CM2307 - Object-Oriented Modelling and Programming Assignment

Student Name: Yash Shah

Student Number: 2078614

Note: All outputs of the tasks are at the bottom

Object-Oriented Modelling and Programming Assignment

Task 1

Part i) The first step to refactor the code was to remove the attribute “name” and the methods “getName()” and “setName()” from the ZebraFinch class and Chinchilla class. Secondly the removed attribute and methods were added to the Pet class. Since all pets usually share the same characteristics ZebraFinch and Chinchilla are subclasses of Pet (they extend Pet). Since they are a subclass of Pet it means they inherit all the attributes and methods of the Pet class which is used as a template. Therefore, there is no need have the getters and setter in the ZebraFinch and Chinchilla classes (sub classes). Lastly in the main method (class Client) a pet object is created and is used to set the name of a Chinchilla and then print out the Chinchilla’s name. Then Polymorphism is applied and the variable is reused to create a ZebraFinch object. The “setName()” method is called for the ZebraFinch and the name of the ZebraFinch is then printed out.

Original Pet

```
public abstract class Pet {
  public abstract String classOfAnimal();
}
```

Modified Pet

```
// This class sets up the common attributes and methods of all Pets
public abstract class Pet {
  private String name;

  public abstract String classOfAnimal();

  public void setName(String petName) {
    name = petName;
  }

  public String getName() {
    return name;
  }
}
```

Original Chinchilla

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

  public void setName(String aName) { name=aName; }

  public String getName() { return name; }

  public String classOfAnimal() { return("Chinchilla"); }
}
```

Modified Chinchilla

```
// Inherits Pet attributes and methods
public class Chinchilla extends Pet {

  public String classOfAnimal() { return("Chinchilla"); }
}
```

Original ZebraFinch

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

  public void setName(String aName) { name=aName; }

  public String getName() { return name; }

  public String classOfAnimal() { return("ZebraFinch"); }
}
```

Modified ZebraFinch

```
// Inherits Pet attributes and methods
public class ZebraFinch extends Pet {

  public String classOfAnimal() { return("ZebraFinch"); }
}
```

Original Client

```
public class Client {
  public static void main(String[] args) {
    Chinchilla c = new Chinchilla();
    c.setName("Grumpy");
    System.out.println("The " + c.classOfAnimal() + "'s name is " + c.getName());

    ZebraFinch z = new ZebraFinch();
    z.setName("Happy");
    System.out.println("The " + z.classOfAnimal() + "'s name is " + z.getName());
  }
}
```

Modified Client

```
public class Client {
  public static void main(String[] args) {
    Chinchilla c = new Chinchilla();
    c.setName("Grumpy");
    System.out.println("The " + c.classOfAnimal() + "'s name is " + c.getName());

    ZebraFinch z = new ZebraFinch();
    z.setName("Happy");
    System.out.println("The " + z.classOfAnimal() + "'s name is " + z.getName());

    // My addition
    // Chinchilla has all the attributes and methods from Pet
    Pet p = new Chinchilla();
    p.setName("Grumpy");
    System.out.println("The " + p.classOfAnimal() + "'s name is " + p.getName());

    // Polymorphism has been performed on ZebraFinch
    // ZebraFinch also has attributes of Pet
    p = new ZebraFinch();
    p.setName("Happy");
    System.out.println("The " + p.classOfAnimal() + "'s name is " + p.getName());
  }
}
```

