CMR INSTITUTE OF TECHNOLOGY

USN

CMRIT — CMR INSTITUTE OF TECHNOLOGY, BENGALURU. ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test III – September 2023

Sub:	Object Oriented Programming with Java							Sub Code:	22MCA22
Date:	21.09.23	Duration:	90 min's	Max Marks:	50	Sem:	II	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

	PART I	MARKS	OBE	
			CO	RBT
1	a. Define automatic type conversion. Explain with example. b. Explain the following: i. new operator ii. Super keyword **OR**	[6+4]	CO1	L2
2	What are the different iteration statements and jump statements in Java? Given examples	[10]	CO1	L3
	PART II			
3	Give any five methods of StringBuffer class with syntax and example. **OR**	[10]	CO1	L2
4	Explain the multiple usage of static keyword in Java with example program.	[10]	CO1	L2

	PART III			
5	What is nested interface? Give an example to explain how it works. **OR**	[10]	CO2	L2
6	Demonstrate with the help of a program how can you implement user defined exceptions in Java.	[10]	CO3	L4
	PART IV			
7	Briefly discuss annotations in Java. **OR**	[10]	CO1	L2
8	Explain thread communication using notify(), wait() and notifyAll() methods.	[10]	CO1	L2
	PART V			
9	Write short notes about: InetAddress class, URL Connection class. **OR**	[5+5]	CO2	L1
10	Discuss the classes and interfaces that inherit properties of collection interface.	[10]	CO2	L4

1. a. Define automatic type conversion. Explain with example.

Widening conversion or automatic type conversion takes place in Java when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign a value of a smaller data type to a bigger data type.

For Example, in java, the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

```
// Java Program to Illustrate Automatic Type Conversion
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        int i = 100;

        // Automatic type conversion
        // Integer to long type
        long l = i;

        // Automatic type conversion
        // long to float type
        float f = l;

        // Print and display commands
        System.out.println("Int value " + i);
        System.out.println("Long value " + l);
        System.out.println("Float value " + f);
    }
}
```

b. Explain the following: i. new operator   ii. Super keyword

i. The new operator is used in Java to create new objects. It can also be used to create an array object.
Let us first see the steps when creating an object from a class −
Declaration − A variable declaration with a variable name with an object type.
Instantiation − The 'new' keyword is used to create the object.
Initialization − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

```
public class Puppy {
   public Puppy(String name) {
      // This constructor has one parameter, name.
      System.out.println("Passed Name is : " + name );
   }
   public static void main(String []args) {
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "jackie" );
   }
}
```

ii. The super keyword in Java is a reference variable which is used to refer immediate parent class object.
Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.
Usage of Java super Keyword:
A. super can be used to refer immediate parent class instance variable.

```
class Animal{
String color="white";
}
class Dog extends Animal{
```

```
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

B. super can be used to invoke immediate parent class method.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

C. super() can be used to invoke immediate parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

2. What are the different iteration statements and jump statements in Java? Given examples

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the break statement or the continue statement is encountered.

Loops are basically means to do a task multiple times, without actually coding all statements over and over again. For example, loops can be used for displaying a string many times, for counting numbers and of course for displaying menus.

Loops in Java are mainly of three types :-

1. 'while' loop
2. 'do while' loop
3. 'for' loop

The 'while' loop example :-

```
class WhileLoop
```

```
  {

     public static void main(String args[])

      {

           int i=1;
           while(i<=3)
                {
                     System.out.println(i);
                     i++;
                 }
         }
   }
```

The output of the above code will be :-
1
2
3
The 'do while' loop :-
It is very similar to the 'while' loop shown above. The only difference being, in a 'while' loop, the condition is checked beforehand, but in a 'do while' loop, the condition is checked after one execution of the loop.

Example code for the same problem using a 'do while' loop would be :-

```
class

DoWhileLoop
  {

     public static void main(String args[])

