

### **Unit - 3**

#### **Exception Handling**

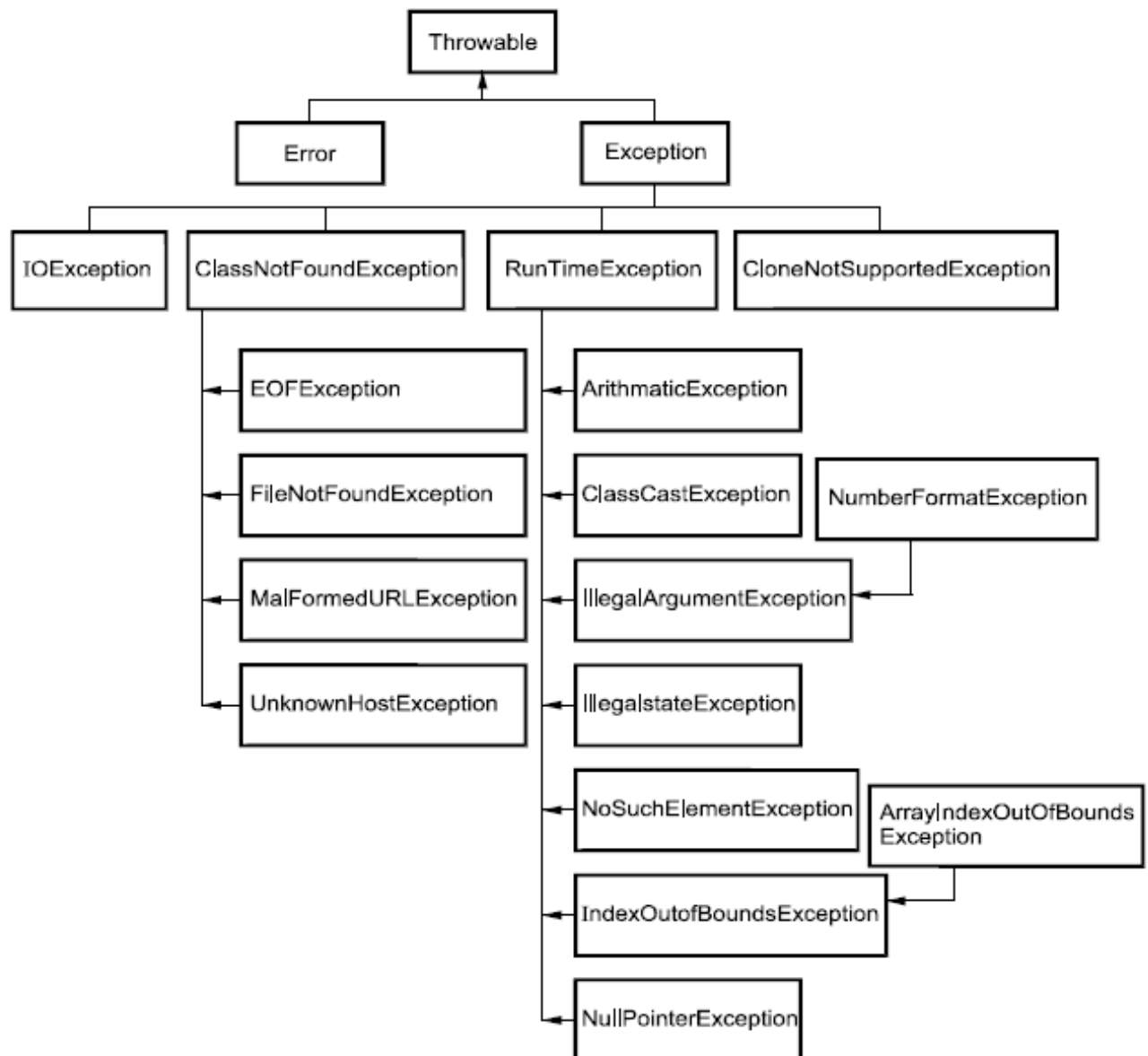
1. The Java programming language uses exceptions to handle errors and other exceptional events.
2. An exception is an abnormal condition that arises in a code sequence at run time.
3. In other words, an exception is a run-time error.
4. In Java, an exception is an object.
5. Exceptions can be generated by - the Java run-time system, or - they can be manually generated by our code.
6. Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.

#### **Exception Hierarchy**

The exception hierarchy is derived from the base class Throwable. The Throwable class is further divided into two classes - Exceptions and Errors.

