UNIT-3(Exceptional Handling,I/O)

Exception :

an exception is an abnormal condition that disrupts the normal flow of the program. It is an object which is thrown at runtime.

For example take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

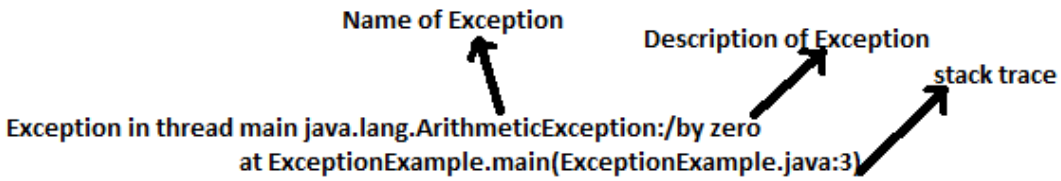
Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed..

Exception Example and Default Exception Handling in Java:

```
class ExceptionExample{
        public static void main(String args[]){
                int a=100/0;//exception raise
                  //rest of code
                int b=100/2;
                System.out.println("hello after exception:"+b);
                }
                }
```

```
C:\javaprograms\innerclasses>java ExceptionExample
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionExample.main(ExceptionExample.java:3)
```

In the above example an Arithmetic exception arise at line number 3 in main method, due to which program is terminated abnormally and rest of the code after exception is not executed. Since we did not handle it in the main method so JVM terminates the main method and give control to java Default Exception handler handles this exception as shown in above output as:



Exception Handling:

=>Defining alternative way to define the exception is called Exception handling.
=>Exception Handling is a mechanism to handle runtime errors such as ArithmeticException, ClassNotFoundException, IOException, SQLException, RemoteException, etc.

=>to handle exception we use try and catch block .
=>In try block we write the code that generate Exception.

=>In catch block we write the code that handle the Exception.

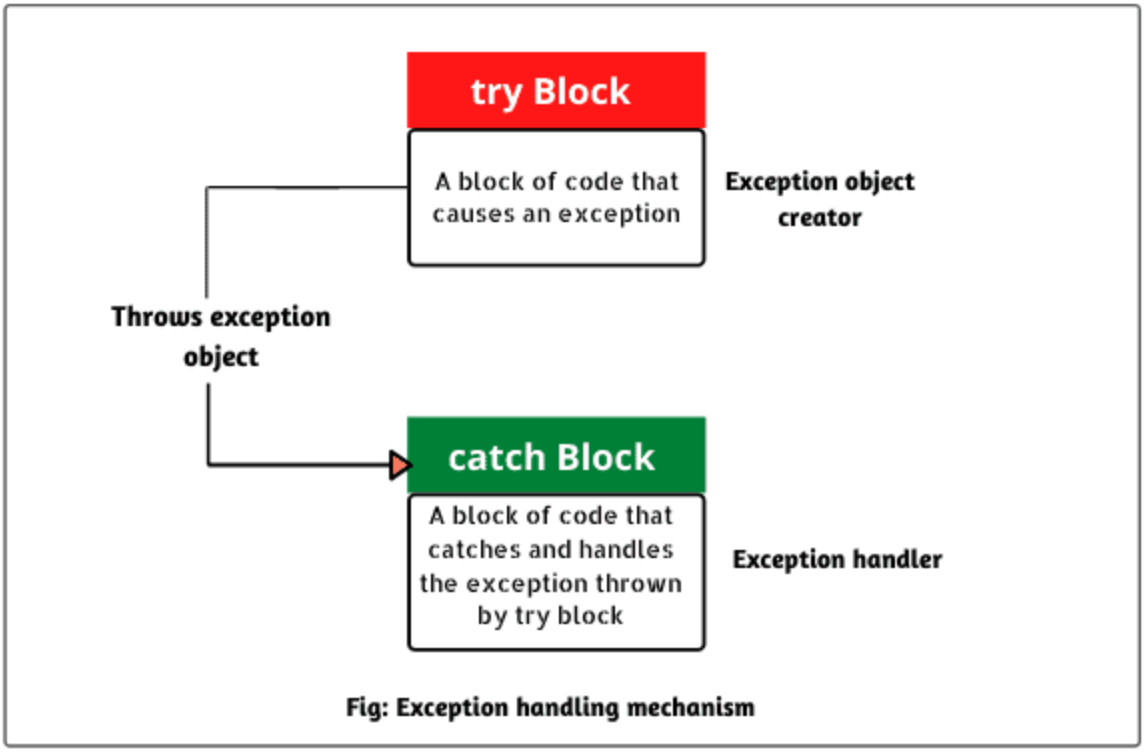


Fig: Exception handling mechanism

try-catch block: try and catch are keywords

try block:

try block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, not to keep the code in try block that will not throw an exception.

catch block:

catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Syntax of try-catch:

```
try{
//code that may throw an exception
}
catch(Exception_class_Name ref){

}
```

Example:

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

```
int a=100/0;//exception raise
  }
  catch(ArithmeticException e){
        System.out.println(e);
  }
int b=100/2;
//rest of code
System.out.println("hello after exception:"+b);
}
}
```

Output:

```
C:\javaprograms\innerclasses>java ExceptionExample
java.lang.ArithmeticException: / by zero
hello after exception:50
```

Now in the above program the program does not terminate abnormally since Exception is handled using try and catch block so rest of the code is also executed.