       {

               int i=1;
               do
                 {
                     System.out.println(i);
                      i++;
                   } while(i<=3);
      }
}
```

The output of the above code will be :-
1
2
3
The 'for' loop :-
This is probably the most useful and the most used loop in Java The syntax is slightly more complicated than that of 'while' or the 'do while' loop.

The general syntax can be defined as :-

```
for(<initial value>;<condition>;<increment>)
class

ForLoop
  {

     public static void main(String args[])

       {


           for(int i=1;i<=3;i++)
```

```
                {
                        System.out.println(i);
                    }
            }
    }
```

The output of the above code will be :-
1
2
3

Following are the Jump statements in Java:

- Break Statement
- Continue Statement
- Return Statement

Break Statement
The break statement is commonly used with the iteration statements such as for loop, while loop, and do-while loop.

Break statements in Java generally used in 3 ways:

- Exiting a Loop
- As a form of Goto
- In a Switch case

Continue Statement
Unlike break, the continue statement in java can only work inside loops. It doesn’t come out of the loop like a break. Instead, it forces the next iteration of the loop to bypass all the statements falling under it.

```
class continue_statement {
   public static void main(String[] args) {
      for(int i=0; i<=10;i++)
      {
         if (i<3)
         {
            Continue; //continues and goes to next iteration
         }
         System.out.println(i);
      }
   }
}
```

Return Statement
Return statements are the jump statements used inside methods (or functions) only. Any code in the method which is written after the return statement might be treated as unreachable by the compiler.
Then return statement terminates the execution of the current method and passes the control to the calling method.

It can exist in 2 forms:
Return with a Value: Here, the return statement returns a value to the calling method. The value can be a primitive data type or reference type.
Return without a Value: It simply transfers the control back to the calling method without any value.

```
class continue_statement {
   public static void main(String[] args) {
      int age = 12;
      System.out.println("Using Return");
      if (age<18)
         return; //terminates the method
      System.out.println("Will not get executed!");
   }
}
```

3. Give any five methods of StringBuffer class with syntax and example

length():

It defines the number of characters in the String.

```
import java.io. * ;
class Main {
  public static void main(String[] args) {
    StringBuffer str = new StringBuffer("ContentWriter");
    int len = str.length();
    System.out.println("Length : " + len);
  }
}
```

capacity():
It defines the capacity of the string occupied in the buffer. The capacity method also counts the reserved space occupied by the string.

```
import java.io. * ;
class Main {
  public static void main(String[] args) {
    StringBuffer str = new StringBuffer("ContentWriter");
    int cap = str.capacity();
    System.out.println("Capacity : " + cap);
  }
}
```

append():
The append() method is used to append the string or number at the end of the string.

```
import java.io.*;
class Main
{
public static void main(String[] args)
{
StringBuffer str = new StringBuffer("Tech");
str.append("Vidvan"); // appends a string in the previously defined string.
System.out.println(str);
str.append(0); // appends a number in the previously defined string.
System.out.println(str);
}
}
```

insert():
The insert() method is used to insert the string into the previously defined string at a particular indexed position. In the insert method as a parameter, we provide two-argument first is the index and second is the string.

```
public class StringBufferInsert {
  public static void main(String[] args) {
    StringBuffer stringName = new StringBuffer(" Welcome");
    System.out.println(stringName);
    stringName.insert(8, " to ");
    System.out.println(stringName);
    stringName.insert(12, "TechVidvan ");
    System.out.println(stringName);
    stringName.insert(22, " Tutorial ");
    System.out.println(stringName);
    stringName.insert(31, " of ");
    System.out.println(stringName);
    stringName.insert(35, "Java");
    System.out.println(stringName);
  }
}
```

reverse():
The reverse() method reverses all the characters of the object of the StringBuffer class. And, as an output, this method returns or gives the reversed String object.