**Exceptions :** Exceptions are thrown if any kind of unusual condition occurs that can be caught. Sometimes it also happens that the exception could not be caught and the program may get terminated.

**Errors :** When any kind of serious problem occurs which could not be handled easily like Out Of Memory Error then an error is thrown.



## Types of Exception

- There are two type of exceptions in Java

**Checked Exception :** These types of exceptions need to be handled explicitly by the code itself either by using the try-catch block or by using throws. These exceptions are extended from the java.lang.Exception class.

**For example :** IOException which should be handled using the try-catch block.

**Unchecked Exception :** These type of exceptions need not be handled explicitly. The Java Virtual Machine handles these type of exceptions. These exceptions are extended from **java.lang.RuntimeException** class.

**For example :** `ArrayIndexOutOfBoundsException`, `NullPointerException`, `RunTimeException`.

### Throwing and Catching Exceptions

Various keywords used in handling the exception are -

**try** - A block of source code that is to be monitored for the exception.

**catch** - The catch block handles the specific type of exception along with the try block. Note that for each corresponding try block there exists the catch block.

**finally** - It specifies the code that must be executed even though exception may or may not occur.

**throw** - This keyword is used to throw specific exception from the program code.

**throws** - It specifies the exceptions that can be thrown by a particular method.

### try-catch Block

The statements that are likely to cause an exception are enclosed within a try block. For these statements the exception is thrown.

- There is another block defined by the keyword catch which is responsible for handling the exception thrown by the try block.
- As soon as exception occurs it is handled by the catch block.
- The catch block is added immediately after the try block.

Following is an example of **try-catch** block

```
try
{
//exception gets generated here
}
catch(Type_of_Exception e)
{
//exception is handled here
}
```

- If any one statement in the try block generates exception then remaining statements are skipped and the control is then transferred to the catch statement.

### Java Program[RunErrDemo.java]

```
class RunErrDemo
{
    public static void main(String[] args)
    {
        int a,b,c;
        a=10;
        b=0;
        try
        {
            c=a/b;
        }
        catch(ArithmeticException e)
        {
            System.out.println("\n Divide by zero");
        }
        System.out.println("\n The value of a: "+a);
        System.out.println("\n The value of b: "+b);
    }
}
```

Exception occurs because the element is divided by 0.

Exception is handled using **catch** block

### Output

Divide by zero

The value of a: 10

The value of b: 0

Note that even if the exception occurs at some point, the program does not stop at that point.

