

Unit 3 - Exception Handling and I/O

1.Exception Hierarchy

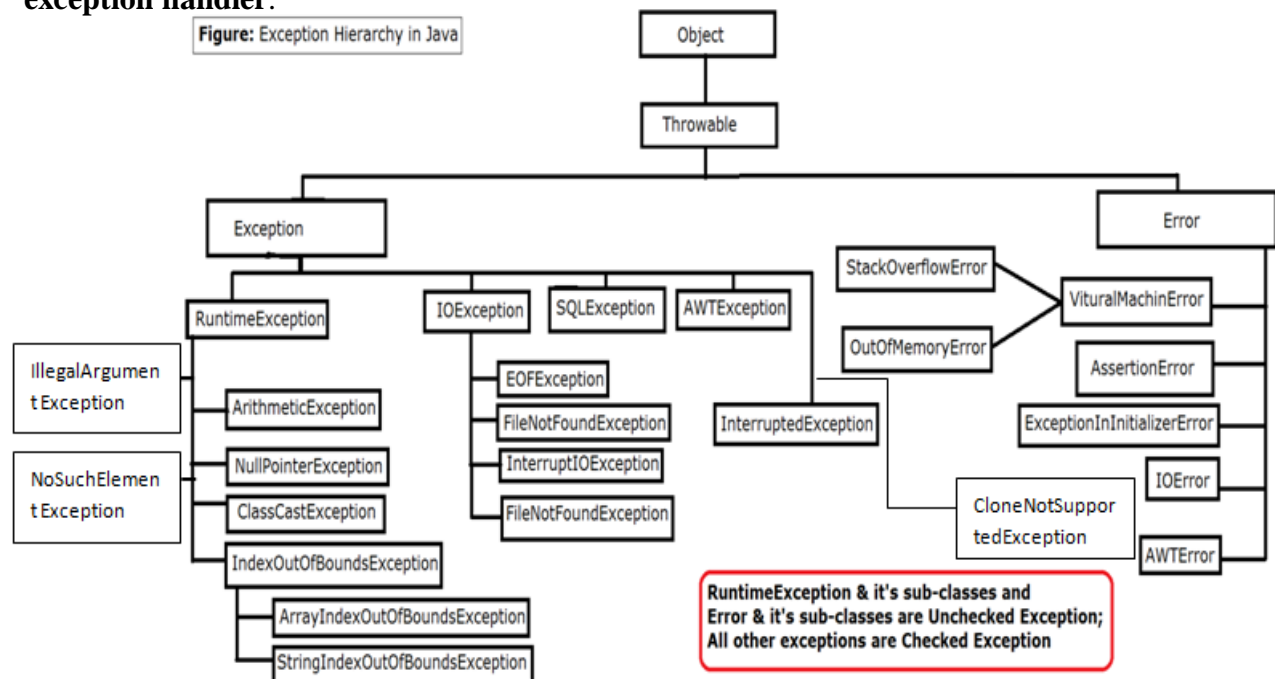
Exception is an event that occurs **during** the **execution** of the program (run time). In java, exceptions are objects. | Exception is also called as **abnormal** termination of a program.

Step1: When an error occurs within a method, it creates an object and handover it to the runtime system. This object is called **exception object** which contains information about error, its type, state of program.

Creating an exception object and handling it to run time system is called **throwing an exception**.

Step2: the runtime system attempts to find something (list of methods) to handle it. This is called **call stack**.

Step3: the call stack method contains block of code to handle exception. This code is called **exception handler**.



The **throwable** class is the superclass of all errors and exceptions. Instances of two subclasses Error and Exception are used to indicate exceptional situations. **Java.lang.Object** | **Java.lang.Throwable**

In other words, throwable class contains a snapshot of execution stack.

A throwable class extends the properties of Object class and also implements the serializable interface.