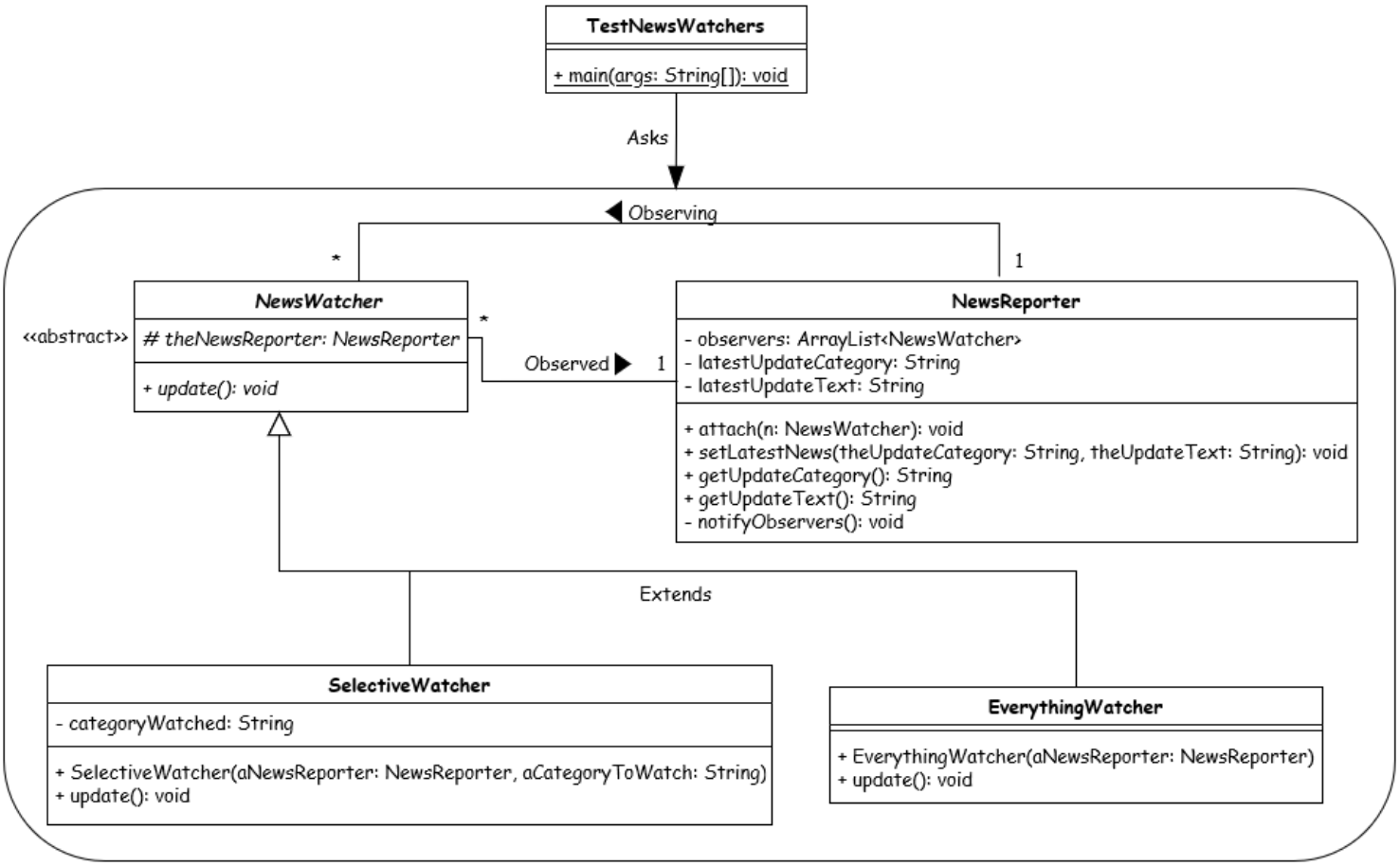
Part ii) The abstract class in the code provides a template for other classes to extend which is a good way of reducing code duplication as any class that extends the super class inherits the super classes attributes and methods. So, for the example the ZebraFinch and Chinchilla do not have to implement a getter and setter for the name as it has already been inherited from the Pet class. Code complexity can be hidden from the end user and only the necessary functions are shown to the user. In the code the user would not know how the method "setName()" and "getName()" works internally but the user is aware the method exists and how to use it. In addition, abstract classes enable code reusability in which many objects can use the same method but overwrite the method for the specific objects needs. In the code changes can be made to the internal code without affecting the other classes or concrete methods can be added which can be used by the classes in the future. For example an age() method can be added to the Pet class which does not have to be used but can be used in the future by one of the classes. A class can only extend one abstract class at a time. Since Chinchilla extends Pet, it cannot extend another abstract class.

An interface only stores the method signature and not the method definition which also improves readability. If Pet was an interface, it would only contain methods with no implementation details and no attributes. Any class that would then implement Pets would need to implement and override the methods created in Pet. Having only the method signature makes an interface suitable for complete abstraction by hiding the method implementation from the user. Classes can implement multiple interfaces meaning a class is able to inherit methods from different interfaces. Different classes can implement one interface and override a method to meet the specific requirements of the class. For example, ZebraFinch and Chinchilla can implement Pet if it was an interface and override the method with their own internal class specific details. Interfaces are loosely coupled meaning method definitions do not depend on other methods or classes. Since interfaces cannot have implementation details in a method getters and setter are not possible in an interface as well as declaring attributes. Any class implementing an interface must implement its methods as

well even if it is not being used. For example, if a hundred classes are implementing an interface and a new method is added to the interface then all hundred classes will be affected and will need to implement that new method which could take up a lot of time.

Task 2

Part ii)



Observer is a behavioural design pattern that consists of two objects a subject and an observer. The subject object maintains a list of its dependents which are called observers, and notifies them automatically of any state changes, by calling one of their methods.