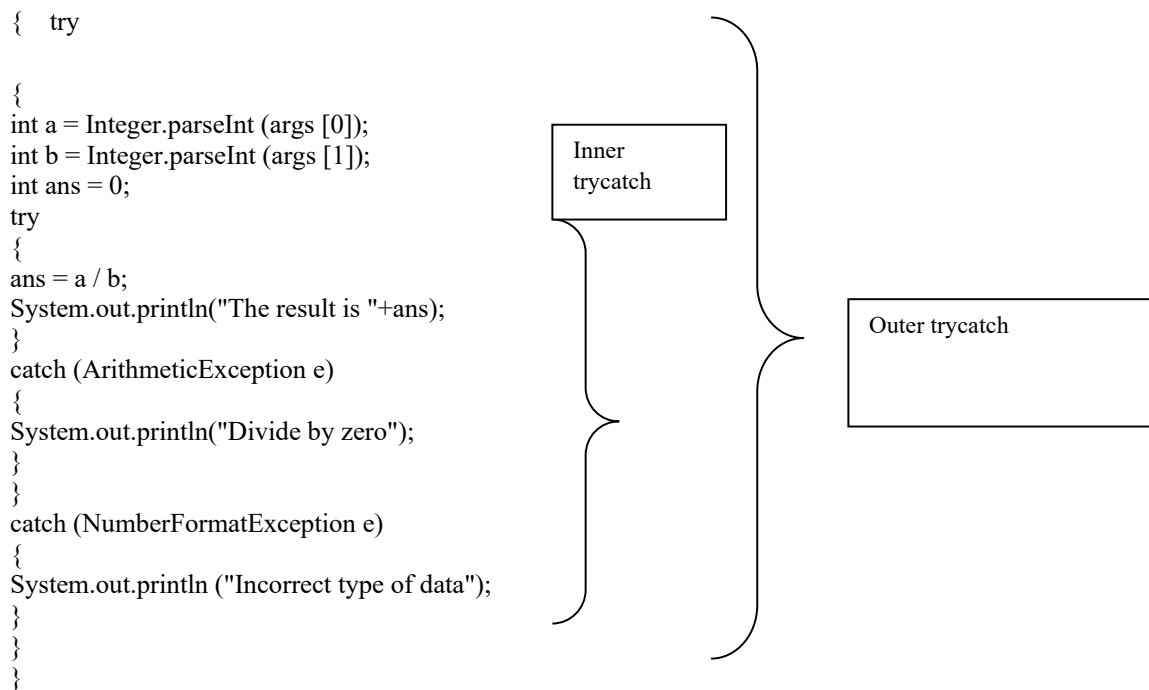
### Nested try Statements

When there are chances of occurring multiple exceptions of different types by the same set of statements then such situation can be handled using the nested try statements.

Following is an example of nested try-catch statements –

### Java Program[NestedtryDemo.java]

```
class NestedtryDemo
{
    public static void main (String[] args)
```



### Output

D:\>javac NestedtryDemo.java

D:\>java NestedtryDemo 20 10

The result is 2

D:\>java NestedtryDemo 20 a

Incorrect type of data

D:\>java NestedtryDemo 20 0

Divide by zero

Outer catch handles the error

Inner catch handles the error

### Multiple Catch

- It is not possible for the try block to throw a single exception always.
- There may be the situations in which different exceptions may get raised by a single try block statements and depending upon the type of exception thrown it must be caught.

- To handle such situation multiple catch blocks may exist for the single try block statements.

The syntax for single try and multiple catch is –

```
try
{
...
...//exception occurs
}
catch(Exception_type e)
{
...//exception is handled here
}
catch(Exception_type e)
{
...//exception is handled here
}
catch(Exception_type e)
{
...//exception is handled here
}
```

### Using finally

- Sometimes because of execution of try block the execution gets break off. And due to this some important code (which comes after throwing off an exception) may not get executed.
- That means, sometimes try block may bring some unwanted things to happen.
- The finally block provides the assurance of execution of some important code that must be executed after the try block.
- Even though there is any exception in the try block the statements assured by finally block are sure to execute. These statements are sometimes called as clean up code.

The syntax of finally block is

```
finally
{
//clean up code that has to be executed finally
}
```

The finally block always executes. The finally block is to free the resources.

#### **Java Program [finallyDemo.java]**

```
/*
```

This is a java program which shows the use of finally block for handling exception

```
*/
```

```
class finallyDemo
```

```
{
public static void main(String args[])
{
int a=10,b=-1;
try
{
b=a/0;
}
catch(ArithmeticException e)
{
System.out.println("In catch block: "+e);
}
finally
{
if(b!=-1)
System.out.println("Finally block executes without occurrence of exception");
else
System.out.println("Finally block executes on occurrence of exception");
}
}
}
```

#### **Output**

In catch block: java.lang.ArithmeticException: / by zero

Finally block executes on occurrence of exception

### Using throws

When a method wants to throw an exception then keyword throws is used.

The syntax is - method\_name(parameter\_list) throws exception\_list

```
{
}
```

Let us understand this exception handling mechanism with the help of simple Java program.

### Java Program

/\* This programs shows the exception handling mechanism using throws

\*/

```
class ExceptionThrows
```

```
{
```

```
static void fun(int a,int b) throws ArithmeticException
```

```
{
```

```
int c;
```

```
try
```

```
{
```

```
c=a/b;
```

```
}
```

```
catch(ArithmeticException e)
```

```
{
```

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

```
}
```

```
}
```

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

```
{
```

```
int a=5;
```

```
fun(a,0);
```



```
}
}
```

### Output

Caught exception: java.lang.ArithmeticException: / by zero

- In above program the method *fun* is for handling the exception *divide by zero*. This is an arithmetic exception hence we write

```
static void fun(int a,int b) throws ArithmeticException
```

- This method should be of *static* type. Also note as this method is responsible for handling the exception the **try-catch** block should be within *fun*.

### Using throw

- For explicitly throwing the exception, the keyword throw is used. The keyword throw is normally used within a method. We can not throw multiple exceptions using throw.

