Collections

Tuples

- Aggregate type containing elements of possibly different types
  - Like struct

```
scala> val myTuple = ( 1, "One" )
myTuple: (Int, String) = (1,One)

scala> val myData = ( 3.5, 2, new java.util.Date )
myData: (Double, Int, java.util.Date) = (3.5,2,Thu Aug 11 22:04:28 CST 2016)
```

- Instances of a range of types
  - Tuple1, Tuple2,… Tuple22
  - Tuple2 aliased to Pair
- Known as Product Types
  - Cartesian product

Tuples

- Syntactic sugar available for Pair (Tuple2)

```
scala> val myPair = "George" -> 21
myPair: (String, Int) = (George,21)
```

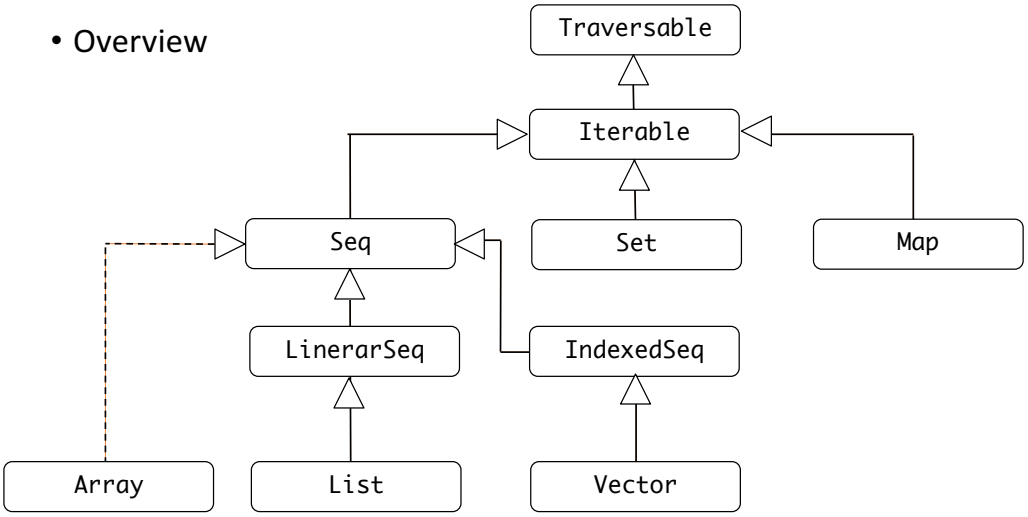
- Elements of tuple can be accessed using special names
  - _1 for first element
  - *Compiler* checks for "range"

```
scala> myPair._1
res87: String = George

scala> myPair._4
<console>:13: error: value _4 is not a member of (String, Int)
       myPair._4
              ^
```


Collections Types

- Overview



Collections Packages

- Based on implementation characteristics
- Immutable
  - Package scala.collection.immutable
  - Cannot be changed in place (may generate new copies if updates allowed)
  - scala.Predef defines type aliases so default collection types are immutable
  - (Except for Seq, to allow for Arrays)
- Mutable
  - Package scala.collecton.mutable
  - Collections may be changed in place
- Beware – this is not related to val / var

Working with Collections

- Syntactic sugar allows convenient notation to be used
  - Companion object and factory methods

```
scala> val mySeq = List(1,2,3)
mySeq: List[Int] = List(1, 2, 3)

scala> val s1 = Set(1, "Hello")
s1: scala.collection.immutable.Set[Any] = Set(1, Hello)
```

- Note collection types are parameterised
  - Type inference engine can deduce element type from initialisation
  - Can specify element type if extra checking required

```
scala> val v1 = Vector[Int](12, 13, 14)
v1: scala.collection.immutable.Vector[Int] = Vector(12, 13, 14)
```

Seq

- Ordered collection of items
  - Indexed using Int expressions
- Subtypes differ in implementation approach
  - Array – use JVM array types and associated bytecodes
  - IndexedSeq (Vector) – constant time access to elements
  - LinearSeq (List) – Fast head/tail, length functions
- All collection types are parameterised
  - Even Array

```
scala> val a = Array(1, 2, 3)
a: Array[Int] = Array(1, 2, 3)

scala> a(2)
res91: Int = 3

scala> a(1) = 0

scala> a
res93: Array[Int] = Array(1, 0, 3)
```


