Exercice 5

EXERCICES : TYPES DE DONNÉES

1. a) Comprendre la différence, également dans le contexte de l'empreinte mémoire et de la plage, entre les différents types numériques dans scala.

Type de données	Empreinte mémoire	Plage
Byte	Valeur signée 8 bits	-128 à 127
Short	Valeur signée 16 bits	-32768 à 32767
Int	Valeur signée 32 bits	-2147483648 à 2147483647
Long	Valeur signée 64 bits	-9223372036854775808 à 9223372036854775807
Float	Flottant simple précision IEEE 754 32 bits	
Double	Flottant double précision IEEE 754 64 bits	

b) disons quelles sont les limites de chacun et quand est-il judicieux d'utiliser l'un plutôt que l'autre :

Les types *Decimal*, *Single* et *Double* sont trois types basiques pour stocker des nombres à virgule flottante, c'est-à-dire des nombres ayant une partie décimale. Le type *Double* peut représenter ces nombres de façon plus précise que le type *Simple* et est utilisé presque exclusivement dans les calculs scientifiques. Une variable de type *Integer* pourra stocker n'importe quel nombre, mais si on essaye de lui affecter un nombre décimal, la partie décimale sera purement et simplement tronquée, sans autre forme de procès !

Ainsi le choix du type adapté pour notre variable doit être le plus judicieux possible en fonction des valeurs qui pourront, au cours de l'exécution du programme, lui être affectées afin de trouver le juste milieu entre précision des données et vitesse d'exécution, sachant que la vitesse d'exécution est, en quelque sorte, inversement proportionnelle à la précision. C'est pour cela qu'il est important de connaître précisément les caractéristiques de chaque type.

2. rechercher quelles fonctions sont disponibles dans le type de données entier, c'est-à-dire l'addition, la soustraction, la multiplication et la division

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

val a:Int=5
val b:Int=10
val somme = a + b
val soustraction = b-a
val multiplication = a*b
val division = b / a
println("la somme de a et b est : " + somme)
println("la difference de a et b est :" + soustraction)
println("la multiplication de a et b est : " + multiplication)
println("la division de b sur a est :" + division)

// Exiting paste mode, now interpreting.

la somme de a et b est : 15
la difference de a et b est :5
la multiplication de a et b est : 50
la division de b sur a est :2
a: Int = 5
b: Int = 10
somme: Int = 15
soustraction: Int = 5
multiplication: Int = 50
division: Int = 2

scala>
```

3. Recherchons quels opérateurs sur les types numériques sont prioritaires, c'est-à-dire, dans une expression, quelle opération sera exécutée avant l'autre.

Lorsque plusieurs opérateurs apparaissent. Dans une même expression, il est nécessaire de savoir dans quel ordre ils sont mis en jeu. En scala, comme dans les autres langages, les règles sont naturelles et rejoignent celles de l'algèbre traditionnelle (en ce qui concerne les opérateurs arithmétiques)

Les opérateurs unaires + et - ont la priorité la plus élevée, On trouve on suite à un même niveau es opérateurs *, / et %. Enfin sur un dernier niveau apparaissent les opérateurs binaires + et -.

En cas de priorité identique, les calculs s'effectuent de gauche à droite. Enfin, des parenthèses permettent d'outre passer ces règles de porte en forçant e calcul préalable de l'expression qu'elles contiennent.

EXERCICES : TYPE BOOLÉEN

1. Recherchons les différents types d'opérateurs logiques disponibles dans Scala. Essayons de les utiliser dans scala repL.

Operateurs	Description
&&	Et logique
||	Ou logique
!	Non logique

Utilisation :

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

var a = true;
      var b = false;

      println("a && b = " + (a&&b) );

      println("a || b = " + (a||b) );

      println("!(a && b) = " + !(a && b) )

// Exiting paste mode, now interpreting.

a && b = false
a || b = true
!(a && b) = true
a: Boolean = true
b: Boolean = false

scala>
```