### Java Programming Example

```
class ExceptionThrow
{
static void fun(int a,int b)
{
int c;
if(b==0)
throw new ArithmeticException("Divide By Zero!!!");
else
c=a/b;
}
public static void main(String args[])
{
int a=5;
fun(a,0);
}
```

}

**Output**

Exception in thread "main" java.lang.ArithmeticException: Divide By Zero!!!  
 at ExceptionThrow.fun(ExceptionThrow.java:7)  
 at ExceptionThrow.main(ExceptionThrow.java:14)

**Difference between throw and throws**

Throw	Throws
For explicitly throwing the exception, the keyword throw is used.	For declaring the exception the keyword throws is used.
Throw is followed by instance.	Throws is followed by exception class.
Throw is used within the method.	Throw is used with method signature.
We cannot throw multiple exceptions.	It is possible to declare multiple exceptions using throws.

**Built in Exceptions**

➤ Various common exception types and causes are enlisted in the following table –

Exception	Description
ArithmeticException	This is caused by error in Math operations. For e.g. Divide by zero.
NullPointerException	Caused when an attempt to access an object with a Null reference is made.
IOException	When an illegal input/output operation is performed then this exception is raised.
IndexOutOfBoundsException	An index when gets out of bound ,this exception will be caused.
ArrayIndexOutOfBoundsException	Array index when gets out of bound, this exception will be caused.
ArrayStoreException	When a wrong object is stored in an array this exception must occur.
EmptyStackException	An attempt to pop the element from empty stack is made then this exception occurs
NumberFormatException	When we try to convert an invalid string to number this exception gets caused.
RuntimeException	To show general run time error this exception must be raised.
Exception	This is the most general type of exception
ClassCastException	This type of exception indicates that one tries to cast an object to a type to which the object can not be casted.

IllegalStateException	This exception shows that a method has been invoked at an illegal or inappropriate time. That means when Java environment or Java application is not in an appropriate state then the method is invoked.
-----------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Creating Own Exceptions

We can throw our own exceptions using the keyword **throw**.

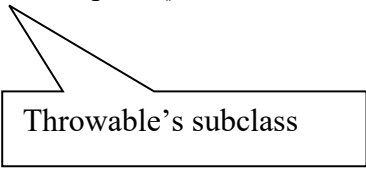
The syntax for throwing our own exception is –

**throw new** Throwable's subclass

Here the Throwable's subclass is actually a subclass derived from the **Exception** class.

**For example -**

`throw new ArithmeticException();`



Throwable's subclass

Let us see a simple Java program which illustrates this concept.

### Java Program[MyExceptDemo.java]

```
import java.lang.Exception;
class MyOwnException extends Exception
{
    MyOwnException(String msg)
    {
        super(msg);
    }
}
class MyExceptDemo
{
    public static void main (String args [])
    {
        int age;
        age=15;
        try
        {
            if(age<21)
                throw new MyOwnException("Your age is very less than the condition");
        }
        catch (MyOwnException e)
        {
            System.out.println ("This is My Exception block");
            System.out.println (e.getMessage());
        }
    }
}
```

```

}
finally
{
System.out.println ("Finally block:End of the program");
}
}
}
}

```

### Output

This is My Exception block  
Your age is very less than the condition  
Finally block:End of the program

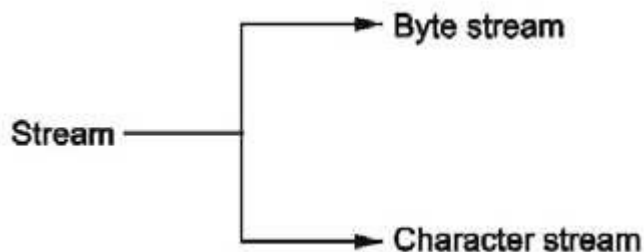
## Streams

### Definition :

- Stream is basically a channel on which the data flow from sender to receiver.
- An input object that reads the stream of data from a file is called **input stream**.
- The output object that writes the stream of data to a file is called **output stream**.