```
import java.util. * ;
public class StringBufferReverse {
```

```
  public static void main(String[] args) {
    StringBuffer stringName = new StringBuffer("Welcome to TechVidvan");
    System.out.println("Original String: " + stringName);
    stringName.reverse();
    System.out.println("Reversed String: " + stringName);
  }
}
```

4. Explain the multiple usage of static keyword in Java with example program.

The static keyword in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

The static keyword is a non-access modifier in Java that is applicable for the following:

Blocks
Variables
Methods
Classes

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. For example, in the below java program, we are accessing static method m1() without creating any object of the Test class.

```
class Test
{
    // static variable
    static int a = 10;
    static int b;

    // static block
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    public static void main(String[] args)
    {
       System.out.println("from main");
       System.out.println("Value of a : "+a);
       System.out.println("Value of b : "+b);
    }
}
```

5. What is nested interface? Give an example to explain how it works.

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

- The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.
- Nested interfaces are declared static

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
```

```
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

6. Demonstrate with the help of a program how can you implement user defined exceptions in Java.

Following are a few of the reasons to use custom exceptions:

To catch and provide specific treatment to a subset of existing Java exceptions.
Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.
In order to create a custom exception, we need to extend the Exception class that belongs to java.lang package.

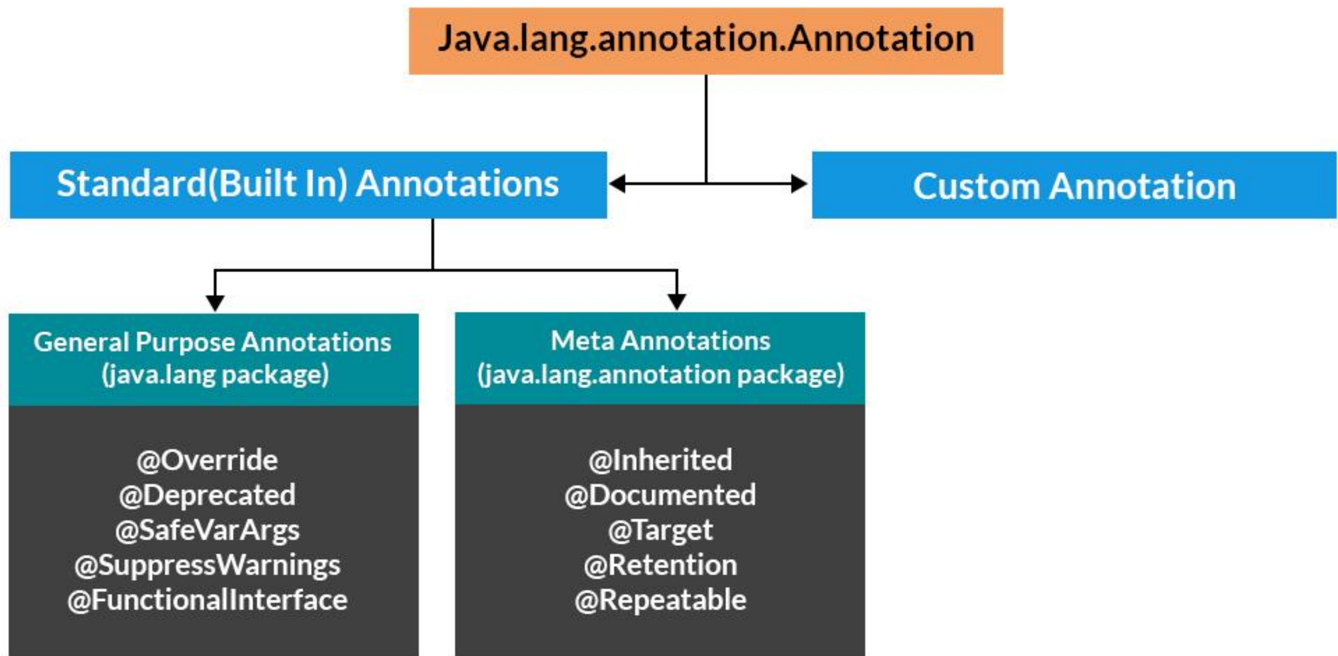
```
// A Class that represents use-defined exception

class MyException extends Exception {
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main {
    // Driver Program
    public static void main(String args[])
    {
        try {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex) {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

7. Briefly discuss annotations in Java

Annotations are used to provide supplemental information about a program.

