Unit–5: File Handling and Collections Framework

Introduction:

When we use variables and array for storing data inside the programs,We face two Problems: 1) The data is lost either when a variable goes out of scope or when the program is terminated. The storage is temporary. 2) It is difficult to handle large volumes of data using variables and arrays.

We can overcome these problems by storing data into secondary storage devices. We can store data using concept of files. Data stored in file is often called persistent data.

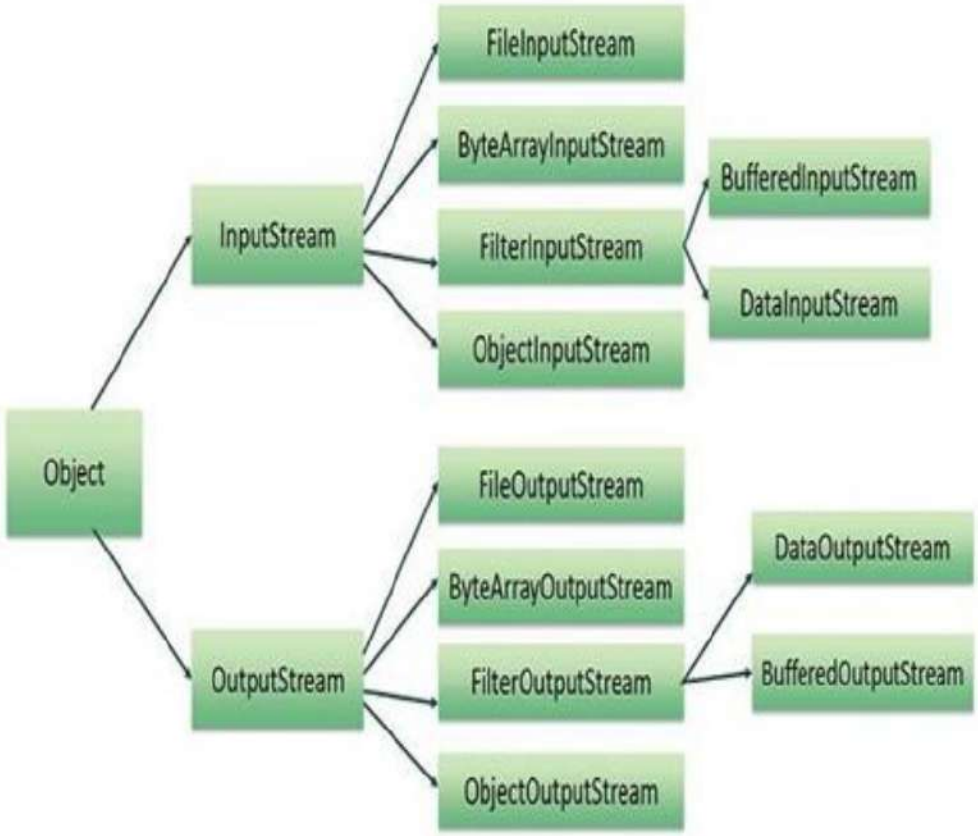
- A file is collection of related records placed area on the disk. A record is composed of several fields.
- Field is a group of characters. Storing and managing data using file is known as file processing which includes tasks such as creating files, updating files and manipulation of data.
- Reading and writing of data in a file can be done at the level of bytes or characters or fields depending on the requirement of application.java provide capabilities to read and write class object directly.
- The process of reading and writing objects is called **serialization.**

❖ Concept of stream

- In file processing, input refers to the flow of data into a program and output means the flow of data out of a program.
- Input to a program may come from the keyboard, mouse, memory, disk a network or another program.
- Output from a program may go to the screen, printer, memory disk, network or another program.
- Input and output share certain common characteristics like unidirectional movement of data, treating data as a sequence of bytes or characters and support to sequential access to data.
- Java uses concept of stream to represent ordered sequence of data, a common characteristics shared by all input/output devices.
- A stream in java is a path along which data flows (like pipe along which water flows). It has source (of data) and destination (for that data).Both the source and destination may be physical devices or programs or other streams in same program.
- The concept of sending data from one stream to another has made streams in java a powerful tool for file processing. We can build a complex file processing sequence using a series of simple stream operation. This feature is used to filter data along the pipeline of streams so that we obtain data in desired format.
- A stream can be defined as a sequence of data. There are two kinds of Streams:

- **InputStream:** The InputStream is used to read data from a source.
- **OutputStream:** The OutputStream is used for writing data to a destination.

