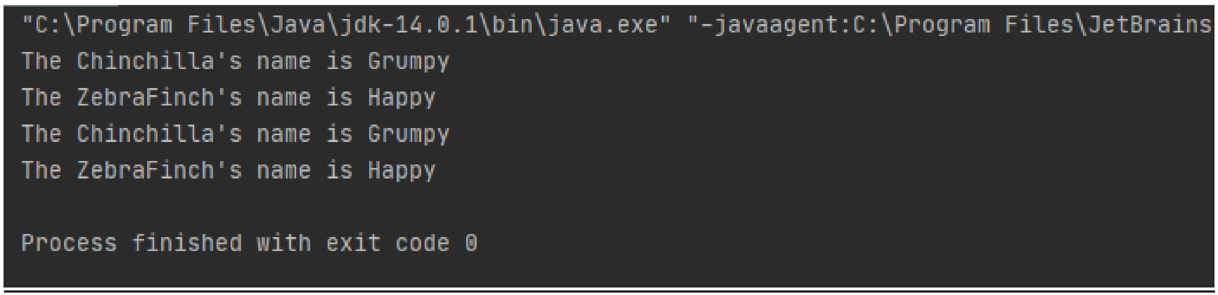
CM2307- Object Orientation, Algorithms and Data Structures

Report

C2032382

Task 1

Output:

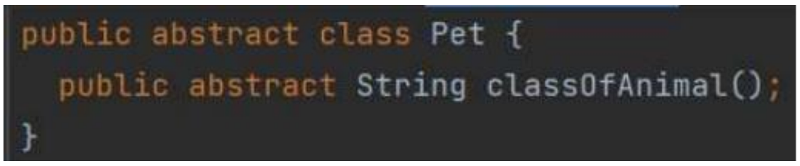


```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
The Chinchilla's name is Grumpy
The ZebraFinch's name is Happy
The Chinchilla's name is Grumpy
The ZebraFinch's name is Happy

Process finished with exit code 0
```

Part i.)

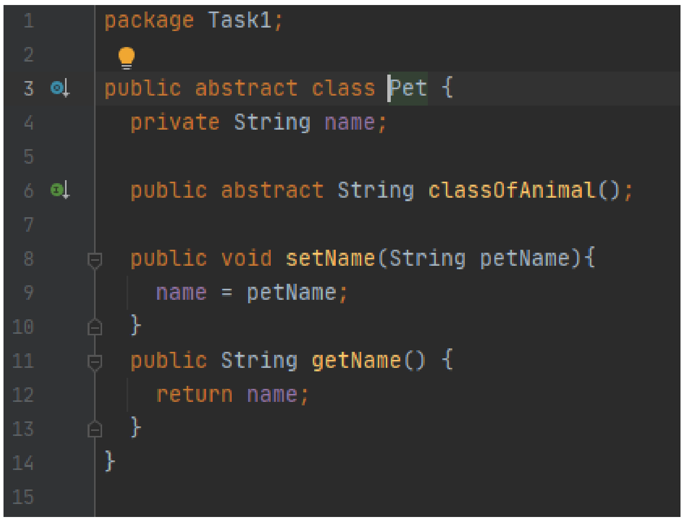
The attribute "name" and the methods "getName()" and "setName()" from the ZebraFinch and Chinchilla classes were removed as the initial step in refactoring the code. Second, the Pet class received the removed attribute and methods. ZebraFinch and Chinchilla are subclasses of Pet since all pets have similar traits (they extend Pet). They are a subclass of Pet, which means they inherit all of the characteristics and methods of the Pet template class. As a result, the getters and setters in the ZebraFinch and Chinchilla classes are unnecessary (sub classes). Finally, in the main method (class Client), a pet object is generated, which is used to set a Chinchilla's name and then print it out. After that, polymorphism is utilised to construct a ZebraFinch object, and the variable is reused. For the ZebraFinch, the "setName()" method is called, and the ZebraFinch's name is printed.

Original Pet



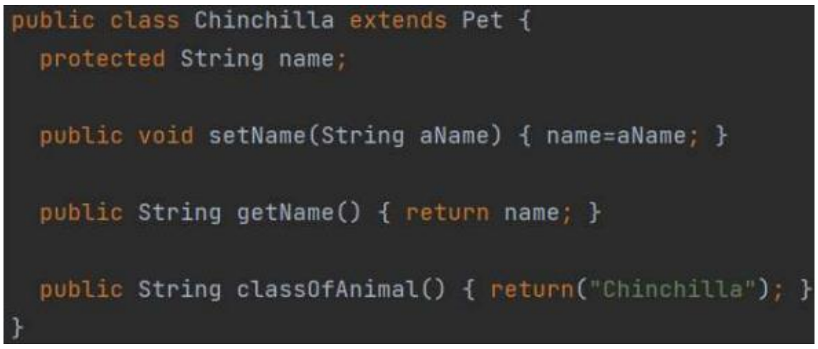
```
public abstract class Pet {
  public abstract String classOfAnimal();
}
```

