The game we will be implementing is called Decade. It's a one player version of a game where the player trades gems to acquire points, mediated by the cards available to them. The basic gameplay is relatively simple, however the combination of elements can make it tactically interesting (at least in the original multiplayer version, which has more complicated rules and a different name).

The player has a "caravan" of gems, which they use to eventually claim points. Gems come in four types, Yellow, Green, Blue and Pink, ranked in that order. The player starts the game with 3 Yellow gems.

The way the gems can be used is mediated by the cards the player has available. Playable cards come in two types:

Trade cards let the player trade a specific combination of gems for a different combination.
Upgrade cards allow the player to improve some quantity of their gems.

The player starts with a Trade card that costs nothing and gives 2 Yellow gems, and an Upgrade card that upgrades 2 gems.

A third type of card, the Points card, can be bought with gems, but once bought simply remains in the player's possession, contributing points at the end of the game.

The player can acquire Trade and Upgrade cards from the Merchant. The Merchant has 6 cards available at a time (drawn from a deck), the cards could be Trade or Upgrade cards. The player can purchase the $n^{th}$ card by paying n-1 gems of any type. The merchant then shifts everything after the purchased card down and adds a new card to the available set.

The player claims Point cards from the pool by paying the specific cost of the Point card. 5 Point cards are available at once, and are refilled when purchased in a similar fashion to the merchant cards (though the order doesn't matter).

Thus play proceeds by the player taking one of 4 actions:

Play a card in their hand.
If it's a Trade card and they can afford the cost, they proceed with the trade.
If it's an Upgrade card, the player performs as many gem upgrades as noted on the card, starting with the lowest ranked gem in their caravan. If they do not have sufficient upgradeable gems, any excess upgrade capacity is lost.
After a card is played, it gets put in the player's used card pile.
Rest. This action returns all the player's used cards to the player's hand.
Buy a card from the merchant (described above).
Claim a Point card (described above).

The game ends when either the player has 5 or more Point cards, or the Point card pool is empty. The player's score (at any point in the game, but particularly at the end) is the sum of the points values on the Point cards in the player's possession, plus the number of non-Yellow gems the player has.

Implementation Details

The game will be implemented to fulfil the rules above, with the following implementation choices:

1. All ordered things will be 0-indexed. In particular the cards in the Merchant's available cards will be numbered from 0 to 5 (as there are six cards), and the player's hand will be number from 0 to the size of the hand less 1. In reference to the rules above, the nth card has index $n-1$.

2. Paying for cards from the Merchant will spend lower rank gems first - e.g. if we need to spend 3 gems, and the player has two yellow and three green, the spend will be the two yellow gems and one green, leaving the player with two green gems.

3. Upgrades work in the same way, lower ranked gems are upgraded first, and upgrading continues until all the upgrades are used or there are no gems to upgrade. Thus a player with 1 yellow gem who plays an upgrade 3 card will end with 1 pink gem (yellow to green, then green to blue, then blue to pink).

4. Cards in hands/piles/deck/etc. should maintain their ordering according to the order they are added to the hand/pile/deck etc.. In particular the initial order of cards in the player's hand is the trade card first, then the upgrade card.

5. When convenient, trade cards will have their trade represented as ordered lists of 4 numbers, separated by an arrow, e.g. [1,0,0,1] -> [0, 0, 3, 1]. The list of all zeros can be shortened to []. This is only for communication, not part of the game.

**Part 1**
Getting Carded
For the first task, we will lay some of the groundwork and construct classes to represent the cards and the gems. We will take an Object Oriented approach, and flex some of our Java skills.

This approach will employ three key elements to standardise the structure and interfaces:

A sealed class Card:
We haven't seen this before, but there's nothing tricky here. A sealed class is just a normal class with the additional property that it specifies all of its child classes.
All of its children must be one of:
final
sealed
Non-sealed
Otherwise it's the same old inheritance structure.
The advantage of a sealed class is that we can do pattern-matching in a switch, and know that it's exhaustive. You don't actually need this to finish the assignment, but it's a cool feature to have available.
The full implementation is given as part of the scaffold, and you don't need to do anything to it.
Note that is is also abstract, as we shouldn't be able to get an object of type Card - we need a specific subtype.

An enum.
The interface Map and its implementation HashMap (though we won't need that part until later).
This is just the standard Java implementation of what Python calls a dict, so there's actually nothing new here, it's just a data structure that stores key-value pairs, with the value accessed by the key. The key methods you'll need from a Map<K, V> for this assessment (apart from the constructor later on) are put(K, V) and get(K). The linked documentation of course has more detail.

**The Task**
Create 3 classes that are all final and extend Card and one enum (which does not extend Card, I guess it can be final but it doesn't matter):
Gem, an enum, with 4 instances, YELLOW, GREEN, BLUE and PINK.
PointCard. It should have the following methods:
A constructor with two parameters:
A Map<Gem, Integer> which represents the cost of the card.

An int which represents the number of points the card is worth.
A method points that returns an int representing the number of points the card is worth.

A toString method that returns a String representing the card in the format Points: [cost] -> [points], where [cost] is replaced by a String with the cost of the card written as a comma separated list of the number of each gem and its type in the card cost, in order of increasing gem quality, if there is a non-zero cost, and "Nothing" if it has no cost, and [points] is replaced by the points value of the card. For example: Points: 3 GREEN, 2 PINK -> 14.

TradeCard. It should have the following methods:

A constructor with two parameters:

A Map<Gem, Integer> which represents the cost of the card.
A Map<Gem, Integer> which represents the value of the card (the gems gained by paying the cost).
A method value that returns a Map<Gem, Integer> representing the value of the card.
A toString method that returns a String representing the card in the format Trade: [cost] -> [value], where [cost] and [value] are formatted as before. For example: Trade: 2 BLUE -> 2 GREEN, 1 PINK.
UpgradeCard. It should have the following methods:
A constructor with one parameter:

An int representing the number of gems that the card will upgrade.

A method upgrades which returns an int that represents the number of gems the card will upgrade.

A toString method that returns a String representing the card in the format Upgrade: [upgrades], where [upgrades] is the number of upgrades the card can perform. For example: Upgrade: 2.

Some useful points:

Any data members in these should be private, though you can add other public (and private) methods if you so choose. None are needed, but they might be helpful.

You will need to get the basic framework of these classes in place before the tests will run - in particular you will need to fulfil the requirements imposed by Card being sealed (and create Gem).

You can import anything you like from the Java libraries, though you should only need java.util.Map. If you are importing other things, you may be overcomplicating things.

Note that for the three subclasses of Card you will have to explicitly call the constructor of Card. This may require some thought for Upgrade, which has no cost.

Note that there is no specification for the internal representation of this data - you are free to choose any way you think works, but keeping it simple is a good idea.

You may make supporting classes (though I can't see much utility in doing this except for some very specific things).

You are provided with a Runner class that contains the main method run when the Run button is used. This is not part of the assignment and is just there for convenience.

Code Java:

```
//Card.java
mport java.util.Map;

public abstract sealed class Card permits PointCard, TradeCard, UpgradeCard {

   private Map<Gem, Integer> cost;

   public Card(Map<Gem, Integer> cost) {
      this.cost = cost;
   }

   public Map<Gem, Integer> cost() {
      return this.cost;
   }

}
//Runner.java
public class Runner {
   public static void main(String[]) {
      //This is just for your convenience, it is not
      //part of the assignment.
   }
}
```