Seq

- Many methods and properties available

```
scala> val s = Seq( 1, 2, 3, 4 )
s: Seq[Int] = List(1, 2, 3, 4)

scala> s.length
res94: Int = 4

scala> s.indices
res95: scala.collection.immutable.Range = Range(0, 1, 2, 3)

scala> s.reverse
res96: Seq[Int] = List(4, 3, 2, 1)

scala> s.contains(2)
res97: Boolean = true
```

Seq

```
scala> s :+ 5
res98: Seq[Int] = List(1, 2, 3, 4, 5)

scala> 0 +: s
res99: Seq[Int] = List(0, 1, 2, 3, 4)

scala> s ++ Seq(4, 5, 6, 7)
res100: Seq[Int] = List(1, 2, 3, 4, 4, 5, 6, 7)

scala> s.count ( _ == 4 )
res101: Int = 1

scala> val names = Seq("tom", "dick", "harry")
names: Seq[String] = List(tom, dick, harry)

scala> names.sorted
res103: Seq[String] = List(dick, harry, tom)

scala> names.sortWith((a,b) => a.length.compareTo(b.length) < 0)
res105: Seq[String] = List(tom, dick, harry)
```

Right binding operator

Note List is immutable type

Set

- Unordered collection, no duplicates
  - Smaller number of methods

```
scala> val odds = Set( 1, 3, 5, 7, 9 )
odds: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3)

scala> odds.contains(3)
res110: Boolean = true

scala> odds & Set( 3, 4, 5 )
res109: scala.collection.immutable.Set[Int] = Set(5, 3)

scala> odds | Set( 3, 4, 5 )
res111: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3, 4)

scala> odds &~ Set( 3, 4, 5 )
res113: scala.collection.immutable.Set[Int] = Set(1, 9, 7)

scala> odds + 4
res115: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3, 4)

scala> odds + 11
res116: scala.collection.immutable.Set[Int] = Set(5, 1, 9, 7, 3, 11)
```

Map

- Collection of (key, value) pairs
  - Can be constructed out of a collection of Pair(Tuple2) objects

```
scala> val capitals = Map( "Denmark" -> "Copenhagen",
                           "Sweden" -> "Stockholm",
                           "Norway" -> "Oslo" )
capitals: scala.collection.immutable.Map[String,String] = Map(Denmark ->
Copenhagen, Sweden -> Stockholm, Norway -> Oslo)

scala> capitals.keys
res117: Iterable[String] = Set(Denmark, Sweden, Norway)

scala> capitals.values
res118: Iterable[String] = MapLike(Copenhagen, Stockholm, Oslo)
```


Map

- Lookup operation through apply method
  - "Associative Array"
  - Exception if no such key, alternative approaches available

```
scala> capitals("Norway")
res120: String = Oslo

scala> capitals("Finland")
java.util.NoSuchElementException: key not found: Finland
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
  ... 32 elided

scala> capitals.getOrElse("Finland", "I have no idea")
res123: String = I have no idea
```

Map

- Use MutableMap type to add or change entries

```
scala> val mCapitals = collection.mutable.Map() ++ capitals
mCapitals: scala.collection.mutable.Map[String,String] = Map(Norway -> Oslo,
Denmark -> Copenhagen, Sweden -> Stockholm)

scala> mCapitals("Finland") = "Helsinki"

scala> mCapitals += "Iceland" -> "Reykjavik"
res131: mCapitals.type = Map(Norway -> Oslo, Denmark -> Copenhagen, Iceland ->
Reykjavik, Sweden -> Stockholm, Finland -> Helsinki)

scala> val capitals2 = mCapitals toMap
capitals2: scala.collection.immutable.Map[String,String] = Map(Denmark ->
Copenhagen, Iceland -> Reykjavik, Finland -> Helsinki, Sweden -> Stockholm,
Norway -> Oslo)
```

Change back to immutable Map

Dealing With Optional Values

- What should be returned when we use a non-existent key to retrieve value from a Map?
  - Throw Exception…
  - Return special value (e.g null)
- Neither of these is type safe
  - What is the type of null in Java???
- Need an equivalent to SQL's NULL
  - Representation of "no value"

Dealing With Optional Values

- Option[T] is the type of a value that may or may not be present
  - If present, the value will have type T
- Represented by a sealed type hierarchy
  - Two case class subtypes of Option[T]
  - Some[T] is a container for a value of type T
  - None represents no value

```
scala> capitals.get("Denmark")
res0: Option[String] = Some(Copenhagen)

scala> capitals.get("Finland")
res1: Option[String] = None
```

