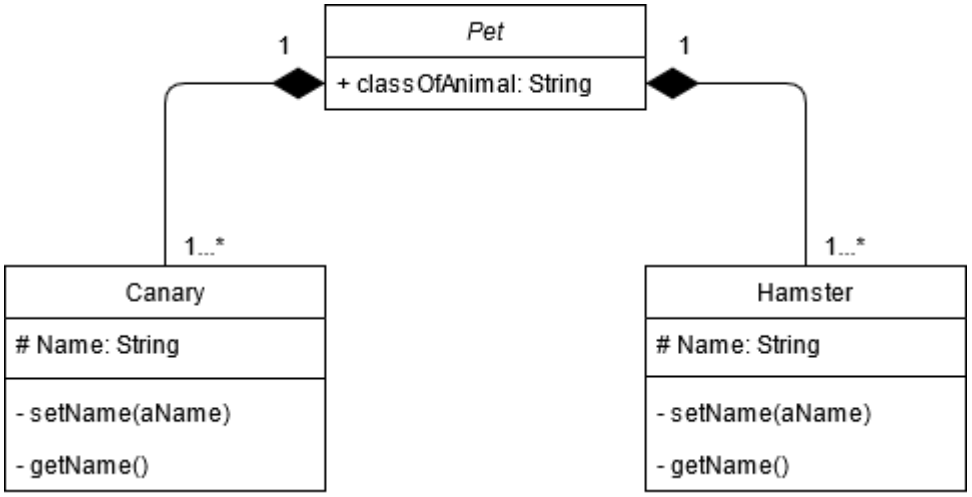
Object-Oriented Modelling and Programming Assignment

Task 1

i)



ii)

The first problem I noticed when refactoring the given code is the duplication of methods used in both Hamster.java and Canary.java. To solve this problem is by using inheritance, where I put methods that resided in both classes into the Pet.java class and generalised them so it returns the correct value depending on the class that is calling the methods. This is done, such that, when the commands “Pet p = new Canary(); p.setName(“Annabel”); “ are executed, and the Canary (or any other animal) contains “extends Pet” in the class definition, the method of “setName” will be used directly from the Pet class so Canary.java doesn’t need to contain it. This also means that Hamster.java doesn’t need to contain the duplicated methods also, it just needs to extend the Pet class.
Everything I have done is to make the subclasses inherit Pet.java’s methods to maximise inheritance. By containing the methods in just one superclass, it increases the reusability of the code and maximises efficiency for the program, as more subclasses can be added quicker (no need to duplicate methods with each class and customise the methods depending on the class). It also means that the program is much more concise, making the code more readable and making debugging a lot easier.

```
public class Canary extends Pet {
    protected String name;

}
```

```
public class Hamster extends Pet implements Vegetarian {
    protected String name;
```

```
public class Client1 extends Pet {

    public static void main(String[] args) {
        Hamster h=new Hamster();
        Pet p = h;
        Vegetarian v = h;
        h.setName("Cookie");
        System.out.println(p.getName() + " eats " + v.food());
    }

}
```

```java
public  class Pet {
    String name;
    public Pet() {}

    public String classOfAnimal() { return("Pet"); }
    public void setName(String aName) { name = aName; }
    public String getName() { return name; }

}
```

iii)

When defining the Hamster.java class, I decided to make it polymorphic by extending from Pet.java and implementing Vegetarian.java. This makes Hamster.java polymorphic because it inherits from multiple objects: Pet.java and Vegetarian.java. By Vegetarian.java being an interface and by using “implements Vegetarian” in the Hamster.java definition, it means that Hamster.java is able to no only inherit the methods and attributes, but also override them using “@Override” to customise the methods. This means that when the commands in Client1.java are executed, The Hamster can be initialised, then it can be defined as a Vegetarian due to the Vegetarian interface, then when calling the vegetarian method of food(), the Hamster.java overridden method will be used, so it can retrieve the food that the Hamster eats.
This decouples the class from the implementation code and doesn’t make all the classes hard-coded, which gives the client class sufficient knowledge to execute the commands. This means the polymorphic behaviour loosely couples the program which makes it more efficient than it would be without the polymorphic behaviour (being hard-coded)

```java
public class Client1 extends Pet {

    public static void main(String[] args) {
        Hamster h=new Hamster();
        Pet p = h;
        Vegetarian v = h;
        h.setName("Cookie");
        System.out.println(p.getName() + " eats " + v.food());
    }


}
```

```java
public interface Vegetarian {
    public String food();
}
```

```java
public class Hamster extends Pet implements Vegetarian {
    protected String name;


    @Override
    public String food() { return("beans"); }
}
```