For the news system as can be seen from the UML diagram above the NewsReporter class is the subject and it maintains a list of observers which are called NewsWatcher and also contains its own attributes and methods. The NewsWatcher class is an abstract class that contains an attribute which stores a news reporter and an abstract method which is to be extended and overwritten by any watchers that extend the NewsWatcher class. The SelectiveWatcher and EverythingWatcher are subclasses of the NewsWatcher class therefore they overwrite the update() method that was created in the NewsWatcher class with their own implementations. The NewsReporter class contains a notifyObservers() method which loops through the list of observers and calls their update() method. The notifyObservers() is only called whenever there is a sate change or in this case whenever there is a news update. Any observers part of the EverythingWatcher class will be updated on every news event. Where as an observer that is part of the SelectiveWatcher class will only be updated with certain news events based on the category they have subscribed to. So e.g., if an observer has

subscribed to the business category, they will only get news updates that are business related and should not get updates from any other category.

Task 3

Part 3ai) The program that has been created is a game where the user is able to choose between two games a card game or a die game. The user is able to choose a game by entering "c" for the card game or "d" for the die game. Whichever game is chosen some attributes are initialised and their respective play game (playCardGame() or playDieGame()) methods are called. Each game has a play game method which further calls other method which initialise the games, runs the main game and lastly declares if the user playing the game won or lost the game. Since both games have a similar structure in terms of the methods used an interface would be suitable here. Additionally, a factory method would be suitable to implement as there are multiple subclasses of the game interface. **Potential classes that can be chosen are**: CardGame class, DieGame class, Game Interface, ChooseGame class, LinearCongruentialGenerator class, RandomInterface interface and Main class.

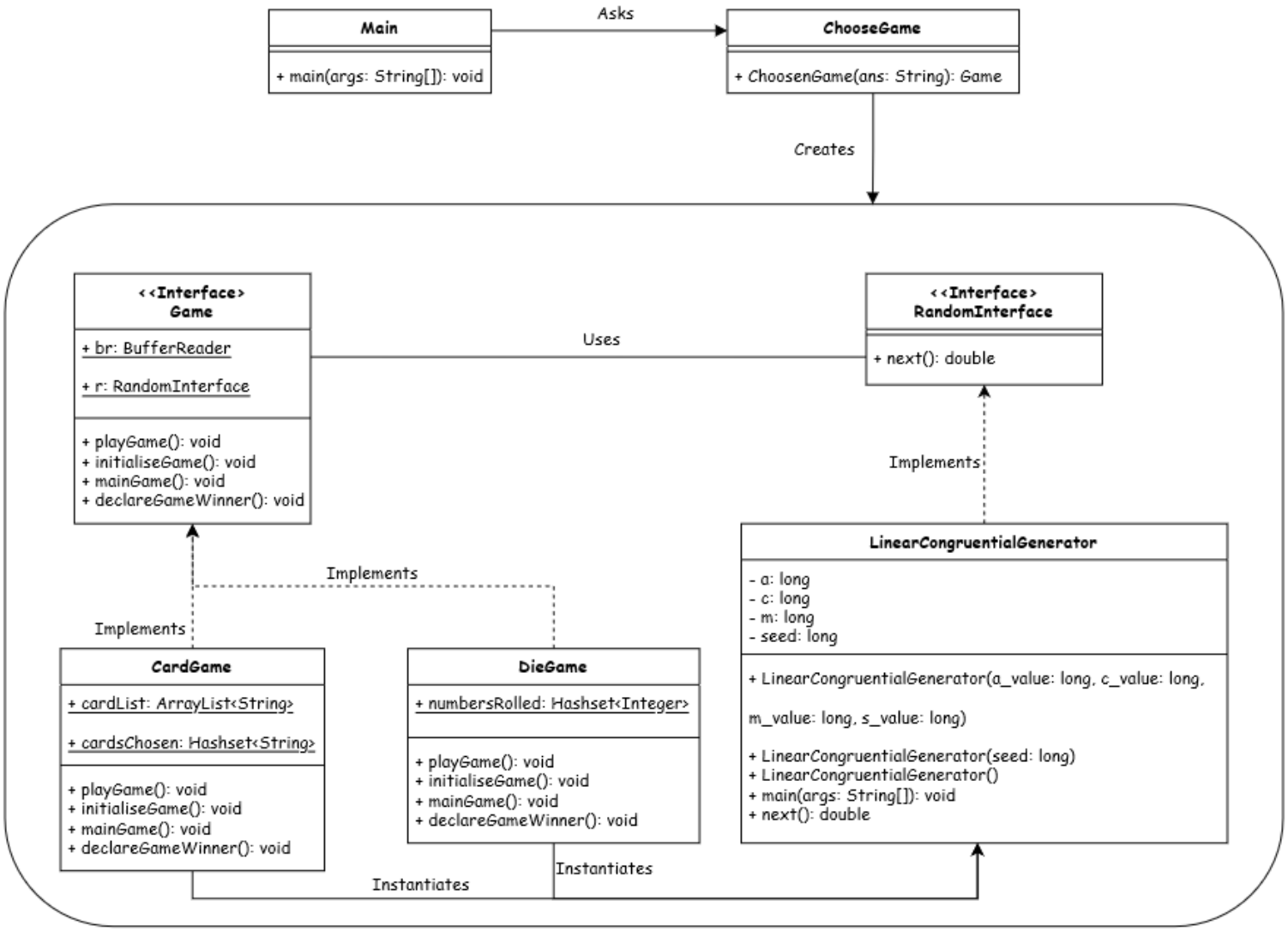
For example, if the user wants to play the card game the user will run the game program. In running the game program, a buffer reader is created as well as a random number generator which will be used for the game. When the program is running the user is prompted to choose a game so the user will enter "c". When the user enters "c" their input is read by the buffer reader and is stored in a variable ans which is then passed onto an if statement. The if condition is run and it sees that the user has entered "c" so it will then call the playCardGame() method. The playCardGame() method further calls the other methods initialiseCardGame(), mainCardGame() and declareCardGameWinner(). When the initialiseCardGame() method is called it creates a list of cards then randomly chooses two indexes from the card list and swaps those cards. The cards are shuffled a hundred times. Then the mainCardGame() method is run which prompts the user to hit the return key in doing so a random number is generated between 1 and 52 as there are 52 cards in the deck which ever number is generated the card at that index is removed from the card list and is returned as the users first card choice then the user is prompted to hit the return key again for another random number to be generated and another card to be returned as their second choice. Lastly the declareCardGameWinner() method is run which checks which cards the user has got and if any of the cards the user has is an ace then the user has won the game if not then the user has lost the game.

