Chapter 5: Data Types

Exercise1:

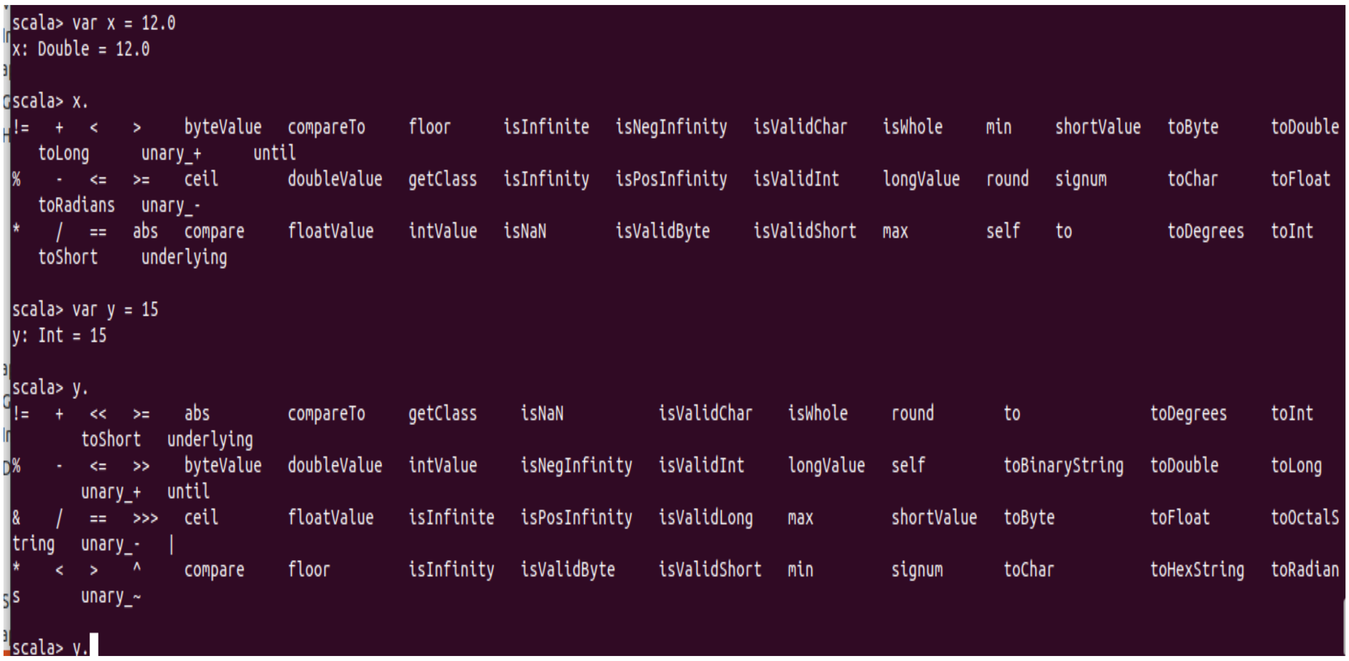
1. Understand the difference, also in the context of memory footprint and range, between different numeric types in Scala. What are the limitations of each and when does it make sense to use one over the other?
2. Research what functions are available in the integer data type, i.e. addition, subtraction, multiplication, and division. Use them in Scala REPL.
3. Research which operators on numeric types have precedence, i.e., in an expression, which operation will be executed before the other. Understand these basic concepts on your own.

Answer1 :

1.

Sr.No	Data Type & Description
1	**Byte** 8 bit signed value. Range from -128 to 127
2	**Short** 16 bit signed value. Range -32768 to 32767
3	**Int** 32 bit signed value. Range -2147483648 to 2147483647
4	**Long** 64 bit signed value. -9223372036854775808 to 9223372036854775807
5	**Float** 32 bit IEEE 754 single-precision float
6	**Double** 64 bit IEEE 754 double-precision float

