UNIT–5

FILE HANDLING AND COLLECTIONS FRAMEWORK

File handling in Java involves reading from and writing to files using classes provided by the Java I/O (Input/Output) package. This package includes a set of classes and interfaces that allow you to perform file-related operations such as creating, reading, writing, deleting, and renaming files.

Here are the basic steps for performing file handling in Java:

1. Opening a file
2. Reading from a file
3. Writing to a file
4. Closing a file

To perform various file handling operations Java provides various stream classes.Java.io package contains all the classes required for input and output operations.Streams are ordered sequences of data that have: A source (input streams) or A destination (output stream).

STREAM CLASSES

Stream

A stream is a sequence of data. In Java a stream is composed of bytes. Streams are used for sending and receiving information in Java programs.

Three streams are created for us automatically

1. System.out: standard output stream
2. System.in: standard input stream
3. System.err: standard error

Stream Types

Streams are also classified in to two categories

1. Character streams
2. Byte streams

Input streams and Output streams are of the type Byte streams.

Reader and Writer streams are of the type Character streams.

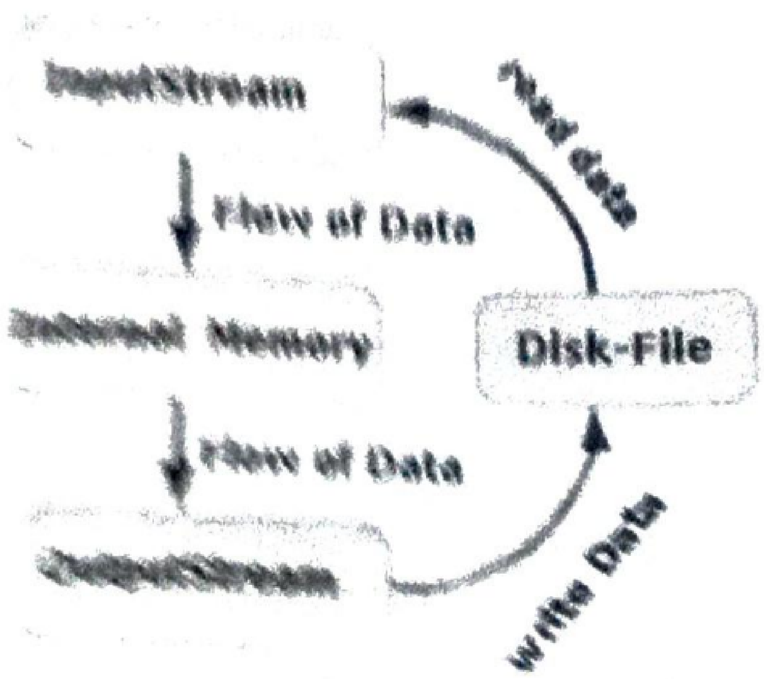
Stream Types

	Byte Streams	Character Streams
Source Streams	InputStream	Reader
Sink Streams	OutputStream	Writer

Each type of stream results in four actual streams: Byte based input, Byte based output, Character based input (Reader), and Character based output (Writer).

How Files and Streams Work?

- Java uses streams to handle I/O operations through which the data is flowed from one location to another.
- For example, an InputStream can flow the data from a disk file to the internal memory and an OutputStream can flow the data from the internal memory to a disk file.
- The disk-file may be a text file or a binary file.