- Annotations start with '@'.
- Annotations do not change the action of a compiled program.
- Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.

- Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
- Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.




```
// Java Program to Demonstrate that Annotations
// are Not Barely Comments

// Class 1
class Base {

    // Method
    public void display()
    {
        System.out.println("Base display()");
    }
}

// Class 2
// Main class
class Derived extends Base {

    // Overriding method as already up in above class
    @Override public void display(int x)
    {
        // Print statement when this method is called
        System.out.println("Derived display(int )");
    }

    // Method 2
    // Main driver method
    public static void main(String args[])
    {
        // Creating object of this class inside main()
        Derived obj = new Derived();
```

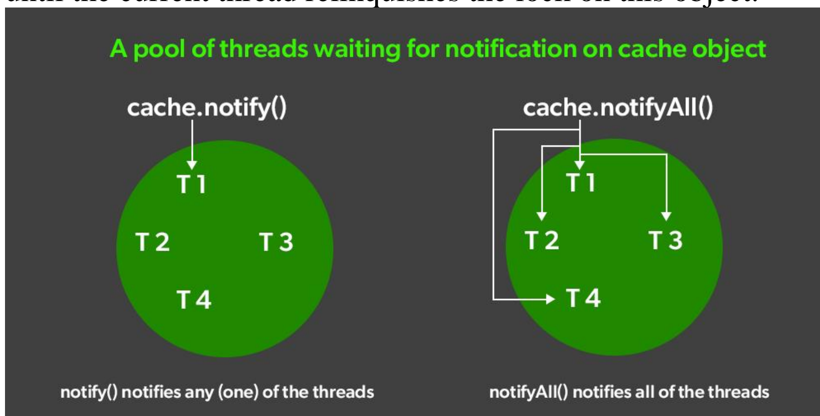
```
        // Calling display() method inside main()
        obj.display();
    }
}
```

8. Explain thread communication using notify(), wait() and notifyAll() methods

The notify() and notifyAll() methods with wait() methods are used for communication between the threads. A thread that goes into waiting for state by calling the wait() method will be in waiting for the state until any other thread calls either notify() or notifyAll() method on the same object.

notify(): The notify() method is defined in the Object class, which is Java's top-level class. It's used to wake up only one thread that's waiting for an object, and that thread then begins execution. The thread class notify() method is used to wake up a single thread.

notifyAll(): The notifyAll() wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.




```
// Java program to illustrate the
// behaviour of notify() method
class Geek1 extends Thread {
        public void run()
        {
                synchronized (this)
                {
                        System.out.println(
                                Thread.currentThread().getName()
                                + "...starts");
                        try {
                                this.wait();
                        }
                        catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        System.out.println(
                                Thread.currentThread().getName()
                                + "...notified");
                }
        }
}
class Geek2 extends Thread {
        Geek1 geeks1;

        Geek2(Geek1 geeks1){
```

```
        this.geeks1 = geeks1;
        }

        public void run()
        {
                synchronized (this.geeks1)
                {
                        System.out.println(
                                Thread.currentThread().getName()
                                + "...starts");

                        try {
                                this.geeks1.wait();
                        }
                        catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        System.out.println(
                                Thread.currentThread().getName()
                                + "...notified");
                }
        }
}
class Geek3 extends Thread {
        Geek1 geeks1;
        Geek3(Geek1 geeks1) { this.geeks1 = geeks1; }
        public void run()
        {
                synchronized (this.geeks1)
                {
                        System.out.println(
                                Thread.currentThread().getName()
                                + "...starts");
                        this.geeks1.notify();
                        System.out.println(
                                Thread.currentThread().getName()
                                + "...notified");
                }
        }
}
class MainClass {
        public static void main(String[] args)
                throws InterruptedException
        {

                Geek1 geeks1 = new Geek1();
                Geek2 geeks2 = new Geek2(geeks1);
                Geek3 geeks3 = new Geek3(geeks1);
                Thread t1 = new Thread(geeks1, "Thread-1");
                Thread t2 = new Thread(geeks2, "Thread-2");
                Thread t3 = new Thread(geeks3, "Thread-3");
                t1.start();
                t2.start();
                Thread.sleep(100);
                t3.start();
        }
}
```

