```
-- static members (methods or fields) do not exist in Scala.
Rather than defining static members, the Scala programmer declares these members in singleton objects.

object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}

--Instead of javac we can compile using scalac
  Ex : HelloWorld.scala
  Compile : >scalac HelloWorld.scala  (Here HelloWorld.class file will be created)
  Execute : >scala HelloWorld

--Interaction with Java
   All classes from the java.lang package are imported by default, while others need to be imported
explicitly.
    import java.util.{Date, Locale}
        import java.text.DateFormat
        import java.text.DateFormat._

        object FrenchDate {
                def main(args: Array[String]) {
                        val now = new Date
                        val df = getDateInstance(LONG, Locale.FRANCE)
                        println(df format now)
         }
        }
        -Multiple classes can be imported from the same package by enclosing them in curly braces as on
the first line
        -We can import all the names of a package or class, one uses the underscore character (_)
instead of the asterisk (*).
           That’s because the asterisk is a valid Scala identifier (e.g. method name).
        -The import statement on the third line therefore imports all members of the DateFormat class.
         This makes the static method getDateInstance and the static field LONG directly visible.
        -Methods taking one argument can be used with an infix syntax. That is, the expression
          df format now we can also use df.format(now)
    -It is also possible to inherit from Java classes and implement Java interfaces directly in Scala.

--Everything is an Object
   Scala is a pure object-oriented language in the sense that everything is an object, including numbers
or functions.
   It differs from Java in that respect, since Java distinguishes primitive types (such as boolean and
int) from reference types
   and does not enable one to manipulate functions as values

   --Functions are objects
     functions are also objects in Scala. It is therefore possible to pass functions as arguments,
         to store them in variables, and to return them from other functions.
         This ability to manipulate functions as values is one of the cornerstone of a very interesting
programming paradigm called functional programming

          object Timer {
                def oncePerSecond(callback: () => Unit) {
                        while (true) { callback(); Thread sleep 1000 }
                }
                def timeFlies() {
                        println("time flies like an arrow...")
                }
                def main(args: Array[String]) {
                        oncePerSecond(timeFlies)
                }
          }
   --Anonymous functions
     functions without a name
          object Timer {
                def oncePerSecond(callback: () => Unit) {
                        while (true) { callback(); Thread sleep 1000 }
                }
                def main(args: Array[String]) {
                         oncePerSecond(() =>
              println("time flies like an arrow..."))
                }
          }

--Classes
   Classes in Scala are declared using a syntax which is close to Java’s syntax.
   One important difference is that classes in Scala can have parameters

   class Complex(real: Double, imaginary: Double) {
        def re() = real
        def im() = imaginary
   }

   These arguments must be passed when creating an instance of class Complex, as follows: new
Complex(1.5, 2.3).
   The class contains two methods, called re and im, which give access to these two parts
   object ComplexNumbers {
                def main(args: Array[String]) {
                val c = new Complex(1.2, 3.4)
                println("imaginary part: " + c.im())
    }
   }

   --Methods without arguments
   class Complex(real: Double, imaginary: Double) {
        def re = real
        def im = imaginary
   }

   --Inheritance and overriding
    All the scala clases super type is Any
    It is possible to override methods inherited from a super-class in Scala.
        It is however mandatory to explicitly specify that a method overrides another one using the
override modifier,
        in order to avoid accidental overriding
        class Complex(real: Double, imaginary: Double) {
                def re = real
                def im = imaginary
                override def toString() =
                        "" + re + (if (im < 0) "" else "+") + im + "i"
        }

  --Implicit Classes
      Implicit classes must be defined inside another class/object/trait (not in top level).
          Implicit classes may only take one non -implicit argument in their constructor.
          Implicit classes may not be any method, member or object in scope with the same name as the
implicit class
          Implicit class is a class marked with ‘implicit’ keyword



--Genericity (similar to java generics)
    class Reference[T] {
        private var contents: T = _
                def set(value: T) { contents = value }
                def get: T = contents
        }
    _ represents a default value.
        This default value is 0 for numeric types, false for the Boolean type, () for the Unit type and
null for all object types

--Variables
   Scala has two types of variables
   val : is final variable, we can't reassign  (immutable variable)
   var : is non-final variable, we can reassign (mutable variable)

   ex : val test = "test";
         test = "test2" // compilation error
```

```
                 var test = "test";
          test = "test2" // no compilation error

        Variables are nothing but reserved memory locations to store values. This means that when you
create a variable, you reserve some space in memory.
