IO PACKAGE

- Contains many classes, interfaces and methods.
- Java defines 2 complete I/O systems:
  - **byte I/O**
  - **character I/O**
- **I/O classes support**
  - **text-based console I/O**
  - **and file I/O.**

STREAMS

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- Thus, the same I/O classes and methods can be applied to any type of device.
- This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.
- Similarly , an output stream may refer to the console, a disk file, or a network connection.
- Java implements streams within class hierarchies defined in the **java.io** package.

BYTE STREAMS AND CHARACTER STREAMS

- Java defines two types of streams: byte and character.
  - *Byte streams* provide a convenient means for handling input and output of bytes.
    - Byte streams are used, for example, when reading or writing binary data.
  - *Character streams* provide a convenient means for handling input and output of characters.
    - They use Unicode and, therefore, can be internationalized.

THE BYTE STREAM CLASSES

- Byte streams are defined by using two class hierarchies.
- At the top are two abstract classes:
  - **InputStream-** defines the characteristics common to byte input streams
  - **OutputStream-** describes the behavior of byte output streams

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStrea m	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for

	reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print( ) and println( )
PushbackInputStream	Input stream that supports one-byte “unget,” which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array

CHARACTER STREAM CLASSES

- Topped by 2 abstract classes Reader and Writer
- Reader is used for input and Writer for output

CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print( ) and println( )
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

READING CONSOLE INPUT AND WRITING CONSOLE OUTPUT

READING CONSOLE INPUT: Reading console input means reading input from standard input device i.e keyboard

- Input can be in two forms
    1. Byte-oriented
    2. Character-oriented

1. Byte-Oriented:

- read() method is used for reading byte oriented data from the console
  - System.in is an instance of InputStream
  - read() is a method of InputStream
    - there are 3 versions of read()
      1. int read() throws IOException
         - reads a single character from the keyboard.returns -1 when end of stream is encountered.
      2. int read(byte[] buffer) throws IOException
         - reads bytes from input stream and puts them into buffer until either the array is full or end of stream
      3. int read(byte[] buffer, int offset, int numofbytes) throws IOException
         - reads input into buffer beginning at the location specified by offser upto numofbytes are stores.

- **Ex:** byte[] data=new byte[10];
  int n=System.in.read(data)

2. Character-Oriented:

- The statement used for reading character-based input from the keyboard(console) is

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

  - ✓ After this statement executes, **br** is character-based that is linked to the console through **System.in** and can be used to read the input(characters and strings)

i. Reading Characters

- read() method is used for reading the input
- The version of **read( )** that we will be using is

- int read( ) throws IOException
  - Each time that **read( )** is called, it reads a character from the input stream and returns it as an integer value.
  - It returns –1 when the end of the stream is encountered.
  - it can throw an **IOException**.

Example :

```
import java.io.*;

class BRRead {
```

```
public static void main(String args[]) throws IOException
{
int c;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
 do {
c =br.read();
 System.out.print(c);
} while(c != 'q'); }
}
```

ii. Reading Strings

- readLine() method is used to read a string from the keyboard

Syntax:

String readLine( ) throws IOException

Example:

```
import java.io.*;
class BRReadLines {
public static void main(String args[]) throws IOException
{
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 String str;
System.out.println("Enter lines");
System.out.println("Enter 'stop' to quit.");
do {
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop")); }
}
```

WRITING CONSOLE OUTPUT: means writing the output on standard output device i.e screen

- output can be in two forms
    1. Byte-oriented
    2. Character-oriented

1. **Byte-Oriented:**

- System.out is a byte stream connected to console(screen)
- write() method of PrintStream class is used to display the output on the screen

  syntax: void write(int b)

  Example:

  - ✓ int b='X';
  - ✓ System.out.write(b);

2. **Character-Oriented:**

- for writing character based data also using **System.out** is acceptable.
- **PrintWriter** is the character-based class
- **PrintWriter** defines several constructors. The one we will use is shown here:

  PrintWriter(OutputStream *os*, boolean *flushOnNewline*)
- **PrintWriter** supports the **print( )** and **println( )** methods for all types including **Object**.