Part 3bi) For the new and improved design each game has been given its own class. So, the card game and die game have their own classes (**CardGame** class and **DieGame** class). Since each game has a similar structure an interface class has been created (**Game** interface). This interface is used as a blue print for the games therefore each game has to implement and overwrite its methods. Using an interface improves code readability as it hides code complexity from the user. In addition, a random number generator class has been included which implements a random Interface (**LinearCongruentialGenerator** class and **RandomInterface** interface). Lastly following the factory design pattern, a factory class has been created which creates and returns a game object based on the user's input (**ChooseGame** class). The reason for using a factory is that it deals with the problem of creating objects without having to specify the exact class of the object that will be created rather than having to call a constructor. Lastly a main method class has been created which is used to get user input and passing it to the factory method which creates the game object and runs one of the games based on the user's input (**Main** class).

The new design supports low coupling and high cohesion. It supports low coupling as any changes made to the inner structure of the card game does not affect any other module or game such as the die game and vice versa each module is independent. The code also supports high cohesion since each of the games have a similar structure an interface has been used to maintain this structure and keeps all the related method in one interface. Therefore, a developer can easily implement a new game as the structure will be kept the same for each game but each games internal structures will be different.

Part 3bii)

UML Diagram of Improved Program



Task 4

Part i) When running the code several times the output values were between 9997 and 1000, but in most runs the output returned 9999 for e.getVal() and e.getUpdateCount(). This code maybe be thread unsafe as it firstly needs to initialise the "myInstance" attribute. If there are multiple threads running then they could all end up running the getInstance() method at the same time which would end up creating multiple objects of the Example class, which violates the singleton concept. Since threads share memory space, they could each try overwrite a shared resource which would end up causing a race condition. Secondly since each thread is running concurrently, they all attempt to access the shared resource but as they try, they overlap each other. The overlap is caused by threads interfering with each other when the setVal() method is executed which is causing the output to be unpredictable. Lastly since the increment operation is not atomic it has to go through a three-step process of fetching, increment and reassigning the updated values back to the val and updateCount attributes. During this three-step process there is data being lost as different threads may be accessing a value one after another and incrementing it but since another thread has already incremented the values first the value will not change and therefore that is why the output is mostly short of the correct value.

```
public static Example getInstance() {
    if (myInstance == null) {myInstance = new Example();}
    return myInstance;
}
```

Part ii)

```
class Example {
    private static Example myInstance;
    private int updateCount = 0;
    private int val = 0;

    private Example() {
    }

    public synchronized static Example getInstance() {
        if (myInstance == null) {
            myInstance = new Example();
        }
        return myInstance;
    }

    public synchronized void setVal(int aVal) {
        val = aVal;
        updateCount++;
    }

    public int getVal() {
        return val;
    }

    public int getUpdateCount() {
        return updateCount;
    }
}
```

Part iii) To make the code tread safe the synchronized keyword has been used for the getInstance() method and the setVal() method. This prevents multiple treads from executing the getInstance() or setVal() method and only one thread is able to do so at a time. Once one thread finishes with exectuition is when another thread can start the execution. Having this

