**Scala Exception Handling:**

The purpose of Exception Handling is to handle aberrant Conditions, can interrupt our program at any time unexpectedly.

Unlike java, Scala has only UnChecked Exception which is checked at runtime. All Exceptions are unchecked in it. We can handle Exception in different ways Like try-catch, throw, Throws, and many more. Let see how.

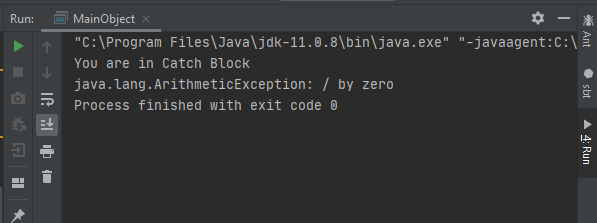
**Try-Catch Exception Handling**

In, Try we used to enclose the code to check the exception if exist we used catch block to handle the abnormal exception to prevent from terminating the program unexpectedly.

**Example**

class ErrorHandling{  
 def div(a:Int, b:Int) = {  
 try{  
 a/b  
 }catch{  
  
 case e: ArithmeticException => {  
*println*(e)  
 }  
 }  
*println*(" Rest Executed !!!!!")  
 }  
}  
object MainObject{  
 def main(args:Array[String]){  
 var e = new ErrorHandling()  
e.div(5,0)  
 }  
}

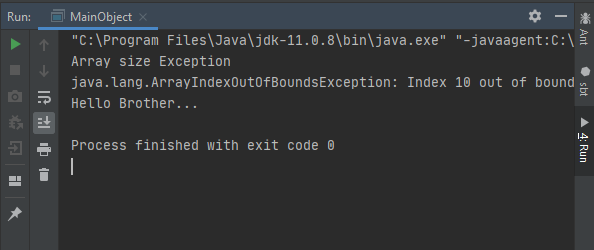
**Output**



Note: we can have multiple cases in catch to handle the exception also if at initial stage Exception occur it will terminate the program which means if other exceptions left in try will not run.

class ExceptionExample{  
 def divide(a:Int, b:Int) = {  
 try{  
 var arr = *Array*(1,2)  
arr(10)  
 a/b  
 }catch{  
 case a: ArithmeticException =>*println*(a)  
 case b: Throwable =>*println*("Array size Exception \n"+ b)  
 }  
*println*("Hello Brother...")  
 }  
}  
object MainObject{  
 def main(args:Array[String]){  
 var e = new ExceptionExample()  
e.divide(100,0)  
  
 }  
}

**Output**



Note: Here java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 2 error will be catch first and java.lang.ArithmeticException: / by zero will be ignored.

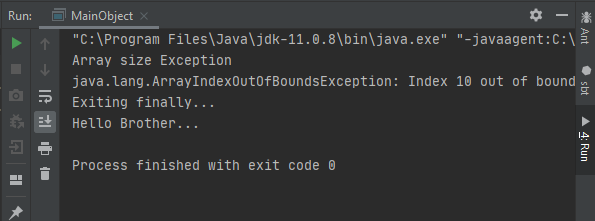
**Final Block in Exception Handling:**

Code inside the final block will be executed no matters how your program ends or terminate. Final Block can only be defined in Exception handling, not as a function, and must declare just after the try-catch block.

**Example**

class ExceptionExample{  
 def divide(a:Int, b:Int) {  
 try{  
 var arr = *Array*(1,2)  
arr(10)  
 a/b  
 } catch {  
 case a: ArithmeticException =>*println*(a)  
 case b: Throwable =>*println*("Array size Exception \n"+ b)  
 }  
 finally{  
*println*("Exiting finally...")  
 }  
*println*("Hello Brother...")  
 }  
}  
object MainObject{  
 def main(args:Array[String]){  
 var e = new ExceptionExample()  
e.divide(100,0)  
 }  
}

**Output**



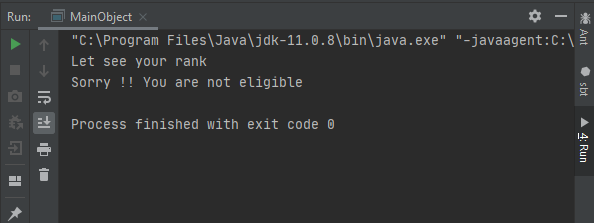
**Throw Keyword:**

Mainly we can create custom exceptions explicitly using this keyword. Means can have our own exception handling where we required. Exceptions classes are the same as java.

**Example**

import java.sql.SQLException  
  
class Statusneo{  
 def Select(Total\_Rank:Int)={  
 if(Total\_Rank< 10)  
 throw new SQLException("Hey ! Welcome for this intern")  
 else  
*println*("Sorry !! You are not eligible")  
 }  
 }  
  
 object MainObject{  
 def main(args:Array[String]){  
 var S = new Statusneo()  
*println*("Let see your rank")  
S.Select(10)  
 }  
 }

**Output**



**ScalaThrows Keyword :**

When a method needs to throw the specific Exception then only we use Throws. A method can have multiple Exceptions which can also be known as custom Exception.

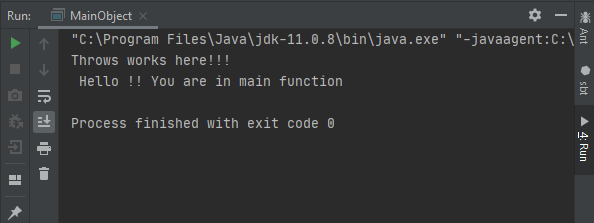
Syntax :

@throws(classOf[Exception])  
 def Intern{  
 // exception throwing code here ...  
}

**Example**

class ErrorHandling{  
 @throws(classOf[NumberFormatException])  
 def LetsTry(a:Int, b:Int)={  
 a/b  
*println*(" Hey !! Remember Work hard ")  
 }  
}  
object MainObject{  
 def main(args:Array[String]){  
 var E = new ErrorHandling()  
 try{  
E.LetsTry(10, 0)  
 }catch{  
  
 case x : ArithmeticException =>*println*("Throws works here!!! ")  
 }  
*println*(" Hello !! You are in main function")  
 }  
}