Syntax: public class Throwable extends Object implements Serializable

Constructors of Throwable class

Throwable() | Throwable(String msg) | Throwable(String msg, Throwable cause)

1.1.Types of exceptions

Two types, **1. Checked Exception (Compile time)**

These exceptions checked by the code itself. Using try-catch or throws i.e compiler will check these exceptions. From **java.lang.Exception** class. Ex: IOException

2. Unchecked Exception (Run time)

These exceptions are not checked by compiler. JVM will check these exceptions. From **java.lang.RuntimeException** class. Ex: ArrayIndexOutOfBoundsException, RuntimeException.

1.2. Using Multiple Catch Clauses

To catch different types of exceptions multiple catch clause can be used.

Example:

```
public class TestMultipleCatchBlock{
    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(ArithmeticException e){
            System.out.println("task1 is completed");}
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println("task 2 completed");}
        catch(Exception e){
            System.out.println("common task completed");}
        System.out.println("rest of the code...");
    }
}
```

Output:task1 completed
rest of the code...

1.3. Try block can be nested

A try, catch or finally block can contain another set of try catch and finally sequence.

Example:

```
class Excep6{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b =39/0;
            }catch(ArithmeticException e)
            { System.out.println(e);}

            try{
                int a[]=new int[5];
                a[5]=4;
            }catch(ArrayIndexOutOfBoundsException e)
            { System.out.println(e);}

            System.out.println("other statement");
        }catch(Exception e)
        { System.out.println("handeled");}

        System.out.println("normal flow..");
    }
}
```

Output: going to divide
 Java.lang. ArithmeticException: /by zero
 Java.lang. ArrayIndexOutOfBoundsException: 5
 Other statement
 Normal flow..

2. Throwing and Catching Exceptions

Five keywords to handle exception,

1. **Try** – to monitor exception | to try critical block
2. **Catch** – handles specific exception with try block
3. **Finally** – code executed even exception may or may not occur. Denotes end of the program. | optional to use |
4. **Throw** – used to throw specific exception
5. **Throws** - used to throw specific exception by a particular method.

a). Example : try-catch-finally keyword

class myException

```
{    //static int i=5;
    //static int j = 0;
    public static void main(String s[])
    {
        int i=5, j=0;
        System.out.println("Try started");
        try
        {
            int temp = i/j;
            System.out.println("Inside try");
        }
        catch(Exception e)
        {
            System.out.println("Inside catch");
            System.out.println("Divide by 0");
        }
        finally
        {
            System.out.println("Finally block");
        }
    }
}
```

Output: try started
 Inside catch
 Divide by 0
 Finally block

Difference between Throw and Throws

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

b).Example using throw #UserDefined Exception

```
public class Example
{
    void checkAge(int age)
    { if(age<18)
        throw new ArithmeticException("Not Eligible for voting"); //inside Method //UserDefined
    else
        System.out.println("Eligible for voting");
    }
    public static void main(String args[])
    {
        Example1 obj = new Example1();
        obj.checkAge(13);
        System.out.println("End Of Program");
    }
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException:
Not Eligible for voting
at Example1.checkAge(Example1.java:4)
at Example1.main(Example1.java:10)

c).Example using throws

```
public class Example1
{
    int division(int a, int b) throws ArithmeticException //Method signature, Supports Multiple Exception
    { int t = a/b;
      return t;
    }
    public static void main(String args[])
    {
        Example1 obj = new Example1();
        try{
            System.out.println(obj.division(15,0));
        }
        catch(ArithmeticException e)
        {
            System.out.println("You shouldn't divide number by zero");
        }
    }
}
```

Output:

You shouldn't divide number by zero

Points to remember: Java.lang – class, data types
Java.util – collections framework – scanner, ArrayList, etc.,
Java.IO – API to read n write data.

3.Built in Exceptions

S.No	Exception	Description
1.	ArithmeticException	This caused by error in math operation. E.g divide by zero
2.	NullPointerException	Causes when access to object with Null reference
3.	IOException	Occurs when illegal I/O performed
4.	IndexOutOfBoundsException	Caused when Array index out of limit/bound
5.	ArrayIndexOutOfBoundsException	Caused when Array index out of limit/bound
6.	ArrayStoreException	Occurs when a wrong object stored
7.	EmptyStackException	Occurs while popping empty stack
8.	NumberFormatException	Occurs when converting string to number
9.	RuntimeException	General run time error
10.	Exception	General exception
11.	ClassCastException	Occurs while casting object. i.e Decimal to int
12.	IllegalStateException	Occurs when illegal method invoked

What is exception chaining?

Whenever in a program, the first exception causes another exception is called exception chaining.

Difference between next() and nextLine()

next() can read the input only till the space. It can't read two words separated by space. Also, **next()** places the cursor in the same line after reading the input. **nextLine()** reads input including space between the words (that is, it reads till the end of line n).

Example:

```
import java.util.Scanner;
class zzz
{
    public static void main (String args[])
    {
        String s;
        Scanner in =new Scanner(System.in);
        System.out.println("Enter string:"); //Welcome to Builders Engineering College
        s=in.nextLine(); //s=in.next();
        System.out.println("Output:" +s);
    }
}
```

What is integer Parseint in Java? (String to Int)

The **Integer.parseInt()** java method is used primarily in **parsing** a String method argument into an **Integer** object.

Is BufferedReader faster than scanner?

The **Scanner** has a little buffer (1KB char buffer) as opposed to the **BufferedReader** (8KB byte buffer), but it's more **than** enough. **BufferedReader** is a bit **faster** as compared to **scanner** because **scanner** does parsing of input data and **BufferedReader** simply reads sequence of characters.

4.Creating Own Exceptions

```
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}
// A Class that uses above MyException
public class Main
{
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("Hi");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

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

Output:

Caught
Hi

5.Stack Trace Elements

It is created to represent the specific execution point.

Syntax:

StackTraceElement(String ClassName, String methodName, String filename, int lineNumber);

```
//import java.lang.Thread;
//import java.lang.StackTraceElement; //The class in java.lang are normally available to be used
// without an explicit import. "A compilation unit automatically has access to all public types in
// java.lang"
```

```
public class stackDemo{

    public static void main(String[] args) {
        function1();
    }
}
```

```

public static void function1() {
    function2(); //if function2 is non-static use "new stackDemo().function2();"
}

public static void function2() {
    int i;
    System.out.println("method name : ");

    // print stack trace
    for( i = 1; i <= 3; i++ ) {
        System.out.println(Thread.currentThread().getStackTrace()[i].
            getMethodName());
    }
}
}

```

Output:

```

method name :
function2
function1
main

```

What is wrapper class?

A Wrapper class is a class which 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.

Wrapper classes are used to convert any data type into an object. The primitive data **types** are not objects; they do not belong to any class; they are defined in the language itself.

Eight wrapper classes exist in java.lang package that represent 8 data types.

All primitive **wrapper classes** (Integer, Byte, Long, Float, Double, Character, Boolean and Short) are **immutable**.

What is Autoboxing and unboxing in Java with example?

Autoboxing is the automatic conversion that the **Java** compiler makes between the primitive types and their corresponding object wrapper classes. For **example**, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called **unboxing**.

What is Type casting?

The process of forcing a conversion from one type to another is called casting.

Two types: 1). Widening casting (Implicit - JVM) Ex: int to double

2). Narrowing casting (Explicit - User) Ex: double to int

6.I/O

The java.io package deals with input and output. Java provides streams to deal with IO. Streams implement sequential access of data. Java IO concepts implemented based on UNIX OS.

Input to a program may come from keyboard, mouse, memory, disk, network. All inputs are done using methods and classes. These classes are found in java.IO package.

Streams

- Stream is a channel in which data flow from sender to receiver | Sequence of objects and methods pipelined together to produce results.
- An input object **reads** the stream of data from a file is called **input stream**.
- The output object **writes** the stream of data to a file is called **output stream**.