### Byte Streams and Character Streams

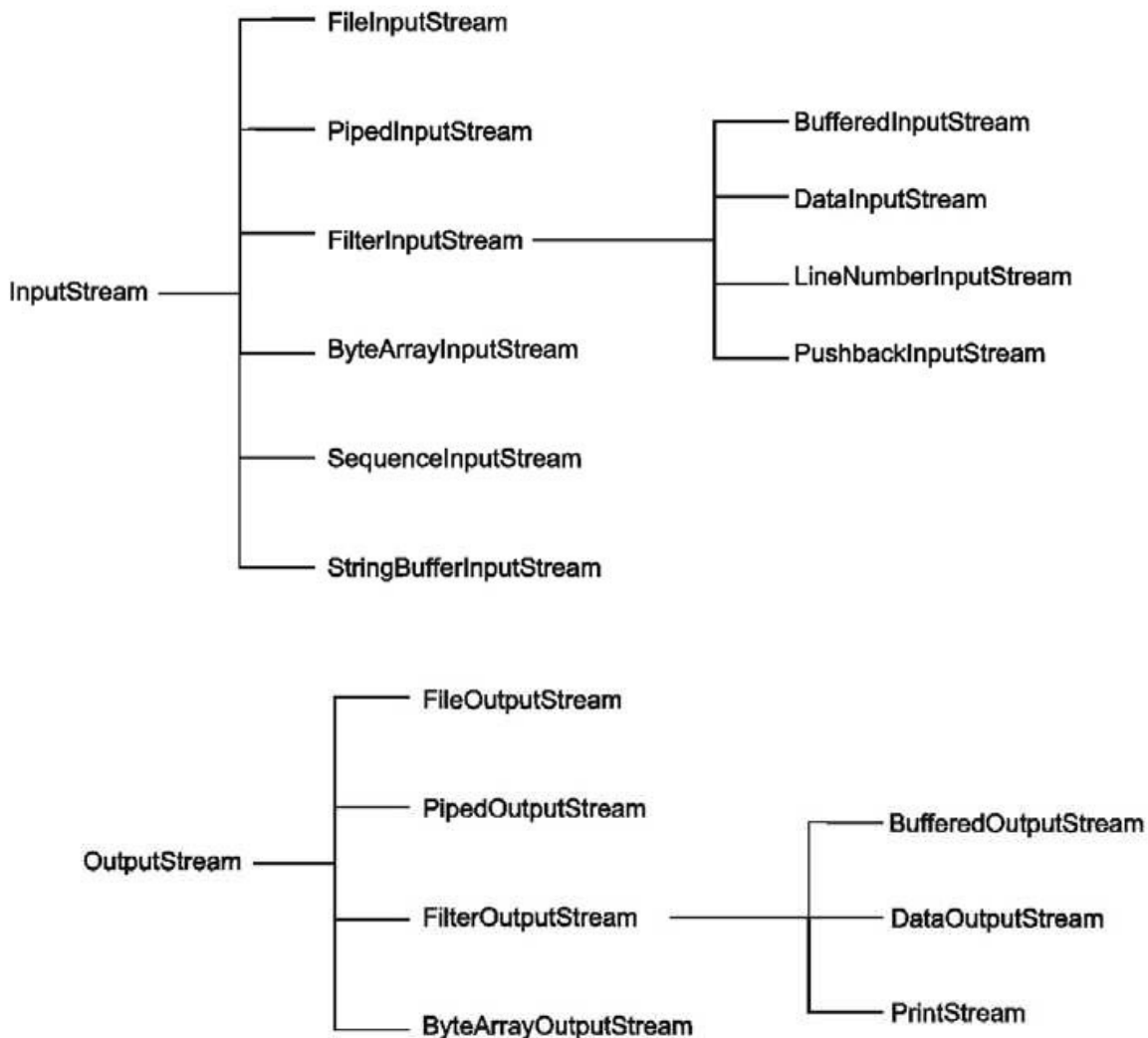
- In Java input and output operations are performed using streams. And Java implements streams within the classes that are defined in **java.io**.
- There are basically two types of streams used in Java.



### Byte Stream

- The byte stream is used for inputting or outputting the bytes.
- There are two super classes in byte stream and those are **InputStream** and **OutputStream** from which most of the other classes are derived.
- These classes define several important methods such as **read()** and **write()**.

