Lecture 5
Type Parameterisation, Pattern Matching & Variance

12.3.2023

Iflaah Salman

Type Parameterisation

- We are already familiar with type parameterisation
  - in the collection types where
    - A List can be parameterised to hold only Strings, Person objects or integers.
- We now, look at how we can create our own types that are
  - generic and
  - can be parameterised by concrete type.
- The generic class and type parameterisation
  - provide a powerful construct for creating type-safe and reusable code.

We can create generic Classes and Traits as both can be instantiated with a concrete type. But, not generic Objects as we cannot instantiate an Object (this is handled for us by Scala at runtime).

Type Parameterisation

By convention, the letter 'T' is used to indicate a type to specify.

The immutable Queue Class with parameters into the primary constructor

- head passed into Queue is of type T and the tail is a list of type T.
- The class creates a new Queue when a value is enqueued.
- Returns a new Queue (with the current head removed) in response to a dequeue.
- The class provides a peek method to see what is currently at the head of the queue.

```scala
package com.jjh.scala.collection

class Queue[T](val head: T, val tail: List[T]) {

  def enqueue(x: T) = new Queue(x, head :: tail)

  def peek = head

  def dequeue = new Queue(tail.head, tail.tail)

  override def toString =
    s"$head: ${(tail mkString ",")}"

      }
```

This class is used in exactly the same way as one of the built-in collection classes.

LUT University

Type Parameterisation

```scala
object QueueTest extends App {
  val q =
    new Queue[String]("John",
                      List("Denise", "Phoebe"))
  println("q = " + q)
  println("q.peek = " + q.peek)
  val q2 = q.dequeue
  println("q2 = " + q2)
  println("q = " + q)
}
```

```scala
package com.jjh.scala.collection

class Queue[T](val head: T, val tail: List[T]) {

  def enqueue(x: T) = new Queue(x, head :: tail)

  def peek = head

  def dequeue = new Queue(tail.head, tail.tail)

  override def toString =
    s"$head: ${(tail mkString ",")}"

    }
```

- 'T' has been replaced within the instance of the Queue class by the concrete type String.
- The type of the parameter X in the enqueue method is String as is the type returned by the dequeue method.
- We can actually create a Queue parameterised by Int, Boolean, Double or any user-defined type such as Person.

LUT University

Trait

- Traits can play in developing reusable behaviour that simplifies the development of new types.