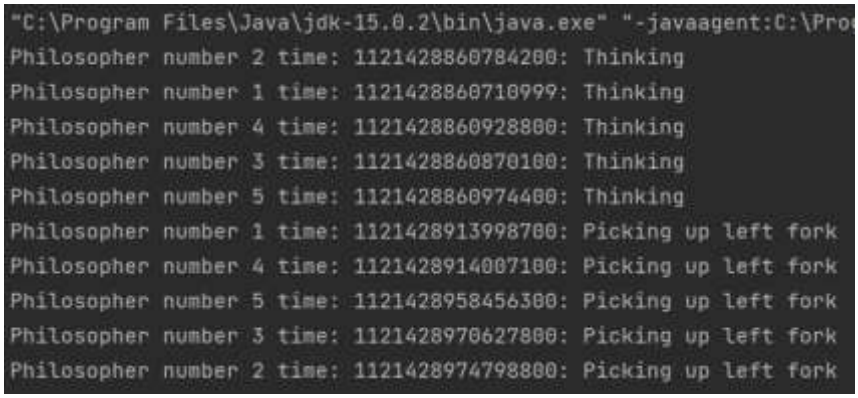
synchronisation prevents multiple objects from being created and also solves the issue of incrementing as only one thread can increment at a time.

Task 5

Part i) The synchronized keyword on the Fork object is used as a lock to guard the fork object meaning that only one thread can execute/access the object at a time. In the case of this program the synchronized keyword has been used on a block of code (image below) so that means only one thread can execute that block of code at a time. And all variable within this synchronized block are to be flushed and updated directly on the main memory. The volatile key word is used as an indicator for the Java compiler and the threads to not cache the value of the variable with the volatile keyword and to instead always read it from the main memory. In this case if the value of "inUse" were to be cached and another thread modifies its value then any thread using the cached value will have an incorrect output. The reason for the variable being superfluous is that it makes it easier to see which fork objects "inUse" value is being modified (left or right) without having to create an object of type Fork. Since this program has not been packaged and the "inUse" variable is public it can be accessed from any class and any application. If these variables were to be removed the program would still work normally.

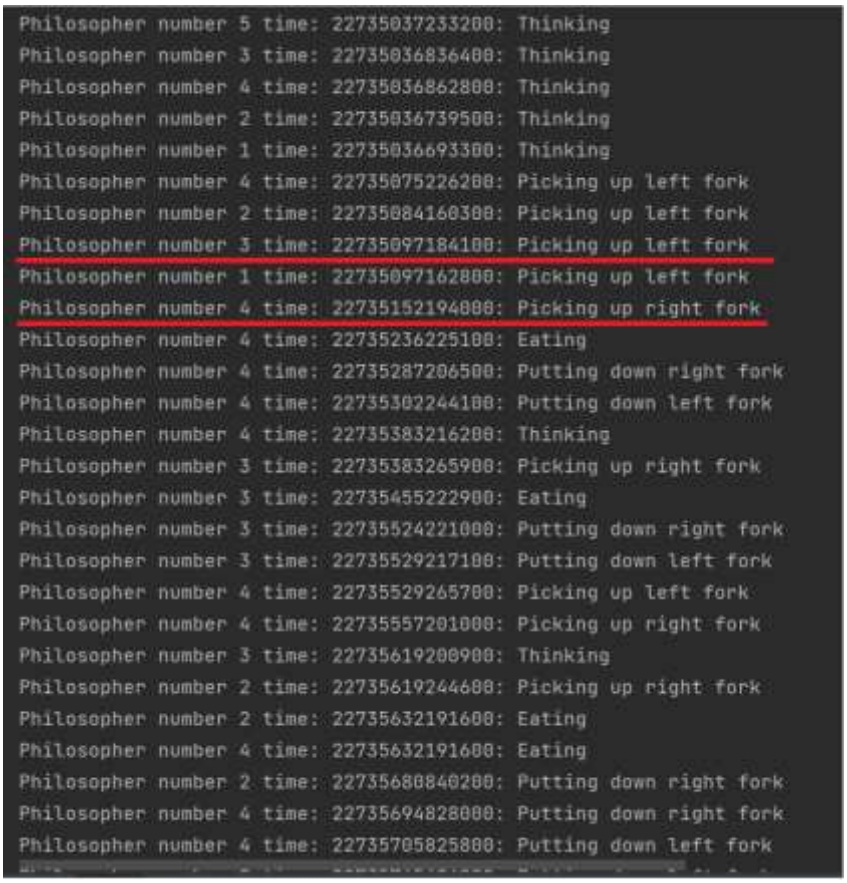
```
synchronized (leftFork) {
    leftFork.inUse=true;
    doAction("Picking up left fork");
    synchronized (rightFork) {
        // eating
        rightFork.inUse=true;
        doAction("Picking up right fork");
        doAction("Eating");
        doAction("Putting down right fork");
        rightFork.inUse=false;
    }
    // Back to thinking
    doAction("Putting down left fork");
    leftFork.inUse=false;
}
```