Advantage of Exception Handling:

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling.

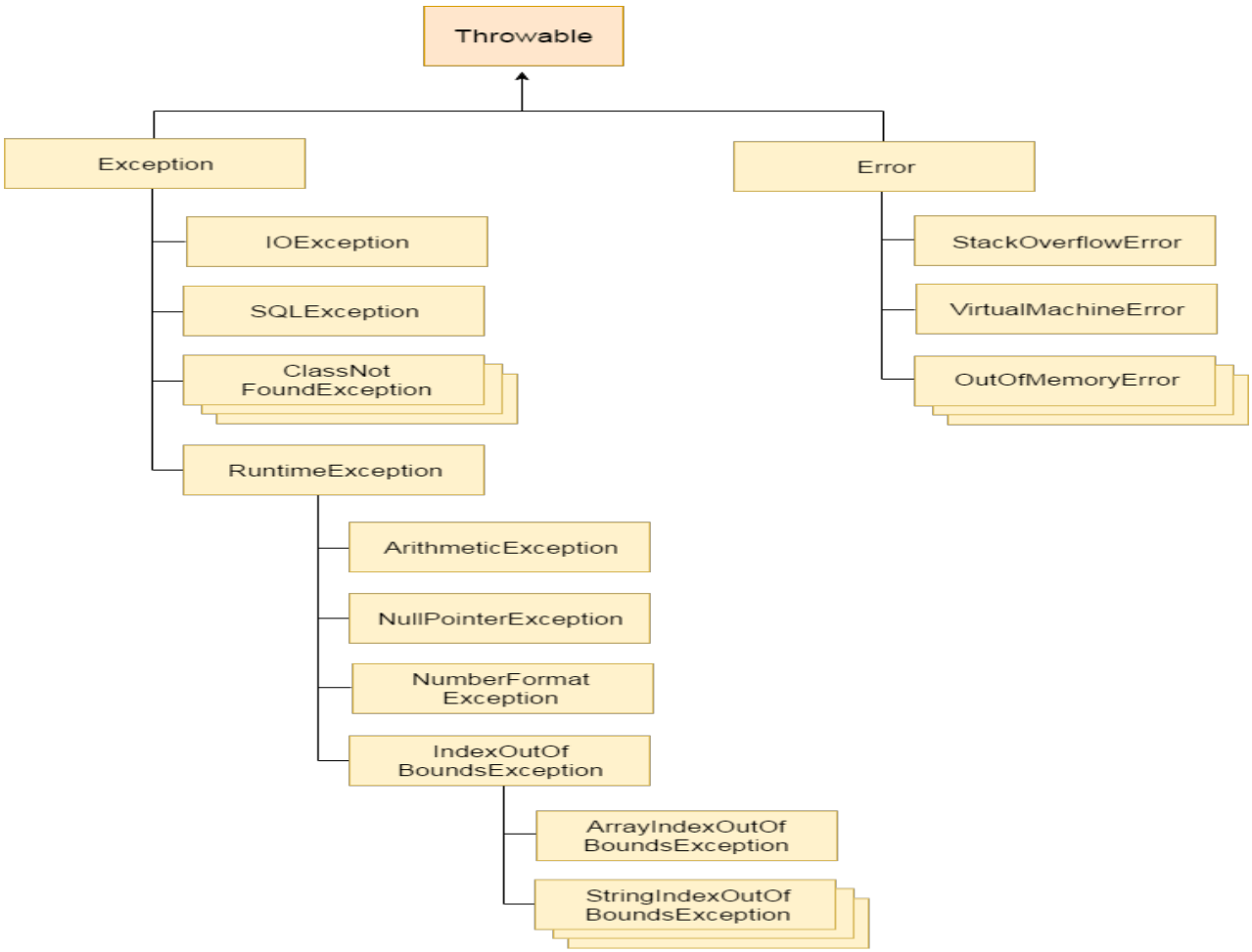
For example take a scenario:

statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling .

Hierarchy of Java Exception classes:

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception: The exception which are checked by compiler for smooth execution of program is called checked Exception. i.e compiler only checked whether these exception are handle by programmer or not because these are most frequently happened exceptions so these are checked by compiler.

The classes which directly inherit Throwable class except RuntimeException and Error are known as

checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

Example:

```
import java.io.*;
class CheckedEx{
public static void main(String args){
PrintWriter pw=new PrintWriter(abc.txt);
pw.println("Hello");
}
}
```

Output:

```
C:\javaprograms\innerclasses>javac CheckedEx.java
CheckedEx.java:4: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
PrintWriter pw=new PrintWriter("abc.txt");//file not found exception can be generated
             ^
```

This output is not program exception. This is only checked by compiler and found that FileNotFoundException is not handled ,so compiler is only telling to programmer to handle this exception.

