

Strings:

In Scala string is an immutable object, i.e. an object that cannot be modified. On the other hand, objects that can be modified, like arrays, are called mutable objects. Strings are very useful objects and let's see important methods of java.lang.String class.

Creating a Scala String:

The following code can be used to create a String:

```
scala> var name = "Saif"
name: String = Saif

scala> var name: String = "Saif"
name: String = Saif
```

```
object StringEx {
  val greeting: String = "Hello, Saif!"

  def main (args: Array [String]) {
    println (greeting)
  }
}
```

Output:

Hello, Saif!

String Length:

Methods used to obtain information about an object are known as accessor methods. One accessor method that can be used with strings is the length () method, which returns the number of characters contained in the string object.

```
scala> var name: String = "Saif"
name: String = Saif

scala> name.length()
res2: Int = 4
```

Here are more than a couple more examples:

```
scala> "Saif".length()
res4: Int = 4

scala> "".length()
res5: Int = 0

scala> " ".length()
res6: Int = 1
```

```
object StringLength {  
  def main (args: Array [String]) {  
    var name = "My Name is Saif"  
    var len= name.length()  
  
    println ("String Length is: " + len)  
  }  
}
```

Output:

String Length is: 15

Concatenating Strings:

Concatenating two strings in Scala means joining the second to the end of the first. For this, we have the method `concat ()` in the `String` class:

Syntax:

`string1.concat(string2)`

```
scala> "My name is ".concat("Saif")  
res7: String = My name is Saif
```

Strings are more commonly concatenated with the `+` operator:

```
scala> "Hello, " + "Saif" + "!"  
res8: String = Hello, Saif!
```

Let's try using more than one:

```
scala> "Saif".concat(" ").concat("Shaikh")  
res10: String = Saif Shaikh
```

```
object ConcatString {  
  def main (args: Array[String]) {  
    var str1 = "Saif"  
    var str2 = "Shaikh!"  
  
    println ("Hello, " + str1 + " " + str2)  
  }  
}
```

Output:

Hello, Saif Shaikh!

Creating Format Strings:

You have `printf()` and `format()` methods to print output with formatted numbers. The `String` class has an equivalent class method, `format()`, that returns a `String` object rather than a `PrintStream` object.

```
scala> val (a: Int, b: Double, c: Float, d: String) = (10, 1.0, 123.45f, "Saif")
a: Int = 10
b: Double = 1.0
c: Float = 123.45
d: String = Saif
```

```
scala> printf ("a=%d, b=%f, c=%f, d=%s", a, b, c, d)
a=10, b=1.000000, c=123.449997, d=Saif
scala> printf("a=%d, b=%.2f, c=%.2f, d=%s", a, b, c, d)
a=1, b=15.00, c=12.25, d=Saif
```

For Integer data, we used `%d`. For Float and String data, we use `%f` and `%s`, respectively.

```
object FormatString {
  def main (args: Array [String]) {
    var floatVar = 12.456
    var intVar = 2000
    var stringVar = "Hello, Scala!"

    var fs = printf ("Float variable is: " + "%f, \nInteger " + "variable is: %d, \nString
Variable " + "is: %s", floatVar, intVar, stringVar)
    println (fs)
  }
}
```

Output:

```
Float variable is: 12.456000,
Integer variable is: 2000,
String Variable is: Hello, Scala! ()
```

Scala String Comparison Example:

In scala, you can compare two string objects by using `==` (equal) method. The following program describes how to use equal operator. It returns Boolean value either true or false.

```
scala> var s1 = "Scala Programming"
s1: String = Scala Programming
```

```
scala> var s2 = "Saif"
s2: String = Saif
```

```
scala> var s3 = "Saif"
s3: String = Saif
```

```
scala> println (s1 == s2)
false
```

```
scala> println (s2 == s3)
true
```

Scala String Method with Syntax and Method:

List of String Method in Scala with Example

1) char charAt (int index):

This method returns the character at the index we pass to it.

```
scala> "Saif Shaikh".charAt(5)
res33: Char = S
```

2) String concat (String str):

This will concatenate the string in the parameter to the end of the string on which we call it. We saw this when we talked about strings briefly.

```
scala> "Saif".concat("Shaikh")
res40: String = SaifShaikh

scala> "Saif".concat(" ").concat("Shaikh")
res41: String = Saif Shaikh
```

3) Boolean endsWith (String suffix):

This Scala String Method returns true if the string ends with the suffix specified; otherwise, false.

```
scala> "Saif".endsWith("f")
res58: Boolean = true

scala> "Saif".endsWith("i")
res59: Boolean = false
```

4) Boolean equals (Object):

This Scala String Method returns true if the string and the object are equal; otherwise, false.

```
scala> val a = "Saif"
a: String = Saif

scala> "Saif".equals("Saif")
res52: Boolean = true

scala> "Saif".equals(a)
res53: Boolean = true
```

5) Boolean equalsIgnoreCase (String anotherString):

This is like equals (), except that it ignores case differences.

```
scala> "Saif".equalsIgnoreCase("SAIF")
res65: Boolean = true
```

6) int length ():

This Scala String Method simply returns the length of a string.

```
scala> "Saif Shaikh".length()
res54: Int = 11

scala> " ".length()
res55: Int = 1
```

7) String replace (char oldChar, char newChar):

It replaces all occurrences of oldChar with newChar, and return the resultant string.

```
scala> "Saif Shaikh".replace('S','$')
res86: String = $aif $haikh

scala> "Saif Shaikh".replace('s','$')
res87: String = Saif Shaikh

scala> "Saif Shaikh".toLowerCase().replace('s','$')
res88: String = $aif $haikh
```

8) String replaceAll (String regex, String replacement):

This will replace each substring matching the regular expression. It will replace it with the replacement string we provide.

```
scala> "potdotnothotokayslot".replaceAll(".o.", "*")
res90: String = ****okays*

scala> "potdotnothotokayslot".replaceAll(".ot", "*")
res91: String = ****okays*
```

```
scala> "potdotSaif".replaceAll("Sa*", "\\$")
res93: String = potdot$if
```

9) Boolean startsWith (String prefix):

If the string starts with the prefix we specify, this returns true; otherwise, false.

```
scala> "Saif Shaikh".startsWith("S")
res99: Boolean = true

scala> "Saif Shaikh".startsWith("Sa")
res100: Boolean = true

scala> "Saif Shaikh".startsWith("a")
res101: Boolean = false
```

10) Boolean startsWith (String prefix, int toffset):

If the string starts with the specified prefix at the index we specify, this string method returns true; otherwise, false.

```
scala> "Saif Shaikh".startsWith("a", 1)
res102: Boolean = true

scala> "Saif Shaikh".startsWith("S", 5)
res103: Boolean = true

scala> "Saif Shaikh".startsWith("S", 1)
res104: Boolean = false
```

11) String substring (int beginIndex):

This Scala String Method returns the contents of the string beginning at beginIndex.

```
scala> "Saif Shaikh".substring(5)
res101: String = Shaikh
```

12) String substring (int beginIndex, int endIndex):

This returns the part of the string beginning at beginIndex and ending at endIndex. Wait, isn't that the same as subsequence? Check the types of the results:

```
scala> "Saif Shaikh".substring(0, 4)
res109: String = Saif

scala> "Saif Shaikh".subSequence(0, 4)
res110: CharSequence = Saif
```

While one returns a String, the other returns a CharSequence.

13) String toLowerCase ():

This String Method in Scala converts all characters in the string to lower case, and then returns the resultant string.

```
scala> "Saif Shaikh".toLowerCase()
res112: String = saif shaikh
```

14) String toUpperCase ():

This Scala String Method is like toLowerCase, and converts all characters in the string to upper case.

```
scala> "Saif Shaikh".toUpperCase()
res116: String = SAIF SHAIKH
```

15) String trim ():

trim will elide the leading and trailing whitespaces from the string, and then return it.