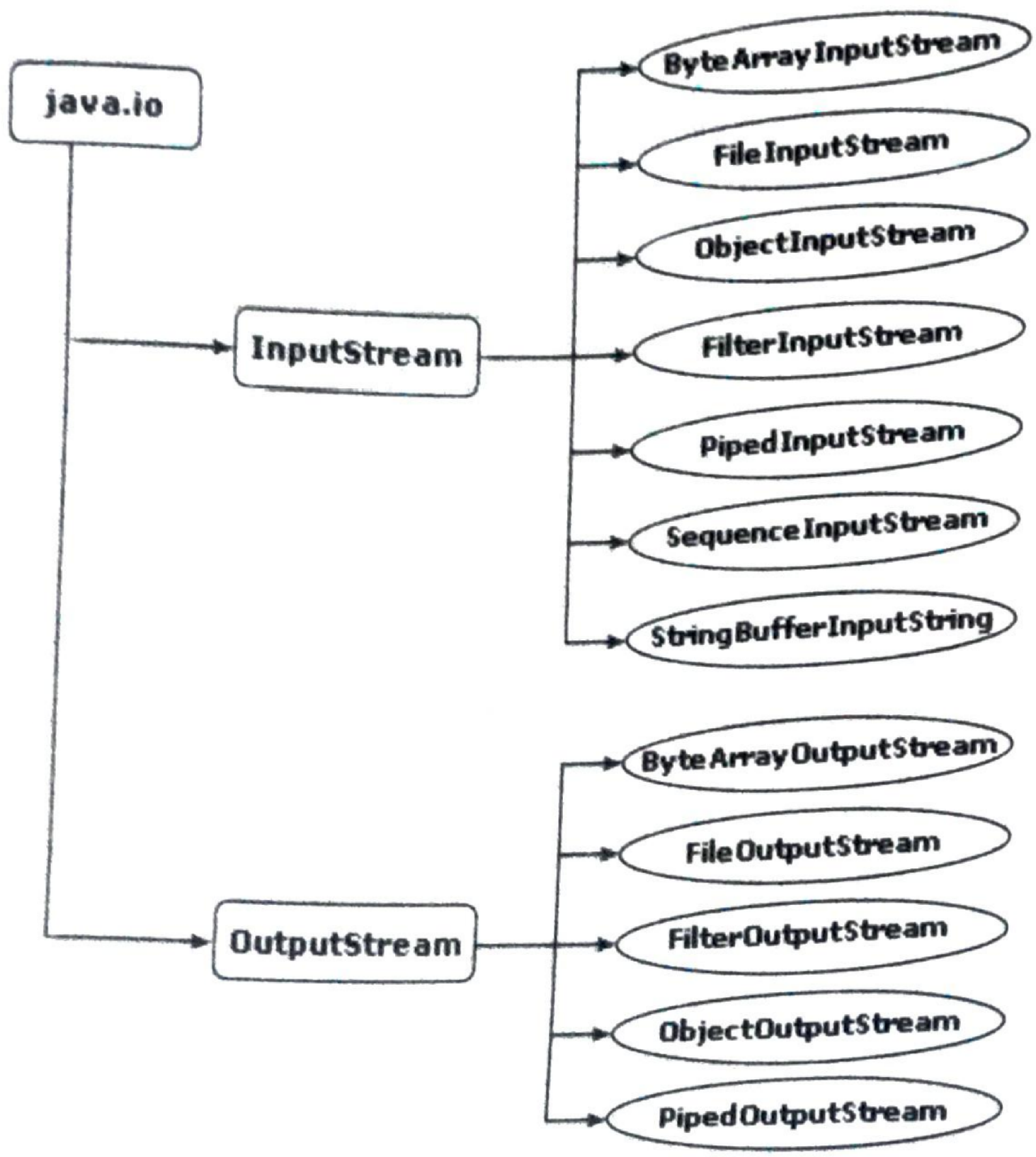


FILES AND STREAM INTERACTION

- When we work with a text file, we use a character stream where one character is treated as per byte on disk.
- When we work with a binary file, we use a binary stream.
- The working process of the I/O streams can be shown in the given diagram.
- User can read from input stream but can't write to it.
- User can write to output stream but can't read from it.
- All the classes required handling input and output are given in Java.io package.

STREAM CLASS HIERARCHY

The Java.io package can be categories along with its stream classes in a hierarchy structure shown below:



JAVA STREAM CLASSES

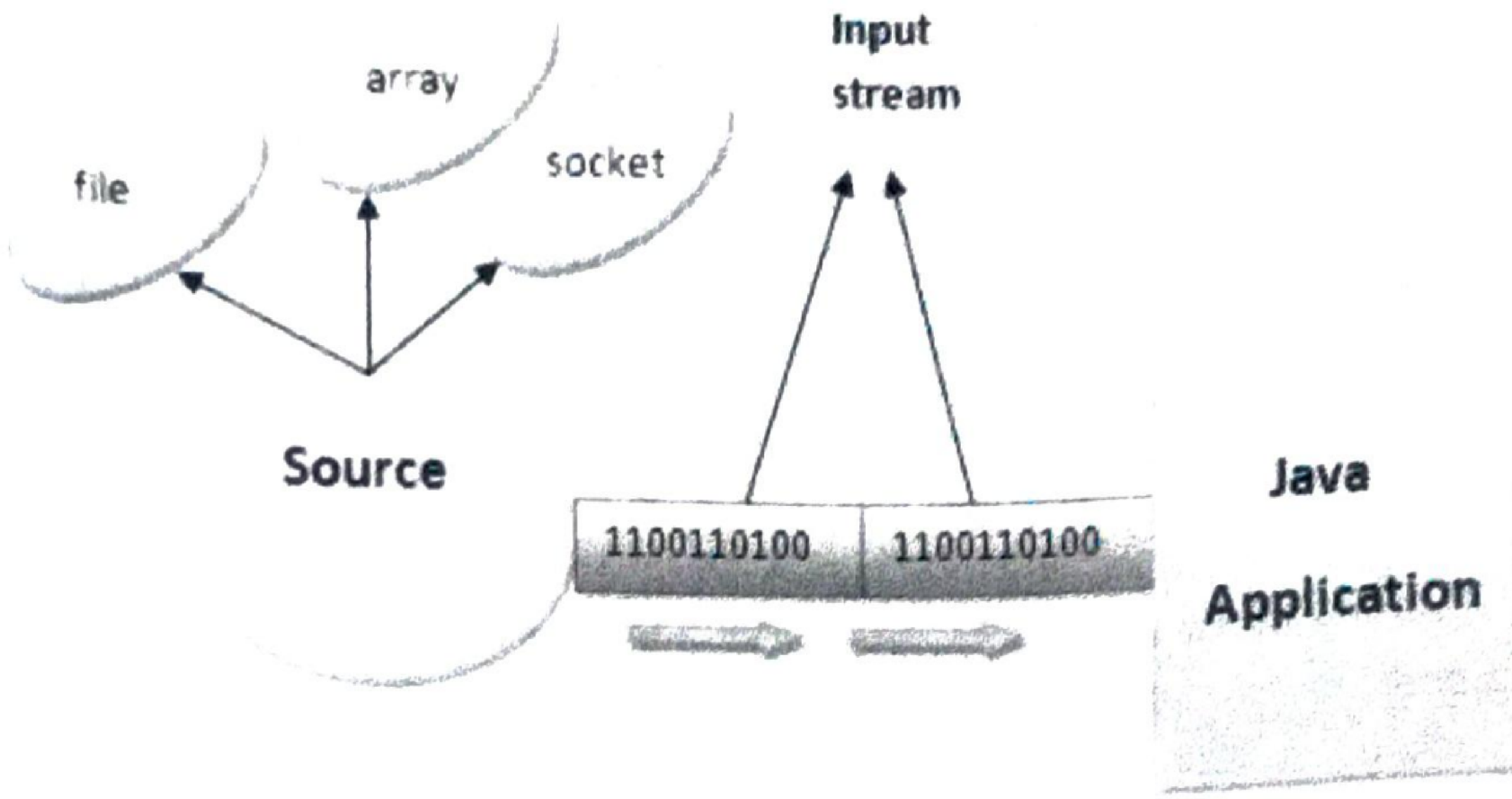
USEFUL I/O CLASSES

The Java Input/output (I/O) is a part of Java.io package. The Java.io package contains a relatively large number of classes that support input and output operations. The InputStream and OutputStream are central classes in the package which are used for reading from and writing to byte streams, respectively.

InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

An input stream is automatically opened when you create it.