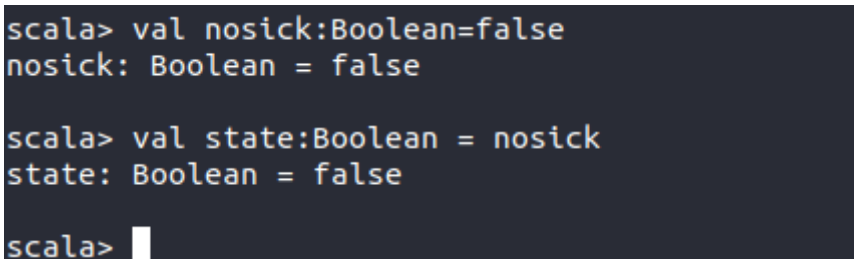
2. Essayons d'affecter une variable booléenne à une variable entière.

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

scala> val sick:Boolean=false
sick: Boolean = false

scala> val age:Int=sick
<console>:12: error: type mismatch;
 found   : Boolean
 required: Int
       val age:Int=sick
                   ^

scala>
```

Conclusion

Nous constatons que ça ne marche pas, on obtient une erreur.

3. Essayons d'ajouter deux valeurs booléennes.



Nous obtenons un boolean en sortie.

EXERCICES : TYPES DE CORDES

1. Créons une variable de chaîne, puis tapons. Et Tab Pour voir la liste des fonctions disponible

```
scala> val formation="bigdata"
formation: String = bigdata

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

2. Essayons de convertir les types numériques et les types booléens en types de chaîne.

- Types numériques en types string

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

scala> a.toString
res5: String = 10

scala> val b:Long = 1000
b: Long = 1000

scala> b.toString
res6: String = 1000
```

```
scala> val c:Float = 10
c: Float = 10.0

scala> c.toString
res8: String = 10.0

scala> val d:Double = 100
d: Double = 100.0

scala> d.toString
res9: String = 100.0

scala> val e:Short = 1
e: Short = 1

scala> e.toString
res10: String = 1

scala> val f:Byte = 1
f: Byte = 1

scala> f.toString
res11: String = 1

scala>
```

- Type boolean en types string

```
scala> etat.toString
res12: String = true

scala> val etat:Boolean = false
etat: Boolean = false

scala> etat.toString
res13: String = false

scala>
```

EXERCICES : TYPE CASTING

1. Essayez de convertir un Double (par exemple,10.5) en Int

```
scala> val note:Double=15
note: Double = 15.0

scala> note.toInt
res14: Int = 15
```

```
scala> val note=15.50
note: Double = 15.5

scala> note.toInt
res16: Int = 15
```

La partie du nombre après la décimale a été supprimée.

2. a) Essayons de convertir"10" en Int.

```
scala> val nbre="10"
nbre: String = 10

scala> nbre.toInt
res17: Int = 10
```

Nous constatons que ça marche bien

b) essayons de convertir "two" en Int

```
scala> val nbre="two"
nbre: String = two

scala> nbre.toInt
java.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

scala>
```

Nous constatons que cela ne marche pas

3. Recherchons comment travailler généralement avec des valeurs nulles dans scala.

Utilisation de Option/Some/None

Nous partirons d'un exemple :

Imaginez que vous vouliez écrire une méthode pour faciliter la conversion de chaînes en valeurs entières, et que vous vouliez une manière élégante de gérer les exceptions qui peuvent être levées lorsque votre méthode obtient une chaîne comme <<foo>> au lieu de quelque chose qui se convertit en nombre, comme « 1 ».

Solution avec Option/Some/None

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def toInt(s: String): Option[Int] = {
    try {
        Some(Integer.parseInt(s.trim))
    } catch {
        case e: Exception => None
    }
}

// Exiting paste mode, now interpreting.

toInt: (s: String)Option[Int]

scala> val a = toInt("1")
a: Option[Int] = Some(1)

scala> val a = toInt("foo")
a: Option[Int] = None

scala>
```