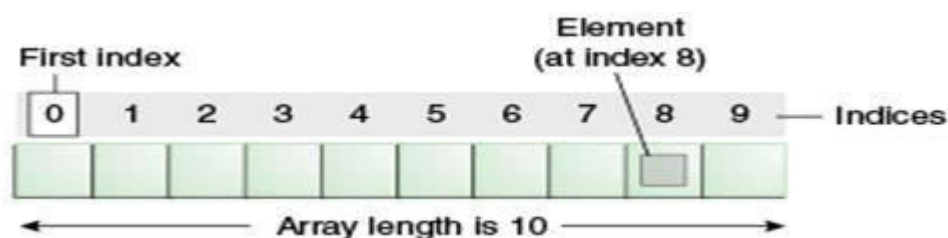
```
scala> " Saif Shaikh ".trim()
res117: String = Saif Shaikh
```

Arrays:

Scala provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Array is a collection of mutable values. It is an index based data structure which starts from 0 index to n-1 where n is length of array.

Following image represents the structure of array where first index is 0, last index is 9 and array length is 10.



Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array and you must specify the type of array the variable can reference.

Syntax:

```
var z: Array[String] = new Array[String](3)
```

```
var z = new Array[String](3)
```

```
var z = Array("Saif", "Arif", "Ram")
```

```
scala> val z = Array("Saif", "Arif", "Ram")
z: Array[String] = Array(Saif, Arif, Ram)
```

```
var a = Array(1,2,3)
```

```
scala> var z = Array(1, 2, 3)
z: Array[Int] = Array(1, 2, 3)
```

Processing an Array:

When processing array elements, we often use loop control structures because all of the elements in an array are of the same type and the size of the array is known.

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

```
scala> for (i <- a) {println (i)}
1
2
3
```

To calculate the sum of all the elements:

```
scala> var sum = 0
sum: Int = 0

scala> for (i <- a) {sum = sum + i}

scala> sum
res2: Int = 6
```

Finding the highest value from the array:

```
scala> var max = a(0)
max: Int = 1

scala> for (i <- a) {if (i > max) {max = i} }

scala> max
res5: Int = 3
```

```
package scala_practice
```

```
object Arrays {
  var myList = Array (1, 2, 3)
  var total = 0
  var max = myList (0)
  def main (args: Array [String]) {
    for (i <- myList) {
      println (i)
    }
    for (i <- 0 to (myList.length - 1)) {
      total += myList (i)
      println ("Total is: " + total)
    }
    for (i <- 1 to (myList.length - 1)) {
      if (myList(i) > max) max = myList(i)
      println ("Max is " + max)
    }
  }
}
```

Output:

```
1
2
3
Total is: 1
Total is: 3
Total is: 6
Max is 3
```


Concatenating Arrays:

We can append a Scala array to another using the `concat()` method. This takes the arrays as parameters- in order.

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

scala> val b = Array (4, 5, 6)
b: Array[Int] = Array(4, 5, 6)
```

We'll need to import the `Array._` package

```
scala> import Array._
import Array._
```

Now, let's call `concat()`

```
scala> var c = concat (a, b)
c: Array[Int] = Array(1, 2, 3, 4, 5, 6)
```

Creating an Array with Range:

Use of `range()` method to generate an array containing a sequence of increasing integers in a given range. You can use final argument as step to create the sequence, if you do not use final argument, then step would be assumed as 1.

Let us take an example of creating an array of range (10, 20, 2). It means creating an array with elements between 10 and 20 and range difference 2. Elements in the array are 10, 12, 14, 16, and 18.

Another example: `range (10, 20)`. Here range difference is not given so by default it assumes 1 element. It creates an array with the elements in between 10 and 20 with range difference 1. Elements in the array are 10, 11, 12, 13, ..., and 19.

We can use this to create an array to iterate on.

```
scala> val a = range (1, 5)
a: Array[Int] = Array(1, 2, 3, 4)

scala> val a = range (1, 10, 2)
a: Array[Int] = Array(1, 3, 5, 7, 9)

scala> val a = range (2, 10, 2)
a: Array[Int] = Array(2, 4, 6, 8)
```

```
package scala_practice
import Array._

object Array_Ranges {
  def main (args: Array[String]) {
    var myList1 = range (10, 20, 2)
    var myList2 = range (10, 20)
    // Print all the array elements
    for (x <- myList1) {
      print (" " + x)
    }
    println ()
    for (x <- myList2) {
      print (" " + x)
    }
  }
}
```

Output:

```
10 12 14 16 18
10 11 12 13 14 15 16 17 18 19
```

Collections:

Scala has a rich set of collection library. Collections are containers of things. Those containers can be sequenced, linear sets of items like List, Tuple, Option, Map, etc. The collections may have an arbitrary number of elements or be bounded to zero or one element (e.g., Option).

Collections may be strict or lazy. Lazy collections have elements that may not consume memory until they are accessed, like Ranges. Additionally, collections may be mutable (the contents of the reference can change) or immutable (the thing that a reference refers to is never changed). Note that immutable collections may contain mutable items.

For some problems, mutable collections work better, and for others, immutable collections work better. When in doubt, it is better to start with an immutable collection and change it later if you need mutable ones.

- 1) List
- 2) Set
- 3) Map
- 4) Tuple

List:

- The Scala List is an immutable sequence of elements, implemented as a linked list & each element need not be of the same data type and can't be changed by assignment.
- List can contain duplicate elements & it maintains the order of elements.
- List represent a linked list whereas arrays are flat.

Declaring a Scala List:

While declaring a Scala list, we can also denote the type of elements it will hold. If it will hold more than one kind, we use 'Any'. Here are few examples.

List of Strings:

```
scala> val StringList: List[String] = List ("Apple", "Banana", "Oranges")  
StringList: List[String] = List(Apple, Banana, Oranges)
```

List of Integers:

```
scala> val NumList: List[Int] = List (1, 2, 3, 4, 5)  
NumList: List[Int] = List(1, 2, 3, 4, 5)
```

List of Chars:

```
scala> val CharList: List[Char]= List('a','b','c')  
CharList: List[Char] = List(a, b, c)
```

List of Nothing:

```
scala> val EmptyList: List[Nothing] = List()  
EmptyList: List[Nothing] = List()
```

List inside List (Nested List):

```
scala> val NestedList: List[List[Int]] = List(List(1,2,3),List(4,5,6),List(7,8,9))  
NestedList: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))
```

List using range:

```
scala> val x = List.range(1,10)  
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)  
  
scala> val x = List.range(1,10,2)  
x: List[Int] = List(1, 3, 5, 7, 9)
```

Defining Lists using :: and Nil

All lists can be defined using two fundamental building blocks, a tail Nil and :: which is pronounced cons. Nil also represents the empty list.

List of Strings:

```
scala> val StringList = "Saif" :: ("Arif" :: ("Ram" :: Nil))
StringList: List[String] = List(Saif, Arif, Ram)

scala> val StringList = "Saif"::"Arif"::"Ram"::Nil
StringList: List[String] = List(Saif, Arif, Ram)
```

List of Integers:

```
scala> val NumList = 1 :: (2 :: (3 :: (4 :: (5 :: Nil) ) ) )
NumList: List[Int] = List(1, 2, 3, 4, 5)

scala> val NumList = 1::2::3::4::5::Nil
NumList: List[Int] = List(1, 2, 3, 4, 5)
```

List of Chars:

```
scala> val CharList = 'a'::("b" :: ("c" :: Nil ) )
CharList: List[Any] = List(a, b, c)

scala> val CharList = "a"::"b"::"c"::Nil
CharList: List[String] = List(a, b, c)
```

List inside List (Nested List):

```
scala> val NestedList = (1::(2::(3::Nil)))::(4::(5::(6::Nil)))::Nil
NestedList: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6))

scala> val NestedList = (1::(2::(3::Nil)))::("Arif"::("Saif"::("Ram"::Nil)))::Nil
NestedList: List[List[Any]] = List(List(1, 2, 3), List(Arif, Saif, Ram))

scala> val NestedList = (1::2::3::Nil)::("Arif"::"Saif"::"Ram"::Nil)::Nil
NestedList: List[List[Any]] = List(List(1, 2, 3), List(Arif, Saif, Ram))
```

Operations on Lists:

We can carry out three operations on a Scala list to get a result:

a) head:

This returns the first element of a list.

```
scala> val NumList = 1::2::3::4::5::Nil
NumList: List[Int] = List(1, 2, 3, 4, 5)

scala> NumList.head
res3: Int = 1
```

b) tail:

This returns all elements of a list except first.

