Task 1

(ii)

(a)

Advantages of using abstract classes:

We’ll be able to reduce the code redundancy by abstracting common code into the abstract class.

Disadvantages of using abstract classes:

It’s hard to handle the situation like this: some of these specific animals have a method called “fly()” while others don’t, it would somehow cause code redundancy.

(b)

By using interfaces instead of abstract classes, we can easily handle that situation above. We first define a interface with an abstract method “fly()”, and then we let classes which has the method “fly()” to implement this interface and override this method. In that case, we can reduce code redundancy as well as limit the format of specific methods.

Task 2

(i)

I define a CardDeckTest class which extends CardDeck and implements Runnable. When I set two threads to call dealCard() concurrently, I actually get more than 52 cards and showed repeated results.

(ii)

In single-threaded scenarios, there will not be anty exceptions been thrown.

In multi-threaded scenarios, dealCard() method could throw ArrayIndexOutOfBoundException when its not thread-safe, because if one thread has already generated a cardChoiceIndex which equals to cardList’s when another thread occupy the CPU, it would reach the area beyond the range of cardList.

(iii)

I defined a ThreadSafeCardDeck class with synchronized code to ensure its thread safe, you can see the result by running main method of ThreadSafeCardDeckTest.

Task 3

Part3(a)

(i)

There are two types of game called card game and die game respectively. In card game, the player has two chances to choose a card in the card set and the choices are generated randomly, the card would be removed from the card set once it is chosen. If player’s chosen cards contain at least an Ace, he/she wins. Otherwise, he/she loses.

In die game, the player has two chances to roll a number and the choices are generated randomly. If If player’s rolled numbers contain number “1”, he/she wins. Otherwise, he/she loses.

(ii)

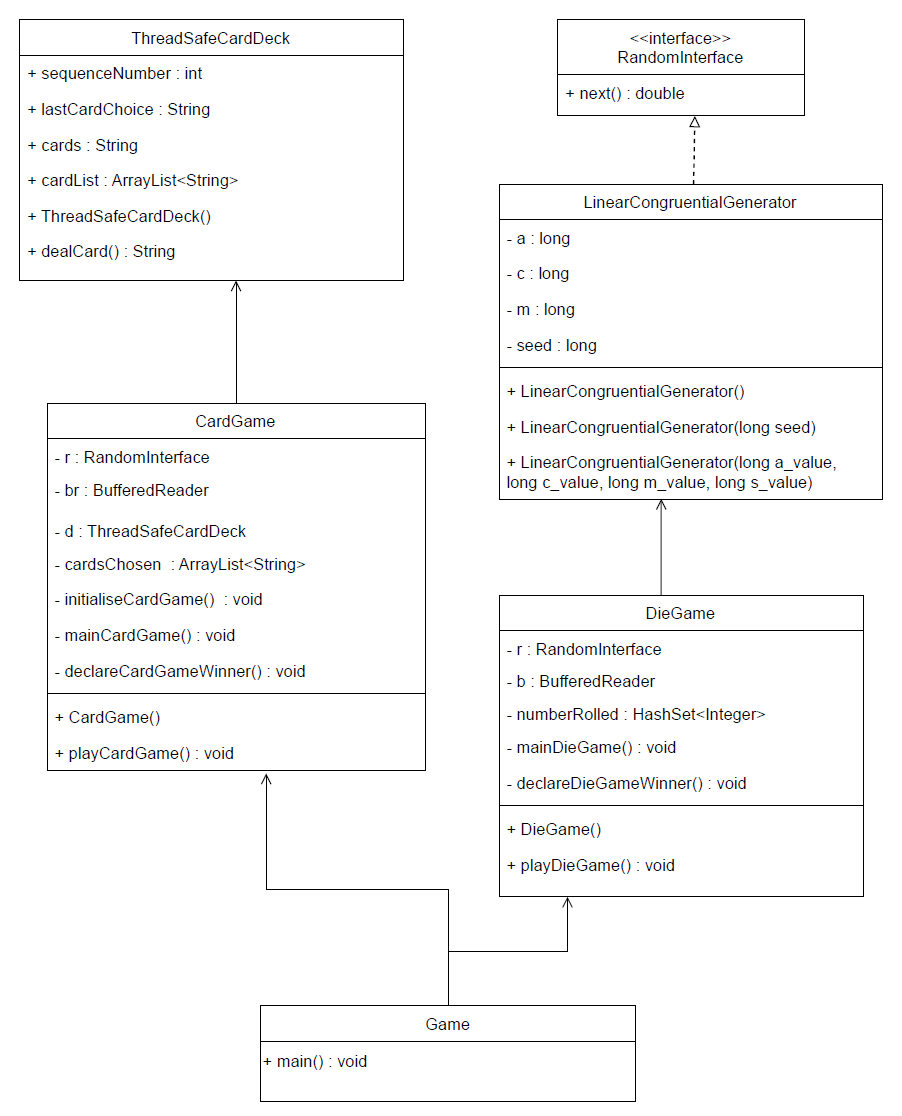
The cohesion of this project is no enough, in the Game class, there are two separate parts which does not have any relation to each other. And the coupling of this project is rather high, because there are too many public static methods so we can easily call any of these methods at outside, which do not make sense. Therefore there are not enough classes, we should at least separate two games into two classes. And the players are not really playing the game because they don’t have any choice since they are made randomly.

Part3(b)

(i)

There are five classes and an interface in my improved version of the program, including CardGame, DieGame, ThreadSafeCardDeck, Game, LinearCongruentialGenerator and RandomInterface. CardGame defines how the card game goes. DieGame defines how the die game goes. ThreadSafeCardDeck is used to implement card selection actions. LinearCongruentialGenerator implements RandomInterface to generate a random number. And Game is a test class which only contains a main method to run the game.

(ii)



(iii)

I make all of the variables and most of the methods private, so that they cannot be called out of the class. And I add a Game class which works as a test class.

Task 4

(i)

There are five philosophers and five forks and each fork is between two philosophers, vice-versa. Each philosopher has to pick up the fork on the left first and then the one on the right before he can start dining. Once a fork is picked up, it will be tagged as “inUse” so that no one else could pick it up again. If the fork a philosopher is trying to reach is in use, the philosopher could only wait until it’s released by someone else.

(ii)

As has been described above, the philosopher could do nothing but wait if the fork he’s trying to reach has been occupied. And all of them follows the protocol “first left and second right”. So if each one of them has occupied the fork on their left, liveness hazard happens, they have to wait forever.

(iii)

If we let one of the philosophers to pick up the fork on the right and then the left one, there would never be a situation like all of them has picked up the fork on the left and wait. So deadlock could never happens.

I don’t think the output would be incorrect at any situation.

(iv)

It may not be optimal or fair, as the philosopher who picks up the right fork first may have an advantage or disadvantage over the others, depending on the order of execution.

For example, suppose that philosopher 1 is the one who picks up the right fork first (fork 2) and then the left fork (fork 1), while the others pick up the left fork first and then the right fork. If philosopher 1 picks up fork 2 before any other philosopher picks up their left fork, then he can eat without waiting for anyone else. However, if philosopher 1 picks up fork 2 after all other philosophers have picked up their left forks, then he will have to wait until one of them finishes eating and releases both forks.

(v)

As has shown in the program, I designed a PhilosopherOptional class which contains a private static lock so that there would only be one philosopher can have the action “checking if both of the forks are available”. If it is, then pick up both forks, and if it’s not, then do nothing.