

Java coursework:

1i)

```
public abstract class Pet {
    public abstract String classOfAnimal();
    protected String name;
    public void setName(String aName) { name=aName; }
    public String getName() { return name; }
}
//moved the methods into the abstract class which is inherited into the two sub classes zebra and chinchilla
//then we call upon sub classes and through polymorphism they take the methods from the pet class
```

added code that is given in task 1

```
public static void main(String[] args) {
    Pet p=new Chinchilla();

    p.setName("Grumpy");
    System.out.println("The " + p.classOfAnimal() + "'s name is " + p.getName());

    p = new ZebraFinch();

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

Output

```
C:\Program Files\Java\jdk-15.0.2\bin
The Chinchilla's name is Grumpy
The ZebraFinch's name is Happy

Process finished with exit code 0
```

1ii)

abstract classes are faster than interfaces, have flexibility in implementation as you can either implement them fully or partially and can easily be changed without breaking the derived classes. However, it does not support multiple inheritances and cannot be instantiated.

Interfaces' pros are that they allow multiple inheritances, provide abstraction by not exposing what exact kind of object is being used in the context and provide consistency by a specific signature of the contract. However once defined it cannot change without breaking all the classes, it cannot have variables/delegates and it must implement all the contracts defined.

2i)

```
public void attach(NewsWatcher n) {
// Complete this method so that it adds the observer to the list of observers
    observers.add(n);
}
```

Observers are now added to ArrayList.

```
private void notifyObservers() {
// Complete this method to go through each observer in turn,
// sending it a message to notify that an update has occurred

    String updatedcategory = getUpdateCategory();
    if (updatedcategory == "other"){
    }
    else {
        System.out.println("The news watcher watching for " + updatedcategory + " has received a new alert:");
    }
}
```

This section of code creates the user if the user is watching the other category of news.

```
public void update() {
// Modify this so that it always prints out the update text,
// regardless of the update category
//String x = theNewsReporter.getUpdateCategory();
    String y = theNewsReporter.getUpdateText();
    System.out.println("The news watcher watching for everything has received a new alert:");
    System.out.println(y);
    System.out.println();
}
```

This section of code is from the everything watcher class and this update method takes the getupdatetext() as a variable and prints it out always as this is the watcher watching all news

