Module 3

More features of Java:

Packages and Interfaces - Defining Package, CLASSPATH, Access Protection, Importing Packages, Interfaces.

Exception Handling - Checked Exceptions, Unchecked Exceptions, try Block and catch Clause, Multiple catch Clauses, Nested try Statements, throw, throws and finally.

Input/Output - I/O Basics, Reading Console Input, Writing Console Output, PrintWriter Class, Object Streams and Serialization, Working with Files.

Packages

1. Packages are containers for classes.
    - Defines a namespace in which classes are stored.
    - Classes are stored in system directories with the name of the package.
2. A package in Java is used to group related classes and interfaces.
3. It is a naming and visibility control mechanism used to avoid naming conflicts.
4. Packages are stored in a hierarchical manner.
    - Packages are organized in a structured, tree-like directory format.
    - For example, the package `java.util.Scanner` is part of the `util` package:

    java
    └── util
    └── Scanner.java

    that is the folder structure will be
    - **java/util/**
    - Scanner.java files will be inside **java/util/**

Type of packages

Two types of packages in java

1. **User defined**
2. **Built-in**

- User defined - created by programmers.

Eg– package cbse or package icse

cbse
└── Teacher.java
icse
└── Teacher.java

- InBuilt - prvided by java.

– java.lang:
    primitive types, strings, math functions, threads, and exception
– java.util:
    Contains classes such as vectors, hash tables, date etc.
– java.io: Stream classes for I/O
– java.awt: Classes for implementing GUI – windows, buttons, menus etc.
– java.net: Classes for networking
– java.applet: Classes for applet development

Defining a Package

1. To create a package, simply include the package command:

```
...java code
package package_name;
// Example: package Sample;
...
```

Accessing a Package

1. Use import statement, eg,

format: import pkg1[.pkg2].(classname|*);

```
...java code
import java.util.Scanner;
import java.util.*;
...
```

2. Without importing, eg

```
...java code
cbse.Teacher t1 = new cbse.Teacher();
isce.Teacher t2 = new isce.Teacher();
...
```

Uniersity Question

Create a package reversepackage. Add a class Reverse in it with a method reverse() to print the reverse of a string without using built-in methods. Create a class outside the package and use this method to reverse a string
Ans:

1. create folder **reversepackage** in working directory and implement class Reverse.java inside reversepackage folder

```
// reversepackage/Reverse.java
package reversepackage;
public class Reverse {
        public void reverse(String str) {
                int len = str.length();
                for(int i=len; i>0; i--) {
                        System.out.print(str.charAt(i-1));
                }
        }
}//end of Reverse class
```

2. return to working directory, and implement class TestPkg.java

```
// TestPkg.java
import reversepackage.*;
public class TestPkg {
        public static void main(String[] str) {
                Reverse rev = new Reverse();
                rev.reverse("alice");
        }
} // end of TestPkg class
```

3. excecute as below from workingdirectory

```
/> javac reversepackage/Reverse.java
/> javac TestPkg.java
/> java TestPkg
```

PATH & CLASSPATH Environment Variables

PATH

Definition: It’s an environment variable.
Purpose: Specifies the directories where executable programs are located.
Multiple Entries: The path can include multiple entries, separated by **:** (for UNIX/Linux/Mac) or ; (for Windows).

Purpose of the PATH Variable in Java
The PATH variable allows you to compile and run Java programs from any directory, as the JDK's "bin" directory (where the java and javac executables are located) is added to the PATH environment variable during installation.

so while running
$> java Palindrome
OS will search **'java'** executable in the PATH directory, and execute from there Otherwise, we would be using
$> /bin/java Palindrome

CLASSPATH

Definition: It’s an environment variable.
Purpose: The classpath tells the JVM or compiler where to look for the .class files or libraries needed by the program.

External Libraries: When using external libraries (e.g., JAR files), the classpath specifies where these libraries are located.
Multiple Entries: The classpath can include multiple entries, separated by : (for UNIX/Linux/Mac) or ; (for Windows).

Example Commands:
$>javac -cp .:lib/somelib.jar MyClass.java
$>java -cp .:lib/somelib.jar MyClass

Default Classpath:
If the CLASSPATH environment variable is not set, Java uses the current working directory (.) as the default classpath.

PATH & CLASSPATH Environment Variables

PATH

Definition: It’s an environment variable.
Purpose: Specifies the directories where executable programs are located.
Multiple Entries: The path can include multiple entries, separated by **:** (for UNIX/Linux/Mac) or ; (for Windows).

Purpose of the PATH Variable in Java
The PATH variable allows you to compile and run Java programs from any directory, as the JDK's "bin" directory (where the java and javac executables are located) is added to the PATH environment variable during installation.

so while running
$> java Palindrome
OS will search **'java'** executable in the PATH directory, and execute from there Otherwise, we would be using
$> /bin/java Palindrome

CLASSPATH

Definition: It’s an environment variable.
Purpose: The classpath tells the JVM or compiler where to look for the .class files or libraries needed by the program.

External Libraries: When using external libraries (e.g., JAR files), the classpath specifies where these libraries are located.
Multiple Entries: The classpath can include multiple entries, separated by : (for UNIX/Linux/Mac) or ; (for Windows).

Example Commands:
$>javac -cp .:lib/somelib.jar MyClass.java
$>java -cp .:lib/somelib.jar MyClass

Default Classpath:
If the CLASSPATH environment variable is not set, Java uses the current working directory (.) as the default classpath.

FILE SYSTEM

1. Input / Output Basics

Java programs perform I/O operations through **streams**.

What is a sream ?

A stream is an abstraction for reading/writing a sequence of data.
However, stream does not represent data, instead stream is linked to a physical device by the Java I/O system to read and write data.

Two types of streams (based on direction of data flow)

1. **input** stream : A stream from which data is read from data end points.
2. **output** stream : A stream to which data is written to data end points.

The data end points are

- Keyboard/Console
- File
- Network

Stream Classes in Java

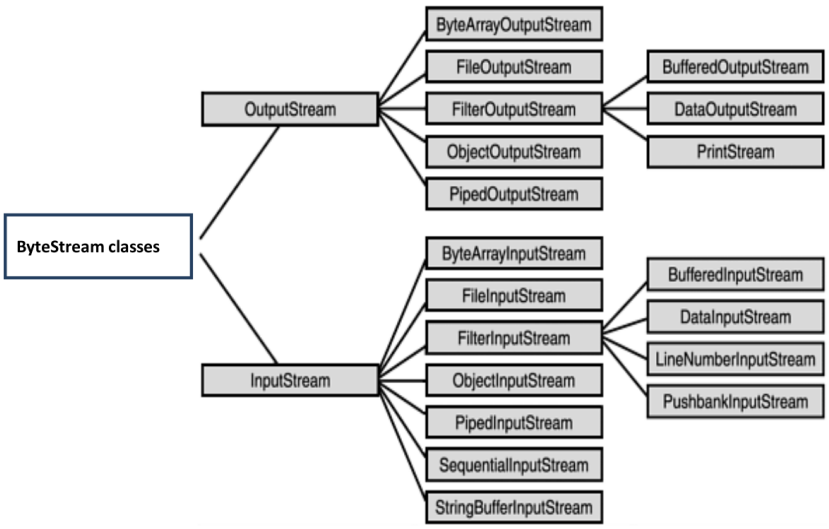
The **java.io** package contains all the classes required for input and output operations.
Java defines two types of streams (based on data type) for IO operation.

- B**yte** stream
- C**harater** stream

Byte Streams: reading/writing data as **bytes**. These streams are used for any kind of data, including text, images, audio, etc as 1 byte ASCII characters (English letters, numbers, etc).

Character streams: reading/writing data as **2-byte unicode characters** which means they can process not just ASCII characters but also characters from international languages (e.g., Chinese, Japanese, Arabic, Hindi...etc).

Byte Stream Classes



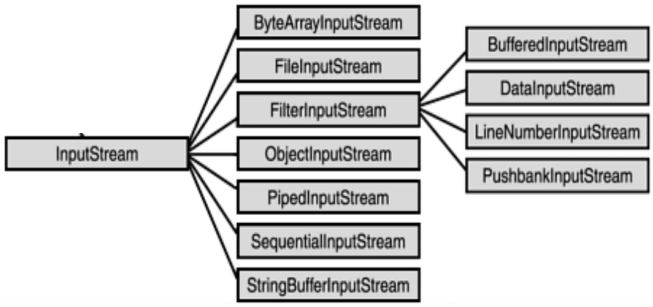
Byte streams are defined by using two class hierarchies as given below

1. **InputStream** – Used to read data from data end points.
2. **OutputStream** – Used to write data to data end points.