2) Unchecked Exception:

Exception which are not checked by compiler whether these are handle by programmer or not are called unchecked exception. These are rarely generated exception so compiler does not check these exceptions.

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Example of Runtime Exception:

```
class  ExceptionExample{
        public static void main(String args[]){
                int a=100/0;//exception raise
                  //rest of code
                int b=100/2;
                System.out.println("hello after exception:"+b);
                }
                }
```

```
C:\javaprograms\innerclasses>java ExceptionExample
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionExample.main(ExceptionExample.java:3)
```

3) Error: Error are not caused by our code. Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use multi-catch block.

NOTE:

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

example of java multi-catch block:
Example:--

```
class MultiEx{
public static void main(String args[]){
int a[]={8,12,16,20,24};
int b[]={4,8,0,5};
 for(int i=0;i<a.length;i++){
try{
System.out.println("The value="+a[i]/b[i]);
   }
catch(ArithmeticException e){
System.out.println("Divide by zero ");
}
catch(ArrayIndexOutOfBoundsException e){
System.out.println("Array index is out of bound");
}
}
}}
```

Nested try block

=>The try block within a try block is known as nested try block.
=>Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Example:

```
class Nestedtry{
public static void main(String args[]){
int a[]={8,16,20,24,30};
 int b[]={2,4,0,8};
  for(int i=0;i<a.length;i++){
  try{
      try{
        System.out.println("The value="+a[i]/b[i]);
           }
       catch(ArithmeticException e){
       System.out.println("Number is divide by zero");
                                              }
       }
catch(ArrayIndexOutOfBoundsException e){
System.out.println("invalid array idex is accessing");
                                                      }
  }}}
```

finally block

=>finally block is always executed whether exception is handled or not.
=> finally block must be followed by try or catch block.
=> finally block is a block that is used *to execute important code* such as closing connection, closing files etc.
Example:

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int a=25/0;
   System.out.println(a);
     }
  catch(ArithmeticException e){
System.out.println("Divide by zero");
```

```
    }
  finally{
     System.out.println("finally block is always executed");
            }
  System.out.println("rest of the code...");
       }
}
```

Output:Divide by zero
finally block is always executed
rest of the code...

Note: For each try block there can be zero or more catch blocks, but only one finally block.

Built - in Exceptions:

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

1) A scenario where ArithmeticException occurs

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

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

---

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

---

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s="hello";
int i=Integer.parseInt(s);//NumberFormatException
```

---

4) A scenario where ArrayIndexOutOfBoundsException occurs

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

```
int a[]=new int[5];
```

a[6]=50; //ArrayIndexOutOfBoundsException

Checked Exception: Some checked exception are

FileNot FoundException:

InterruptedException:

ClassNotFoundException:

IOException:

throw keyword:

The throw keyword is used to explicitly throw an exception. We can throw either checked or uncheked exception by throw keyword. The throw keyword is mainly used to throw custom exception or user defined exception.
Sometimes it is required that Programmer throw an exception from his program and jvm catch that exception for normal flow of the program.

The syntax of java throw keyword is given below.

```
throw new exception-type;  //throw keyword is used to throw Exception object manually to
                           //JVM by the programmer explicitly.
```

example:

```
throw new ArithmeticException("divie by zero error");  //to handle Arithmetic exception
                                                        //object to JVM manually we
                                                       // we use throw keyword.
```

throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class ThrowEx{
   void validate(int age){
    if(age<18)
     throw new ArithmeticException("Age is not valid for voting");
     else
     System.out.println("welcome to vote");
  }
  public static void main(String args[]){
       ThrowEx ob=new ThrowEx();

      ob.validate(13);
          }
}
```

Output: Age is not valid for voting

throws keyword

The **throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as ArrayIndexOutOfBoundsException, it is programmers fault that he is not performing

check up before the code being used.

Syntax of throws
```
return_type   method_name()  throws  checkedException_class_name{
//method code
                    }
```

In a program, if there is a chance of raising an checked exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying **unreported exception must be caught or declared to be thrown**. To prevent this compile time error we can handle the exception in two ways:

1. By using **throws** keyword
2. By using try catch

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

```
// Java program to illustrate error in case
// of unhandled checked exception
class tst
{
   public static void  main(String[] args)
   {
      Thread.sleep(10000);
      System.out.println("Hello Checked Excetpion");
   }
}
```

Output:

error: unreported exception InterruptedException; must be caught or declared to be thrown

Explanation: In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute main() method which will cause InterruptedException.

1.By using throws keyword:

if we do not want to handle the checked exception thrown by the method we can also throw them using throws keyword from our method which is using it.

```
class ThrowsEx
{
public static void main(String[] args)throws InterruptedException
   {
      Thread.sleep(10000);
      System.out.println("Hello Checked Exception");
   }
}
```

Output:

Hello Checked Exception

2.By using try catch:

We can handle checked exception in our class by using them in try/cacth block.

