Unit – III

Exception Handling and I/O

Exceptions-exception hierarchy-throwing and catching exceptions-built-in exceptions, creating own exceptions, Stack Trace Elements. Input /Output Basics-Streams-Byte streams and character streams-Reading and Writing Console-Reading and Writing Files Templates

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applicationsshould not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditionsand take necessary corrective actions. Few examples

- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

- An exception (or exceptional event) is a problem that arises during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

Ex: Exception in thread "main"

java.lang.ArithmeticException: / by zero at

ExceptionDemo.main(ExceptionDemo.java:5)

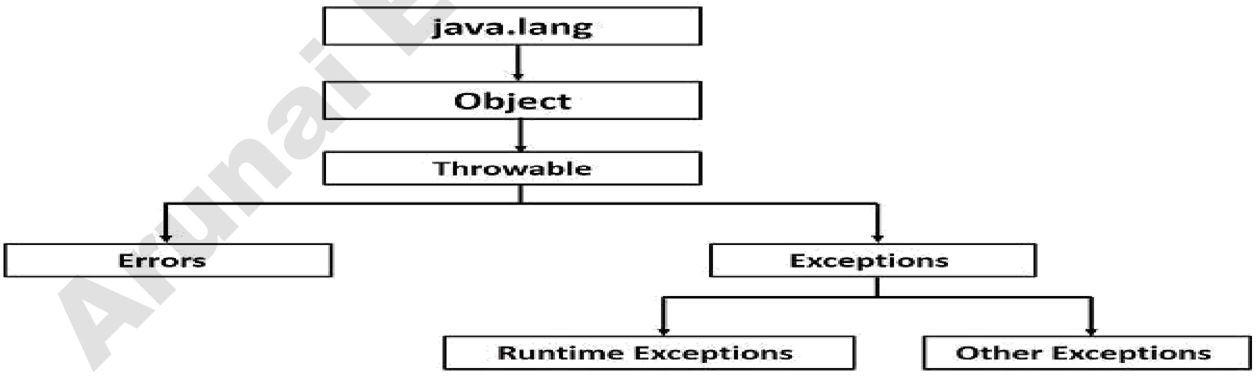
Where,	ExceptionDemo : The class name
	main : The method name
	ExceptionDemo.java : The filename
	java:5 : Line number

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an **invalid data.**
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM hasrun out of memory.

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class.



Key words used in Exception handling

There are 5 keywords used in java exception handling.

1. try	A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.
2. catch	A catch statement involves declaring the type of exception we are trying to catch.
3. finally	A finally block of code always executes, irrespective of occurrence of an Exception.
4. throw	It is used to execute important code such as closing connection, stream etc. throw is used to invoke an exception explicitly.
5. throws	throws is used to postpone the handling of a checked exception.