**Output**



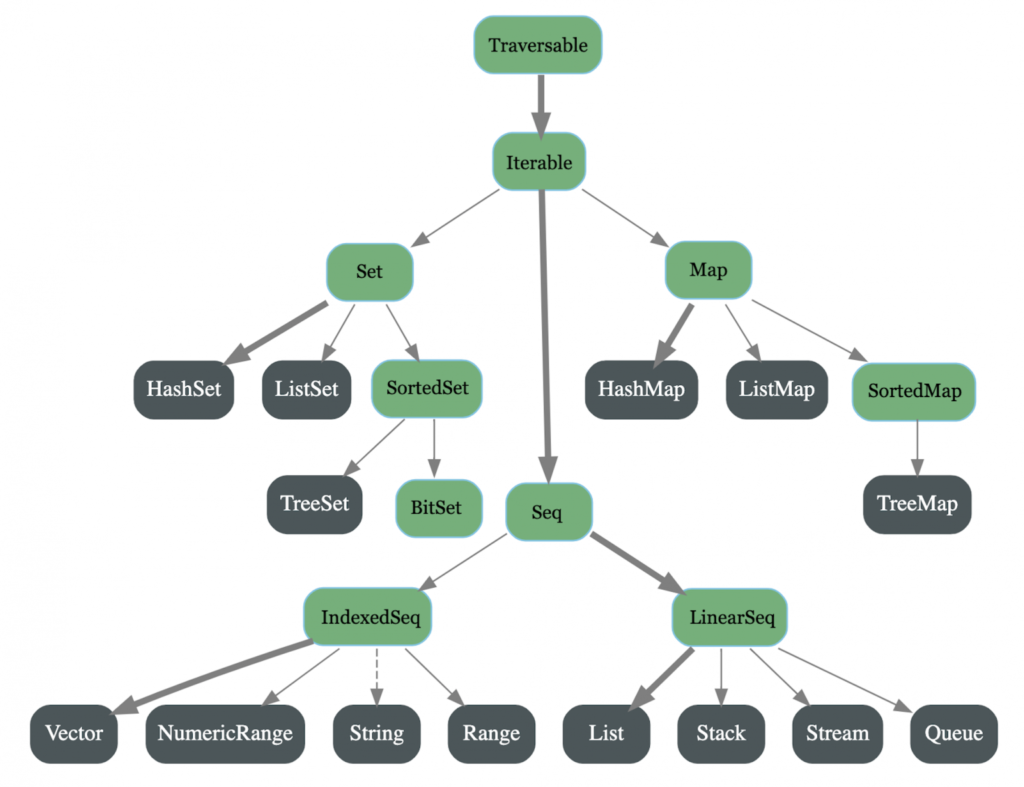
**Scala Collection:**

It is type of library which have classes and traits in it to collect data. These can be of two types mutable or immutable according to our requirement

* **Scala.collection.mutable** (this package contain all mutable collections)
* **Scala.collection.immutable** (this package contain all immutable collections)

The main difference between mutable and immutable package is that we can modify our data (add, remove and update) using mutable package but not with immutable

**Scala Immutable Collections Hierarchy**



**Scala Set**

It is used to store elements in a set and it does not have to follow any order. We can also perform various in sets

**Syntax**

**val** variableName = Set(element1, element2,... elementN)

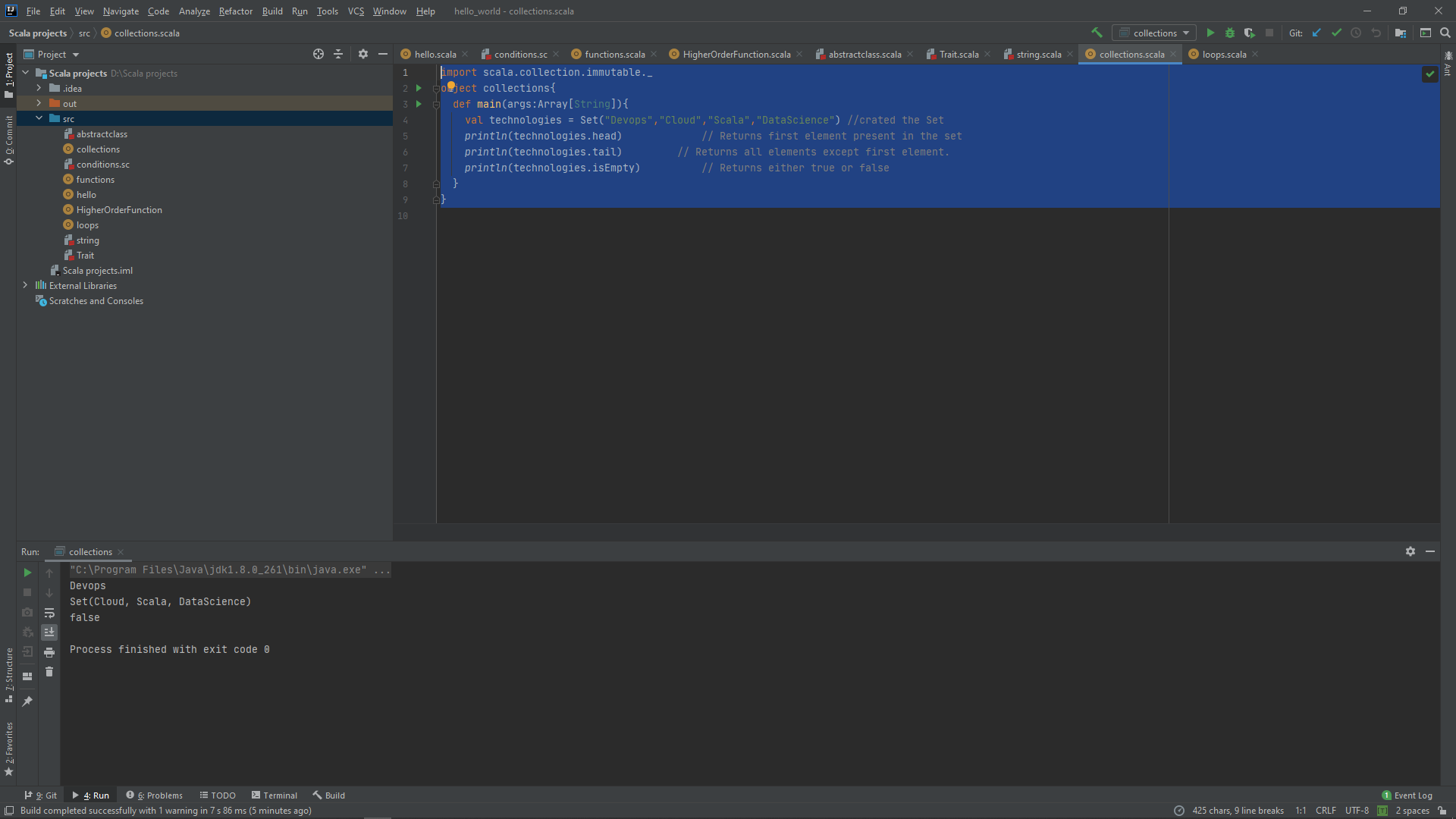
**set creation**

we can understand it with an example where we have created a simple set and used some inbuilt function to know about the properties of set