Success and failure cases both have the same type

Dealing With Optional Values

- Retrieve the actual value using the get mathod
  - Returns value of type T if present
  - Throws NoSuchElementException if no value
- Use getOrElse to return "default" value in case where no value is present

```
scala> capitals.get("Denmark").get
res2: String = Copenhagen

scala> capitals.get("Finland").get
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:347)
  at scala.None$.get(Option.scala:345)
  ... 32 elided

scala> capitals.get("Finland").getOrElse("I don't know")
res4: String = I don't know
```

Dealing With Optional Values

- Pattern matching can also be used

```
scala> val capitalOf = ( country: String ) => capitals.get(country) match {
     |   case Some(city) => city
     |   case None => "Unknown"
     | }
capitalOf: String => String = <function1>

scala> capitalOf("Norway")
res5: String = Oslo

scala> capitalOf("Iceland")
res6: String = Unknown
```

- Further possibilities (see later)

Functional Programming and Collections

- Conventional approach based on Iterator pattern
  - Process each element in turn
  - "External" iteration
- Collections lend themselves to functional programming style
  - Algorithms can elegantly be specified
  - "Internal" iteration
- Based on Higher Order Functions
  - map
  - flatMap
  - filter
  - …

Using Higher Order Functions

- Large range of methods defined on collections

```
scala> val range1 = 1 to 5
range1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> range1.foreach( i => print(s"$i ") )
1 2 3 4 5

scala> range1 forall ( i => i <= 5 )
res133: Boolean = true

scala> range1 exists ( i => i == 10 )
res134: Boolean = false
```


Using map

- Return a new collection
  - Apply function to elements in source to generate elements in result

```
scala> val range1 = 1 to 5
range1: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)

scala> val plusOne = range1.map( i => i + 1 )
plusOne: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 3, 4, 5, 6)

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

scala> sList.map( s => s.toUpperCase )
res135: List[String] = List(ONE, TWO, THREE)
```

Combining Elements from a Collection

- foldLeft and foldRight
  - Function argument determines how elements are combined
  - Notice functions are defined in "curried" form – two argument lists
  - foldRight scans elements right to left

```
val theSum = range1.foldLeft(0)( (i, j) => i + j )
```

Initial Value — Accumulating "Total" — Element to process

```
scala> val pathName = sList.foldLeft("")( (a, b) => s"$a/$b" )
pathName: String = /one/two/three

scala> val pathName = sList.foldRight("")( (a, b) => s"$a/$b" )
pathName: String = one/two/three/
```

Initial Value — Element to process — Accumulating "Total"

Selecting Elements from a Collection

- filter
  - Elements placed into new collection based on supplied predicate function

```
scala> val oddNumbers = range1.filter ( _ % 2 != 0 )
oddNumbers: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 3, 5)

scala> val strings = sList.filter( s => s.length > 3 )
strings: List[String] = List(three)

scala> val word = "fortunately"
word: String = fortunately

scala> word.seq
res139: scala.collection.immutable.WrappedString = fortunately

scala> word.seq.filter("aeiou".contains(_) )
res140: scala.collection.immutable.WrappedString = ouae
```

Partitioning a Collection

- partition method
  - Return a Tuple2 of collections

```
scala> range1.partition( _ % 2 == 0 )
res141: (scala.collection.immutable.IndexedSeq[Int],
scala.collection.immutable.IndexedSeq[Int]) = (Vector(2, 4),Vector(1, 3, 5))

scala> word.seq.partition ( "aeiou".contains(_) )
res142: (scala.collection.immutable.WrappedString,
scala.collection.immutable.WrappedString) = (ouae,frtntly)
```

- groupBy gives more general grouping/partitioning capability

```
scala> 1 to 10 groupBy ( _ % 3 )
res143:
scala.collection.immutable.Map[Int,scala.collection.immutable.IndexedSeq[Int]]
= Map(2 -> Vector(2, 5, 8), 1 -> Vector(1, 4, 7, 10), 0 -> Vector(3, 6, 9))
```


Joining Collections

- Zip
  - Combines Seq[A] and Seq[B] objects into Seq[Tuple2[A,B]]

```
scala> 1 to 3 zip sList
res145: scala.collection.immutable.IndexedSeq[(Int, String)] =
        Vector((1,one), (2,two), (3,three))
```

