NWEN303 Assignment 3 Due date: 04 May at midnight
All answers should be submitted in a single PDF document [Total Marks: 100]

Introduction. Read the following text carefully [1 automatic mark]
Actors need to exchange as messages only immutable or encapsulated objects.
Messages are objects that either are sent (using “tell”, “ask” or “pipe”) or received by an actor.
In this assignment we assume that code will be running under a strict **SecurityManager** that will properly prevent access to private data and update of final fields using reflection.

##Examples about deeply immutable objects and references##

-"Hi"
is an expression referring to a deeply immutable (String) object.

- if a method has a parameter **'String x'**, **x** always refers to a deeply immutable object, since String is a final class.

-Collections.unmodifiableList(Arrays.asList("hello","world"))
is an expression referring to a deeply immutable object, since the inner list can only be accessed using the unmodifiableList wrapper.

-Collections.unmodifiableList(x)
is an expression referring to a mutable or immutable object, depending on how **x** is exposed to other code.

-new BigInteger("3")
is an expression referring to a deeply immutable object.

-If a method has a parameter **'BigInteger x'**, it is possible that **x** refers to a mutable object, since **'EvilBigInteger extends BigInteger'** could be present somewhere in the program.

##Examples about encapsulated objects and references##

-List.of(new BigInteger("3"))
is an expression referring to an encapsulated object

-List.of(x),
where **x** is declared directly above as **'x=new BigInteger("3")' :**
List.of(x) is an expression referring to an encapsulated object:
x can be aliased around but we know that in that moment x referred to a deeply immutable object

-List.of(x),
where **x** is a method parameter declared as **'BigInteger x':**
List.of(x) is an expression that may refer to externally aliased mutable state

-new ArrayList<String>(x),
where **x** is a method parameter declared as **'List<String> x':**
new ArrayList<String>(x) is an expression referring to an encapsulated object: it does not contain references to the list **x** but only to the immutable Strings inside of **x**.

Task 1 **[24 Marks]**

Consider the following code

```
@SuppressWarnings("serial")
class Point implements Serializable{
  public final int x; public final int y;
  public Point(int x, int y){this.x=x;this.y=y;}
  }

@SuppressWarnings("serial")
class Person implements Serializable{
  public String name;
  public Point location;
  public Person(String name, Point location){
    this.name=name;
    this.location=location;
    }
  public Person deepClone() {
    return new Person(this.name,new Point(location.x,location.y));
    }
  }
```

Questions: all questions are worth 6 marks each.
For each question, answer Yes/No, then Justify your answer.

Q1.1 Is '**new Point(0,0)**' an expression referring to a deeply immutable object?

Q1.2 Is '**new Person("bob",new Point(0,0))**' an expression referring to a deeply immutable object?

Q1.3 If a method has a parameter '**Point x**', will **x** always refer to a deeply immutable object (or null)?

Q1.4 If a method has a parameter '**Person x**', will **x** always refer to a deeply immutable object (or null)?

Task2 **[30 Marks]**

##Examples using .tell(..) correctly##

-actorRef.tell(new ArrayList<String>(List.of("hello","world",myString)),self())
Is correct since the newly created array list is encapsulated:
-no references to the arrayList or mutable objects in its ROG are kept by the actor.

-void msg(Object o){
List<String> l=new ArrayList<String>();
actorRef.tell(new ArrayList<List<String>>(List.of(l)),self());
}
Is correct as above: the newly created list is encapsulated; it does not matter if there are mutable subobjects, the important thing is that references to those subobjects are not kept by the actor.
That is, an encapsulated object can reference both deeply immutable objects and other mutable objects, but those mutable objects can only be reachable through the encapsulated object,
The encapsulated object and the mutable objects it contains are allowed to change.
Only one worker/actor at a time can refer to a specific encapsulated object.

-void msg(Object o){actorRef.tell(o,self());}
Is correct since the object 'o' must have been encapsulated or deeply immutable when it was received; we can assume that all received messages are encapsulated or deeply immutable, and we need to ensure that the sent messages also are encapsulated or deeply immutable.

-void msg(Object o){this.f=o; actorRef.tell(o,self());}
Is not correct, since the object 'o' is now stored in the ROG of the actor, and it could have been non-immutable (just encapsulated).

-void msg(String o){this.f=o; actorRef.tell(o,self());}
Is correct, since the String 'o' is immutable, is not a problem if it is stored in the ROG of the actor.

-void msg(String o){actorRef.tell(this.f,self());}
Could be correct only if we are sure than 'this.f' is a reference to a deeply immutable object.

Consider the following code

```
class A extends AbstractActor{
  ActorRef b;
  Point point;
  List<Person> persons;
  A(ActorRef b, Point point, List<Person> persons){
    this.b=b;
    this.point=point;
    this.persons=persons;
    }

  public Receive createReceive() {
    return receiveBuilder()
      .match(String.class,this::msg1)
      .match(Integer.class,this::msg2)
      .match(Double.class,this::msg3)
      .match(Point.class,this::msg4)
      .match(Character.class,this::msg5)
      .match(Person.class,this::msgHard)
      .build();
    }

  void msg1(String m){b.tell(new Point(0,0), self());}
  void msg2(Integer m){b.tell(new Person("Bob",new Point(0,0)), self());}
  void msg3(Double m){b.tell(this.point, self());}
  void msg4(Point m){b.tell(m, self());}
  void msg5(Character m){b.tell(this.persons.get(0), self());}

  void msgHard(Person m){b.tell(m, self());}//1

  void msgHard(Person m){b.tell(new Person("Bob",m.location), self());}//2

  void msgHard(Person m){//3
    b.tell(new Person("Bob",new Point(m.location.x,m.location.y)), self());}

  void msgHard(Person m){//4
    this.persons.add(m);
    b.tell(m, self());}

  void msgHard(Person m){//5
    this.persons.add(m);
    b.tell(m.deepClone(), self());}

  void msgHard(Person m){//6
    b.tell(this.persons, self());}

  void msgHard(Person m){//7
    b.tell(Collections.unmodifiableList(this.persons), self());}

  void msgHard(Person m){//8
    b.tell(new ArrayList<>(this.persons), self());}

  void msgHard(Person m){//9
    b.tell(this.persons.subList(0, 3), self());}
  }
```

Questions: all questions are worth 6 marks each.
For each question, answer Yes/No, then Justify your answer.

In the following you can assume:
-after construction the actor '**A**' controls its reachable object graph, thus no other actor can refer to mutable objects reachable by '**A**'.
-all objects **received** as messages are either encapsulated or deeply immutable.

What methods of '**A**' are correct? that is, what methods use '**.tell(..)**' only to send encapsulated or deeply immutable objects?

Q2.1 Is msg1 correct?

Q2.2 Is msg2 correct?

Q2.3 Is msg3 correct?

Q2.4 is msg4 correct?

Q2.5 Is msg5 correct?

Task3 **[45 Marks]**

The code above have 9 different versions of method **msgHard**.
what versions are correct, that is, what versions use **'.tell(..)'** only to send encapsulated or deeply immutable objects?

Questions: all questions are worth 5 marks each.
For each question, answer Yes/No, then Justify your answer.

Q3.1 Is version1 correct?

Q3.2 Is version2 correct?

Q3.3 Is version3 correct?

Q3.4 Is version4 correct?

Q3.5 Is version5 correct?

Q3.6 Is version6 correct?

Q3.7 Is version7 correct?

Q3.8 Is version8 correct?

Q3.9 Is version9 correct?

SUBMISSION OF ASSIGNMENT 3:
Submit all your answers in a single PDF file (or txt file).