Task 2

i)

The Singleton design pattern's purpose is to solve two problems violating the Single Responsibility Principle by making sure each class only has one instance and creating a global access point to the single instance. This means that some object can be accessed from anywhere within the program, but it cannot be edited/overwritten by other parts of the program.

ii)

```
public class ExampleTest {
public static void main(String[] args) {
ExampleSingleton s = ExampleSingleton.getInstance();
System.out.println("The ExampleSingleton has been "
+ "accessed via the getInstance() method "
+ s.accessCount()
+ " time(s)");
s = ExampleSingleton.getInstance();
System.out.println("The ExampleSingleton has been "
+ "accessed via the getInstance() method "
+ s.accessCount()
+ " time(s)");
}
}
```

```
public class ExampleSingleton{
    private static int accessCount = 0;
    private static ExampleSingleton singletonInstance = new ExampleSingleton();


    private ExampleSingleton() { System.out.println("I, the ExampleSingleton, am being created"); }

    public static ExampleSingleton getInstance(){
        System.out.println("The sole instance of ExampleSingleton is being retrieved");
        accessCount++;
        return singletonInstance;
        }

    public int accessCount() { return accessCount; }

}
```

Task 3

i)

The Adapter design pattern's purpose is to allow objects incompatible interfaces to interact with each other and convert a class' interface into a different interface. In this context, BookAdapter makes IncompatibleBooks's methods compatible with Book such that, when the methods of Book is called in BookAdapter, the relevant methods in IncompatibleBook can be used.

ii)

```
public class IncompatibleBook {
    public String titleString = new String();

    public void setTitle(String aString) { titleString = aString; }
    public String getTitle() { return titleString; }
}
```

iii)

```
public class Test {
    public static void main(String[] theArguments) {
        Book b=new BookAdapter();
        b.setTitleString("Gone with the Wind");
        System.out.println("We have a book with title: " + b.getTitleString());
    }
}
```

```
public class BookAdapter extends Book {
    IncompatibleBook incompatibleBook = new IncompatibleBook();

    @Override
    public void setTitleString(String aString) { incompatibleBook.setTitle(aString); }

    @Override
    public String getTitleString() { return incompatibleBook.getTitle(); }
}
```

```
public abstract class Book {
public abstract void setTitleString(String aString);
public abstract String getTitleString();
        }
```