**Example**

import scala.collection.immutable.\_  
object collections{  
 def main(args:Array[String]){  
val technologies = Set("Devops","Cloud","Scala","DataScience") //created the Set  
*println*(technologies.head) // Returns first element present in the set  
*println*(technologies.tail) // Returns all elements except first element.  
*println*(technologies.isEmpty) // Returns either true or false  
 }  
}

**Output**



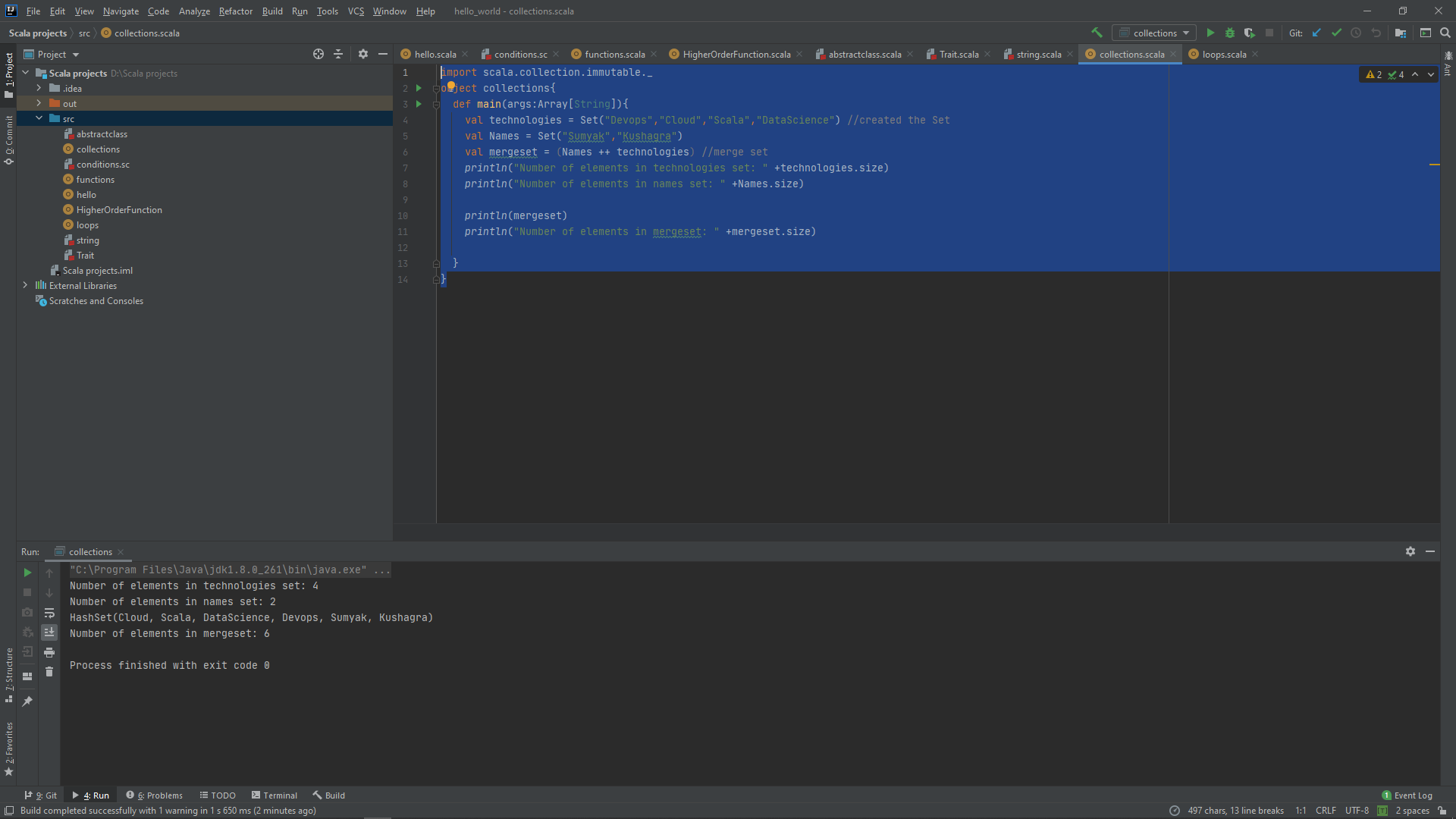
**Merge set**

We can merge two sets into one singleton set lets understand with it an example

**Example**

import scala.collection.immutable.\_  
object collections{  
 def main(args:Array[String]){  
val technologies = Set("Devops","Cloud","Scala","DataScience") //created the Set  
val Names = Set("Sumyak","Kushagra")  
valmergeset = (Names ++ technologies) //merge set  
*println*("Number of elements in technologies set: " +technologies.size)  
*println*("Number of elements in names set: " +Names.size)  
  
*println*(mergeset)  
*println*("Number of elements in mergeset: " +mergeset.size)  
  
 }  
}

**Output**



**NOTE:** Here you can see in the output it’s given **Hashset**it is just a sealed class which uses hash code to store elements. In Scala, A concrete implementation of Set semantics is known HashSet