- The input and output stream classes are

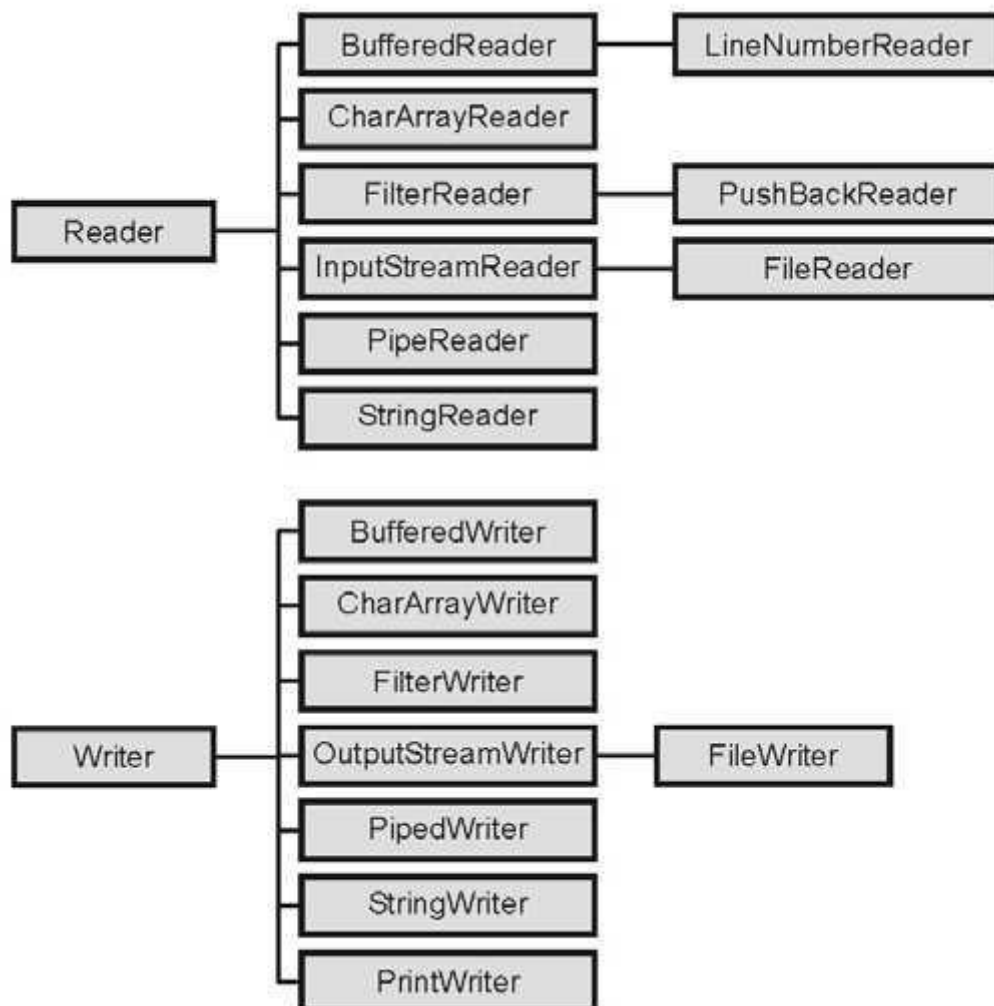


### Character Stream

- The character stream is used for inputting or outputting the characters.
- There are two super classes in character stream and those are **Reader** and **Writer**.
- These classes handle the Unicode character streams.
- The **two important methods** defined by the character stream classes are **read()** and **Write()** for reading and writing the characters of data.
- Various Reader and Writer classes are –

FileReader	FileWriter
FilterReader	FilterWriter
BufferedReader	BufferedWriter

DataReader	DataWriter
LineNumberReader	LineNumberWriter
ByteArrayReader	ByteArrayWriter



Reader and **InputStream** define similar APIs but for different data types. For example, Reader contains these methods for reading characters and arrays of characters :

```
int read()
int read(char buf[])
```

```
int read(char buf[], int offset, int length)
```

**InputStream** defines the same methods but for reading bytes and arrays of bytes :

```
int read()
int read(byte buf[])
int read(byte buf[], int offset, int length)
```

**Writer** and **OutputStream** are similarly parallel. **Writer** defines these methods for writing characters and arrays of characters :

```
int write(int c)
int write(char buf[])
int write(char buf[], int offset, int length)
```

and **OutputStream** defines the same methods but for bytes :

```
int write(int c)
int write(byte buf[])
int write(byte buf[], int offset, int length)
```

### Comparison between Byte Stream and Character Stream

Byte Stream	Character Stream
The byte stream is used for inputting and outputting the bytes.	The character stream is used for inputting and outputting the characters.
There are two super classes used in byte stream and those are – <b>InputStream</b> and <b>OutputStream</b>	There are two super classes used in byte stream and those are – <b>Reader</b> and <b>Writer</b>
A byte is a 8-bit number type that can represent values from – 127 to 127. Ascii values can be represented with exactly 7 bits.	A character is a 16 bit number type that represents Unicode.
It never supports Unicode characters	It supports Unicode characters.