```
scala> val NumList = 1::2::3::4::5::Nil
NumList: List[Int] = List(1, 2, 3, 4, 5)

scala> NumList.tail
res4: List[Int] = List(2, 3, 4, 5)
```

c) isEmpty:

If the list is empty, this returns a Boolean true; otherwise false.

```
scala> val NothingList = Nil
NothingList: scala.collection.immutable.Nil.type = List()
```

```
scala> NothingList.isEmpty
res5: Boolean = true
```

```
scala> NumList.isEmpty
res6: Boolean = false
```

Concatenating Lists:

You can use either **:::** operator or **List.:::()** method or **List.concat()** method to add two or more lists.

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

scala> val b = List(4, 5, 6)
b: List[Int] = List(4, 5, 6)
```

a) The :: Operator:

```
scala> a ::: b
res7: List[Int] = List(1, 2, 3, 4, 5, 6)
```

b) The List.:::() Method:

We can call the :::() method on the first list.

```
scala> a.:::(b)
res9: List[Int] = List(4, 5, 6, 1, 2, 3)

scala> b.:::(a)
res10: List[Int] = List(1, 2, 3, 4, 5, 6)
```

c) The List.concat () Method:

We can also call the concat () method on List in Scala Collections. We pass both the lists as arguments.

```
scala> List.concat(a, b)
res11: List[Int] = List(1, 2, 3, 4, 5, 6)
```

Creating Uniform Lists:

The method List.fill () creates a list and fills it with zero or more copies of an element.

```
scala> val a = List.fill (3) ("Saif")
a: List[String] = List(Saif, Saif, Saif)

scala> val a = List.fill (3) (5)
a: List[Int] = List(5, 5, 5)
```

Reversing a List:

This reverses the order of elements in a list using the List.reverse method.

```
scala> val ReverseList = List(1, 2, 3, 4, 5)
ReverseList: List[Int] = List(1, 2, 3, 4, 5)

scala> ReverseList.reverse
res1: List[Int] = List(5, 4, 3, 2, 1)
```

Prepend to List:

You can prepend items to a Scala List using :: method:

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

scala> val y = 99 :: x
y: List[Int] = List(99, 1, 2, 3)
```

Scala Lists and the 'for' expression:

The Scala *for expression* is not specific to lists, but is an extremely powerful way to operate on lists. Here's a simple example of how to iterate over a list using for expression:

```
scala> val names = List("Bob", "Fred", "Joe", "Julia", "Kim")
names: List[String] = List(Bob, Fred, Joe, Julia, Kim)

scala> for (name <- names) println(name)
Bob
Fred
Joe
Julia
Kim
```

Now let's add a simple "if" clause with for expression to print only the elements we want to print:

```
scala> val names = List("Bob", "Fred", "Joe", "Julia", "Kim")
names: List[String] = List(Bob, Fred, Joe, Julia, Kim)

scala> for (name <- names if name.startsWith("J")) println(name)
Joe
Julia
```

Filtering Scala lists:

```
scala> val x = List(1,2,3,4,5,6,7,8,9,10)
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val evens = x.filter(a => a % 2 == 0)
evens: List[Int] = List(2, 4, 6, 8, 10)
```

Access the elements of a list:

```
scala> val names = List("Fred", "Joe", "Bob")
names: List[String] = List(Fred, Joe, Bob)

scala> names(0)
res22: String = Fred

scala> names(2)
res23: String = Bob
```

Append elements in front & at the end of list:

```
scala> val names = List("Fred", "Joe", "Bob")
names: List[String] = List(Fred, Joe, Bob)

scala> val a = names :+ "Saif"
a: List[String] = List(Fred, Joe, Bob, Saif)

scala> val a = "Saif" +: names
a: List[String] = List(Saif, Fred, Joe, Bob)
```

Empty List:

```
scala> val a = List.empty
a: List[Nothing] = List()
```

Methods on a List in Scala:

We can call the following methods on a Scala List.
(Note that these don't modify the Lists)

1) def :::

This joins the List in the argument to the other List.

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

scala> a ::: (List(4, 5, 6))
res15: List[Int] = List(4, 5, 6, 1, 2, 3)

scala> val b = a ::: (List(4, 5, 6))
b: List[Int] = List(4, 5, 6, 1, 2, 3)

scala> b.sorted
res16: List[Int] = List(1, 2, 3, 4, 5, 6)
```

2) def ::

This adds an element to a List's beginning.

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

scala> a :: (2)
res31: List[Int] = List(2, 1, 2, 3)

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


3) def contains (elem: Any): Boolean

If the list contains a certain element, this returns true; otherwise, false.

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

```
scala> a.contains(2)
res33: Boolean = true
```

```
scala> a.contains(5)
res34: Boolean = false
```

4) def distinct

Let's take a new Scala List for this.

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

```
scala> a.distinct
res53: List[Int] = List(1, 2, 3)
```

5) def drop (n: Int)

This returns all elements except the first n.

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

```
scala> a.drop(3)
res54: List[Int] = List(1, 2, 3)
```

6) def dropRight (n: Int)

This returns all elements except the last n.

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

```
scala> a.dropRight(3)
res55: List[Int] = List(1, 2, 2)
```

7) def equals :Boolean

This compares two sequences.

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

```
scala> val b = List(4, 5, 6)
b: List[Int] = List(4, 5, 6)
```

```
scala> a.equals(b)
res57: Boolean = false
```

```
scala> a.equals(List(1, 2, 3))
res58: Boolean = true
```

8) def length: Int

This returns a List's length.

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

scala> a.length
res86: Int = 3
```

9) def map

This applies the function to every element in the list.

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

scala> a.map(x => x*x)
res87: List[Int] = List(1, 4, 9)
```

```
scala> val names = List("Fred", "Joe", "Bob")
names: List[String] = List(Fred, Joe, Bob)

scala> val lower = names.map(_.toLowerCase)
lower: List[String] = List(fred, joe, bob)

scala> val upper = names.map(_.toUpperCase)
upper: List[String] = List(FRED, JOE, BOB)
```

10) def max: A

This returns the highest element.

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

scala> a.max
res88: Int = 3
```

11) def min: A

This returns the lowest element.

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

scala> a.min
res89: Int = 1
```

12) def mkString: String

This displays all elements of a list in a String.

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

scala> a.mkString
res90: String = 123
```

13) def mkString(sep: String): String

This lets us define a separator for mkString.

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

scala> a.mkString(",")
res91: String = 1,2,3
```

14) def reverse:

This reverses a List of Scala Collections.

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

scala> a.reverse
res92: List[Int] = List(3, 2, 1)
```

15) def sum: A

This returns the sum of all elements.

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

scala> a.sum
res94: Int = 6
```

16) def take (n: Int)

This returns the first n elements.

```
scala> val a = List(1, 1, 2, 3, 3, 5)
a: List[Int] = List(1, 1, 2, 3, 3, 5)

scala> a.take(2)
res95: List[Int] = List(1, 1)
```

17) def takeRight (n: Int)

This returns the last n elements.

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

scala> a.takeRight(3)
res97: List[Int] = List(3, 3, 3)
```

Sets:

Scala Set is a collection of pairwise different elements of the same type. In other words, a Set is a collection that contains no duplicate elements. There are two kinds of Sets, the immutable and the mutable.

By default, Scala uses the immutable Set. If you want to use the mutable Set, you'll have to import **scala.collection.mutable.Set** class explicitly. If you want to use both mutable and immutable sets in the same collection, then you can continue to refer to the immutable Set as Set but you can refer to the mutable Set as mutable.Set

Syntax:

```
val variableName: Set[Type] = Set(element1, element2, ... elementN)
```

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

1) Declaring an Empty Set:

We must include the type annotation when declaring an empty set, so it can be decided what concrete type to assign to a variable.

```
scala> val s: Set[Int] = Set()
s: Set[Int] = Set()
```

2) Declaring a Set with Values:

We may choose to elide the type annotation. When we do so, Scala will infer that from the type of values inside the set.

