Scala

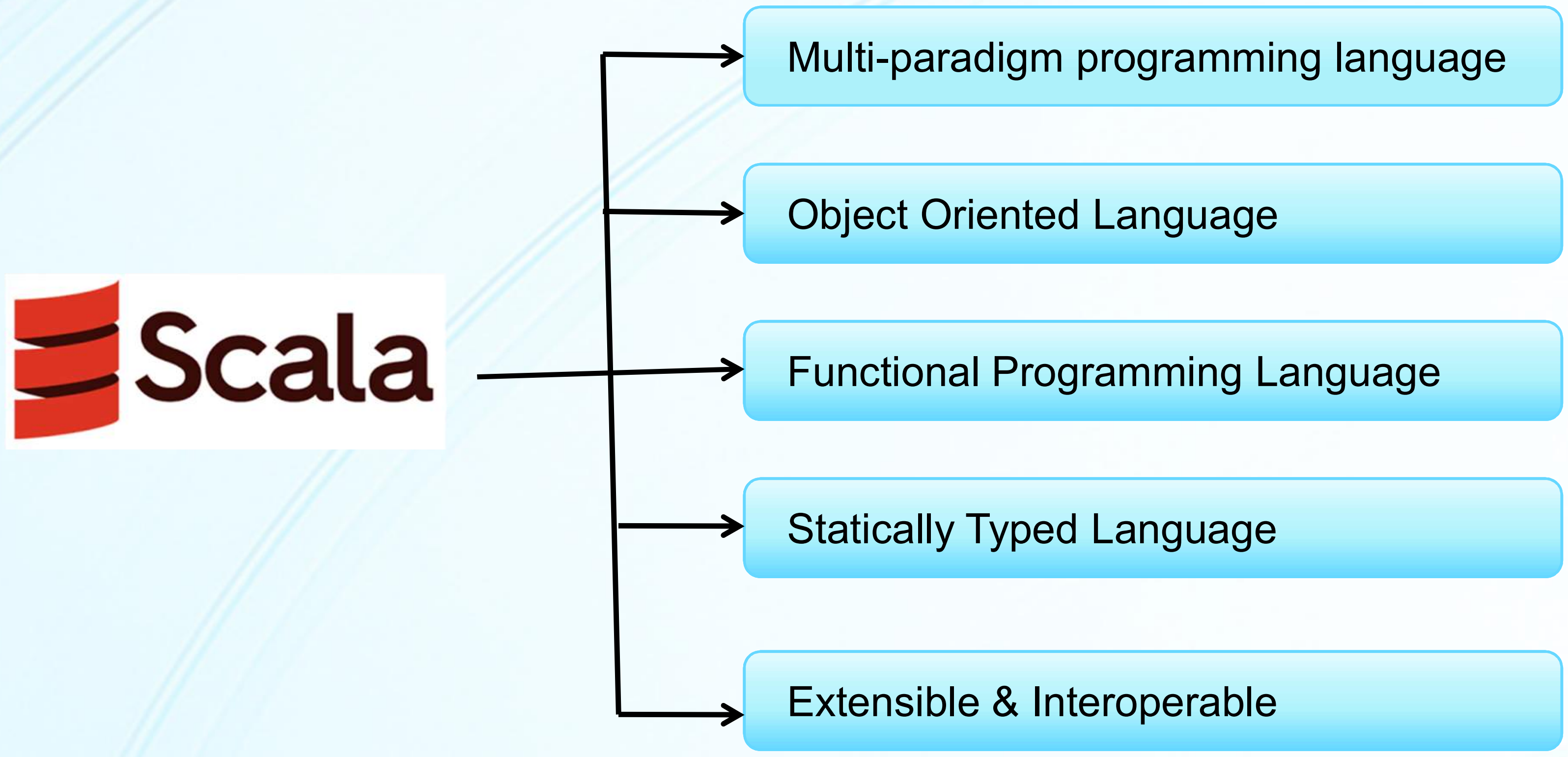
SCALA

Agenda

In this module, we are going to look at the following topics:

- ✓ Scala Basics
- ✓ Scala Programming Constructs
- ✓ Methods, Procedures & Functions
- ✓ File Handling
- ✓ Regular Expressions
- ✓ Collections
- ✓ Closures

What is Scala?


Scala
Multi-paradigm programming language
Object Oriented Language
Functional Programming Language
Statically Typed Language
Extensible & Interoperable

Getting Started with Scala

- The most popular way to get Scala is to use Scala through an IDE.
- First, make sure you have the Java 8 JDK installed.
- Then install an IDE with Scala support. The most popular options are:
    - Scala IDE for Eclipse - http://scala-ide.org
    - IntelliJ IDEA with Scala Plug-in - https://www.scala-lang.org/download

Scala REPL (Interpreter)

To start the Scala interpreter:

- Install Scala.
- Make sure that the scala/bin directory is on the PATH.
- Open a command shell in your operating system.
- Type Scala followed by the Enter key.

