Imperative Programming 2:
Introduction

Hongseok Yang
University of Oxford

Programming courses so far

- Michaelmas 2012 -- Functional programming.
  - Recursion, list, higher-order function, etc.
- Hillary 2013 -- Imperative programming 1.
  - Iteration, array, searching, sorting, invariant, etc.
- Emphasised skills for writing small tricky programs.

Imperative programming 2

- Emphasises skills for writing well-modularised software components, such as libraries.
- Main topics:
  - Basic object-oriented programming.
  - Effective combination of multiple programming paradigms (FP, IP, OOP).
  - Advanced Scala features.

List library in Scala

- Very powerful.
- With Scala lists, you can do almost all the things that you did with lists in Haskell.

Implementation of Scala List

```
sealed abstract class List[+A] extends AbstractSeq[A]
                                  with LinearSeq[A]
                                  with Product
                                  with GenericTraversableTemplate[A, List]
                                  with LinearSeqOptimized[A, List[A]] {
  override def companion: GenericCompanion[List] = List
```

```
import scala.collection.{Iterable, Traversable, Seq, IndexedSeq}

def isEmpty: Boolean
def head: A
def tail: List[A]
```

Implementation of Scala List

```
sealed abstract class List[+A] extends AbstractSeq[A]
                               with LinearSeq[A]
                               with Product
                               with GenericTraversableTemplate[A, List]
                               with LinearSeqOptimized[A, List[A]] {
  override def companion: GenericCompanion[List] = List
```

```
import scala.collection.{Iterable, Traversable, Seq, IndexedSeq}

def isEmpty: Boolean
def head: A
def tail: List[A]
```

1. Inheritance and mixin.

Implementation of Scala List

```
sealed abstract class List[+A] extends AbstractSeq[A]
                                  with LinearSeq[A]
                                  with Product
                                  with GenericTraversableTemplate[A, List]
                                  with LinearSeqOptimized[A, List[A]] {
  override def companion: GenericCompanion[List] = List
```

```
import scala.collection.{Iterable, Traversable, Seq, IndexedSeq}

def isEmpty: Boolean
def head: A
def tail: List[A]
```

1. Inheritance and mixin.
2. Type parameter and variance.

Implementation of Scala List

```
sealed abstract class List[+A] extends AbstractSeq[A]
                                  with LinearSeq[A]
                                  with Product
                                  with GenericTraversableTemplate[A, List]
                                  with LinearSeqOptimized[A, List[A]] {
  override def companion: GenericCompanion[List] = List
```

```
import scala.collection.{Iterable, Traversable, Seq, IndexedSeq}
```

```
def isEmpty: Boolean
def head: A
def tail: List[A]
```

1. Inheritance and mixin.
2. Type parameter and variance.
3. Abstract members.

Implementation of Scala List

```
def +:[B >: A, That](elem: B)(implicit bf: CanBuildFrom[List[A], B, That])
```

1. Inheritance and mixin.
2. Type parameter and variance.
3. Abstract members.
4. Implicit parameter.

Implementation of Scala List

```
@inline final override def foreach[U](f: A => U) {
  var these = this
  while (!these.isEmpty) {
    f(these.head)
    these = these.tail
  }
}
```

1. Inheritance and mixin.
2. Type parameter and variance.
3. Abstract members.
4. Implicit parameter.
5. High-order function.

Implementation of Scala List

```
@inline final override def foreach[U](f: A => U) {
  var these = this
  while (!these.isEmpty) {
    f(these.head)
    these = these.tail
  }
}
```

Advanced features

OO features

Multi paradigms

1. Inheritance and mixin.
2. Type parameter and variance.
3. Abstract members.
4. Implicit parameter.
5. High-order function.

Scala features and library design

- The List library uses the features in the previous slides to achieve the following goals:

  1. The library is easy to use.

  2. It works well with the Scala type system.

  3. No code duplication in its implementation.

- In IP2, we will study how to achieve these goals using OO, multi-paradigms and advanced features of Scala.

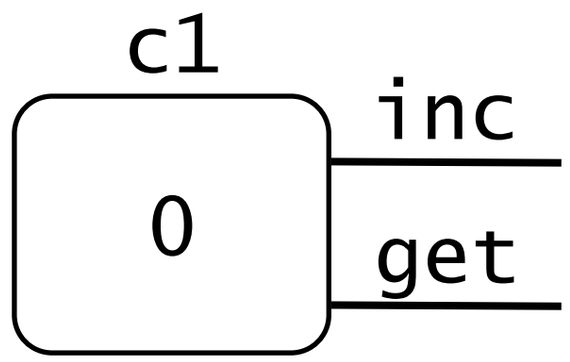
Review of Scala

Object

- An object is an encapsulation of a state. It provides methods for accessing the state.

```
class Counter {
  private var n = 0
  def inc() { n += 1 }
  def get: Int = n
}

val c1 = new Counter
```