        Based on the data type of a variable, the compiler allocates memory and decides what can be
stored in the reserved memory

        -Declaration
           var myVar : String = "Foo"
           syntax : val or val VariableName : DataType = [Initial Value]

        -Variable Type Inference
            var myVar = "Foo"
                When you assign an initial value to a variable, the Scala compiler can figure out the
type of the variable
                based on the value assigned to it. This is called variable type inference

    -Variable Scope
       1)Fields Like instance variable in java. can be immutable or mutable
           2)Method parameters :  can be immutable only
           3)Local variable : can be inside method.  can be immutable or mutable


--Scala runs on the JVM : Scala is compiled into Java Byte Code which is executed by the Java Virtual
Machine (JVM)

--Scala vs Java
        Scala has a set of features that completely differ from Java. Some of these are

        All types are objects
        Type inference
        Nested Functions
        Functions are objects
        Domain specific language (DSL) support
        Traits
        Closures
        Concurrency support inspired by Erlang

--Scala web framework : Play framework

--Scala datatypes : Scala has all the same data types as Java with same memory
    and it has extra datatypes  1)Unit : Corresponds to no value
    2)Null: null or empty reference
        3)Nothing : The subtype of every other type; includes no values
        4)Any : The supertype of any type; any object is of type Any
        5)AnyRef : The supertype of any reference type

--Multi-Line Strings
  A multi-line string literal is a sequence of characters enclosed in triple quotes """ ... """.
  ex : """hello
          how are you"""

--Access Modifiers
  Private Members && Protected Members same as java
  public Members : we can't specify explicitly members as public.
  If we not specify modifier for members then by default scope is public

  Scope of Protection : Access modifiers in Scala can be augmented with qualifiers. A modifier of the
form private[X] or protected[X]
  where X designates some enclosing package, class or singleton object

  package society {
   package professional {
      class Executive {
         private[professional] var workDetails = null //will be available in  professional package
         private[society] var friends = null //will be available in  society package
         private[this] var secrets = null ////will be available in  with in the class
      }
   }
 }

--Support all the operators what java support logical,arithmetic, relational, bitwise and assignment

--Same if else and while and do while loop like java

--For loop
  -for loop with ranges
    for( var x <- Range ){
                statement(s);
        }
        The left-arrow ← operator is called a generator, so named because it's generating individual
values from a range.
        Range could be a range of numbers and that is represented as i to j or sometime like i until j

    ex1:  for( a <- 1 to 10){
                        println( "Value of a: " + a );
                  }
    ex2:  for( a <- 1 until 10){
                        println( "Value of a: " + a );
                  }
        You can use multiple ranges separated by semicolon (;)
      ex : for( a <- 1 to 3; b <- 1 to 3){
                                println( "Value of a: " + a );
                                println( "Value of b: " + b );
                        }

   -for Loop with Collections
    for( var x <- List ){
                statement(s);
        }
                ex : val numList = List(1,2,3,4,5,6);
                var a = 0;
                for( a <- numList ){
                        println( "Value of a: " + a );
                }
   -for loop with Filters
     for( var x <- List
      if condition1; if condition2...
          ) {
                statement(s);
                }
             ex : for( a <- numList
                                        if a != 3; if a < 8 ){
                                                println( "Value of a: " + a );
                                        }

    -for loop with yield
          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 retVal = for{ var x <- List
                                                                if condition1; if condition2...
                                                        }
                                                  yield x

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

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

                        // Now print returned values using another loop.
                        for( a <- retVal){
                                        println( "Value of a: " + a );
                        }

-Functions
    A function is a group of statements that perform a task
    Scala function's name can also have characters like +, ++, ~, &,-, --, \, /, :, etc
```

```
        -Function Declarations
           def functionName ([list of parameters]) : [return type]
         Methods are implicitly declared abstract if you don’t use the equals sign and the method body.