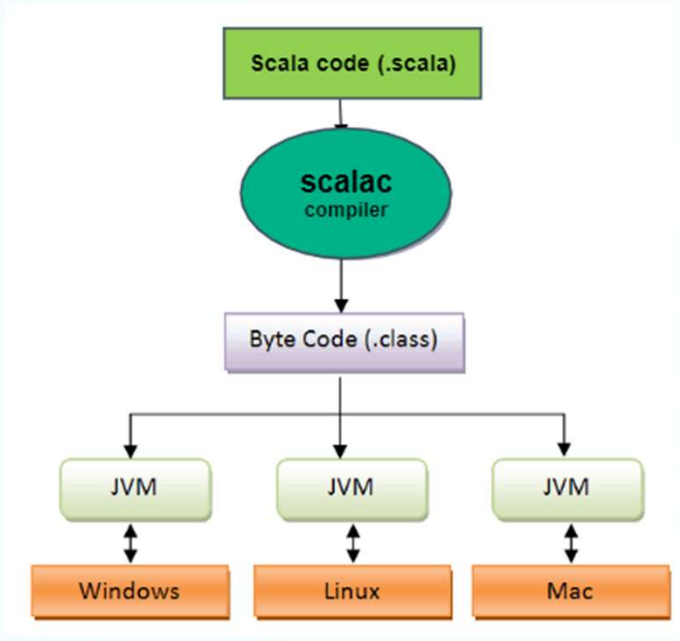
```
scala> 8 * 5 + 2
res0: Int = 42
```

Scala execution steps

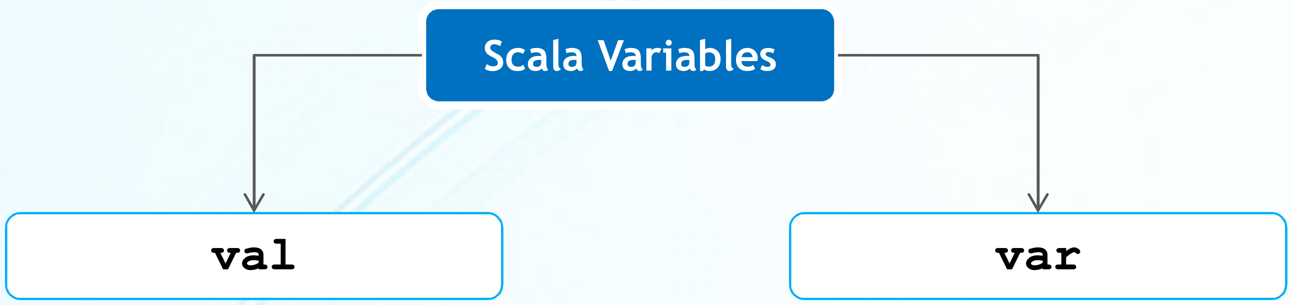
1. Source Code is compiled by the Scala Compiler.

2. Scala Compiler creates executable byte code

3. The executable is executed in a JVM environment to produce the result.



Scala Basics - Variables



- Immutable

```
val i = 10
i = 11
//error: reassignment to val
```

- Mutable

```
var i = 10
i = 11
//allowed here
```

Scala Basics - Expressions

An expression is a computable statement.

Examples:

```
“Hello Scala” (simple expression)
 1 + 2 (compound expression)
```

Type Declaration

Type Declaration:

```
val x : Int = 10
var str : String = “Hello, world”
```

Data Types: Byte, Char, Short, Int, Long, Float, Double, Boolean etc.

Type Inference

- Scala automatically infers the type of the value or variable if not declared.
- Once type is assigned, it remains same for the entire scope.
- Thus, Scala is “statically typed” language.

Blocks

- A block is a list of expressions or statements surrounded by { }
- A block returns the value of the last expression in the block.

```
println( {  val x = 9 + 1
            x + 10 })

val i = {val x = 5; val y = 10; x*y}
println(i)
```

Operator Methods

In Scala, everything is a object. There are no primitives.

- The operators are all methods that operate on objects.
- In the expression “a + b”, a and b are objects and “+” is a method operating on a and b.

```
a.+(b) ➔ a +(b) ➔ a + b (also called infix notation)
```

- Scala do not support ++ & -- operators. Use += 1, -= 1 instead.

Operator Projection in Scala

- **object.method(parameter)** can be written as **operator method parameter**
- Instead of “.” operator, we can use ‘space’. If we pass only one parameter to a method, we can put it after ‘space’ instead of parenthesis
- So, 3.+(4) → 3 +(4) → 3 + 4 // Here + is a method defined in the Int class.