Modified Pet



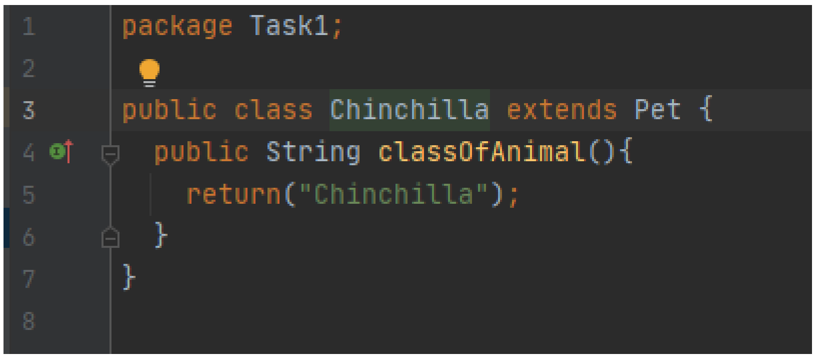
```
package Task1;

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



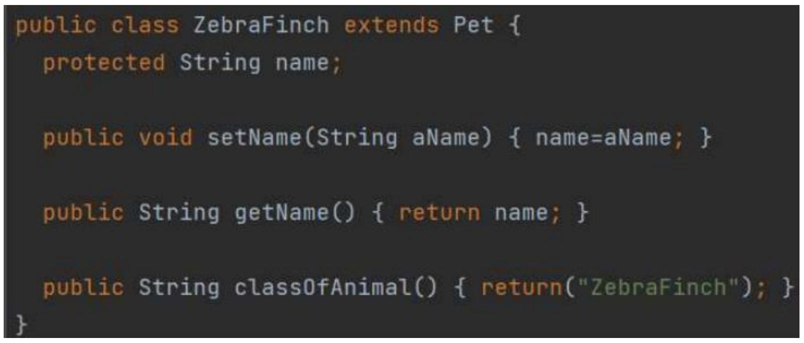
```java
public class Chinchilla extends Pet {
  protected String name;

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

  public String getName() { return name; }

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

Modified Chinchilla



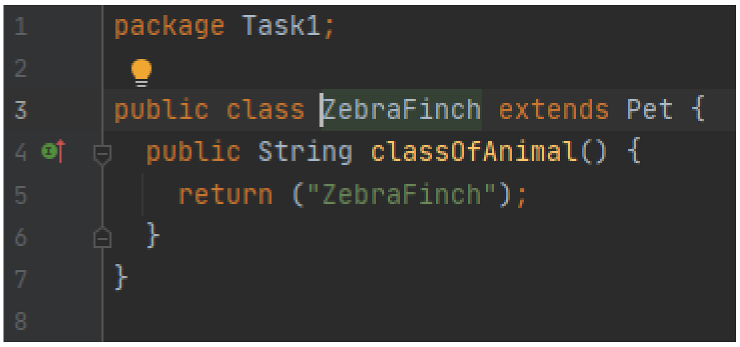
```java
package Task1;

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

Original ZebraFinch



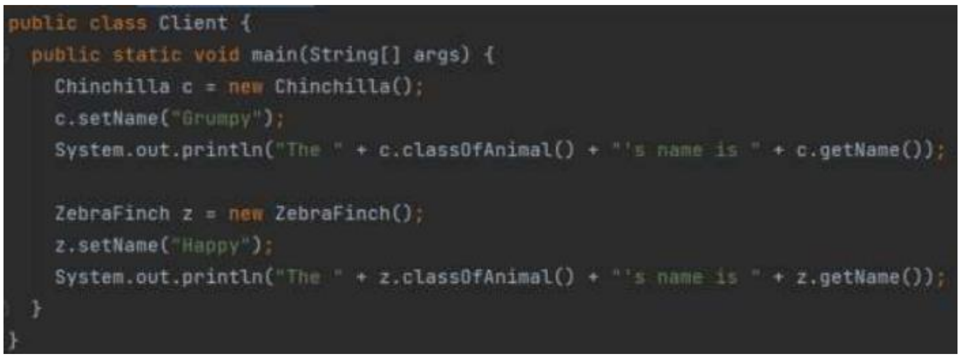
```java
public class ZebraFinch extends Pet {
  protected String name;

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

  public String getName() { return name; }

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

Modified ZebraFinch



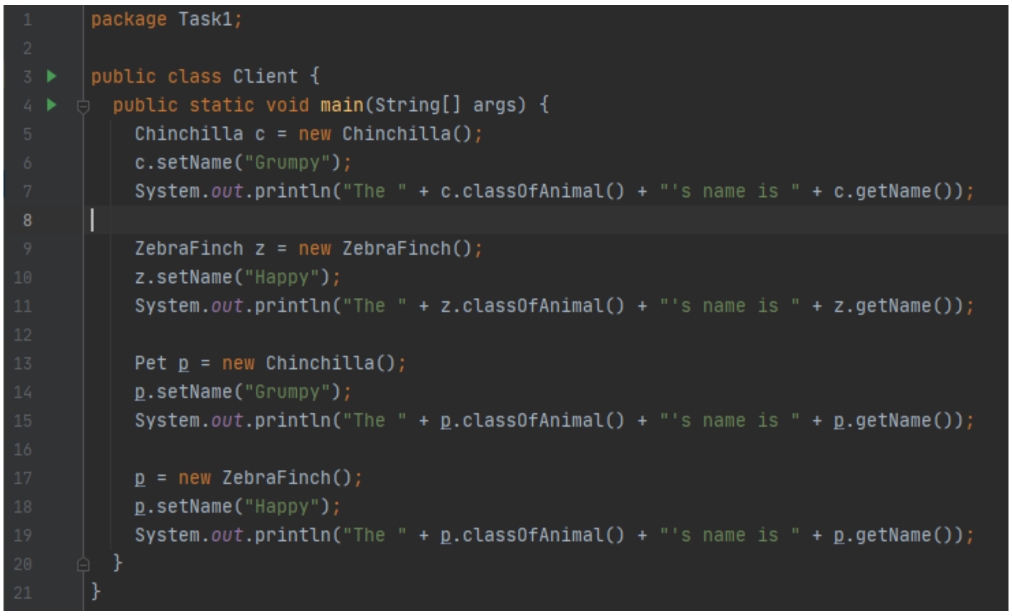
```java
package Task1;

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

Original Client