Example:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo {
public static void main(String args[]) {
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
int i = -7;
 pw.println(i);
 double d = 4.5e-7;
  pw.println(d);
} }
```

FILE CLASS

- File class deals with files and the file system.
- the **File** class does not specify how information is retrieved from or stored in files;
- it describes the properties of a file
- A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

- A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)

File(URI *uriObj*)

```
// Program illustrating  File class

import java.io.File;
import java.util.Date;
class FileDemo
{
static void p(String s)
{ System.out.println(s);
}

public static void main(String args[])
{
File f1 = new File("FileDemo.java");
 p("File Name: " + f1.getName());
 p("Path: " + f1.getPath());
p("Abs Path: " + f1.getAbsolutePath());
 p("Parent: " + f1.getParent());
 p(f1.exists() ? "exists" : "does not exist");
p(f1.canWrite() ? "is writeable" : "is not writeable");
p(f1.canRead() ? "is readable" : "is not readable");
p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
p(f1.isFile() ? "is normal file" : "might be a named pipe");
 p("File last modified: " + f1.lastModified());
p("File size: " + f1.length() + " Bytes"); }
}
```

READING AND WRITING FILES

- ✓ In Java ,
  - all files are byte-oriented
  - Provides methods to read and write bytes from and to a file
  - To create a byte stream linked to a file use FileInputStream and FileOutputStream.
  - To open a file simply create an object of one of these classes ,specifying the name of the file as an argument to the constructor .

INPUTTING FROM A FILE:

- ✓ FileInputStream(Stringfname)throws FileNotFoundException

- ✓ int read() throws IOException
- ✓ void close() throws IOException

WRITING TO A FILE

- ✓ FileOutputStream(String *filePath*) ---GIVE FULL PATH OF FILE AS PARAMETER
- ✓ FileOutputStream(File *fileObj*) ----GIVE OBJECT OF CLASS FILE
- ✓ FileOutputStream(String *filePath*, boolean *append*) ---IF APPEND IS TRUE THEN THE FILE IS OPENED IN APPEND MODE OTHERWISE IS OVERWRITTEN
- ✓ FileOutputStream(File *fileObj*, boolean *append*)

/* PROGRAM ILLUSTRATING COPYING OF FILES- INPUT AND OUTPUT FILE NAMES should be given as commandline arguments*/

```
import java.io.*;

class CopyFile {

public static void main(String args[]) throws IOException
{

 int i;
 FileInputStream fin;
 FileOutputStream fout;

try {

// open input file
try
{
fin = new FileInputStream(args[0]);
}
catch(FileNotFoundException e)
{
System.out.println("Input File Not Found");
return;
}

// open output file
 try
{
fout = new FileOutputStream(args[1]);
 }
 catch(FileNotFoundException e)
{
System.out.println("Error Opening Output File");
return;
}

}
catch(ArrayIndexOutOfBoundsException e)
```

```
{ System.out.println("U DIDNOT GIVE FILE NAMES IN COMMANDLINE");
 return;
}

// Copy File
 try
 {
do {

i = fin.read();

if(i != -1)
fout.write(i);
 }
 while(i != -1);
}
catch(IOException e)
{ System.out.println("File Error");
}

fin.close(); fout.close();

} }
```

RANDOM ACCESS IN FILES

- Reading and writing in files is done sequentially
- If we want to read and write in a file randomly then we use the object of the class RandomAccessFile of io package.
- **RandomAccessFile** supports positioning requests—that is, you can position the *file pointer* within the file.
- It has these two constructors:

  RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException

  RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

  - → In the first form, *fileObj* specifies the name of the file to open as a **File** object..
  - → In the second form, the name of the file is passed in *filename.*
  - → In both cases, *access* determines what type of file access is permitted.
  - → If it is "r", then the file can be read, but not written.
  - → If it is "rw", then the file is opened in read-write mode.
  - → The method **seek** is used to set the current position of the file pointer within the file:

Syntax:

void seek(long *position*) throws IOException

- Here, *position* specifies the new position, in bytes, of the file pointer from the beginning of the file.
- After a call to **seek( )**, the next read or write operation will occur at the new file position.

Some

void	close()	It closes this random access file stream and releases any system resources associated with the stream.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.

//PROGRAM ILLUSTRATING RANDOMACCESSING IN FILES USING *RANDOMACCESSFILE* CLASS

```
import java.io.*;
class RandomAccessFileDemo
{
public static void main(String args[])throws IOException
{
RandomAccessFile raf=new RandomAccessFile("sample.txt","rw");
raf.seek(3);
System.out.println((char)raf.read());
raf.seek(5);

System.out.println((char)raf.read());
}}
```

Java Console Class

- → The Java Console class is used to get input from console.
- → It provides methods to read texts and passwords.
- → If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally.

String readLine()	It is used to read a single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.

How to get the object of Console