    -Function Definitions
          def functionName ([list of parameters]) : [return type] = {
                                              function body
                                              return [expr]
          }

      Ex1 : object add {
                              def addInt( a:Int, b:Int ) : Int = {
                                      var sum:Int = 0
                                      sum = a + b
                              return sum
                              }
                      }
        -Functions Call-by-Name
          object Demo {
                def main(args: Array[String]) {
                        delayed(time());
                }

                def time() = {
                        println("Getting time in nano seconds")
                        System.nanoTime
                }
                def delayed( t: => Long ) = {
                        println("In delayed method")
                        println("Param: " + t)
                }
        }
        -Functions with Named Arguments
           object Demo {
                        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 );
                        }
                }
        -Function with Variable Arguments
          object Demo {
                  def main(args: Array[String]) {
                        printStrings("Hello", "Scala", "Python");
                  }

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

                        for( arg <- args ){
                         println("Arg value[" + i + "] = " + arg );
                        i = i + 1;
                   }
                 }
      }
        -Recursion Functions
      object Demo {
                        def main(args: Array[String]) {
                        for (i <- 1 to 10)
                                println( "Factorial of " + i + ": = " + factorial(i) )
                        }

                        def factorial(n: BigInt): BigInt = {
                        if (n <= 1)
                                        1
                        else
                                n * factorial(n - 1)
                        }
        }
    -Default Parameter Values for a Function
      object Demo {
                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
                }
        }
     -Nested Functions : We can write function with in function and then we can call that function in
that function
     -Anonymous Functions
        Ex1 : var inc = (x:Int) => x+1
              var x = inc(7)-1
        Ex2: var mul = (x: Int, y: Int) => x*y
             println(mul(3, 4))
        Ex3: var userDir = () => { System.getProperty("user.dir") }
             println( userDir )

     -Partially Applied Functions
       Ex1:  val date = new Date
                        val logWithDateBound = log(date, _ : String)
                        logWithDateBound("message1" )
         -Currying Functions
            Ex1 : def strcat(s1: String)(s2: String) = s1 + s2 or def strcat(s1: String) = (s2: String)
=> s1 + s2
                                strcat("foo")("bar")


-String Interpolation
  We can embed variable references directly in process string literal
  Ex1 : val name = “James”
        println(s “Hello, $name”) //output: Hello, James
  Ex 2: println(s “1 + 1 = ${1 + 1}”) //output: 1 + 1 = 2

-Traits
   -A trait encapsulates method and field definitions, which can then be reused by mixing them into
classes.
   -Unlike class inheritance, in which each class must inherit from just one superclass, a class can mix
in any number of traits.
   -Traits are used to define object types by specifying the signature of the supported methods.
   -Scala also allows traits to be partially implemented but traits may not have constructor parameters
   -A trait definition looks just like a class definition except that it uses the keyword trait
    Syntax:
        trait Equal {
                def isEqual(x: Any): Boolean
                def isNotEqual(x: Any): Boolean = !isEqual(x)
        }
        -Here, we have not given any implementation for isEqual where as another method has its
implementation.
        -Child classes extending a trait can give implementation for the un-implemented methods.
        -So a trait is very similar to what we have abstract classes in Java
         Ex :trait Equal {
                        def isEqual(x: Any): Boolean
                        def isNotEqual(x: Any): Boolean = !isEqual(x)
                }

                class Point(xc: Int, yc: Int) extends Equal {
                        var x: Int = xc
                        var y: Int = yc

                        def isEqual(obj: Any) = obj.isInstanceOf[Point] && obj.asInstanceOf[Point].x ==
y
                }
                object Demo {
                        def main(args: Array[String]) {
                        val p1 = new Point(2, 3)
                        val p2 = new Point(2, 4)
                        val p3 = new Point(3, 3)

                        println(p1.isNotEqual(p2))
                        println(p1.isNotEqual(p3))
                        println(p1.isNotEqual(2))
                        }
                }
     -Here obj.isInstanceOf [Point] To check Type of obj and Point are same are not.
      obj.asInstanceOf [Point] means exact casting by taking the object obj type and returns the same
obj as Point type.

        --Value classes and Universal Traits
       -Value classes are new mechanism in Scala to avoid allocating runtime objects.
           -It contains a primary constructor with exactly one val parameter.
           -It contains only methods (def) not allowed var, val, nested classes, traits, or objects.
           -Value class cannot be extended by another class. This can be possible by extending your
value class with AnyVal
           -A value class not allowed to extend traits.
           -To permit value classes to extend traits, universal traits are introduced which extends for
Any
           Ex : trait Printable extends Any {
                                def print(): Unit = println(this)
                        }
                        class Wrapper(val underlying: Int) extends AnyVal with Printable

                        object Demo {
                                def main(args: Array[String]) {
                                        val w = new Wrapper(3)
                                        w.print() // actually requires instantiating a Wrapper instance
                                }
                        }
        --When to Use Traits?
      -If the behavior will not be reused, then make it a concrete class. It is not reusable behavior
after all.
      -If it might be reused in multiple, unrelated classes, make it a trait. Only traits can be mixed
into different parts of the class hierarchy.
      -If you want to inherit from it in Java code, use an abstract class.
      -If you plan to distribute it in compiled form, and you expect outside groups to write classes
inheriting from it, you might lean towards using an abstract class.