*Both these classes are **abstract classes**, that is programmer can not instantiate objects.*
As in diagram, each abstract classes got concreate subclasses to handle different data end points, devices such as Files, Network, Memory buffers.

Two of the most important are **read( )** and **write( )**,
– These methods are overridden by derived stream classes.

InputStream Hirarchy



Some important classes in the hirarchy are discussed below

1. FileInputStream: One of the few classes that can directly **read data** from Disk File.

Constructor and Description

FileInputStream(File file) :
Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.

Eg. opens a file input.txt to read

```
// create File object
File file = new File( "input.txt");
FileInputStream fis = new FileInputStream(file);
```

FileInputStream(String name)
Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system

Eg. opens a file input.txt using file name to read instead of File object.

```
FileInputStream fis = new FileInputStream("input.txt");
```

Read functions and Description

int read()
Reads a byte of data from this input stream.
Eg. read data using above created fis object

```
int data = 0;
while ((data = fis.read()) != -1) {
        System.out.print((char) data);
}
```

int read(byte[] byteArray)
Reads up to **byteArray.length** bytes of data from this input stream into an array of bytes.
Eg. read data using above created fis object

```
byte[] buffer = new byte[1024];
int bytesRead;
while ((bytesRead = fis.read(buffer)) != -1) {
        System.out.print(new String(buffer, 0, bytesRead));
}
```

int read(byte[] b, int off, int len)
Reads up to **len** bytes of data from this input stream into an array of bytes.

Eg. read data using above created fis object

```
byte[] buffer = new byte[1024]; // Define a buffer to hold the data
int bytesRead;
int offset = 0;
int length = 20; // Set the length to read
while ((bytesRead = fis.read(buffer, offset, length)) != -1) {
        // Convert the read bytes to a string and print them
        String output = new String(buffer, 0, bytesRead);
        System.out.println(output);
}
```

Important Note:
*1. There is NO function available in FileInputStrean to read a **String data**.*
2. Classes like ***BufferedInputStream***, ***DataInputStream***, ***ObjectInputStream etc*** *are not capable of reading* data directly from Disk file. Hence these classes **wrap** a FileInputStream object to do **File** I/O operation on respective requirements.

2. BufferedInputStream:
Wraps another InputStream, such as FileInputStream, to improve efficiency by buffering input data (reading larger chunks at once and reducing disk access).
It does not read from the disk directly but uses FileInputStream as its underlying source to get data from Disk file.

Constructor and Description
Eg Create Object of BufferedInputStream

```
FileInputStream fis = new FileInputStream("input.txt");
// wrap FileInputStream with BufferedInputStream
BufferedInputStream bis = new BufferedInputStream(fis);
```

Read functions and Description
Can use any of the read functions given below
int read()
int read(byte[] byteArray)
int read(byte[] b, int off, int len)
Eg. read data using above created **bis** object.

```
int read()
while ((data = bis.read()) != -1) {
        System.out.print((char) data);
        bos.write(data);
}
```

3. DataInputStream:

So far, the data is read as bytes. But some binary files (.bin or .dat extensions) need to read data as primitive data types (e.g., int, float, double, etc.). DataInputStream is ueed to fullfil such requiremets. DataInputStream also be used to read from Network (sockets).

It does not read from the disk directly but uses FileInputStream as its underlying source to get data from Disk file.

Eg Create Object of DataInputStream

```
FileInputStream fis = new FileInputStream("binary_data.dat");
DataInputStream din = new DataInputStream(fis);
```

Can use any of the read functions given below

```
int read()
int read(byte[] byteArray)
int read(byte[] b, int off, int len)
boolean readBoolean()
byte readByte()
char readChar()
double readDouble()
float readFloat()
int readInt()
```

Eg. read data using above created din object.

```
// read double value
double dVal = din.readDouble();
// read int value
int iVal = din.readInt();
// read boolean value
boolean bVal = din.readBoolean();
// read char value
char cVal = din.readChar();

System.out.println("dVal = " + dVal);
System.out.println("iVal = " + iVal);
System.out.println("bVal = " + bVal);
System.out.println("cVal = " + cVal);
```

4. ObjectInputStream:

ObjectOutputStream is used to serialize objects (save them in byte form).
ObjectInputStream is used to deserialize objects (read them back into their original form).Both streams are typically used for persistent storage or object transfer in Java.

```
FileInputStream fileInputStream = new FileInputStream("serialized_object.dat");
ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
// Read and deserialize the object from the file
Person person = (Person) objectInputStream.readObject();
```

5. PipedInputStream : Used to create a **data pipe**.
It facilitates communication between two threads in the same program by creating a data pipe. One thread writes data to the pipe using a PipedOutputStream, and another thread reads the data from the pipe using a PipedInputStream

```
PipedOutputStream outputStream = new PipedOutputStream();
PipedInputStream inputStream = new PipedInputStream(outputStream);

String message = "Hello from the producer!";
outputStream.write(message.getBytes());

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

6. ByteArrayInputStream: allows an application to create an input stream from a byte array. It is used when you want to **read data from memory** (i.e., a byte array) **rather than** from an external data source like a file, socket, or pipe**.**

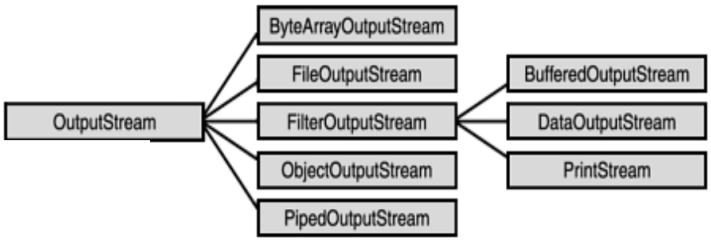
```
FileInputStream fileInputStream = new FileInputStream("firmware.bin");
byte[] fileData = new byte[fileInputStream.available()];
fileInputStream.read(fileData);

ByteArrayInputStream input = new ByteArrayInputStream(fileData);

// Calculate the SHA-256 checksum of the data
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] buffer = new byte[4096];
int bytesRead;

while ((bytesRead = input.read(buffer)) != -1) {
   md.update(buffer, 0, bytesRead);
}
```

OutputStream Hirarchy



Some important classes in the hirarchy are discussed below

1. FileOutputStream: One of the few classes that can directly **write data** from Disk File.

Constructor and Description

FileOutputStream(File file)
Creates a file output stream to write to the file represented by the specified File object.

Open in write mode

```
File file = new File( "output.txt");
FileOutputStream fos = new FileOutputStream(file);
```

FileOutputStream(File file, boolean append)

Open in write mode, if boolean append = false
Open in append mode, if boolean append = true

```
File file = new File( "output.txt");
FileOutputStream fos = new FileOutputStream(file, append);
```

FileOutputStream(String name)
Creates a file output stream to write to the file represented by the specified f**ile name**.

```
FileOutputStream fos = new FileOutputStream(“output.txt”);
```

FileOutputStream(String name, boolean append)
Creates a file output stream to write to the file with the specified **file name**.

Open in write mode, if boolean append = false
Open in append mode, if boolean append = true

```
FileOutputStream fos = new FileOutputStream("output.txt", append);
```

If file doesnt exists, this will create an empty file with the given fine name.

Write functions and Description

Eg: Write data using above created **fos** object

void write(int b)
Writes the specified byte to this file output stream.

```
FileOutputStream fos = new FileOutputStream("output.txt"));
fos.write(65);  // Writes the byte for ASCII 'A'
System.out.println("Single byte (A) written to the file.");
```

void write(byte[] b)
Writes b.length bytes from the specified byte array to this file output stream.

```
FileOutputStream fos = new FileOutputStream("output.txt"));
String message = "Genesis 1:1 In the beginning God created the heavens and the earth";

byte[] byteArray = message.getBytes();
fos.write(byteArray);
```

void write(byte[] b, int offset, int len)
Writes len bytes from the specified byte array starting at offset off to this file output stream.