**Adding and removing of element from set**

We can add or remove the element from set using

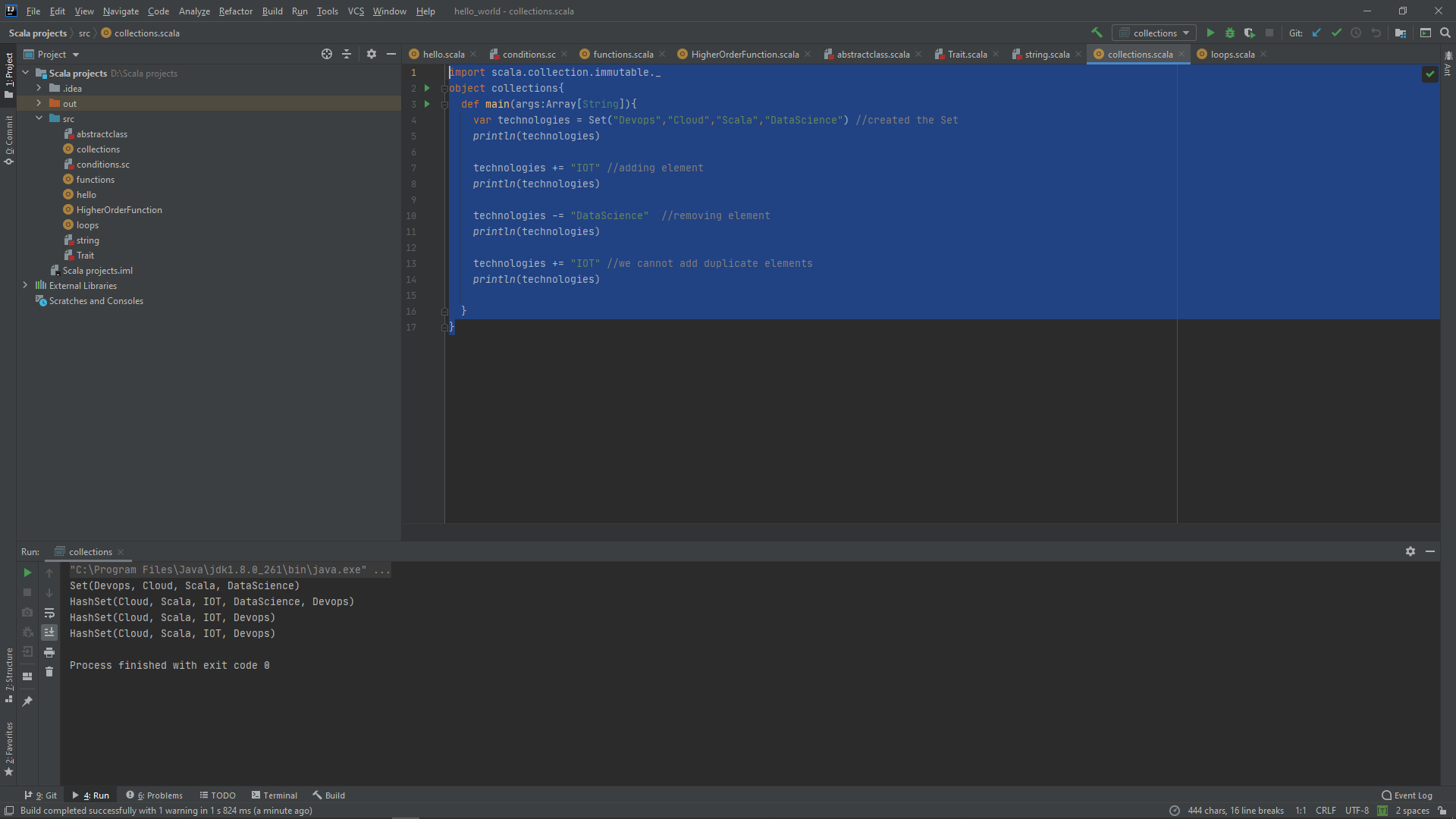
* **+=** 🡪 add element
* **-=** 🡪 remove element

Also, we cannot add similar elements in the set because duplicacy in set is not allowed

**Example**

import scala.collection.immutable.\_  
object collections{  
 def main(args:Array[String]){  
 var technologies = Set("Devops","Cloud","Scala","DataScience") //created the Set  
*println*(technologies)  
  
 technologies += "IOT" //adding element  
*println*(technologies)  
  
 technologies -= "DataScience" // removing element  
*println*(technologies)  
  
 technologies += "IOT" //duplicacy not allowed  
*println*(technologies)  
  
 }  
}

**Output**



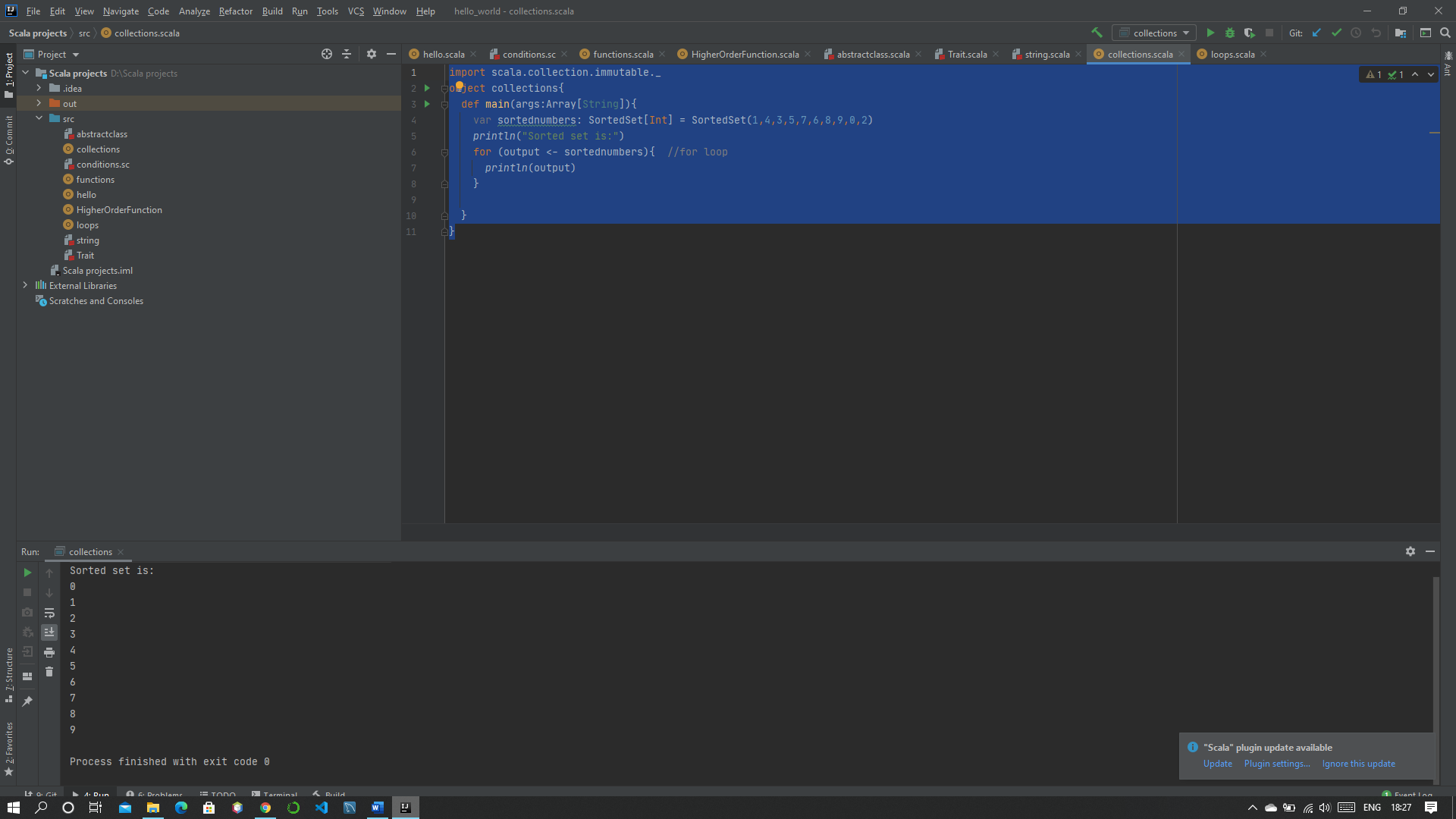
**Scala Sorted set**

This extends set trait and in return provides us the sorted set. We can either sort integer or string using this according to our requirements.