Output:

Thread-1...start
Thread-2...starts
Thread-3...starts
Thread-3...notified
Thread-1...notified

```
// Java program to illustrate the
// behavior of notifyAll() method

class Geek1 extends Thread {
	public void run()
	{
		synchronized (this)
		{
			System.out.println(
				Thread.currentThread().getName()
				+ "...starts");
			try {
				this.wait();
			}
			catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(
				Thread.currentThread().getName()
				+ "...notified");
		}
	}
}
class Geek2 extends Thread {
	Geek1 geeks1;

	Geek2(Geek1 geeks1){
	this.geeks1 = geeks1;
	}

	public void run()
	{
		synchronized (this.geeks1)
		{
			System.out.println(
				Thread.currentThread().getName()
				+ "...starts");

			try {
				this.geeks1.wait();
			}
			catch (InterruptedException e) {
				e.printStackTrace();
			}
			System.out.println(
				Thread.currentThread().getName()
				+ "...notified");
		}
```

```
	}
}
class Geek3 extends Thread {
	Geek1 geeks1;
	Geek3(Geek1 geeks1) { this.geeks1 = geeks1; }
	public void run()
	{
		synchronized (this.geeks1)
		{
			System.out.println(
				Thread.currentThread().getName()
				+ "...starts");

			this.geeks1.notifyAll();
			System.out.println(
				Thread.currentThread().getName()
				+ "...notified");
		}
	}
}
class MainClass {
	public static void main(String[] args)
		throws InterruptedException
	{

		Geek1 geeks1 = new Geek1();
		Geek2 geeks2 = new Geek2(geeks1);
		Geek3 geeks3 = new Geek3(geeks1);
		Thread t1 = new Thread(geeks1, "Thread-1");
		Thread t2 = new Thread(geeks2, "Thread-2");
		Thread t3 = new Thread(geeks3, "Thread-3");
		t1.start();
		t2.start();
		Thread.sleep(100);
		t3.start();
	}
}
```

Output

```
Thread-1...starts
Thread-2...starts
Thread-3...starts
Thread-3...notified
Thread-1...notified
Thread-2...notified
```

9. Write short notes about: InetAddress class, URL Connection class.

Java **InetAddress class** represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name for example www.javatpoint.com, www.google.com, www.facebook.com, etc.

An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name. There are two types of addresses: Unicast and Multicast. The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	It returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	It returns the instance of InetAdddress containing local host name and address.
public String getHostName()	It returns the host name of the IP address.
public String getHostAddress()	It returns the IP address in string format.

```
import java.io.*;
import java.net.*;
public class InetDemo{
public static void main(String[] args){
try{
InetAddress ip=InetAddress.getByName("www.javatpoint.com");
System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}
}
}
```

The Java **URLConnection class** represents a communication link between the URL and the application. It can be used to read and write data to the specified resource referred by the URL.