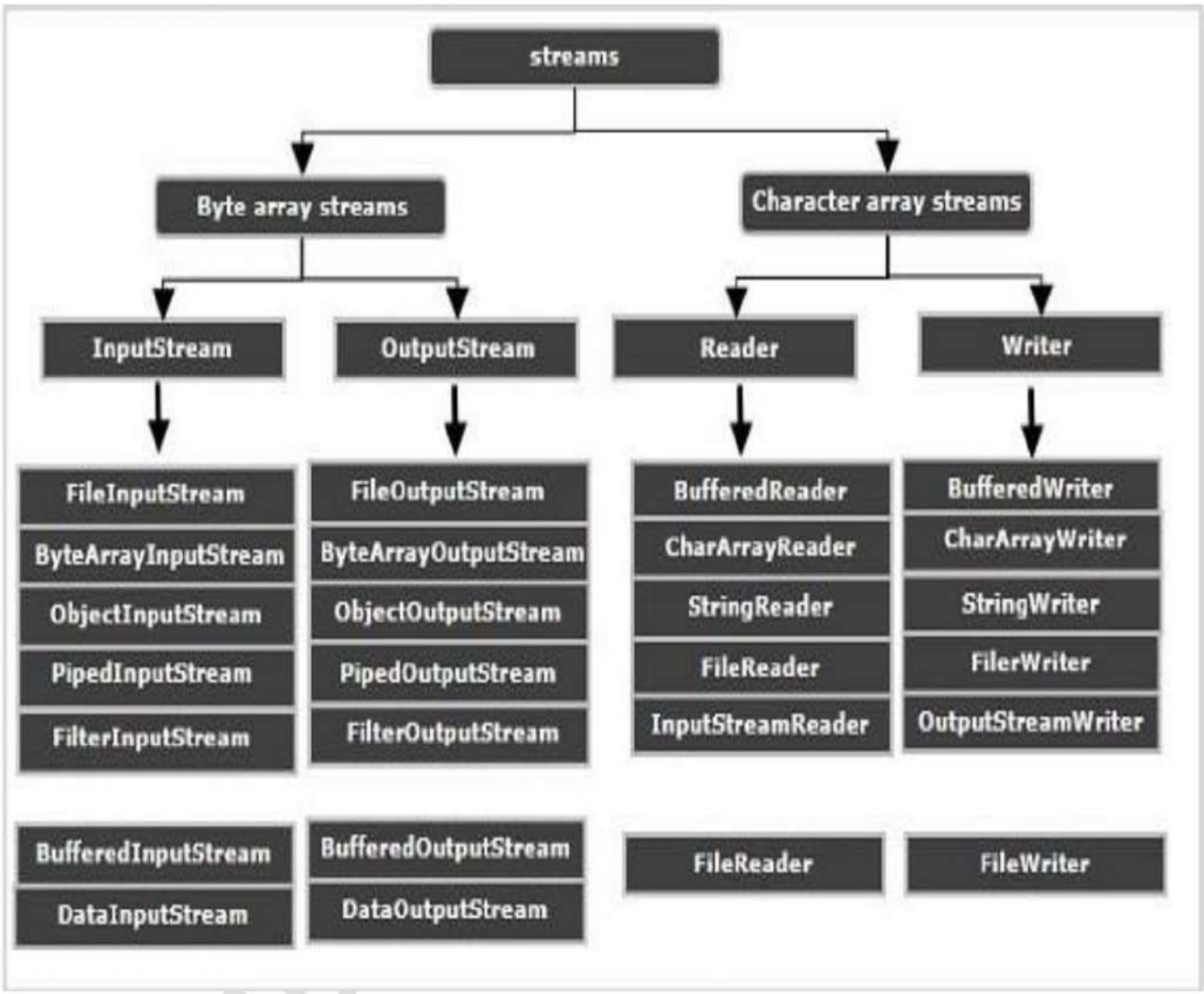
Here is a hierarchy of classes to deal with Input and Output streams.