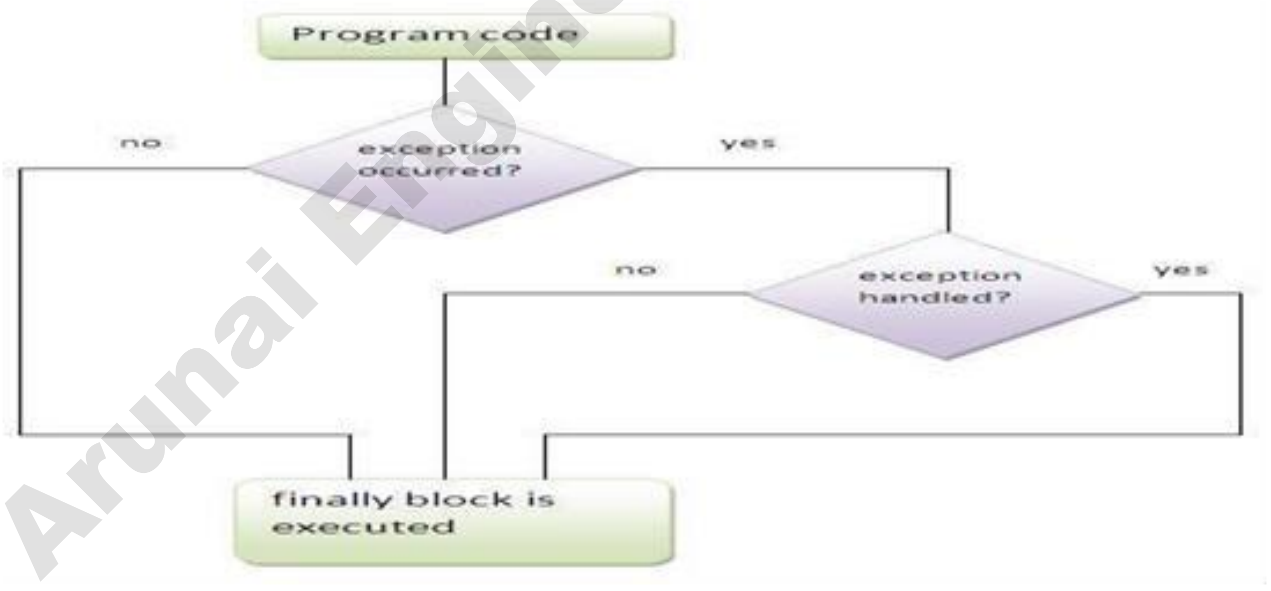
Syntax :	

```
try
{
   //Protected code
}

catch(ExceptionType1 e1)

{
   //Catch block
}
catch(ExceptionType2 e2)
{
   //Catch block
}
```
	```
	//Example-predefined Excetion - for
	//ArrayindexoutofBounds Exception
	public class ExcepTest
	{
	 public static void main(String args[])
	  { int a[] = new int[2];
	     try
	     { System.out.println("Access element three :" + a[3]);
	       }
	      catch(ArrayIndexOutOfBoundsException e)
	      { System.out.println("Exception thrown :" + e);
	       }
	     finally
	      { a[0] = 6;
	```

catch(ExceptionType3 e3)
 	
{
//Catch block
}
finally
{
 	
//The finally block always executes.
}
	System.out.println("First element value: " + a[0]);
	System.out.println("The finally statement is executed");
	}
	}
	}
	Output
	Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
	First element value: 6
	The finally statement is executed
	***Note :** here array size is 2 but we are trying to access 3rdelement.*



Uncaught Exceptions

This small program includes an expression that intentionally causes a divide-by-zero error: class Exc0 { public static void main(String args[]) { int d = 0; int a =

42 / d; } } When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately

Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:
java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

Stack Trace:
Stack Trace is a list of method calls from the point when the application was started to the point where the exception was thrown. The most recent method calls are at the top. A stacktrace is a very helpful debugging tool. It is a list of the method calls that the application was in the middle of when an Exception was thrown. This is very useful because it doesn't only show you where the error happened, but also how the program ended up in that place of the code.

Using try and Catch
To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. A try and its catch statement form a unit. The the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

The call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. 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.

The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
```

```
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

throw

it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Primitive types, such as int or char, as well as non-Throwable classes, such as String and

Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

Throws

If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
```

```
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional.

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
```

```
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
```

}

Here is the output generated by the preceding program:

```
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally
```

Categories of Exceptions

Checked exceptions −A checked exception is an exception that occurs at the compiletime, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

Unchecked exceptions − An unchecked exception is an exception that occurs at thetime of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

Common scenarios where exceptions may occur:

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java’s Built-in Exceptions

User-defined Exceptions

All exceptions must be a child of **Throwable.**

If we want to write a checked exception that is automatically enforced by the Handle ,we need to extend the Exception class.

User defined exception needs to inherit (extends) Exception class in order to act as an exception.

throw keyword is used to throw such exceptions.