INPUTSTREAM

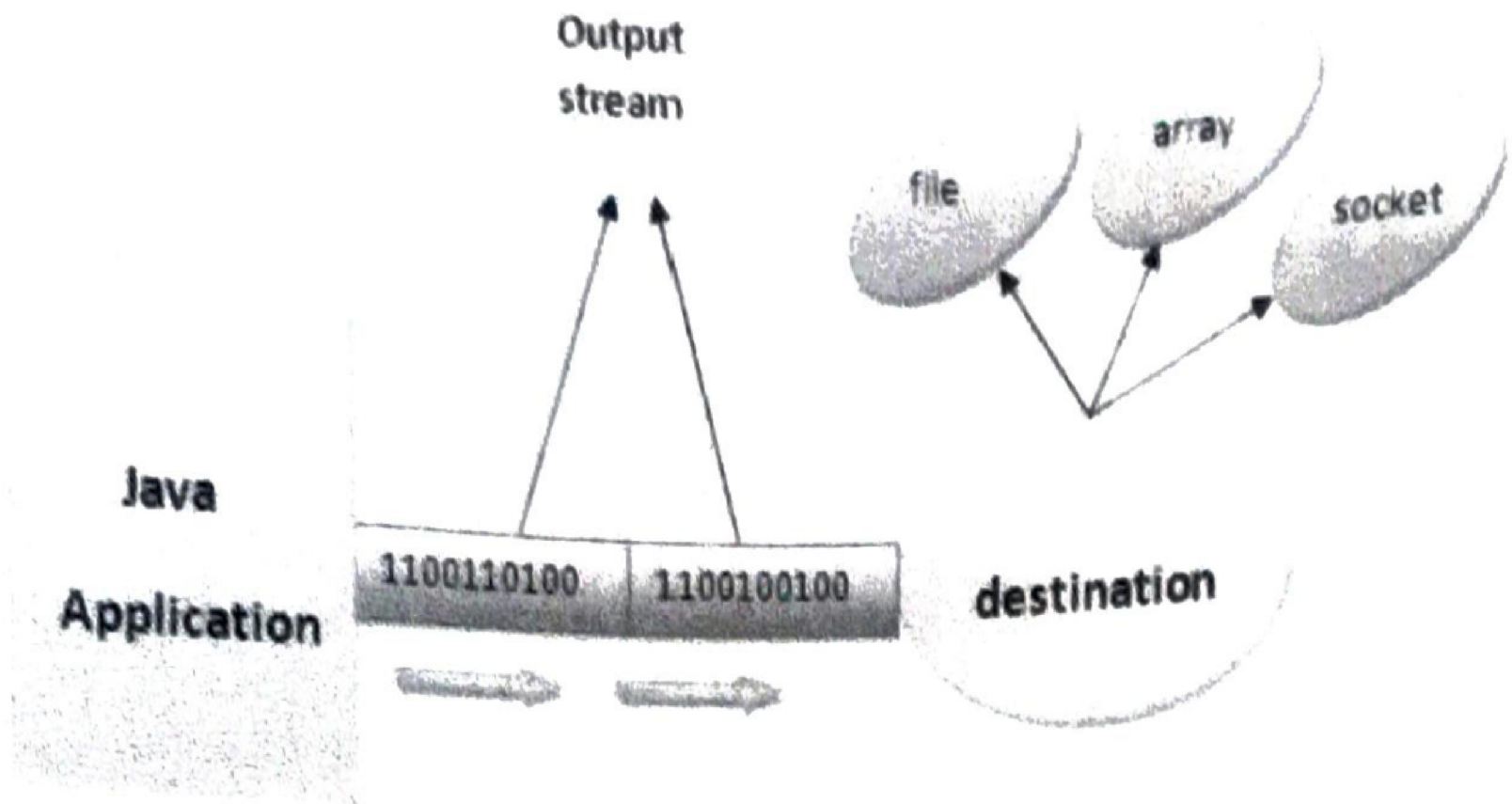
You can explicitly close a stream with the close( ) method, or let it be closed implicitly when the object is found as a garbage.InputStream is inherited from the Object class. Each class of the InputStreams provided by the Java.io package is intended for a different purpose.

Methods of InputStream

Syntax	Description
read( )	It reads bytes of data from a stream. If no data is available it blocks the method. When a method is blocked, the thread executing waits until the data is available. It throws an IOException if an error occurs.
available( )	This method returns the number of available bytes that can be read without blockage.
close( )	This method closes the input stream. It releases the resources associated with the stream. It throws an IOException if an error occurs.
mark( )	This method is used to mark the current position in the stream.
markSupported( )	This method returns a Boolean value indicating whether the stream supports mark and reset capabilities. It returns true if the stream type supports it and otherwise false.
reset( )	This method repositions the stream to the last marked position. This method throws an IOException if an error occurs.
skip( )	This method skips n bytes of input. –n is the parameter used which specifies the number of bytes to be skipped. It throws an IOException if an error occurs.

OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, a string, peripheral device or socket.



OUTPUT STREAM

Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the close( ) method, or let it be closed implicitly when the object is garbage collected.OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the Java.io package is intended for a different purpose.

Methods of OutputStream

Syntax	Description
write( )	This method writes a byte and throws an IOException if an error occurs. This method actually blocks until the byte is written. The thread waits until the write operation is been completed.
flush( )	This method flushes the stream. The buffered data is written to the output stream. It throws an IOException if an error occurs.
close( )	This method closes the stream. It is used to release any resource associated with the stream. It throws an IOException if an error occurs.

FILEINPUTSTREAM

In Java, a FileInputStream is an input stream that is used to read data from a file. It is a subclass of the InputStream class and provides a convenient way to read binary data from a file in a platform-independent way.