- **Types of Streams:**

Depending upon the data a stream holds, it can be classified into:

- **Byte Stream**
- **Character Stream**



1. Byte Stream:

- Byte stream is used to read and write a single byte (8 bits) of data.
- All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

- **Java InputStream Class:**

  - Input stream classes that are used to read 8-bit bytes include super class known as InputStream and a number of subclasses for supporting various input-related functions.
  - The super class InputStream is an abstract class, and, therefore, we cannot create instances of this class.
  - Rahter,we must use the subclasses that inherit from this class.

- The InputStream class defines method for performing input functions such as:
  - Reading bytes,
  - Closing streams,
  - Marking position in streams,
  - Skipping ahead in a stream,
  - Finding the number of bytes in a stream

- **InputStream Methods:**

1. **read() :** Reads a byte from the input stream
2. **read(byte b[]) :** Reads an array of bytes into b
3. **read(byte b[],int n,int m) :** Reads m bytes into b starting from nth byte
4. **available() :** Gives number of bytes available in the input
5. **skip(n) :** Skips over n bytes from the input stream
6. **reset() :** Goes back to the beginning of the stream
7. **close() :** Closes the input stream.

- **Java OutputStream Class:**

- Output stream classes are derived from the base class OutputStream as shown in figure. Like InputStream, the OutputStream is an abstract class and therefore we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations.
- The OutputStream includes methods that are designed to perform the following task:
  - Writing bytes
  - Closing stream
  - Flushing stream

- **OutputStream Methods:**

1. **write()** : Writes a byte to the output stream
2. **write(byte b[])** : Writes all bytes in the array b to the output stream
3. **write(byte b[],int n,int m):** Writes m bytes from array b starting from nth byte
4. **close()** : Close the output stream 5 flush() Flushes the output stream

2. Character stream classes:

- Character stream is used to read and write a single character of data.
- All the character stream classes are derived from base abstract classes Reader and Writer.

- **Java Reader Class:**

  - Reader stream classes are designed to read character from the files. Reader class is the base class for all other classes in this group as shown in figure.
  - These classes are functionally very similar to the input stream classes, except input stream use bytes as their fundamental unit of information, while reader stream use characters.
  - The Reader class contains method they are identical to those available in the InputStream class, except Reader is designed to handle characters. Therefore, reader classes can perform all the functions implemented by the input stream classes.

- **Java Writer Class:**

  - Like output stream classes, the writer stream classes are designed to perform all output operations on files.
  - Only difference is that while output stream classes are designed to write bytes, the writer stream classes are designed to write characters.
  - The Writer class is an abstract class which acts as a base class for all other writer stream classes.
  - This base class provides support for all output operations by defining method that are identical to those in OutputStream class.

Using the file class:

- The java.io package includes a class known as the File class that provides support for creating files and directories. The class includes several constructors for instantiating the File objects.
- File class provides methods for operations like:
  - Creating a file
  - Opening a file
  - Closing a file
  - Deleting a file
  - Getting the name of a file
  - Getting the size of a file
  - Checking the existence of a file
  - Renaming a file
  - Checking whether the file is writable
  - Checking whether the file is readable
  - Input/output Exceptions

- When creating files and performing I/O operations on them, the system may generate I/O related exceptions.
- The basic I/O related exception classes and their functions:

	I/O exception class:	Function
1	EOFException	Signals that an end of file or end of stream has been reached unexpectedly during input
2	FileNotFoundException	Informs that a file could not be found
3	InteruuptedIOException	Warns that an I/O operations has been interrupted
4	IOException	Signals that an I/O exception of some sort has occurred

1. Creation of files:

If we want to create and use a disk file, we need to decide the following about the file and its intended purpose:

- Suitable name for the file.
- Data type to be stored
- Purpose (reading, writing, or updating)
- Method of creating the file

2. Reading/writing bytes:

- We have used FileReader and FileWriter classes to read and write 16-bit characters. Most file systems use only 8-bit bytes. As pointed out earlier, java i/o system provides a number of classes for handling bytes are FileInputStream and FileOutputStream classes. We can use them in place of FileReader and file writer.

Example-1: Write a program which creates file and writes byte into that file.

```
import java.io.*;

public class WriteByte

{

public static void main(String args[])

 {

        File fl=new File("input.txt"); \\ to create new file
```

```
        FileOutputStream outfile = null;
        byte cities[] = {'I','L','O','V','E','J','A','V','A'};
try
{
        .write(cities);
}
catch(IOException e)
{
        System.out.println(e);
        System.exit(-1);
}
System.out.println ("Write Byte");
System.out.println ("Thank You...!!!");
}
}
```

Output:

Write Byte

Thank You!!!

