UNIT – IV

1. a) *What is exception handling? Explain an example of exception handling in the case of division by zero.*

Ans: In Java, exception is an event that occurs during the execution of a program and disrupts the normal flow of the program's instructions, which aborts/terminates the program.

- In java, when the interpreter encounters a run-time error, ***it creates an exception object and throws it.***
- If the exception object is not caught and handled properly, it terminates the program.
- Exceptions are not recommended, therefore these exceptions are to be handled to continue the execution of the program.

Program:

```
import java.util.Scanner;
class Exception1{
Scanner input = new Scanner(System.in);
void Div(){
        try{
         System.out.print("Enter the Numerator: ");
         int a = input.nextInt();
         System.out.print("Enter the Denominator: ");
         int b = input.nextInt();
         int res = a/b;
         System.out.println("The Quotient is: "+res);
            }
       catch (ArithmeticException e1) { //Handling the Exception
         System.out.println("You gave the Denominator as Zero")
          }
public static void main(String[] args){
         Exception1 E1 = new Exception1();
         E1.Div();  } }
```

Input:

Enter the Numerator: 20

Enter the Denominator: 0

Output:

You gave the Denominator as Zero.

b) Write a Java program that illustrates the application of multiple catch statements.

Ans:

Program:

```
public class MultipleCatchBlock{  //class Starts
        public static void main(String[] args) {  //main method starts
       try{
            int a[]=new int[5];
            a[5]=30/0;
          }
      catch(ArithmeticException e){
                System.out.println("Arithmetic Exception occurs");
                }
      catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("ArrayIndexOutOfBounds Exception occurs");
                }
    catch(Exception e) {
                System.out.println("Parent Exception occurs");
               }
             System.out.println("Rest of the code");
   }  //main method ends
}    //class Ends
```

Output:

```
Arithmetic Exception occurs
Rest of the code
```

2. a)*What are the advantages of using the exception handling mechanism?*

Ans:

1. ***Error Detection and Debugging***: Exception handling helps detect errors and exceptions that occur during program execution, making it easier to identify and debug issues.
2. ***Program Robustness:*** By handling exceptions gracefully, you can prevent your program from crashing or terminating abruptly due to unexpected conditions, improving its robustness.
3. ***Separation of Error-Handling Logic***: Exception handling allows you to separate error-handling logic from the main program flow, making your code more organized and maintainable.
4. ***Maintainability:*** Makes code more maintainable and readable by clearly delineating error-handling code from regular program logic.
5. ***Enhanced User Experience***: Properly handled exceptions contribute to a more user-friendly experience by presenting meaningful error messages

b)Explain Java exceptions keywords with an example code.

In Java exceptions can be handled using five keywords. The following table describes each.

1. try ➢ The "**try**" keyword is used to specify a block where we place the code/program to monitor/observe the exceptions. ➢ If an exception occurs in **try** block, it is thrown (informs to the user/programmer). ➢ The **try** block must be followed by either **catch** or **finally**.
2. catch ➢ The "**catch**" block is used to handle the exception. ➢ It must be preceded by try block , means we can't use catch block alone. ➢ It can be followed by finally block later.
3. finally ➢ The "**finally**" block is used to place the code that must be executed, whether an exception is handled or not.
4. throw ➢ The "**throw**" keyword is used to manually throw an exception.
5. throws ➢ The "**throws**" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. ➢ It doesn't throw an exception. It is always used with method signature.

3. a) Explain in detail Java's built-in exceptions.

Java provides a variety of built-in exceptions, which are organized under two main categories: checked exceptions and unchecked exceptions.

i) Checked Exceptions

Checked exceptions are those that are checked at compile-time. They must be either caught or declared in the method signature using the throws keyword.

a) **IOException:** Thrown when an I/O operation fails or is interrupted.

b) **FileNotFoundException**: Thrown when an attempt to open a file denoted by a specified pathname has failed.

c) **SQLException:** Thrown when there is a database access error or other errors related to SQL operations.

d) **ClassNotFoundException:** Thrown when an application tries to load a class through its string name but no definition for the class with the specified name could be found.

ii) Unchecked Exceptions

Unchecked exceptions are those that are not checked at compile-time, meaning they can be thrown at runtime and do not need to be declared in a method's throws clause.

i) **RuntimeException:** The superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. Some common subclasses are:

ii) **NullPointerException:** Thrown when an application attempts to use null where an object is required.