Input

- To read a line of input from the standard input (keyboard) use **`scala.io.StdIn.readLine`** method.

- To read numeric, boolean or char values, use **`readInt, readLong, readDouble, readFloat, readByte, readBoolean & readChar`**

```
import scala.io

val name = StdIn.readLine("Enter your name: ")
println("Enter your age: ")

val age = StdIn.readInt()
println(s"Name: ${name}  Age: ${age}")
```

Output

- To print a value use **`print`** or **`println`** function.
- **`println`** adds a newline character after the printout.
- **`printf`** prints a formatted string (C – style)

```
printf("We are using %s ver %f", "Scala", "2.12")
// prints: We are using Scala ver 2.12
```

String interpolators

- Scala provides three string interpolation methods out of the box: they are “s”, “f” and “raw” interpolators

```
val name = "Raju";  val age = 40; val height = 1.9d

println(s"Hello, $name! Age: ${age+2}")

println(f"$name%s is $height%2.2f meters tall")

println(raw”a \n b”)
```

Lazy Values

- You can define a value as lazy in Scala.

```
lazy val filex = Source.fromFile(“myfile.txt”)
```

- Lazy value initialization is differed until it is accessed for the first time.

- Lazy value do not give error on initialization where as non-lazy values can give errors on initialization.

- Lazy values are useful for delaying costly initialization instructions such as from reading a file etc.

if - else if - else

- Scala has an if / else construct with the same syntax as in Java.
- But, in Scala, an if/else has a value, namely the value of the expression that follows the if or else.

```
if (x > 0) s = 1 else s = -1
val s = if (x > 0) 1 else -1
```

- Other if constructs: else if && nested if are same as Java
- Type of a mixed expression is the super-type of all branches. Any is the super type of all classes in Scala class hierarchy

```
val s = if (x > 0 && x < 6) “positive” else 0
```

Loops

foreach Loop

- **`foreach`** provides a looping construct to loop through any iterable object such as Arrays, Lists, Vectors etc.

```
var str1 = "SCALA"
str1.foreach(println)

var arr = Array(1,2,3,4,5)
arr.foreach(a => print(a + "/"))

(1 to 5).foreach(print)
```

while Loop

- `while` loop is similar to that in Java
- Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

do-while loop

- Unlike **while** loop, which tests the loop condition at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop.

- A **do-while** loop is similar to a while loop, except that a do-while loop is guaranteed to execute at least one time.

```
do {
   statement(s);
}
while( condition );
```

for Loop

- A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

- The for loop in Scala has no direct analog with that of Java. The for loop in much richer and has many options.

```
for( a <- 1 to 10){
   println( "Value of a: " + a );
}
```

for loop - multiple generators

- You can have multiple generators of the form `variable <- expression` separated by semicolons.

```
for (i <- 1 to 3; j <- 1 to 3) {
        print((10 * i + j) + " ")
}
```

for loop - guards

- Each generator can have a guard, a Boolean condition preceded by if.

```
for (i <- 1 to 3; j <- 1 to 3 if i != j) {
        print((10 * i + j) + " ")
}
```

for comprehension

- You can store return values from a "for" loop in a variable or can return through a function.
- To do so, you prefix the body of the 'for' expression by the keyword **yield**.

```
var a = 0;
val numList = List(1,2,3,4,5,6,7,8,9,10);

var retVal =
   for{ a <- numList if a != 3; if a < 8 } yield a

for( a <- retVal) println( "Value of a: " + a );
```

Exceptions

Exception Handling

- Scala's exceptions work like exceptions in many other languages like Java. Instead of returning a value in the normal way, a method can terminate by throwing an exception.
- As in Java, the objects that you throw need to belong to a subclass of `java.lang.Throwable`. But unlike Java, Scala has no checked exceptions i.e. you don't have to declare that a method throw an exception.
- When you want to handle exceptions, you use a `try{} catch{} finally {}` block
- catch block uses matching to identify and handle the exceptions.

Methods, Functions & Procedures

Methods

- Methods are defined with the **def** keyword followed by a name, parameter lists, a return type, and a body.
- Return type is declared *after* the parameter list and a colon :

```
def add(x: Int, y: Int): Int = { x + y }

println( add(1, 2) )
```

Methods

- Methods can take multiple parameter lists or no parameters at all.

```
def addProd:(x: Int, y: Int)(prod: Int): Int = (x + y) * prod
println(addProd(5, 3)(2))


def name: String = System.getProperty("user.name“)


def getCube(input: Int): String = {
    val cube = input * input * input
    cube.toString
}
```

Procedures

- **Procedures** are like methods with **no return value**. The method body is enclosed in braces without a preceding = symbol.