      -If efficiency is very important, lean towards using a class.

-Exception Handling
   Same try/catch/finally block like java
   Scala doesn't have checked exceptions.
   Catch block will have exception case pattern matching
   try {
         val f = new FileReader("input.txt")
      } catch {
         case ex: FileNotFoundException => {
            println("Missing file exception")
         }

         case ex: IOException => {
            println("IO Exception")
         }
      } finally {
         println("Exiting finally...")
      }
-Pattern Matching
  A pattern match includes a sequence of alternatives, each starting with the keyword case.
  Each alternative includes a pattern and one or more expressions,
  which will be evaluated if the pattern matches. An arrow symbol => separates the pattern from the
expressions
  ex1 :
  object Demo {
   def main(args: Array[String]) {
      println(matchTest(3))
   }

   def matchTest(x: Int): String = x match {
      case 1 => "one"
      case 2 => "two"
      case _ => "many"
   }
 }
 o/p : many
 ex2 :
 object Demo {
   def main(args: Array[String]) {
      println(matchTest("two"))
      println(matchTest("test"))
      println(matchTest(1))
   }

   def matchTest(x: Any): Any = x match {
      case 1 => "one"
      case "two" => 2
      case y: Int => "scala.Int"
      case _ => "many"
   }
}
 o/p : 2
      many
          one
    --Matching using Case Classes
                The case classes are special classes that are used in pattern matching with case
expressions.
                Syntactically, these are standard classes with a special modifier: case.
                object Demo {
   def main(args: Array[String]) {
      val alice = new Person("Alice", 25)
      val bob = new Person("Bob", 32)
      val charlie = new Person("Charlie", 32)

      for (person <- List(alice, bob, charlie)) {
         person match {
            case Person("Alice", 25) => println("Hi Alice!")
            case Person("Bob", 32) => println("Hi Bob!")
            case Person(name, age) => println(
               "Age: " + age + " year, name: " + name + "?")
         }
      }
   }
   case class Person(name: String, age: Int)
}
  First, the compiler automatically converts the constructor arguments into immutable fields (vals). The
val keyword is optional.
  If you want mutable fields, use the var keyword. So, our constructor argument lists are now shorter.

  Second, the compiler automatically implements equals, hashCode, and toString methods to the class,
  which use the fields specified as constructor arguments. So, we no longer need our own toString()
methods.

-Regular Expressions
   -Regex class available in the scala.util.matching package.
    import scala.util.matching.Regex

        object Demo {
                def main(args: Array[String]) {
                        val pattern = "Scala".r
                        val str = "Scala is Scalable and cool"

                        println(pattern findFirstIn str)
                }
        }
        -We create a String and call the r( ) method on it.
```

```
         Scala implicitly converts the String to a RichString and invokes that method to get an instance
of Regex
        -To find a first match of the regular expression, simply call the findFirstIn() method
    -To find all occurrences of the matching word, we can use the findAllIn( ) method and in case there
are multiple Scala words available in the target string,
          this will return a collection of all matching words

    -you can use a pipe (|) to search small and capital case of Scala and
          you can use Regex constructor instead or r() method to create a pattern
          import scala.util.matching.Regex

                object Demo {
                        def main(args: Array[String]) {
                                val pattern = new Regex("(S|s)cala")
                                val str = "Scala is scalable and cool"

                                println((pattern findAllIn str).mkString(","))
                        }
                }
    -If you would like to replace matching text, we can use replaceFirstIn( ) to replace the first match
          or replaceAllIn( ) to replace all occurrences

    --Forming Regular Expressions
          -Scala inherits its regular expression syntax from Java, which in turn inherits most of the
features of Perl
          Subexpression | Matches
          -----------------------
                ^  Matches beginning of line.
                $  Matches end of line.
                .  Matches any single character except newline. Using m option allows it to match
newline as well.
                [...]  Matches any single character in brackets.
                [^...]  Matches any single character not in brackets
                \\A    Beginning of entire string
                \\z   End of entire string
                \\Z   End of entire string except allowable final line terminator.
                re*   Matches 0 or more occurrences of preceding expression.
                re+   Matches 1 or more of the previous thing
                re?   Matches 0 or 1 occurrence of preceding expression.
                re{ n}   Matches exactly n number of occurrences of preceding expression.
                re{ n,}  Matches n or more occurrences of preceding expression.
                re{ n, m}  Matches at least n and at most m occurrences of preceding expression.
                a|b  Matches either a or b.
                (re)  Groups regular expressions and remembers matched text.