```
scala> val s = Set(1, 2, 2, 3)
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

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

Scala Set Example:

In this example, we have created a set. You can create an empty set also. Let's see how to create a set.

```
import scala.collection.immutable._
```

```
object MainObject {
```

```
  def main (args: Array[String]) {
```

```
    val set1 = Set () // An empty set
```

```
    val games = Set ("Cricket", "Football", "Hockey", "Golf") // Creating a set with elements
```

```
    println (set1)
```

```
    println (games)
```

```
  }
```

```
}
```

Operations on Scala Sets:

1) head:

This will return the first element of a Scala set.

```
scala> val s = Set{1, 2, 2, 3}
s: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

```
scala> s.head
res2: Int = 1
```

2) tail:

This will return all elements of a set except the first.

```
scala> s.tail
res3: scala.collection.immutable.Set[Int] = Set(2, 3)
```

3) isEmpty:

If the set is empty, this will return a Boolean true; otherwise, false.

```
scala> s.isEmpty
res4: Boolean = false
```

Scala Set Example 2:

In Scala, Set provides some predefined properties to get information about set. You can get first or last element of Set and many more. Let's see an example.

```
import scala.collection.immutable._
```

```
object MainObject {
```

```
  def main (args: Array[String]) {
```

```
    val games = Set("Cricket", "Football", "Hockey", "Golf")
```

```
    println (games.head)           // Returns first element present in the set
```

```
    println (games.tail)           // Returns all elements except first element.
```

```
    println (games.isEmpty)        // Returns either true or false
```

```
  }
```

```
}
```

Output:

Cricket

Set (Football, Hockey, Golf)

false

Concatenating Scala Sets:

You can use either ++ operator or Set.++() method to concatenate two or more sets, but while adding sets it will remove duplicate elements.

```
scala> val s1 = Set(1, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> val s2 = Set(4, 5, 6)
s2: scala.collection.immutable.Set[Int] = Set(4, 5, 6)
```

1) The ++ Operator:

```
scala> s1 ++ s2
res13: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 3, 4)
```

2) The Set.++() Method:

```
scala> s1.++(s2)
res14: scala.collection.immutable.Set[Int] = Set(5, 1, 6, 2, 3, 4)
```

This example also proves that the set does not maintain order to store elements.

Max and Min in a Set:

We can find the maximum and minimum values in a set.

```
scala> val s = Set(1, 2, 3, 4, 5)
s: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

1) max:

This returns the maximum value from a set.

```
scala> s.max
res17: Int = 5
```

2) min:

This returns the minimum value from a set.

```
scala> s.min
res18: Int = 1
```

Finding Values Common to Two Sets:

You can use either Set.&() method or Set.intersect() method to find out the common values between two sets.

```
scala> val s1 = Set(1, 2, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> val s2 = Set(4, 2, 5, 3)
s2: scala.collection.immutable.Set[Int] = Set(4, 2, 5, 3)
```

1) The Set.&() Method:

We can call &() on one set, and pass another to it.

```
scala> s1.&{s2}
res19: scala.collection.immutable.Set[Int] = Set(2, 3)
```

2) Set.intersect() Method:

We can use the intersect () method.

```
scala> s1.intersect(s2)
res20: scala.collection.immutable.Set[Int] = Set(2, 3)
```

Methods to Call on a Set:

Playing around with sets in Scala, you can call these methods on them.

(Note that they do not modify the set itself)

1) def +

This adds an element to the set and returns it. (But doesn't modify the original set)

```
scala> s1
res17: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s1.+(5)
res18: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 5)

scala> s1
res19: scala.collection.immutable.Set[Int] = Set(1, 2, 3)
```

2) def -

This removes the element from the set and then returns it.

```
scala> val s1 = Set(1, 2, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s1.-(2)
res24: scala.collection.immutable.Set[Int] = Set(1, 3)
```

3) def contains (elem: A): Boolean

If the set contains that element, this returns true; otherwise, false.

```
scala> val s1 = Set(1, 2, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s1.contains(2)
res25: Boolean = true

scala> s1.contains(5)
res26: Boolean = false
```

4)def &

This returns an intersection of two sets, as we just saw.

```
scala> val s1 = Set(1, 2, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> val s2 = Set(1, 5, 3)
s2: scala.collection.immutable.Set[Int] = Set(1, 5, 3)

scala> s1.&(s2)
res31: scala.collection.immutable.Set[Int] = Set(1, 3)

scala> s1&s2
res32: scala.collection.immutable.Set[Int] = Set(1, 3)
```

5)def +

This adds multiple elements to a Scala set and returns it.

```
scala> val s1 = Set(1, 2, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s1.+(4, 5)
res35: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

6)def ++

This concatenates a set with another collection.

```
scala> val s1 = Set(1, 2, 2, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> val s2 = Set(1, 5, 3)
s2: scala.collection.immutable.Set[Int] = Set(1, 5, 3)

scala> s1.++(s2)
res41: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 5)

scala> s1.++(Set(1, 3, 5))
res42: scala.collection.immutable.Set[Int] = Set(1, 2, 3, 5)
```

7)def count

This returns the count of elements that satisfy the predicate.

```
scala> val s1 = Set(2, 4, 3)
s1: scala.collection.immutable.Set[Int] = Set(2, 4, 3)

scala> s1.count(x => {x%2 !=0 })
res57: Int = 1

scala> s1.count(x => {x%2 == 0 })
res58: Int = 2
```


8) def diff

This returns the set difference (elements existing in one set, but not in another)

```
scala> val s1 = Set(1, 5, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 5, 3)

scala> val s2 = Set(2, 5, 6)
s2: scala.collection.immutable.Set[Int] = Set(2, 5, 6)

scala> s1.diff(s2)
res66: scala.collection.immutable.Set[Int] = Set(1, 3)

scala> s2.diff(s1)
res67: scala.collection.immutable.Set[Int] = Set(2, 6)
```

9) def drop

This returns all elements except the first n.

```
scala> val s1 = Set(1, 5, 6, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 5, 6, 3)

scala> s1.drop(2)
res69: scala.collection.immutable.Set[Int] = Set(6, 3)
```

10) def dropRight

This Scala set returns all elements except the last n.

```
scala> val s1 = Set(1, 5, 6, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 5, 6, 3)

scala> s1.dropRight(2)
res70: scala.collection.immutable.Set[Int] = Set(1, 5)
```

11) def equals : Boolean

This set in Scala compares the set to another sequence.

```
scala> val s1 = Set(1, 5, 6, 3)
s1: scala.collection.immutable.Set[Int] = Set(1, 5, 6, 3)

scala> val s2 = Set(1, 2, 3)
s2: scala.collection.immutable.Set[Int] = Set(1, 2, 3)

scala> s1.equals(s2)
res75: Boolean = false

scala> s2.equals(Set(1, 2, 3))
res76: Boolean = true
```

12) def head

This Scala Set returns the first element from the set.

```
scala> val s1 = Set(3, 7, 21)
s1: scala.collection.immutable.Set[Int] = Set(3, 7, 21)

scala> s1.head
res96: Int = 3
```

13) def isEmpty: Boolean

This set in Scala returns true if the set is empty; otherwise, false.

```
scala> s1.isEmpty
res100: Boolean = false
```

14) def intersect

This returns the intersection of two sets (elements common to both).

```
scala> val s1 = Set(3, 7, 21)
s1: scala.collection.immutable.Set[Int] = Set(3, 7, 21)

scala> val s2 = Set(4, 7, 25)
s2: scala.collection.immutable.Set[Int] = Set(4, 7, 25)

scala> s1.intersect(s2)
res99: scala.collection.immutable.Set[Int] = Set(7)
```

15) def last

This returns the last element from a set.

```
scala> val s1 = Set(1, 3, 3, 5)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5)

scala> s1.last
res10: Int = 5
```

16) def max: A

This returns the highest value from the set.

```
scala> val s1 = Set(1, 3, 3, 5)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5)

scala> s1.max
res12: Int = 5
```

17) def min: A

This returns the lowest element.

```
scala> val s1 = Set(1, 3, 3, 5)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5)

scala> s1.min
res13: Int = 1
```

18) def mkString: String

This Scala set represents all elements of the set as a String.

```
scala> val s1 = Set(1, 3, 3, 5)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5)

scala> s1.mkString
res14: String = 135
```

19) def mkString (sep: String): String

This lets us define a separator for the above method's functionality.