```java
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



```java
package Task1;

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());

    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());
  }
}
```

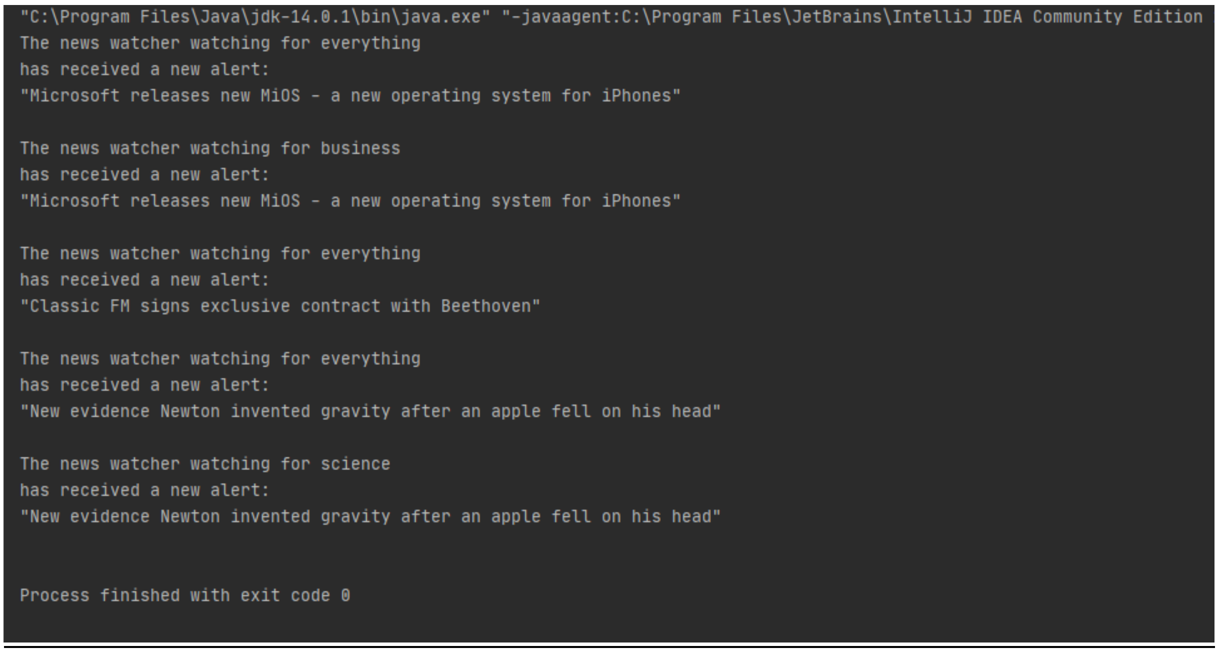
Part ii.)

The abstract class in the code serves as a template for other classes to extend, which is an excellent technique to reduce code duplication because every class that extends the super class inherits the properties and methods of the super class. Because the name has already been inherited from the Pet class, the ZebraFinch and Chinchilla do not need to implement a getter and setter for it. The complexity of the code can be hidden from the end user, leaving only the relevant functions visible. The user may not understand how the methods "setName()" and "getName()" function internally in the code, but they are aware that they exist and how to use them. Furthermore, abstract classes facilitate code reuse by allowing multiple objects to use the same method while overriding it to meet the needs of the individual object. Changes to the internal code can be done without affecting the other classes, or concrete methods that can be utilised by the classes in the future can be added.

The method signature is stored in an interface rather than the method specification, which increases readability. If Pet were an interface, it would have simply methods and no implementation details or properties. Any class that implements Pets must then implement and override the methods defined in Pet. By hiding the method implementation from the user, an interface with merely the method signature is ideal for complete abstraction. Classes can implement several interfaces, which means they can inherit methods from other interfaces. Different classes can implement the same interface and override a function to satisfy the class's individual needs. For example, if Pet were an interface, ZebraFinch and Chinchilla might implement it and override the method with their own internal class-specific features. Method definitions are not dependent on other methods or classes since interfaces are loosely connected. Because interfaces cannot have implementation details in methods, getters, and setters, as well as specifying attributes, are not possible. Even if the interface isn't used, any class that implements it must implement its methods.

<u>Task 2</u>

<u>Output:</u>

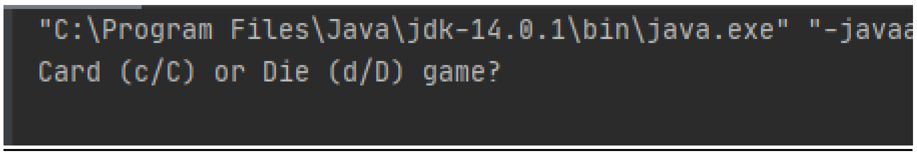


```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition
The news watcher watching for everything
has received a new alert:
"Microsoft releases new MiOS - a new operating system for iPhones"

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"


Process finished with exit code 0
```

Part ii.)

