OBJECT ORIENTED PROGRAMMING
ASSIGNMENT - 2

Name: Shravani Ankur Wanjari
class: Rhinos (B)
Roll no: 31
Student ID: 22WU0101093

Date of submission:

1. ~~Describe~~ Describe ~~Eto~~ CLASSPATH environmental variable?

→ The 'classpath' environmental variable is used in Java programming to specify the locations where the JAVA virtual Machine (JVM) should look for class files. It is basically a list of ~~directly~~ directories and/or JAR files (Java archive files) that contain the compiled Java classes that a Java program needs to run.

2. Differentiate between Character stream and Byte stream? List the Byte stream classes and Character stream classes?

→ A byte stream handles binary data, which, typically, is represented as a series of bytes. It is used to read and write raw binary data, such as images, audio files, and compressed files. ~~On the other hand, a character stream~~

On the other hand, a character stream handles textual data, which is typically represented using a character encoding, such as ASCII or Unicode. It is used to read and write text data, such as plain text files or XML files.

• Byte stream classes:- At the top are two abstract classes:
① InputStream ② OutputStream

• Character stream classes: At the top are two abstract classes:
① Reader ② Writer.

Pg 02

3. Design a JAVA program to read a character from a console?

→
```
import java.io.BufferedReader;
import java.io.IOException;
import java.~~Input~~io.InputStreamReader;

public class ReadCharFromConsole {
   public static void main (String[]args){
      BufferedReader reader = new BufferedReader(new InputStreamReader
                                                           (System.in));
      System.out.println ("Enter a Character:");

      try{
         Char c= (Char)reader.read ( );
         System.out.println ("You entered: "+ c );
      }Catch (IOException e){
            System.out.println ("An error occured while reading
                                         the input.");
            e.printStackTrace();
         }
   }
```

4. What is Java Exception Handling? Why do you need Java Exception Handling?

→ Java Exception Handling is a Mechanism that allows a programmer to handle runtime errors and exceptional events that may occur during the ~~exce~~ execution of a program. In Java, exceptions are ~~handled~~ objects that represent errors or exceptional situations, such as an invalid input, a network failure, or a file not found.

Java Exception Handling provides a way to handle exception~~s~~ gracefully and prevent the program from crashing. It allows the programmer to catch the exception and take corrective actions, such as displaying a meaningful error message, logging the exception, or retrying the operation that caused the exception.

In addition, Java Exception Handling ~~is an essential to higher~~ provides a way to separate error handling code from the regular code, making the program more modular and maintainable.

5. What is the Java Exception Hierarchy? What is the difference between Errors and Exceptions?

→ The Java Exception Hierarchy is a hierarchical structure of classes and interfaces that define the different types of exceptions that can occur in a Java program. This hierarchy is built on top of the Throwable class, which is the root of the exception hierarchy.

Difference between exception and error:

Exception class represents recoverable exceptions that can be handled by the programmer, while the Error class represents irrecoverable errors that cannot be handled by the programmers

The difference between Errors and Exception in the Errors are caused by system failures or other catastrophic events that cannot be recovered from; while Exceptions are caused by conditions that can be handled programmatically.

Example: OutofMemoryError is an error that occurs when JVM runs out of memory, while ~~the~~ ArithmeticException is an exception that can come while carrying out an arithmetic operation. ~~which can be~~ The Error can't be handled by programmer but the Exception can be caught and handled.

6. Differentiate between multiprocessing and multithreading. What is to be done to implement these in a ~~go~~ program?

→ ~~Mult~~ Multiprocessing involves running multiple processes simultaneously, where each process has its own memory space and can run on a separate CPU core. Each process can perform independant tasks, and they communicate with each other using inter-process communication mechanisms like pipes, shared memory, or sockets. ~~Multiprocessing~~

Multithreading involves running multiple threads within the same process, where each thread shares the same memory space. Threads are lighter than processes and can communicate with each other directly using shared variables. Multithreading is suitable for I/O-bound tasks that involve a lot of waiting, such as network communication, disk I/O, or user input/output.

To implement multiprocessing or multithreading in a Java program, you can use the java.util.concurrent package, which provide classes and interfaces for concurrency programming.