2.



```
scala> var x = 12.0
x: Double = 12.0

scala> x.
!=   +   <    >     byteValue   compareTo    floor      isInfinite   isNegInfinity   isValidChar    isWhole     min      shortValue   toByte      toDouble
   toLong      unary_+      until
%    -   <=   >=    ceil        doubleValue  getClass   isInfinity   isPosInfinity   isValidInt     longValue   round    signum       toChar      toFloat
   toRadians   unary_-
*    /   ==   abs   compare     floatValue   intValue   isNaN        isValidByte     isValidShort   max         self     to           toDegrees   toInt
   toShort     underlying

scala> var y = 15
y: Int = 15

scala> y.
!=   +   <<   >=    abs         compareTo    getClass     isNaN           isValidChar    isWhole     round         to               toDegrees     toInt
        toShort   underlying
%    -   <=   >>    byteValue   doubleValue  intValue     isNegInfinity   isValidInt     longValue   self          toBinaryString   toDouble      toLong
        unary_+   until
&    /   ==   >>>   ceil        floatValue   isInfinite   isPosInfinity   isValidLong    max         shortValue    toByte           toFloat       toOctalS
tring   unary_-   |
*    <   >    ^     compare     floor        isInfinity   isValidByte     isValidShort   min         signum        toChar           toHexString   toRadian
s       unary_~

scala> y.
```

3. Operators Precedence in Scala

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Take a look at the following table. Operators with the highest precedence appear at the top of the table and those with the lowest precedence appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() []	Left to right
Unary	! ~	Right to left
Multiplicative	* / %	Left to right

Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	

Exercise2:

4. Research the different types of logical operators available in Scala. Try to use them in Scala REPL.
5. Try assigning a Boolean variable to an Integer variable. What do you get? Research whether you can do this in other languages.
6. Try adding two Boolean values. What do you get?

Answer 2:

1. Logical Operators

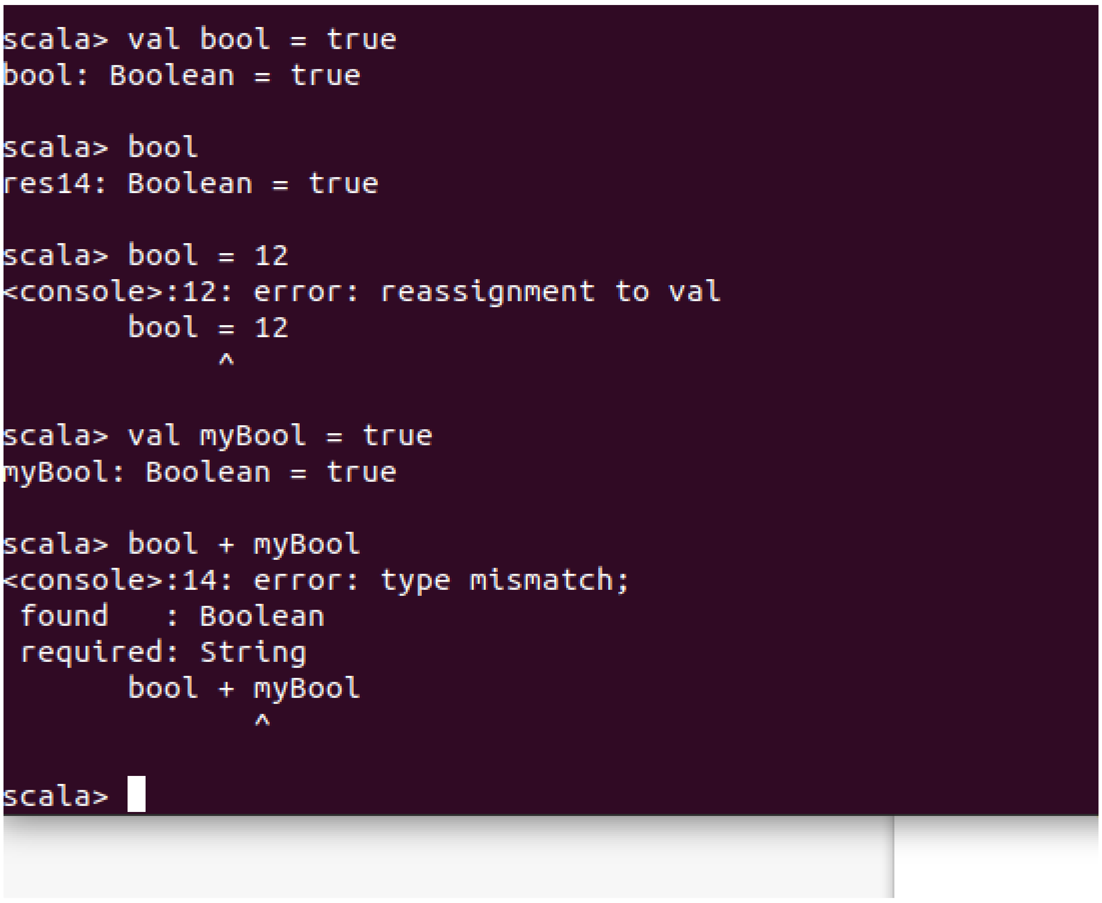
The following logical operators are supported by Scala language. For example, assume variable A holds 1 and variable B holds 0, then −