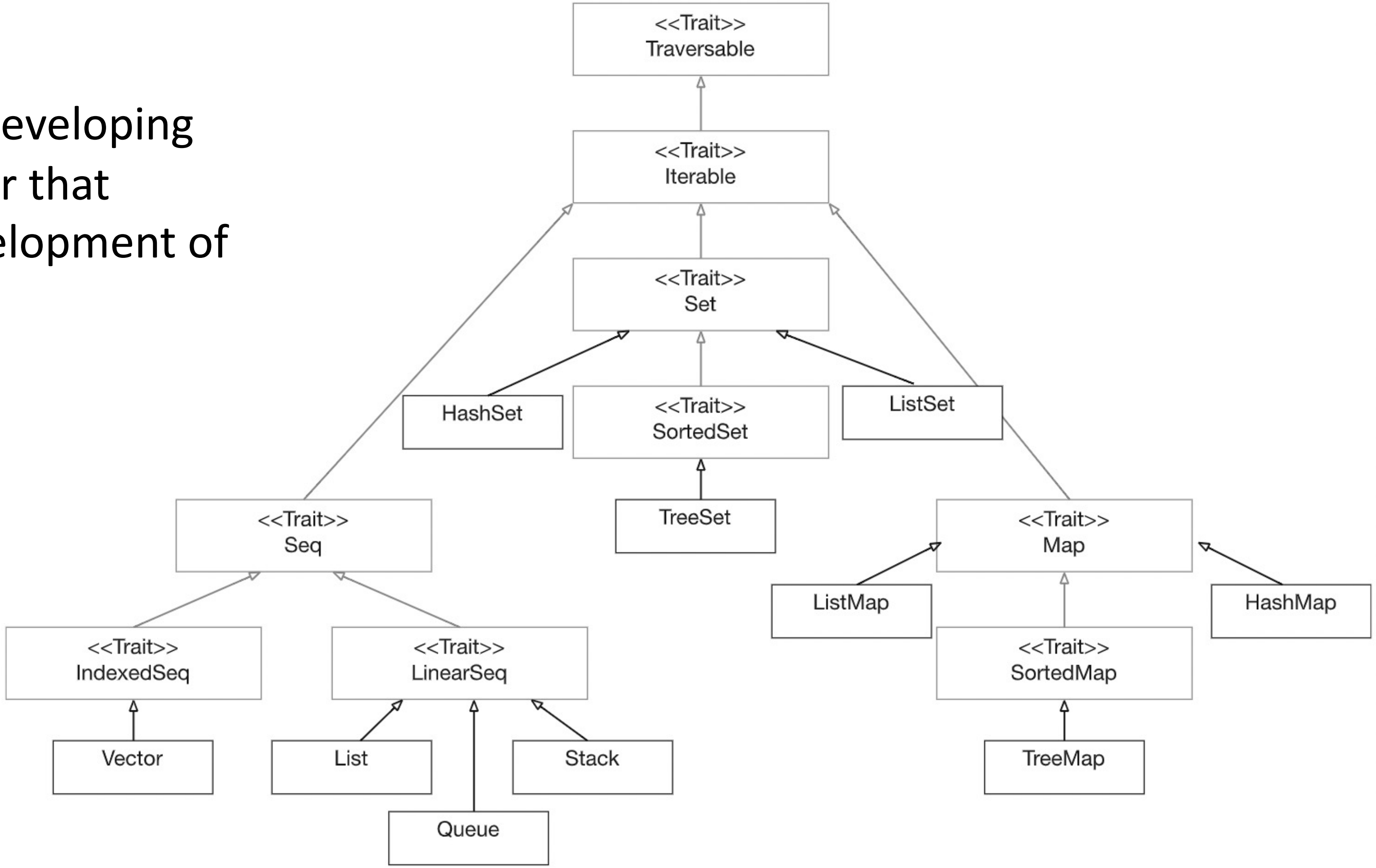


Fig. 27.1 Key classes and traits in the scala.collection.immutable package

LUT University

Variance

If we have a Set that can hold Persons (Set[Person] - parent type), should it be able to hold references to instances of the class Employee (Set[Employee]– subtype) if it is a subtype of Person?

This situation is referred to as variance and within a type system, in a programming language, a type rule is:

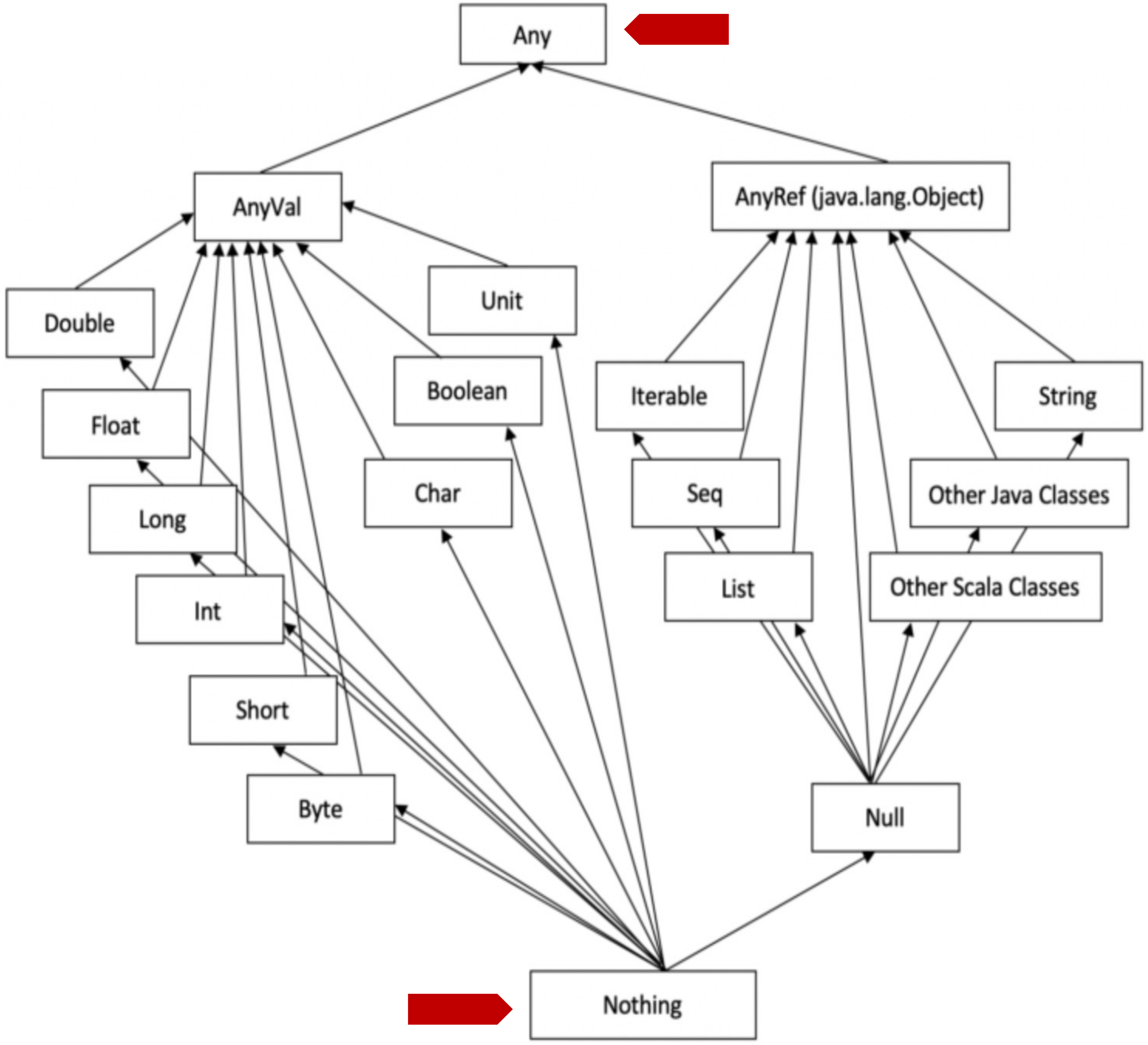
- Covariant if it preserves the ordering of types from more specific to more generic.
- Contravariant if it reverses the ordering.
- Invariant if neither of these applies.