```
class MyOwnException extends Exception
{                         public
   MyOwnException(String
   msg) { super(msg);
    }
}
class EmployeeTest
{
    static void employeeAge(int age) throws
     MyOwnException {
        if(age < 0)
          throw new MyOwnException("Age can't be less
      than zero"); else
          System.out.println("Input is valid!!");
     }
public static void main(String[] args)
{
        try { employeeAge(-2);
            }
        catch (MyOwnException e)
        {
            e.printStackTrace();
          }
 }
}
```

Advantages of Exception Handling

Exception handling allows us to control the normal flow of the program by using exception handling in program.

It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

IO IN JAVA

Java I/O (Input and Output) is used to process the input and produce the output based on the input. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Stream

A stream can be defined as a sequence of data. there are two kinds of Streams

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

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes FileInputStream , FileOutputStream.

Character Streams

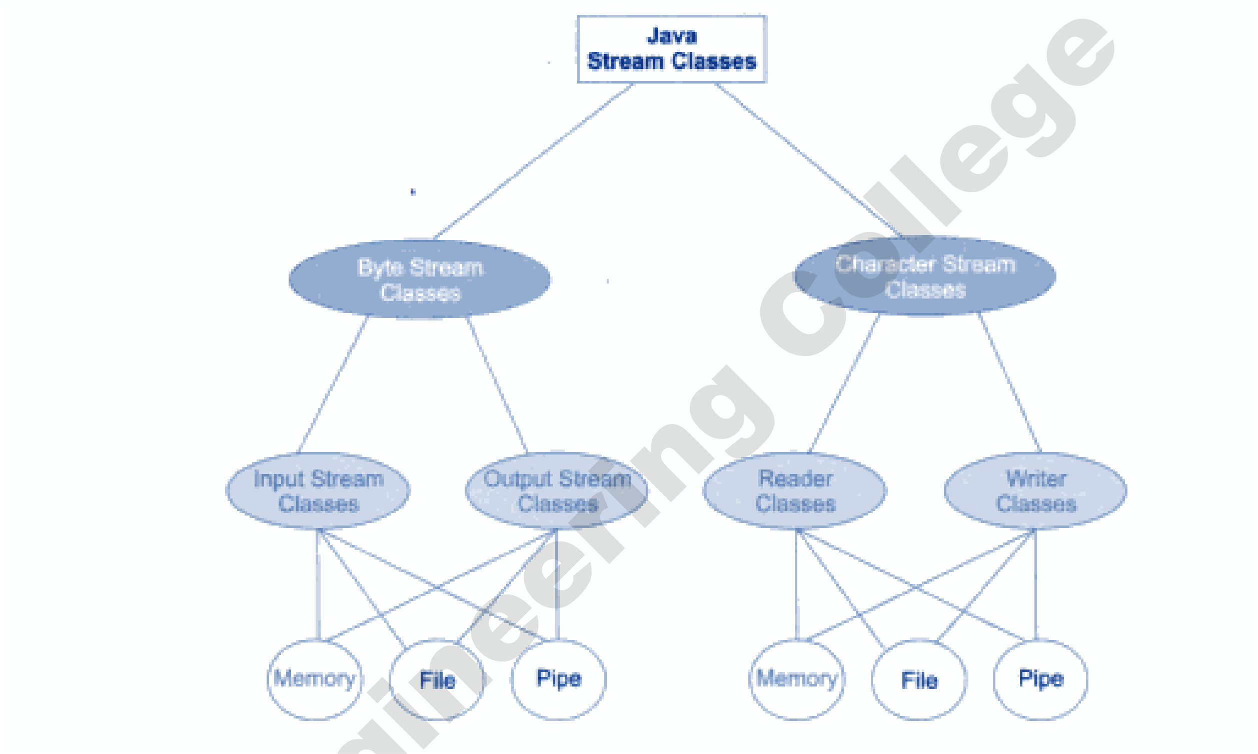
Java Character streams are used to perform input and output for 16-bit unicode. FileReader , FileWriter

Standard Streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as System.out.

- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as System.err.

Classification of Stream Classes:

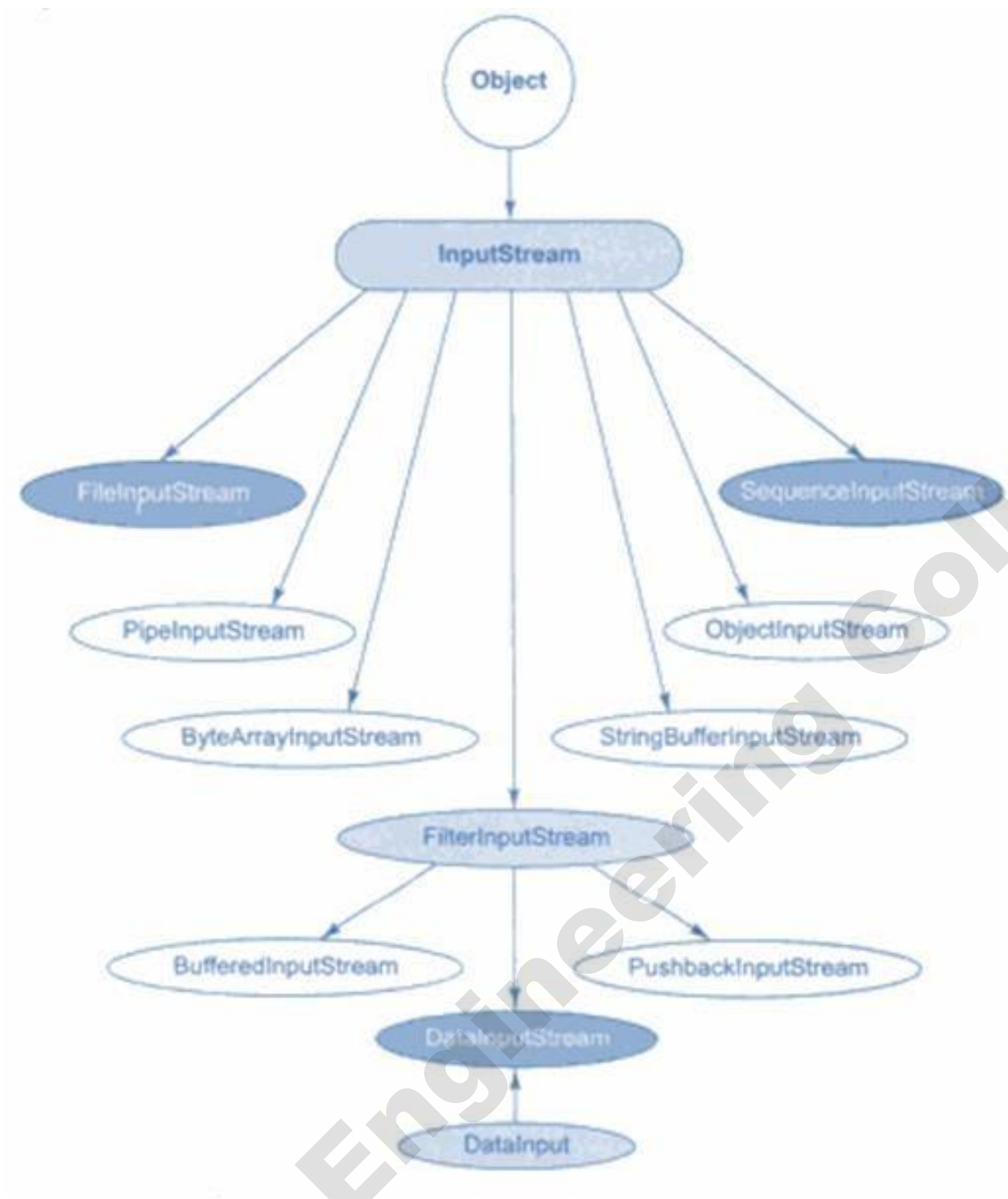


Byte Stream Classes:

ByteStream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes in only one direction and therefore, Java provides two kinds of byte stream classes: InputStream class and OutputStream class.