- Also zipWithIndex

```
scala> val colours = List("Red", "Green", "Blue")
colours: List[String] = List(Red, Green, Blue)

scala> colours zipWithIndex
warning: there was one feature warning; re-run with -feature for details
res146: List[(String, Int)] = List((Red,0), (Green,1), (Blue,2))
```

Nested Collections

- Collections can contain other collections
  - Often the result of call to map

```
scala> val text = List("Here is the first line", "this is the second line")
text: List[String] = List(Here is the first line, this is the second line)

scala> val splitlines = text.map( _.split(" ") )
splitlines: List[Array[String]] =
List(Array(Here, is, the, first, line), Array(this, is, the, second, line))
```

- Flatten method removes one level of nesting

```
scala> splitlines flatten
res150: List[String] =
List(Here, is, the, first, line, this, is, the, second, line)
```


Combining map and flatten

- flatMap
  - Very important function
  - Forms basis of many idioms in Scala and functional programming
  - Used with many types, not just collections
  - E.g. Option[T], Future[T], Try{T]
  - Monads

```
scala> val words = text.flatMap( _.split(" ") )
words: List[String] =
         List(Here, is, the, first, line, this, is, the, second, line)
```

Option[T] and the Higher Order Functions

- map, flatMap, foreach, etc all defined on Option[T] type
  - Allow processing on results of operations without forcing null check

```
scala> capitals.get("Norway").map(_.toUpperCase)
res10: Option[String] = Some(OSLO)

scala> capitals.get("Finland").map(_.toUpperCase)
res11: Option[String] = None

scala> capitals.get("Norway").foreach( println(_) )
Oslo

scala> capitals.get("Iceland").foreach( println(_) )

scala>
```


The for Comprehension

- Used to generate a new instance of a container type
  - E.g. collections
  - Yield operator specifies how elements are generated

```
scala> val squares = for ( i <- 1 to 5 ) yield i * i
squares: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16, 25)
```

- Equivalent to map
  - Compiler rewrites for comprehension to use map

```
scala> val squares = 1 to 5 map ( i => i * i )
squares: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16, 25)
```

The for Comprehension

- Can also specify nested for comprehensions

```
scala> val coordinates = for ( x <- 1 to 3;
     |                         y <- 6 to 9 )
     |                     yield (x,y)
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] =
             Vector((1,6), (1,7), (1,8), (1,9), (2,6), (2,7), (2,8), (2,9),
                    (3,6), (3,7), (3,8), (3,9))
```

- Equivalent to combination of flatMap and map

```
scala> val coordinates = 1 to 3 flatMap ( x => 6 to 9 map ( y => (x,y) ) )
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] =
             Vector((1,6), (1,7), (1,8), (1,9), (2,6), (2,7), (2,8), (2,9),
                    (3,6), (3,7), (3,8), (3,9))
```


The for Comprehension

- for comprehension can include guard

```
scala> val coordinates = for ( x <- 1 to 3 if x % 2 != 0;
     |                         y <- 6 to 9 if y %2 == 0 )
     |                     yield ( x, y )
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] =
             Vector((1,6), (1,8), (3,6), (3,8))
```

- Equivalent to use of filter

```
scala> val coordinates = 1 to 3 filter ( _ % 2 != 0 ) flatMap (
                              x => 6 to 9 filter ( _ % 2 == 0 ) map (
                                        y => (x,y) ) )
coordinates: scala.collection.immutable.IndexedSeq[(Int, Int)] = Vector((1,6),
(1,8), (3,6), (3,8))
```

The for Loop

- Special case of for comprehension
  - Function argument returns Unit

```
scala> for ( i <- 1 to 5 ) print (s"$i ")
1 2 3 4 5
```

- Equivalent to foreach higher order function

```
scala> 1 to 5 foreach { i => print(s"$i ") }
1 2 3 4 5
```

- Can also be used with guards

Option[T] with for Comprehensions

- Recall that for comprehension is rewritten as combinations of map, flatMap and filter
  - Option[T] supports these methods

```
scala> val countries = List("Denmark", "USA", "Sweden")
countries: List[String] = List(Denmark, USA, Sweden)

scala> for ( c <- countries;
     |       cap <- capitals.get(c) )
     |     yield(c, cap)
res14: List[(String, String)] = List((Denmark,Copenhagen), (Sweden,Stockholm))
```

```
scala> countries flatMap ( c => capitals.get(c) map ( cap => (c,cap) ) )
res15: List[(String, String)] = List((Denmark,Copenhagen), (Sweden,Stockholm))
```