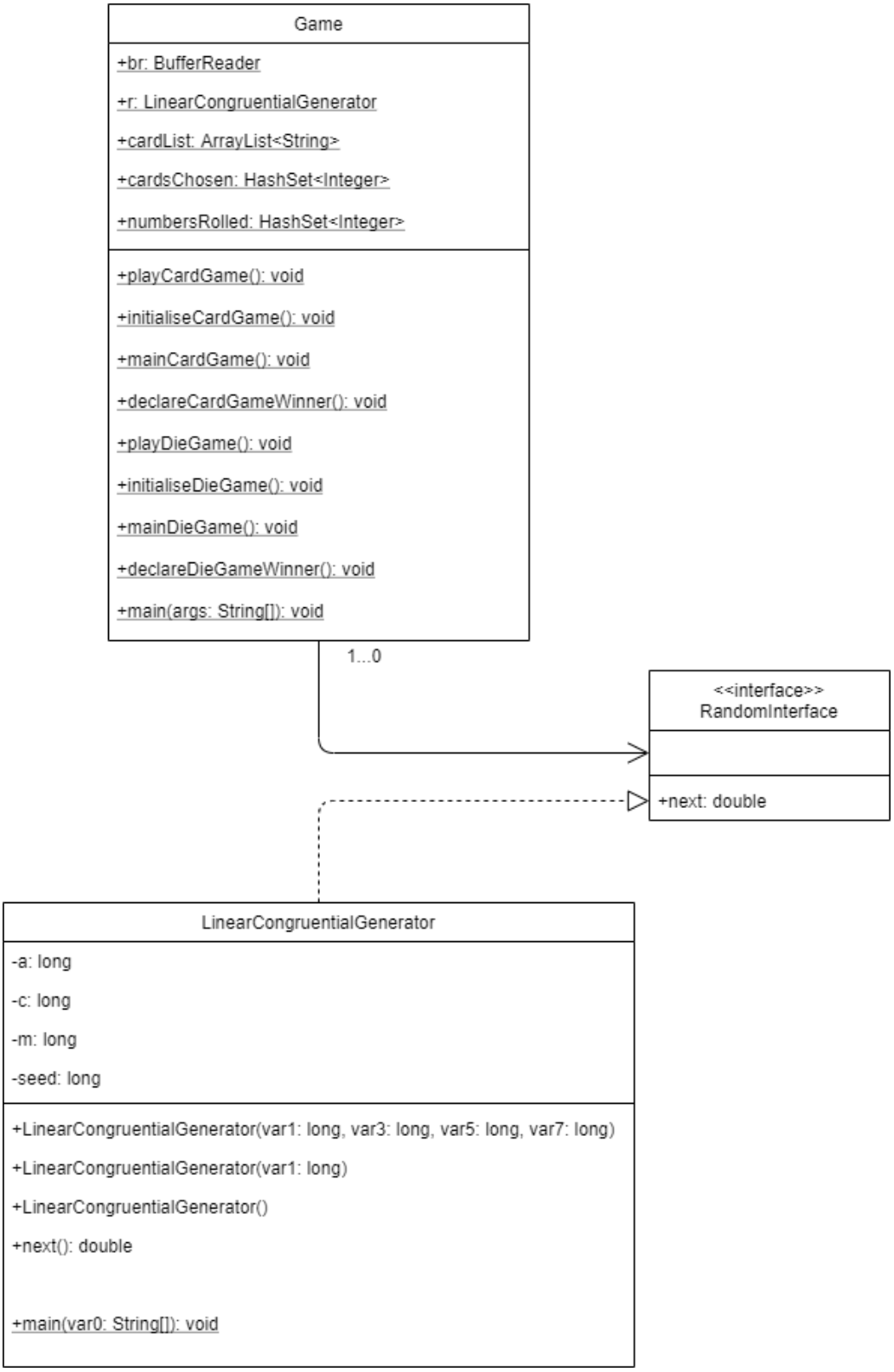
Task 4

i)

```
public class LinearCongruentialGenerator implements RandomInterface  {
// Generates pseudo-random numbers using:
// X(n+1) = (aX(n) + c) (mod m)
// for suitable a, c and m. The numbers are "normalised" to the range
// [0, 1) by computing X(n+1) / m.

  private long a, c, m, seed;
// Need to be long in order to hold typical values ...

  public LinearCongruentialGenerator(long a_value, long c_value, long m_value, long s_value) {
    a=a_value; c=c_value; m=m_value; seed=s_value;
  }

  public LinearCongruentialGenerator(long seed) {
  // Set a, c and m to values suggested in Press, Teukolsky, et al., "Numberical Recipies"
    this( a_value: 1664525,  c_value: 1013904223,  m_value: 4294967296l, seed);
  // NB "l" on the end is the way that a long integer can be specified. The
  // smaller ones are type-cast silently to longs, but the large number is too
  // big to fit into an ordinary int, so needs to be defined explicitly
  }

  public LinearCongruentialGenerator() {
  // (Re-)set seed to an arbitrary value, having first constructed the object using
  // zero as the seed. The point is that we don't know what m is until after it has
  // been initialised.

  this( seed: 0);  seed=System.currentTimeMillis() % m;

  }

  public static void main(String args[]) {
  // Just a little bit of test code, to illustrate use of this class.
    RandomInterface r=new LinearCongruentialGenerator();
    for (int i=0; i<10; i++) System.out.println(r.next());
```

```
    // Since RandomInterface doesn't know about the instance variables defined in this
    // particular implementation, LinearCongruentialGenerator, we need to type-cast
    // in order to print out the parameters (primarily for "debugging" purposes).

      LinearCongruentialGenerator temp=(LinearCongruentialGenerator) r;
      System.out.println("a: " + temp.a + "  c: " + temp.c + "  m: " + temp.m + "  seed: " + temp.seed);

    }


    public double next() {
      seed = (a * seed + c) % m;
      return (double) seed/m;
    }
  }
```

ii)