- The return type is **Unit**.

```
def box(s : String) {
    val border = "-" * s.length + "--\n"
    println(border + "|" + s + "|\n" + border)
}
```

Functions

- Functions are expressions that take parameters. Functions are similar to methods but they do not operate on an object.
- Functions are treated as ‘first-class’ i.e similar to a primitive type like Int.
- **Anonymous functions** are functions with no name.
- Functions may or may not have name or parameters.

```
(x: Int) => x + 1

val prod = (x: Int, y: Int) => x * y
println(prod(5, 2))       // prints 10

val getTheAnswer = () => “John Doe”
println(getTheAnswer())
```

Functions Call-by-Name

- Typically, parameters to functions are *by-value* parameters; that is, the value of the parameter is determined before it is passed to the function.
- Alternatively, Scala offers call-by-name parameters, that are not evaluated until it's called within a function.
- A call-by-name mechanism passes a code block to the call and each time the call accesses the parameter, the code block is executed and the value is calculated.

Example: *com.tekcrux.basics.FunctionsCallByName*

Functions with Named Args

- Named arguments allow you to pass arguments to a function in a different order.
- The syntax is simply that each argument is preceded by a parameter name and an equals sign.

```
def main(args: Array[String]) {
   printInt(b = 5, a = 7);
}

def printInt( a:Int, b:Int ) = {
     println("Value of a : " + a );
     println("Value of b : " + b );
}
```

Functions with variable args

- Scala allows you to indicate that the last parameter to a function may be repeated.

```
def main(args: Array[String]) {
   printStrings("Hello", "Scala", "Python");
}

def printStrings( args:String* ) = {
   var i : Int = 0;

   for( arg <- args ){
     println(s"Arg value[$i] = $arg");
     i = i + 1;
   }
}
```

Functions with default args

- Scala allows you to indicate that the last parameter to a function may be repeated.

```
def main(args: Array[String]) {
   println( "Returned Value : " + addInt() );
}

def addInt( a:Int = 5, b:Int = 7 ) : Int = {
   var sum:Int = 0
   sum = a + b

   return sum
}
```

Higher-order Functions

- Scala allows the definition of higher-order functions.
- These are functions that take other functions as parameters, or whose result is a function

```
println( apply( upper, "Hello") )

 def apply(f: Any => String, v: Any) = f(v)

 def upper[A](x: A) = "[" + x.toString().toUpperCase() + "]"
```

Recursive Functions

- Scala supports function recursion. Recursion means a function can call itself repeatedly.

```
for (i <- 1 to 5) println( factorial(i) )

 def factorial(n: BigInt): BigInt = {
     if (n <= 1)
        1
     else
     n * factorial(n - 1)
 }
```

Nested Functions

- Scala allows you to define functions inside a function and functions defined inside other functions are called ***local functions***.

```
println( factorial(5) )

 def factorial(i: Int): Int = {
    def fact(i: Int, accumulator: Int): Int = {
       if (i <= 1)
          accumulator
       else
          fact(i - 1, i * accumulator)
    }
    fact(i, 1)
 }
```

File Operations

File Handling

- Shown below are some of the basic file operations:

```
import scala.io.Source

val source = Source.fromFile("myfile.txt", "UTF-8")

val lineIterator = source.getLines
for (l <- lineIterator) process l

val lines = source.getLines.toArray

val contents = source.mkString
for (c <- source) process c
```

File Handling

- Scala has no provision for reading binary files. You need to use the Java library.
- Scala has no built-in support for writing files. Use Java `java.io.PrintWriter`
- There are currently no official Scala classes for visiting all files in a directory or for recursively traversing directories. Need to use Java.
- **Serialization:** Serialization is used to transmit objects to other virtual machines or for short-term storage. Here is how you declare a serializable class in Scala.

```
class Person extends Serializable
```

Source

- The `Source` object has methods to read from sources other than files :

```
val source1 = Source.fromURL("http://tekcrux.com", "UTF-8")
val source2 = Source.fromString("Hello, World!")
val source3 = Source.stdin
```

Regular Expressions

Regular Expressions

- The **`scala.util.matching.Regex`** provides methods to work with Regular Expressions.

- To construct a Regex object, use the r method of the String class:

```
val numPattern = "[0-9]+".r
```

- If the regular expression contains backslashes or quotation marks use raw string syntax, """...""".

```
val wsnumwsPattern = """\s+[0-9]+\s+""".r
```

- The `findAllIn` method returns an iterator through all matches. `findFirstIn` methods returns the first occurrence of the match.