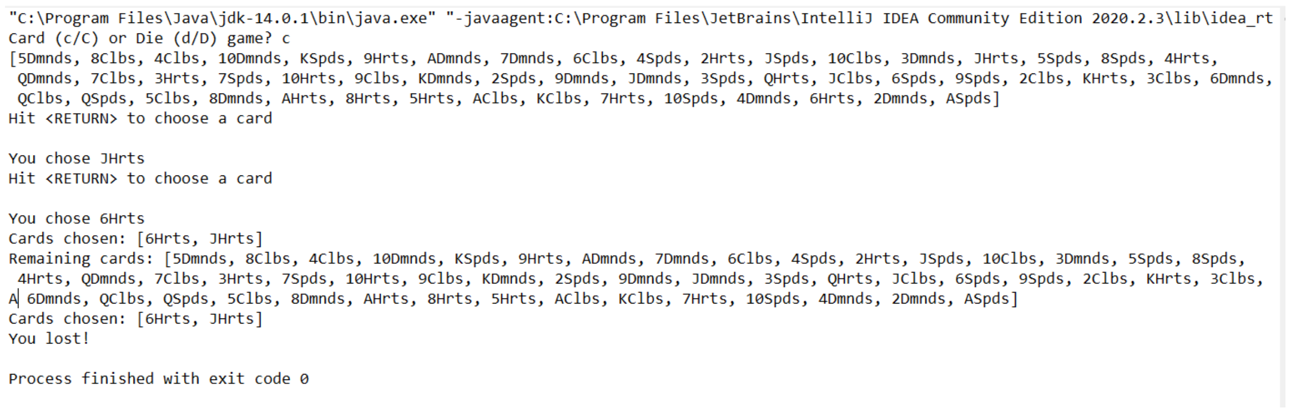
<u>Task 3</u>

<u>Output:</u>



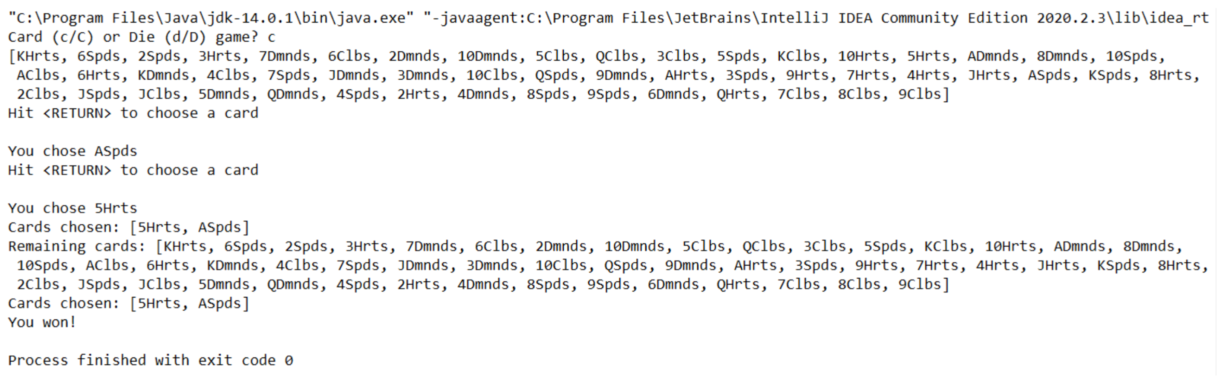
```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaa
Card (c/C) or Die (d/D) game?
```

Card Game Loss



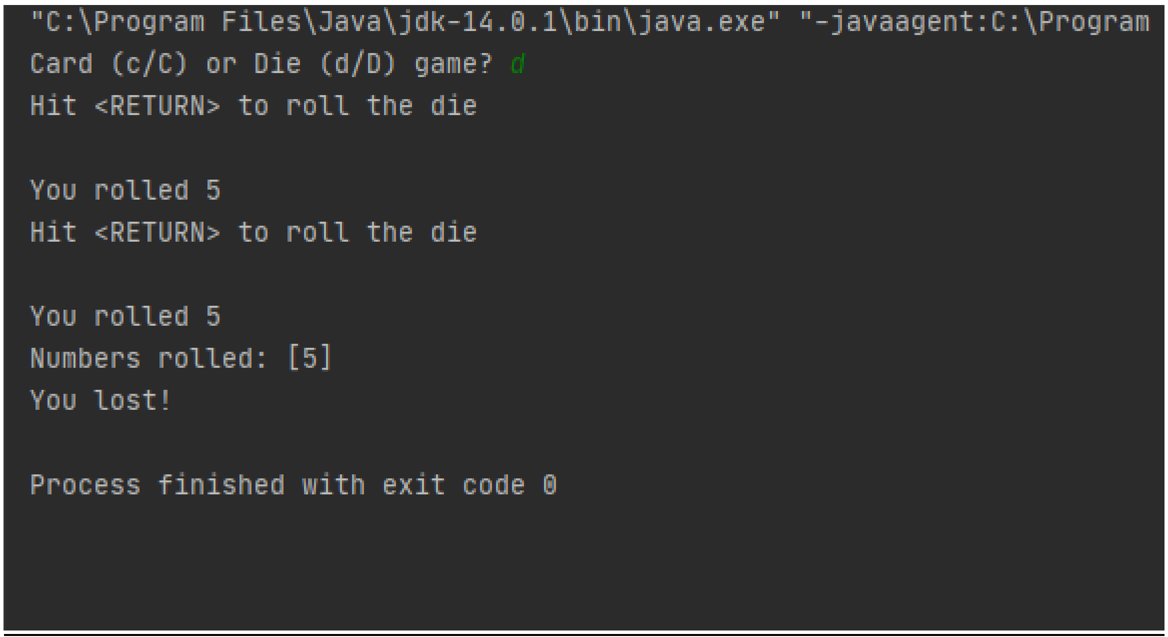
```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2020.2.3\lib\idea_rt
Card (c/C) or Die (d/D) game? c
[5Dmnds, 8Clbs, 4Clbs, 10Dmnds, KSpds, 9Hrts, ADmnds, 7Dmnds, 6Clbs, 4Spds, 2Hrts, JSpds, 10Clbs, 3Dmnds, JHrts, 5Spds, 8Spds, 4Hrts,
 QDmnds, 7Clbs, 3Hrts, 7Spds, 10Hrts, 9Clbs, KDmnds, 2Spds, 9Dmnds, JDmnds, 3Spds, QHrts, JClbs, 6Spds, 9Spds, 2Clbs, KHrts, 3Clbs, 6Dmnds,
 QClbs, QSpds, 5Clbs, 8Dmnds, AHrts, 8Hrts, 5Hrts, AClbs, KClbs, 7Hrts, 10Spds, 4Dmnds, 6Hrts, 2Dmnds, ASpds]
Hit <RETURN> to choose a card

You chose JHrts
Hit <RETURN> to choose a card

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

Process finished with exit code 0
```