```
FileOutputStream fos = new FileOutputStream("output.txt"));
String message = "Genesis 1:1 In the beginning God created the heavens and the earth";
```

byte[] byteArray = message.getBytes();
fos.write(byteArray, 12, message.length());

offset = 12, len = message.length() = 74
Therefore bytes between 12 to 74 will be written into the file.
That is " In the beginning God created the heavens and the earth"

2. BufferedOutputStream:

Wraps another OutputStream, such as FileOutputStream, to improve efficiency by buffering output data (reading larger chunks at once and reducing disk access).

It does not wrte to disk directly but uses FileOutputStream as its underlying source to write data to Disk file.

Eg Create Object of BufferedOutputStream

FileOutputStream fos = new FileOutputStream("output.txt");
BufferedOutputStream bos = new BufferedOutputStream(fos);

Can use any of the read functions given below

int read()
int read(byte[] b, int off, int len)

Eg. read data using above created ***bos*** object.

Using int write()

```
// Example 1: Write a single byte using write(int b)
int singleByte = 65; // ASCII value of 'A'
bos.write(singleByte);
```

Using int write(byte[] byteArray)

```
// Example 2: Write a byte array using write(byte[] b, int off, int len)
String data = "Genesis 1:1 In the beginning God created the heavens and the
earth";
byte[] byteArray = data.getBytes();
int offset = 0; // Start writing from beginning
int length = data.length(); // Write till end
bos.write(byteArray, offset, length);
```

3. DataOutputStream:

So far, the data is written as bytes. But some binary files (.bin or .dat extensions) need to write data as primitive data types (e.g., int, float, double, etc.). DataOutputStream is ueed to fullfil such requiremets. DataOutputStream also be used to write data to Network (sockets).

It does not write to the disk directly but uses FileOutputStream as its underlying source to write data to Disk file.

Constructor

```
public DataOutputStream(OutputStream out)
```

Write Functions

```
void write(byte[] b, int off, int len)
void write(int b)
void writeBoolean(boolean v)
void writeByte(int v)
void writeBytes(String s)
void writeChar(int v)
void writeChars(String s)
void writeDouble(double v)
void writeFloat(float v)
void writeInt(int v)
void writeLong(long v)
void writeShort(int v)
```

```
//Eg, write to a binary file
FileOutputStream fos = new FileOutputStream("binary_data.dat");
DataOutputStream dout = new DataOutputStream(fos);

dout.writeDouble(3.14);
dout.writeInt(100);
dout.writeBoolean(true);
dout.writeChar('A');
```

4. ObjectOutputStream:

ObjectOutputStream is used to serialize objects (save them in byte form). ObjectInputStream is used to deserialize objects (read them back into their original form).Both streams are typically used for persistent storage or object transfer in Java.

Eg: Serialize object of class Person

```
FileOutputStream fileOutputStream = new
FileOutputStream("serialized_object.dat");
ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream);

// Create an object to serialize
Person person = new Person("Alice", 30);

// Serialize and write the object to the file
objectOutputStream.writeObject(person);
```

5. ByteArrayOutputStream

This itself does not write directly to the file system. It writes data into an in-memory byte array (buffer). However, you can take the byte data stored in the ByteArrayOutputStream and manually write it to a file using other classes like FileOutputStream.

```
// Data to be written into ByteArrayOutputStream
String message = "Genesis 1:1 In the beginning God created the heavens and
the earth";
ByteArrayOutputStream baos = new ByteArrayOutputStream()) {

// Convert the string message to a byte array
byte[] byteArray = message.getBytes();
```

```
// Write the byte array into the ByteArrayOutputStream
baos.write(byteArray);

// Convert the ByteArrayOutputStream to byte array
byte[] outputBytes = baos.toByteArray();

// Now write the byte array to a file
FileOutputStream fos = new FileOutputStream("output.txt")
fos.write(outputBytes);
```

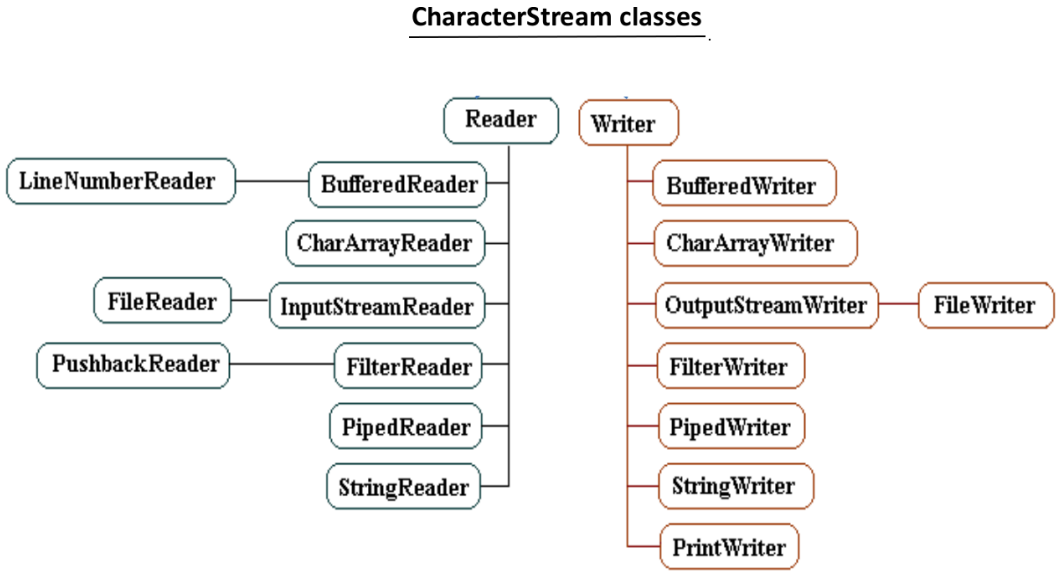
Character Stream Classes

Character streams are defined by using two class hierarchies.

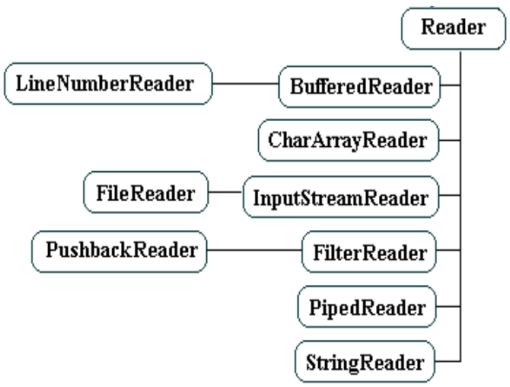
At the **top are two abstract** classes,

- Reader class
- Writer class .

Java has several concrete subclasses of each of these, as given in belo diagram



Reader Class Hiratchy



Some Important Classes are

1. FileReader
2. BufferedReader
3. InputStreamReader

1. FileReader

Only class capable of reading a disk file directly.
It read data as a sequence of 16 bit (2-byte) unicode characrer.
Suitable for reading languages other than English, for eg Chinese, Arabic etc.

Constructors

FileReader(File file) :
Creates a FileReader object by opening a connection to an actual file, the file named by the File object file in the file system.

Eg. opens a file input.txt to read

```
// create File object
File file = new File( "input.txt");
FileReader fin = new FileReader(file);
```

FileReader(String name)
Creates a FileReader by opening a connection to an actual file, the file named by the path name name in the file system

Eg. opens a file input.txt using file name to read instead of File object.

```
FileReader fin = new FileReader("input.txt");
```

Read functions and Description

*All below functions use **fin** object created above.*

int read()
Reads a character of data from this file reader.
Eg. read data using above created fis object

```
int data;
while ((data = fin.read()) != -1) {
        System.out.print((char)data);
}
```

int read(char[] cbuf)

```
char arr[] = new char[20];
int charRead = 0;
while ((charRead  = fin.read(arr)) != -1) {
    System.out.println(new String(arr));
}
```

Reads up to **charRead** bytes of data from this file reader
into an array of bytes.

int read(char[] b, int off, int len)
Reads up to **len** characters of data from this file reader into an array of bytes.

Eg. read data using above created fis object

```
char [] buffer = new char[1024]; // Define a buffer to hold the data
int charRead;
int offset = 0;
int length = 20; // Set the length to read
```

```
while (( charRead = fin.read(buffer, offset, length)) != -1) {
        // Convert the read bytes to a string and print them
        String output = new String(buffer, 0, charRead);
        System.out.println(output);
}
```

2. BufferedReader
is a Java class that reads text from an file, for eg, efficiently buffering the characters to improve performance. It's used to read data in large chunks rather than one character at a time, which can significantly speed up input operations, especially when reading from slower sources like files or network streams.

It does not read from the disk directly but uses **FileReader** as its underlying source to get data from Disk file.

Line-by-Line Reading: Provides a method **readLine()** that reads a full line of text at a time, making it easy to handle text input.

Note:
This is the **only class** that support **reading** as a **String** class , **String readLine()**

Constructors

BufferedReader(FileReader in)
Creates a buffering character-input stream that uses a default-sized input buffer.
BufferedReader(FileReader in, int sz)
Creates a buffering character-input stream that uses an input buffer of the specified size.

BufferedReader(FileReader in)
FileReader fis = new FileReader("malayalam.txt");
BufferedReader **reader** = new BufferedReader(fis);

Read functions and Description
*All below functions use **reader** object created above.*
int read()
int read(char[] array)
int read(char[] b, int off, int len)
String readLine()
Reads a line of text/String.

Eg, reads a malayalam file as lines/String.