                (?: re)  Groups regular expressions without remembering matched text.
                (?> re)  Matches independent pattern without backtracking.
                \\w  Matches word characters.
        \\W  Matches nonword characters.
                \\s  Matches whitespace. Equivalent to [\t\n\r\f].
                \\S  Matches nonwhitespace.
                \\d  Matches digits. Equivalent to [0-9].
                \\D  Matches nondigits.
                \\A  Matches beginning of string.
                \\Z  Matches end of string. If a newline exists, it matches just before newline.
                \\z  Matches end of string.
                \\G   Matches point where last match finished.
                \\n  Back-reference to capture group number "n"
                \\b  Matches word boundaries when outside brackets. Matches backspace (0x08) when inside
brackets.
                \\B  Matches nonword boundaries.
                \\n, \\t, etc.  Matches newlines, carriage returns, tabs, etc.
                \\Q  Escape (quote) all characters up to \\E
                \\E  Ends quoting begun with \\Q

    Regular-Expression Examples
        ----------------------------
                . Match any character except newline
                [Rr]uby Match "Ruby" or "ruby"
                rub[ye] Match "ruby" or "rube"
                [aeiou] Match any one lowercase vowel
                [0-9]  Match any digit; same as [0123456789]
                [a-z]  Match any lowercase ASCII letter
                [A-Z]  Match any uppercase ASCII letter
                [a-zA-Z0-9]  Match any of the above
                [^aeiou]  Match anything other than a lowercase vowel
                [^0-9]  Match anything other than a digit
                \\d  Match a digit: [0-9]
                \\D   Match a nondigit: [^0-9]
                \\s  Match a whitespace character: [ \t\r\n\f]
                \\S  Match nonwhitespace: [^ \t\r\n\f]
                \\w  Match a single word character: [A-Za-z0-9_]
                \\W  Match a nonword character: [^A-Za-z0-9_]
                ruby?  Match "rub" or "ruby": the y is optional
                ruby*  Match "rub" plus 0 or more ys
                ruby+  Match "rub" plus 1 or more ys
                \\d{3}  Match exactly 3 digits
                \\d{3,}  Match 3 or more digits
                \\d{3,5}  Match 3, 4, or 5 digits
                \\D\\d+  No group: + repeats \\d
                (\\D\\d)+/   Grouped: + repeats \\D\d pair
                ([Rr]uby(, )?)+ Match "Ruby", "Ruby, ruby, ruby", etc.

-Files IO
        Scala is open to make use of any Java objects and
        java.io.File is one of the objects which can be used in Scala programming to read and write
files
        Ex : import java.io._
    object Demo {
                def main(args: Array[String]) {
                        val writer = new PrintWriter(new File("test.txt" ))

                        writer.write("Hello Scala")
                        writer.close()
                }
        }
        It will create a file named Demo.txt in the current directory and writes Hello Scala and prints
in the console

        -Reading a Line from Command Line
         Ex : object Demo {
                                def main(args: Array[String]) {
                                print("Please enter your input : " )
                                val line = Console.readLine

                                println("Thanks, you just typed: " + line)
                                }
                        }
        -Reading File Content
      You can use Scala's Source class and its companion object to read files
      Ex : import scala.io.Source
                        object Demo {
                                        def main(args: Array[String]) {
                                        println("Following is the content read:" )

                                        Source.fromFile("Demo.txt" ).foreach {
                                                                        print
                                                        }
                                        }
                        }
======================================
//Number operations
//Ranges
//creates a range between 1 to 10 inclusive
val range = 1 to 10
//creates a range between 1 to 10 exclusive
val range2 = 1 until 10
//from 2 to 10 with jumps of 3
val range3 = 2 until 10 by 3

println(range3.toList) //List(2, 5, 8)
```

```
//Number convinience methods
val num = -5
val numAbs = num.abs //absolute value
val max5or7 = numAbs.max(7)
val min5or7 = numAbs.min(7)
println(numAbs) //5
println(max5or7) //7
println(min5or7) //5

//String operations

val reverse = "Scala".reverse //reverse a string
println(reverse) //alacS

val cap = "scala".capitalize //make first char caps
println(cap) //Scala

val multi = "Scala!" * 7 //repeat n times
println(multi) //Scala!Scala!Scala!Scala!Scala!Scala!Scala!