**Example**

import scala.collection.immutable.\_  
object collections{  
 def main(args:Array[String]){  
 var sortednumbers: SortedSet[Int] = SortedSet(1,4,3,5,7,6,8,9,0,2)  
*println*("Sorted set is:")  
 for (output <- sortednumbers){ //for loop  
*println*(output)  
 }  
  
 }  
}

**Output**



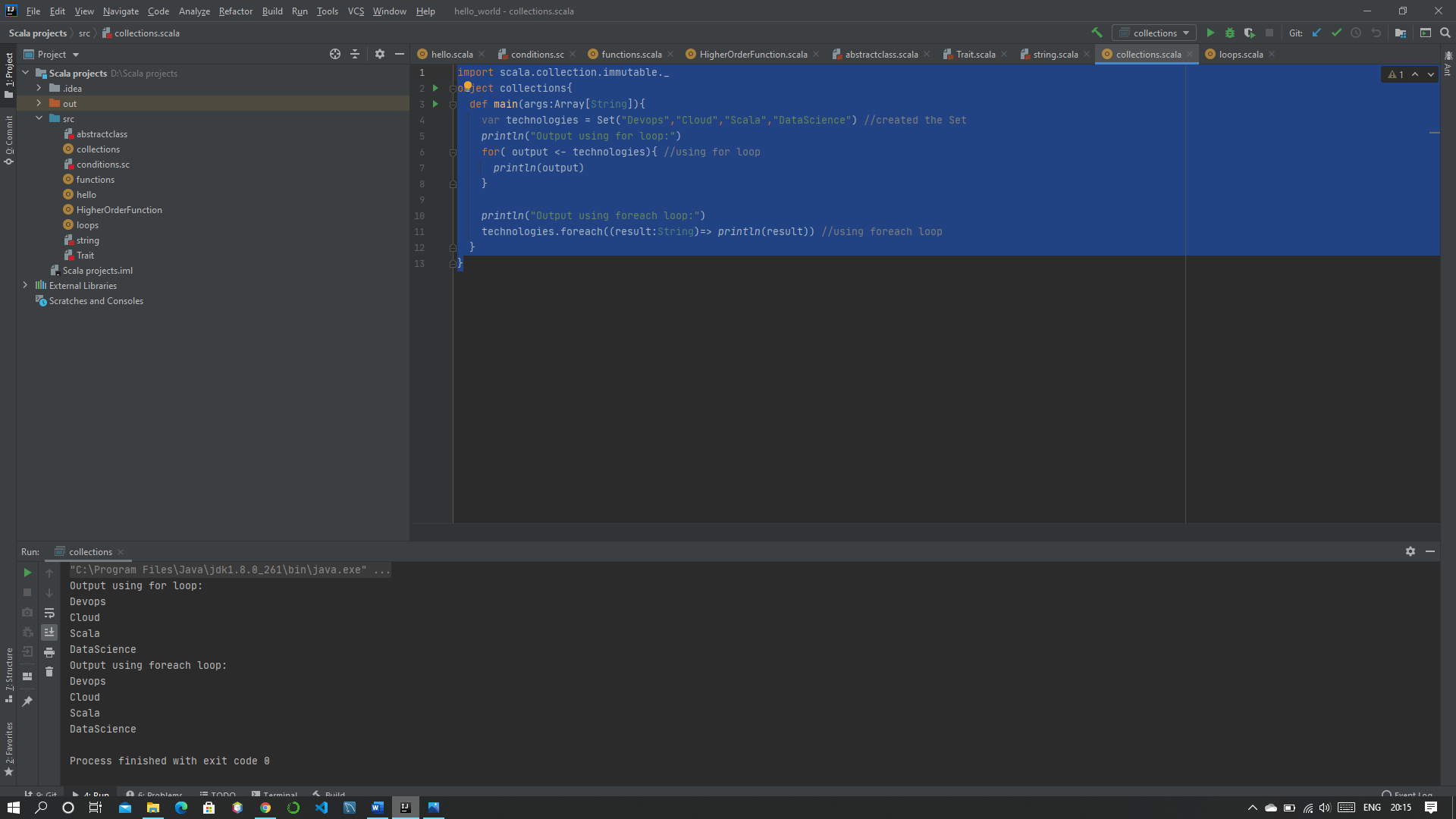
**Iterating set elements using loop**

We iterate elements using for loop or foreach loop in a set

**Example**

import scala.collection.immutable.\_  
object collections{  
 def main(args:Array[String]){  
 var technologies = Set("Devops","Cloud","Scala","DataScience") //created the Set  
*println*("Output using for loop:")  
 for( output <- technologies){ //using for loop  
*println*(output)  
 }  
  
*println*("Output using foreach loop:")  
technologies.foreach((result:String)=>*println*(result)) //using foreach loop  
 }  
}

**Output**



**Scala Seq**

It is a type of trait which follow indexed sequence to represent and also, we can access the elements using indexes also which maintains the order of insertion for our elements.