### Reading and Writing Console

#### Reading Console Input

- In this section, we will understand how to *read the input from console*?
- In Java, **System** is a class defined in **java.lang** and **in**, **out** and **err** are the variables of the class **System**.
- Hence **System.out** is an object used for standard output stream and **System.in** and **System.err** are the objects for standard input stream and error.

### Method 1 : Reading input using Buffered Reader, Inputstream Reader, system. In

- When we want to read some character from the console we should make use of **system.in**.
- The character stream class **BufferedReader** is used for this purpose. In fact we should pass the variable **System.in** to **BufferedReader** object. Along with it we should also mention the abstract class of **BufferedReader** class which is **InputStreamReader**.

Hence the syntax is,

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

Enables us to read console input

For example the object named *obj* can be created as

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

- Then, using this object we invoke **read()** method. For e.g. **obj.read()**.
- But this is not the only thing needed for reading the input from console; we have to mention the exception **IOException** for this purpose. Hence if we are going to use **read()** method in our program then function **main** can be written like this –

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

- Let us understand all these complicated issues with the help of some simple Java program

### Writing Console Output

The simple method used for writing the output on the console is **write()**.

Here is the syntax of using **write()** method –

```
System.out.write(int b)
```

The bytes specified by *b* has to be written on the console. But typically **print()** or **println()** is used to write the output on the console. And these methods belong to **PrintWriter** class.

### Reading and Writing Files

**InputStream** is an abstract class for streaming the byte input. Various methods defined by this class are as follows.



Methods	Purpose
int available()	It returns total number of byte input that are available currently for reading.
void close()	The input source gets closed by this method. If we try to read further after closing the stream then IOException gets generated.
void reset()	This method resets the input pointer to the previously set mark.
long skip(long <i>n</i> )	This method ignores the <i>n</i> bytes of input. It then returns the actually ignored number of bytes.
int read()	It returns the next available number of bytes to read from the input stream. If this method returns the value -1 then that means the end of file is encountered.
int read(byte buffer[ ])	This method reads the number of bytes equal to length of <i>buffer</i> . Thus it returns the actual number of bytes that were successfully read.
int read(byte <i>buffer</i> [ ], int offset,int <i>n</i> )	This method returns the <i>n</i> bytes into the <i>buffer</i> which is starting at offset. It returns the number of bytes that are read successfully.

**OutputStream** is an abstract class for streaming the byte output. All these methods under this class are of void type. Various methods defined by this class are as follows

Methods	Purpose
void close()	This method closes the output stream and if we try to write further after using this method then it will generate <b>IOException</b>
void flush()	This method clears output buffers.
void write(int val)	This method allows to write a single byte to an output stream.
void write(byte buffer[ ])	This method allows to write complete array of bytes o an output stream.
void write(byte buffer[ ],int offset, int <i>n</i> )	This method writes <i>n</i> bytes to the output stream from an array <i>buffer</i> which is beginning at <i>buffer[offset]</i>

## **FileInputStream / FileOutputStream**

The **FileInputStream** class creates an **InputStream** using which we can read bytes from a file.

The two common constructors of **FileInputStream** are -

```
FileInputStream(String filename);
FileInputStream(File fileobject);
```

In the following Java program, various methods of **FileInputStream** are illustrated -

### **Java Program[FileStreamProg.java]**

```
import java.io.*;
class FileStreamProg
{
public static void main(String[] args)throws Exception
{
int n;
InputStream fobj=new FileInputStream("f:/I_O_programs/FileStreamProg.java");
System.out.println("Total available bytes: "+(n=fobj.available()));
int m=n-400;
System.out.println("\n Reading first "+m+" bytes at a time");
for(int i=0;i<m;i++)
System.out.print((char)fobj.read());
System.out.println("\n Skipping some text");
fobj.skip(n/2);
System.out.println("\n Still Available: "+fobj.available());
fobj.close();
}
}
```

The **FileOutputStream** can be used to write the data to the file using the **OutputStream**.

The constructors are-

```
FileOutputStream(String filename)
FileOutputStream(Object fileobject)
FileInputStream(String filename,boolean app);
FileInputStream(String fileobject,boolean app);
```

The *app* denotes that the append mode can be true or false. If the append mode is true then you can append the data in the file otherwise the data can not be appended in the file.

In the following Java program, we are writing some data to the file.