Part ii) A deadlock is a situation where the progress of a system is halted as each process is waiting to acquire a resource held by some other process. Deadlocks usually occur in situations when a thread is waiting for an object lock, that is acquired by another thread and the other thread is waiting for an object lock that is acquired by another thread. Since, all threads are waiting for each other to release the lock they reach a deadlock and will be in this state of waiting forever. In the situation of the Dining Philosophers a dead lock occurs when each of the Philosophers picks up their left forks meaning there is no right fork left as their neighbour has already acquired that fork. Therefore, this leads to the Philosophers going into an endless waiting state which is knowns as the deadlock as they will each wait until a right fork is available. A way to solve this maybe to assign the odd philosopher the left forks and the even philosopher the right forks in the start.



Part iii) The deadlock situation is occurring as all philosophers pick up their left forks. Meaning there are no right forks available so all philosophers go into an endless circular waiting state which is known as deadlock. Therefore, to break this circular waiting state one philosopher must pick up a right fork. Therefore, in the code modification I have made all philosophers will pick up their left forks except the last philosopher who will pick up their right fork essentially breaking the waiting cycle. An assumption is since the last philosopher goes for their right fork; it won't be available as philosopher 4 may have already picked it up. Since philosopher 5's right fork may not be available he/she may not eat and therefore still remain in a thinking state. Philosopher 5 not being able to eat has stopped the deadlock from occurring as now philosopher 1 can now start eating as philosopher 1's right fork is available since philosopher 5 went for their right fork first which was not available. Once philosopher 1 is done eating philosopher 2 can start eating. Once philosopher 2 is done eating philosopher 3 can start eating as well as philosopher 1 can start eating again. When philosopher 3 is done eating philosopher 4 can start eating and philosopher 2 can eating again when philosopher 1 is done eating. Lastly philosopher 5 can start eating once philosopher 4 is done eating. That is one cycle done and the process continues. During these cycles only two people will be able to eat at any given one time and no two neighbours will be able to eat at the same time. The issue with the first cycle is that philosopher 5 could end up starving if the rest of the philosopher end up taking too long to eat. From my testing with the modification, I have made it stops deadlocks from occurring but the outputs are completely random when run each time and are wrong as can be seen in the terminal screenshot below. Since philosopher 3 has picked up their left fork philosopher 4 should not have been able to pick up their right fork as it has already been picked by philosopher 3. It can also be seen it takes very long until philosopher 5 gets a chance to eat in which philosopher 5 could end up starving.

```
if(i == philosophers.length - 1) {
    philosophers[i] = new Philosopher(rightFork, leftFork,  philNumber: i+1);
}
else {
    philosophers[i] = new Philosopher(leftFork, rightFork,  philNumber: i+1);
}
```



Part iv) Similar to the discussion in part iii some philosopher such as philosopher 4 and 5 could end up starving if the first 3 philosophers end up taking a very long time to eat. In the beginning all philosophers go for their left forks first except the last philosopher who goes for their right fork. The last philosopher maybe unsuccessful in getting their right fork as philosopher 4 may have already picked it up leaving philosopher 5 in a thinking state. Since philosopher 5 cannot eat philosopher 1 has the chance to eat as their right fork is available. Philosopher 2 cannot eat until philosopher 1 is done eating same with philosopher 3, 4 and 5 they cannot start eating until the previous philosophers are done eating. Therefore, since all philosophers are depending on the previous philosopher to think and eat within a reasonable amount of time, philosopher 1 could end up causing more philosophers to starve if he/she takes a very long time thinking and eating. If every philosopher gets a chance to eat in the first cycle, then it is now possible for two philosophers to eat at the same time. No two neighbours will be able to eat at the same time only philosophers opposite each other can eat at the same time. Another situation maybe when two philosophers sitting opposite each other are fast thinkers and fast eaters. They think fast and get hungry fast. Because they are so fast, it is possible they can lock their forks and eat. After finishing eating and before their neighbours can lock the folks and eat, they come back again and lock the forks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat and may end up starving.