7.Byte streams and character streams

7.1.Byte streams(Binary stream)

- It is used for inputting and outputting the **bytes**.
- Java.util.InputStream** | **Two super classes** - InputStream and OutputStream
- Methods** – read() and write()

InputStream(classes)	FileInputStream	<i>BufferedInputStream</i> <i>DataInputStream</i> <i>LineNumberInputStream</i> <i>PushBackInputStream</i>
	PipedInputStream	
	FilterInputStream ➡	
	ByteArrayInputStream	
	SequenceInputStream	
	StringBufferInputStream	

OutputStream(classes)	FileOutputStream	<i>BufferedOutputStream</i> <i>DataOutputStream</i> <i>PrintStream</i>
	PipedOutputStream	
	FilterOutputStream ➡	
	ByteArrayOutputStream	
	SequenceOutputStream	
	StringBufferOutputStream	

InputStream Methods: available(), close(), mark(), read(), read(byte[]), reset(), skip()

OutputStream Methods: void close(), void flush(), void write(), abstract void write()

7.2 Character streams(Text stream)

- Character stream is used for inputting and outputting the **characters**.
- Two super classes – **Reader** and **Writer**
- Methods – read() and write()

Reader-(classes)	Writer-(classes)
FileReader	FileWriter
PipeReader	PipeWriter
FilterReader	FilterWriter
BufferedReader	BufferedWriter
DataReader	DataWriter
LineNumberReader	LineNumberWriter
PushBackReader	PushBackWriter
ByteArrayReader	ByteArrayWriter
SequenceReader	SequenceWriter
StringBufferReader	StringBufferWriter

FileInputStrem- Data is read as bytes from file. | **FilterInputStream-** Returns no.of bytes it can read | **DataInputStream-** filter allows binary representation of primitive values |

ObjectInputStream –allows binary rep.of objects & primitive values to read specific i/p stream.

Example: InputStream and Reader are similar but different data type

InputStream	Reader
Int read()	Int read()
Int read(byte buf[])	Int read(char buf[])
Int read(byte buf[], int offset, int length)	Int read(char buf[], int offset, int length)

Likewise, OutputStream and Writer are similar but different data type

OutputStream	Writer
Int write()	Int write()
Int write(byte buf[])	Int write(char buf[])
Int write(byte buf[], int offset, int length)	Int write(char buf[], int offset, int length)

Difference Between ByteStream and CharacterStream

S.No	ByteStream	CharacterStream
1.	It is used for inputting & outputting bytes.	It is used for characters.
2.	Two super classes – InputStream and OutputStream	Reader and Writer
3.	A byte is 8 bit represents -127 to 127. ASCII is 7 bit	A character is 16 bit represents Unicode
4.	Never support unicode	Supports Unicode

8. Reading and writing console

The scanner class in java.util allows user to read values.

Methods:

Methods	Returns
nextInt()	Integer value
nextDouble()	Decimal value
next()	String
nextLine()	Entire line as a string

Example 1:

```
import java.util.Scanner;
```

```
class ip
```

```
{
```

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

```
{
```

```
    Scanner in = new Scanner(System.in);
```

```
    System.out.println("what is ur name?"); //1st input
```

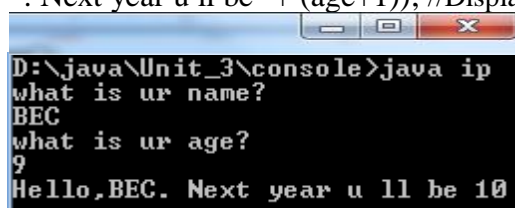
```
    String name = in.nextLine();
```

```
    System.out.println("what is ur age?"); //2nd input
```

```
    int age = in.nextInt();
```

```
    System.out.println("Hello," + name + ". Next year u ll be " + (age+1)); //Display output in console
```

```
}}
```



```
D:\java\Unit_3\console>java ip
what is ur name?
BEC
what is ur age?
9
Hello,BEC. Next year u ll be 10
```

The simplest way to read character is to call **System.in.read()**

Example 2:

```
import java.io.*;
class inputDemo
{
    public static void main(String a[]) throws IOException
    {
        char ch;
        System.out.print("Type a character followed by enter:");
        ch=(char)System.in.read();
        System.out.println("You entered:" +ch);
    }
}
```