iii) **ArrayIndexOutOfBoundsException:** Thrown to indicate that an array has been accessed with an illegal index.

iv) **IllegalArgumentException:** Thrown to indicate that a method has been passed an illegal or inappropriate argument.

v) **NumberFormatException:** A subclass of IllegalArgumentException, thrown when an attempt is made to convert a string to a numeric type but the string does not have the appropriate format.

vi) **ArithmeticException:** Thrown when an exceptional arithmetic condition has occurred, such as division by zero.

b) Write a program with nested try statements for handling exceptions.

```
public class NestedTry{
    public static void main(String[] args){
        try{     //Outer try block
            int res = 30/0; //Arithmetic exception raises in Outer try
            System.out.println(res);
            try{   //Inner try block
                int array[] = new int[5];
                array[3] = 30;
               }
            catch(ArrayIndexOutOfBoundsException i){ //Inner Catch Block
                System.out.println("Caught the Inner Exception");
               }
           }
        catch (ArithmeticException O){ //Outer Catch Block
            System.out.println("Caught the Outer Exception");
            }
        }
    }
```

Output:
Caught the Outer Exception

4. a) Explain how to create your own exception in a Java program.

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, the user can also create exceptions which are called 'user-defined Exceptions'.

1) Create the new exception class extending Exception class
2) Create a public constructor for a new class with string type of parameter
3) Pass the string parameter to the super class
4) Create try block inside that create a new exception and throw it based on some condition
5) write a catch block and use some predefined exceptions
6) write the optionally finally block.

Program:

```
import java.util.Scanner;
class Age_Exception extends Exception {
        public Age_Exception (String message) {
                super(message);  }

public class UserException{
    static  Scanner input = new Scanner(System.in);
    public static void VoterAge() {
     try  {
        System.out.println("Enter the age: ");
        int age = input.nextInt();

        if (age<18) {
             throw new Age_Exception ("He/she is below 18 Years"); }
       else {
            System.out.println("Person is eligible to vote");  }
      }
    catch (Age_Exception e){
        System.out.println("Error: " + e.getMessage());  }
   }
public static void main(String args[]){
 VoterAge();  }
}
```

Input: Enter the age: 15

Output: He/she is below 18 Years

b) Write a Java program that illustrates the try-catch-finally clause.

- Java finally block is a block used to execute important code such as closing the connection, etc.
- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- The finally block follows the try-catch block.

Program:

```
public class TestFinally{
    public static void main(String args[]){
       try {    //below code throws divide by zero exception
            System.out.println("Inside the try block");
             int data=25/0;
             System.out.println(data);
             }
    catch(NullPointerException e){
      System.out.println(e);        //cannot handle Arithmetic type exception
                                    //can only accept Null Pointer type exception
           }
    finally {  //executes regardless of exception occured or not
      System.out.println("finally block is always executed");
    }
    }
```

Output:

```
Inside the try block
finally block is always executed
Exception in thread “main" java.lang.ArithmeticException: /by zero
```

5. Explain the following:

a) Checked exceptions and Unchecked exceptions.

Checked exceptions:

- Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- The compiler ensures whether the programmer handles the exception or not.
- The programmer should have to handle the exception; otherwise, the system has shown a compilation error.
- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions.

UnChecked exceptions:

- The **unchecked** exceptions are just opposite to the **checked** exceptions.
- The compiler will not check these exceptions at compile time.
- Usually, it occurs when the user provides bad data during the interaction with the program.
- The classes that inherit the RuntimeException are known as unchecked exceptions.

  Eg: ArithmeticException,

  IllegalArgumentException,

  IndexOutOfBoundsException,

  NegativeArraySizeException and NullPointerException.

b)Explain the difference between throw and throws keywords in Java.

Name	***throw***	***throws***
Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
Type of exception	Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
Declaration	throw is used within the method.	throws is used with the method signature.
Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

6. *a) Define the following:*

(i) Single-tasking:

Single tasking in computing refers to the ability of an operating system to manage single task at a time.

Eg: DOS

(ii) Multitasking:

Multitasking in computing refers to the ability of an operating system to manage multiple tasks at the same time.

Eg: Windows.

(iii) Multiprocessing:

- Multi process in computing refers to the ability of an operating system to manage multiple processes / programs at the same time.
- Multiprocessing requires more than one CPU to increase computing power, speed and memory.

