Iterators and Streams

Traversable, Iterable and Iterator

- Traversable is the base trait of all collections
  - One abstract method – foreach
- Iterable trait extends Traversable
  - Like Java's Iterable interface
  - One abstract method – iterator
  - Implements foreach using iterator methods
- Iterator trait
  - Abstract iterator object
  - Two abstract methods – hasNext and next

Working With Iterators

- Iterators can be used in the same way as Java
    - External iteration

```
scala> val sl = List ("one", "two", "three", "four")
sl: List[String] = List(one, two, three, four)

scala> val it = sl.iterator
it: Iterator[String] = non-empty iterator

scala> while ( it.hasNext )
     |   println(it.next)
one
two
three
four
```

Working With Iterators

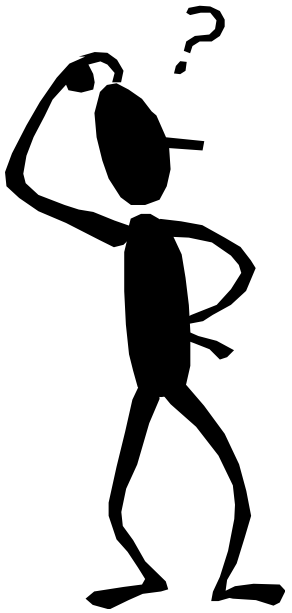
- Iterators support internal iteration
  - HOFs like collection types
  - Subtle differences in behaviour

```
scala> val it = sl.iterator
it: Iterator[String] = non-empty iterator

scala> it foreach ( println(_) )
one
two
three
four

scala> it foreach ( println(_) )


scala> it.next
java.util.NoSuchElementException: next on empty iterator
```



Working With Iterators

- Iterators support internal iteration
  - HOFs like collection types
  - Subtle differences in behaviour

```
scala> val it = sl.iterator
it: Iterator[String] = non-empty iterator

scala> it map ( _.toUpperCase )
res158: Iterator[String] = non-empty iterator

scala> it map ( _.toUpperCase ) foreach ( println(_) )
ONE
TWO
THREE
FOUR
```

Working With Iterators

- Many useful methods available

Method	Purpose
nonEmpty	Is there any data?
size	How many elements to iterate through
min, max, sum	
drop, take, dropWhile, takeWhile	
sameElements	Compare two iterators for equality
mkString	Convert to a String
slice	Subset of elements
toList, toSeq, etc	Convert into collections

Streams

- Streams present a similar abstraction to Lists
- Streams are *lazy*
  - Elements are evaluated/calculated only when requires
  - List requires all elements to be present
- Useful for working with sequences whose elements are not (yet) available
- Also infinite sequences
  - E.g. prime numbers

Defining Streams

- Streams are defined as an Algebraic Sum Type

- A Stream is
  - Empty
  - An element followed by another Stream (cons)

- BUT…

- The tail is only calculated when it is required

```
object Stream {
  def cons[T](hd: T, tl: => Stream[T]) =
    new Stream[T] {
      def isEmpty = false
      def head = hd
      lazy val tail = tl
    }
  val empty = new Stream[Nothing] {
    def isEmpty = true
    def head = throw
       new NoSuchElementException("empty.head")
    def tail = throw
       new NoSuchElementException("empty.tail")
  }
}
```

Defining Streams

- Defining a Stream of Int

```
scala> val str1: Stream[Int]  = Stream.cons(0, Stream.cons(1, Stream.empty))
str1: Stream[Int] = Stream(0, ?)
```

- Use operator #:: as alternative notation

```
scala> val str2 = 0 #:: 1 #:: empty
str2: scala.collection.immutable.Stream[Int] = Stream(0, ?)
```

- ? in toString output indicates tail is calculated on demand

Defining Streams

- Seq[A] can be converted to a Stream [A]

```
scala> val l1 = List(1,2,3,4,5,6,7,8)
l1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)

scala> l1 toStream
res171: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> 1 to 10 toStream
res170: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

- Elements will be pulled through stream as required

Defining Streams

- Stream can be defined explicitly
  - Function to calculate "next" element

```
scala> def rangeToStream ( from: Int, to: Int ): Stream[Int] = {
     |   if ( from >= to ) Stream.empty
     |   else Stream.cons(from, rangeToStream(from+1, to) )
     | }
rangeToStream: (from: Int, to: Int)Stream[Int]

scala> rangeToStream(1,4)
res164: Stream[Int] = Stream(1, ?)

scala> rangeToStream(1,4) take(2) toList
warning: there was one feature warning; re-run with -feature for details
res165: List[Int] = List(1, 2)
```

Infinite Streams

- Streams do not require a finite length

```
scala> def streamOfSquares(from: Int): Stream[Int] =
                (from*from) #:: streamOfSquares(from + 1)
streamOfSquares: (from: Int)Stream[Int]
```

- or

```
scala> def streamOfSquares( from: Int ): Stream[Int] =
                Stream.cons( (from*from), streamOfSquares(from+1))
streamOfSquares: (from: Int)Stream[Int]
```

Infinite Streams

- Operations on infinite stream return a stream

```
scala> streamOfSquares(1)  map ( _ + 1 )
res177: scala.collection.immutable.Stream[Int] = Stream(2, ?)
```

- Use other methods to "materialize" the stream

```
scala> streamOfSquares(1) take(5) foreach ( println(_) )
1
4
9
16
25
```

Stream vs List

- List requires all elements to be in place

```
scala> l1 map ( _ * 2 )
res172: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16)
```

- Stream calculates elements as required

```
scala> str1 map ( _ * 2 )
res173: scala.collection.immutable.Stream[Int] = Stream(2, ?)

scala> str1 map ( _ * 2 ) take 3 toList
warning: there was one feature warning; re-run with -feature for details
res175: List[Int] = List(2, 4, 6)
```

Useful Stream Methods

- Stream.empty
  - Return an empty stream
- Stream.from(n)
  - Return a stream that supplies Int values starting at n and incrementing by 1 each time
- Stream.continually
  - Return a stream that uses the supplied expression to calculate the next element
- Stream.range
  - Return a stream that supplies the values defined in the range
  - Like the rangeToStream method shown earlier

Streams vs Iterators

- Iterators are mutable, streams are immutable
  - be careful about iterators "escaping" from an "immutable" function
  - Iterators are not strictly functional
- Streams cache values, may be more efficient
- Iterator supports once only traversal of values