Example 3:

```
Import java.io.*;
Class readchar
{
    Psvm(s a[]) throws IOException
    {
        BufferedReader obj = new BufferedReader(new InputStreamReader(System.in));
        int count=0;
        char ch;
        System.out.println("\n Enter 5 char");
        while(count<5)
        {
            ch=(char)obj.read();
            s.o.p(ch);
            count++;
        }
    }
}
```

Enter 5
char
Hello
H
e
l
l
o

Note: **InputStream** and **OutputStream** can read and write **raw bytes**. **DataInputStream** and **DataOutputStream** writes formatted binary data. (higher datatype)

9. Reading and writing files

The abstract classes **InputStream** and **OutputStream** are the root of the inheritance hierarchies for handling the reading and writing of byte streams.

Their subclasses override the methods of **InputStream** and **OutputStream**.

9.1 Declaration of **InputStream** and **OutputStream**

- public abstract class **InputStream** extends **Object** implements **Closeable**
- public abstract class **OutputStream** extends **Object** implements **Closeable**

Constructors

- public **InputStream()**
- public **OutputStream()**

Methods Refer above

Reading Binary values from a file

- 1) Create a FileInputStream,
FileInputStream fis = new FileInputStream(abc.txt);
- 2) Create DataInputStream which is chained to FIS,
DataInputStream dis = new DataInputStream(fis);
- 3) Read the primitive values in same order as written,
Ex: int i = inputStream.readInt();
- 4) Close filter stream,
dis.close();

Writing Binary values from a file

- 1) Create a FileOutputStream,
FileOutputStream fos = new FileOutputStream (abc.txt);
- 2) Create DataOutputStream which is chained to FIS,
DataOutputStream dos = new DataOutputStream (fos);
- 3) Write the primitive values using write() methods,
Ex: dos.writeBytes(variable);
- 4) Close filter stream,
dos.close();

Example: read and write operations on file using InputStream and OutputStream

Step1:

```
import java.io.*;
class output
{public static void main(String args[])
    {
        String s="This is my file";
        int a=5;
        Double d=5.35;
        try
        {
            FileOutputStream fos= new FileOutputStream("abcd.txt");
            DataOutputStream dos = new DataOutputStream(fos);
            dos.writeBytes(s);
            dos.writeInt(a);
            dos.writeDouble(d);
            dos.close();
        }
        catch(IOException ex)
        {ex.printStackTrace();}
    }
}
```

Step 2:

```
import java.io.*;
class input
{public static void main(String args[])
{
    try
    {
        FileInputStream fin= new FileInputStream("abc.txt"); //to read data from a file in bytes
        DataInputStream din = new DataInputStream(fin); // read primitive Java data types
        String line=null;
        while((line =din.readLine())!=null)
        {
            System.out.println(line);
        }
        din.close();
    }
    catch(Exception ex)
    {ex.printStackTrace();}
}
}
```

Compile: 1).javac Output.java

Run: 2).java Output //abc.txt file created n the text written on the file

Compile: 3). Javac Input.java

Run: 4). java input //the written text on the file will be displayed in console. To make sure check the file too.

Example2:**1. Using BufferedReader**

```
// dosDemo.java
import java.io.*;
public class dosDemo
{
    public dosDemo()
    {
        try
        {BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter filename:");
        String filename = br.readLine();

        System.out.println("Enter emp ID:");
        int id=Integer.parseInt(br.readLine());

        System.out.println("Enter emp name:");
        String name=br.readLine();

        System.out.println("Enter phone.no:");
```

```

        long phone=Long.parseLong(br.readLine());

        System.out.println("Enter salary:");
        double salary =Double.parseDouble(br.readLine());

        OutputStream os=new FileOutputStream(filename);
        DataOutputStream dos = new DataOutputStream(os);

        dos.writeInt(id);
        dos.writeUTF(name); // Unicode Transformation Format-encoding scheme allows to operate with ASCII and Unicode
        dos.writeLong(phone);
        dos.writeDouble(salary);
        dos.close();
    }
    catch(Exception e)
    { e.printStackTrace(); }
    }

    public static void main(String args[])
    {
        dosDemo dos = new dosDemo();
    }
}

//disDemo.java
import java.io.*;
public class disDemo
{
    public disDemo()
    {
        try
        {
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter filename:");
            String filename=br.readLine();

            DataInputStream dis = new DataInputStream(new FileInputStream(filename));

            int id=dis.readInt();
            String name = dis.readUTF();
            long phone = dis.readLong();
            double salary = dis.readDouble();
            dis.close();

            System.out.println("Emp ID:" +id);

