LAB 2 (part B)

Use the **`lab2`** project you created in Part A for your work here. The framework code is available from moodle. Use packages, because you'll be making a bunch of classes.

Lab 2 is due on Monday, March 16th, before the lab on that day.

1. Aliens

This problem is taken from the book: programming project 7.3 on page 477 (in the 5th edition), and is reproduced in the shaded box below. There are additional requirements below the reproduction.

> The following is some code designed by J. Hacker for a video game. There is an **`Alien`** class to represent a monster and an **`AlienPack`** class that represents a band of aliens and how much damage they can inflict:
>
> ```
> <<Alien and AlienPack class definitions in the framework code>>
> ```
>
> The code is not very object oriented and does not support information hiding in the **`Alien`** class. Rewrite the code so that inheritance is used to represent the different types of aliens instead of the "type" parameter. This should result in deletion of the "type" parameter. Also rewrite the **`Alien`** class to hide the instance variables and create a **`getDamage`** method for each derived class that returns the amount of damage the alien inflicts. Finally, rewrite the **`calculateDamage`** method to use **`getDamage`** and write a **`main`** method that tests the code (*changed below: use an **`AlienTester`** class)*.

Additionally:

- You will need to add a **`getDamage`** method to the **`Alien`** class as well as to the classes you derive from **`Alien`**.
- The **`AlienPack`** class contains an array to store the **`Alien`** objects. Rewrite this to use an **`ArrayList<Alien>`** . You should also
  - change the constructor, so that it doesn't require an initial size parameter
  - add an **`addAlien`** method that doesn't require an index parameter
  - make the **`getAliens`** method work with the original return value of **`Alien[]`**, using the **`ArrayList.toArray`** method. In order to make this work, you need to pass an argument of the type **`Alien[]`** into to **`toArray`** call. Typically, programmers make an array as the same length of the **`ArrayList`**:

```
Alien[] temp = new Alien[aliens.size()];
return aliens.toArray(temp);
```

- Your rewritten **`AlienPack`** class uses a *has-a* relationship with the **`ArrayList`** class: it uses a reference to an **`ArrayList`** object. Make a new class **`AlienPack2`** which exactly reproduces the functionality of the rewritten **`AlienPack`** but uses an *is-a*

relationship with the **`ArrayList`** class. That is, **`AlienPack2`** should **`extend ArrayList<Alien>`**.

- Put your testing code into a **`AlienTester`** class, making sure to test both the **`AlienPack`** and **`AlienPack2`** classses as well as your new subclasses of **`Alien`**.
- In a comment in the **`AlienTester`** class, describe which of the **`AlienPack`**/**`AlienPack2`** classes you prefer. Which was easier to write? Which would be easier for some else to use? Which would be safer for someone else to use?

2. The prisoner's dilemma

The *Prisoner's Dilemma* is a famous two-player, turn-based game. On each turn, each player says whether or not s/he will *cooperate* with or *compete* with the opponent.

- If both players cooperate, they each receive 3 points.
- If both players compete, they each receive 1 point.
- If one player cooperates and the other competes, the successful competitor will receive 5 points and the unsuccessful cooperator none.

This process is repeated a specified number of turns. The object of the game is to earn as many points as possible.

It should be obvious that, in a single turn, it's in a player's interest to compete rather than cooperate. Over the course of the entire game, however, players can observe and react to each other's choices; thus the incentive to compete can be overcome by the greater potential payoff for cooperation.

In this problem you will code players with different types of strategies and have the players play against each other. In the framework code, a **`Player`** class implements a player that always cooperates. A **`Simulation`** class provides code to play a game with two players and see the scores that result. Inspect and run the **`Simulation`** class, and confirm that each player earns 3 points on each turn because they cooperate every time.

Provide the following:

- Add a class named **`Competitor`** that extends **`Player`**. A **`Competitor`** always competes, never cooperates. Take advantage of methods provided in the **`Player`** class as much as possible.
- Add a class named **`RandomChooser`** that extends **`Player`**. A **`RandomChooser`** chooses randomly between cooperating and competing.
- Add a class named **`Unforgiving`** that extends **`Player`**. An **`Unforgiving`** player will cooperate on every turn in a given game until its opponent competes. After that, the **`Unforgiving`** player will also compete for the rest of the game.
- Add a class named **`TitForTat`** that extends **`Player`**. A **`TitForTat`** player will cooperate on the first turn, then on every subsequent turn it does the same thing that its opponent did on the previous turn.

Test your classes in **`Simulation.main()`**. In a comment in the **`Simulation`** class, answer:

- Which kind of player wins against the most other kinds of players?
- Which two kinds of players score the highest together?
- Which kind of player scores the highest against the range of other player types?
- Describe a new kind of player (that could be coded) that would play better, in some way, than any of the 5 players you've defined. Just describe the player, don't code it!