Features of URLConnection class:

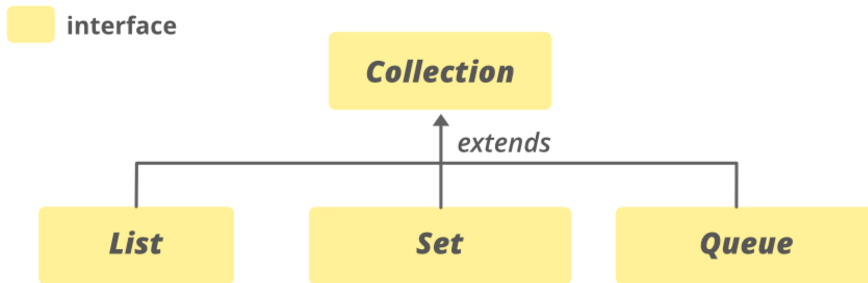
- URLConnection is an abstract class. The two subclasses HttpURLConnection and JarURLConnection makes the connetion between the client Java program and URL resource on the internet.
- With the help of URLConnection class, a user can read and write to and from any resource referenced by an URL object.
- Once a connection is established and the Java program has an URLConnection object, we can use it to read or write or get further information like content length, etc.

```
import java.io.*;
import java.net.*;
public class URLConnectionExample {
public static void main(String[] args){
try{
URL url=new URL("http://www.javatpoint.com/java-tutorial");
URLConnection urlcon=url.openConnection();
InputStream stream=urlcon.getInputStream();
int i;
while((i=stream.read())!=-1){
System.out.print((char)i);
}
}catch(Exception e){System.out.println(e);}
}
}
```

10. Discuss the classes and interfaces that inherit properties of collection interface.

The Collection interface is a member of the Java Collections Framework. It is a part of java.util package. It is one of the root interfaces of the Collection Hierarchy. The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like List, Queue, and Set.

For Example, the HashSet class implements the Set interface which is a subinterface of the Collection interface. If a collection implementation doesn't implement a particular operation, it should define the corresponding method to throw UnsupportedOperationException.



SubInterfaces of Collection Interface

All the Classes of the Collection Framework implement the subInterfaces of the Collection Interface. All the methods of Collection interfaces are also contained in it's subinterfaces. These subInterfaces are sometimes called as Collection Types or SubTypes of Collection. These include the following:

List: This is a child interface of the collection interface. This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it. This list interface is implemented by various classes like ArrayList, Vector, Stack, etc. Since all the subclasses implement the list, we can instantiate a list object with any of these classes. For example,

```
List <T> al = new ArrayList<> ();
List <T> ll = new LinkedList<> ();
List <T> v = new Vector<> ();
```
Where T is the type of the object

Set: A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc. Since all the subclasses implement the set, we can instantiate a set object with any of these classes. For example,

```
Set<T> hs = new HashSet<> ();
Set<T> lhs = new LinkedHashSet<> ();
Set<T> ts = new TreeSet<> ();
```
Where T is the type of the object.

SortedSet: This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted. The class which implements this interface is TreeSet. Since this class implements the SortedSet, we can instantiate a SortedSet object with this class. For example,

```
SortedSet<T> ts = new TreeSet<> ();
```
Where T is the type of the object.

Queue: As the name suggests, a queue interface maintains the FIFO (First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket. There are various classes like PriorityQueue, Deque, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes. For example,

```
Queue <T> pq = new PriorityQueue<> ();
Queue <T> ad = new ArrayDeque<> ();
```
Where T is the type of the object.

Deque: This is a very slight variation of the queue data structure. Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue. This interface extends the queue interface. The class which implements this interface is ArrayDeque. Since this class implements the deque, we can instantiate a deque object with this class. For example,

```
Deque<T> ad = new ArrayDeque<> ();
```
Where T is the type of the object.

```
// Java program to illustrate Collection interface

import java.io.*;
import java.util.*;

public class CollectionDemo {
	public static void main(String args[])
	{

		// creating an empty LinkedList
		Collection<String> list = new LinkedList<String>();

		// use add() method to add elements in the list
		list.add("Geeks");
		list.add("for");
		list.add("Geeks");

		// Output the present list
		System.out.println("The list is: " + list);

		// Adding new elements to the end
		list.add("Last");
		list.add("Element");

		// printing the new list
		System.out.println("The new List is: " + list);
	}
}
```