```

```

        System.out.println("Emp name:" +name);
        System.out.println("Emp phone:" +phone);
        System.out.println("Emp salary:" +salary);
    }
    catch(Exception e)
    { e.printStackTrace(); }
    }
    public static void main(String[] args)
    {
        disDemo dis = new disDemo();
    }
    }

```

Compile: javac dosDemo.java

Run: java dosDemo

Compile: javac disDemo.java

Run: java disDemo

9.2 File class

The **java.io.File** class is an abstract representation of file and directory pathnames.

- Instances may or may not denote actual file system.
- File system may implement restrictions
- Instances of file class are immutable(i.e once created the file object will never change).

Class Declaration

public class File extends Object implements Serializable, Comparable <File> Field

Class constructors

File(File parent, String child) | File(String parent, String child)

File(String pathname) | File(URI uri) //Uniform Resource Identifier(**URI**)-identify resource(network location)

Class Methods

boolean CreateNewFile()		boolean delete()
boolean canRead()		boolean canWrite()

These methods are inherited from java.io.Object class.

Example:

Import java.io.*;

Class fileDemo

```

{ psvm(s[] a)throws Exception
    { File f = new File("abc123.txt"); //file object created
      s.o.p(f.exists());
      f.createNewFile(); //file created
      s.o.p(f.exists()); } }

```

Output: false

true

Example: to create a folder

```
import java.io.*;
class fileDemo
{
    public static void main(String[] args)throws Exception
    {
        File f = new File("remo"); //file object created
        f.mkdir();
        System.out.println(f.exists());
    }
}
```

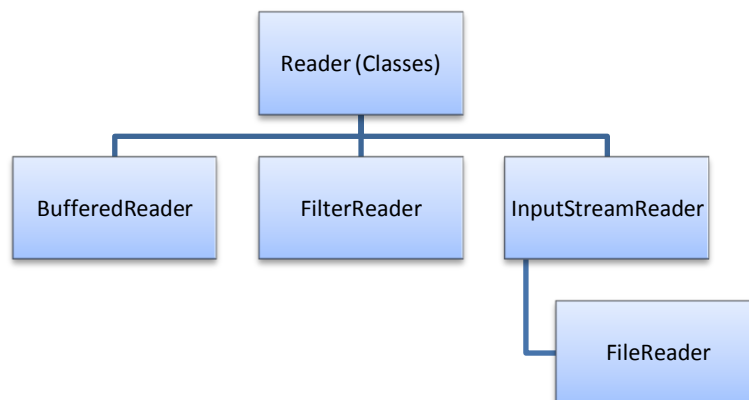
9.2.1 Reader class & Writer class

A character encoding is a scheme for representing characters. Java represents char type in 16 bit Unicode character encoding.

1) Readers

A reader is an input character stream that reads a sequence of Unicode characters and writer is an output character stream that writes a sequence of Unicode characters.

- **BufferedReader** – A reader that buffers the characters to read
- **InputStreamReader** – characters read from byte input stream
- **FileReader** – Reads characters from a file using encoding.