Array & ArrayBuffer

Array

- Scala provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type.
- Arrays are mutable, indexed collections of values.
- Don’t use new when supplying initial values. Use () to access elements

```
val intArr = new Array[Int](5)
val strArr = new Array[String](5)
val initArr = Array("Hello", “Scala")
for (i <- 0 until intArr.length) intArr(i) = i*i
for (element <- intArr) println(element)
```

ArrayBuffer

- To create a mutable, indexed sequence whose size can change **`ArrayBuffer`** class is used.
- **`scala.collection.mutable.ArrayBuffer`** is equivalent to Java **`ArrayList`**

```
val arrBuf = ArrayBuffer[Int]()
arrBuf += 1
arrBuf += (2, 3, 5)
arrBuf ++= Array(8, 13, 21)
arrBuf.trimEnd(2)
arrBuf.insert(2, 6)
arrBuf.remove(2)
arrBuf.remove(2, 3)
```

Transforming Array & ArrayBuffer

- It is very easy to take an Array or ArrayBuffer and transform it in some way.
- Transformations don’t modify the original, but they yield a new one.
- Common Array & ArrayBuffer methods

```
Array(1, 5, 8, 9).sum
ArrayBuffer(“Scala", “programming", “arrays").max
val aSorted = a.sortWith(_ < _)
a.mkString("<", ",", ">")
a.count(_>3)
```

Multi-Dimensional Arrays

```
val matrix = Array.ofDim[Double](3, 4)
matrix(0)(1) = 42
```

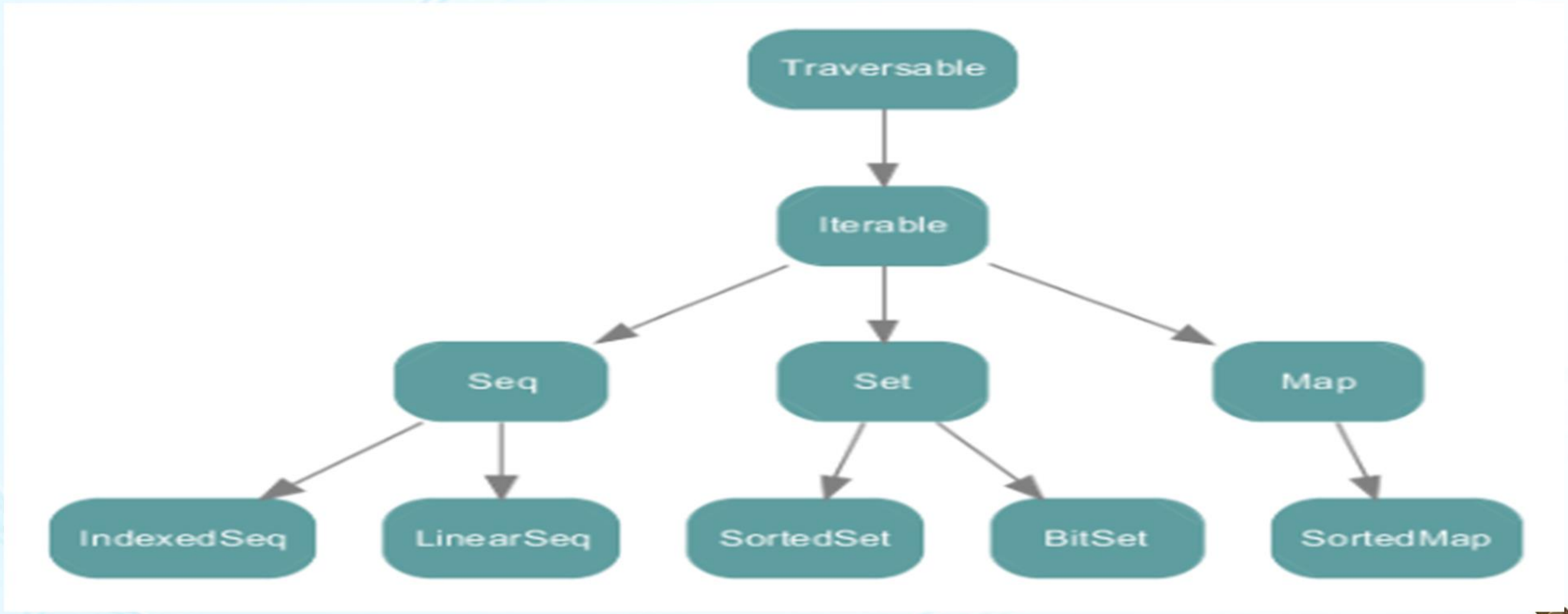
You can make ragged arrays, with varying row lengths:

```
val triangle = new Array[Array[Int]](10)
```

Collections

Map

- All collections extend **`Iterable`** trait
- The major categories of collections are: Sequence, Set & Map
- Scala has mutable and immutable versions of most collections
- An **`Iterable`** is any collection that can yield an iterator