### **Java Program[FileOutStreamProg.java]**

```
import java.io.*;
class FileOutStreamProg
```

```

{
public static void main(String[] args)throws Exception
{
String my_text="India is my Country\n"+"and I love my country very much.";
byte b[]=my_text.getBytes();
OutputStream fobj=new FileOutputStream("f:/I_O_programs/output.txt");
for(int i=0;i<b.length;i++)
fobj.write(b[i]);
System.out.println("\n The data is written to the file");
fobj.close();
}
}

```

### **FilterInputStream / FilterOutputStream**

FilterStream class are those classes which wrap the input stream with the byte. That means using input stream you can read the bytes but if you want to read integers, doubles or strings you need to a filter class to wrap the byte input stream.

The syntax for FilterInputStream and FilterOutputStream are as follows –

FilterOutputStream(OutputStream o)

FilterInputStream(InputStream i)

The methods in the Filter stream are similar to the methods in **InputStream** and **OutputStream**.

### **DataInputStream / DataOutputStream**

**DataInputStream** reads bytes from the stream and converts them into appropriate primitive data types whereas the **DataOutputStream** converts the primitive data types into the bytes and then writes these bytes to the stream. The superclass of **DataInputStream** class is **FilterInputStream** and superclass of **DataOutputStream** class is **FilterOutputStream** class.

Various methods under these classes are –

Methods	Purpose
boolean readBoolean()	Reads Boolean value from the inputstream.
byte readByte()	Reads byte value from the inputstream.
char readChar()	Reads char value from the inputstream.

float readFloat()	Reads float value from the inputstream.
double readDouble()	Reads double value from the inputstream.
int readInt()	Reads integer value from the inputstream.
long readLong()	Reads long value from the inputstream.
String readUTF()	Reads string in UTF form.

### **BufferedInputStream / BufferedOutputStream**

The **BufferedInputStream** and **BufferedOutputStream** is an efficient class used for speedy read and write operations. All the methods of **BufferedInputStream** and **BufferedOutputStream** class are inherited from **InputStream** and **OutputStream** classes. But these methods add the buffers in the stream for efficient read and write operations. We can specify the buffer size otherwise the default buffer size is 512 bytes.

In the following Java program the **BufferedInputStream** and **BufferedOutputStream** classes are used.

## Autoboxing and Unboxing:

In Java, primitive data types are treated differently so do there comes the introduction of wrapper classes where two components play a role namely Autoboxing and Unboxing.

**Autoboxing** refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
- Assigned to a variable of the corresponding **wrapper class**.

**Unboxing** on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int. The Java compiler applies to unbox when an object of a wrapper class is:

- Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
- Assigned to a variable of the corresponding **primitive type**.

***Advantage of Autoboxing and Unboxing:***

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

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

**Simple Example of Autoboxing in java:**

```

class BoxingExample1 {
    public static void main(String args[]){
        int a=50;
        Integer a2=new Integer(a);//Boxing
        Integer a3=5;//Boxing
        System.out.println(a2+" "+a3);
    }
}

```

### Simple Example of Unboxing in java:

The automatic conversion of wrapper class type into corresponding primitive type, is known as Unboxing. Let's see the example of unboxing:

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

### Enumeration

**Enumerations or Java Enum** serve the purpose of representing a group of named constants in a programming language. Java Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command-line flags, etc. The set of constants in an enum type doesn't need to stay fixed for all time.

#### *Declaration of enum in Java*

Enum declaration can be done outside a Class or inside a Class but not inside a Method

#### **1. Declaration outside the class**

```
enum Color {
    RED,
    GREEN,
    BLUE;
}
public class Test {
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

#### **Declaration inside a class**

```
public class Test {
    enum Color {
        RED,
        GREEN,
```

```

        BLUE;
    }

    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}

```

### ***Properties of Enum in Java***

There are certain properties followed by Enum as mentioned below:

- Every enum is internally implemented by using Class.
- Every enum constant represents an **object** of type enum.
- Enum type can be passed as an argument to **switch** statements.
- Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using the enum Name. Since it is **final**, we can't create child enums.
- We can declare the **main() method** inside the enum. Hence we can invoke the enum directly from the Command Prompt.

### **Generics**

Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

**Note:** **Generics** does not work with primitive types (int, float, char, etc).

#### ***Java Generics Class***

We can create a class that can be used with any type of data. Such a class is known as generics Class.

// To create an instance of generic class

```
BaseType <Type> obj = new BaseType <Type>()
```

#### ***Java Generics Method***

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as Generics Method.

We have created a generic method named `genericsMethod`.

```
Public <T> void genericMethid(T data) { ....}
```