Game
+br: BufferReader
+r: LinearCongruentialGenerator
+cardList: ArrayList<String>
+cardsChosen: HashSet<Integer>
+numbersRolled: HashSet<Integer>
+playCardGame(): void
+initialiseCardGame(): void
+mainCardGame(): void
+declareCardGameWinner(): void
+playDieGame(): void
+initialiseDieGame(): void
+mainDieGame(): void
+declareDieGameWinner(): void
+main(args: String[]): void
1...0
<<interface>>
RandomInterface
+next: double
LinearCongruentialGenerator
-a: long
-c: long
-m: long
-seed: long
+LinearCongruentialGenerator(var1: long, var3: long, var5: long, var7: long)
+LinearCongruentialGenerator(var1: long)
+LinearCongruentialGenerator()
+next(): double
+main(var0: String[]): void

iii)

The purpose of this program is to let the user choose between two games, and execute the chosen game. The games available are a card game and a die game, where the card game is where you randomly select two cards and if one of the cards is an ace, you win, the die game is you roll a die twice and if one of the rolls is the number one, you win.
Game.java has a method to initialise the game, play the game and declare the winner for both the card and the die game. To get the random cards/numbers the program uses LinearCongruentialGenerator.java. This generates a random number/card and implements RandomInterface.java to use its next() method to get its next seed. LinearCongruentialGenerator.java generates pseudo-random numbers by normalising values within a custom range (from 1-6 for the die game and a list of cards for the card game). This makes a random seed that will create the random number which can be used to select a number between 1 and 6 or use to select from a list of cards.

Task 5

i)

When looking at Task4 and the way it was designed, I could see it was very poor and contained a lot of bad code smells, so I have decided to use a design pattern to make the code more flexible, more maintainable and more reusable.

The design pattern I have decided to use is the Factory pattern, this is because the code requires the user to choose a game, and run the game based on their input - and the Factory design pattern is an extremely good way to create an object (in this case, the object is the card/die game).

I have decided to split up the different games (card and die) into two separate subclasses, implementing methods from a new interface I have decided to create called GameInterface. In Game.java currently, the card game and the die game both use an Initialisation method, a play method, a main method and a declare winner method. My interface is designed such that both the die game class and the card game class can implement the methods play(), main(), and declareWinner(). Using an interface, the code will interact only with the resultant interface, meaning that the program will become loosely-coupled, and means that any additional game that can be created in the future can be easily implemented by creating its own game class which also implements the game interface. This makes the code more maintainable and reusable.

By creating separate classes for each game, I am able to maximise cohesion within the program. Through the files received in task 4, all of the methods to play the game were in one big Game class, containing lots of processes that are not similar to each other - reducing the reusability of the program, but as the methods are now in an