```
String line;
while ((line = reader.readLine()) != null) {
        String str = line + "\n";
        System.out.println(str);
}
```

3. InputStreamReader
is a class in Java that serves as a **bridge between Byte streams and Character streams**. It reads bytes from an input source and decodes them into characters using a specified charset or the platform's default charset.

Converts byte streams (e.g., data from a file or network) into character streams.

Eg, reading a ASCII file, decoding into unicode character

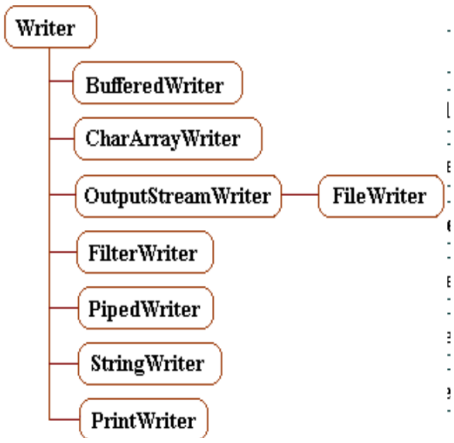
```
//create construcor
// FileInputStream is byte stream
InputStream inputStream = new FileInputStream("input.txt");

// its converted into Char stream, Specify character encoding UTF8
InputStreamReader inputStreamReader = new InputStreamReader(inputStream,
"UTF-8");

// Char Stream is wrapped by BufferReader
BufferedReader reader = new BufferedReader(inputStreamReader);

// reading file
String line;
while ((line = reader.readLine()) != null) {
        System.out.println(line);
}
```

Writer Class Hiratchy




```
Some Important classes
   1. FileWriter
   2. BufferedWriter
   3. OutputStreamWriter
```

1. FileWriter

Only class capable of writing a disk file directly.
It writes data as a sequence of 16 bit (2-byte) unicode characrer.
Suitable for writing Non ASCII languages, for Eg Chinese, Arabic etc.

Constructors

FileWriter(File file)
Creates a file output stream to write to the file represented by the specified File object. Open in write mode

```
File file = new File( "malayalam.txt");
FileWriter fw = new FileWriter(file);
```

FileWriter(File file, boolean append)
Open in write mode, if boolean append = false
Open in append mode, if boolean append = true

```
File file = new File( "malayalam.txt");
FileWriter fw = new FileWriter(file, append);
```

FileWriter(String name)
Creates a file output stream to write to the file represented by the specified file name.
FileWriter fw = new FileWriter(“malayalam.txt”);

FileWriter(String name, boolean append)
Creates a file output stream to write to the file with the specified file name.
Open in write mode, if boolean append = false
Open in append mode, if boolean append = true

FileWriter fw = new FileWriter("malayalam.txt", append);

If file doesnt exists, this will create an empty file with the given fine name.

write functions and Description

*All below functions use **fw** object created above.*

void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int off, int len)
void write(String cbuf)

void write(int c)
// Example 1: Write a single character using write(int c)
// int singleChar = 'ആ'; // Unicode for Malayalam character 'ആ'
//// also you can use corresponding hex values
int singleChar = 0x0D06; // Hexadecimal representation of ആ
fw.write(singleChar); // Writes the single character 'ആ'
fw.write('\n'); // Newline for clarity

void write(char[] cbuf)
// Example 2: Write an array of characters using write(char[] cbuf)
char[] verse1Array = "ആദിയിൽ ദൈവം ആകാശവും ഭൂമിയും സൃഷ്ടിച്ചു.\n".toCharArray();
// also you can use array of corresponding hex values
fw.write(verse1Array); // Writes the first verse as a char array

void write(char[] cbuf, int off, int len)
// Example 3: Write a portion of the character array using write(char[] cbuf, int off, int len)
// Writes first 10 characters of the first verse
fw.write(verse1Array, 0, 10);
fw.write('\n');

void write(String cbuf)
// Example 4: Write a full string using write(String str)
String verse2 = "ഭൂമി ശൂന്യവും പെരുപ്പും ആയിരുന്നു; അതിന്റെ മേൽ ഇരുട്ടും ആഴങ്ങളുടെ മേൽ ദൈവത്തിന്റെ ആത്മാവും ഉണ്ടായിരുന്നു.\n";
// also you can use array of corresponding hex values
// Writes the full string of verse 2
fw.write(verse2);

2. BufferedWriter
is a Java class that writes text from an file, for eg, efficiently buffering the characters to improve performance. It's used to write data in large chunks rather than one character at a time, which can significantly speed up input operations, especially when writing to slower sources like files or network streams.

It does not write tothe disk directly but uses **FileWriter** as its underlying source to purt data into Disk file.

Constructors

BufferedWriter(FileWriter in)
Creates a buffering character-input stream that uses a default-sized input buffer.
BufferedWriter(FileWriter in, int sz)
Creates a buffering character-input stream that uses an input buffer of the specified size.

***BufferedWriter*(*FileWriter* in)**
FileWriter fir = new FileWriter("malayalam.txt");
BufferedWriter **writer** = new BufferedReader(fis);

write functions and Description
*All below functions use **writer** object created above.*

void write(int c) : Writes a single character.
*void write(String s): w*rites a String.
*void write(String s, int off, int len): w*rites a portion of a String.
void write(char[] cbuf, int off, int len) : Writes a portion of an array of characters.

Eg. Write/append unicode char to a file.
void append(String str)
// Create a BufferedWriter wrapped around a FileWriter
BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));

// Write a line of text
writer.write("Hello, World!");
writer.newLine(); // Using newLine() method to add a newline

// Write multiple lines using the append method
writer.append("This is a line added with append()").append("\n");
writer.write("This is another line.");

3. OutputStreamWriter
is a class in Java that bridges byte streams and character streams. It is used to write characters to an output stream by converting them into bytes using a specified character encoding.

Eg Program, char array encoded into UTF 8 and write unicode file
String fileName = "output.txt";

char[] charArray= "Genesis1:1 ആദിയിൽ ദൈവം ആകാശവും ഭൂമിയും സൃഷ്ടിച്ചു";

```
String textToWrite  =  charArray.toCharArray();

// Create a FileOutputStream to write to the file
FileOutputStream fos = new FileOutputStream(fileName);

// Create an OutputStreamWriter with UTF-8 encoding
OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF-8");

// Write the text to the file
osw.write(textToWrite);
```

WORKING WITH FILES

1. Tiny Editor

Create a editor, that echows the keybord inputs and exit when “quit” is typed.

```
import java.io.*;
public class TinyEditor {
	public static void main(String[] args) {
		try {
			BufferedReader br = new BufferedReader(
					new InputStreamReader(System.in));
			System.out.println("Enter a line of text");
			System.out.println("Enter a quit to stop");

			String str[] = new String[100];
			for (int i = 0; i < 100; i++) {
				str[i] = br.readLine();
				if (str[i].equalsIgnoreCase("quit") == true) {
					break;
				}
			}
			System.out.println("Exit...!");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

2, Read an ASCII file and write to another file – Use byte stream classes

```
import java.io.*;
public class BytesStremTest {
	public static void main(String[] args) {
		try {
			FileInputStream fis = new FileInputStream("input.txt");
			FileOutputStream fos = new FileOutputStream("output.txt");
			byte[] bArr = new byte[50];
			int b = 0;
			while ( (b = fis.read(bArr)) != -1) {
				System.out.println(new String(bArr));
				fos.write(bArr);
			}
```

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

3. Read any file and write to another file – Use char stream classes

```
import java.io.*;
public class CharStreamTest {
	public static void main(String[] args) {
		try {
			FileReader fis = new FileReader("malayalam.txt");
			BufferedReader reader = new BufferedReader(fis);

			FileWriter fos = new FileWriter("utf_output.txt");
			BufferedWriter writer = new BufferedWriter(fos);

			while ((line = reader.readLine()) != null) {
				System.out.println(line);
				writer.write(line);
			}
			writer.close();
			fos.close();

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

4. Test DataInputStream & DataOutputStream

```
import java.io.*;
public class DataStreams {
	public static void main (String[] s) {
		try {
			//1. open a file
			File f1 = new File("out.dat");
			if (f1.exists() == false) {
				f1.createNewFile();
			}
			//2. create FileInputStream wrapped by DataInputStream
			FileOutputStream fos = new FileOutputStream(f1);
			DataOutputStream dout = new DataOutputStream(fos);

			//3. write primitive data (float, int, boolean, char)
			dout.writeDouble(3.14);
			dout.writeInt(100);
```

```
                dout.writeBoolean(true);
                dout.writeChar('A');

            // read data from the same file out.data
            //1. create FileOutputStream wrapped by DataOutputStream
            // using same file object f1

                    FileInputStream fis = new FileInputStream(f1);
                    DataInputStream din = new DataInputStream(fis);

                        //2 Read data
                    double dVal = din.readDouble();
                    int iVal = din.readInt();
                    boolean bVal = din.readBoolean();
                    char cVal = din.readChar();

                System.out.println("dVal = " + dVal);
                System.out.println("iVal = " + iVal);
                System.out.println("bVal = " + bVal);
                System.out.println("cVal = " + cVal);

                dout.close();
                fos.close();

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

5. University exam question...!

Enter a string via command line argument. Write the string into a file. Read it from the file and verify is it a palindrome. ?