val int = "123".toInt //parse as Int
println(int)

//Useful methods on collections

//filter - keep only items larger than 4
val moreThan4 = range.filter(_ > 4)
println(moreThan4) //Vector(5, 6, 7, 8, 9, 10)

//map - transform each item in the collection
val doubleIt = range2.map(_ * 2)
println(doubleIt) //Vector(2, 4, 6, 8, 10, 12, 14, 16, 18)

Ex2 : def add(x:Int, y:Int) = { //result type is inferred
  x + y //"return" keyword is optional
}
println(add(42,13))

Ex3 : //Curly braces are optional on single line blocks
def add(x:Int, y:Int) = x + y
println(add(42,13))

Ex4: Anonymous function
def add1(x:Int, y:Int) = x + y //method
val add2 = (x:Int, y:Int) => x + y //anonymous function
val add3:(Int,Int)=>Int = _ + _ //alternate way
val add4 = (_ + _):(Int,Int)=>Int //alternate way, rare

Ex5 : Assign multiple variables
var (x, y, z, c, python, java) = (1, 2, 3, true, false, "no!")

Ex6: Loops without loops
(0 until 10).sum

Ex7: //Mutable array of type Array[Int]
val array1 = Array(1, 2, 3)   //>array1 = [I@6ae01647
printArray(array1)//>Array(1, 2, 3)
//Mutable array of type Array[Any]
val array2 = Array("a", 2, true)   //>array2 = [Ljava.lang.Object;@1dd6c622
printArray(array2)//>Array(a, 2, true)
//Mutable array of type Array[String]
val array3 = Array("a", "b", "c")   //>array3 = [Ljava.lang.String;@7f69f17b
printArray(array3)  //>Array(a, b, c)
//Access items using (index) not [index]
val itemAtIndex0 = array3(0)   //>itemAtIndex0 = a

//Modify items the same way
array3(0) = "d"
printArray(array3)  //>Array(d, b, c)
```

```
//Concatenation using the ++ operator,
//Prepending items using +: and appending using :+
val concatenated = "prepend" +: (array1 ++ array2) :+ "append"  //>concatenated =
[Ljava.lang.Object;@46d0397
printArray(concatenated)  //>Array(prepend, 1, 2, 3, a, 2, true, append)

//Finding an index of an item
array3.indexOf("b") //>1

//Diff
val diffArray = Array(1,2,3,4).diff(Array(2,3))  //>diffArray = [I@1106b0c6
printArray(diffArray) //>Array(1, 4)

//Find (stops when item is found)
val personArray = Array(("Alice",1), ("Bob",2), ("Carol",3))  //>personArray = [Lscala.Tuple2;@4e3f9fe5
def findByName(name:String) = personArray.find(_._1 == name).getOrElse(("David",4))  //>findByName(name
= "foo") => (David,4)
val findBob = findByName("Bob")  //>findBob = (Bob,2)
val findEli = findByName("Eli")  //>findEli = (David,4)

val bobFound = findBob._2  //>bobFound = 2
val eliFound = findEli._2  //>eliFound = 4
==================================================================================
Ex8: List
//Immutable list of type List[Int]
val list1 = List(1, 2, 3) //> list1 = List(1, 2, 3)
//Immutable list of type List[Any]
val list2 = List("a", 2, true) //> list2 = List(a, 2, true)
import collection.mutable
//the "mutable version" of List
val mlist = mutable.ArrayBuffer("a", "b", "c") //> mlist = ArrayBuffer(d, b, e, f, g)

//Access items using (index) not [index]
val firstItem = list1(0) //> firstItem = 1

//Modify items the same way  (mutable Lists only)
mlist(0) = "d"
mlist //> ArrayBuffer(d, b, e, f, g)

//Concatenation using the ++ operator or ::: (lists only)
list1 ++ list2 //> List(1, 2, 3, a, 2, true)
list1 ::: list2 //> List(1, 2, 3, a, 2, true)

//Prepending an item using either :: (lists only) or +:
0 :: list1 //> List(0, 1, 2, 3)
0 +: list1 //> List(0, 1, 2, 3)

//appending an item using :+ (not efficient for immutable List)
list1 :+ 4 //> List(1, 2, 3, 4)

//all together