```
//Console c=System.console();
```

PROGRAM :

Java Console Example to read STRINGs

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
}
}
```

Java Console Example to read password

```
import java.io.Console;
class ReadPasswordTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter password: ");
char[] ch=c.readPassword();
String pass=String.valueOf(ch);//converting char array into string
System.out.println("Password is: "+pass);
}  }
```

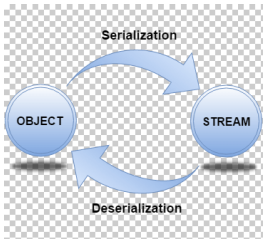
Serialization and Deserialization in Java

Serialization:

- **Serialization in Java** is a mechanism of *writing the state of an object into a byte stream*.
- The byte stream created is platform independent.
- So, the object serialized on one platform can be deserialized on a different platform.
- The reverse operation of serialization is called *deserialization*.

Advantages of Java Serialization

1. To save/persist state of an object.
2. It is mainly used to travel object's state on the network (which is known as marshaling).



java.io.Serializable interface

- Serializable is a marker interface (has no data member and method).
- It is used to "mark" Java classes so that objects of these classes may get the certain capability.
- The Cloneable and Remote are also marker interfaces.

- → It must be implemented by the class whose object you want to persist.
- → The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

ObjectOutputStream class

- → The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream.
- → Only objects that support the java.io.Serializable interface can be written to streams.

Constructor:

Important Methods

1) public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

Deserialization : Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method	Description
1) public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.
2) public void close() throws IOException {}	closes ObjectInputStream.

EXAMPLE PROGRAM

Name it as Student.java

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;  }  }
```

Name it as Persist.java(Serialization)

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

```
class Persist{
 public static void main(String args[])throws Exception{
  Student s1 =new Student(211,"ravi");
  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);
  out.writeObject(s1);
  out.flush();
  System.out.println("success");
 }
}
```

Name it as Depersist.java (DESERIALIZATION)

```
import java.io.*;
class Depersist{
 public static void main(String args[])throws Exception{
   FileInputStream fin=   new FileInputStream("f.txt");
  ObjectInputStream in=new ObjectInputStream(fin);
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name);
    in.close();   }  }
```

ENUMERATIONS

Java Enum

Enum in java is a data type that contains fixed set of constants.

- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc.
- The java enum constants are static and final implicitly.
- Java Enums can be thought of as classes that have fixed set of constants.

Points to remember for Java Enum

- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple example of java enum

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER, FALL }
  public static void main(String[] args) {
for (Season s : Season.values())
System.out.println(s);  }}
```

→ The java compiler internally adds the values() method when it creates an enum.
→ The values() method returns an array containing all the values of the enum.

Defining Java enum

→ The enum can be defined within or outside the class because it is similar to a class.

Java enum example: defined outside class

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}}
```

Java enum example: defined inside class

```
class EnumExample3{
enum Season { WINTER, SPRING, SUMMER, FALL}
public static void main(String[] args) {
Season s=Season.WINTER
System.out.println(s);
}}
```

Initializing specific values to the enum constants

- → The enum constants have initial value that starts from 0, 1, 2, 3 and so on.
- → But we can initialize the specific value to the enum constants by defining fields and constructors.
- → As specified earlier, Enum can have fields, constructors and methods.

Example of specifying initial value to the enum constants

```
class EnumExample4{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);
int value;
 Season(int value){
this.value=value;  }  }
public static void main(String args[]){
for (Season s : Season.values())
System.out.println(s+" "+s.value);  }}
```

Java enum in switch statement

- → We can apply enum on switch statement as in the given example:

Example of applying enum on switch statement:-

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
public static void main(String args[]){
Day day=Day.MONDAY;
  switch(day){
case SUNDAY:
 System.out.println("sunday");
 break;
case MONDAY:
 System.out.println("monday");
 break;
default:
System.out.println("other day");
}
}}
```

TYPE WRAPPERS

- → A Wrapper class is a class whose object wraps or contains a primitive data types.
- → When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types.
- → In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing and Unboxing:

- The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.

Advantage of Autoboxing and Unboxing:

No need of conversion between primitives and Wrappers manually so less coding is required.