Example-2: Write a program which reads byte from file.

```
import java.io.*;
public class ReadingByte
{
public static void main(String args[])
```

```
{
        FileInputStream infile = null;
        int b;
try
        {
        infile = new FileInputStream("in.txt");
        while((b = infile.read()) != -1)
        {
                System.out.println((char)b);
        }
        infile.close();
}
catch(IOException e)
{
        System.out.println("Sorry..!! File Not Found...!!!");
}
}
}
```

Output :

I LOVE JAVA

3. Reading/writing characters

- As pointed out earlier, subclass of Reader and Writer implement streams that can handle characters. The two subclasses used for handling characters in files are FileReader and FileWriter.

Example-3 : Write a program which creates file and writes character into that file.

```
import java.io.*;
class CharacterWrite
{
public static void main(String args[])
{
        File fl=new File("input1.txt");
        FileWriter fw = null;
try
{       fw=new FileWriter(fl);
        fw.write("ahmedabad \n");
        fw.write(" baroda \n");
        fw.close();
}
catch(FileNotFoundException e)
{
        System.out.println("Sorry..!! File Not Found...!!!");
}
catch(IOException e)
{System.out.println(e.getMessage());
}
System.out.println(" write operation done!!");
}
}
```

Output:

write operation done

Example-4: Write a program which reads character from file.

```
import java.io.*;
class Readchar
{
public static void main(String args[])
{
        FileReader fr =null;
try
{       fr = new FileReader("input.txt");
        int ch;
        while((ch = fr.read()) != -1)
        {
                System.out.print((char)ch);
        }
        System.out.println("Reading complete");
        fr.close();
}
catch(FileNotFoundException e)
{
        System.out.println("Sorry..!! File Not Found...!!!");
}
```

```
catch(IOException e)
{
        System.out.println(e.getMessage());
}
}
}
```

Output:

```
ahmedabad
baroda
Reading complete
```

Example -5: Write a program to read one byte at a time from a file and copy it into another

file immediately.

```
import java.io.*;
class CopyByte
{
public static void main(String args[])
{
try
{       byte b=0;
        FileInputStream infile = new FileInputStream("in.txt");
        FileOutputStream outfile = new FileOutputStream("out.txt");
        while(byteread != -1)
        {
```

```
        b = (byte)infile.read();
        outfile.write(b);
        }
        System.out.println("Byte Copied From in.txt to out.txt FIle ");
}
catch(FileNotFoundException e)
{
        System.out.println("Sorry..!! File Not Found...!!!");
}
catch(IOException e)
{
        System.out.println(e.getMessage());
}
}
}
```

Output :

Byte Copied From in.txt to out.txt File

Example -6: Write a program to merge two files in third file.

```
import java.io.*;
class FileMergeDemo
{
public static void main(String args[])
{
```

```
try
{
        FileInputStream file1 = new FileInputStream ("File1.txt");
        FileInputStream file2 = new FileInputStream ("File2.txt");
        SequenceInputStream file3 = new SequenceInputStream (file1, file2);
        BufferedInputStream br1 = new BufferedInputStream (file3);
        BufferedOutputStream br2 = new BufferedOutputStream (System. out);
        int ch;
        while((ch = br1.read())!=-1)
        {
                br2.write((char)ch);
        }
        br1.close();
        br2.close();
        file1.close();
        file2.close();
        System.out.println("Merge Two File Sucessfully ");
}
catch(IOException e)
{
        System.out.println("Sorry..!! File Not Found...!!!");
}
}
}
```

Output:

Vpmpldrp

Example -7: Write a program which creates file and perform read and write operations on that file.

```
import java.io.*;
public class p30
{
public static void main(String args[])
 {
        File fl=new File("test.txt");

        FileOutputStream outfile = null;
         byte cities[] = {'I','L','O','V','E','J','A','V','A'};
         try
         {
                outfile = new FileOutputStream(fl);
                outfile.write(cities);
         }
        catch(Exception e)
        {
                System.out.println(e);
                 System.exit(-1);
        }
System.out.println("Write operation successfully completed!!!");
```

```
System.out.println("Thank You...!!!");
 FileInputStream infile = null;
 int b;
 try
 {
 infile = new FileInputStream("vpmp.txt");
 while((b = infile.read()) != -1)
 {
 System.out.println((char)b);
 }
 infile.close();
 }
 catch(Exception e)
 {
 System.out.println("Sorry..!! File Not Found...!!!");
 }
System.out.println("Reading operation successfully completed!!!");
 }
}
```