// using Character stream class

```
import java.io.*;
public class PalindromeCheck {
        public static void checkPalindrome(String fname) {
                boolean flag = true;
                try {
                        BufferedReader reader = new BufferedReader(new FileReader(fname));
                        if(reader != null) {
                                String line = reader.readLine();
                                if (line  != null) {
                                        int len = line.length();
                                        for (int i=0; i<len/2; i++) {
                                                if (line.charAt(i) != line.charAt(len-1-i)) {
                                                        flag = false;
                                                        break;
                                                }
```

```
                        }
                    }
                    reader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            if (flag == true) {
                System.out.println("Entered string  is a Palindrome...!");
            } else {
                System.out.println("Entered string  is NOT a Palindrome...!");
            }
        }

        public static void writeString(String fname, String line) {
            try {
                BufferedWriter writer = new BufferedWriter (new FileWriter(fname));
                if (writer != null) {
                    writer.write(line);
                    writer.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        public static void main(String[] args) {
            String line;
            if (args.length > 0) {
               line = args[0];
               System.out.println("String =" + line);

               writeString("test.txt", line);
               checkPalindrome("test.txt");
            }
        }
}
```

How to run

```
$ java PalindromeCheck malayalam
String : malayalam
Entered string  is a Palindrome...!
```

6. University exam question...!

Enter a string via command line argument. Write the string into a file. Read it from the file and verify is it a palindrome. ? Do it using Byte stream classes.

```
import java.io.*;
public class PalindromeCheck {
        public static void checkPalindromeEx(String fname) {
            boolean flag = true;
            try {
                FileInputStream fis  = new FileInputStream(fname);
```

```
                    if(fis != null) {
                        byte [] b = new byte [1024];
                        int bRead = fis.read(b);
                        if (bRead  > 0) {
                           String line = new String(b, 0, bRead);
                           int len = line.length();
                              System.out.println("Line = " + line);
                              System.out.println("length = " + len);
                              for (int i=0; i<len/2; i++) {
                                 if (line.charAt(i) != line.charAt(len-1-i)) {
                                    flag = false;
                                    break;
                                 }
                              }
                        }
                        fis.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
                if (flag == true) {
                    System.out.println("Entered string  is a Palindrome...!");
                } else {
                    System.out.println("Entered string  is NOT a Palindrome...!");
                }
        }

public static void writeStringEx(String fname, String line) {
                try {
                    FileOutputStream fos = new FileOutputStream (fname);
                    if (fos != null) {
                        fos.write(line.getBytes());
                        fos.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
        public static void main(String[] args) {
                String line;
                if (args.length > 0) {
                   line = args[0];
                   System.out.println("String =" + line);
                   writeStringEx("test.txt", line);
                   checkPalindromeEx("test.txt");
                }
        }
}
```

Reading Console Input / Writing Console Output

Relevent classes

Reading Console Input classes	Writing Console Output classes
Scanner	*System.out (PrintStream)*
BufferedInputSream	*PrintWriter*
DataInputStream	*BufferedWriter*
BuferedReader	
InputStreamReader	

The Predefined Streams

System.in
System.out
System.err

Package java.lang defines a class called System.
System contains three predefined stream variables
– in, out, and err
These are standard input, output, error streams.
These are difined in Syatem class.

```
static InputStream  in;
static PrintStream  out;
static PrintStream  err;
```

Reading Console Input classes

1. **Scanner class**

package java.utils

```
/*------Class 1: Scanner class------*/
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your name: ");
// Read a line of text from the console
String name = scanner.nextLine();
// Display the input received
System.out.println("Hello, " + name + "!");
/*--------------------------------------*/
```

2. BufferedInputSream class

package java.io

```
/*------Class 2: BufferedReader class------*/
BufferedInputStream bis = new BufferedInputStream(System.in);
System.out.print("Enter data : ");
byte[] b = new byte[100];
bis.read(b);
System.out.println("You entered : " + new String(b));
/*--------------------------------------*/
```

3. BufferedReader class

package java.io

```
/*------Class 3: BufferedReader class------*/
```

```
InputStreamReader Isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(Isr);
System.out.print("Enter your name: ");
try {
        name = br.readLine();
        System.out.println("Hello, " + name + "!");
} catch (IOException e) {
System.out.println(e.getMessage());
}
/*-------------------------------------------------*/
```

4. DataInputStream class

```
package java.io
/*------Class 4: DataInputStream class------*/
DataInputStream dis = new DataInputStream(System.in);
System.out.print("Enter your name: ");
try {
        name = dis.readLine();
} catch (IOException e) {
System.out.println(e.getMessage());
}
System.out.println("Hello, " + name + "!");
/*-------------------------------------------------*/
```

5. **InputStreamReader class**

```
package java.io
/*------Class 5: DataInputStream class------*/
InputStreamReader Isr1 = new InputStreamReader(System.in);
char[] c = new char[20];
try {
        Isr1.read(c);
        System.out.println("Hello, " + new String(c));
} catch (IOException e) {
System.out.println(e.getMessage());
}
/*-------------------------------------------------*/
```

Note

*Why do we **wrap** InputStreamReader in BufferedReader as below*

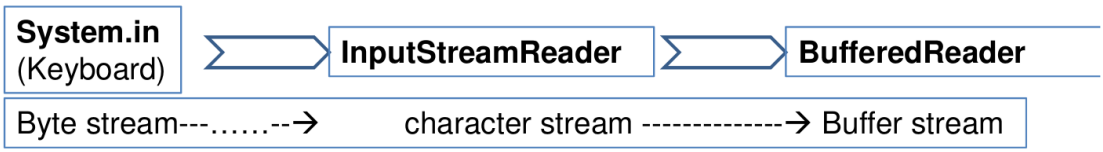
```
InputStreamReader Isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(Isr);
```

Ans:

refer

```
InputStreamReader Isr = new InputStreamReader(System.in);
```

*Here **System.in** is keyboard, which **byte stream**, that is transfer data in **bytes***
*BufferedReader is used to improve efficiency, but it works with **Character stream**, which can not process bytes.*
*Hence, InputStreamReader is **converts byte stream into character stream**, so that BufferedReader can be used to improve efficiently.*



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

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

Writing Console Output classes

1. **System.out (PrintStream) class**

package java.utils

That is, 'out' is an object of PrintStream class in System class.

```
/*Class 1: System.out------------*/
// Print with a newline
System.out.println("This is a message with a newline.");
// Print formatted text using printf
int number = 5;
System.out.printf("Formatted output: The number is %d \n", number);
//Print without a newline
System.out.print("This is a message without a newline.");
/*-----------------------------------*/
```

2. **PrintWriter class**

package java.io

```
/*Class 2: PrintWriter------------*/
PrintWriter pWriter = new PrintWriter(System.out, true);
pWriter.println("This is a message using PrintWriter.");
pWriter.printf("Formatted message: %d apples.%n", 5);
/*--------------------------------*/
```

3. **BufferedWriter class**

package java.io

```
/*Class 3: BufferedWriter------------*/
OutputStreamWriter Osw = new OutputStreamWriter(System.out);
BufferedWriter bWriter = new BufferedWriter(Osw);
try {
        bWriter.write("BufferedWriter example: writing to console.");
        bWriter.newLine();
        bWriter.write("This is another line.");
        bWriter.flush();
} catch (IOException e) {
System.out.println(e.getMessage());
}
```

PrintWritter Class

PrintWriter used for console output in Java

PrintWriter(System.out, true): This creates a PrintWriter object that wraps System.out. The second argument (true) ensures that the stream automatically flushes the buffer after each write operation (e.g., after println() or printf()).
writer.println(): This writes a line of text with a newline.
writer.printf(): This allows formatted text output (like System.out.printf()), useful for variables and formatted data.
writer.print(): This prints text without appending a newline at the end.

```
// Create a PrintWriter that writes to the console (System.out)
PrintWriter writer = new PrintWriter(System.out, true); // 'true' enables auto-flush

// Write a simple message to the console
writer.println("This is a message written using PrintWriter.");

// Use formatted output with printf (similar to System.out.printf)
writer.printf("Formatted output: %d apples and %d oranges.%n", 5, 10);

// Use print (without newline)
writer.print("This is a single line printed without a newline.");

// Close the PrintWriter (optional when using System.out)
writer.close();
```

Note
what will happen if set auto-flush is set to false as below
*PrintWriter writer = new PrintWriter(System.out, **false**); // 'true' enables auto-flush*
Ans:
Without flush(), Output Will Not Appear Until Closing:
*If you don't call **flush()** or **close()**, the output will remain buffered and will not appear on the console immediately.*

System.out Vs PrintWriter

System.out

- System.out is a byte stream.
- **System.out** refers to the standard output stream(monitor).
- **System:** It is a final class defined in the java.lang package.
- **out:** This is an instance of PrintStream type, which is a public and static member field of the System class.