The methods provided by the FileInputStream class in Java for reading data from a file:

Method	Description
FileInputStream(String name)	Creates a new FileInputStream object given the name of the file to be read.
int read()	Reads a single byte of data from the file and returns it as an integer value between 0 and 255.
int read(byte[] b)	Reads up to b.length bytes of data from the file into the specified byte array and returns the number of bytes read.
int read(byte[] b, int off, int len)	Reads up to len bytes of data from the file into the specified byte array starting at the offset off.

long skip(long n)	Skips over n bytes of data in the file and returns the number of bytes actually skipped.
int available()	Returns the number of bytes that can be read from the file without blocking.
void close()	Closes the FileInputStream and releases any resources associated with it.

Note that all of these methods may throw an IOException if an I/O error occurs while reading from the file. It is important to handle these exceptions properly to ensure that your code is reliable and robust.

Here's an example of how to use a FileInputStream to read data from a file:

```
import java.io.FileInputStream;
import java.io.IOException;
public class FileInputStreamExample
{
    public static void main(String[] args)
    {
        try
        {
            FileInputStream inputStream = new FileInputStream("file.txt");

            int data;
            while ((data = inputStream.read()) != -1)
            {
                System.out.print((char) data);
            }
            inputStream.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

In this example, we create an instance of FileInputStream and pass the name of the file we want to read as a parameter. We then use a while loop to read each byte of data from the file using the read() method of FileInputStream. The read() method returns -1 when the end of the file is reached, so we continue reading until we reach the end of the file.

Each byte of data is converted to a character using a type cast to char and printed to the console using System.out.print(). Finally, we close the input stream to release any resources associated with it.

Assuming that the file "file.txt" contains the text "Hello, world!", the output of this program would be:

```
Hello, world!
```

Overall, FileInputStream is a useful class for reading binary data from a file in Java. It provides a simple and efficient way to read large amounts of data from a file without having to load the entire file into memory.

FILEOUTPUTSTREAM

In Java, a FileOutputStream is an output stream that is used to write data to a file. It is a subclass of the OutputStream class and provides a convenient way to write binary data to a file in a platform-independent way.

Methods provided by the FileOutputStream class in Java for writing data to a file:

Method	Description
FileOutputStream(String name)	Creates a new FileOutputStream object given the name of the file to be written.
FileOutputStream(File file)	Creates a new FileOutputStream object given a File object representing the file to be written.
void write(int b)	Writes a single byte of data to the file.
void write(byte[] b)	Writes the entire byte array to the file.
void write(byte[] b, int off, int len)	Writes len bytes of data from the specified byte array starting at offset off to the file.
void flush()	Flushes the output stream, forcing any buffered data to be written to the file.
void close()	Closes the output stream and releases any resources associated with it.
void writeBoolean(boolean v)	Writes a boolean value to the file as a single byte.
void writeByte(int v)	Writes a byte value to the file.
void writeShort(int v)	Writes a short value to the file as two bytes, in network byte order (big-endian).
void writeChar(int v)	Writes a char value to the file as two bytes, in network byte order (big-endian).
void writeInt(int v)	Writes an int value to the file as four bytes, in network byte order (big-endian).
void writeLong(long v)	Writes a long value to the file as eight bytes, in network byte order (big-endian).
void writeFloat(float v)	Writes a float value to the file as four bytes, using the IEEE 754 floating-point format.
void writeDouble(double v)	Writes a double value to the file as eight bytes, using the IEEE 754 floating-point format.
void writeBytes(String s)	Writes the string as a sequence of bytes, one byte per character.
void writeChars(String s)	Writes the string as a sequence of characters, each character represented by two bytes, in network byte order (big-endian).

Note that some of these methods may throw an IOException if an I/O error occurs while writing to the file. It is important to handle these exceptions properly to ensure that your code is reliable and robust.

Here's an example of how to use a FileOutputStream to write data to a file:

```
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamExample
{
    public static void main(String[] args)
    {
        try
        {
            FileOutputStream outputStream = new FileOutputStream("file.txt");

            String message = "Hello, world!";
            byte[] messageBytes = message.getBytes();
            outputStream.write(messageBytes);

            outputStream.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

In this example, we create an instance of FileOutputStream and pass the name of the file we want to write as a parameter. We then create a String object containing the message we want to write to the file, and convert it to an array of bytes using the getBytes() method of String.

We write the byte array to the file using the write() method of FileOutputStream, and then close the output stream to ensure that all data has been written to the file.

Assuming that the file "file.txt" does not exist and is created by this program, its contents after running the program would be:

```
Hello, world!
```

CREATION OF TEXT FILE

The createNewFile() method is used to create a file in Java, and return aBoolean value : true if the file is created successful; false if the file is already exists or the operation failed.The following example shows the usage of Java.io.File.createNewFile() method.

Example: Creating new text file

```
import java.io.File;
public class Student
{
    public static void main(String[] args)
    {
        File f = null;
        Boolean bool = false;
        try
        {
            // create new file
            f = new File("d://Lab/Student.txt");
            // tries to create new file in the system
            bool = f.createNewFile();
            // prints
            System.out.println("File created: "+bool);
            // deletes file from the system
            f.delete();
            // delete() is invoked
            System.out.println("delete() method is invoked");
            // tries to create new file in the system
            bool = f.createNewFile();
            // print
            System.out.println("File created: "+bool);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

```
File created: false
delete() method is invoked
File created: true
```

READING AND WRITING TEXT FILES

FileInputStream classes are used to read and write data in file. In another words, they are used for file handling in Java.

1. **Opening a file:**

To open a file, you need to create an instance of the File class and pass the file path as a parameter. For example:

```
File file = new File("file.txt");
```

2. **Reading from a file:**

To read data from a file, you can use a FileReader or a BufferedReader class. Here's an example:

```
try
{
    FileReader fileReader = new FileReader("file.txt");
    BufferedReader bufferedReader = new BufferedReader(fileReader);

    String line;
    while ((line = bufferedReader.readLine()) != null)
    {
        System.out.println(line);
    }
} catch (IOException e)
{
    e.printStackTrace();
}
```

3. **Writing to a file:**

To write data to a file, you can use a FileWriter or a BufferedWriter class. Here's an example:

```
try
{
    FileWriter fileWriter = new FileWriter("file.txt");
    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

    bufferedWriter.write("Hello, world!");
    bufferedWriter.newLine();
    bufferedWriter.write("This is a test.");
    bufferedWriter.close();
} catch (IOException e)
{
    e.printStackTrace();
}
```

4. **Closing a file:**

Once you have finished reading from or writing to a file, you should close the file to release the resources associated with it. Here's an example:

```
try
{
    BufferedReader bufferedReader = new BufferedReader(new FileReader("file.txt"));

    // read data from file

    bufferedReader.close();
}
catch (IOException e)
{
    e.printStackTrace();
}
```

To illustrate reading and writing to text file in we take an example program that reads data from one text file and writes it to another text file:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class ReadWriteTextFileExample
{
    public static void main(String[] args)
    {
        try
        {
            BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
            BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));

            String line;
            while ((line = reader.readLine()) != null)
            {
                writer.write(line);
                writer.newLine();
            }

            reader.close();
            writer.close();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

In this example, we create an instance of BufferedReader to read data from the input file "input.txt", and an instance of BufferedWriter to write data to the output file "output.txt".

We use a while loop to read each line of data from the input file using the readLine() method of BufferedReader. We then write each line of data to the output file using the write() method of BufferedWriter, and add a newline character using the newLine() method.

Finally, we close both the input and output streams to release any resources associated with them.

Assuming that the input file "input.txt" contains the text:

```
This is line 1.
This is line 2.
This is line 3.
```

The output file "output.txt" would contain the same text after running the program.

COLLECTIONS FRAMEWORK OVERVIEW

The Collections Framework is a set of classes and interfaces in Java that provides a unified architecture for storing and manipulating groups of objects. It was introduced in Java 1.2 and has since become an integral part of Java development.

The main components of the Collections Framework are:

1. **Interfaces:** These define the common operations that can be performed on a collection of objects, such as adding or removing elements, iterating over the collection, and checking its size.
2. **Classes:** These provide concrete implementations of the collection interfaces. Some of the commonly used classes are ArrayList, LinkedList, HashSet, and TreeMap.
3. **Algorithms:** These are a set of methods that can be used to perform common operations on collections, such as sorting, searching, and shuffling.

The advantages of using the Collections Framework are:

1. **Consistent API:** The framework provides a consistent and easy-to-use API for manipulating collections of objects.
2. **Reusability:** Since the collection interfaces are standardized, code that uses collections can be easily reused and adapted to different types of collections.
3. **Efficiency:** The framework provides efficient and optimized implementations of common collection classes.
4. **Type Safety:** The framework provides type safety through the use of generics, which ensures that collections can only contain objects of a specific type.

Overall, the Collections Framework is an essential tool for Java developers working with collections of objects, and it simplifies the task of managing complex data structures in Java programs.

COLLECTION CLASSES

Java provides several collection classes in the java.util package that can be used to store and manipulate groups of objects. Here are some of the commonly used collection classes in Java:

1. ArrayList
2. LinkedList
3. HashSet
4. TreeSet
5. HashMap
6. TreeMap

1. ArrayList:

An ArrayList is a dynamic array that can grow or shrink as needed. It's useful when you need to store a list of objects that can change in size. Here's an example program:

```
import java.util.ArrayList;
public class ArrayListExample
{
    public static void main(String[] args)
    {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");
        System.out.println(names);
    }
}
```

This program creates an ArrayList of Strings called names, adds four names to it using the add() method, and then prints out the contents of the list using the toString() method. The output of this program would be:

```
[Alice, Bob, Charlie, David]
```

2. LinkedList:

A LinkedList is a doubly-linked list that can also grow or shrink as needed. It's useful when you need to manipulate the order of elements frequently, such as when implementing a queue or stack. Here's an example program:

```
import java.util.LinkedList;
public class LinkedListExample
{
    public static void main(String[] args)
    {
        LinkedList<String> names = new LinkedList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");
        System.out.println(names);
    }
}
```

This program creates a LinkedList of Strings called names, adds four names to it using the add() method, and then prints out the contents of the list using the toString() method. The output of this program would be the same as the previous example.

3. HashSet:

A HashSet is an unordered set of elements with no duplicates. It's useful when you need to store a collection of objects without worrying about the order or duplicates. Here's an example program:

```
import java.util.HashSet;
public class HashSetExample
{
    public static void main(String[] args)
    {
        HashSet<String> names = new HashSet<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");
        System.out.println(names);
    }
}
```

This program creates a HashSet of Strings called names, adds four names to it using the add() method, and then prints out the contents of the set using the toString() method. The output of this program might vary since the order is not guaranteed, but there should be no duplicates:

```
[Alice, Charlie, Bob, David]
```

4. TreeSet:

A TreeSet is an ordered set of elements with no duplicates. It's useful when you need to store a collection of objects in sorted order without worrying about duplicates. Here's an example program:

```
import java.util.TreeSet;
public class TreeSetExample
{
    public static void main(String[] args)
    {
        TreeSet<String> names = new TreeSet<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");
        System.out.println(names);
    }
}
```

This program creates a TreeSet of Strings called names, adds four names to it using the add() method, and then prints out the contents of the set using the toString() method. The output of this program would be:

```
[Alice, Bob, Charlie, David]
```

5. HashMap:

A HashMap is an unordered map of key-value pairs with no duplicate keys. It's useful when you need to store a collection of objects with a unique identifier. Here's an example program:

```
import java.util.HashMap;
public class HashMapExample
{
    public static void main(String[] args)
    {
        HashMap<String, Integer> ageMap = new HashMap<>();

        ageMap.put("Alice", 25);
        ageMap.put("Bob", 30);
        ageMap.put("Charlie", 35);
        ageMap.put("David", 40);