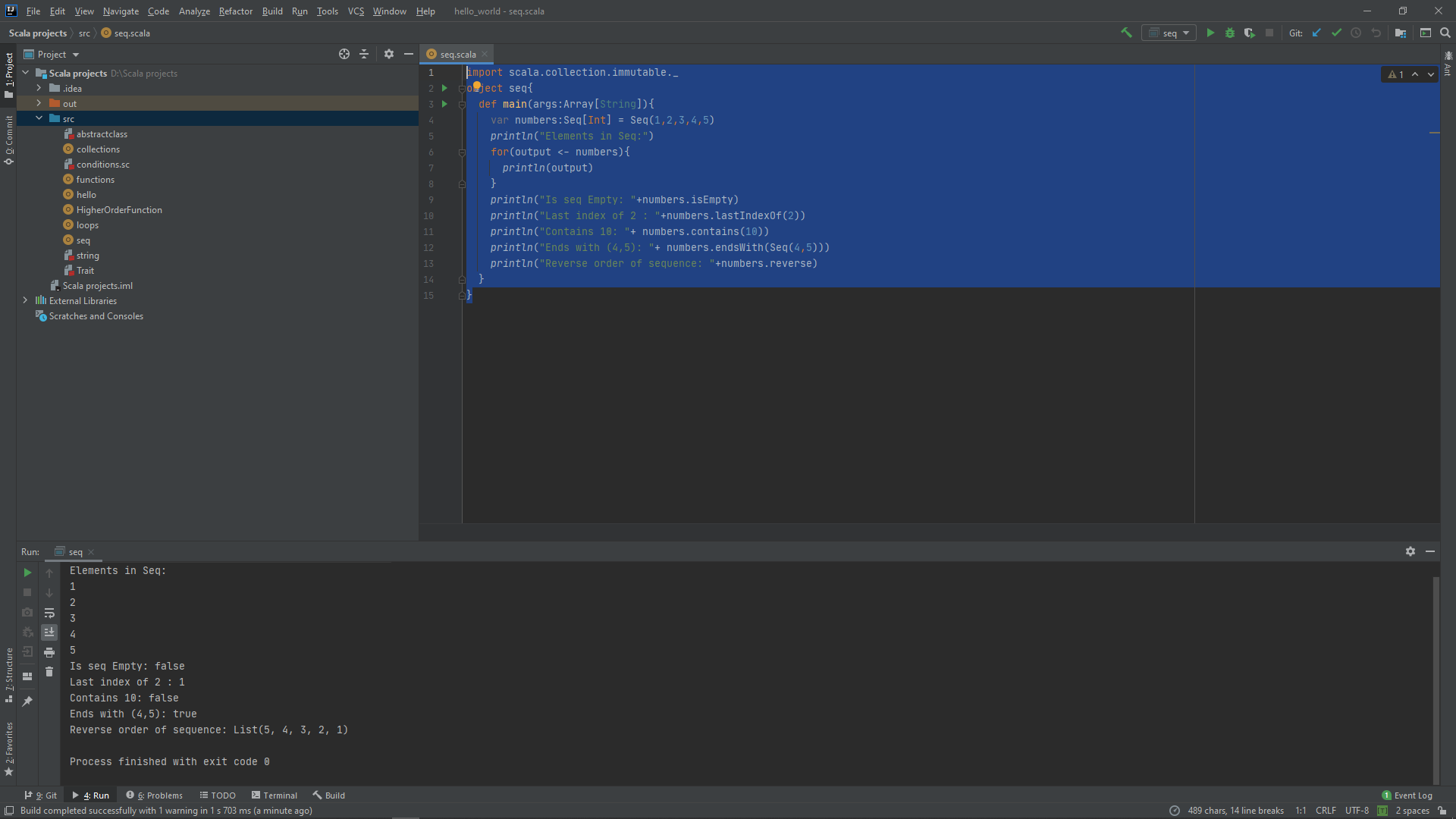
### Commonly used Methods of Seq

|  |  |
| --- | --- |
| Method | Description |
| def contains[A1 >: A](elem: A1): Boolean | Check whether the given element present in this sequence. |
| def copyToArray(xs: Array[A], start: Int, len: Int): Unit | It copies the seq elements to an array. |
| def endsWith[B](that: GenSeq[B]): Boolean | It tests whether this sequence ends with the given sequence or not. |
| def head: A | It selects the first element of this seq collection. |
| def indexOf(elem: A): Int | It finds index of first occurrence of a value in this immutable sequence. |
| def isEmpty: Boolean | It tests whether this sequence is empty or not. |
| def lastIndexOf(elem: A): Int | It finds index of last occurrence of a value in this immutable sequence. |
| def reverse: Seq[A] | It returns new sequence with elements in reversed order. |

**Example**

import scala.collection.immutable.\_  
object seq{  
 def main(args:Array[String]){  
 var numbers:Seq[Int] = Seq(1,2,3,4,5)  
*println*("Elements in Seq:")  
 for(output <- numbers){  
*println*(output)  
 }  
*println*("Is seq Empty: "+numbers.isEmpty)  
*println*("Last index of 2 : "+numbers.lastIndexOf(2))  
*println*("Contains 10: "+ numbers.contains(10))  
*println*("Ends with (4,5): "+ numbers.endsWith(Seq(4,5)))  
*println*("Reverse order of sequence: "+numbers.reverse)  
 }  
}

**Output**



**Scala Vector**

It is a type of immutable data structure and it provides random access of elements. Scala vector can be used for large collection of elements.

We can also add or remove elements from a vector.

**Example**

import scala.collection.immutable.\_  
object vector{  
def main(args:Array[String]){  
 var domain = Vector("Devops","cloud","DataScience")  
 var domain1 = Vector("IOT")  
*println*("Vector Elements: ")

domain.foreach((element:String) =>*println*(element+" "))  
 var newVector = domain :+ "ML" // Adding a new element into vector  
*println*("Vector Elements after adding: ")

newVector.foreach((element:String) =>*println*(element+" "))

var mergeTwoVector = newVector ++ domain1 // Merging two vector  
*println*("Vector Elements after merging: ")