Card Game Win



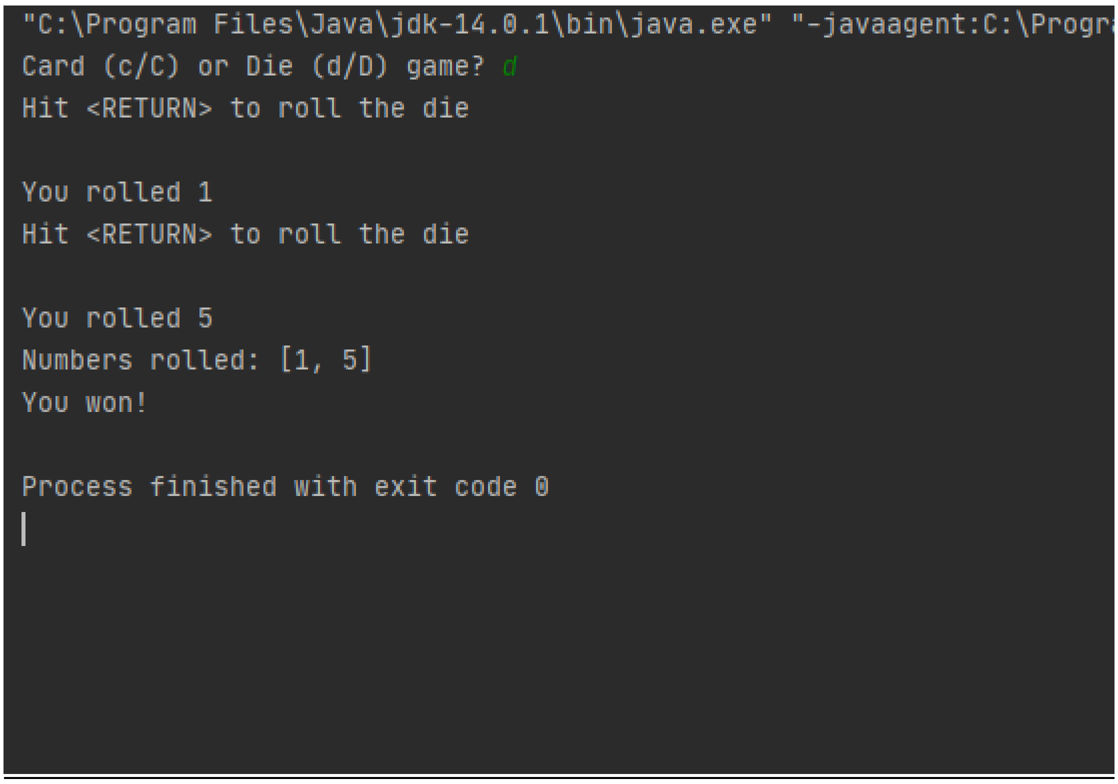
```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2020.2.3\lib\idea_rt
Card (c/C) or Die (d/D) game? c
[KHrts, 6Spds, 2Spds, 3Hrts, 7Dmnds, 6Clbs, 2Dmnds, 10Dmnds, 5Clbs, QClbs, 3Clbs, 5Spds, KClbs, 10Hrts, 5Hrts, ADmnds, 8Dmnds, 10Spds,
 AClbs, 6Hrts, KDmnds, 4Clbs, 7Spds, JDmnds, 3Dmnds, 10Clbs, QSpds, 9Dmnds, AHrts, 3Spds, 9Hrts, 7Hrts, 4Hrts, JHrts, ASpds, KSpds, 8Hrts,
 2Clbs, JSpds, JClbs, 5Dmnds, QDmnds, 4Spds, 2Hrts, 4Dmnds, 8Spds, 9Spds, 6Dmnds, QHrts, 7Clbs, 8Clbs, 9Clbs]
Hit <RETURN> to choose a card

You chose ASpds
Hit <RETURN> to choose a card

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

Process finished with exit code 0
```

Die Game Loss



```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program
Card (c/C) or Die (d/D) game? d
Hit <RETURN> to roll the die

You rolled 5
Hit <RETURN> to roll the die

You rolled 5
Numbers rolled: [5]
You lost!

Process finished with exit code 0
```

Die Game Win



```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Progr
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 5
Numbers rolled: [1, 5]
You won!

Process finished with exit code 0
```

Part a i.)