val concatenated = 1 :: list1 ::: list2 ++ mlist :+ 'd' //> concatenated = List(1, 1, 2, 3, a, 2, true,
d, b, c, d)
//concatenation doesn't modify the lists themselves
list1 //> List(1, 2, 3)

//Removing elements (mutable list only, creates a new array):

//creates a new array with "c" removed, mlist is not touched
mlist - "c" //> ArrayBuffer(d, b)
//creates a new array with e, f removed, mlist is not touched
mlist -- List("e", "f") //> ArrayBuffer(d, b, c)
//mlist not modified
mlist //> ArrayBuffer(d, b, e, f, g)
//Removing elements (mutable Lists only):

//removes c from the list itself
mlist -= "c" //> ArrayBuffer(d, b, e, f, g)
mlist //> ArrayBuffer(d, b, e, f, g)
```

```
//removes e and f from mlist itself
mlist --= List("e", "f") //> ArrayBuffer(d, b, e, f, g)
mlist //> ArrayBuffer(d, b, e, f, g)

//Adding elements (mutable Lists only)
mlist += "e" //> ArrayBuffer(d, b, e, f, g)
mlist ++= List("f", "g") //> ArrayBuffer(d, b, e, f, g)

mlist //ArrayBuffer(d, b, e, f, g) //> ArrayBuffer(d, b, e, f, g)

//Diff
val diffList = List(1,2,3,4) diff List(2,3) //> diffList = List(1, 4)

//Find (stops when item is found)
val personList = List(("Alice",1), ("Bob",2), ("Carol",3)) //> personList = List((Alice,1), (Bob,2),
(Carol,3))
def findByName(name:String) = personList.find(_._1 == name).getOrElse(("David",4)) //> findByName(name =
"foo") => (David,4)
val findBob = findByName("Bob") //> findBob = (Bob,2)
val findEli = findByName("Eli") //> findEli = (David,4)

findBob._2 //> 2
findEli._2 //> 4

Ex9: Set
val set1 = Set(1, 2, 3) //Immutable set of type Set[Int]
val set2 = Set("a", 2, true) //Immutable list of type Set[Any]
import collection.mutable
val mset = mutable.HashSet("a", "b", "c") //the "mutable version" of Set

//Sets remove duplicates
println(Set(1,2,3,2,4,3,2,1,2)) //Set(1, 2, 3, 4)

//Check if item exists using (item)
val oneExists = set1(1)
val fourExists = set1(4)
println(oneExists) // true
println(fourExists) // false

//You can "modify" items the same way as for Lists
//(DON'T use it this way, use mset -="a" instead)
mset("a") = false
println(mset) //Set(c, b)

//Concatenation using the ++ operator
//(removes duplicates, order not guaranteed)
val concatenated = set1 ++ set2 ++ mset
println(concatenated) // Set(1, a, true, 2, b, 3, c)
//Concatenation doesn't modify the sets themselves
println(set1) //Set(1, 2, 3)

//Removing elements (mutable Sets only)
mset -= "c"
println (mset) //Set("b")

//Adding elements (mutable Lists only)
mset += "e"
mset ++= Set("f", "g")

println (mset) //Set(f, g, e, b)

//Diff
val diffSet = Set(1,2,3,4) diff Set(2,3)
println(diffSet) // Set(1, 4)

//Find (stops when item is found)

//Note that this is not an ideal use for Set,
//a Map would be much better data structure
//Just for illustration purposes
```

```
val personSet = Set(("Alice",1), ("Bob",2), ("Carol",3))
def findByName(name:String) = personSet.find(_._1 == name).getOrElse(("David",4))
val findBob = findByName("Bob")
val findEli = findByName("Eli")

println(findBob._2) //2
println(findEli._2) //4

Ex9 : Map
val map1 = Map("one" -> 1, "two" -> 2, "three" -> 3)
//Map of type Map[String, Int]
val map2 = Map(1 -> "one", "2" -> 2.0, 3.0 -> false)
//Map of type Map[Any, Any]

import collection.mutable
val mmap = mutable.HashMap("a" -> 1, "b" -> 2 , "c" -> 3)
//the "mutable version" of Map

//Maps remove duplicate keys:
println(Map("a" -> 1, "a" -> 2)) //Map(a -> 2)

//Get items using map(key)
val one = map1("one")

//NoSuchElementException will be thrown if key doesn't exist!