In Scala, by default, we are not able to do so. But, we can address this limitation with the help of variance annotations.

Variance

- We can define our generic types to be
  - covariant or
  - contravariant
- For example:
  - Queue[+ T](...){...} indicates covariant where Queue[String] is considered a subtype of Queue[AnyRef]
  - Queue[−T](...){...} indicates contravariance where Queue[AnyRef] would be considered a supertype of Queue[String].

We are declaring the Queue data type to be polymorphic in the type of elements it contains.



https://www.baeldung.com/scala/type-hierarchies

LUT University

Lower and Upper Bounds

- We use lower bound on the type specification to limit the types that can be used for T or any supertype of T.
- When we enqueue an item then the type of this item must be of type T or a super type of T.
- if we have a queue of Employees this would allow us to enqueue a Person to the Queue (although the resulting Queue only guarantee that it holds references to Person objects (or subtypes of Person).

```
def enqueue[U >: T](x: U) =
        new Queue(x, _head :: tail)
```

- If you want to limit the types
  - that can be used with T or
  - any subclass of T
  - *then* you can use an upper bound!

```
def enqeue[U <: T](x: U) =
        new Queue(x, _head :: tail)
```

Combining Variance and Bounds

We can combine variance and bounds together for flexible containers.

- FlexiQueue class indicates that type T indicates covariance.
- the methods **enqueue** and **dequeue** have lower bounds: the type T and its super types may be used with these methods.
- If we create a FlexiQueue of Employees and then subsequently enqueue a Person this will return a FlexiQueue of Person types.

```scala
case class FlexiQueue[+T](head: T, tail: List[T]) {
  def enqueue[U >: T](x: U) =
                   new FlexiQueue(x, head :: tail)

  def peek = head

  def dequeue[U >: T] =
              new FlexiQueue[U](tail.head, tail.tail)
}
```

LUT University

Combining Variance and Bounds

Type inferred by Scala for q1 and q2:

- q1 holds a reference to a FlexiQueue[Employee] type of instance.
- q2 holds a reference to a FlexiQueue[Person] type of instance.
- Once we added a Person to the FlexiQueue, the only thing we could now guarantee is that the contents of the queue are now at least Person instance.

```scala
class Person(val name: String)

class Employee(_name: String) extends Person(_name)

object FlexiQueueTest extends App {
  val q1 = FlexiQueue[Employee](
    new Employee("John"),
    List(new Employee("Denise"),
      new Employee("Phoebe")))
  println(q1)
  val q2 = q1.enqueue(new Person("Adam"))
  println(q2)
}
```

The type to be used should be a Person or a subtype of Person.