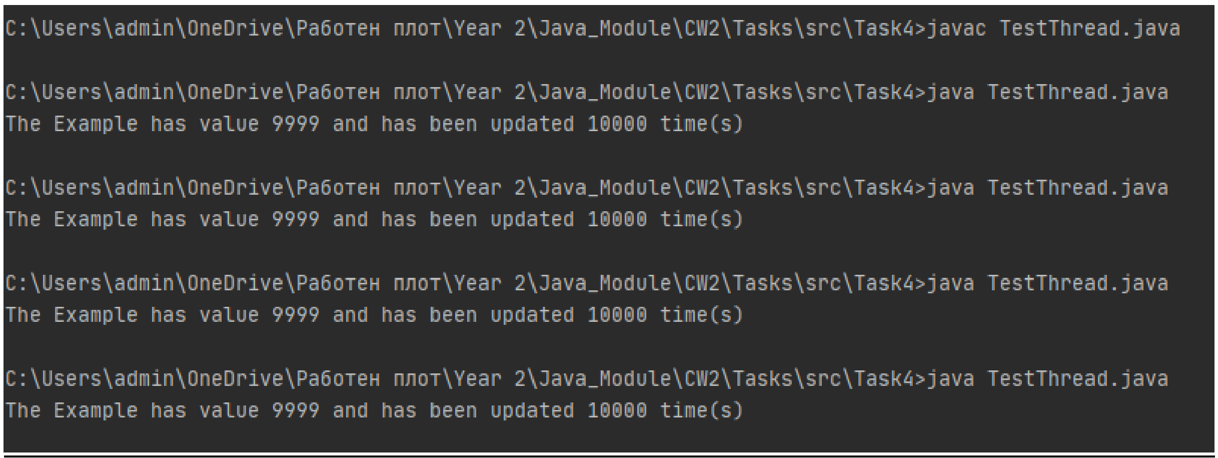
The application is a game in which the user can pick between two types of games: card games and dice games. The user can select a game by typing "c" for card games and "d" for dice games. Whichever game is selected, some characteristics are set and the play game (playCardGame() or playDieGame()) procedures are invoked. Each game contains a play game method that calls other methods to initialise the games, run the main game, and finally declare whether the user who is playing the game won or lost. Because the methods employed in both games are similar, an interface would be appropriate here. Additionally, because the game interface has several subclasses, a factory method would be appropriate to implement.

Part b i.)

Each game now has its own class under the new and improved design. As a result, the card game and the die game have separate classes. An interface class was built because each game has a similar structure (Game interface). Because this interface serves as a blueprint for the games, each one must implement and replace its methods. The usage of an interface increases code readability by hiding the complexity of the code from the user. A random number generator class that implements a random Interface has also been added. Finally, a factory class has been constructed that builds and returns a game object based on the user's input, following the factory design pattern. The advantage of utilising a factory is that it solves the problem of producing objects without needing to specify the specific class.

Part b ii.)

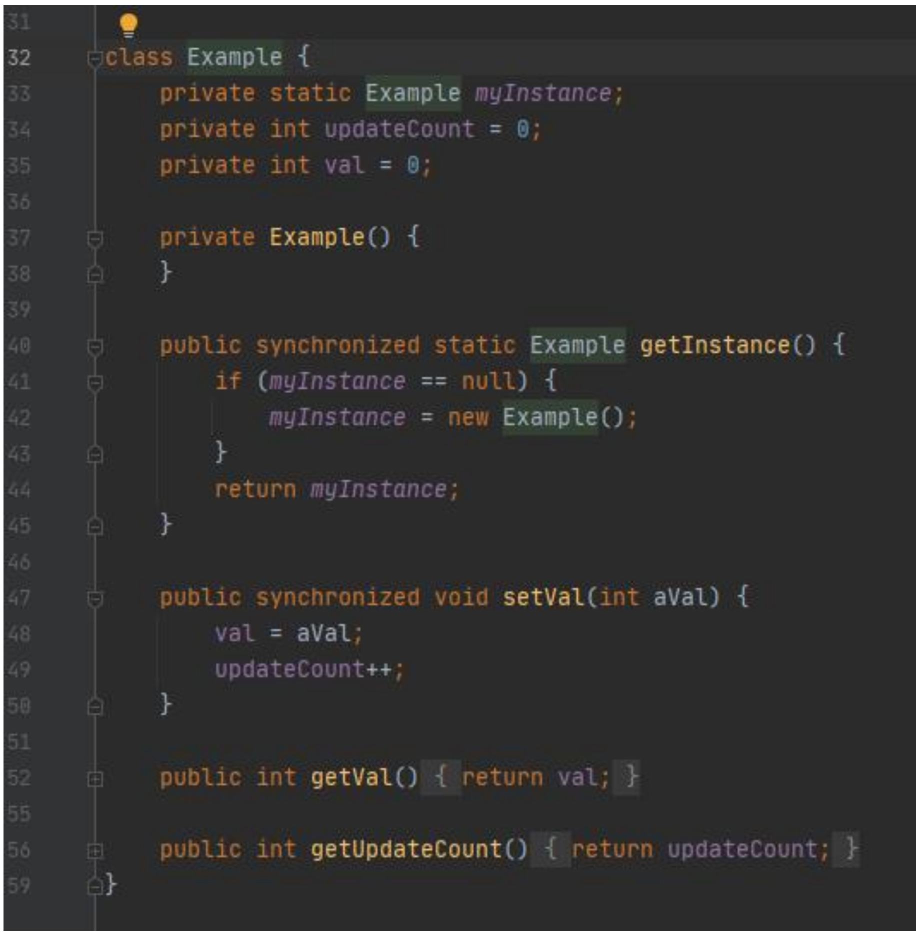
Task 4



```
C:\Users\admin\OneDrive\Работен плот\Year 2\Java_Module\CW2\Tasks\src\Task4>javac TestThread.java

C:\Users\admin\OneDrive\Работен плот\Year 2\Java_Module\CW2\Tasks\src\Task4>java TestThread.java
The Example has value 9999 and has been updated 10000 time(s)

C:\Users\admin\OneDrive\Работен плот\Year 2\Java_Module\CW2\Tasks\src\Task4>java TestThread.java
The Example has value 9999 and has been updated 10000 time(s)

C:\Users\admin\OneDrive\Работен плот\Year 2\Java_Module\CW2\Tasks\src\Task4>java TestThread.java
The Example has value 9999 and has been updated 10000 time(s)

C:\Users\admin\OneDrive\Работен плот\Year 2\Java_Module\CW2\Tasks\src\Task4>java TestThread.java
The Example has value 9999 and has been updated 10000 time(s)
```