Part v) Comments have been added to the code

Task 1 Output After Code Modifications

```
g Semester 2022\Assessment 2022\Task1> java Client
The Chinchilla's name is Grumpy
The ZebraFinch's name is Happy
The Chinchilla's name is Grumpy
The ZebraFinch's name is Happy
```

Task 2 Output After Code Modifications

```
g Semester 2022\Assessment 2022\Task2> java TestNewsWatchers
The news watcher watching for everything
has received a new alert:
"Microsoft releases new MiOS - a new operating system for iPhones"

The news watcher watching for business
has received a new alert:
"Microsoft releases new MiOS - a new operating system for iPhones"

The news watcher watching for everything
has received a new alert:
"Classic FM signs exclusive contract with Beethoven"

The news watcher watching for everything
has received a new alert:
"New evidence Newton invented gravity after an apple fell on his head"

The news watcher watching for science
has received a new alert:
"New evidence Newton invented gravity after an apple fell on his head"
```

Task 3 Output After Code Modifications

Card Game Loss

```
g Semester 2022\Assessment 2022\Task3> java Main
Card (c/C) or Die (d/D) game? c
[QDmnds, 10Spds, 8Clbs, 2Clbs, 5Spds, QSpds, 3Hrts, 5Clbs, 6Dmnds, 2Dmnds, 3Clbs, QHrts, 3Dmnds, 5Dmnds, KSpds, 6Clbs, 9
Spds, 7Hrts, AHrts, 8Spds, 8Hrts, KHrts, 9Clbs, JDmnds, 2Spds, 7Dmnds, 4Dmnds, ADmnds, JClbs, 4Spds, 10Dmnds, 4Hrts, 2Hr
ts, 4Clbs, JHrts, KDmnds, JSpds, QClbs, 6Spds, AClbs, 10Hrts, KClbs, 3Spds, 9Hrts, ASpds, 7Clbs, 8Dmnds, 5Hrts, 10Clbs,
6Hrts, 7Spds, 9Dmnds]
Hit <RETURN> to choose a card

You chose 4Spds
Hit <RETURN> to choose a card

You chose KSpds
Cards chosen: [4Spds, KSpds]
Remaining cards: [QDmnds, 10Spds, 8Clbs, 2Clbs, 5Spds, QSpds, 3Hrts, 5Clbs, 6Dmnds, 2Dmnds, 3Clbs, QHrts, 3Dmnds, 5Dmnds
, 6Clbs, 9Spds, 7Hrts, AHrts, 8Spds, 8Hrts, KHrts, 9Clbs, JDmnds, 2Spds, 7Dmnds, 4Dmnds, ADmnds, JClbs, 10Dmnds, 4Hrts,
2Hrts, 4Clbs, JHrts, KDmnds, JSpds, QClbs, 6Spds, AClbs, 10Hrts, KClbs, 3Spds, 9Hrts, ASpds, 7Clbs, 8Dmnds, 5Hrts, 10Clb
s, 6Hrts, 7Spds, 9Dmnds]
Cards chosen: [4Spds, KSpds]
You lost!
```

Card Game Win

```
g Semester 2022\Assessment 2022\Task3> java Main
Card (c/C) or Die (d/D) game? c
[6Spds, 3Spds, JHrts, 7Spds, 2Dmnds, 8Spds, 4Clbs, 2Spds, 8Dmnds, 7Hrts, JClbs, 8Clbs, QDmnds, AHrts, KDmnds, KSpds, 7Cl
bs, 2Clbs, QClbs, 7Dmnds, 9Hrts, 4Dmnds, 6Dmnds, 10Clbs, 8Hrts, 6Hrts, QHrts, AClbs, KHrts, 4Spds, 10Hrts, 5Dmnds, 5Spds
, 2Hrts, 6Clbs, QSpds, 5Clbs, JSpds, 9Clbs, 10Dmnds, 3Clbs, 9Dmnds, KClbs, JDmnds, 9Spds, 5Hrts, 4Hrts, ADmnds, 10Spds,
ASpds, 3Hrts, 3Dmnds]
Hit <RETURN> to choose a card

You chose AClbs
Hit <RETURN> to choose a card

You chose 5Spds
Cards chosen: [5Spds, AClbs]
Remaining cards: [6Spds, 3Spds, JHrts, 7Spds, 2Dmnds, 8Spds, 4Clbs, 2Spds, 8Dmnds, 7Hrts, JClbs, 8Clbs, QDmnds, AHrts, K
Dmnds, KSpds, 7Clbs, 2Clbs, QClbs, 7Dmnds, 9Hrts, 4Dmnds, 6Dmnds, 10Clbs, 8Hrts, 6Hrts, QHrts, KHrts, 4Spds, 10Hrts, 5Dm
nds, 2Hrts, 6Clbs, QSpds, 5Clbs, JSpds, 9Clbs, 10Dmnds, 3Clbs, 9Dmnds, KClbs, JDmnds, 9Spds, 5Hrts, 4Hrts, ADmnds, 10Spd
s, ASpds, 3Hrts, 3Dmnds]
Cards chosen: [5Spds, AClbs]
You won!
```

Die Game Loss

```
g Semester 2022\Assessment 2022\Task3> java Main
Card (c/C) or Die (d/D) game? d
Hit <RETURN> to roll the die

You rolled 4
Hit <RETURN> to roll the die

You rolled 2
Numbers rolled: [2, 4]
You lost!
```

