Task1

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?

Yes.

1. The fields of the point object are final and they are primitive type int, which means that they cannot be changed in any way (object states cannot be changed).
2. The whole object cannot be extended by other class(no method in Point has parameter which could refer to mutable object.).

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

No.

This expression is almost immutable because the two arguments passed into the constructor of Person class are both immutable (we know that a string object is immutable and a point object is immutable from Q1.1)

The deepClone method also does not change the state of the person object.

However, the two fields of the person class are declared public, which means that these two fields can be modified elsewhere. Hence this expression is not deeply immutable.

Moreover, the constructor of Person has “Point x” as parameter. Potentially we could pass in an object that extends Point class and that object is not guaranteed immutable but in this case we do not have to worry about this because we know we are passing in a “new Point(0,0)”.

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

No. As mentioned in Q1.2. “Point x” could refer to an object of a class that extends Point class because Point class is not final. This class could have mutable fields or methods that modify its own state, which causes this object to be not immutable.

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

No. person objects are not deeply immutable.

The String field name is immutable.

The deepClone method at this moment is immutable (it uses the immutable string field and an immutable point object to construct and return a new Person and this method does not change the state of the current person object in any way).

However, the “Point location” parameter of the constructor could refer to a object of a class that extends Point class, which causes the Person object to be mutable because the field “Point location” could refer to a mutable object.

Moreover, “Person x” could refer to an object of a class that extends Person class, which adds even more possibilities of it referring to a mutable object.

Note that this code could have potential NullPointerException when the constructior passes “null” to the “Point location” field and the deepClone method is called on this person object because “location” could be null and calling x and y on a null causes NullPointerException. It is not going to affect the message passing process because before the message is received, an exception stops the process.

Task2

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

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());}
  }
```

Q2.1 Is msg1 correct?

Yes. New Point(0, 0) is a deeply immutable object, which can safely be sent.

Q2.2 Is msg2 correct?

Yes. The person object encapsulates a deeply immutable string object and a deeply immutable point object. However, The person object itself is not deeply immutable because it has public fields which are not properly encapsulated. The person object can be safely sent as a message because it is an encapsulated object.

Q2.3 Is msg3 correct?

Could be correct only if we are sure that the "this.point" field refers to a deeply immutable object. Since point has Point type, it is possible that it refers to a mutable object of a class that extends Point class. Also since class A is not properly encapsulated: (class A is not declared final!), it is possible that some classes extend A and in those classes there are methods that modify the public fields.

Q2.4 is msg4 correct?

Yes. We assume that the message received is either encapsulated or deeply immutable. In this case we assume that m refers to a deeply immutable object. So it is safe to send it as message.

Q2.5 Is msg5 correct?

No. Since class A is not final, there could be classes that extend A somewhere in the code and in these classes, there could potentially be methods that modify the "this.persons" field. So it is not safe to send it as message.

Task3

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?

Yes. Since we assume that the received message is either deeply immutable or encapsulated object, we can safely send the received message elsewhere.

Q3.2 Is version2 correct?

Yes. we know that the input m must refer to either a deeply immutable object or an encapsulated object. In this case the new person object is not deeply immutable because it has public fields, which could be seen and modified by others. But it is an encapsulated object because the reachable object graph of is disconnected with any fields of class A.

Q3.3 Is version3 correct?

Yes. As reasoned above in Q3.2, the new Person object will not be deeply immutable, but it will be an encapsulated object and it will be a safe message to send because the reachable object graph of this new person object will be disconnected with all fields in class A. The main difference between this question and Q3.2 is that this question uses the integer values x and y to construct new point object

whereas in Q3.2 it just uses the point object in the received person object. This is safe because both point objects are deeply immutable. There will not be race condition.

Q3.4 Is version4 correct?

No. The received message is first added to the persons field and then sent. This is problematic because class A is not final and m could be not deeply immutable, There could be classes extending class A somewhere in the code and in these classes some methods could modify the persons field. If the message sent to others is being modified and some methods are modifying the same object in an object extending A concurrently, there would be race condition.

Q3.5 Is version5 correct?

No. The input Person m could refer to a subclass of Person. The deepclone method may be overridden in the subclass and thus it is not guaranteed that the overridden deepclone method could be safe to send because it may have aliasing with the persons field.

Q3.6 Is version6 correct?

No. The field persons is sent out as message. The other actor who received the persons list and the current actor could potentially be modifying the list concurrently and thus there could be race condition and the contract of ArrayList could be broken: for example. Two add operations on the arraylist simultaneously.

Q3.7 Is version7 correct?

No. The unmodifiableList restricts operations like adding, removing, but we could still get a specific person in the list and change the state of this person because the fields of Person class are all public. It is not deeply immutable and it is not encapsulated because the field persons has the same reachable object graph as the unmodifiableList. Thus there would be race condition as two same person objects could be concurrently modified by 2 threads.

Q3.8 Is version8 correct?

No. wrapping the persons field in a new ArrayList object and sending it as message will still cause race condition because the persons list sent and the field persons share the same objects. If the field and the sent message are modified concurrently there may be race condition and the contract of arraylist may be broken.

Q3.9 Is version9 correct?

No. similar to the explanations in Q3.8 and Q3.7, the subList method returns only a wrapper object of the original list. The persons list and the relevant objects in the same list still share the same array and thus race condition cannot be avoided.