//e.g. this code: val fourExists = map1("four")
//throws NoSuchElementException: key not found: four
//the get method returns an Option, which will be explained later
val fourExistsOption = map1.get("four")

println(one) // 1
println(fourExistsOption.isDefined) // false

//You can set / modify items using map(key) = value
mmap("d") = 4
println(mmap) //Map(b -> 2, d -> 4, a -> 1, c -> 3)

//Concatenation using the ++ operator
//(removes duplicate keys, order not guaranteed)
val concatenated = map1 ++ map2 ++ mmap
println(concatenated)
// Map(three -> 3, 1 -> one, two -> 2, a -> 1, b -> 2, 3.0 -> false, 2 -> 2.0, c -> 3, one -> 1, d -> 4)
//Concatenation doesn't modify the maps themselves
println(map1) //Map(one -> 1, two -> 2, three -> 3)

//Removing elements (mutable Sets only)
mmap -= "c"
println (mmap) //Map(b -> 2, d -> 4, a -> 1)

//Adding elements (mutable Lists only)
mmap += "e" -> 5
mmap ++= Map("f" -> 6, "g" -> 7)

println (mmap) //Map(e -> 5, b -> 2, d -> 4, g -> 7, a -> 1, f -> 6)

//Find
val personMap = Map(("Alice",1), ("Bob",2), ("Carol",3))
def findByName(name:String) = personMap.getOrElse(name, 4)
val findBob = findByName("Bob")
val findEli = findByName("Eli")

println(findBob) //2
println(findEli) //4

Ex10: Mutable Collections
import scala.collection.mutable

val arrayBuffer = mutable.ArrayBuffer(1, 2, 3)
val listBuffer = mutable.ListBuffer("a", "b", "c")
val hashSet = mutable.Set(0.1, 0.2, 0.3)
val hashMap = mutable.Map("one" -> 1, "two" -> 2)

Operations :
arrayBuffer += 4
listBuffer += "d"
arrayBuffer -= 1
listBuffer -= "a"
hashMap += "four" -> 4
hashMap -= "one"

arrayBuffer ++= List(5, 6, 7)
hashMap ++= Map("five" -> 5, "six" -> 6)
hashMap --= Set("one", "three")

println(arrayBuffer)
println(listBuffer)
println(hashMap)

Immutable collections with var
import scala.collection.mutable

var immutableSet = Set(1, 2, 3)

immutableSet += 4
//this is the same as:
immutableSet = immutableSet + 4

//compare to
val mutableSet = mutable.Set(1, 2, 3)

mutableSet += 4
// this is the same as:
mutableSet.+=(4)

println(immutableSet, mutableSet)
=================================================================================================
//Simple class that does nothing
class Person(fname:String, lname:String)
val p1 = new Person("Alice", "In Chains")
//p1.fname / lname is not accessible

//A class with a method
class Person2(fname:String, lname:String){
  def greet = s"Hi $fname $lname!"
}
val p2 = new Person2("Bob", "Marley")
println(p2.greet)
//p2.fname / lname is not accessible

//A class with a public read only variable
class Person3(fname:String, lname:String){
  // a public read only field
  val fullName = s"$fname $lname"
  def greet = s"Hi $fullName!"
}

val p3 = new Person3("Carlos", "Santana")
println(p3.greet)
println(p3.fullName)
//p3.fname / lname is not accessible

//auto creates a getter for fname, and getter + setter to lname
class Person4(val fname:String, var lname:String)

val p4 = new Person4("Dave", "Matthews") {
  //override the default string representation
  override def toString = s"$fname $lname"
}
println(p4.fname)
println(p4.lname)
```

```
//lname is defined as var, so it has a setter too
p4.lname = "Grohl"
println(p4)
```

Scala’s getters and setters use the principle of uniform access, e.g. if you change the implentation of a field declared var name to a method def name you will not need to recompile the code
Therefore there can’t be a variable or method (def, val or var, private or public) that has the same name in a class

```
//A full Java boilerplate style class (not idiomatic Scala!)
class JPerson() {
  var _name: String = null
  def this(_name:String) = {
    this()
         this._name = _name
  }
  //Scala style getters and setters
  def name_=(_name:String) = this._name = _name
  def name = this._name

  //Java style getters and setters
  def getName() = name
  def setName(name:String) = this.name = name
}

//Which can be generated in 1 line of idiomatic Scala
import beans._
class SPerson(@BeanProperty var name:String)
//Note: @BeanProperty is optional
//(only if you need Java style getters and setters)

val jp = new JPerson("Java Style")
val sp = new SPerson("Scala Style")

println(jp.name)
println(sp.name)

jp.name += " sucks!"
sp.name += " rocks!"

println(jp.getName)
println(sp.getName)
```

```
-What is the difference between :: and ::: operators
  ::  prepends an item to list meaning this item top on the list
  ::: prepends list to list

  Ex : val list1 = List(1,2)
       val list 2 = 0 :: list1  o/p is List(0, 1, 2)

           val list3 = list1 ::: list3  o/p is List(1, 2, 0, 1, 2)
```

http://aperiodic.net/phil/scala/s-99/

http://scalatutorials.com/tour/interactive_tour_of_scala_arrays.html