PrintWritter

- **PrintWriter** should be used to write a stream of characters
- PrintWriter is a subclass of **Writer** (character stream class)
- It is used in real world programs to make it easier to internationalize the program

Object Streams and Serialization

Serialization is the process of writing(converting) the state/data of an object to a byte stream.

– This is useful when we want to
  • save(store) the state/data of the program to file or
  • when we want to send it over network.

– The ObjectOutputStream is used to serialize an object, for eg **student** object of **Student class** and write it to a file, for eg, **student.ser**

Deserialization.

– Deserialization converts the stored byte streams in file to object
  The ObjectInputStream is used to read the object back from the file.
  The byte stream is converted back into a Person object.

The Person class implements **Serializable**, which is required for objects to be serialized in Java. The **Serializable** interface has no methods or fields and serves only to identify the semantics of being serializable

Step 1: Create a class, for eg Sudent implementing Serializable

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

Step 2: Create a test class to do serialization

```
public class SerializationExample {
   public static void main(String[] args) {
      // Serialize the object
      try {
            FileOutputStream fos = new FileOutputStream("student.ser");
            ObjectOutputStream out = new ObjectOutputStream(fos);
            Student student = new Student(1, "Alice");
            out.writeObject(student);
            System.out.println("Student object serialized and saved to student.ser");
      }catch (IOException e) {
            System.out.println(e.getMessage());
      }
```

```
        // Deserialize the object
        try {
                FileInputStream fis = new FileInputStream("student.ser");
                ObjectInputStream Oin= new ObjectInputStream(fis);
                Student deserializedStudent = (Student) Oin.readObject();
                System.out.println("Deserialized Student: ID - " +
                                        deserializedStudent.getId());
                System.out.println("Name - " + deserializedStudent.getName());
           } catch(IOException e) {
                System.out.println(e.getMessage());
           } catch (ClassNotFoundException e) {
                System.out.println(e.getMessage());
           }
    }
}
```

EXCEPTIOS...!

Exception

```
try {
        System.out.println("Inside try Block");
        data = num / 0;
} catch (ArithmeticException e) {
        System.out.println("Inside Catch Block");
        System.out.println(e.getMessage());
} finally {
        System.out.println("Inside Finally Block");
}
```

An Exception is an **abnormal condition** that occur in a code at **run time**.

– Exception is a RUN TIME ERROR.

Eg, Zero division error on Integers.

• A Java **exception is an object** that describes an exceptional condition (that is, error) that occurred in a piece of code during **run time**. If the exception is NOT handled in a **catch block**, the program will crash.

How Java handle exception

1. When an Exception occurs, JVM detects it and create corresponding Exception class objects, for eg, if Zero division on integers, JVM creates object of class ArithmeticException.

*2. Then the JVM halts the normal execution and **throws** the created object to the method it created.*

*3. JVM first check, if the function created exception has a matching **catch** block implemented, handle it there, so that prgram wont crash, as given below.*

```
try {
        System.out.println("Inside try Block");
        data = num / 0;
} catch (ArithmeticException e) {
        System.out.println("Inside Catch Block");
        System.out.println(e.getMessage());
}
```

*4. If a matching **catch** block is **not available**, JVM passes (propogates) the exception object to the method that called it. (previous method in the Call stack).*

*5. if still matching catch is not avaiable, propogate to the next function in the Call stack, this continue untill it unwinds the call stack, that till reached main(). This is called **exception propogation**.*

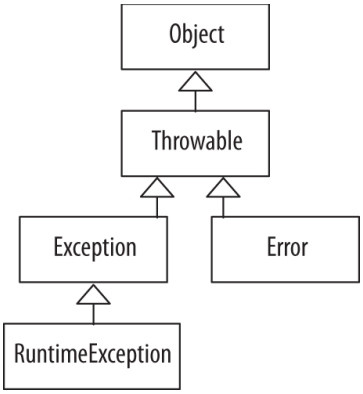
6. If the matching catch is not found in Call stack, program terminates.

Example.

```
public static int devideProc(int num, int den) {
        return num / den;
       /* ArithmeticException object will be created
              and propogated/thrown to caller main() */
}
public static void main(String[] args) {
      try {
```

```
            int n = devideProc(5,0);
        } catch(ArithmeticException e) {
            /* ArithmeticException  object will be trapped/handled
                    here in matching catch block – program will NOT crash */
            System.out.println(e.getMessage());
        }
    }
```

Exception Class Hirarchy



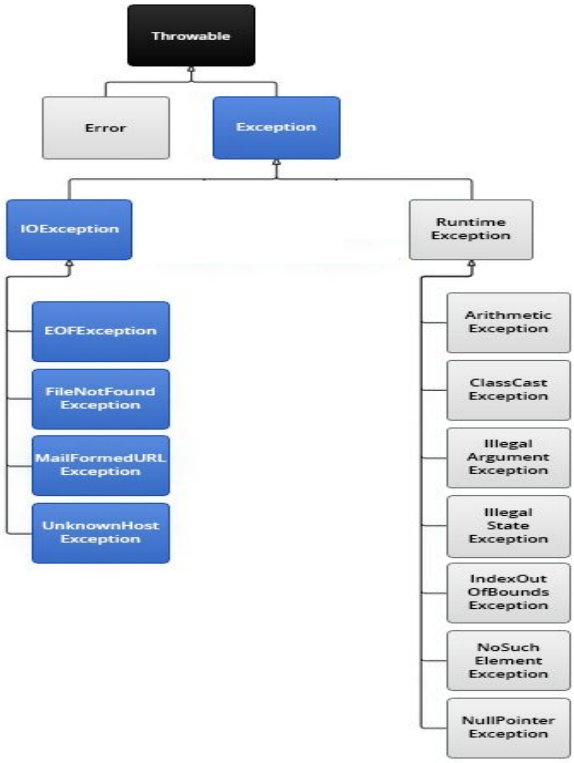
Object: The root class of the Java class hierarchy; every class in Java inherits from Object.

Throwable: The superclass of all errors and exceptions in Java.

Exception: Represents exceptions that are conditions a program might want to catch and handle, such as I/O errors.

RuntimeException: A subclass of Exception for exceptions that occur during the program's runtime, typically due to programming errors like NullPointerException.

Error: Represents serious issues beyond the program's control, often related to the JVM environment, such as OutOfMemoryError. ***It can not be handled,*** *Results in program crash*.



Type Of Exceptions

1. **Checked Exception –** Checks at Compile time.
   If not handled, will get compilation error.
2. **Unchecked Exception –** Checks at Runtime,
   If not handled, will get runtime error, and program ends.

Checked exceptions	Unchecked exceptions
• Checked at compile time.(COMPILE TIME EXCEPTIONS)	• NOT checked at compile time.(RUN TIME EXCEPTINS)
• Not sub class of RunTimeException	• Sub class of RunTimeException
• The method must either handle the exception or it must specify the exception using *throws* keyword.	• It is NOT needed to handle or catch these exceptions
• Shows compile error if checked exception is not handled.	• DO NOT Show compile error if exception is not handled. But shows run-time error.
• E.g. *ClassNotFoundException, IOException*	• Eg. *ArithmeticException, ArrayIndexOutOfBoundsException*

Exception handling fundamentals

5 Key words

1. **try**
2. **catch**
3. **finally**
4. **throws**
5. **throw**

1. try

In exception handling, the try block is used to define a section of code where exceptions might occur.

It ensures that if an exception occurs, the program can transfer control to a corresponding catch block to handle the exception, preventing the program from crashing.

2. catch

In exception handling, the catch block is used to handle exceptions that are thrown from a corresponding try block.

When an exception occurs in the try block, the catch block is used to define what should happen in response to that specific type of exception.