```
public void update() {
// Modify this so that it only prints out the
String y = theNewsReporter.getUpdateText();
System.out.println(y);
System.out.println();
}
```

This section of code is for the selective watcher class even though here it does show that it chooses a certain type of news depending on what the watcher wants to see the code for that is in the testnews watcher class

Output

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

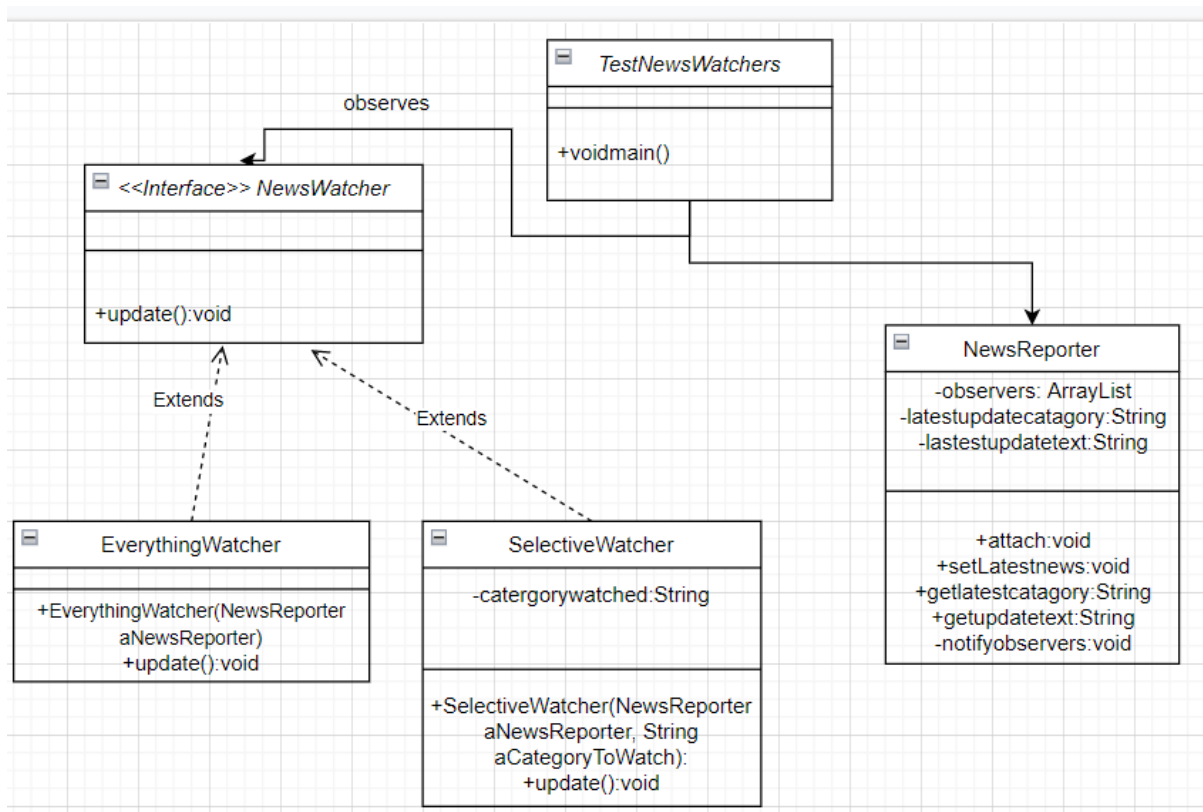
The news watcher watching for everything has received a new alert:
Microsoft releases new MiOS - a new operating system for iPhones

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

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

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

2ii)



3a)

game.java starts by prompting users to choose one of two games card or die, if cards are chosen a deck of cards is randomly shuffled 100 times by swapping two cards around (two positions in the ArrayList) Then prompts users to press enter, and they receive a random card (this is done by accessing the LCG class which generates a pseudo-random number) from the deck this is repeated twice so the user has two cards. The user wins if one or both cards are an Ace

If the die is chosen the die is rolled twice, gives a random number between 1 and 6 and shows the user said value, this value is added to the hash set named numbers rolled, this is then done again for the second die value. Then the HashSet is searched and if there is a one in the numbers rolled HashSet the player wins.

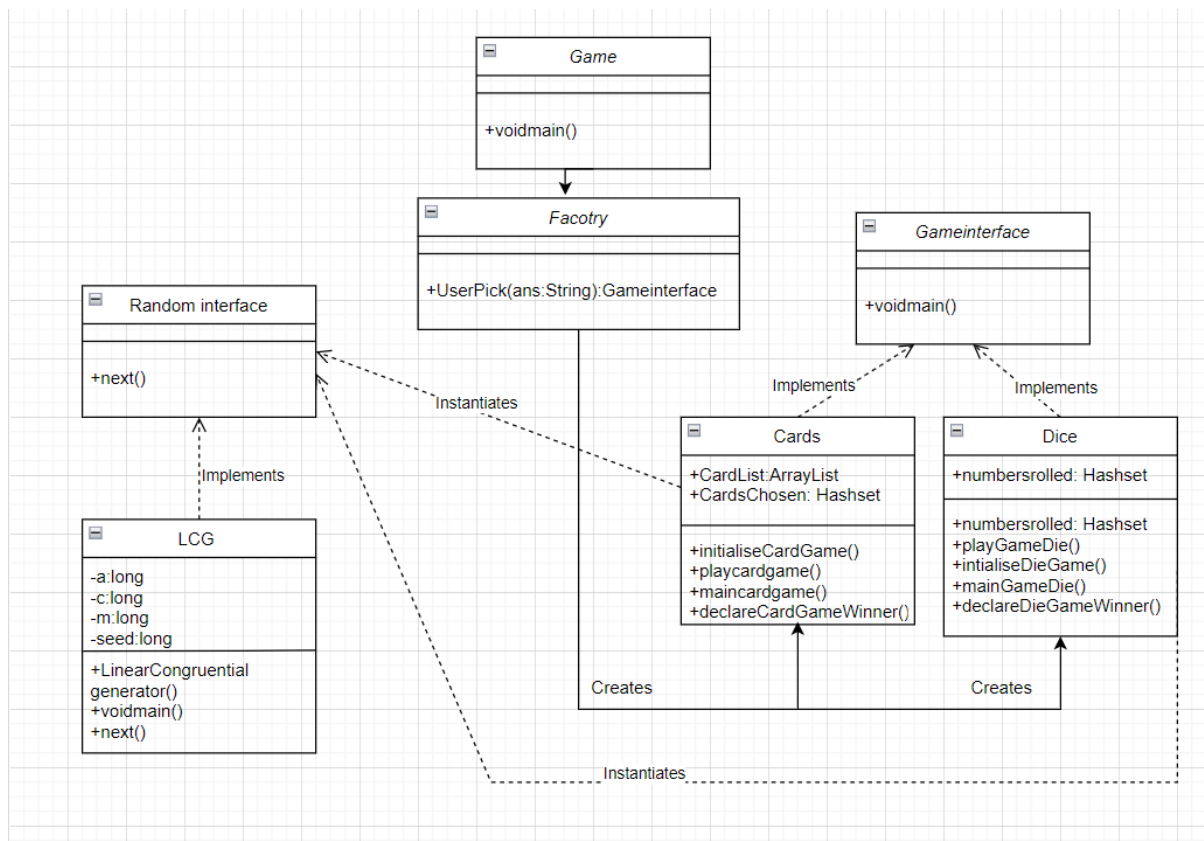
3b) (i)

Class name	Description
Game class	This will be the main class which we will be running and will connect to the factory class
Dice class	This class will play the dice game
Cards class	This class will play the cards game
Linear Congruential Generator class	This provides the random generator needed for the two games
RandomInterface	Defines the method for retrieving the random number
Game interface	Used to define the behaviour of what game is being played and forces class relationship with similar classes
Factory Class	User input decides which object is going to be made in this case which game.

My choice of design pattern is the factory design pattern I have chosen to use this design pattern because firstly it allows the sub-classes to choose the type of objects to create so in my circumstances it's the different types of games and it promotes loose coupling.

3b(ii)

Factory design pattern



3b(iii)

