BASIC

1) PROGRAM – CHECKED I/O EXCEPTION

AIM: To implement Checked Exception (IO Exception)

THEORY

- Exceptions are unexpected event that occurs during execution of a program.
- It affects the flow of the program instructions, which can cause the program to terminate abnormally.
- There are 3 types of Exception: (i) Checked, (ii) Unchecked Exceptions & (iii) Errors
- Checked Exceptions are defined as exceptions which are identified during the compile time of a program.
- We usually handle the exception using a try-catch block of program code.
- One such example is 'IO Exception', this exception throws when an input & output operation fails

PROGRAM CODE

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class IO_Example
{public static void main(String[] args)
{ String fn = "nonExistentFile.txt"; // This file does not exist
try
{FileReader fr = new FileReader(fn); // Attempt to open and read from the non-existent file
BufferedReader br = new BufferedReader(fr);
String line;
while ((line = br.readLine()) != null)
{System.out.println(line);}
// Close the resources (this part will not be reached if an exception occurs during opening)
br.close(); fr.close();}
catch (IOException e)
{System.err.println("An IOException occurred: " + e.getMessage());}}}
```

COMPILATION STEPS

```
javac IO_Example.java
```

java IO_Example

OUTPUT

An IOException occurred: nonExistentFile.txt (The system cannot find the file specified)

2) PROGRAM – UNCHECKED EXCEPTION

AIM: To implement Unchecked Exceptions.
(ArithmeticException, Null Pointer Exception, Array Index Out Of Bounds Exception).

THEORY

- Exceptions are unexpected event that occurs during execution of a program.
- It affects the flow of the program instructions, which can cause the program to terminate abnormally.
- There are 3 types of Exception: (i) Checked, (ii) Unchecked Exceptions & (iii) Errors
- Unchecked Exceptions are defined as exceptions which are identified during the run time of a program.
- We usually handle the exception using a try-catch block of program code.
- Examples of Unchecked Exceptions are – ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException

PROGRAM CODE/COMPILATION STEPS/OUTPUT

//ArithmeticException

Program Code -

```
public class A_Example
{public static void main(String[] args)
{int result, a=10,b=0;
try
{result=a/b;
System.out.println("Result:"+result);}
catch (ArithmeticException e)
{System.out.println("An ArithmeticException occured: Division by Zero");}}}
```

Compilation Steps –

javac A_Example.java

java A_Example

Output –

An ArithmeticException occured: Division by Zero

//ArrayIndexOutOfBoundsException

Program Code –

```
public class AI_Example
{public static void main(String[] args)
{int arr[]={1,2,3};
try
{System.out.println("Array Value At Index 4:"+arr[4]);}
catch (ArrayIndexOutOfBoundsException e)
{System.out.println("An ArrayIndexOutOfBoundsException occured: Index value out of Range");}}}
```

Compilation Steps –

```
javac AI_Example.java
java AI_Example
```

Output –

```
An ArrayIndexOutOfBoundsException occured: Index value out of Range
```

//NullPointerException

Program Code –

```
public class NP_Example
{public static void main(String[] args)
{String s=null;
try
{System.out.println("Length of String:"+s.length());}
catch (NullPointerException e)
{System.out.println("A NullPointerException occured: No value assigned");}}}
```

Compilation Steps –

```
javac NP_Example.java
java NP_Example
```

Output –

```
A NullPointerException occured: No value assigned
```

3) PROGRAM – THREAD SYNCHRONIZATION

AIM: To implement the concept of Thread Synchronization

THEORY

- A thread represents a single, independent path of execution within a program.
- It is the smallest unit of execution that can be managed by the operating system's scheduler.
- Thread Synchronization in java is a mechanism used to control access to shared resources by multiple threads in a concurrent environment.
- It's primary purpose is to prevent data inconsistency and race conditions that can arise when multiple threads attempt to modify the same shared data simultaneously.
- Synchronized Method – When a method is declared 'synchronized', only 1 thread can execute that method on a given object at a time. The lock is acquired on the object instance (this).

PROGRAM CODE

```
//Counter class is created with synchronized increment()
class counter
{private int c=0;
public synchronized void increment()
{c++;}
public int getCount()
{return c;}}
public class execute
{public static void main(String[] args)
{counter c=new counter();
Thread t1=new Thread();
for(int i=0;i<3;i++)
{c.increment();}
Thread t2=new Thread();
{for(int i=0;i<3;i++)
{c.increment();}}
t1.start();t2.start();
try
{t1.join();t2.join();}
catch (InterruptedException e)
```

```
{e.printStackTrace();}
System.out.println("Final Count:"+c.getCount());}}
```

COMPILATION STEPS

```
javac execute.java
java execute
```

OUTPUT

```
Final Count:6
```

4) PROGRAM – ARRAYLIST

AIM: To create an Array List for adding, removing, printing, searching employee names.

THEORY

- An ArrayList in Java is a resizable (or dynamic) array from the java.util package that can grow or shrink automatically as elements are added or removed, unlike regular arrays with a fixed size.
- Elements can be accessed using their index, just like the arrays.
- Duplicate elements are allowed.
- Elements are stored in the order they are inserted.
- ArrayList is not Synchronized, that is it is not thread-safe

PROGRAM CODE