The catch block handles the exception, allowing the program to continue running smoothly after an error is detected in the try block.

3. finally

In exception handling, the finally block is used to define code that will always be executed, whether an exception is thrown or not.

To ensure that certain code runs no matter what happens during the try-catch execution., fo eg, certain cleanup tasks such as closing resources.

Format:

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
} finally {
   // Code that always executes
}
```

Example:

```
try {
   int[] numbers = {1, 2, 3};
   System.out.println(numbers[5]);
   // This will throw ArrayIndexOutOfBoundsException
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Exception caught: " + e.getMessage());
} finally {
   System.out.println("This block is always executed.");
}
```

4. throws

is used in a method declaration when the method **chooses not to handle potential exceptions itself** within the method.

It indicates that the method may throw one or more exceptions during execution.

It informs the compiler and the caller that they need to be aware of these exceptions and be prepared to handle them:

Either by using a try-catch block to catch the exception.
Or by letting the exception propagate further up the call stack,
possibly to another caller.

This approach is typically used with checked exceptions, which must be either caught or declared in the method signature.

Example:

```
class ThrowsTest {
    public static int divisionTest(int a, int b) throws ArithmeticException {
        return a/b;
    }
    public static void main(String[] args) {
        try {
            divisionTest(5,0);
```

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

5. throw

In Java, the throw keyword is used to explicitly throw an exception from a method or a block of code.

Once the exception is thrown, the program doesn't proceed with normal execution until the exception is either handled by an appropriate catch block in the call stack, or the program terminates if the exception remains unhandled.

```
class ThrowTest {
    public static int divisionTest(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Zero Division Error");
        }
        return a/b;
    }
    public static void main(String[] args) {
        try {
            divisionTest(5,0);
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Creating User defined Exception

In java we can create custome or user defined exception. For eg, the requirement is to create an exception, say InvalidAgeException, when the entered age is below 18 or a negative value etc. The steps involved are

Example:

```
// Custom checked exception by extending Exception class
class InvalidAgeException extends Exception {
    // Constructor with a custom message
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class TestCustomException {
    // Method that throws the custom checked exception
    public static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age less than 18 is not allowed.");
        } else {
            System.out.println("Valid age");
```

```
        }
    }

    public static void main(String[] args) {
        try {
            // This will throw the custom InvalidAgeException
            checkAge(16);
        } catch (InvalidAgeException e) {
            // Handle the custom exception
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

Multiple Catch clauses

In some cases, more than one exception could be raised by a single block of code.
For eg, *String num1 = "**one**", String den = "0";*
int num1 = Integer.parseInt(num);
int den1 = Integer.parseInt(den);
int div = num/den;
Above code snippet can generate NumberFormatException and ArithemticException.

To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.

After one catch statement executes, the others are bypassed, and execution continues after the try / catch block.

Exception Order in Catch
Subclasses must come before any of their superclasses

This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error, and result in COmpilation Error

```
class MultipleCatch {
    static void testMultipleCatch(String num, String den) {
        try {
            int num1 = Integer.parseInt(num);
            int den1 = Integer.parseInt(den);
            int res = num1/den1;
        } catch (NumberFormatException e1) {
            System.out.println(e1.getMessage());
        }catch (ArithmeticException e2) {
```

```
                System.out.println(e2.getMessage());
            }
        }
        public static void main(String[] args) {
            // pass string values
            testMultipleCatch("10", "0");
        }
}
```

Order of Exception in Catch

Subclasses *must come* ***first*** *before any of their* ***superclasses***

This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error, and result in ***Compilation Error***

Nested try block

The try statement can be nested within another try.

When entering a try statement, the context of that exception is added to a **stack**.

If an inner try lacks a handler for a specific exception, the stack is checked, and the next try's handlers are inspected.

This process continues until a matching catch is found,
or all nested try blocks are checked.

If no catch matches, the Java runtime system handles the exception and program terminates.

```
/*
 * If an inner try lacks a handler for a specific exception,
   the stack is checked, and the next try's handlers are inspected.
   here res = num1/den1; is not handled in inner try,
   but eventually its get handled in outer exception...!
 */
class MultipleTry {
static void testMultipleTry(String num, String den) {
        try {
                int num1 = Integer.parseInt(num);
                int den1 = Integer.parseInt(den);
                try {
                        int res = num1/den1;
                }catch (NullPointerException e2) {
                        System.out.println("Exception-Inner ..! "+ e2.getMessage());
                }
        } catch (Exception e1) {
                System.out.println("Exception-Outer ..!" + e1.getMessage());
        }
```

```
    }

    public static void main(String[] args) {
            testMultipleTry("10", "0");
        }
    }
```

Output:

```
(base) administrator@administrator-Lenovo-ThinkBook-14-IML:/u/JavaTest$ java MultipleTry
Exception-Outer ..!/ by zero
(base) administrator@administrator-Lenovo-ThinkBook-14-IML:/u/JavaTest$ ~
```

Note: int res = num1/den1; is **not handled in inner try**, but eventually its get handled in **outer exception...**!

University exam questions

1. Differentiate between checked and unchecked exceptions in Java with examples.?

Ans: *refer "Type Of Exceptions" in this doc*

2. Demonstrate the significance of the keywords 'try', 'catch', 'finally', 'throw' and 'throws' in exception handling of Java with appropriate examples.

Ans: refer "Exception handling fundamentals" in this doc.

3. What is exception? List any four exception classes in Java.

Ans*: An Exception is an abnormal condition that occur in a code at run time.*
– Exception is a RUN TIME ERROR.
Eg, Zero division error on Integers.

• A Java exception is an object that describes an exceptional condition (that is, error) that occurred in a piece of code during run time. If the exception is NOT handled in a catch block, the program will crash.

Classes : IOException, FileNotFoundException – these are checked exceptions
ArithmeticException, NumerFormatException – these are unchecked exception

4. Briefly explain various exception handling keywords in Java, with examples.

Ans: refer "Exception handling fundamentals" in this doc.

5. Explain in detail how exception handling mechanism used in Java using BufferedReader class

Ans

1. FileNotFound Exception – in opening a file, that is creting a reader object. This will be throwned, if the file does not exist.

2. IOException: in reading data using reader object. This is because, input source can result in various I/O errors (e.g., file not found, stream closed, network issues, etc.).

```
class BufferedReaderTest {
    void readFile(String filename) {
        FileReader freader = null;
        try {
            freader = new FileReader(filename);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        BufferedReader reader = new BufferedReader(freader);
        try {
            String str = reader.readLine();
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
public class Lab_eception {
        public static void main(String[] args) {
                BufferedReaderTest brt = new BufferedReaderTest();
                brt.readFile("input.txt");
        }
}
```

6. How user defined eception is created ?. ==> important question

Ans*: Refer "Creating User defined Exception" in this doc*

7. Write a Java program to read characters from the console using 'throw' and 'throws

Ans:

```
import java.io.*;
class ConsoleReader {
        static void readFunction() throws IOException{
                InputStreamReader irs = new InputStreamReader(System.in);
                BufferedReader reader = new BufferedReader(irs);
                System.out.print("Enter a string :");
                String str = reader.readLine();
                if(str.isEmpty() || str.length() <= 0) {
                        throw new IOException("Exception:/read empty
string...!");
                }
        }
        public static void main(String[] args) {
                try {
                        readFunction();
                } catch (IOException e) {
                        System.out.println(e.getMessage());
                }
        }
}
```

8. **Explain the scenario under which the following three exceptions occur, NumberFormatException, ArithmeticException, and ArrayIndexOutOfBoundsException.**

Ans:

```
ArithmeticException:
        int x = 5/0;
NumberFormatException
        String strNumber = "one";
        int digit = Integer.parseInt( strNumber);
ArrayIndexOutOfBoundsException:
        int[] arr = new int[3];
        arr[5] = 100;
```

9. What is an exception? Why it needs to be handled?

Ans: refer question 3

If exception is not handles, program will crash for unchecked exception.

If exception is not handles, compilation error for checked exception.

10: illustrate order of exception in multiple catch blocks ?

Ans: refer “Multiple Catch clauses” in this doc.

University exam questions – Module 3

1. Describe various methods of reading data from the keyboard with appropriate examples in Java

Ans:

1. Scanner class

```
package java.utils
/*------Class 1: Scanner class------*/
Scanner scanner = new Scanner(System.in);
System.out.print("Enter your name: ");
// Read a line of text from the console
String name = scanner.nextLine();
// Display the input received
System.out.println("Hello, " + name + "!");
/*---------------------------------------*/
```

2. BufferedReader class

