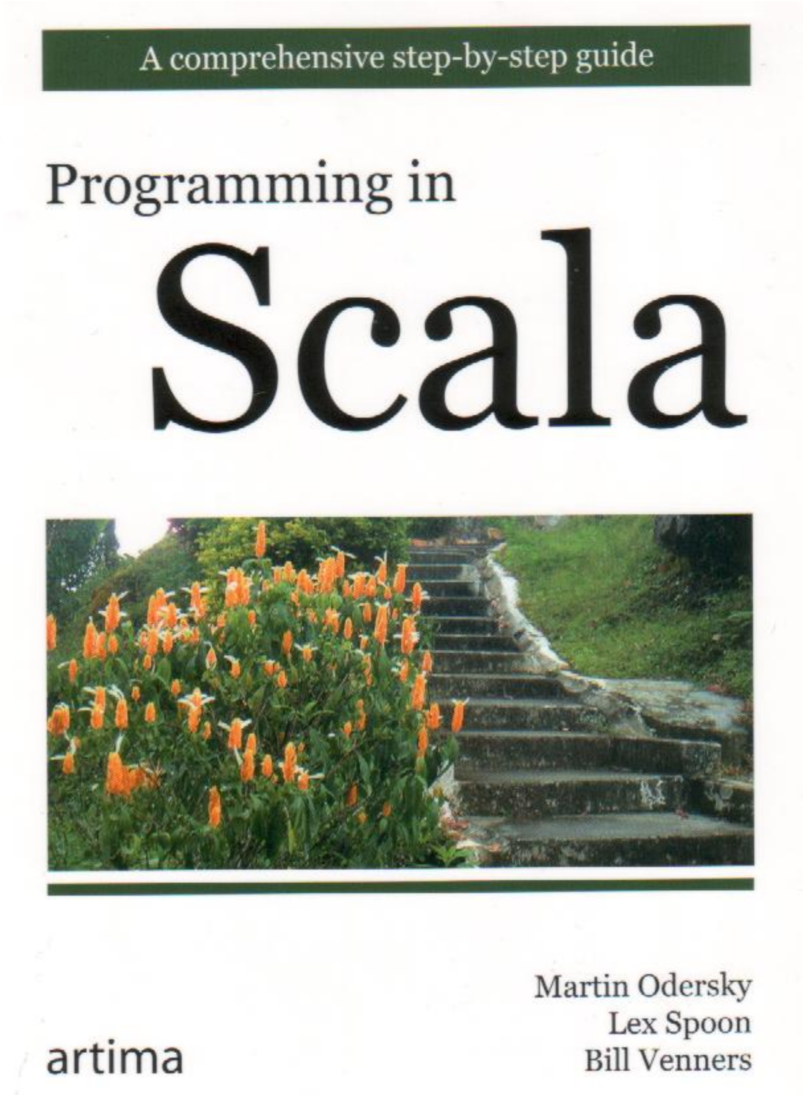
- remember : companion object instead of static members in Scala
- class and companion object share visibility rights

```
class C {
  private var local_i = 0;
  def inc() {
      local_i = local_i+1
      C.common_i = C.common_i + 1
  }
}

object C {
  private var common_i = 0;
  val ac: C = new C
  ac.local_i = 100
}
```

A class and its companion object sharing private members

Reference


A comprehensive step-by-step guide
Programming in
Scala
Martin Odersky
Lex Spoon
Bill Venners
artima