```

public static void main(String[] args) throws Exception {
    // Ask whether to play a card game or a die game
    System.out.print("Card (c) or Die (d) game? ");
    String ans=br.readLine();
    //making a factory as picked factory design pattern
    GameInterface UserPick = Factory.UserPick(ans);
}
  
```

Where the factory class is called and takes the user's input here to decide which object to make or in this case what game to play.

```
//calling upon the game classes
Cards c = new Cards();
Dice d = new Dice();
//user picks what game they want to play
if (ans.equals("c")) {
    c.playGame();
} else if (ans.equals("d")) {
    d.playGame();
} else {
    System.out.println("Input not understood");
    System.exit(status: 1);
}
```

Here is how the game classes are called upon to run, the answer. equals refer to the previous class where we take ans as a variable and pass it through the UserPick method.

```
public void playGame(){
    // Play card game
    // Initialise the game
    initialiseGame();

    // Play the main game phase
    mainGame();

    // Now see if (s)he has won!
    declareGameWinner();
}
```

This section of code is the same in the card and dice classes as this is where the game interface is implemented to define the behaviour of the classes.

```
public interface GameInterface {
    public void playGame();
    public void initialiseGame();
    public void mainGame();
    public void declareGameWinner();
}
//added another game interface so the games can implement this interface
```

Here you can see the game interface with all the abstract methods used to define a contract if the class implements this interface.

4i)

The code is not thread-safe because the values of the data are participating in data races with other threads which causes data to either correct or incorrect values, depending upon the order in which multiple threads access and modify the data. This happens because one the variables are shared across threads and two the variables used in the code are not immutable so they can be changed.

4ii)

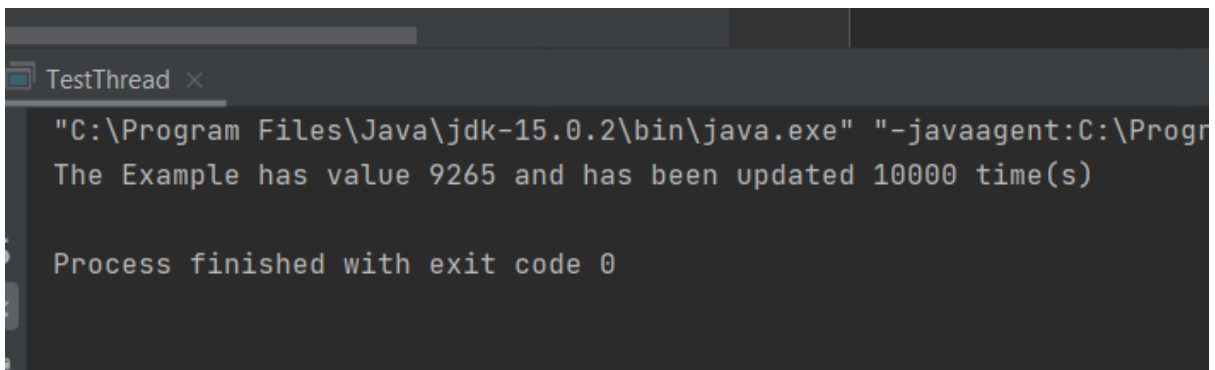
```
public synchronized void run() {
    Example e = Example.getInstance();
    e.setVal(theValue);
}