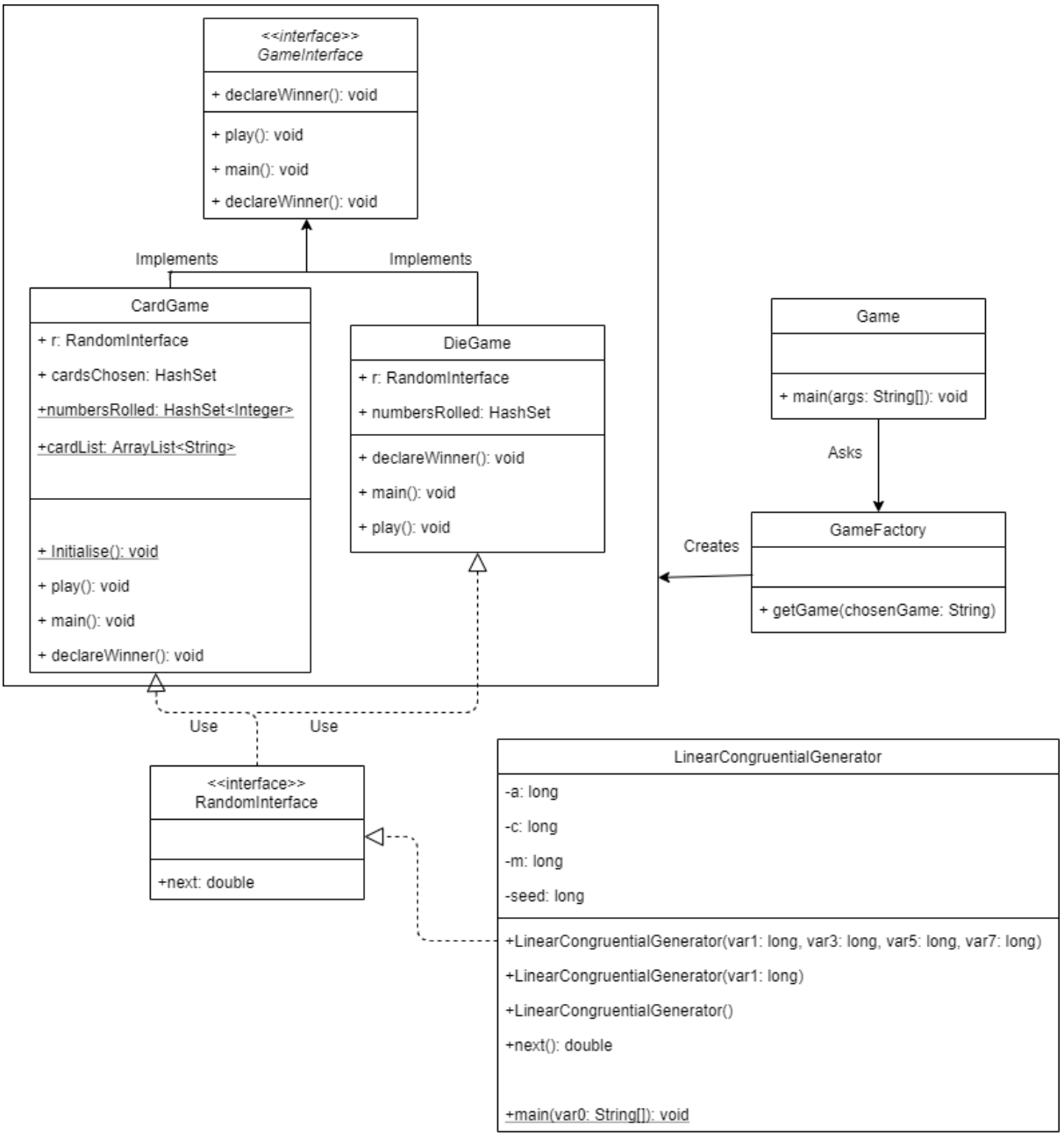
interface, where each class can use them and the can be separately defined within the sub-classes, it means that all methods that are similar are in the same part of the program, increasing cohesion and making sure there are more smaller files that are executing a specific job. The reason that initialise() will not be in the game interface is because it is a local method within the card game class. This is because not all games need initialising, and this would be a bad code smell (Refused Request) to have classes inherit a method it doesn't need.
Below are screenshots of the Game Factory, the Game Interface and Game.java to show How I use the Game Factory to create either a Die Game or Card Game object of their class.

```java
public class GameFactory {
    // This is a Game Factory that generates an object of a concrete class
    //It does this depending on what game the user selects in Game.java

    public GameInterface getGame(String chosenGame){

        //Generate a new Card Game if the user types 'c'
        if (chosenGame.equals("c")) {
            return new CardGame();
        }

        //Generate a new Die Game if the user types 'd'
        else if (chosenGame.equals("d")) {
            return new DieGame();
        }

        else {
            System.out.println("Input not understood");
            return null;
        }
    }
}
```

```java
public interface GameInterface {
  // An interface to define all the methods a every Game needs
  void play() throws Exception;
  //Initialise() isn't in this interface as DieGame does not need to be initialised,
  //Using it would be a bad code smell.
  void main() throws Exception;
  void declareWinner() throws Exception;
```

```java
import java.io.*;

public class Game {
        public static void main(String[] args) throws Exception {
            // The BufferedReader used to ask which game the user would like to play
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

            //Initialise the Game Factory
            GameFactory GameFactory = new GameFactory();

            //Ask the user which game to run and store their answer
            System.out.print("Card (c) or Die (d) game? ");
            String ans=br.readLine();

            //Get an object of whichever game the user chooses
            GameInterface chosenGame = GameFactory.getGame(ans);

            // Call the play() method of the chosen game to run the game.
            chosenGame.play();

        }
}
```

ii)



ii)

As the files are large or there are lots of files, I am unable to screenshot all of the files and show them here, please look at my zip file for proof for Task 5 part iii.