(iv) Multithreading:

Multithreading in computing refers to a single process with multiple code segments (Threads) run concurrently and parallel to each other to increase computing power

b) ***Explain creating a thread by extending the thread class with an example code.***

```
class A extends Thread{
    public void run(){
        for (int i=1;i<=2;i++){
            System.out.println("Thread A:"+i);}
        System.out.println("Thread A Executed");
    }
}
class Main{
    public static void main(String[] args){
    A t1 = new A();
    t1. start();
    }
}
```

7. a) What is the differentiate between a thread and a process.

	Process	Thread
1	Process means any program is in execution.	Thread means a segment of a process**.**
2	The process is isolated.	Threads share memory.
3	The process does not share data with each other.	Threads share data with each other.
4	The process is less efficient in terms of communication.	Thread is more efficient in terms of communication.
5	The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
6	The process takes more time to terminate.	The thread takes less time to terminate.
7	It takes more resources.	It takes less resources.
8	If one process is obstructed then it will not affect the operation of another process.	If one thread is obstructed then it will affect the execution of another process.

b) What is the differentiate between multiprocessing and multithreading.

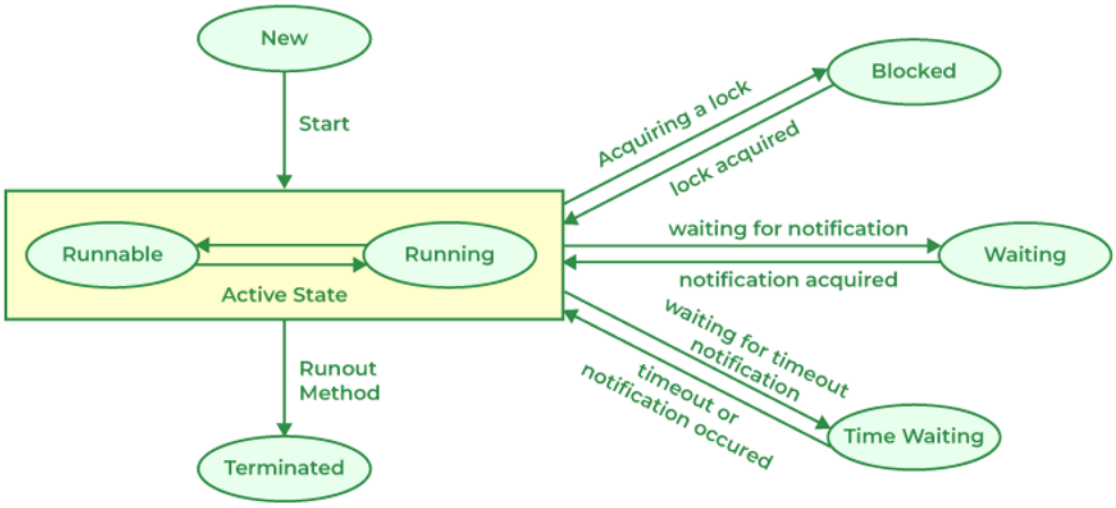
	Multi Processing	Multi Threading
1	Multiprocessing involves more than one CPU (processor) to execute the processes /programs	Multithreading involves a single CPU (processor)to execute a single process with multiple code segments (i.e., Threads) run concurrently.
2	Multiprocessing increase computing power, speed and memory.	Multithreading uses a single process and parallel to each other to increase computing power.
3	Multiprocessing is for increasing speed	Multithreading is for hiding latency.
4	Multiprocessing is parallelism	Multithreading is concurrency.
5	Multiprocessing allocates separate memory and resources for each process or program	Multithreading threads belonging to the same process share the same memory and resources as that of the process.
6	Multiprocessing is slow compared to multithreading.	Multithreading is faster compared to Multiprocessing.
7	Multiprocessing is best for computations.	Multithreading is best for IO.
8	Less economical.	Economical.

8. Explain the complete life cycle of a thread with an example code.

The life cycle of a thread in Java can be broken down into several states: New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated. Here's an explanation of each state along with a simple example to illustrate these states.

Thread Life Cycle States

1. **New:** A thread is in this state when it is created but not yet started. It is the state when a thread object is instantiated.
2. **Runnable:** A thread enters this state when start() method is called. The thread is ready to run and is waiting for CPU time.
3. **Blocked:** A thread enters this state when it is waiting for a monitor lock to enter or re-enter a synchronized block/method.
4. **Waiting:** A thread is in this state when it is waiting indefinitely for another thread to perform a particular action (e.g., using wait()).
5. **Timed Waiting:** A thread is in this state when it is waiting for another thread to perform a specific action within a stipulated amount of time (e.g., using sleep(), wait(long timeout), join(long millis)).
6. **Terminated:** A thread enters this state when it has finished its execution or is terminated due to an exception.