```
class ThrowsEx
{
   public static
void
main(String[]
args)
 {
Try{
       Thread.sle
ep(10000);
}
catch(Interrupt
edException)
       System.ou
t.println("Hello
Checked
Exception");
   }
}
```

Difference between throw and throws in Java:

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

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

throw example

```
void m(){
throw new ArithmeticException("You are eligible for vote");
}
```

throws example

```
void m() throws InterruptedException{
//method code
}
```

Custom Exception or User Defined Exception:---

=>If you are creating your own Exception that is known as custom exception or user-defined exception.

=>Java custom exceptions are used to customize the exception according to user need.

=>By the help of custom exception, you can have your own exception and message.

=> For creating custom Exception we make a subclass of Exception class create constructor of this class and override toString method.

example of custom exception:--

```
class MyException extends Exception{
 String s;
 MyException(String s){
 this.s=s;
                }
```

```
public String toString(){
return s;
                    }
          }
class CustomException{
  void validate(int age) throws MyException{
    if(age<18)
     throw new MyException("not valid for voting");
    else
     System.out.println("valid,welcome to vote");
                          }
   public static void main(String args[]){
     CustomException ob=new CustomException();
     try{
     ob.validate(13);
     }
catch(MyException e)
{
System.out.println(e);
} }
}
```

Output: not valid for voting

Input/Output(I/O)

I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream:

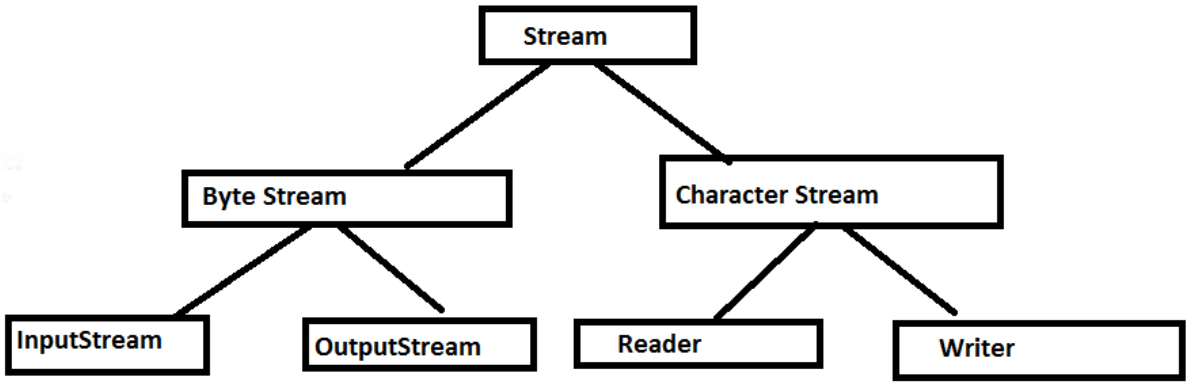
A stream is a sequence of data. A stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

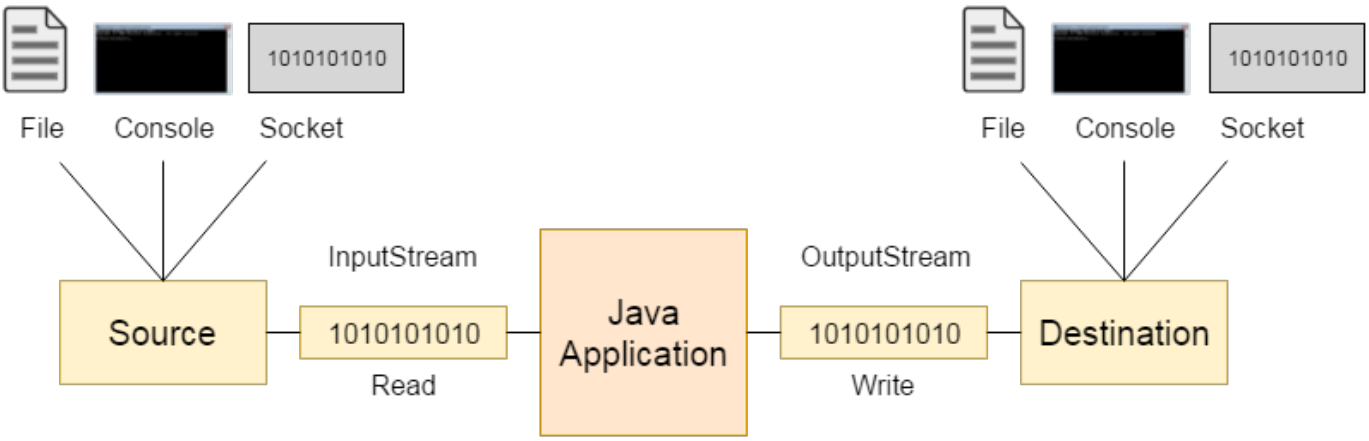
In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream





OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

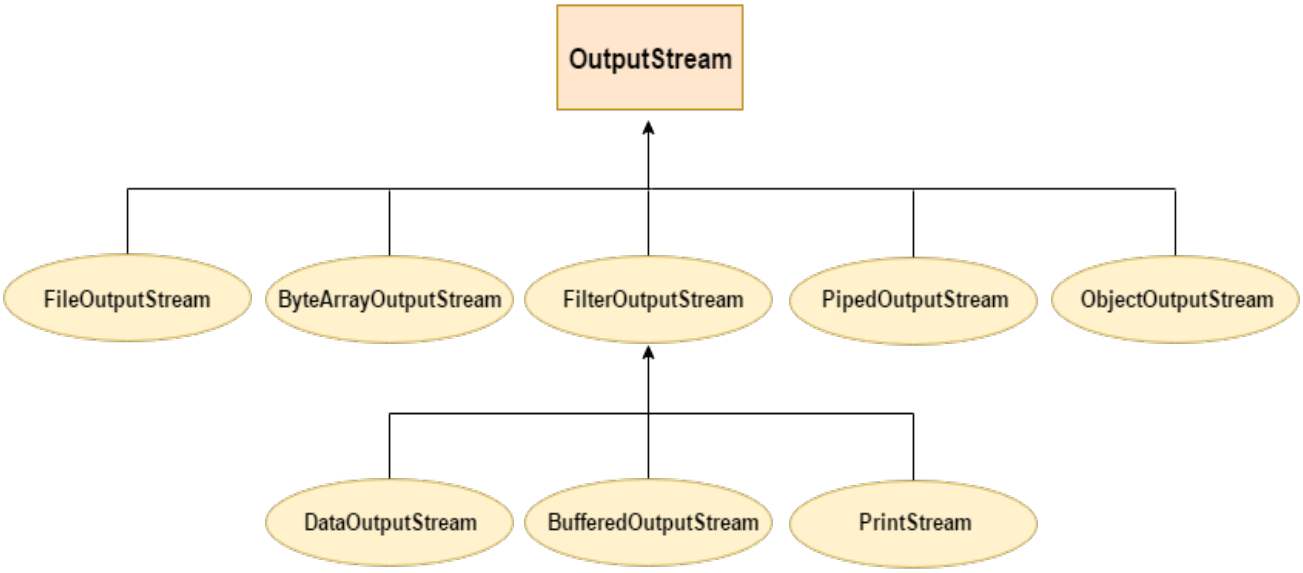
OutputStream class:

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to destination. .

Useful methods of OutputStream:-

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



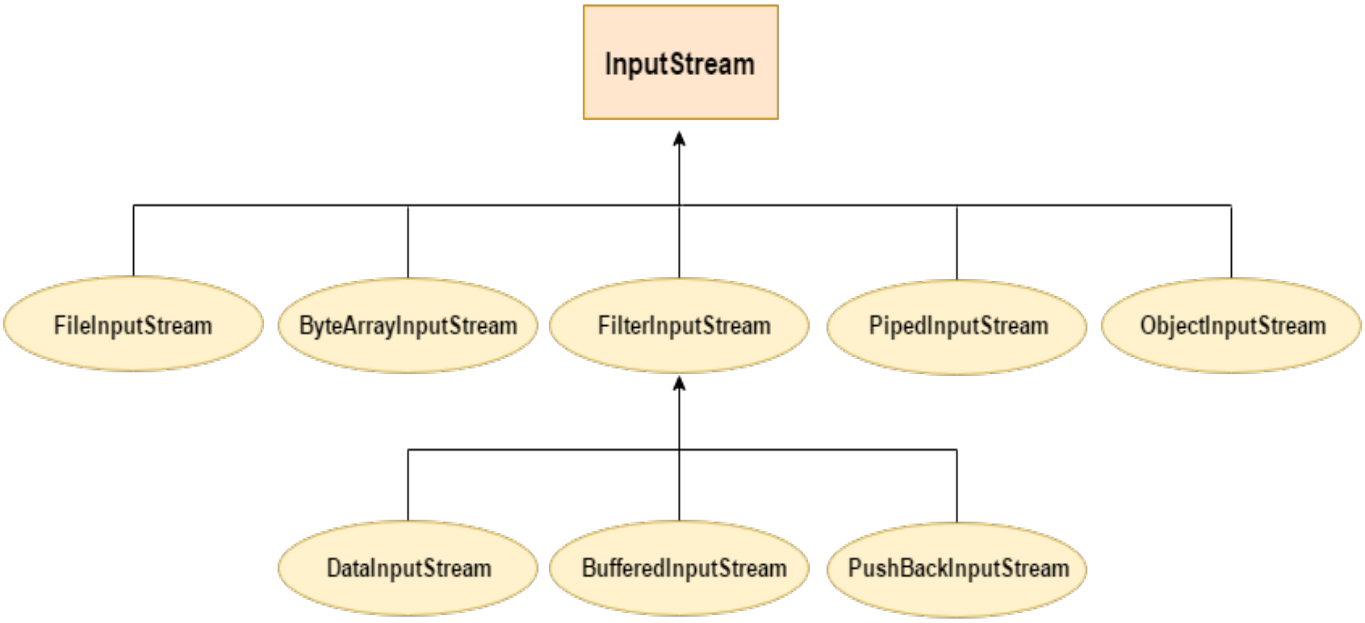
InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



FileOutputStream :

FileOutputStream is an output stream used for writing data to a file.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FileOutputStream Example 1: write byte