Part i)

Part ii)



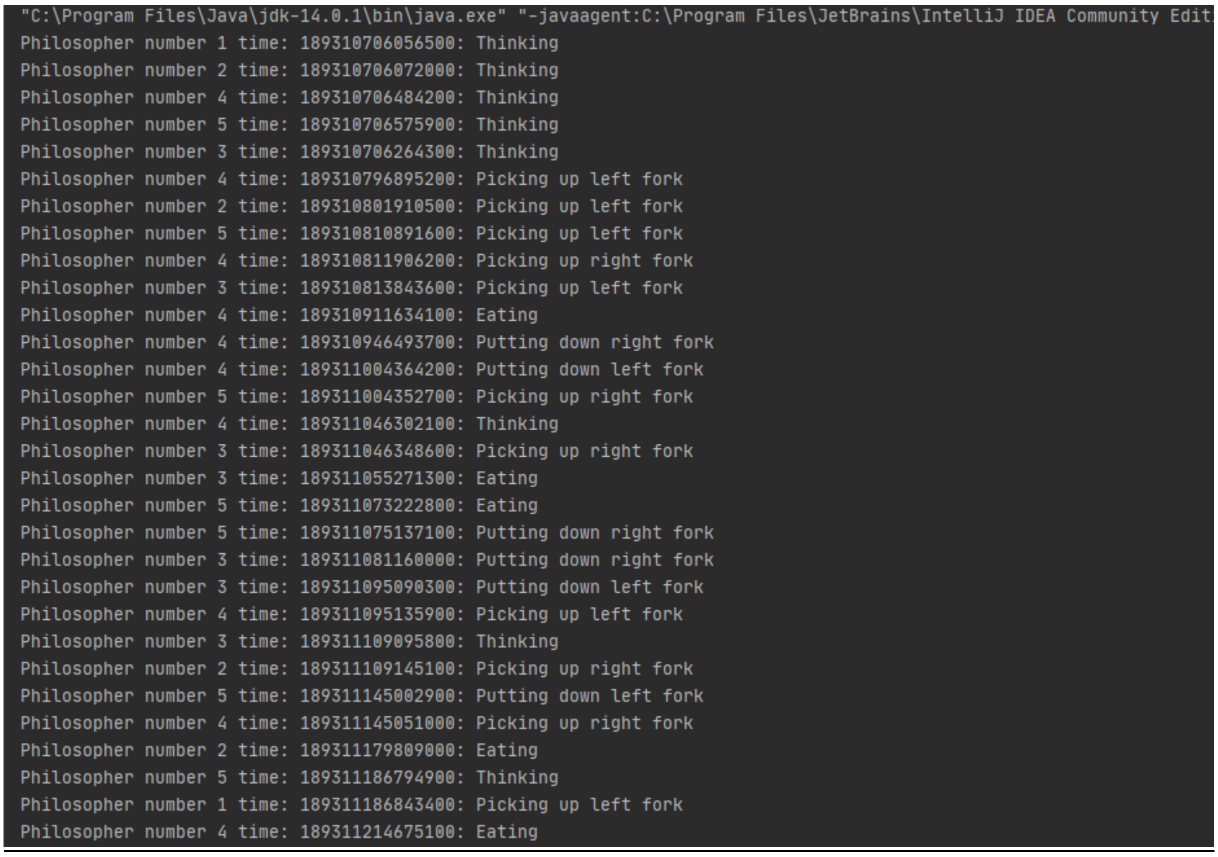
```java

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)

The synchronised keyword has been used for the getInstance() and setVal() methods to make the code tread safe. This makes it impossible for several threads to use the getInstance() or setVal()

methods at the same time, and only one thread can do so at a time. When one thread has completed execution, another thread can begin execution. This synchronisation avoids the creation of additional objects and also solves the issue of incrementing, as only one thread can increment at a time.

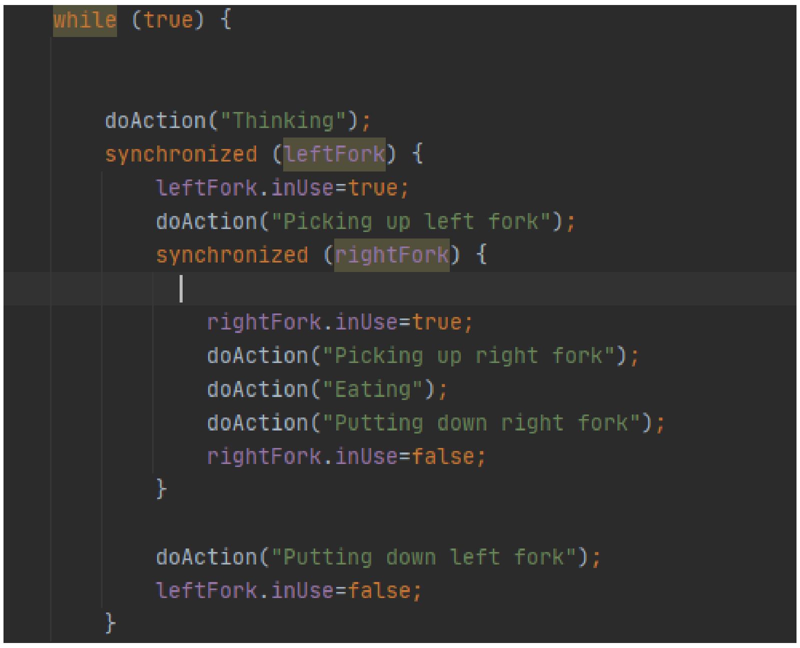
<u>Task 5</u>