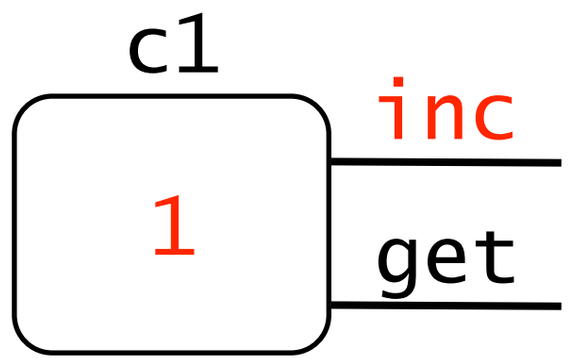


Object

- An object is an encapsulation of a state. It provides methods for accessing the state.

```
class Counter {
  private var n = 0
  def inc() { n += 1 }
  def get: Int = n
}

val c1 = new Counter
c1.inc()
```

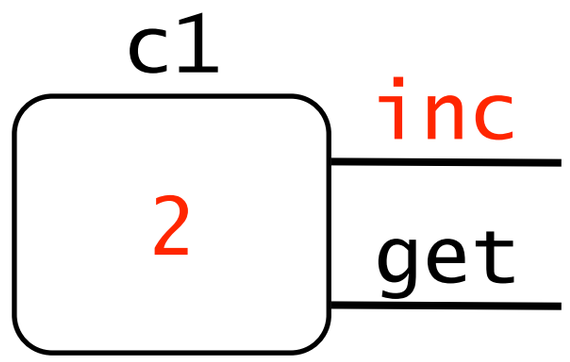


Object

- An object is an encapsulation of a state. It provides methods for accessing the state.

```
class Counter {
  private var n = 0
  def inc() { n += 1 }
  def get: Int = n
}

val c1 = new Counter
c1.inc()
c1.inc()
```

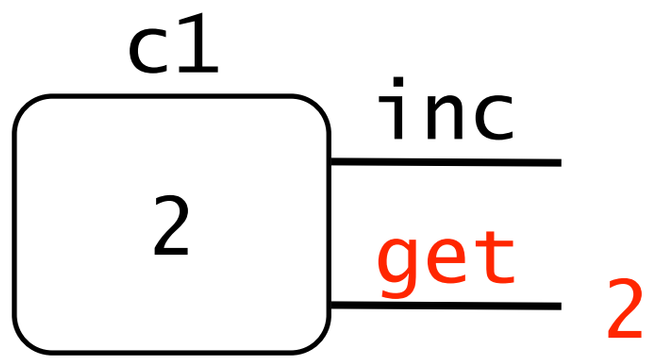


Object

- An object is an encapsulation of a state. It provides methods for accessing the state.

```
class Counter {
  private var n = 0
  def inc() { n += 1 }
  def get: Int = n
}

val c1 = new Counter
c1.inc()
c1.inc()
println(c1.get)
```



Scala is a pure OO language

- In Scala, every computation is done by a method call on an object.
- [Q] Rewrite the following phrases to the standard form of a method call `o.meth(..)`:

(1) `3 + 4`

(2) `x.f = 3`

(3) `println(3)`

(4) `List(4,5)`

Scala is a pure OO language

- In Scala, every computation is done by a method call on an object.
- [Q] Rewrite the following phrases to the standard form of a method call `o.meth(..)`:

(1) `3 + 4` ===> `3.+(4)`

(2) `x.f = 3` ===> `x.f_=(3)`

(3) `println(3)` ===> `Predef.println(3)`

(4) `List(4,5)` ===> `List.apply(4,5)`

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala

```
(0 to 100).toList.filter(x => x*x*x - 27 == 0)
```

```
(0.to(100)).toList.filter(x => x.*(x).*(x).-(27).==(0))
```

with explicit method calls

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala

```
(0 to 100).toList.filter(x => x*x*x - 27 == 0)
```

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala

```
(0 to 100).toList.filter(x => x*x*x - 27 == 0)
```

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala

```
(0 to 100).toList.filter(x => x*x*x - 27 == 0)
```

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala

```
(0 to 100).toList.filter(x => x*x*x - 27 == 0)
```

Scala fully supports FP

- A rule of thumb -- In Scala, you can do most of the things that you did with Haskell.

Haskell

```
filter
  (\x -> x*x*x - 27 == 0)
  [0..100]
```

Scala

```
(0 to 100).toList.filter(x => x*x*x - 27 == 0)
```

[Q] Wait. Surely, functions are not objects. Does this contradict Scala being a pure OO language?

Functions in Scala

- Functions are objects with a method `apply`.
- A function application is expanded to the call of this `apply` method by the Scala compiler.

Functions in Scala

- Functions are objects with a method `apply`.
- A function application is expanded to the call of this `apply` method by the Scala compiler.

```
val f = ((x:Int) => x*x*x - 27 == 0)
f(3)
```

```
val f = ((x:Int) => x*x*x - 27 == 0)
f.apply(3)
```

Other FP features of Scala

- Scala supports pattern matching.

```
def len(l : List[Any]): Int = l match {
  case Nil => 0
  case _::rest => 1+len(rest)
}
```

- Scala supports the list comprehension of Haskell via the for and yield constructs.

Haskell

```
[x | x <- [0..100], mod x 3 == 0]
```

Scala

```
for(x <- (0 to 100).toList; if x % 3 == 0)
  yield x
```

Resources

- Textbook.
- Scala API: http://www.scala-lang.org/api
- Source code of Scala compiler and library: https://github.com/scala/scala/tree/master/src

Scala is a bit of a chameleon. It makes many programming tasks refreshingly easy and at the same time contains some pretty intricate constructs that allow experts to design truly advanced typesafe libraries.

Martin Odersky