public static void main(String[] args) {
    // ...
}

public static synchronized 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; }
}
```

Output



```
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:C:\Progr
The Example has value 9265 and has been updated 10000 time(s)

Process finished with exit code 0
```

4iii) The code is now thread-safe because it uses synchronised methods that only allow one thread to access the methods, ensuring that if the data is changed, it changes without other threads interacting before it changes using intrinsic locks which lock out the other threads from running the method.

5i)

The synchronised keyword on Fork ensures that only one thread has access to that method at one time. The volatile declarations are used is because a thread might not immediately see when the variable has been updated in the cache memory and will use past instances of that variable for its methods and calculations. This may be desirable for our program as it

would always make sure that the updated value is visible to all threads, but it will increase the run time due to the volatile variable being stored on the main memory rather than the cache memory and being read from there. Right the InUse variable does not do anything because picking up the right fork causes a deadlock.

5ii)

For a deadlock to occur multiple threads need to access the same locks but obtain them in a different order for example when a thread is waiting for an object lock, that is acquired by another thread and the second thread is waiting for an object lock that is acquired by the first thread. So, the problem which is causing a deadlock in the given code is that each philosopher picks up their left fork but halts because another philosopher holds the right fork.

5iii)

How does it help avoid deadlock?

What I added to the code was an if statement in the constructor of the philosophers class that the first time it starts the first philosopher will pick up a fork to the right, this stops a deadlock from occurring as it always ensures that one philosopher will always have no forks on the first loop due to the first philosopher picking up their fork, this also leaves a spare fork to the left of the first philosopher which they can pick up and eat, ensuring no deadlocks occur where every philosopher picks up all the left forks. However, the output is messed up as when the first philosopher talks about the left spoon it is talking about the right spoon and vice versa.

```
for (int i = 0; i < problemSize; i++) {
    rightFork = forks[i];
    leftFork = forks[(i + 1) % problemSize];
    //added if statement such that the first phil picks up the right fork first
    if (i == 0){
        philosophers[i] = new Philosopher(rightFork, leftFork, philNumber: i+1);
        Thread t = new Thread(philosophers[i]);
        t.start();
    }else{
        philosophers[i] = new Philosopher(leftFork, rightFork, philNumber: i+1);

        Thread t = new Thread(philosophers[i]);
        t.start();
    }
}
```

Output


```
philosopher number 1 time: 1300820940251200: Picking up right fork
philosopher number 1 time: 1300820959349800: Eating
philosopher number 1 time: 1300820991351300: Putting down right fork
philosopher number 1 time: 1300821083464900: Putting down left fork
philosopher number 2 time: 1300821083465100: Picking up right fork
philosopher number 5 time: 1300821083678700: Picking up left fork
philosopher number 1 time: 1300821083669300: Thinking
philosopher number 2 time: 1300821128413500: Eating
philosopher number 2 time: 1300821224364400: Putting down right fork
philosopher number 2 time: 1300821322311200: Putting down left fork
philosopher number 3 time: 1300821399284200: Picking up right fork
philosopher number 2 time: 1300821399376200: Thinking
philosopher number 3 time: 1300821439244600: Eating
```

5iv)

If some philosophers take a long time to eat and a short time to think these threads are essentially locking out other threads from accessing the resources or in this case the forks used to eat, this can be made worse considering the thread sleep function is a randomly generated number in seconds between 1 and 100 thus meaning threads might only be sleeping for one second before they start competing for resources again which could cause one thread to never have access to a fork thus never eating and starving.

5v)

N/A