```
import java.io.FileOutputStream;
public class FileOutputStreamEx {
   public static void main(String args[]){
        try{
```

```
          FileOutputStream fout=new FileOutputStream("C:\\javaprograms\\io\\a.txt");
          fout.write(65);
          fout.close();
          System.out.println("success...");
         }catch(Exception e){System.out.println(e);}
    }
}
```

FileOutputStream example 2: write string

```
import java.io.FileOutputStream;
public class FileOutputStreamEx {
   public static void main(String args[]){
        try{
          FileOutputStream fout=new FileOutputStream("C:\\javaprograms\\io\\c.txt");
               String s="Hello FileInputStream";
                 byte[]b=s.getBytes();
          fout.write(b);
          fout.close();
          System.out.println("file write successfully...");
         }catch(Exception e)
                        {
                                System.out.println(e);
                                }
    }
}
```

FileInputStream :

FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.

FileInputStream example 1: read single character

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
      FileInputStream fin=new FileInputStream("C:\\javaprograms\\io\\c.txt");
         int i=fin.read();
          System.out.print((char)i);
           fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

FileInputStream example 2: read all characters

```
import java.io.FileInputStream;
public class InputStreamExample {
public static void main(String args[]){
try{
FileInputStream fin=new FileInputStream("C:\\javaprograms\\io\\c.txt ");
while(true){
int i=fin.read();
if(i==-1)//end of file
break;
System.out.print((char)i);
}
fin.close();

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

BufferedOutputStream :

BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws Exception{
FileOutputStream fout=new FileOutputStream("C:\\javaprograms\\io\\c.txt ");
BufferedOutputStream bout=new BufferedOutputStream(fout);
String s="Welcome to java BufferedOutputStream.";
byte b[]=s.getBytes();
bout.write(b);
bout.flush();
bout.close();
fout.close();
System.out.println("file write successfully");
}
}
```

BufferedInputStream : BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

Example of Java BufferedInputStream:

```
import java.io.*;
public class BufferedInputStreamEx{
public static void main(String args[]){
try{
FileInputStream fin=new FileInputStream("C:\\javaprograms\\io\\c.txt ");
BufferedInputStream bin=new BufferedInputStream(fin);
while(true){
 int i= bin.read();
  if(i==-1)
          break;
System.out.print((char)i);
}
bin.close();
fin.close();
}catch(Exception e){System.out.println(e);}
}
}
```

SequenceInputStream :

SequenceInputStream class is used to read data from multiple streams. It reads data sequentially (one by one).

SequenceInputStream Example:

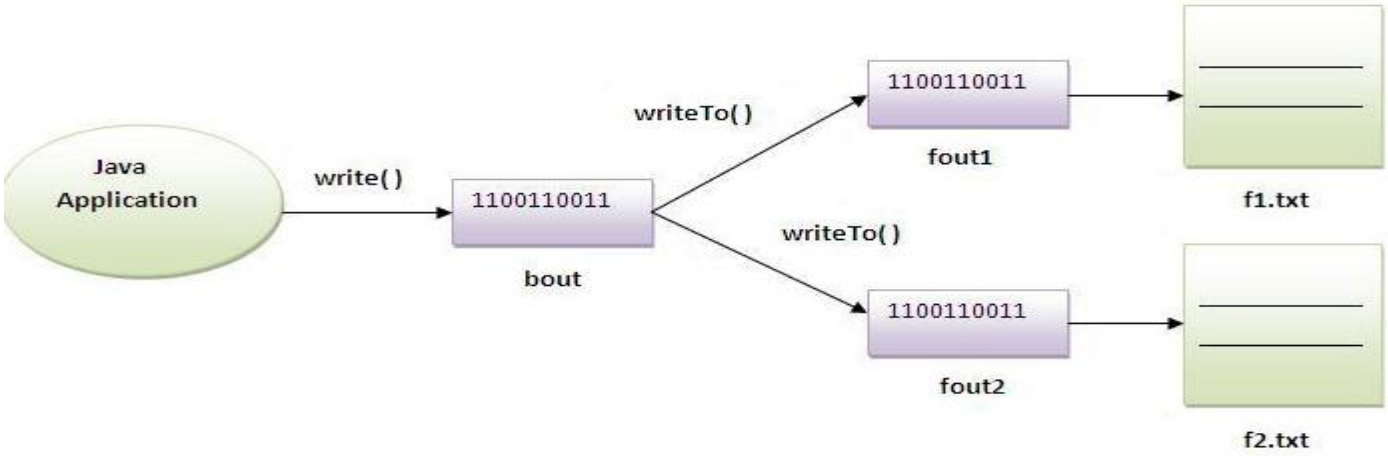
```
import java.io.*;
class SequenceInputStreamEx {
public static void main(String args[])throws Exception{
FileInputStream fin1=new FileInputStream("C:\\javaprograms\\io\\c.txt");
FileInputStream fin2=new FileInputStream("C:\\javaprograms\\io\\d.txt");
SequenceInputStream inst=new SequenceInputStream(fin1, fin2);
while(true){
int i=inst.read();
if(i==-1)
        break;
System.out.print((char)i);
}
inst.close();
fin1.close();
fin2.close();
}
}
```

ByteArrayOutputStream : ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.
The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

Example of ByteArrayOutputStream:

ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.