Map

- Scala map is a collection of key/value pairs.
- Any value can be retrieved based on its key.
- Keys are unique in the Map, but values need not be unique. Maps are also called Hash tables.
- There are two kinds of Maps, the **immutable** and the **mutable**.
- The difference between mutable and immutable objects is that when an object is immutable, the object itself can't be changed

Some Map Ops

```
val wd = Map("Sun" -> 1, "Mon" -> 2, "Tue" -> 3)
val wd = Map(("Sun", 1), ("Mon", 2), ("Tue", 8))

val wd = scala.collection.mutable.Map("Sun" -> 1, "Mon" ->
2, "Tue" -> 3)
val monday = wd("Mon")
val weekend = if (wd.contains("Sun")) wd("Sun") else 5
val weekend = wd.getOrElse("Sun", 5)
```

- Calling `map.get(key)` returns an `Option` object that is either `Some` or `None`.

Some Map Ops

- In a **mutable** map, you can update a map value, or add a new one.
- You can use the += operation to add multiple associations.
- Use -= to remove an element.

```
wd(“wed”) = 4   // add a new pair or update an existing pair
wd += (“thu” -> 5, “fri” ->6, “junk” -> 100)
wd -= “junk”    // remove an element by key
```

Some Map Ops

- You can't update an immutable map, but you can obtain a new map that has the desired update using `+ & –` operators.

```
val wdNew = scores + ("fri" -> 6, "sat" -> 7, "junk" -> 100)
val wdNew2 = scores – "junk"
```

- To iterate over k/v pairs of a map: **`for((k, v)<-map) process k & v`**

```
scores.keySet
for (v <- scores.values) println(v)
for ((k, v) <- map) yield (v, k)
```

Tuples

- Tuples are the aggregates of values of different types.
- Maps are collection of Pairs. Pair is the simplest case of Tuple with only two elements.
- A tuple value is formed by enclosing individual values in parentheses.

```
val t = (1, 1.5, "Raju") → Tuple3[Int, Double, java.lang.String]
```

- You can access Tuple's components with the methods _1, _2, _3 etc.
- The component positions of a tuple start with 1, not 0.

```
val second = t._2
```

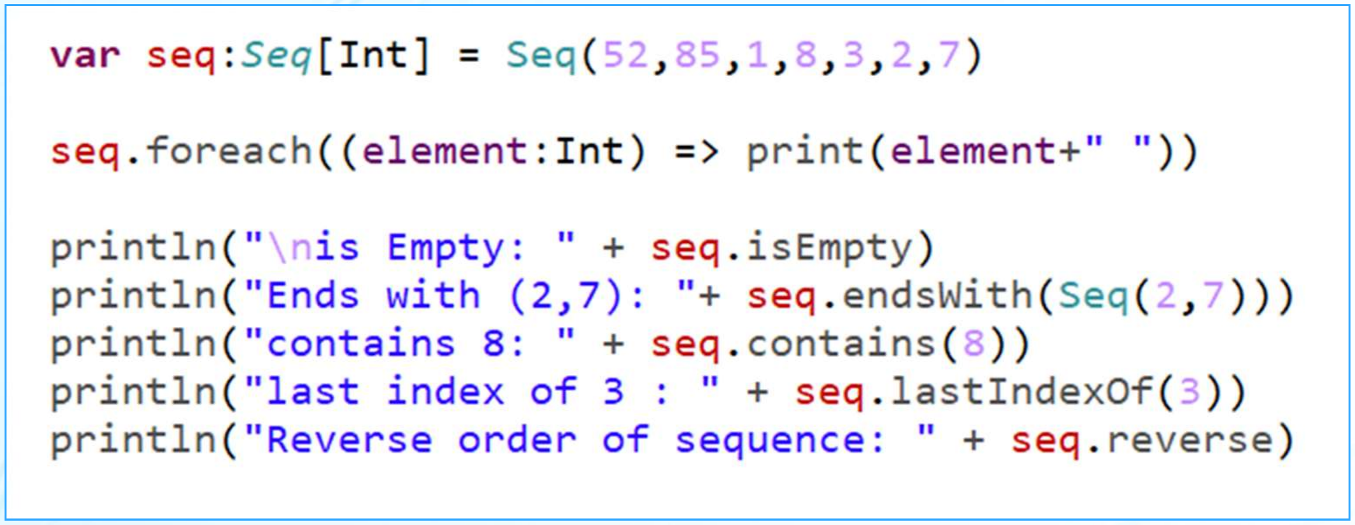
Tuples

- Pattern matching: `val (first, second, third) = t`
- Use _ if you don't need all components: `val(first, second, _) = t`
- Tuples are useful for functions that return more than one value.
- *zip* method pairs two arrays together into an array of pairs

```
val symbols = Array("<", "-", ">")
val counts = Array(2, 10, 2)
val pairs = symbols.zip(counts) → Array(("<", 2), ..)
```

Sequences

- An ordered sequence of values.
- **`IndexedSeq`** allows fast random access through an integer index.
- `Array` & `ArrayBuffer` implement `IndexedSeq`.