```
"C:\Program Files\Java\jdk-14.0.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edit
Philosopher number 1 time: 189310706056500: Thinking
Philosopher number 2 time: 189310706072000: Thinking
Philosopher number 4 time: 189310706484200: Thinking
Philosopher number 5 time: 189310706575900: Thinking
Philosopher number 3 time: 189310706264300: Thinking
Philosopher number 4 time: 189310796895200: Picking up left fork
Philosopher number 2 time: 189310801910500: Picking up left fork
Philosopher number 5 time: 189310810891600: Picking up left fork
Philosopher number 4 time: 189310811906200: Picking up right fork
Philosopher number 3 time: 189310813843600: Picking up left fork
Philosopher number 4 time: 189310911634100: Eating
Philosopher number 4 time: 189310946493700: Putting down right fork
Philosopher number 4 time: 189311004364200: Putting down left fork
Philosopher number 5 time: 189311004352700: Picking up right fork
Philosopher number 4 time: 189311046302100: Thinking
Philosopher number 3 time: 189311046348600: Picking up right fork
Philosopher number 3 time: 189311055271300: Eating
Philosopher number 5 time: 189311073222800: Eating
Philosopher number 5 time: 189311075137100: Putting down right fork
Philosopher number 3 time: 189311081160000: Putting down right fork
Philosopher number 3 time: 189311095090300: Putting down left fork
Philosopher number 4 time: 189311095135900: Picking up left fork
Philosopher number 3 time: 189311109095800: Thinking
Philosopher number 2 time: 189311109145100: Picking up right fork
Philosopher number 5 time: 189311145002900: Putting down left fork
Philosopher number 4 time: 189311145051000: Picking up right fork
Philosopher number 2 time: 189311179809000: Eating
Philosopher number 5 time: 189311186794900: Thinking
Philosopher number 1 time: 189311186843400: Picking up left fork
Philosopher number 4 time: 189311214675100: Eating
```

Part i.)

The synchronised keyword on the Fork object acts as a lock, ensuring that only one thread can execute/access the object at any given moment. The synchronised keyword has been used on a piece of code (picture below), which signifies that only one thread can execute that block of code at a time. And all variables in this synchronised block must be flushed and updated in main memory. The volatile keyword tells the Java compiler and threads that the value of the variable with the volatile keyword should never be cached and should always be read from main memory. If the value of "inUse" is cached and another thread updates it, then any thread that uses the cached value will produce inaccurate results. The variable is unnecessary since it makes it easy to identify which fork object's "inUse" value is being changed (left or right) without having to build a Fork object. Because this software isn't packed and the "inUse" variable is public, any class or application can access it. The programme would still function normally if these variables were removed.



```
while (true) {


    doAction("Thinking");
    synchronized (leftFork) {
        leftFork.inUse=true;
        doAction("Picking up left fork");
        synchronized (rightFork) {

            rightFork.inUse=true;
            doAction("Picking up right fork");
            doAction("Eating");
            doAction("Putting down right fork");
            rightFork.inUse=false;
        }

        doAction("Putting down left fork");
        leftFork.inUse=false;
    }
```

Part ii.)

A deadlock occurs when a system's progress is blocked because each process is waiting on a resource held by another process. Deadlocks typically arise when one thread is waiting for an object lock that has been acquired by another thread, while the other thread is waiting for an object lock that has been gained by another thread. Because all threads are waiting for each other to release the lock, they have reached a deadlock and will continue to wait indefinitely. A deadlock occurs when each of the Dining Philosophers takes up their left forks, indicating that there is no right fork left because their neighbour has already taken that fork. As a result, the Philosophers enter an infinite waiting state known as the impasse, in which they will each wait until a right fork becomes available.

Part iii.) As all philosophers pick up their left forks, the impasse situation occurs. Because there are no right forks available, all philosophers are trapped in a never-ending circular waiting condition known as stalemate. As a result, one philosopher must take a right fork to break the circle of waiting.

As a result of the programming change, all philosophers will pick up their left forks except the last, who will pick up their right fork, effectively ending the waiting cycle. Only two persons will be able to eat at any given time during these cycles, and no two neighbours will be able to eat at the same time. The problem with the first cycle is that philosopher 5 may go hungry if the other philosophers take too long to eat. My testing with the modification has shown that it prevents deadlocks, but the outputs are utterly random and incorrect each time they are executed.

Part iv.)

If the first three philosophers take a long time to eat, certain philosophers, such as philosophers 4 and 5, may go hungry. Except for the last philosopher, who takes the right fork, all philosophers start with their left forks. The last philosopher may have been unsuccessful in obtaining their right fork because philosopher 4 may have already obtained it, leaving philosopher 5 in a state of thought. Because philosopher 5 is unable to eat, philosopher 1 has the opportunity to do so because their right fork is available. Philosopher 2 is unable to eat until philosopher 1 has finished eating, and similarly, philosophers 3, 4, and 5 are unable to eat until the prior philosophers have finished eating. As a result, because all philosophers rely on the previous philosopher to think and eat in a suitable length of time, if philosopher 1 takes an excessive amount of time thinking and eating, more philosophers may starve. If each philosopher had a chance to eat in the first cycle, two philosophers can now eat at the same time in the second cycle. No two neighbours will be able to eat at the same time; only philosophers on separate sides of the room will be able to do so.