```
scala> val s1 = Set(1, 3, 3, 5)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5)

scala> s1.mkString(",")
res15: String = 1,3,5
```

20) def splitAt

This splits the set at the given index and returns the two resulting subsets.

```
scala> val s1 = Set(1, 3, 3, 5, 5, 7)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5, 7)

scala> s1.splitAt(2)
res19: (scala.collection.immutable.Set[Int], scala.collection.immutable.Set[Int]) = (Set(1, 3), Set(5, 7))
```

21) def take

This Scala set returns the first n elements from the set.

```
scala> val s1 = Set(1, 3, 3, 5, 5, 7)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5, 7)

scala> s1.take(2)
res28: scala.collection.immutable.Set[Int] = Set(1, 3)
```

22) def takeRight

This returns the last n elements.

```
scala> val s1 = Set(1, 3, 3, 5, 5, 7)
s1: scala.collection.immutable.Set[Int] = Set(1, 3, 5, 7)

scala> s1.takeRight(1)
res29: scala.collection.immutable.Set[Int] = Set(7)
```

Maps:

Map stores elements in pairs of key and values. In scala, you can create map by using two ways either by using comma separated pairs or by using rocket operator.

A Map in Scala is a collection of key-value pairs, and is also called a hash table. We can use a key to access a value. These keys are unique; however, the values may be common. The default Scala Map is immutable. To use a mutable Map, we use the **scala.collection.mutable.Map** class. To use both in the same place, refer to the immutable Map as Map, and to the mutable Map as mutable.Map.

Syntax:

```
var m1: Map[String, Int] = Map(("Key", "Value"), ("Key", "Value"))
var m2 = Map("Key" -> "Value", "Key" -> "Value")
```

```
scala> val m1: Map[Int, String] = Map((101, "Saif"), (102, "Arif"))
m1: Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> val m2 = Map((101 -> "Saif"), (102 -> "Arif"))
m2: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)
```

a) Declaring an Empty Scala Map

```
scala> val m: Map[String, Int] = Map()
m: Map[String,Int] = Map()
```

While defining empty map, the type annotation is necessary as the system needs to assign a concrete type to variable.

b) Declaring a Map with Values

When we provide values, Scala will use those to infer the type of variables. So, we don't need to include the type annotations.

```
scala> val m1 = Map((101, "Saif"), (102, "Arif"))
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> val m2 = Map((101 -> "Saif"), (102 -> "Arif"))
m2: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)
```

Now, to add a key-value pair to this, we do the following:

```
scala> var m1 = Map((101 -> "Saif"), (102 -> "Arif"))
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> m1 += (103 -> "Ram")

scala> m1
res7: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif, 103 -> Ram)
```

Scala maps Example:

In the following example, we have both approaches to create map.

```
package scala_practice
```

```
object Map_Ex {  
  val m1: Map [String, Int] = Map (("Saif", 101), ("Arif", 102))  
  val m2: Map [String, Int] = Map ("Saif" -> 101, "Arif" -> 102)  
  val m3 = Map (("Saif", 101), ("Arif", 102))  
  val m4 = Map ("Saif" -> 101, "Arif" -> 102)  
  val m5 = Map ()  
  def main (args: Array [String]) {  
    println (m1); println (m2); println (m3); println (m4); println (m5)  
  }  
}
```

Operations on a Map in Scala:

These are the basic operations we can carry out on a Map:

a) keys:

This returns an iterable with each key in the Map.

```
scala> val m1 = Map({101 -> "Saif"}, {102 -> "Arif"})  
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)  
  
scala> m1.keys  
res10: Iterable[Int] = Set(101, 102)
```

b) values:

This returns an iterable with each value in the Scala Map.

```
scala> val m1 = Map({101 -> "Saif"}, {102 -> "Arif"})  
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)  
  
scala> m1.values  
res11: Iterable[String] = MapLike(Saif, Arif)
```

c) isEmpty:

If the Map is empty, this returns true; otherwise, false.

```
scala> val m1 = Map({101 -> "Saif"}, {102 -> "Arif"})  
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)  
  
scala> m1.isEmpty  
res12: Boolean = false
```

Concatenating Maps in Scala:

You can use either ++ operator or Map.++() method to concatenate two or more Maps, but while adding Maps it will remove duplicate keys.

The ++ Operator:

```
scala> val m1 = Map((101 -> "Saif"), (102 -> "Arif"), (101, "Saif"))
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> val m2 = Map(103 -> "Ram", 104 -> "Tausif", 104 -> "Tausif")
m2: scala.collection.immutable.Map[Int,String] = Map(103 -> Ram, 104 -> Tausif)

scala> m1 ++ m2
res14: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif, 103 -> Ram, 104 -> Tausif)
```

Map.++() method:

```
scala> val m1 = Map((101 -> "Saif"), (102 -> "Arif"), (101, "Saif"))
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> val m2 = Map(103 -> "Ram", 104 -> "Tausif", 104 -> "Tausif")
m2: scala.collection.immutable.Map[Int,String] = Map(103 -> Ram, 104 -> Tausif)

scala> m1.++(m2)
res15: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif, 103 -> Ram, 104 -> Tausif)
```

Printing Keys and Values from a Map:

We can use a foreach loop to walk through the keys and values of a Map:

```
scala> val m1 = Map((101 -> "Saif"), (102 -> "Arif"), (101, "Saif"))
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> m1.keys.foreach{i=>println(i)}
101
102

scala> m1.values.foreach{i=>println(i)}
Saif
Arif

scala> m1.keys.foreach{i=>println(i+" "+m1(i))}
101 Saif
102 Arif
```

```
package scala_practice
```

```
object Maps_Key_Value {
  val m1 = Map (101 -> "Saif", 102 -> "Arif", 103 -> "Ram")
  def main (args: Array [String]) {
    m1.keys.foreach {
      i => print ("Key: " +i)
        println ("\tValues: " +m1(i))
    }
  }
}
```

Searching for a Key in a Map:

The Map.contains() method will tell us if a certain key exists in the Map.

```
scala> val m1 = Map((101 -> "Saif"), (102 -> "Arif"), (101, "Saif"))
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> m1.contains(101)
res46: Boolean = true

scala> m1.contains(105)
res47: Boolean = false
```

```
package scala_practice
object Map_Contains {
  val m1 = Map (101 -> "Saif", 102 -> "Arif")
  def main (args: Array [String]) {

    if (m1.contains(101)) {
      println ("Key Exist with Value of: " +m1)
    }
    else {
      println ("Key does not exist")
    }
    if (m1.contains(102)) {
      println ("Key Exist with Value of: " +m1)
    }
    else {
      println ("Key does not exist")
    }
  }
}
```

Methods to Call on a Map:

We can call the following methods on a Map. (Note that they don't modify the original Map)

1) def ++

This concatenates two Maps.

```
scala> val m1 = Map(101 -> "Saif", 102 -> "Arif", 101 -> "Saif")
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> val m2 = Map(103 -> "Ram", 105 -> "Tausif")
m2: scala.collection.immutable.Map[Int,String] = Map(103 -> Ram, 105 -> Tausif)

scala> m1 ++ m2
res55: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif, 103 -> Ram, 105 -> Tausif)

scala> m1.++(m2)
res56: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif, 103 -> Ram, 105 -> Tausif)
```

2) def -

This returns a new Map omitting the pairs for the keys mentioned in the arguments.

```
scala> val m1 = Map(101 -> "Saif", 102 -> "Arif", 101 -> "Saif")
m1: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif)

scala> m1 - (101)
res57: scala.collection.immutable.Map[Int,String] = Map(102 -> Arif)

scala> m1.-(101)
res58: scala.collection.immutable.Map[Int,String] = Map(102 -> Arif)
```

4) def contains Boolean

If the Map contains this key, this returns true; otherwise, false.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.contains(1)
res84: Boolean = true

scala> m1.contains(5)
res85: Boolean = false
```

5) def drop

This returns all elements except the first n.

```
scala> val m1: Map[Int, String] = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.drop(1)
res116: scala.collection.mutable.Map[Int,String] = Map(1 -> One, 3 -> Three)
```

6) def dropRight

This returns all elements except the last n.

```
scala> val m1: Map[Int, String] = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.dropRight(1)
res117: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One)
```

7) def equals Boolean

This returns true if both Maps contain the same key-value pairs; otherwise, false.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> val m2 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m2: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.equals(m2)
res125: Boolean = true
```


8) def last

This returns the last element from the Map in Scala.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.last
res131: (Int, String) = (3,Three)
```

9) def max

This returns the largest element.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.max
res132: (Int, String) = (3,Three)
```

10) def min

This returns the smallest element.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.min
res133: (Int, String) = (1,One)
```

11) def sum

This returns the sum of all elements in the Map in Scala.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.keys.sum
res144: Int = 6
```

12) def tail

This returns all elements except the last.