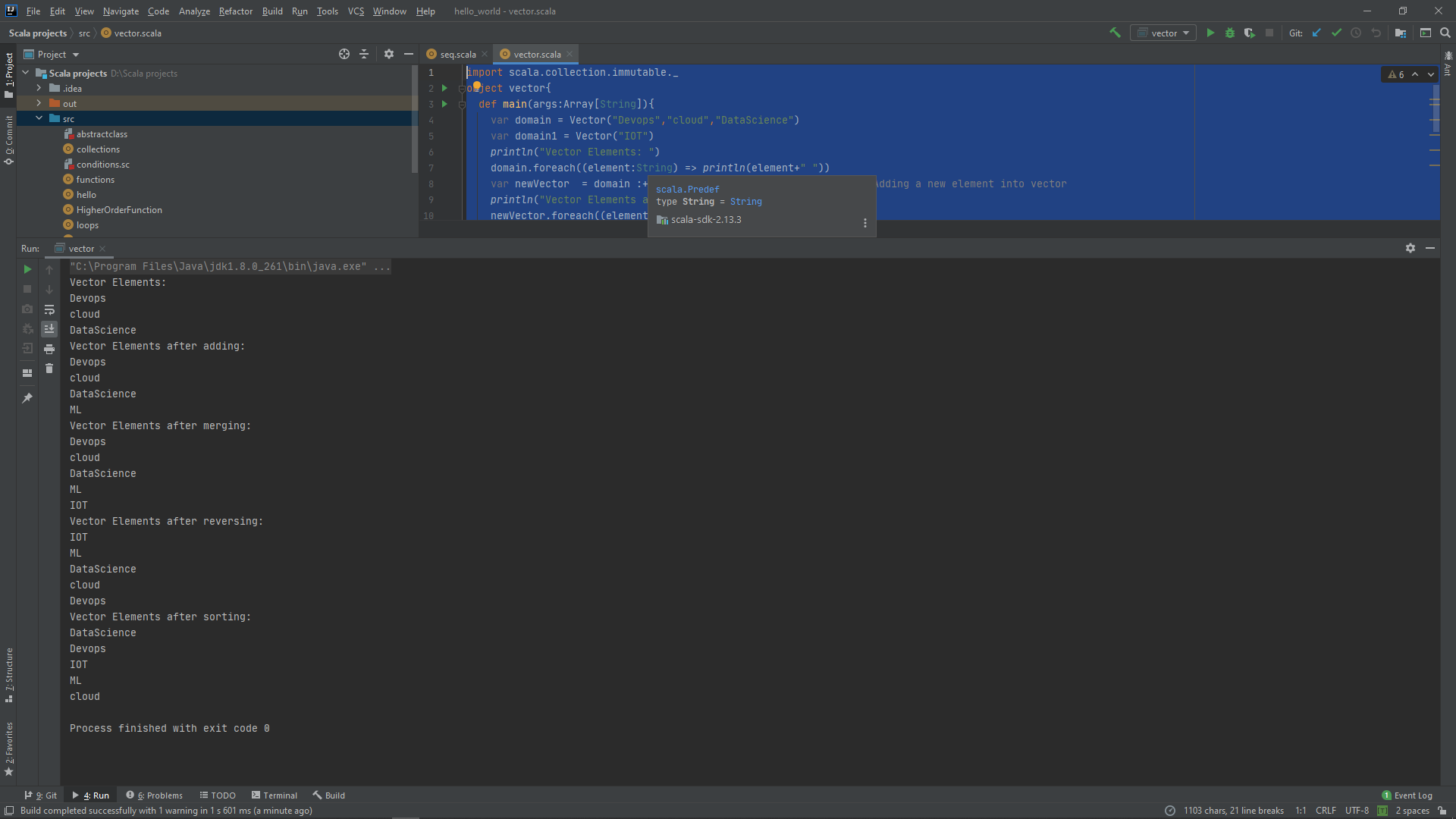
mergeTwoVector.foreach((element:String) =>*println*(element+" "))

var reverse = mergeTwoVector.reverse // Reversing vector elements  
*println*("Vector Elements after reversing: ")

reverse.foreach((element:String) =>*println*(element+" "))

var sortedVector = mergeTwoVector.sorted // Sorting vector elements  
*println*("Vector Elements after sorting: ")  
sortedVector.foreach((element:String) =>*println*(element+" "))  
 }  
}

**Output**



**Scala List**

This follows last-in-first-out (LIFO) pattern to store ordered elements. we can also use predefined function in our list to do various tasks.

**Example**

import scala.collection.immutable.\_  
object list{  
 def main(args:Array[String]){  
 var domain = List("Devops","cloud","DataScience")  
 var domain1 = List("IOT")  
*print*("List Elements: ")  
domain.foreach((element:String) =>*print*(element+" "))

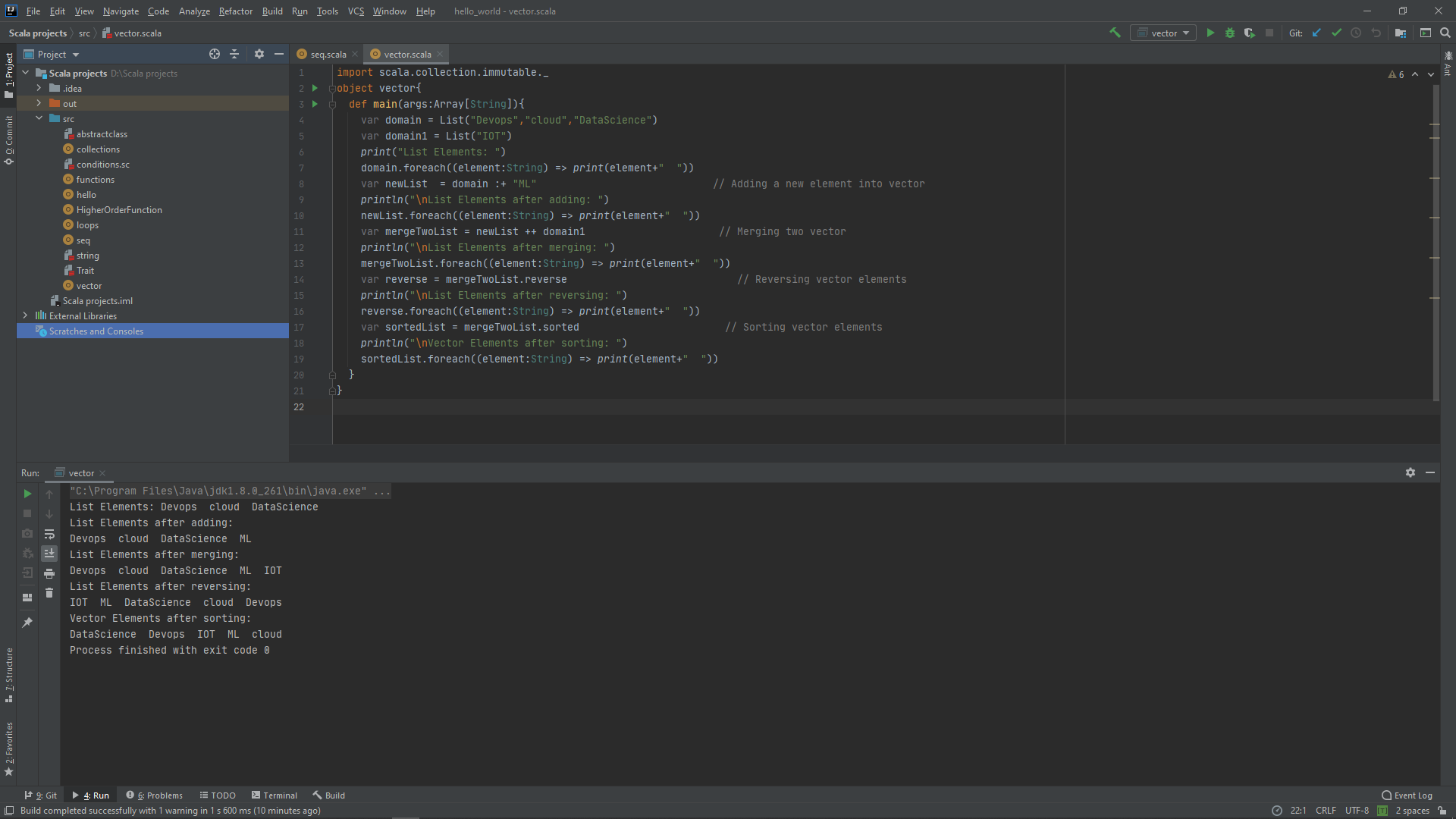
var newList = domain :+ "ML" // Adding a new element into list  
*println*("\nList Elements after adding: ")  
newList.foreach((element:String) =>*print*(element+" "))