        System.out.println("Age of Alice: " + ageMap.get("Alice"));
        System.out.println("Age of Bob: " + ageMap.get("Bob"));
        System.out.println("Age of Charlie: " + ageMap.get("Charlie"));
        System.out.println("Age of David: " + ageMap.get("David"));
        ageMap.put("Alice", 26);
        System.out.println("New age of Alice: " + ageMap.get("Alice"));

        ageMap.remove("David");
        System.out.println("Age of David (after removal): " + ageMap.get("David"));

        for (String name : ageMap.keySet())
        {
            System.out.println(name + " is " + ageMap.get(name) + " years old.");
        }
    }
}
```

Output:

```
Age of Alice: 25
Age of Bob: 30
Age of Charlie: 35
```

```
Age of David: 40
New age of Alice: 26
Age of David (after removal): null
Alice is 26 years old.
Bob is 30 years old.
Charlie is 35 years old.
```

This program creates a HashMap called ageMap, which maps the names of four people to their ages. It then retrieves and prints the ages of each person using the get() method. The age of Alice is then updated and printed again, and the entry for David is removed from the map. Finally, a for-each loop is used to iterate over each key in the map and print the name and age of each person to the console.This output shows that the program successfully created a HashMap called ageMap, added and retrieved values using the put() and get() methods, updated the value for Alice, removed the entry for David, and iterated over the keys in the map using a for-each loop.

6. TreeMap:

TreeMap is a collection class in Java that implements the Map interface and stores its entries in a sorted order. Each entry in a TreeMap consists of a key and a value, and the entries are sorted according to the natural ordering of the keys or according to a Comparator that is provided to the constructor. TreeMap provides efficient operations for adding, removing, and retrieving entries, as well as for finding the first and last entries in the map.

Here is an example program that demonstrates how to use a TreeMap in Java:

```
import java.util.TreeMap;

public class TreeMapExample
{
    public static void main(String[] args)
    {
        TreeMap<String, Integer> ageMap = new TreeMap<>();

        ageMap.put("Alice", 25);
        ageMap.put("Bob", 30);
        ageMap.put("Charlie", 35);
        ageMap.put("David", 40);

        System.out.println("Entries in the TreeMap:");
        for (String name : ageMap.keySet())
        {
            System.out.println("Age of " + name + ": " + ageMap.get(name));
        }

        ageMap.put("Alice", 26);
        ageMap.remove("David");
```

```
        System.out.println("Updated entries in the TreeMap:");
        for (String name : ageMap.keySet())
        {
            System.out.println("Age of " + name + ": " + ageMap.get(name));
        }

        System.out.println("Finding the first and last entries in the TreeMap:");
        System.out.println("First entry: " + ageMap.firstEntry());
        System.out.println("Last entry: " + ageMap.lastEntry());
    }
}
```

This program creates a TreeMap called ageMap and adds four entries to it, each consisting of a name and an age. It then prints the entries in the map using a for-each loop, updates the value for Alice and removes the entry for David, prints the updated entries, and finds the first and last entries in the map.

The output of this program would be:

```
Entries in the TreeMap:
Age of Alice: 25
Age of Bob: 30
Age of Charlie: 35
Age of David: 40
Updated entries in the TreeMap:
Age of Alice: 26
Age of Bob: 30
Age of Charlie: 35
Finding the first and last entries in the TreeMap:
First entry: Alice=26
Last entry: Charlie=35
```

This output shows that the program successfully created a TreeMap called ageMap, added and retrieved entries using the put() and get() methods, updated the value for Alice and removed the entry for David, and found the first and last entries in the map. Note that the entries are printed in sorted order of the keys (names), as is characteristic of a TreeMap.

Here's an example program that demonstrates how to use ArrayList and HashMap to store and manipulate data:

java

```
import java.util.ArrayList;
import java.util.HashMap;

public class CollectionExample
{
    public static void main(String[] args)
    {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("David");

        HashMap<String, Integer> ages = new HashMap<>();
        ages.put("Alice", 25);
        ages.put("Bob", 30);
        ages.put("Charlie", 35);
        ages.put("David", 40);

        for (String name : names)
        {
            int age = ages.get(name);
            System.out.println(name + " is " + age + " years old.");
        }
    }
}
```

In this example, we create an ArrayList called names to store a list of names, and add four names to it using the add() method. We also create a HashMap called ages to store the ages of each person, and use the put() method to add each name and age to the map.

We then use a for-each loop to iterate over each name in the names list, and use the get() method of the ages map to retrieve the age of each person. We then print out a message to the console indicating each person's name and age.

The output of this program would be:

```
Alice is 25 years old.
Bob is 30 years old.
Charlie is 35 years old.
David is 40 years old.
The For-Each loop
```

The For-Each loop (also known as the enhanced for loop) in Java is a concise way of iterating over arrays and collections. It was introduced in Java 5 and is particularly useful when we want to iterate over all the elements in a collection without worrying about the size of the collection or the indices of the elements.Here is an example program that demonstrates how to use the For-Each loop in Java:

```
public class ForEachLoopExample
{
    public static void main(String[] args)
    {
        int[] numbers = {1, 2, 3, 4, 5};

        System.out.println("Using a traditional for loop:");
        for (int i = 0; i < numbers.length; i++)
        {
            System.out.println(numbers[i]);
        }

        System.out.println("Using a For-Each loop:");
        for (int number : numbers)
        {
            System.out.println(number);
        }
    }
}
```

This program creates an array of integers called numbers and then prints its contents using both a traditional for loop and a For-Each loop.The output of this program would be:

```
Using a traditional for loop:
1
2
3
4
5
Using a For-Each loop:
1
2
3
4
5
```

This output shows that the program successfully created an array of integers called numbers, printed its contents using both a traditional for loop and a For-Each loop, and obtained the same results using both approaches. Note that the For-Each loop is more concise and easier to read than the traditional for loop.

MULTIPLE CHOICE QUESTIONS (MCQs)

1. Which of the following classes is used to read data from a file in Java?
   (a) FileWriter (b) FileReader (c) InputStream (d) OutputStream
2. Which of the following methods is used to write data to a file in Java?
   (a) write() (b) read() (c) close() (d) flush()
3. Which of the following classes is used to write data to a file in Java?
   (a) FileWriter (b) FileReader (c) InputStream (d) OutputStream
4. Which of the following methods is used to read a line of text from a file in Java?
   (a) readLine() (b) readChar() (c) read() (d) write()
5. Which of the following methods is used to create a new file in Java?
   (a) createFile() (b) newFile() (c) makeFile() (d) fileCreate()

   Ans: There is no direct method to create a file in Java. You can create a file using the following code:

```
File file = new File("filename.txt");
file.createNewFile();
```

6. Which of the following classes is used to represent a file or directory path in Java?
   (a) File (b) Path (c) Directory (d) Folder
7. Which of the following methods is used to check if a file exists in Java?
   (a) exists() (b) isFile() (c) isDirectory() (d) both A and B
8. Which of the following methods is used to delete a file in Java?
   (a) delete() (b) remove() (c) erase() (d) clear()
9. Which of the following classes is used to write binary data to a file in Java?
   (a) FileWriter (b) FileReader (c) DataOutputStream (d) DataInputStream
10. Which of the following methods is used to append data to an existing file in Java?
    (a) append() (b) add() (c) insert() (d) concat()
11. Which of the following interfaces is used to represent an ordered collection of elements in Java?
    (a) List (b) Set (c) Map (d) Queue
12. Which of the following interfaces is used to represent an unordered collection of elements in Java?
    (a) List (b) Set (c) Map (d) Queue
13. Which of the following interfaces is used to represent a collection of key-value pairs in Java?
    (a) List (b) Set (c) Map (d) Queue
14. Which of the following interfaces is used to represent a collection that allows duplicates and preserves insertion order in Java?
    (a) HashSet (b) TreeSet (c) LinkedHashSet (d) HashMap
15. Which of the following methods is used to add an element to a collection in Java?
    (a) add() (b) remove() (c) clear() (d) size()

16. Which of the following methods is used to remove an element from a collection in Java?
    (a) add() (b) remove() (c) clear() (d) size()
17. Which of the following methods is used to check if a collection contains a specific element in Java?
    (a) add() (b) contains() (c) remove() (d) clear()
18. Which of the following methods is used to sort a collection in natural order in Java?
    (a) sort() (b) sorted() (c) Collections.sort() (d) Arrays.sort()
19. Which of the following methods is used to iterate over a collection in Java?
    (a) for loop (b) while loop (c) do-while loop (d) foreach loop
20. Which of the following interfaces is used to represent a collection that allows elements to be added and removed from both ends in Java?
    (a) List (b) Set (c) Map (d) Deque

Answers to MCQs

1. (b) FileReader
2. (a) write()
3. (a) FileWriter
4. (a) readLine()
5.
6. (a) File
7. (a) exists()
8. (a) delete()
9. (c) DataOutputStream
10. (a) append()
11. (a) List
12. (b) Set
13. (c) Map
14. (c) LinkedHashSet
15. (a) add()
16. (b) remove()
17. (b) contains()
18. (c) Collections.sort()
19. (d) foreach loop
20. (d) Deque

IMPORTANT QUESTIONS

Short Questions 2 Marks

1. List different stream classes.

Ans: There are following Stream classes:

Character Stream Classes

- Reader Stream classes
- Writer Stream classes

Byte Stream Classes

2. List out I/O classes in Java.

Ans: There are following I/O Classes:

InputStream classes

- FileInputStream class

OutputStream classes

- FileOutputStream class

3. Describe stream classes in Java.

Ans:

- Java programs perform I/O through streams.
- Streams are pipelines for sending and receiving information from/to Javaprograms.
- Streams are also classified in to two categories

(1) Character streams classes

(2) Byte streams classes

- Reader and Writerare the fundamental classes of the type character streams, whileInputstream and Outputstream are the fundamental classes of the type Bytestreams.

4. Write syntax of reading text file in Java.

Ans: Syntax:

```
FileInputStream object = new FileInputStream("path:\\txtfile.txt");
object.read();
```

Questions 3 Marks

1. Write a program in Java to create a file and perform read operation on this file.
2. Write a program in Java to create a file and perform write operation on this file.
3. What is stream class? How are the stream classes classified?
4. Write a program in Java to create, write, modify, read operations on a Text file.

Ans:

```
import java.io.*;
class Student
{
    public static void main(String args[])throws Exception
    {
        FileInputStream fin=new FileInputStream("s.txt");
        FileOutputStream fout=new FileOutputStream("s.txt");
        try
        {
            String s="India is Great";
            byte b[]=s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("Writing Complete!");
        }
        catch(Exception e){}
        try
```

```
        {
            int i;
            while((i=fin.read())!=-1)
            System.out.println((char)i);
            fin.close();
            System.out.println("Reading Complete!");
        }
        catch(Exception e) {}
        FileOutputStream fout_modify=new FileOutputStream("s.txt");
        try
        {
            String s1="Computer Engineering";
            byte b1[]=s1.getBytes();
            fout_modify.write(b1);
            fout_modify.close();
            System.out.println("Modification Complete!");
        }
        catch(Exception e) {}
        FileInputStream fin_modify=new FileInputStream("s.txt");
        try
        {
            int r;
            while((r=fin_modify.read())!=-1)
            System.out.println((char)r);
            fin_modify.close();
            System.out.println("Reading Complete!");
        } catch(Exception e) {}
    }
}
```

Output

```
Writing Complete!
I
n
d
i
a

i
s
```

```
G
r
e
a
t
Reading Complete!
Modification Complete!
C
o
o
m
p
u
t
e
r

E
n
g
i
n
e
e
r
i
n
g
Reading Complete!
```

5. **Write Java syntax to create a new text file.**

Ans: Syntax:

```
File object = new File("Path:/txtfilename.txt");
object.createNewFile();
```

6. **Write syntax of writing a text file in Java.**

Ans: Syntax:

```
FileOutputStream obj=new FileOutputStream("F:\\newfile.txt");
byte b[]=string_object.getBytes();
obj.write(b);
```

7. **What are collections in Java?**

Ans: Collections are a group of classes and interfaces in Java that are used to represent groups of objects.

8. What are the main benefits of using collections in Java?

Ans: Collections provide a more flexible and efficient way to store and manipulate groups of objects compared to traditional arrays. They also provide a wide range of operations for searching, sorting, and manipulating data.

9. What is the difference between a List and a Set in Java?

Ans: A List is an ordered collection of elements that allows duplicates, whereas a Set is an unordered collection that does not allow duplicates.

10. What is the difference between an ArrayList and a LinkedList in Java?

Ans: An ArrayList is implemented as a resizable array, while a LinkedList is implemented as a linked list. As a result, ArrayLists are faster for random access and insertion/deletion at the end of the list, while LinkedLists are faster for insertion/deletion in the middle of the list.

11. What is the difference between a HashMap and a TreeMap in Java?

Ans: A HashMap is an unordered collection that uses key-value pairs to store data, while a TreeMap is an ordered collection that sorts elements based on their natural order or a specified Comparator.

■ Questions 4 Marks

1. Write a java program to read a text file in java.
2. Write a java program to write a text file in java.
3. Explain java I/O process.
4. Explain stream classes.
5. Explain how to create a text file in context to file handling.

ASSIGNMENTS

■ File Handling Assignment

1. Create a new file called "myFile.txt" and write the following data to it:

   Hello World!

   This is a simple file handling assignment.

2. Read the data from the "myFile.txt" file and display it on the console.
3. Append a new line to the "myFile.txt" file:

   Java is awesome!

4. Read the data from the "myFile.txt" file again and display it on the console to verify that the new line has been added.
5. Create a new file called "myData.csv" and write the following data to it:

   Name, Age, Gender

   John Doe, 25, Male

   Jane Smith, 30, Female

   David Lee, 20, Male

6. Read the data from the "myData.csv" file and display it on the console.
7. Update the age of "David Lee" to 21 in the "myData.csv" file.
8. Read the data from the "myData.csv" file again and display it on the console to verify that the age has been updated.
9. Delete the "myData.csv" file.
10. Create a new directory called "myDirectory" and move the "myFile.txt" file into it.

■ **Collection Framework Assignment**

11. Create an ArrayList of String type called "colors" and add the following colors to it:

    Red, Green, Blue, Yellow, Orange

12. Display the contents of the "colors" ArrayList on the console.
13. Remove the "Yellow" color from the "colors" ArrayList.
14. Display the contents of the "colors" ArrayList again on the console to verify that the "Yellow" color has been removed.
15. Create a LinkedList of Integer type called "numbers" and add the following numbers to it:

    10, 20, 30, 40, 50

16. Display the contents of the "numbers" LinkedList on the console.
17. Remove the first and last element from the "numbers" LinkedList.
18. Display the contents of the "numbers" LinkedList again on the console to verify that the first and last elements have been removed.
19. Create a HashSet of String type called "animals" and add the following animals to it:

    Dog, Cat, Rabbit, Monkey, Lion

20. Display the contents of the "animals" HashSet on the console.
21. Remove the "Monkey" animal from the "animals" HashSet.
22. Display the contents of the "animals" HashSet again on the console to verify that the "Monkey" animal has been removed.

■ * ■

STREAM CLASSES

Stream

A stream is a sequence of data. In Java a stream is composed of bytes. Streams are pipelines for sending and receiving information in Java programs.

Three streams are created for us automatically:

1. System.out: standard output stream
2. System.in: standard input stream
3. System.err: standard error

Stream Types

Streams are also classified in to two categories

1. Character streams
2. Byte streams

Input streams and Output streams are of the type Byte streams.

Reader and Writer streams are of the type Character streams.

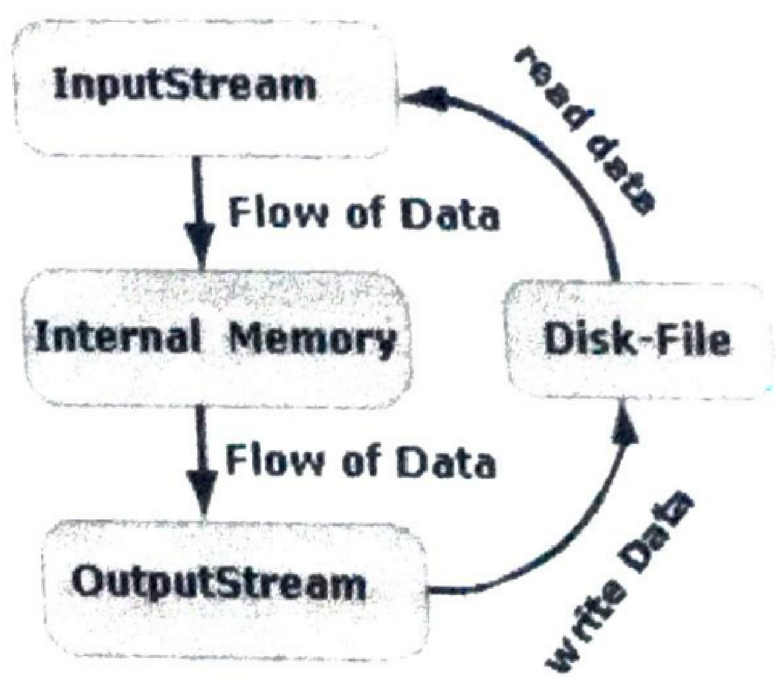
Stream Types

	Byte Streams	Character Streams
Source Streams	InputStream	Reader
Sink Streams	OutputStream	Writer

Each type of stream results in four actual streams: Byte based input, Byte based output, Character based input (Reader), and Character based output (Writer).

How Files and Streams Work?

- Java uses streams to handle I/O operations through which the data is flowed from one location to another.
- For example, an InputStream can flow the data from a disk file to the internal memory and an OutputStream can flow the data from the internal memory to a disk file.
- The disk-file may be a text file or a binary file.



FILES AND STREAM INTERACTION