```
scala> val m2 = Map(101 -> "Saif", 102 -> "Arif", 103 -> "Ram")
m2: scala.collection.immutable.Map[Int,String] = Map(101 -> Saif, 102 -> Arif, 103 -> Ram)

scala> m2.tail
res84: scala.collection.immutable.Map[Int,String] = Map(102 -> Arif, 103 -> Ram)

scala> m2.head
res85: (Int, String) = (101,Saif)
```

13) def take (n: Int)

This returns the first n elements from the Map.

```
scala> val m1 = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One, 3 -> Three)

scala> m1.take(2)
res150: scala.collection.mutable.Map[Int,String] = Map(2 -> Two, 1 -> One)
```

Tuple:

A *tuple* is a Scala collection which can hold multiple values of same or different type together. Unlike an array or list, a tuple can hold objects with different types but they are also immutable. A tuple is a collection of elements in ordered form. If there is no element present, it is called emptytuple.

Let us understand the same with an example of tuple holding an integer, a string, and a double with two different type of Syntax:

```
scala> val t = new Tuple3(1, "Saif", 20.20)
t: (Int, String, Double) = (1,Saif,20.2)

scala> val t = (1, "Saif", 20.20)
t: (Int, String, Double) = (1,Saif,20.2)
```

The actual type of a tuple depends upon the number of elements it contains and the type of those elements. Thus, the type of (1, "Saif", 20.20) is Tuple3[Int, String, Double]. Tuple are of type Tuple1, Tuple2, Tuple3 and so on. There is an upper limit of 22 for the element in the tuple, if you need more elements, then you can use a collection, not a tuple.

To access elements of a tuple t, you can use method t._1 to access the first element, t._2 to access the second, and so on. For example, the following expression computes the sum of all elements of t.

```
scala> val t = (1, 2, 3, 4)
t: (Int, Int, Int, Int) = (1,2,3,4)

scala> t._1
res2: Int = 1

scala> t._3
res3: Int = 3

scala> val sum = t._1 + t._2 + t._3 + t._4
sum: Int = 10
```

```
object Tuple_1 {
  def main (args: Array[String]) {
    val t = (4,3,2,1)
    val sum = t._1 + t._2 + t._3 + t._4
    println ("Sum of elements: " +sum)
  }
}
```

Iteration over the tuple:

You can use Tuple.productIterator() method to iterate over all of the elements of a tuple.

```
scala> val t = (1, 2, 3, 4)
t: (Int, Int, Int, Int) = (1,2,3,4)

scala> t.productIterator.foreach(println)
1
2
3
4

scala> t.productIterator.foreach(i => println ("Value of Tuple is: " +i))
Value of Tuple is: 1
Value of Tuple is: 2
Value of Tuple is: 3
Value of Tuple is: 4
```

Swapping of tuple elements:

We can perform swapping of tuple elements by using tuple.swap method.

```
scala> val t = ("Hello", "World")
t: (String, String) = (Hello,World)

scala> t.swap
res12: (String, String) = (World,Hello)
```

Converting tuple to String:

We can use tuple.toString() method to concatenate elements of tuple to string.

```
scala> val t = (1, "Hello", "World", 20.20)
t: (Int, String, String, Double) = (1,Hello,World,20.2)

scala> t.toString
res14: String = (1,Hello,World,20.2)
```

Closures:

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

We've seen how to declare an anonymous function in Scala:

```
scala> val sum = {a: Int, b: Float} => a + b
sum: (Int, Float) => Float = <function2>
```

Let's make a call to it. This will add the numbers 5 and 5, and returned 10.

```
scala> sum (5, 5)
res0: Float = 10.0
```

Now, consider this version: Here, the function 'sum' refers to the variable value 'c', which isn't a formal parameter to it.

```
scala> val c = 10
c: Int = 10

scala> val sum = {a: Int, b: Int} => {a + b} * c
sum: (Int, Int) => Int = <function2>
```

Let's try calling the function.

```
scala> sum (5, 5)
res1: Int = 100
```

Let's work up another Scala Closures example.

```
scala> val age = 30
age: Int = 30

scala> val sayHello = {name: String} => println (s"I am $name and my age is $age")
sayHello: String => Unit = <function1>
```

Let's call the function:

```
scala> sayHello ("Saif")
I am Saif and my age is 30
```

Now, let's take another function func that takes two arguments: a function and a string, and calls that function on that string. Scala supports **higher-order functions**.

```
scala> def func (f: String => Unit, s: String) { f(s) }
func: (f: String => Unit, s: String)Unit
```

```
scala> func (sayHello, "Saif")
I am Saif and my age is 30
```

Pattern Matching:

Pattern matching is the second most widely used feature of Scala, after function, values and closures. Scala provides great support for 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.

```
scala> import scala.util.Random
import scala.util.Random

scala> val x: Int = Random.nextInt(7)
x: Int = 3

scala> x match {
  | case 1 => "One"
  | case 2 => "Two"
  | case _ => "No Match"
  | }
res3: String = No Match
```

This gives us a random integer between 0 and 7. `x` is the left operand of the ‘match’ operator and the right operand is an expression with three cases/alternatives. In the end, we provide a “catch all” case for any value that doesn’t match any of the given case values. When a value matches the value of `x`, the value of the corresponding expression is evaluated.

In the following example, it tests the integer `x`. If it is 1, it returns the string “One”; if it is 2, it returns “Two”; or if it is anything else, it returns “No Match”. You can say that this maps integers to strings.

```
scala> def test(x: Int): String = x match {
  | case 1 => "One"
  | case 2 => "Two"
  | case _ => "No Match"
  | }
test: (x: Int)String

scala> test(5)
res4: String = No Match

scala> test(2)
res5: String = Two
```

Package scala_practice

```
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"
  }
}
```

Output:

many

Try the following example program, which matches a value against patterns of different types.

```
scala> def test(x: Any): Any = x match {
|   case 1 => "One"
|   case "Two" => 2
|   case (y: Int) => "Scala.Int"
|   case _ => "No Match"
|   }
test: (x: Any)Any
```

```
scala> test(5)
res12: Any = Scala.Int

scala> test("Two")
res13: Any = 2

scala> test("Saif")
res14: Any = No Match
```

```
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"
  }
}
```

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. As we've previously discussed in on Case Classes, a case class is good for modeling immutable data. It has all vals; this makes it immutable.

```
scala> case class Dog(breed: String, name: String, age: Int)
defined class Dog
```

```
scala> val a = new Dog("Tier_1","Leo",2)
a: Dog = Dog(Tier_1,Leo,2)

scala> val b = new Dog("Tier_2","Tommy",5)
b: Dog = Dog(Tier_2,Tommy,5)

scala> val c = new Dog("Tier_3","Jimmmy",3)
c: Dog = Dog(Tier_3,Jimmmy,3)
```

```
scala> for (i <- List(a,b,c)) {
  | i match {
  | case Dog ("Tier_1", "Leo", 2) => println("Hey Leo")
  | case Dog ("Tier_2", "Tommy", 5) => println("Hey Tommy")
  | case Dog (breed,name,age) => println("Hi, "+name)
  | }
  | }
Hey Leo
Hey Tommy
Hi, Jimmmy
```

```
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)
}
```

For the case class, the compiler assumes the parameters to be val. However, we can use the 'var' keyword to attain mutable fields.

Scala Pattern Guards:

We can also add conditions to the code. Simply add an if statement after the case.