```
import java.io.*;
public class ByteArrayOutputStreamExample {
public static void main(String args[])throws Exception{
FileOutputStream fout1=new FileOutputStream("C:\\javaprograms\\io\\c.txt ");
FileOutputStream fout2=new FileOutputStream("C:\\javaprograms\\io\\b.txt ");
ByteArrayOutputStream bout=new ByteArrayOutputStream();
bout.write(65);
bout.writeTo(fout1);
bout.writeTo(fout2);
bout.flush();
bout.close();//has no effect
System.out.println("file write Successfully...");
}
}
```

ByteArrayInputStream :

The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte array as input stream.
ByteArrayInputStream class contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.
The buffer of ByteArrayInputStream automatically grows according to data.

Example of ByteArrayInputStream:

```
import java.io.*;
public class ReadExample {
public static void main(String[] args) throws IOException {
byte[] buf = { 35, 36, 37, 38 };
// Create the new byte array input stream
ByteArrayInputStream byt = new ByteArrayInputStream(buf);
```

```
int k = 0;
while ((k = byt.read()) != -1) {
//Conversion of a byte into character
char ch = (char) k;
System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);
}
}
}
```

Console Class:

Console class is be used to get input from console. It provides methods to read texts and passwords.
If you read password using Console class, it will not be displayed to the user.
The java.io.Console class is attached with system console internally.

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

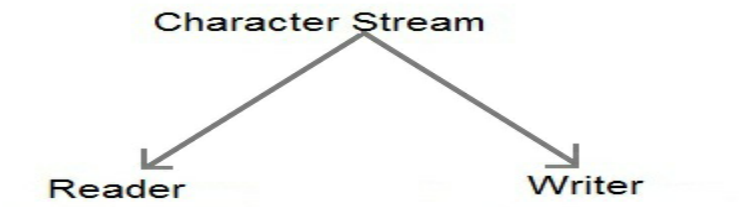
Console Example to read password:

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

Character Stream:

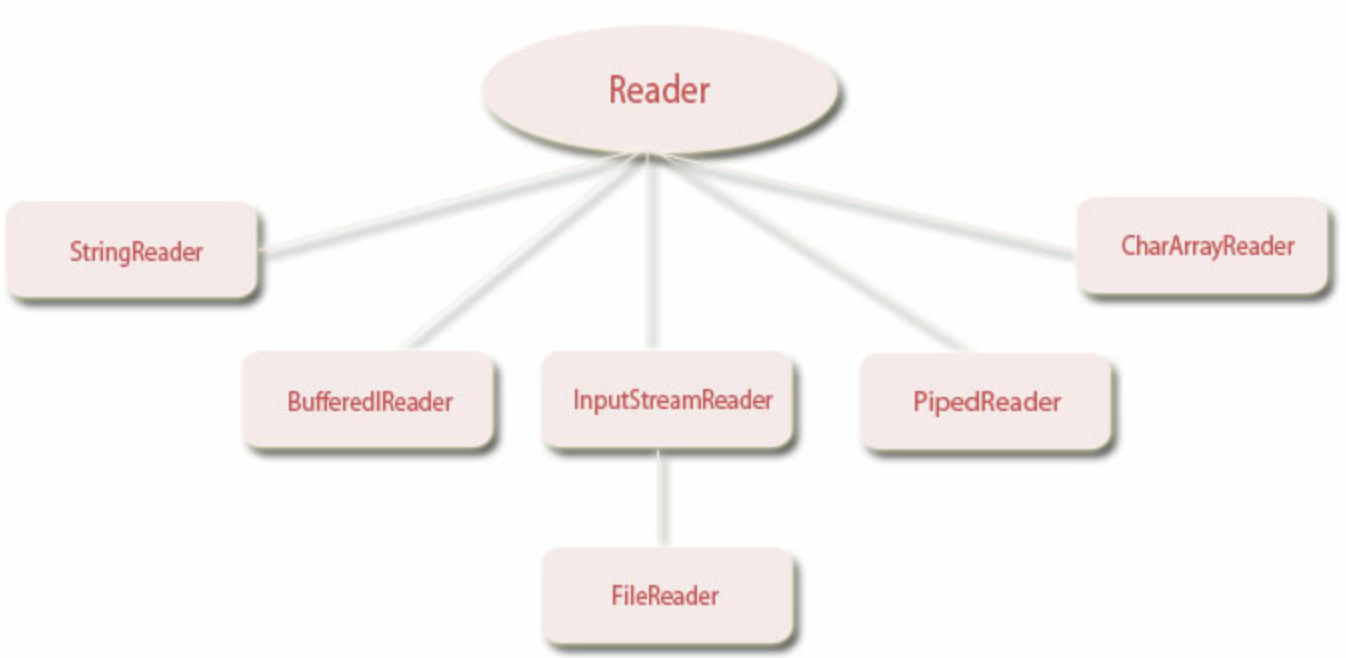
Character Streams

- Character streams are defined by using two class hierarchies
- At the top are two abstract classes, Reader and Writer
- These abstract classes handle Unicode character streams
- The abstract classes Reader and Writer define several key methods that the other stream classes implement
- Two of the most important methods are read( ) and write( ), which read and write characters of data
- These methods are overridden by derived stream classes.



Reader: Reader class and its subclasses are used to read characters from source.

Reader class is a base class of all the classes that are used to read characters from a file, memory or console. Reader is an abstract class and hence we can't instantiate it but we can use its subclasses for reading characters from the input stream.



Hierarchy of Reader classes

Methods of Reader class.

Methods	Description
int read()	This method reads a characters from the input stream.
int read(char[] ch)	This method reads a chunk of characters from the input stream and store them in its char array, ch.
close()	This method closes this output stream and also frees any system resources connected with it.

Writer :

Writer class and its subclasses are used to write characters to a file, memory or console. Writer is an abstract class and hence we can't create its object but we can use its subclasses for writing characters to the output stream.

Hierarchy of Writer classes

Methods: Methods of Writer class provide support for writing characters to the output stream. As this is an abstract class. Hence, some undefined abstract methods are defined in the subclasses of OutputStream.

Methods	Description
abstract void flush()	This method flushes the output steam by forcing out buffered bytes to be written out.
void write(int c)	This method writes a characters(contained in an int) to the output stream.
void write(char[] arr)	This method writes a whole char array(arr) to the output stream.
abstract void close()	This method closes this output stream and also frees any resources connected with this output stream.

FileWriter Class:

FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.
Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.

FileWriter Example