```
package java.io
/*------Class 2: BufferedReader class------*/
InputStreamReader Isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(Isr);
System.out.print("Enter your name: ");
try {
name = br.readLine();
System.out.println("Hello, " + name + "!");
} catch (IOException e) {
System.out.println(e.getMessage());
}
/*-------------------------------------------------*/
```

3. DataInputStream class

```
package java.io
/*------Class 3: DataInputStream class------*/
DataInputStream dis = new DataInputStream(System.in);
System.out.print("Enter your name: ");
try {
name = dis.readLine();
} catch (IOException e) {
System.out.println(e.getMessage());
}
System.out.println("Hello, " + name + "!");
/*-------------------------------------------------*/
```

4. InputStreamReader class

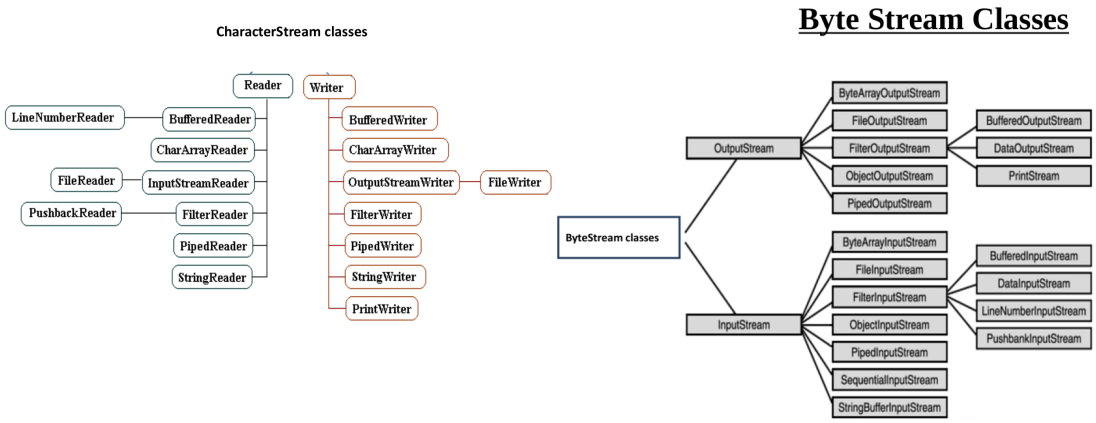
```
package java.io
/*------Class 3: DataInputStream class------*/
InputStreamReader Isr1 = new InputStreamReader(System.in);
char[] c = new char[20];
try {
Isr1.read(c);
System.out.println("Hello, " + new String(c));
} catch (IOException e) {
```

```
System.out.println(e.getMessage());
}
/*-----------------------------------------------*/
```

2. Demonstrate the significance of the keywords 'try', 'catch', 'finally', 'throw' and 'throws' in exception handling of Java with appropriate examples.

Ans: refer "Exception handling fundamentals" in this doc.

3. Write a note on byte stream and character stream related classes.



***Definition: Byte streams** are used to perform input and output of 8-bit bytes. They are suitable for handling raw binary data such as image files, audio files, or any other binary file formats.*

***Definition: Character streams** are used to handle 16-bit Unicode characters, making them suitable for handling text data. Character streams are intended for text-based files such as .txt, .xml, or .html.*

4. Write a Java program that accepts N integers through console and compute their average.

```
import java.util.Scanner;
public class CalcAvg {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter the number of integers (N): ");
      int N = scanner.nextInt();
      int sum = 0;
      for (int i = 1; i <= N; i++) {
         System.out.print("Enter integer " + i + ": ");
         int num = scanner.nextInt();
         sum += num;
      }
      double average = (double) sum / N;
      System.out.println("The average is: " + average);
      scanner.close();
   }
}
```

5. Develop a java package named primepackage, with a class Prime containing a static method that check whether a number is prime or not and returns that information. Import this package in another class and use to check a number is

prime or not.

Ans

1. create a folder primepackage in woring folder.

2. create a class Prime.java inside primepackage folder.

```
package primepackage;
public class Prime {
        public static boolean isPrime(int N) {
                boolean retVal = false;
                if ( N<=1 ) {
                        return false;
                }
                if (N==2) {
                        return true;
                }
                for (int i=2; i<=N/2; i++) {
                        if (N % i == 0) {
                                return false;
                        }
                }
                return true;
        }
}
```

2. create a Test class in working folder as below importing primepackage.

```
import primepackage.*;
public class PrimeTest {
public static void main(String[] s) {
        int N = 5;
                if (true == Prime.isPrime(N)) {
                        System.out.println(N + " is Prime");
                } else {
                        System.out.println(N + " is NOT Prime");
                }
        }
}
```

6. Model a Java class in such a manner that it is restricted to have only one instance throughout the program in which it is used.

This is a good question.

*This is a **Design pattern** called **Singleton**, where only one instance is created for all users.*

```
class Singleton {
   private static Singleton instance;
   private Singleton() {
      System.out.println("Singleton Constructor...!");
   }
   public static Singleton getInstance() {
        if (instance == null) {
                instance = new Singleton();
        }
        return instance;
   }
   public void printMessage() {
      System.out.println("printMessage() hashcode : " + this.hashCode());
```

```
    }
}
public class SingleTonTest {
        public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();
        singleton1.printMessage();
        singleton2.printMessage();
        }
}
```

Output:

```
Singleton Constructor...!
printMessage() hashcode : 312116338
printMessage() hashcode : 312116338
```

7. Can a class in Java implement more than one interfaces, if yes what is the syntax used?

Eg, Multiple Inheritance in java

```
interface Animal {
 void eat();
}
interface Bird {
void fly();
}
class Bat implements Animal, Bird {
        // Implementing abstract method from Animal interface
        public void eat() {
        System.out.println("Bat eats insects.");
        }
        // Implementing abstract method from Bird interface
        public void fly() {
        System.out.println("Bat can fly.");}
        }
public class Main {
public static void main(String[] args) {
        Bat bat = new Bat();
        bat.eat(); // From Animal interface
        bat.fly(); // From Bird interface
}
}
```

8. Write a Java program to create a new file named 'MyFile.txt' and write the statement "This is the University Exam for OODP. This a program to illustrate the use of files." into the file with each sentence in the statement representing a new line in the file

```
import java.io.*;
public class MyFileTest {
        public static void main(String[] args) {
                BufferedWriter bw = null;
                try {
```

```
                    bw = new BufferedWriter( new FileWriter("MyFile.txt"));
                    bw.write("This is the University Exam for OODP.");
                    bw.write("\n");
                    bw.write("This a program to illustrate the use of files.");
                    bw.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
}
```

9. Define two user defined exception EvenNumberException' and 'OddNumberException'. Write a Java class which has a method which checks whether a given number if even or not. The method throws 'EvenNumberException' or 'OddNumberException' if the number is even or odd respectively. Illustrate the handling of the exception with suitable sequence of codes.

```
class EvenNumberException extends Exception {
        EvenNumberException (String str) {
                super(str);
        }
        public String getMsg() {
                return "EvenNumberException".toString();
        }
}
class OddNumberException extends Exception {
        OddNumberException (String str) {
                super(str);
        }
        public String getMsg() {
                return "OddNumberException".toString();
        }
}
public class NumberExcpetionTest {
        public static void checkNumber(int N)
                        throws EvenNumberException, OddNumberException {
                if (N % 2 == 0) {
                        throw new EvenNumberException("Even number...!");
                } else {
                        throw new OddNumberException("Odd number...!");
                }
        }
        public static void main(String[] args) {
                int N = 100;
                try {
                        checkNumber(N);
                } catch(EvenNumberException e) {
                        System.out.println(e.getMessage());
                } catch(OddNumberException e) {
                        System.out.println(e.getMessage());
                }
                N = 99;
                try {
                        checkNumber(N);
```

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

Output:

```
Even number...!
Odd number...!
```

10. Explain the scenario under which the following three exceptions occur, NumberFormatException, ArithmeticException, and ArrayIndexOutOfBoundsException.

```
public class ExcetionTest {
        public static void main(String[] args) {
                // NumberFormatException
                String num = "one";
                try {
                        int n = Integer.parseInt(num);
                } catch(NumberFormatException e) {
                        System.out.println(e.getMessage());
                }
                // ArithmeticException
                int a = 10, b = 0;
                try {
                        int n = a/b;
                } catch(ArithmeticException e) {
                        System.out.println(e.getMessage());
                }
                // ArrayIndexOutOfBoundsException
                try {
                        int[] arr = new int[3];
                        arr[0] = 0;
                        arr[1] = 1;
                        arr[2] = 2;
                        arr[3] = 3;
                } catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println(e.getMessage());
                }
        }
}
```

Output:

```
For input string: "one"
/ by zero
Index 3 out of bounds for length 3
```