BufferedReader – buffers characters from reader | User can specify buffer size

FilterReader- It is an abstract class for reading filtered character streams. It provides default methods

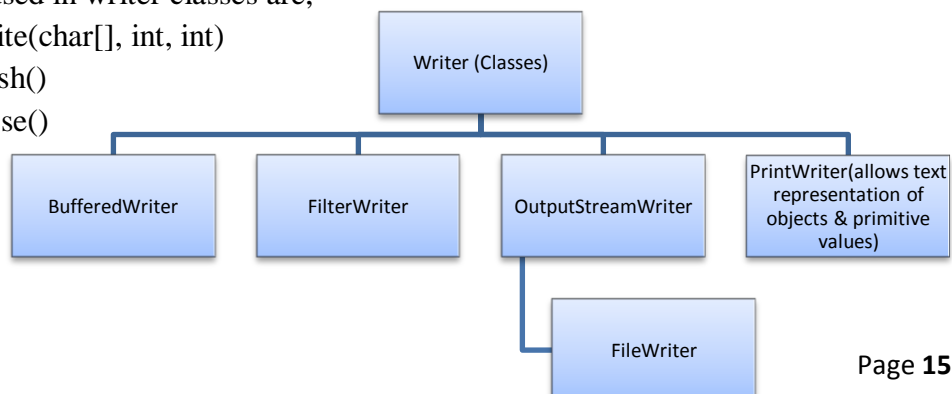
InputStreamReader – characters are read from byte input stream

FileReader – Reads characters from file

2) Writers

Methods used in writer classes are,

- **write(char[], int, int)**
- **flush()**
- **close()**



Example program of Reader and Writer

//writerDemo

```
import java.io.*;
class writerDemo
{
    public static void main(String[] args)
    {
        try
        {
            FileWriter fw = new FileWriter("abc.txt");
            fw.write("Hello, Good Morning"); // fw.write("123")
            fw.close();
        }
        catch(IOException ex)
        {ex.printStackTrace();}
    }
}
```

//readerDemo

```
import java.io.*;
class writerDemo
{
    public static void main(String[] args)
    {
        try
        {
            File f1= new File("abc.txt");
            FileReader fr = new FileReader(f1);
```

```
BufferedReader br=new BufferedReader(fr); //chaining
```

```
String line=null;
while((line=br.readLine()) !=null)
{
    System.out.println(line);
}
br.close();
}
catch(Exception ex)
{ex.printStackTrace();}
}}
```

//Writer

```
import java.io.*;
class writerDemo
{
    public static void main(String[] args)
    {
        FileWriter fw = new FileWriter("abc.txt"); //,true
        fw.write(10);
        fw.write("Hello \n Good morning");
        fw.write("\n");
        char[] ch= {'a','b','c'};
        fw.write(ch);
        fw.flush();
        fw.close();
    }
}
```

True- wont override – data will be repeatedly written when executed more than once.

Part-A

1. What is an exception? Give example
2. **What will happen if an exception is not caught?**
3. Give any methods available in Stack Trace Element.
4. **What is Stack Trace Element? Syntax.**
5. What is the benefit of exception handling?
6. What is difference between error and exception in java?
7. What are all the types of exceptions?
8. What is the use of try, catch, finally keywords?
9. **What is the difference between throw and throws?**
10. What is ArrayIndexOutOfBoundsException?
11. What is the need of multiple catch?
12. **Can we have empty catch block?**
13. What is the super class for all types of errors and exceptions in java?
14. What is input stream and output stream?
15. Difference between byte stream and character stream.
16. What is the purpose of BufferedInputStream and BufferedOutputStream?
17. What is the use of seek() method?
18. What is the use of input Stream Reader and Output Stream Writer?
19. List constructors of a File class.
20. What is file reader and file writer? Why we use BufferedReader & writer?

Part-B

1. **Explain exception hierarchy with flow diagram. 16m**
2. Write the syntax for multiple catch and multiple try blocks.
3. Write a program using try, catch and finally block.
4. **Explain throw and throws differences with example.**
5. List and explain 8 built in exceptions.
6. Explain stack trace elements with example.
7. What is a stream? Explain its types and list its classes.
8. Explain reading and writing in console? Example.
9. **Explain reading and writing files with example.**
10. **What is file class? List its constructors. Example for FileReader and Writer.**