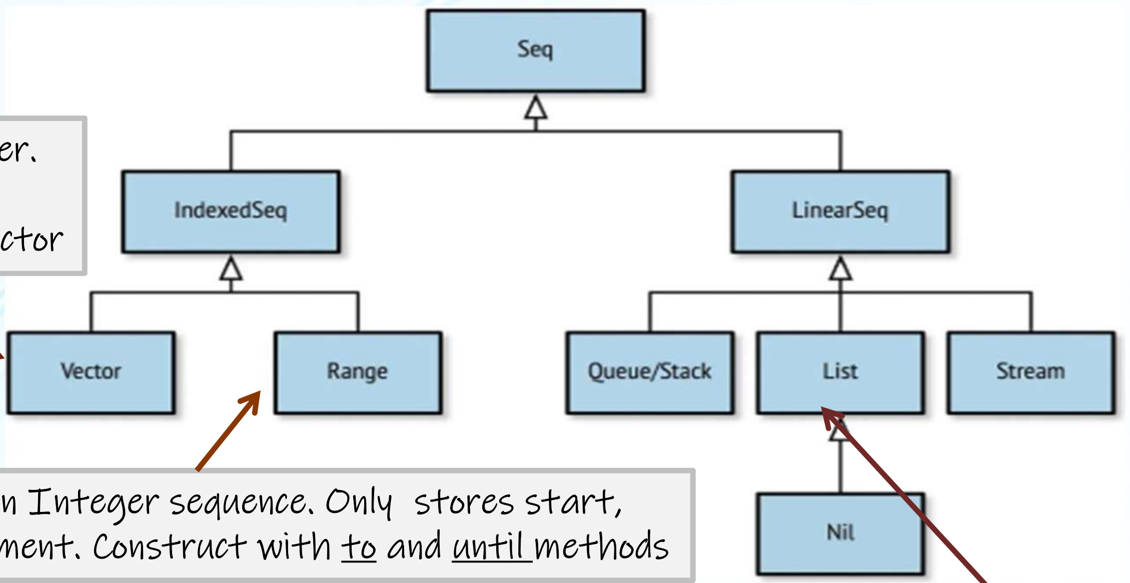
```
var seq:Seq[Int] = Seq(52,85,1,8,3,2,7)

seq.foreach((element:Int) => print(element+" "))

println("\nis Empty: " + seq.isEmpty)
println("Ends with (2,7): "+ seq.endsWith(Seq(2,7)))
println("contains 8: " + seq.contains(8))
println("last index of 3 : " + seq.lastIndexOf(3))
println("Reverse order of sequence: " + seq.reverse)
```

Sequences

- Immutable Sequences:



Immutable equivalent of ArrayBuffer. Provides fast random access. Map method transforms a Range into Vector

Represents an Integer sequence. Only stores start, end and increment. Construct with to and until methods

List has either Nil or an object with head element and a tail that is again a list

- Mutable Sequences:

Seq ← IndexedSeq ← ArrayBuffer

Seq ← Stack, Queue, PriorityQueue, ListBuffer

List

- Scala Lists are quite similar to arrays which means, all the elements of a list have the same type.

- But there are two important differences.
  - First, lists are immutable, i.e elements of a list cannot be changed by assignment.
  - Second, lists represent a linked list whereas arrays are flat.

- A **List** has either a **Nil** (empty) or an object with a **head** element and a **tail** that is again a List.

- A :: operator makes a new list from a given head & tail.

```
val digits = List(4, 2, 8)     // 4 :: 2 :: 8 :: Nil
val digits2 = 9 :: digits      // add an element to a List - 9 :: (4 :: (2 :: (8 :: Nil)))
val sumList = digits2.sum      // sum the elements of a List
def s(lst:List[Int]):Int = if (lst==Nil) 0 else lst.head + s(lst.tail)
```

ListBuffer

- For mutable lists use **ListBuffer.**

```
val lb = ListBuffer("Raju","Veer","Harsha")
lb += "Varma"                  // add element(s) to ListBuffer using +=
lb += ("Aditya", "Amrita")
lb -= "Varma"                  // remove element(s) to ListBuffer using -=
lb -= ("Aditya", "Amrita")
```

Set

- A **`Set`** is a collection of distinct elements

```
Set(1,2,3) + 1  // no effect, already exists
Set(1,2,3) + 4  // Set(1,3,4,2)
```

- Unlike Lists, Sets do not retain the order in which they are inserted.