❖ Collections Framework overview

- Any group of individual objects which are represented as a single unit is known as the collection of the objects.
- In Java, a separate framework named the "Collection Framework" has been defined in JDK 1.2 which holds all the collection classes and interface in it.
- The Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main "root" interfaces of Java collection classes.

➢ What is a Framework?

A framework is a set of classes and interfaces which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework.

➢ Need for a Separate Collection Framework:

Before the Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors, or Hashtables. All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them. And also, it is very difficult for the users to remember all the different methods, syntax, and constructors present in every collection class.

Let's understand this with an example of adding an element in a Hashtables and a vector.

Example1: Java program to demonstrate why collection framework was needed.

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

class CollectionDemo {

   public static void main(String[] args)
   {
      // Creating instances of the array, vector and hashtable
      int arr[] = new int[] { 1, 2, 3, 4 };
      Vector<Integer> v = new Vector();
      Hashtable<Integer, String> h = new Hashtable ();

      // Adding the elements into the vector
      v.addElement (1);
      v.addElement (2);
```

```
        // Adding the element into the hashtable
        h.put (1, "vpmpce");
        h.put (2, "nehapatel");

        // Accessing the first element of the array, vector and hashtable
        System.out.println (arr [0]);
        System.out.println (v.elementAt (0));
        System.out.println (h.get (1));