Task 6

i)

TestThread.java is not thread-safe because it can't be called simultaneously by multiple threads and still give the correct result. The lack of synchronisation in the program leads to concerning unpredictability of the result - due to threads overwriting the value that other threads are using, thus it ends up not being what it should be.

iii)

One of the ways I have made the code thread-safe is by making the methods in the Example class "synchronized". By using the "synchronized" command, the whole method that it is used on becomes a synchronized block of which locks the object the method is referring to, making the object safe to the thread currently in execution.

Another way I have made the code thread-safe is by making all variables in the Example class "volatile". After using the synchronisation on methods (explained above) it is possible that the CPU caches the values within the class, which means there is a chance that a thread may access the cache and overwrite a value that another thread is using - causing unexpected results. Using "volatile" when defining values within the class means the program will read from the machine's main memory, rather than the cache.

I have also initialised myInstance safely first before it is accessed by any part of the program. This means that it will be thread safe as nothing can overwrite it before it has initialised itself.

```
public class TestThread extends Thread {
private int theValue;
public TestThread(int aValue) { theValue=aValue; }

public void run() {
    Example e = Example.getInstance();
    e.setVal(theValue);
    }
public static void main(String[] args) throws java.lang.InterruptedException {
    TestThread[] tests=new TestThread[10000];
    for (int i=0; i<10000; i++) { tests[i]=new TestThread(i); }
    for (int i=0; i<10000; i++) { tests[i].start(); }
    for (int i=0; i<10000; i++) { tests[i].join(); }
    Example e = Example.getInstance();
    System.out.println("The Example has value " + e.getVal() + " and has been updated " +
            e.getUpdateCount() + " time(s)");
    }
}
```

```
class Example {
private volatile static Example myInstance = new Example();
private volatile int updateCount=0;
private volatile int val=0;

private Example() { }
public static Example getInstance() {
    if (myInstance == null) {myInstance = new Example();}
    return myInstance;
}
public synchronized void setVal(int aVal) { val=aVal; updateCount++; }
public synchronized int getVal() { return val; }
public synchronized int getUpdateCount() { return updateCount; }
}
```

Task 7

i)

The “synchronized” associated with the rightfork/leftfork is a synchronised statement. A synchronised statement takes in what object it needs synchronised and wraps it in a synchronised block. In this case, the objects are the right/left forks. By using synchronised statements, it means that each thread that runs in the program should only interfere with the object when it is supposed to, by using an intrinsic lock on the object. This is less recourse-demanding than using synchronised methods (my answer to task 6), which is why it is better for this program, as we are sure which objects specifically should be thread-safe.

ii)

Deadlock is when a running program is halted due to different parts of the program waiting for something to continue processing, but another part of the program is holding what it needs to continue. This occurs after all the philosopher start to think and they reach to pick up their left forks at the same time. Once they have all picked up their left forks, there are no right forks to pick up as their neighbour has it as their left fork. This creates a deadlock because the philosophers will be waiting for an infinite amount of time for a right fork to appear - which it won’t.

iii)

To solve this issue, I simply made the first philosopher to pick up a fork choose the right fork first. I have chosen to do this because then all the other philosophers won’t be able to all pick up the left fork first without a right fork at hand, hence resolving the deadlock and means the program will run smoothly without halting.

```java
public class DiningPhilosophers {

    public static void main(String[] args) throws Exception {

        Philosopher[] philosophers = new Philosopher[5];
        Object[] forks = new Object[philosophers.length];

        for (int i = 0; i < forks.length; i++) {
            forks[i] = new Object();
        }

        for (int i = 0; i < philosophers.length; i++) {
            Object leftFork = forks[i];
            Object rightFork = forks[(i + 1) % forks.length];
            if(i == 0) {
                philosophers[i] = new Philosopher(rightFork, leftFork );
            }
            else{
                philosophers[i] = new Philosopher(leftFork, rightFork);
            }


            Thread t = new Thread(philosophers[i],  name: "Philosopher " + (i + 1));
            t.start();
        }
    }
}
```