Simple Example of Autoboxing in java:

```
class BoxingExample1

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

   int a=50;

      Integer a2=new Integer(a);//Boxing

  Integer a3=5;//Boxing
     System.out.println(a2+" "+a3);   }   }
```

Output:50 5

Simple Example of Unboxing in java:

→ The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing.

```
class UnboxingExample1{
  public static void main(String args[]){
    Integer i=new Integer(50);
      int a=i;
       System.out.println(a);
 }
}
```

Output:50

GENERICS:

→ It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

→ Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

→ Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Generic Methods:

→ You can write a single generic method declaration that can be called with arguments of different types.

→ Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

→ Following are the rules to define Generic Methods −

1. All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

2. Each type parameter section contains one or more type parameters separated by commas.
3. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
4. The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
5. A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example
Following example illustrates how we can print an array of different type using a single Generic method −

```
public class GenericMethodTest {
  // generic method printArray
  public static < E > void printArray( E[] inputArray ) {
    // Display array elements
    for(E element : inputArray) {
      System.out.printf("%s ", element);
    }
    System.out.println();
  }

  public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println("Array integerArray contains:");
    printArray(intArray);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

This will produce the following result −

Output
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O

Generic Classes:

→ A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

→ As with generic methods, the type parameter section of a generic class can have one or more type

parameters separated by commas.

→ These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example
Following example illustrates how we can define a generic class −

```
public class Box<T> {
  private T t;

  public void add(T t) {
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    Box<String> stringBox = new Box<String>();

    integerBox.add(new Integer(10));
    stringBox.add(new String("Hello World"));

    System.out.printf("Integer Value :%d\n\n", integerBox.get());
    System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

This will produce the following result −

Output
Integer Value :10
String Value :Hello World

NOTE:

The notes till now is according to your syllabus and is enough for your Academics. The following information is for knowledge purpose and may be asked in VIVA and placements

I

THE PREDEFINED STREAMS

- All Java programs automatically import the **java.lang** package.
- defines a class called **System**
- **System** contains three predefined stream variables: **in**, **out**, and **err**.
    - These fields are declared as **public**, **static**, and **final** within **System**.
        - **System.out** refers to the standard output stream. By default, this is the console.

- **System.in** refers to standard input, which is the keyboard by default.
- **System.err** refers to the standard error stream, which also is the console by default.

- these streams may be redirected to any compatible I/O device.
- **System.in** is an object of type **InputStream**;
- **System.out** and **System.err** are objects of type **PrintStream**.
- These are byte streams, even though they typically are used to read and write characters from and to the console.

II

TABLE:METHODS DEFINED BY INPUTSTREAM

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an IOException.
void mark(int numBytes)	Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ])	Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

METHODS DEFINED BY OUTPUTSTREAM

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an IOException.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int b)	Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( )

	with expressions without having to cast them back to byte.
void write(byte buffer[ ])	Writes a complete array of bytes to an output stream.
void write(byte buffer[ ], int offset, int numBytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].

III

This is explantion of BufferedReader object

- System.in is used for reading byte-oriented console input.
- So to read character-oriented data from the keyboard we have to wrap **System.in** in a **BufferedReader** object.
- **BufferedReader** supports a buffered input stream.
- Its most commonly used constructor is shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created.

- **Reader** is an abstract class.
- One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters.
- To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

InputStreamReader(InputStream *inputStream*)

- Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*.
- Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

- After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

IV

In the notes we have written different class names belonging to character-oriented category,the below methods are from Reader and Writer classes which are the top classes

in the hierarchy

METHODS BY READER

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an IOException.
void mark(int numChars)	Places a mark at the current point in the input stream that will remain valid until numChars characters are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char buffer[ ])	Attempts to read up to buffer.length characters into buffer and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char buffer[ ], int offset, int numChars)	Attempts to read up to numChars characters into buffer starting at buffer[offset], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns true if the next input request will not wait. Otherwise, it returns false.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numChars)	Skips over numChars characters of input, returning the number of characters actually skipped.

METHODS BY WRITER

Method	Description

Writer append(char ch)	Appends ch to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence chars)	Appends chars to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence chars, int begin, int end)	Appends the subrange of chars specified by begin and end–1 to the end of the invoking ouput stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an IOException.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int ch)	Writes a single character to the invoking output stream. Note that the parameter is an int, which allows you to call write with expressions without having to cast them back to char.
Method	Description
void write(char buffer[ ])	Writes a complete array of characters to the invoking output stream.

abstract void write(char buffer[ ], int offset, int numChars)	Writes a subrange of numChars characters from the array buffer, beginning at buffer[offset] to the invoking output stream.
void write(String str)	Writes str to the invoking output stream.
void write(String str, int offset, int numChars)	Writes a subrange of numChars characters from the string str, beginning at the specified offset.