```
scala> val x: Int = 7
x: Int = 7

scala> x match {
  | case 7 if x%2 == 0 => println("Nos is Even")
  | case 7 if x%2 == 1 => println("Nos is Odd")
  | }
Nos is Odd
```


Exception Handling:

Let's see Scala Exception and Exceptions Handling and learn about the Scala Try-Catch Blocks and the Throws Keyword in Scala. Along with this, we will cover Scala Finally Block and Scala Exception Handling.

There may be situations your code may malfunction when you run it. These abnormal conditions may cause your program to terminate abruptly. Such runtime errors are called exceptions.

Exception handling is a mechanism which is used to handle abnormal conditions. You can also avoid termination of your program unexpectedly.

Scala makes "checked vs unchecked" very simple. It doesn't have checked exceptions. All exceptions are unchecked in Scala, even SQLException and IOException.

Throwing Exceptions:

Throwing an exception looks the same as in Java. You create an exception object and then you throw it with the throw keyword as follows.

throw new IllegalArgumentException

Catching Exceptions:

Scala allows you to try/catch any exception in a single block and then perform pattern matching against it using case blocks. Try the following example program to handle exception.

Scala Exception Example:

Take a look at the following Scala Exception example. We declare a function to divide two integers and return the result.

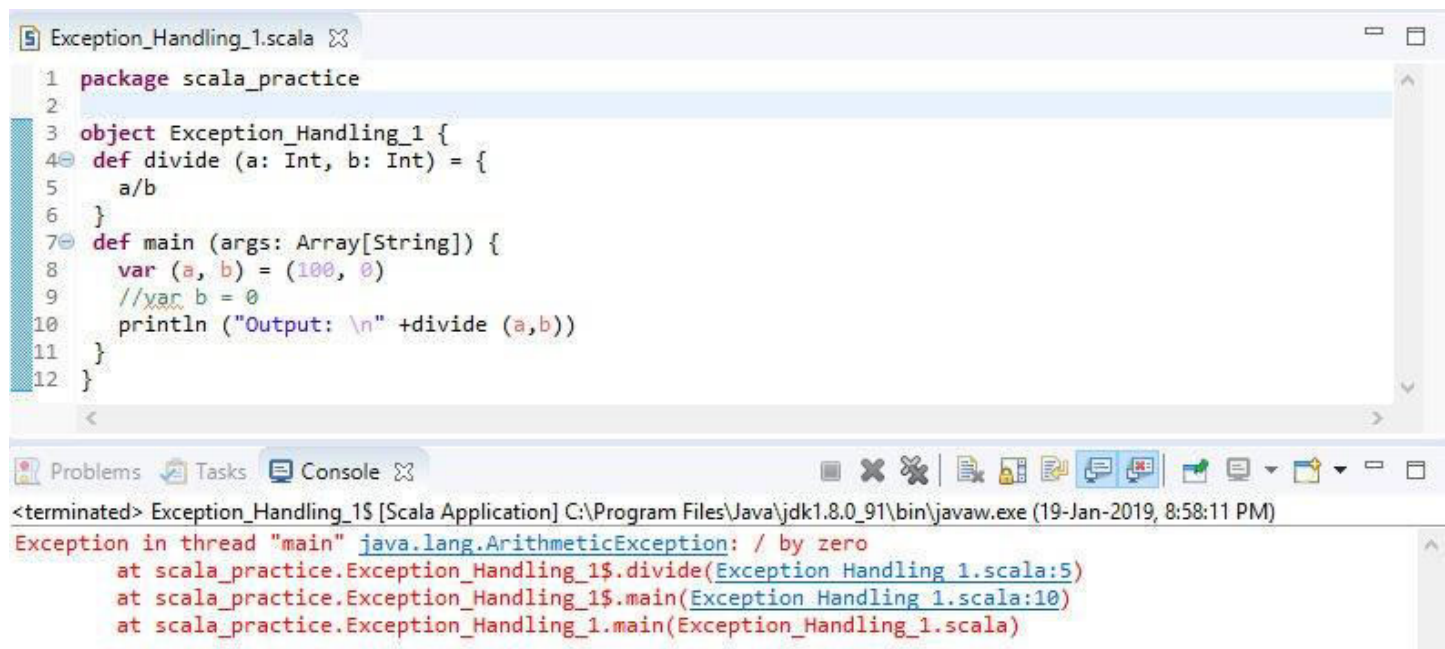
```
scala> def div(a: Int, b: Int): Float = {
|   (a/b)
| }
div: (a: Int, b: Int)Float
```

Now, let's call it:

```
scala> div(1,0)
java.lang.ArithmeticException: / by zero
```

As you can see, this raises an Arithmetic Exception in Scala.

```
class ExceptionExample {  
  def divide (a: Int, b: Int) = {  
    a/b          // Exception occurred here  
    println ("Rest of the code is executing...")  
  }  
}  
  
object MainObject {  
  def main (args: Array[String]) {  
    var e = new ExceptionExample ()  
    e.divide (100,0)  
  }  
}
```



The screenshot shows an IDE window titled "Exception_Handling_1.scala". The code defines a package `scala_practice` with an object `Exception_Handling_1`. Inside the object, there is a `divide` method that takes two integers `a` and `b` and returns `a/b`. The `main` method initializes `a` to 100 and `b` to 0, then calls `divide(a, b)` and prints the result. The console output shows a `java.lang.ArithmeticException: / by zero` error, indicating that the division by zero occurred. The stack trace points to the `divide` method at line 5 and the `main` method at line 10.

```
1 package scala_practice  
2  
3 object Exception_Handling_1 {  
4   def divide (a: Int, b: Int) = {  
5     a/b  
6   }  
7   def main (args: Array[String]) {  
8     var (a, b) = (100, 0)  
9     //var b = 0  
10    println ("Output: \n" +divide (a,b))  
11  }  
12 }
```

<terminated> Exception_Handling_1\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (19-Jan-2019, 8:58:11 PM)
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at scala_practice.Exception_Handling_1\$.divide(Exception_Handling_1.scala:5)
 at scala_practice.Exception_Handling_1\$.main(Exception_Handling_1.scala:10)
 at scala_practice.Exception_Handling_1.main(Exception_Handling_1.scala)

Scala Try-Catch Blocks:

When we suspect that a line or a block of code may raise an exception in Scala, we put it inside a try block. **What follows is a catch block.** We can make use of any number of Try-Catch Blocks in a program.

Scala provides try and catch block to handle exception. The try block is used to enclose suspect code. The catch block is used to handle exception occurred in try block. You can have any number of try catch block in your program according to need. If any exception occurs, catch handler will handle it and program will not terminate abnormally

```
scala> def div (a: Int, b: Int): Float = {
|   try {
|     a/b
|   } catch {
|     case e: ArithmeticException => println(e)
|   }
|   0
| }
div: (a: Int, b: Int)Float
```

Now, let's call it:

```
scala> div (1,0)
java.lang.ArithmeticException: / by zero
res1: Float = 0.0
```

Let's take another example of Scala Exception Handling:

```
scala> def func(n: Int) {
|   try {
|     print(1/n)
|     var arr = Array(1,4)
|     arr(17)
|   } catch {
|     case e: ArithmeticException => println(e)
|     case anon: Throwable => println("Unkown exception: " +anon)
|   }
| }
func: (n: Int)Unit
```

Now, let's call it:

```
scala> func(1)
1Unkown exception: java.lang.ArrayIndexOutOfBoundsException: 17
```

Throwable is a super class in the exception hierarchy. So, if we want our code to be able to handle any kind of exception, we use Throwable.

```

package scala_practice
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Exception_1 {
  def main (args: Array[String]) {
    try {
      val f = new FileReader ("Input.txt")
    } catch {
      case a: FileNotFoundException => {
        println ("Missing File Exception")
      }
      case b: IOException => {
        println ("IO Exception")
      }
    }
  }
}

```

Output:

Missing File Exception

The screenshot shows an IDE window titled 'Exception_Handling_Try_Catch_1.scala'. The code defines a class 'ExceptionExample' with a 'divide' method that uses a try-catch block to handle 'ArithmeticException'. An object 'MainObject' contains a 'main' method that creates an instance of 'ExceptionExample' and calls 'divide(100,0)'. The console output at the bottom shows the execution of the main method, resulting in a 'java.lang.ArithmeticException: / by zero' error, followed by the message 'Rest of the code is executing...'.