Input Stream Classes

Input stream classes that are used to read 8-bit bytes include a super class known as InputStream and number of subclasses for supporting various input-related functions.



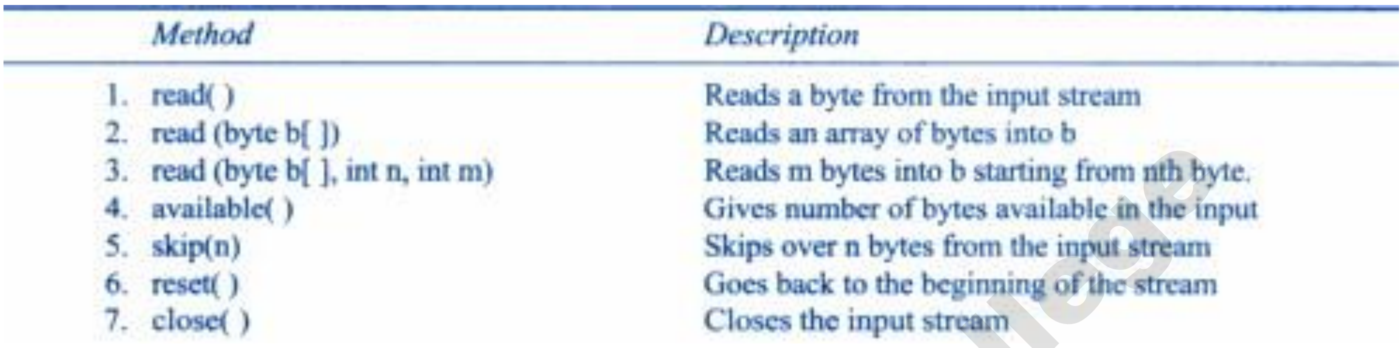
Hierarchy of Input Stream Classes

The super class InputStream is an abstract class, so we cannot create object for the class. InputStream class defines the methods to perform the following functions:-

- Reading Bytes
- Closing Streams
- Marking position in Streams
- Skipping ahead in streams

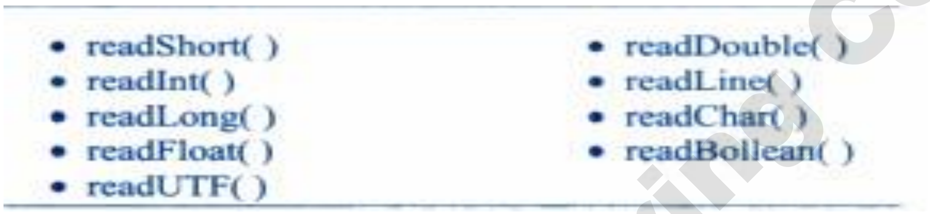
- Finding the number of bytes in stream.

The following are the InputStream methods:



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

The DataInput interface contains the following methods



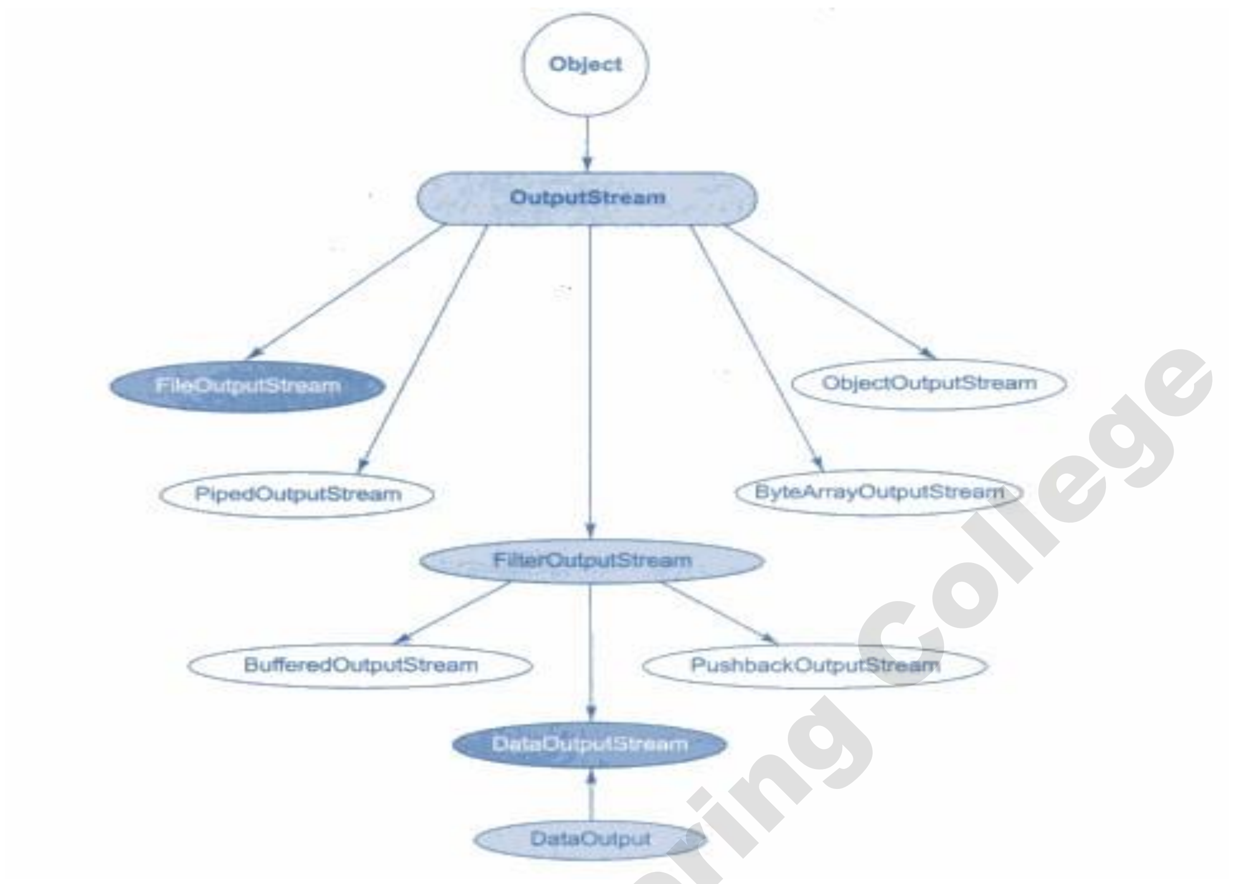
- readShort( )
- readInt( )
- readLong( )