Show Examples

Operator	Description	Example
&&	It is called Logical AND operator. If both the operands are non zero then condition becomes true.	(A && B) is false.
||	It is called Logical OR Operator. If any of the two operands is non zero then condition becomes true.	(A || B) is true.

!	It is called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

2.

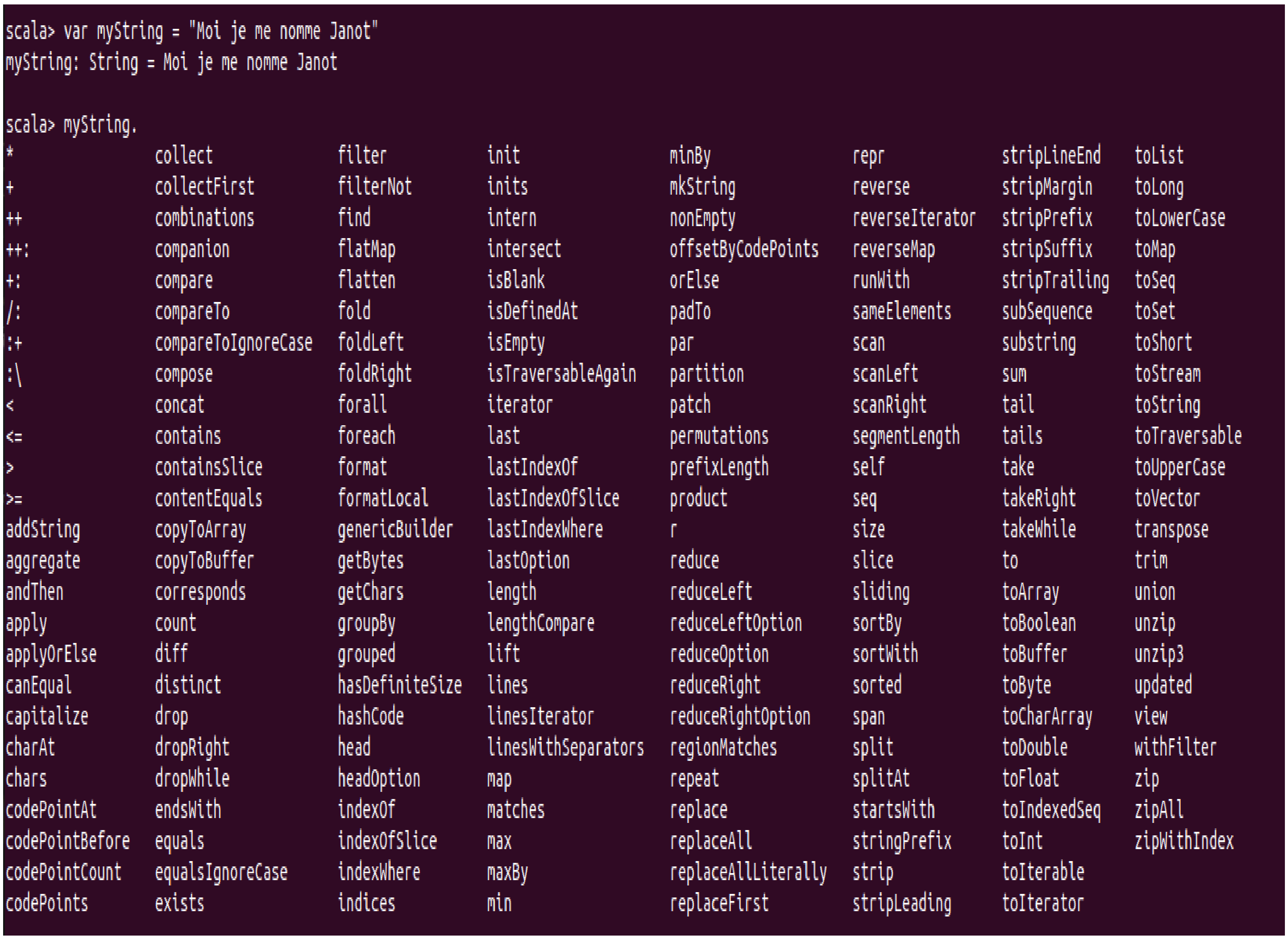


```
scala> val bool = true
bool: Boolean = true

scala> bool
res14: Boolean = true

scala> bool = 12
<console>:12: error: reassignment to val
       bool = 12
            ^

scala> val myBool = true
myBool: Boolean = true

scala> bool + myBool
<console>:14: error: type mismatch;
 found   : Boolean
 required: String
       bool + myBool
              ^

scala> 
```