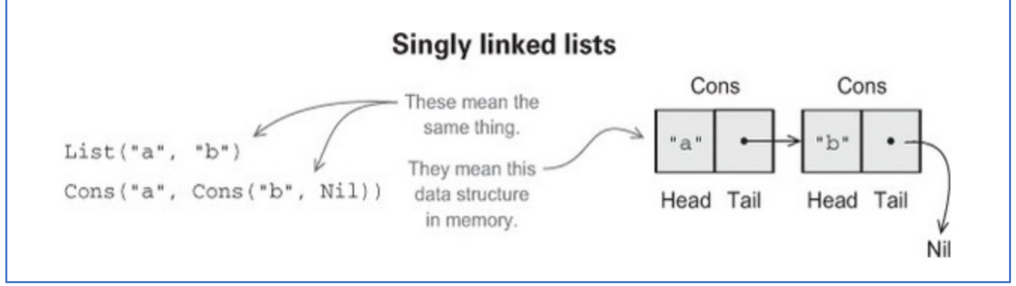
```scala
class Queue[+T <: Person]() {...}
```

LUT University

A trait is an abstract interface that *may optionally* contain implementations of some methods.

Adding **sealed** in front means that all implementations of the trait must be declared in this file.

A companion object in addition to our data type and its data constructors. This is just an object with the same name as the data type (in this case List) where we put various convenience functions for creating or working with values of the data type.



Object Nil lets us write Nil to construct an empty List

```scala
package fpinscala.datastructures

sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {
  def sum(ints: List[Int]): Int = ints match {
    case Nil => 0
    case Cons(x,xs) => x + sum(xs)
  }

  def product(ds: List[Double]): Double = ds match {
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(x,xs) => x * product(xs)
  }

}
```

List companion object. Contains functions for creating and working with lists.

List data type, parameterized on a type, A.

A List data constructor representing the empty list.

Two implementations (data constructors) or two possible forms a List can take; empty (Nil) or non-empty (Cons).

Another data constructor, representing nonempty lists. Note that tail is another List[A], which may be Nil or another Cons.

The sum of a list starting with x is x plus the sum of the rest of the list.

The sum of the empty list is 0.

A function that uses pattern matching to add up a list of integers.

LUT University

Pattern Matching

Pattern matching works a bit like a fancy switch statement.

```
val anInteger = 55
val order = anInteger match {
  case 1 => "first"
  case 2 => "second"
  case 3 => "third"
  case _ => anInteger + "th"
}
```

```
case class Person(name: String, age: Int)
val bob = Person("Bob", 43) // Person.apply("Bob", 43)

val personGreeting = bob match {
  case Person(n, a) => s"Hi, my name is $n and I am $a years old."
  case _ => "Something else"
}
```

```
val aList = List(1,2,3)
val listDescription = aList match {
  case List(_, 2, _) => "List containing 2 on its second position"
  case _ => "unknown list"
}
```

Source: rockthejvm

Pattern Matching

Pattern matching works a bit like a fancy switch statement.

These are recursive definitions/methods

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)
}
```

the sum of a nonempty list is the first element, x, plus the sum of the remaining elements, xs.

```
def product(ds: List[Double]): Double = ds match {

  case Nil => 1.0
  case Cons(0.0, _) => 0.0
  case Cons(x,xs) => x * product(xs)
}
```

Known as target or scrutinee.

product definition states that the product of an empty list is 1.0, the product of any list starting with 0.0 is 0.0, and the product of any other nonempty list is the first element multiplied by the product of the remaining elements.

- Each case in the match consists of a pattern (like Cons(x,xs)) to the left of the => and a result (like x * product(xs)) to the right of the =>.

- If the target matches the pattern in a case, the result of that case becomes the result of the entire match expression.
- If multiple patterns match the target, Scala chooses the first matching case.

Pattern Matching!

- List(1,2,3) match { case _ => 42 }

Here we're using a variable pattern, _, which matches any expression. We could say x or foo instead of _, but we usually use _ to indicate a variable whose value we ignore in the result of the case.

- List(1,2,3) match { case Cons(h , _) => h }

Here we're using a data constructor pattern in conjunction with variables to capture or bind a subexpression of the target.

- List(1,2,3) match { case Cons(_ , t) => t }

- List(1,2,3) match { case Nil => 42 }

References

- Hunt, J. (2018). A Beginner’s Guide to Scala, Object Orientation and Functional Programming. In *A Beginner’s Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing. https://doi.org/10.1007/978-3-319-75771-1
- Chiusano, P., & Bjarnason, R. (2014). *Functional Programming in Scala*. Manning publications.