    }
}
```

Output:

1

1

vpmpce

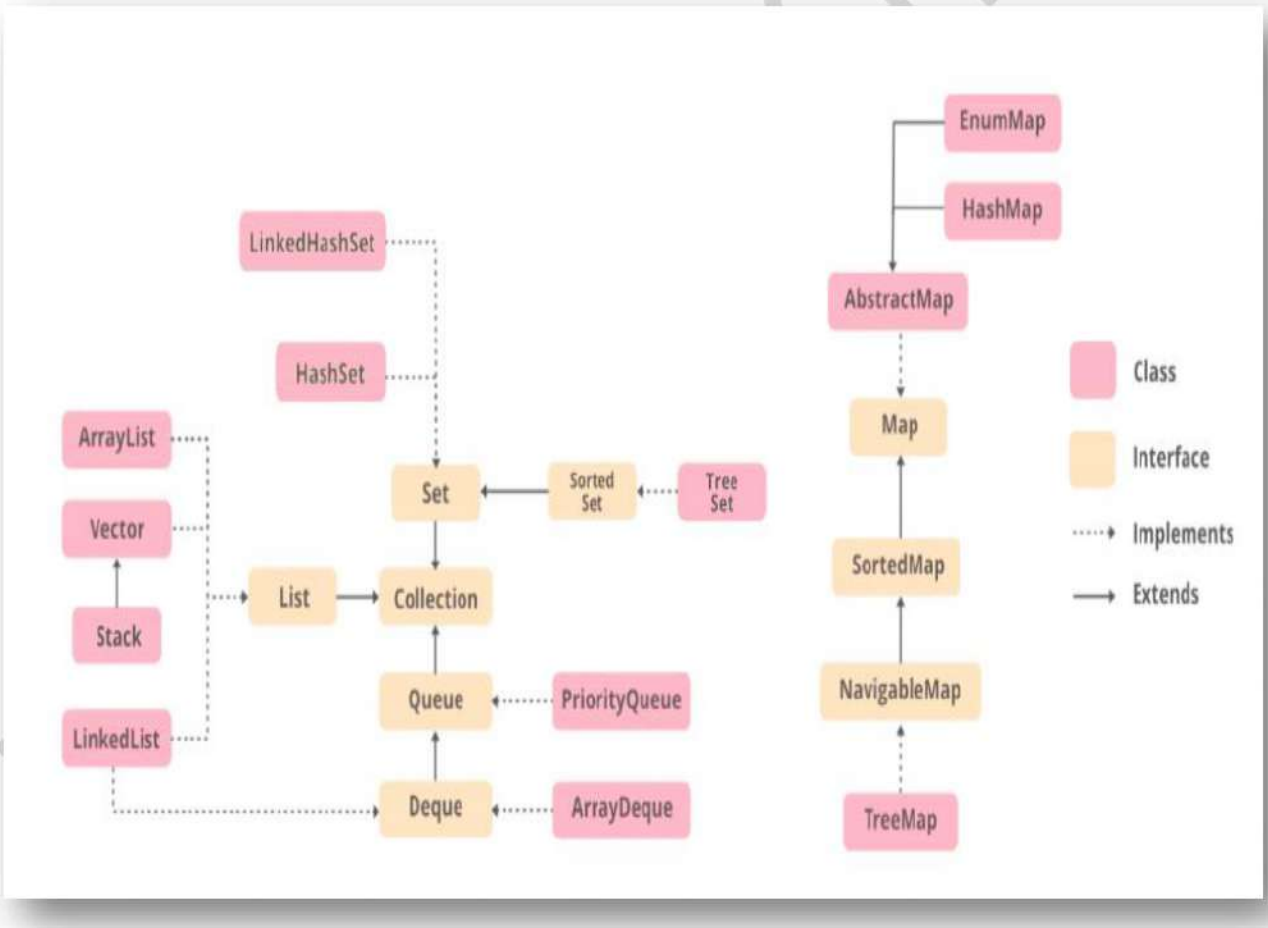
- As we can observe, none of these collections (Array, Vector, or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections.
- Another drawback is that most of the 'Vector' methods are final, meaning we cannot extend the 'Vector' class to implement a similar kind of Collection.
- Therefore, Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2 post which both, legacy Vectors and Hashtables were modified to conform to the Collection Framework.

• Advantages of the Collection Framework:

- **Consistent API:** The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.
- **Reduces programming effort:** A programmer doesn't have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.
- **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms because in this case, the programmer need not think of the best implementation of a specific data structure. He can simply use the best implementation to drastically boost the performance of his algorithm/program.

- **Hierarchy of the Collection Framework:**

- The utility package, (java.util) contains all the classes and interfaces that are required by the collection framework.
- The collection framework contains an interface named an iterable interface which provides the iterator to iterate through all the collections.
- This interface is extended by the main collection interface which acts as a root for the collection framework.
- All the collections extend this collection interface thereby extending the properties of the iterator and the methods of this interface.
- The following figure illustrates the hierarchy of the collection framework.



❖ **The classes which implement the List interface are as follows:**

A. ArrayList:

- ArrayList provides us with dynamic arrays in Java.
- Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
- The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list. ArrayList cannot be used for primitive types, like int, char, etc.We will need a wrapper class for such cases.

Example: Java program to demonstrate the working of ArrayList.

```
import java.io.*;
import java.util.*;
class GFG
{     // Main Method
   public static void main(String[] args)
   {
      // Declaring the ArrayList with initial size n
      ArrayList<Integer> al = new ArrayList<Integer>();
       // Appending new elements at the end of the list
      for (int i = 1; i <= 5; i++)
         al.add(i);
      System.out.println(al);
      al.remove(3);
      System.out.println(al);
     for (int i = 0; i < al.size(); i++)
                System.out.print(al.get(i) + " ");
   }
}
```

Output:

```
[1, 2, 3, 4, 5]

[1, 2, 3, 5]

1 2 3 5
```

B. Linked List:

- Linked List class is an implementation of the Linked List which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.
- The elements are linked using pointers and addresses. Each element is known as a node.

Example: Java program to demonstrate the working of Linked List.

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

class GFG {

   public static void main(String[] args)
   {

      LinkedList<Integer> ll = new LinkedList<Integer>();

      // Appending new elements at the end of the list
      for (int i = 1; i <= 5; i++)
         ll.add(i);

      System.out.println(ll);

      ll.remove(3);

      System.out.println(ll);

      // Printing elements one by one
      for (int i = 0; i < ll.size(); i++)
         System.out.print(ll.get(i) + " ");
   }
}
```

Output:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

- **The following are the classes that implement the Set interface:**

A. HashSet:

- The HashSet class is an inherent implementation of the hash table data structure.
- The objects that we insert into the HashSet **do not guarantee** to be inserted in the same order. The objects are inserted based on their hashcode.
- This class also allows the insertion of NULL elements.