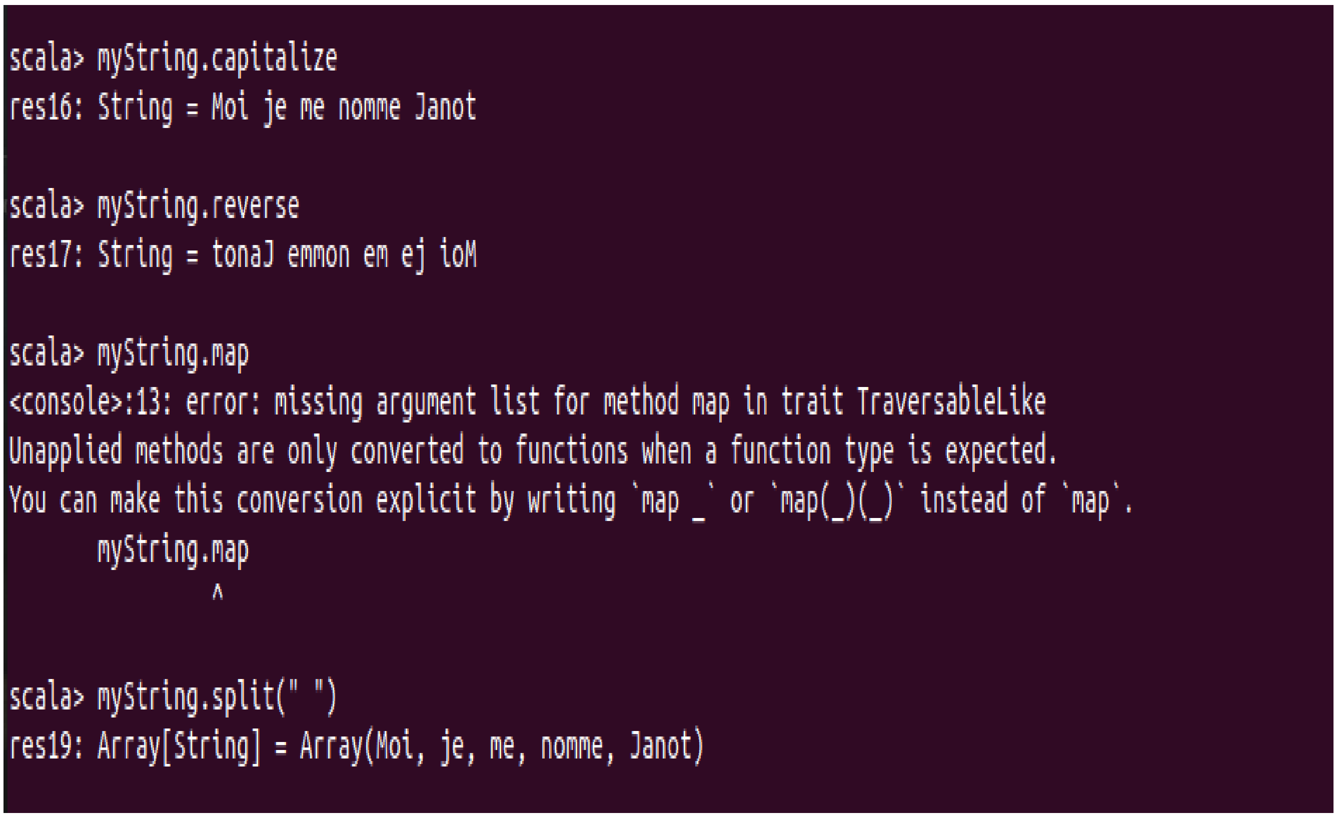
Exercise3:

7. Create a string variable and then type . (the dot character) and press Tab. You will see a list of functions. Many of them are covered in this book; however, explore them and learn what they do. The more you know about them, the better.
8. Try converting numeric types and Boolean types to String types. Did you have any issue in doing so? You shouldn't.

Answer 3:



```
scala> var myString = "Moi je me nomme Janot"
myString: String = Moi je me nomme Janot

scala> myString.
*                collect              filter           init                 minBy                repr             stripLineEnd   toList
+                collectFirst         filterNot        inits                mkString             reverse          stripMargin    toLong
++               combinations         find             intern               nonEmpty             reverseIterator  stripPrefix    toLowerCase
++:              companion            flatMap          intersect            offsetByCodePoints   reverseMap       stripSuffix    toMap
+:               compare              flatten          isBlank              orElse               runWith          stripTrailing  toSeq
/:               compareTo            fold             isDefinedAt          padTo                sameElements     subSequence    toSet
:+               compareToIgnoreCase  foldLeft         isEmpty              par                  scan             substring      toShort
:\               compose              foldRight        isTraversableAgain   partition            scanLeft         sum            toStream
<                concat               forall           iterator             patch                scanRight        tail           toString
<=               contains             foreach          last                 permutations         segmentLength    tails          toTraversable
>                containsSlice        format           lastIndexOf          prefixLength         self             take           toUpperCase
>=               contentEquals        formatLocal      lastIndexOfSlice     product              seq              takeRight      toVector
addString        copyToArray          genericBuilder   lastIndexWhere       r                    size             takeWhile      transpose
aggregate        copyToBuffer         getBytes         lastOption           reduce               slice            to             trim
andThen          corresponds          getChars         length               reduceLeft           sliding          toArray        union
apply            count                groupBy          lengthCompare        reduceLeftOption     sortBy           toBoolean      unzip
applyOrElse      diff                 grouped          lift                 reduceOption         sortWith         toBuffer       unzip3
canEqual         distinct             hasDefiniteSize  lines                reduceRight          sorted           toByte         updated
capitalize       drop                 hashCode         linesIterator        reduceRightOption    span             toCharArray    view
charAt           dropRight            head             linesWithSeparators  regionMatches        split            toDouble       withFilter
chars            dropWhile            headOption       map                  repeat               splitAt          toFloat        zip
codePointAt      endsWith             indexOf          matches              replace              startsWith       toIndexedSeq   zipAll
codePointBefore  equals               indexOfSlice     max                  replaceAll           stringPrefix     toInt          zipWithIndex
codePointCount   equalsIgnoreCase     indexWhere       maxBy                replaceAllLiterally  strip            toIterable
codePoints       exists               indices          min                  replaceFirst         stripLeading     toIterator
```