Program:

```
class MyThread extends Thread {
   public void run() {
      try {
         System.out.println(Thread.currentThread().getName() + " is in
RUNNABLE state");

         Thread.sleep(2000); // TIMED WAITING state
         System.out.println(Thread.currentThread().getName() + " is in TIMED
WAITING state");

         synchronized (this) { // WAITING state
            wait(1000); // Waiting for 1 second
            System.out.println(Thread.currentThread().getName() + " is in
                                                        TIMED WAITING state");
         }
         synchronized (MyThread.class) { // Blocked state
            System.out.println(Thread.currentThread().getName() + " is in
                                                      BLOCKED state"); }

      } catch (InterruptedException e) {
         e.printStackTrace(); }
   }
}

public class ThreadLifecycleDemo {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      System.out.println(t1.getName() + " is in NEW state"); // NEW state
      t1.start();        // Starting the thread - moves to RUNNABLE state

      try {
         t1.join();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }

      System.out.println(t1.getName() + " is in TERMINATED state");
   } }
```

9. List out the ways to create a thread and explain how to create a thread using the Runnable interface.

In java, we can create threads in two ways

1. Extending Thread Class
2. Implementing Runnable interface

Program:

```
class Newthread implements Runnable {
      public void run() {              //Implementing the run() method
            System.out.println("NewThread started : ");
            for (int i=1;i<=3;i++){
                  System.out.println("class A:"+i);}
            System.out.println("NewThread Finished");
       }
}
class Main {
     public static void main(String[] args){
     Newthread t1 = new Newthread();  //Creating an object of the class
     Thread T1 = new Thread(t1);   //Creating an object of thread class and
                                   passing reference t1 to the Thread Constructor
    T1.start(); }
}
```

Output:

```
NewThread started :
class A:1
class A:2
class A:3
NewThread Finished
```

10. a) How do we set priorities for threads?

a) Priority means the number of resources allocated to a particular thread.
b) Every thread created in JVM is assigned a priority. The priority range is between 1 and 10.
   - Minimum priority is 1.
   - Normal priority is 5.
   - Maximum priority is 10.
c) The default priority of any thread is 5.
d) It is advised to adjust the priority using the Thread class's constants,

   Thread.MIN_PRIORITY

   Thread.NORM_PRIORITY (i.e., 5)

   Thread.MAX_PRIORITY

e) We have the ability to adjust the priority of any thread using ***setPriority().***

Program for setting the priority:

```
public class ThreadPriority extends Thread{
   public void run (){
   System.out.println ("Running thread name is:" + Thread.current Thread().
                              getName ());
  System.out.println ("Running thread priority is:" + Thread.currentThread().
                             getPriority ()); }
  public static void main (String args[]){
         ThreadPriority m1 = new ThreadPriority ();
         m1.setPriority (Thread.MIN_PRIORITY);
        m1.start ();
 } }
```

Output:

```
Running thread name is: Thread-0
Running thread priority is:1
```

b) What are interrupting threads?

1. The interrupt() method of thread class is used to interrupt the thread.
2. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behavior and doesn't interrupt the thread but sets the interrupt flag to true.
3. If any thread is in sleeping (sleep()) or waiting state (wait()), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.

Program:

```
class A extends Thread{
public void run(){
    try{
            Thread.sleep(5000);
            for (int i=1;i<=5;i++){
        System.out.println("Thread A:"+i); }
        System.out.println("Thread A Executed");
       }
    catch (InterruptedException e){
   System.out.println ("Thread A Interrupted");
   System.out.println (e);
    }
  }
}
class Main{
public static void main(String[] args){
A t1 = new A();
t1.setName("ThreadA");
t1. start();
System.out.println(t1.getName()+ " is in"+ t1.getState());
t1.interrupt();
System.out.println(t1.getName()+ "is interrupted" +
t1.isInterrupted());
System.out.println(t1.getName()+ "is interrupted" + t1.interrupted());
}
}
```

Output:

ThreadA is in RUNNABLE

Thread A Interrupted

java.lang.InterruptedException: sleep interrupted

ThreadA is interrupted true

ThreadA interrupted false