- By default sets are implemented as ‘HashSets’ in which elements are organized based on the value of a hashCode method.

- Finding an element in a hash set is much faster than in an array or list.

Set

- A **`LinkedHashSet`** remembers the order in which elements were inserted.

- A **`SortedSet`** allows to iterate in a Sorted Order.

```
val hs = scala.collection.mutable.LinkedHashSet("Mo", "Tu", "We‟)
val ss = collection.immutable.SortedSet(1, 2, 3, 4, 5, 6)
```

Option

- Scala Option[T] is a container for zero or one element of a given type.
- An Option[T] can be either **Some[T]** or **None** object, which represents a missing value.
- For instance, the get method of Scala's Map produces **Some(value)** if a value corresponding to a given key has been found, or **None** if the given key is not defined in the Map.

Closures

Closure

- A **closure** is a function, whose return value depends on the value of one or more variables declared outside the function.

```
var factor = 3
val multiplier = (i:Int) => i * factor
```

- There are two free variables in multiplier: i and factor. One of them, i, is a formal parameter to the function. Hence, it is bound to a new value each time multiplier is called.

- Now factor has a reference to a variable outside the function but in the enclosing scope. The function references factor and reads its current value each time.

Pattern Matching

{ match..case }

```
var sign = 0
for (ch <- "+-!") {
  ch match {
    case '+' => sign = 1
    case '-' => sign = -1
    case _ => sign = 0
  }
  println(sign)
}
```

{ match..case }

```
for (ch <- "+-!") {
  sign = ch match {
    case '+' => 1
    case '-' => -1
    case _ => 0
  }
  println(sign)
}
```

{ match..case }

```
 ch match {
  case '+' => sign = 1
  case '-' => sign = -1
  case _ if Character.isDigit(ch)
     => digit = Character.digit(ch, 10)
  case _ => sign = 0
}
```

{ match..case }

```
var str = "+-3!"
for (i <- str.indices) {
 var sign = 0;  var digit = 0
 str(i) match {
   case '+' => sign = 1
   case '-' => sign = -1
   case ch if Character.isDigit(ch)
      => digit = Character.digit(ch, 10)
   case _ => sign = 0
   }
}
```

{ match..case }

```
for (obj<-Array(42, "42", BigInt(42)){
  val result = obj match {
     case x: Int => x
     case s: String => s.toInt
     case _: BigInt => Int.MaxValue
     case BigInt => -1
     case _ => 0
  }
}
```

{ match..case }

```
for (arr <- Array(Array(0), Array(1, 0), Array(0, 1, 0),
Array(1, 1, 0))) {
  val result = arr match {
      case Array(0) => "0"
      case Array(x, y) => x + " " + y
      case Array(0, _*) => "0 ..."
      case _ => "something else"
  }
}
```

{ match..case }

```
for (obj<-Array(Map("A" -> 42),Array(42),Array("A"))) {

   val result = obj match {
    case m: Map[_, _] => "a Map"
    case a: Array[Int] => "It's an Array[Int]"
    case a: Array[_] => "array other than Int"
   }

}
```

{ match..case }

```
for (  lst <- Array(List(0), List(1, 0), List(0, 0, 0),
List(1, 0, 0))) {

    val result = lst match {
      case 0 :: Nil => "0"
      case x :: y :: Nil => x + " " + y
      case 0 :: tail => "0 ..."
      case _ => "something else"
    }

}
```

{ match..case }

- Use | to separate multiple alternatives:
  - `case "0" | "0x" | "0xx" => …`

- If a pattern has alternatives, you can't use variables other than _
  - `case (0,_) | (_,0)` is OK
  - but `case (0,x) | (x,0)` is Error

Patterns in Variables & *for* Expressions

- You can use patterns inside variable declarations to declare multiple variables in a single statement.

```
val (x, y) = (1, 2) // x=1, y=2
val (q, r) = BigInt(10) /% 3  //q=3, r=1
val Array(first, second, _*) = Array(1, 7, 2, 9) //first=1, second=7
val Array(f, s, rest @ _*) = Array(1, 7, 2, 9)  //rest = Vector(2,9)
```

Patterns in Variables & *for* Expressions

- You can use patterns with variables in **`for`** expressions. For each traversed value, the variables are bound. Match failures are silently ignored.

```
for ((k, v) <- System.getProperties.asScala) println(k + " -> " + v)

// match elements with empty value and print the keys.
for ((k, "") <- System.getProperties.asScala) println(k)
for ((k, v) <- System.getProperties.asScala if v == "") println(k)
```

THANK YOU