Die Game Win

```
g Semester 2022\Assessment 2022\Task3> java Main
Card (c/C) or Die (d/D) game? d
Hit <RETURN> to roll the die

You rolled 1
Hit <RETURN> to roll the die

You rolled 1
Numbers rolled: [1]
You won!
```

Task 4 Output After Code Modifications

```
g Semester 2022\Assessment 2022\Task4> java TestThread
The Example has value 9999 and has been updated 10000 time(s)
PS C:\Users\yashs\OneDrive\Documents\Uni_Work\Uni_Programs\Year
g Semester 2022\Assessment 2022\Task4> java TestThread
The Example has value 9999 and has been updated 10000 time(s)
PS C:\Users\yashs\OneDrive\Documents\Uni_Work\Uni_Programs\Year
g Semester 2022\Assessment 2022\Task4> java TestThread
The Example has value 9999 and has been updated 10000 time(s)
PS C:\Users\yashs\OneDrive\Documents\Uni_Work\Uni_Programs\Year
g Semester 2022\Assessment 2022\Task4>
```

Task 5 Output After Code Modifications – No Dead Locks Occur

```
g Semester 2022\Assessment 2022\Task5> java DiningPhilosophers
Philosopher number 4 time: 269859194240400: Thinking
Philosopher number 2 time: 269859193667400: Thinking
Philosopher number 1 time: 269859193548800: Thinking
Philosopher number 3 time: 269859194111600: Thinking
Philosopher number 5 time: 269859194366800: Thinking
Philosopher number 2 time: 269859251226200: Picking up left fork
Philosopher number 2 time: 269859253139800: Picking up right fork
Philosopher number 2 time: 269859259232800: Eating
Philosopher number 1 time: 269859262190199: Picking up left fork
Philosopher number 4 time: 269859265130400: Picking up left fork
Philosopher number 2 time: 269859269175200: Putting down right fork
Philosopher number 4 time: 269859332616699: Picking up right fork
Philosopher number 2 time: 269859352705599: Putting down left fork
Philosopher number 3 time: 269859352738300: Picking up left fork
Philosopher number 2 time: 269859396701000: Thinking
Philosopher number 1 time: 269859396710600: Picking up right fork
Philosopher number 4 time: 269859429683100: Eating
Philosopher number 1 time: 269859444107100: Eating
Philosopher number 1 time: 269859447380299: Putting down right fork
Philosopher number 4 time: 269859467980000: Putting down right fork
Philosopher number 1 time: 269859531950599: Putting down left fork
Philosopher number 2 time: 269859531993100: Picking up left fork
Philosopher number 4 time: 269859565992200: Putting down left fork
Philosopher number 4 time: 269859580984600: Thinking
Philosopher number 3 time: 269859581028500: Picking up right fork
Philosopher number 3 time: 269859583947599: Eating
```

Task 5 Optional - Output After Code Modifications

```
g Semester 2022\Assessment 2022\Task5 - Optional Task> java DiningPhilosophers
Philosopher number 3 time: 269994751690800: Thinking
Philosopher number 1 time: 269994751581800: Thinking
Philosopher number 2 time: 269994751791200: Thinking
Philosopher number 5 time: 269994752089300: Thinking
Philosopher number 4 time: 269994751941600: Thinking
Philosopher number 5 time: 269994790394800: Picking up left fork
Philosopher number 5 time: 269994873930800: Picking up right fork
Philosopher number 5 time: 269994911903099: Eating
Philosopher number 5 time: 269994975786000: Putting down right fork
Philosopher number 5 time: 269995060704200: Putting down left fork
Philosopher number 5 time: 269995152401099: Thinking
Philosopher number 4 time: 269995152499900: Picking up left fork
Philosopher number 4 time: 269995186959900: Picking up right fork
Philosopher number 4 time: 269995231378800: Eating
Philosopher number 4 time: 269995275194400: Putting down right fork
Philosopher number 4 time: 269995347862900: Putting down left fork
Philosopher number 4 time: 269995406429400: Thinking
Philosopher number 1 time: 269995406536300: Picking up left fork
Philosopher number 1 time: 269995489903400: Picking up right fork
Philosopher number 1 time: 269995503565700: Eating
Philosopher number 1 time: 269995587937099: Putting down right fork
Philosopher number 1 time: 269995663516699: Putting down left fork
Philosopher number 1 time: 269995671444399: Thinking
Philosopher number 2 time: 269995671546000: Picking up left fork
Philosopher number 2 time: 269995711197400: Picking up right fork
Philosopher number 2 time: 269995797014400: Eating
Philosopher number 2 time: 269995882291100: Putting down right fork
```