```

1 package scala_practice
2
3 class ExceptionExample {
4   def divide (a:Int, b:Int) = {
5     try {
6       a/b
7     } catch {
8       case e: ArithmeticException => println(e)
9     }
10    println("Rest of the code is executing...")
11  }
12 }
13 object MainObject{
14   def main(args:Array[String]){
15     var e = new ExceptionExample()
16     e.divide(100,0)
17   }
18 }
19 }
20

```

Problems Tasks Console

<terminated> MainObject\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (19-Jan-2019, 9:12:46 PM)

java.lang.ArithmeticException: / by zero

Rest of the code is executing...

Scala Finally Block:

Imagine, if you're working with a resource and an exception occurs in the middle. You still haven't released the resource. This could be a file, a network connection, or even a database connection. To deal with such a situation, we have the Finally Block. We can release all resources in this block. Well, whether an exception happens in your code or not, the code under 'finally' will run, no matter what.

```
scala> def func(a: Int, b: Int): Float = {
      |   try {
      |     a/b
      |   } catch {
      |     case e: ArithmeticException => println(e)
      |   }
      |   finally {
      |     println("This will print no matter what!")
      |   }
      |   0
      | }
func: (a: Int, b: Int)Float
```

Now let's call it:

```
scala> func(1,0)
java.lang.ArithmeticException: / by zero
This will print no matter what!
res3: Float = 0.0
```

```
package scala_practice
import java.io.FileReader
import java.io.FileNotFoundException
import java.io.IOException

object Exception_2 {
  def main (args: Array[String]) {
    try {
      val f = new FileReader ("Input.txt")
    } catch {
      case a: FileNotFoundException => {
        println ("Missing File Exception")
      }
      case b: IOException => {
        println ("IO Exception")
      }
    }

    finally {
      println ("No matter what, This will Print")
    }
  }
}
```

Output:

```
Missing File Exception
No matter what, This will Print
```

```
Exception_Finally_1.scala
1 package scala_practice
2
3 class Exception_Finally_1 {
4     def divide(a:Int, b:Int) = {
5         try {
6             a/b
7             var arr = Array(1,2)
8             arr(10)
9         } catch {
10             case e: ArithmeticException => println(e)
11             case ex: Exception => println(ex)
12             case th: Throwable => println("found a unknown exception"+th)
13         }
14         finally {
15             println("Finally block always executes")
16         }
17         println("Rest of the code is executing...")
18     }
19 }
20 object Exc_Fin {
21     def main(args:Array[String]){
22         var e = new Exception_Finally_1()
23         e.divide(100,10)
24     }
25 }
```

Problems Tasks Console

<terminated> Exc_Fin\$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (19-Jan-2019, 9:23:24 PM)

[java.lang.ArrayIndexOutOfBoundsException: 10](#)

Finally block always executes

Rest of the code is executing...

Scala Throw Keyword:

We can also explicitly throw a Scala exception in our code. We use the Throw Keyword for this. Let's create a custom exception.

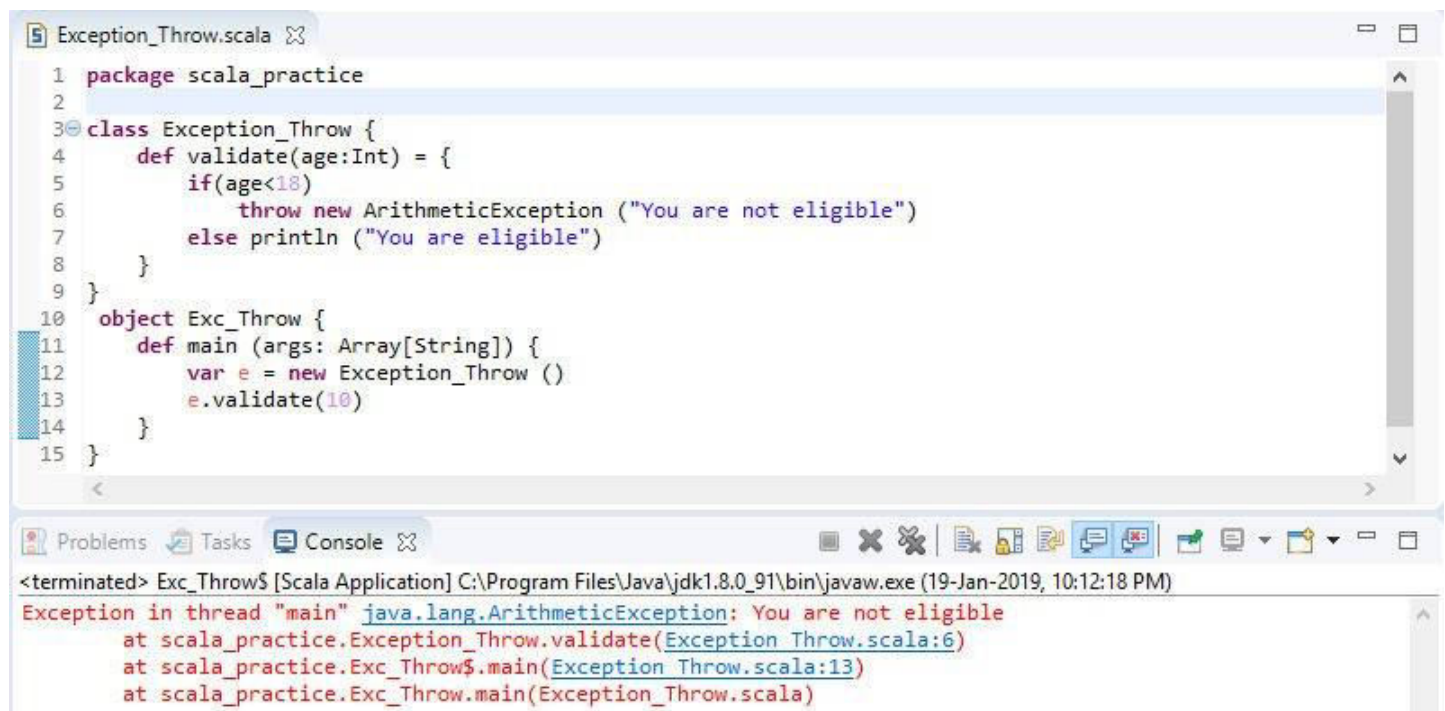
In Jewish culture, Bat Mitzvah is coming with age ceremony with age 12 for young boys.

```
scala> def batmitzvah(age: Int) {
|   if(age<12) {
|     throw new Exception("Sorry, you are not Eligible")
|   } else {
|     println ("You are Eligible")
|   }
| }
batmitzvah: (age: Int)Unit
```

Now, let's call it:

```
scala> batmitzvah(20)
You are Eligible

scala> batmitzvah(10)
java.lang.Exception: Sorry, you are not Eligible
```



The screenshot shows an IDE window titled 'Exception_Throw.scala'. The code defines a package 'scala_practice' containing a class 'Exception_Throw' and an object 'Exc_Throw'. The class has a 'validate' method that throws an 'ArithmeticException' if the age is less than 18, or prints 'You are eligible' otherwise. The object 'Exc_Throw' has a 'main' method that creates an instance of 'Exception_Throw' and calls 'validate' with the value 10. The console output at the bottom shows the execution of the 'main' method, which results in a 'java.lang.ArithmeticException: You are not eligible' being thrown at line 6 of 'Exception_Throw.scala'.

```
1 package scala_practice
2
3 class Exception_Throw {
4   def validate(age: Int) = {
5     if(age<18)
6       throw new ArithmeticException ("You are not eligible")
7     else println ("You are eligible")
8   }
9 }
10 object Exc_Throw {
11   def main (args: Array[String]) {
12     var e = new Exception_Throw ()
13     e.validate(10)
14   }
15 }
```

```
<terminated> Exc_Throw$ [Scala Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (19-Jan-2019, 10:12:18 PM)
Exception in thread "main" java.lang.ArithmeticException: You are not eligible
    at scala_practice.Exception_Throw.validate(Exception_Throw.scala:6)
    at scala_practice.Exc_Throw$.main(Exception_Throw.scala:13)
    at scala_practice.Exc_Throw.main(Exception_Throw.scala)
```