- readFloat( )
- readUTF( )
- readDouble( )
- readLine( )
- readChar( )
- readBollean( )

OutputStream Class

The super class InputStream is an abstract class, so we cannot create object for the class. InputStream class defines the methods to perform the following functions:

- Writing Bytes
- Closing Streams
- Flushing Streams

Hierarchy of OutputStream Classes



OutputStream Methods

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

Character Stream Vs Byte Stream in Java

I/O Stream

A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files.The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.

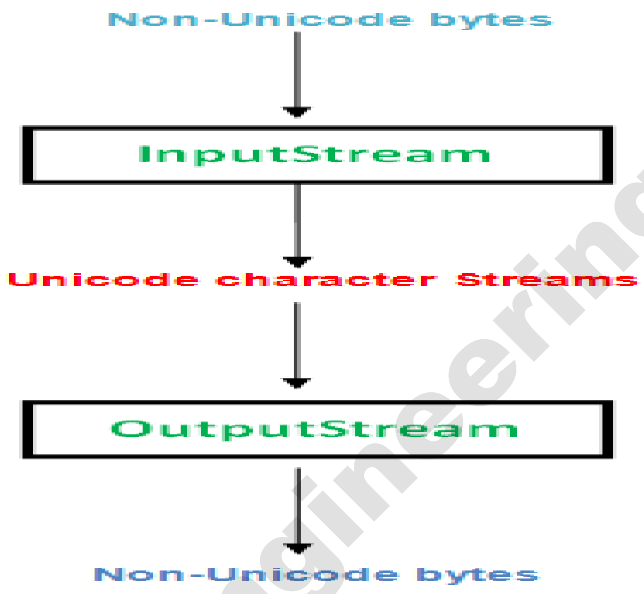
Stream: A sequence of data.

Input Stream: reads data from source.

Output Stream: writes data to destination.

Character Stream

In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source andwrite to destination.



