Programming in Scala

Based on the book
“Programming in Scala Third Edition”

Agenda

- Chapter XVIII - Mutable Objects
  - Functional vs Mutable
  - Recognizing mutability
  - Property
- Chapter XIX - Type Parameterization
  - Type paramter
  - Variance
  - Bounds
  - Mixed variance

Chapter XVIII - Mutable Objects

Functional vs Mutable

- Functional - always returns the same result

```
val cs = List('a', 'b', 'c')
cs.head
```

- Mutable - The result depends on previous operation.

```
var i = 0
i = i + 1
```

Mutability is not apparent

- Var object
- Var definition in the type
- Var definition through inheritance
- Forwarding method calls to mutable objects

- But it can still be made functional

Property

- Every non-private var member implicitly defines a getter and a setter method in addition to a reassignable field
- The field is always `private[this]`
- The methods get the same visibility as the original var
- The methods can be defined directly
- Getter:

```
def hour: Int = h
```

- Setter:

```
def hour_=(x: Int) = { h = x }
```

Bonus question

What is the difference between

```
var celsius: Float = _
```

and

```
var celsius: Float
```

Chapter XIX - Type Parameterization

Generic classes and traits

- Dictionary
  - Type
  - Type parameter
  - Type constructor
  - Subtype, Supertype
  - etc.
- Example - Set class
  - Set[T]
  - Set[String], Set[Int], Set[AnyRef]
  - Set[String] is a subtype of Set[AnyRef]

Example - Functional Queue - 1

- Functional - does not change its contents
- First In First Out data structure
- 3 operations
  - head - returns the first element in the queue
  - tail - returns a queue without the first element
  - enqueue - returns a new queue with the given element appended to the end

```
trait Queue[T] {
    def head: T
    def tail: Queue[T]
    def enqueue(x: T): Queue[T]
}
```

- Efficiency (operation in constant time)?

Example - Functional Queue - 2

- Idea: combine the operations
- Leading list : head will use it
- Trailing list : enqueue appends to it
- Mirror method : if the leading list is empty, returns a new reversed queue
- New primary constructor :

```
Class Queue[T] (
    private val leading: List[T],
    private val trailing: List[T]
) { … }
```

Information hiding

- A constructor can be hidden by making it private

```
Class Queue[T] private ( … ){ … }
```

- Even the primary constructor
  - Auxiliary constructor may be needed
  - Factory method
  - “Neat” way: an object with same name as the class, with apply method

```
object Queue {
    def apply[T](xs: T*) = new Queue[T](xs.toList, Nil)
}
```

- Or use private classes

Variance annotations

- Type parameter + subtyping = ?
- Is `Queue[String]` a subtype of `Queue[AnyRef]` ?
- Solution: prefixing type parameter with variance annotionation

- T : invariant
  - Queue[String] and Queue[AnyRef] are not interchangeable
- +T : covariant
  - Queue[String] is a subtype of Queue[AnyRef]
- -T : contravariant
  - Queue[String] is a supertype of Queue[AnyRef]

Bonus question

```
val c1 = new Cell[String](“abc”)
var c2: Cell[Any] = c1
c2.set(1)
val s: String = c1.get

//JAVA
String[] a1 = { “abc” };
Object[] a2 = a1;
a2[0] = new Integer(17);
String s = a1[0];
//AVAJ
```

Bonus question

```
class Queue[+T] {
    def enqueue(x: T) = …
}
class StrangeIntQueue extends Queue[Int] {
    override def enqueue(x: Int) = {
        println(math.sqrt(x))
        super.enqueue(x)
    }
}
val x: Queue[Any] = new StrangeIntQueue
x.enqueue(“abc”)
```

Bounds

- Supertype and subtype relationships are reflexive
- Lower bound: `U >: T`
  - T is the lower bound for U
  - U is required to be a supertype of T
  - Solution to the previous problem

```
def enqueue[U >: T](x: U) = new Queue[U]( … )
```

- Upper bound: `U <: T`
  - U is required to be a subtype of T

Contravariance

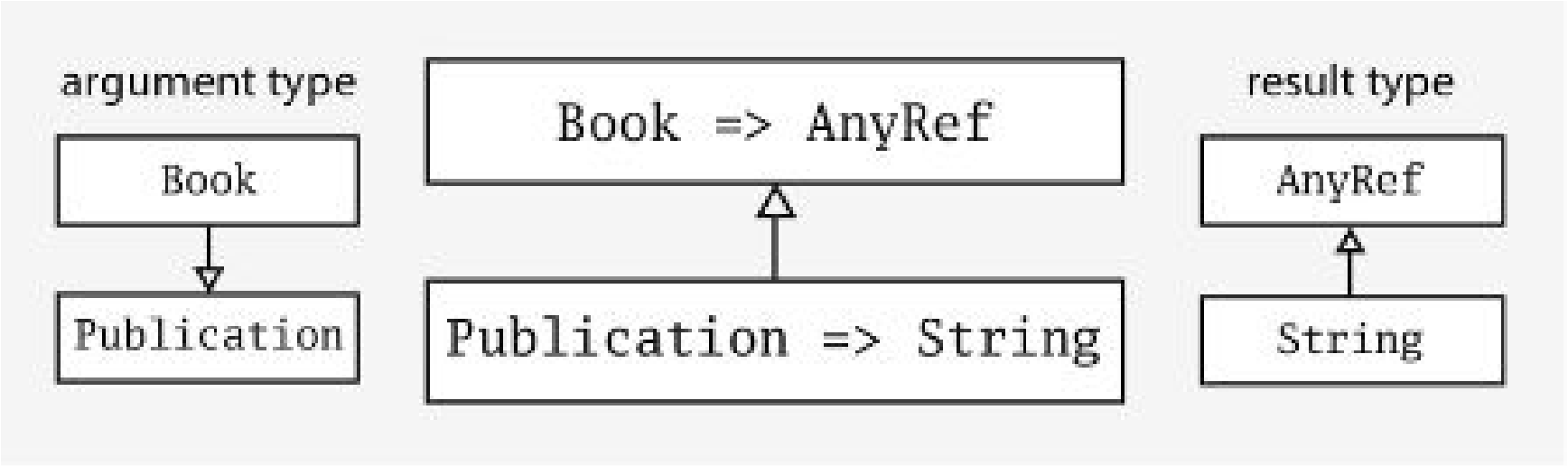
```
trait OutputChannel[-T] {
    def write(x: T)
}

val c: OutputChannel[String] = new OutputChannel[AnyRef]
c.write(“asd”)
```

- *Liskov Substitution Principle :*
  - If you can pass T where U is required, it’s safe to assume T is subtype of U

Mixed covariance and contravariance

- A prominent example is the Scala's function traits
- `A => B` becomes `Function1[A,B]`



Mixed variance - Library example

```
class Publication(val title: String)
class Book(titles: String) extends Publication(title)
object Library {  val books: Set[Book] = Set{…}
    def printBookList(info: Book => AnyRef) = {
        for (book <- books) println(info(book)) }
}

object Customer extends App {
    def getTitle(p: Publication): String = p.title
    Library.printBookList(getTitle)
}
```

+1 Homework

- Implement a **bubble sort** algorithm on a **List** of **Books**(author: String, title: String) using the **Ordered** trait!