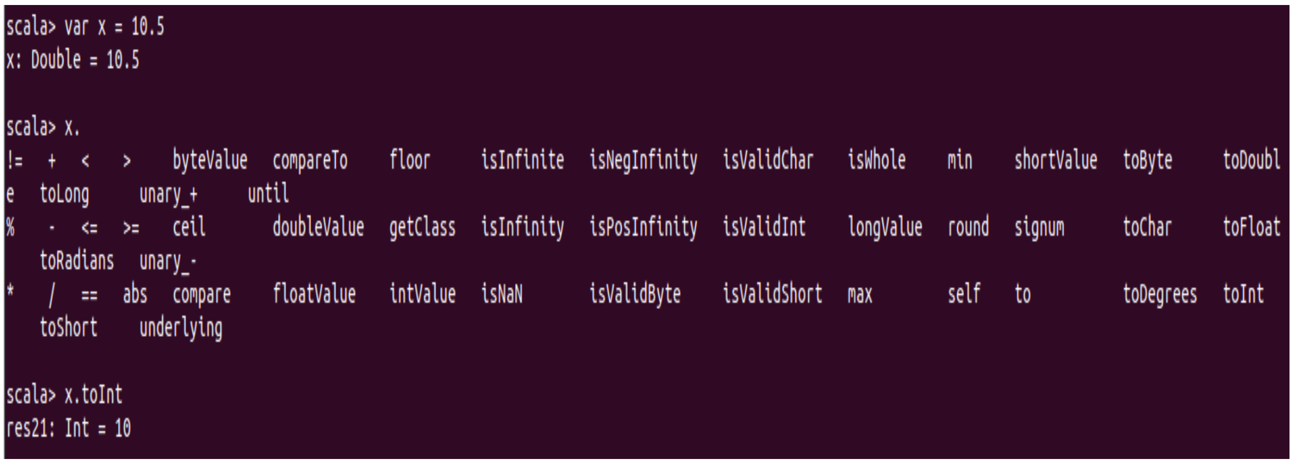


```
scala> myString.capitalize
res16: String = Moi je me nomme Janot

scala> myString.reverse
res17: String = tonaJ emmon em ej ioM

scala> myString.map
<console>:13: error: missing argument list for method map in trait TraversableLike
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `map _` or `map(_)(_)` instead of `map`.
       myString.map
                ^

scala> myString.split(" ")
res19: Array[String] = Array(Moi, je, me, nomme, Janot)
```

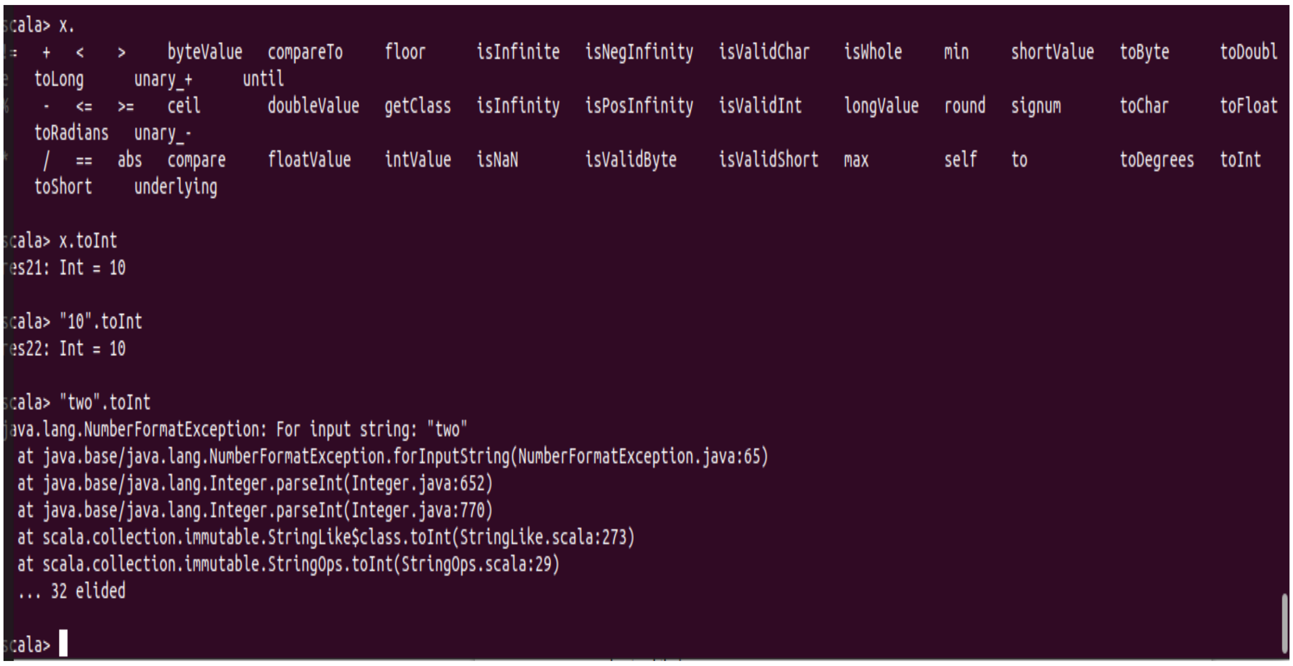
Exercise4:

9. Try converting a Double (e.g., 10.5) to Int. What happens? It will drop the portion of number after the decimal. Beware of such nuances.
10. Try running "10".toInt. Does it work? It should. Try to convert "two".toInt. Does it work? It shouldn't. You can't type cast all the time.
11. Research how you generally work with nulls in Scala. You will find specific types, such as Option and its concrete subtypes (Some, None). Research them and make sure you understand their use.

Answer 4:



```
scala> var x = 10.5
x: Double = 10.5

scala> x.
!=   +   <    >    byteValue   compareTo    floor      isInfinite   isNegInfinity   isValidChar    isWhole     min     shortValue   toByte      toDoubl
e   toLong      unary_+      until
%    -   <=   >=   ceil        doubleValue  getClass   isInfinity   isPosInfinity   isValidInt     longValue   round   signum       toChar      toFloat
   toRadians   unary_-
*    /   ==   abs  compare     floatValue   intValue   isNaN        isValidByte     isValidShort   max         self    to           toDegrees   toInt
   toShort     underlying

scala> x.toInt
res21: Int = 10
```



```
cala> x.
=   +   <    >    byteValue   compareTo    floor      isInfinite   isNegInfinity   isValidChar    isWhole     min     shortValue   toByte      toDoubl
   toLong      unary_+      until
    -   <=   >=   ceil        doubleValue  getClass   isInfinity   isPosInfinity   isValidInt     longValue   round   signum       toChar      toFloat
   toRadians   unary_-
    /   ==   abs  compare     floatValue   intValue   isNaN        isValidByte     isValidShort   max         self    to           toDegrees   toInt
   toShort     underlying

cala> x.toInt
es21: Int = 10

cala> "10".toInt
es22: Int = 10

cala> "two".toInt
ava.lang.NumberFormatException: For input string: "two"
 at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
 at java.base/java.lang.Integer.parseInt(Integer.java:652)
 at java.base/java.lang.Integer.parseInt(Integer.java:770)
 at scala.collection.immutable.StringLike$class.toInt(StringLike.scala:273)
 at scala.collection.immutable.StringOps.toInt(StringOps.scala:29)
 ... 32 elided

cala> 
```

The Empty values in Scala are represented by Null, null, Nil, Nothing, None, and Unit. The explication of these empty values are as follows:

- **null:**

  The reference types such as Objects, and Strings can be nulland the value types such as Int, Double, Long, etc, cannot be null, the null in Scala is analogous to the null in Java.
- **Null:**

  It is a Trait, which is a subset of each of the reference types but is not at all a sub-type of value types and a single instance of Null is null. The reference types can be assigned null but the value types cannot be assigned null.
- **Nothing:**
  Nothing is also a Trait, which has no instances. It is a subset of each of the distinct types. The major motive of this Trait is to supply a return type for the methods which consistently throws an exception i.e, not even a single time returns generally. It is also helpful in providing a type for Nil.
- **Unit:**

  The Unit is Scala is analogous to the void in Java, which is utilized as a return type of a functions that is used with a function when the stated function does not returns anything.
- **Nil:**
  Nil is Considered as a List which has zero elements in it. The type of *Nil* is List[Nothing] and as stated above, that *Nothing* has no instances, we can have a List which is confirmed to be desolated.