```
// Java Program illustrating that we can read a file in
// a human readable format using FileReader
importjava.io.*;   // Accessing FileReader, FileWriter, IOException
publicclassGfG
{
  publicstaticvoidmain(String[] args) throwsIOException
  {
    FileReader sourceStream = null;
    try
    {
      sourceStream = newFileReader("test.txt");
```

```
             // Reading sourcefile and writing content to
            // target file character by character.
            inttemp;
            while((temp = sourceStream.read()) != -1)
                System.out.println((char)temp);
        }
        finally
        {
            // Closing stream as no longer in use
            if(sourceStream != null)
                sourceStream.close();
        }
    }
}
```

Reading and Writing Files:

A stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

TheInputStream is used to read data from a source and the OutputStream is used for writing data to a destination.The two important streams are **FileInputStream** and **FileOutputStream**

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

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file –

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Example:

```
import
java.io.*;
class C{
public    static    void    main(String    args[])throws
Exception{            FileInputStream            fin=new
FileInputStream("C.java");            FileOutputStream
fout=new FileOutputStream("M.java"); int i=0;
while((i=fin.read())!=-
1){ fout.write((byte)i);
}
fin.close();
}
}
```

Byte Stream

Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

```
// Java Program illustrating the Byte Stream to copy
// contents of one file to another file.
```

}

```
importjava.io.*;
publicclassBStream
{
    Public static void main(String[] args) throws IOException
    {
        FileInputStream sourceStream = null;
        FileOutputStream targetStream = null;
        try
        {
            sourceStream = newFileInputStream("sorcefile.txt");
            targetStream = newFileOutputStream ("targetfile.txt");

            // Reading source file and writing content to target
            // file byte by byte
            inttemp;
            while((temp = sourceStream.read()) != -1)
                targetStream.write((byte)temp);
        }
        finally
        {
            if(sourceStream != null)
                sourceStream.close();
            if(targetStream != null)
                targetStream.close();
        }
    }
}
```

Final Keyword In Java – Final variable, Method and Class

final keyword can be used along with variables, methods and classes.

1) final variable

2) final method

3) final class

1) final variable

final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Lets have a look at the below code:

```
classDemo{
finalint MAX_VALUE=99;
void myMethod(){
     MAX_VALUE=101;
}
Public static void main(String args[]){
Demo obj=newDemo();
     obj.myMethod();
}
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
          The final field Demo.MAX_VALUE cannot be assigned
          at beginnersbook.com.Demo.myMethod(Details.java:6)
          at beginnersbook.com.Demo.main(Details.java:10)
```

We got a compilation error in the above program because we tried to change the value of a final variable “MAX_VALUE”.

2) final method

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

Example:

```
class XYZ{
```

```
finalvoid demo(){
System.out.println("XYZ Class Method");
}
}
class ABC extends XYZ{
void demo(){
System.out.println("ABC Class Method");
}
public static void main(String args[]){
    ABC obj=new ABC();
    obj.demo();
}
}
```

The above program would throw a compilation error, however we can use the parent class final method in sub class without any issues. Lets have a look at this code: This program would run fine as we are not overriding the final method. That shows that final methods are inherited but they are not eligible for overriding.

```
class XYZ{
finalvoid demo(){
System.out.println("XYZ Class Method");
}
}
class ABC extends XYZ{
public static void main(String args[]){
    ABC obj=new ABC();
    obj.demo();
}
}
```

Output:

```
XYZ ClassMethod
```

3) final class

We cannot extend a final class. Consider the below example:

```
finalclass XYZ{
}
class ABC extends XYZ{
void demo(){
System.out.println("My Method");
}
Public static void main(String args[]){
    ABC obj=new ABC();
    obj.demo();
}
}
```

Output:

```
The type ABC cannot subclass the final class XYZ
```