var mergeTwoList = newList ++ domain1 // Merging two list  
*println*("\nList Elements after merging: ")  
mergeTwoList.foreach((element:String) =>*print*(element+" "))

var reverse = mergeTwoList.reverse // Reversing list elements  
*println*("\nList Elements after reversing: ")  
reverse.foreach((element:String) =>*print*(element+" "))

var sortedList = mergeTwoList.sorted // Sorting list elements  
*println*("\nVector Elements after sorting: ")  
sortedList.foreach((element:String) =>*print*(element+" "))  
 }  
}

**Output**



# Scala Queue

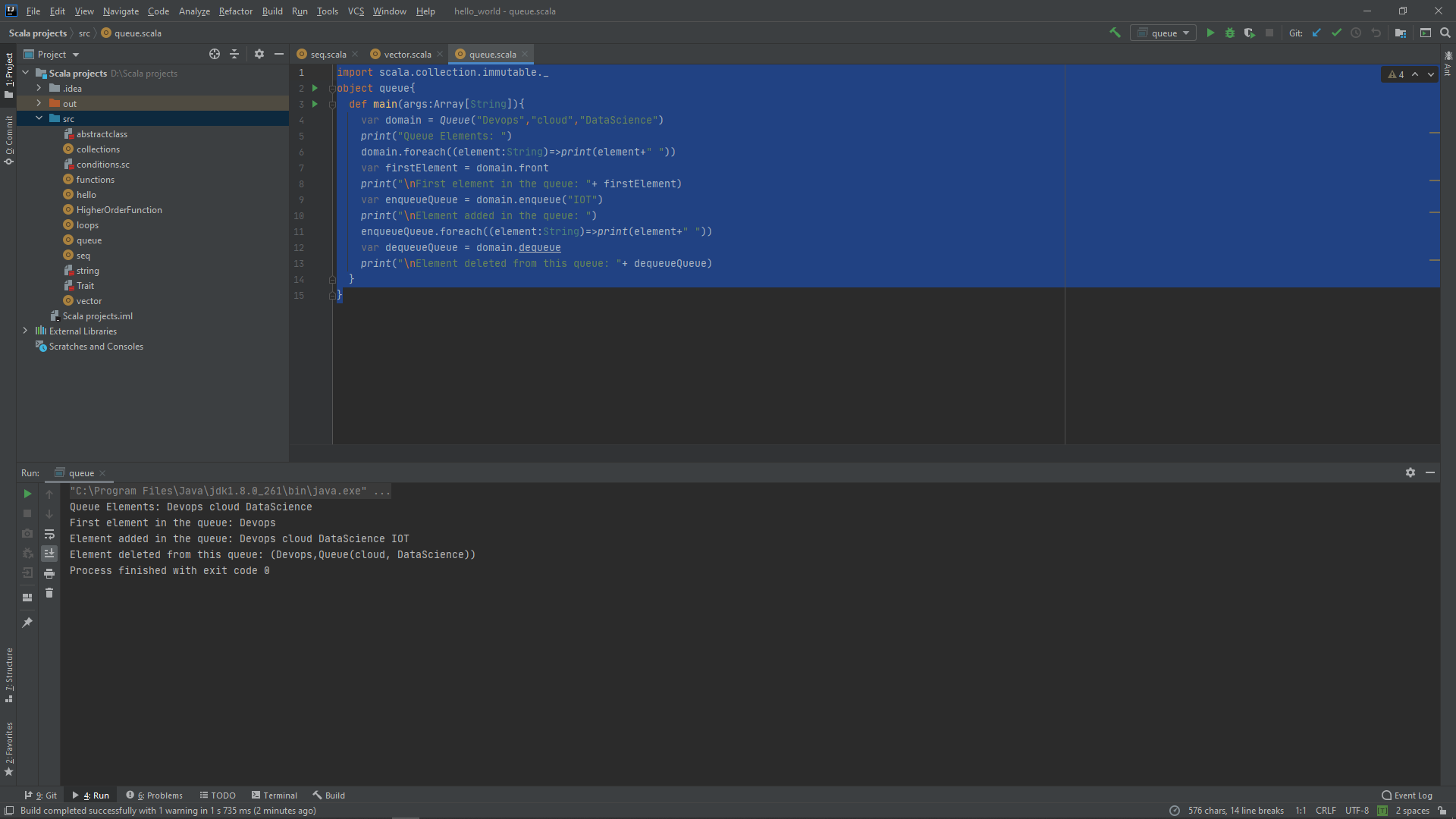
This follows first-in-first-out (FIFO) pattern to implement a data structure for inserting and retrieving elements.

For, implementing a queue we need to have a pair of lists out these one is used to insert elements and another is used to contain deleted elements. Elements are inserted in first list and deleted from second list.

**Example**

import scala.collection.immutable.\_  
object queue{  
 def main(args:Array[String]){  
 var domain = *Queue*("Devops","cloud","DataScience")  
*print*("Queue Elements: ")  
domain.foreach((element:String)=>*print*(element+" "))  
 var firstElement = domain.front  
*print*("\nFirst element in the queue: "+ firstElement)  
 var enqueueQueue = domain.enqueue("IOT")  
*print*("\nElement added in the queue: ")  
enqueueQueue.foreach((element:String)=>*print*(element+" "))  
 var dequeueQueue = domain.dequeue  
*print*("\nElement deleted from this queue: "+ dequeueQueue)  
 }  
}

**Output**



**Scala Maps**

This stores the elements in pairs of key and values we can create maps using two ways

* Comma separated pairs
* Rocket operator

We can understand both using an example

**Example**

object map{  
 def main(args:Array[String]){  
 var map1 = *Map*(("D","Devops"),("S","StatusNeo")) //comma seprated  
 var map2 = *Map*("D"->"Devops","S"->"StatusNeo") //rocket operator  
*println*(map1)  
*println*(map2)  
 var newMap = map1+("C"->"Cloud") // Adding a new element to map  
*println*(newMap)  
 var removeElement = newMap - "D" // Removing an element from map  
*println*(removeElement)  
  
 }  
}

**Output**