7. Write a program to create four threads using Runnable interface.

→
```
public class RunnableThread implements Runnable{
    private String threadName;
    public RunnableThread (String name) {
        this.threadName = name;
}

public void run(){
    System.out.println("Thread" + threadName +"is running");
}

public static void main (String[] args){
    RunnableThread thread1 = new RunnableThread ("Thread 1");
    RunnableThread thread2 = new RunnableThread ("Thread2");
    RunnableThread thread3= new RunnableThread ("Thread 3");
    RunnableThread thread4= new RunnableThread ("Thread 4");

    Thread t1 = new Thread(thread1);
    Thread t2 = new Thread (thread2);
    Thread t3 = new Thread (thread3);
    Thread t4= new Thread (thread4);

     t1.start();
     t2.start();
     t3.start();
     t4.start();
}
```

8. Write a program to create three threads in your program and context switch among the threads using sleep functions. Pg05

→
```
public class ThreadOne implements Runnable{
  private String threadName;

  public ThreadOne(String name){
    this.threadName = name;
  }

  public void run(){
    for (int i= 1; i<=5; i++){
      System.out.println ("Thread" + threadName + "is running
                                          for the" + i+"th time");
      try{
        Thread.sleep(1000);
      }catch (InterruptedException e){
          e.printStackTrace();
      }

    }
}

public static void main (String[] args){
   Thread t1 = new Thread (new ThreadOne ("Thread 1"));
   Thread t2 = new Thread (new ThreadOne ("Thread 2"));
   Thread t3 = new Thread (new ThreadOne ("Thread 3"));

   t1.start();
   try{
     Thread.sleep(1000);
    }catch (InterruptedException e){
       e.printStackTrace();
    }
    t2.start();
    try{
      Thread.sleep(1000)
     }catch (InterruptedException e){
       e.printStackTrace();
     }
      t3.start();
}   }
```

9. How do we start and stop a thread? Pg06

→ • Start() : The 'start()' method is used to start execution of a thread.

'Stop(): The 'stop()' method is used to stop the execution of a thread.

10. What is thread based pre-emptive multitasking?

→ In Java, thread-based preemptive multitasking refers to the ability of the Java Virtual Machine (JVM) to manage and switch between multiple threads of execution preemptively. Preemptive multitasking means that the JVM can interrupt a currently running thread and switching to another thread that is ~~willing~~ waiting to be executed.

Java's preemptive multitasking is based on priorities assigned to threads. Each thread is assigned a priority, which determines its relative importance to the JVM.

11. Write the complete life cycle of a thread in Java.

→ It goes as follows: New → Runnable → Running → Blocked → Terminated.

1. New state: When a thread is created using the $\boxed{\text{new}}$ keyword or the $\boxed{\text{Thread}}$ class constructor, it is in the $\boxed{\text{New}}$ state. In this state has been ~~running~~ created but hasn't started running.

2. Runnable state: In this state, the thread is ready to run, but the scheduler has not yet assigned it a time slice to execute.

3. Running state: When the scheduler assigns a time slice to a thread in the $\boxed{\text{Runnable}}$ state. In this state, the thread is actively executing its code.

4. Blocked state: A thread can enter the Blocked state if it is waiting for a resource that is not available, such as a lock on a synchronized block. When a thread is blocked, it cannot continue executing until the resources becomes available.

Terminated State: When a thread completes executing and it cannot be started again, it enters terminated state.

Q12 Write a program that creates two threads. First thread ~~point~~ prints the numbers from 1 to 100 and the other ~~hand~~ thread prints the numbers from 100 to 1.?

```
→ public class TwoThreads {
    public ~~class~~ static void main (String[] args) {
      Thread thread1 = new Thread (new PrintOne ());
      Thread thread2 = new Thread (new Print Hundred ());
    }
  }
class printOne implements Runnable {
    public void run () {
      for (int i = 1; i<=100; i++) {
        System.out.println(i);
        try{
          Thread.sleep(100);
        } catch (InterruptionException e) {
          e.printStackTrace();
        }
      }
    }
  }
class Printhundred implements Runnable {
    public void run(){
      for (int i = 100; i >= 1; i--){
        System.out.println (i);
        try{
          Thread.sleep (100);
        } catch (InterruptedException e){
          e.printStackTrace();
        }
      }
  } }
```