```
import java.util.ArrayList;
public class EM
{public static void main(String[] args)
{ArrayList<String> e= new ArrayList<String>();
e.add("Rucha"); e.add("Priyanka"); e.add("Ananya");
System.out.println("Employees: "+e);
e.remove("Rucha");
System.out.println("Updated Employee List: " +e);
String s = "Rucha";
if(e.contains(s))
{System.out.println(s+" found in the ArrayList");}
else
{System.out.println(s+" not found in the ArrayList");}
System.out.println("Final Employee List: " + e);}}
```

COMPILATION STEPS

javac EM.java

java EM

OUTPUT

Employees: [Rucha, Priyanka, Ananya]

Updated Employee List: [Priyanka, Ananya]

Rucha not found in the ArrayList

Final Employee List: [Priyanka, Ananya]

5) PROGRAM – LINKEDLIST

AIM: To create a Linked List for adding, removing, printing, searching student names that are ordered by their index position.
(addFirst(),addLast(),add(),remove(),removeLast(),peekLast()).

THEORY

- A LinkedList in java is a dynamic linear data structure where elements are not stored in contiguous memory locations, unlike arrays.
- Instead, each element. called a 'Node', contains both the data and a reference (or link) to the next node in the sequence.
- This structure allows for efficient insertion and deletion of elements without requiring data reorganization, as only the pointers need to be updated.
- LinkedLists can grow or shrink in size dynamically as elements are added or removed.

PROGRAM CODE

```
import java.util.LinkedList;
public class LL
{public static void main(String[] args)
{LinkedList<String> s=new LinkedList<String>();
s.add("Amy"); s.addFirst("John"); s.addLast("Morris"); s.add("Dave");
System.out.println("Students: "+s);
s.remove("Dave");
s.removeLast();
System.out.println("After Removal: "+s);
System.out.println("Last Student (peekLast()):"+s.peekLast());
String name="Amy";
if(s.contains(name))
{System.out.println(name+" found in the List");}
else
{System.out.println(name+" not found in the List");}
System.out.println("Final Student List: "+s);}}
```

COMPILATION STEPS

javac LL.java

java LL

OUTPUT

```
Students: [John, Amy, Morris, Dave]
After Removal: [John, Amy]
Last Student (peekLast()):Amy
Amy found in the List
Final Student List: [John, Amy]
```

6) PROGRAM – TREEMAP

AIM: To create TreeMap of employee names and their age, print sorted employee data by employee names.

THEORY

- In java, a TreeMap is a part of the java.util package that implements the Map interface and NavigableMap interface.
- It stores key-value pairs in a sorted order based on the natura ordering of its keys, or by a custom 'Comparator' provided at the time of creation.
- The primary distinguishing feature of TreeMap is that it maintains its sorted order based on the keys.
- TreeMap does not allow null keys. But the TreeMap does permit null values.
- It is not inherently synchronized.

PROGRAM CODE

```
// Creating a TreeMap to store employee names and their ages

import java.util.*;

public class TM

{public static void main(String[] args)

{TreeMap<String, Integer> em= new TreeMap<>();

em.put("Aditi", 30); em.put("Bindu",25); em.put("Charan", 35);

// Printing the sorted employee data

System.out.println("Employee Data (Sorted by Name): ");

for (Map.Entry<String, Integer> entry: em.entrySet())

{System.out.println("Name: " + entry.getKey()+", Age: " + entry.getValue());}}}
```

COMPILATION STEPS

```
javac TM.java

java TM
```

OUTPUT

```
Employee Data (Sorted by Name):

Name: Aditi, Age: 30
```

Name: Bindu, Age: 25

Name: Charan, Age: 35

7) PROGRAM - HASHMAP

AIM: To create a HashMap that maps employee names to employee salary

THEORY

- In java, HashMap is a class within the java.util package that provides an implementation of the Map interface.
- It stores data in key-value pairs, where each key is unique and maps to a specific value.
- It uses an efficient mechanism for efficient storage and retrieval of elements.
- It is does not guarantee a specific order of elements.
- HashMap is not Thread-safe. External synchronization mechanisms are required to prevent data corruption.
- It permits 1 null key and multiple null values

PROGRAM CODE

```
import java.util.HashMap;

public class HM

{public static void main(String[] args)

{HashMap<String, Integer> es=new HashMap<>();

es.put("Aditi", 1000); es.put("Manvitha",2000); es.put("Jahnvi", 3000);

System.out.println("Employee Salaries: "+es);}}
```

COMPILATION STEPS

```
javac HM.java

java HM
```

OUTPUT

```
Employee Salaries: {Aditi=1000, Manvitha=2000, Jahnvi=3000}
```

8) PROGRAM – ITERATION

AIM: To execute Iteration over Collection using Iterator Interface and List Iterator Interface

THEORY

- The Iterator interface provides a universal way to traverse collections in a forward direction.

- The ListIterator interface is a sub-interface specific to List implementations that allows for bidirectional traversal and additional operations like modifying elements, adding new elements and retrieving indexes.
- ListIterator inherits from Iterator but extends its functionality for ordered, indexed collections.

PROGRAM CODE

```
import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

public class JI

{public static void main(String[] args)

{List<String> l= new LinkedList<>();

l.add("Welcome"); l.add("to"); l.add("programming");

System.out.println("The list is given as: "+ l);

// get the iterator on the list

Iterator<String> itr = l.iterator();

// Returns true if there are more number of elements.

while (itr.hasNext())

{System.out.println(itr.next());}}}// Returns the next element.
```

COMPILATION STEPS

```
javac JI.java

java JI
```

OUTPUT

```
The list is given as: [Welcome, to, programming]

Welcome

to

programming
```