Example: Java program to demonstrate the working of a HashSet.

```
import java.util.*;
public class HashSetDemo {
      public static void main(String args[])
   {  HashSet<String> hs = new HashSet<String>();
      hs.add("VPMP ");
      hs.add("Polytechnic");
      hs.add("Gandhinagar");
      hs.add("Welcome To");
      hs.add("CE Dept.");       // Traversing elements
      Iterator<String> itr = hs.iterator();
      while (itr.hasNext()) {
         System.out.println(itr.next());
      }
   }
}
```

Output:

```
Gandhinagar
Polytechnic
VPMP
CE Dept.
Welcome To
```

❖ Map class : HashMap

- **Map Interface:**

- A map is a data structure that supports the key-value pair for mapping the data. This interface doesn't support duplicate keys because the same key cannot have multiple mappings,
- However it allows duplicate values in different keys. A map is useful if there is data and we wish to perform operations on the basis of the key.

- This map interface is implemented by various classes like HashMap, TreeMap, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes. For example,

  Map<T> hm = new HashMap<> ();
  Map<T> tm = new TreeMap<> ();

  Where T is the type of the object.

- **The frequently used implementation of a Map interface is a HashMap.**

HashMap: HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs.

- To access a value in a HashMap, we must know its key.
- HashMap uses a technique called Hashing.
- **Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster.**
- HashSet also uses HashMap internally.

Example: Java program to demonstrate the working of a HashMap

```
import java.util.*;
public class HashMapDemo
 {       public static void main(String args[])
   {       // Creating HashMap and adding elements
      HashMap<Integer, String> hm = new HashMap<Integer, String>();
      hm.put(1, "AOOP");
      hm.put(2, "VPMP_CE");
      hm.put(3, "NEHA_PATEL");
      // Finding the value for a key
      System.out.println("Value for 1 is " + hm.get(1));
      // Traversing through the HashMap
      for (Map.Entry<Integer, String> e : hm.entrySet())
         System.out.println(e.getKey() + " " + e.getValue());
   }}
```

Output:

```
Value for 1 is AOOP
1 AOOP
2 VPMP_CE
3 NEHA_PATEL
```

❖ For-each loop in Java

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5.

- It starts with the keyword **for** like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.

- It's commonly used to iterate over an array or a Collections class (eg, ArrayList)

Syntax:

```
for (type var : array)

{

   statements using var;

}
```

Example: Simple program with for each loop.

```
import java.io.*;
class Easy
{
   public static void main(String[] args)
   {
      int ar[] = { 10, 50, 60, 80, 90 };
      for (int element : ar)
         System.out.print(element + " ");
   }
}
```

Output

10 50 60 80 90

- **The above syntax is equivalent to:**

```
for (int i=0; i<arr.length; i++)

{    type var = arr[i];

   statements using var;

}
```

EXAMPLE: Java program to illustrate for-each loop

```
class For_Each
{   public static void main(String[] arg)
   {
        int[] marks = { 125, 132, 95, 116, 110 };
        int highest_marks = maximum(marks);
        System.out.println("The highest score is " + highest_marks);
   }
   public static int maximum(int[] numbers)
   {
      int maxSoFar = numbers[0];
       for (int num : numbers)
       {
          if (num > maxSoFar)
          {             maxSoFar = num;
          }
       }
   return maxSoFar;
   }
}
```

Output

The highest score is 132

- **Limitations of for-each loop** :

1. For-each loops are **not appropriate when you want to modify the array**:

```
for (int num : marks)
{   // only changes num, not the array element
   num = num*2;
}
```

2. For-each loops **do not keep track of index**. So we can not obtain array index using For-Each loop

```
for (int num : numbers)
{
   if (num == target)
   {       return ???;   // do not know the index of num
   }
}
```

3. For-each **only iterates forward over the array in single steps**

```
// cannot be converted to a for-each loop
for (int i=numbers.length-1; i>0; i--)
{
    System.out.println(numbers[i]);
}
```

4. For-each **cannot process two decision making statements** at once

```
// cannot be easily converted to a for-each loop
for (int i=0; i<numbers.length; i++)
{   if (numbers[i] == arr[i])
   { ...
   }
}
```

5. For-each also has some **performance overhead** over simple iteration.