```
import java.io.FileWriter;
public class FileWriterExample {
public static void main(String args[]){
try{
FileWriter fw=new FileWriter("c:\\javaprograms\\io\\a.txt");
String s="Welcome to FileWriter class which is characterStream type class";
fw.write(s);
fw.close();
}catch(Exception e){System.out.println(e);}
System.out.println("Write Successfully...");
}
}
```

FileReader Class

FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.
It is character-oriented class which is used for **file** handling in java.

```
import java.io.FileReader;
public class FileReaderExample {
public static void main(String args[]){
try{
FileReader fr=new FileReader("c:\\javaprograms\\a.txt");
while(true)
{
        int i=fr.read();
        if(i==-1)
                break;
        System.out.print((char)i);
}
fr.close();

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

BufferedWriter Class:

BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits Writer class. The buffering characters are used for providing

the efficient writing of single arrays, characters, and strings.

Example of BufferedWriter: writing the data to a text file b.txt using BufferedWriter.

```
import java.io.*;
public class BufferedWriterExample {
public static void main(String[] args)  {
try{
FileWriter writer = new  FileWriter("c:\\javaprograms\\b.txt");
BufferedWriter buffer = new BufferedWriter(writer);
String s="Welcome to BufferedWriter class";
buffer.write(s);
buffer.close();
}
catch(Exception e){
        System.out.println(e);
}
System.out.println("write Successfully..");
}
}
```

BufferedReader Class

BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits Reader class.

BufferedReader Example

Reading the data from the text file **b.txt** using Java BufferedReader class.

```
import java.io.*;
public class BufferedReaderExample {
public static void main(String[] args)  {
try{
FileReader reader = new FileReader("c:\\javaprograms\\b.txt");
BufferedReader buffer = new BufferedReader(reader);
while(true){
int i=buffer.read();
 if(i==-1)
         break;
 System.out.print((char)i);

}
reader.close();
buffer.close();
}
```

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

Reading data from console by InputStreamReader and BufferedReader

connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

```
import java.io.*;
public class BufferedReaderExample{
public static void main(String args[])throws Exception{
InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);
System.out.println("Enter your name");
String name=br.readLine(); //used to read string value
System.out.println("Enter your age");
 int age=br.read();  //used to read integer value

System.out.println("Welcome "+name +"Your age is:"+age);
}
}
```

CharArrayReader Class:

The CharArrayReader is composed of two words: CharArray and Reader. The CharArrayReader class is used to read character array as a reader (stream). It inherits Reader class.

Example of CharArrayReader Class: example to read a character using Java CharArrayReader class.

```
import java.io.CharArrayReader;
public class CharArrayExample{
  public static void main(String[] ag) throws Exception {
    char[] ary = { 'c', 'h', 'a', 'r', 'A', 'r', 'r', 'a', 'y' };
    CharArrayReader     car = new CharArrayReader(ary);

   // Read until the end of a file
        while(true){
                int i=car.read();
                if(i==-1)
                        break;
                char ch = (char)i;
```

```
      System.out.print(ch + " : ");
      System.out.println(i);

        }
       }
}
```

CharArrayWriter Class:

The CharArrayWriter class can be used to write common data to multiple files. This class inherits Writer class. Its buffer automatically grows when data is written in this stream.

Example of CharArrayWriter Class:

writing a common data to 5 files f1.txt, f2.txt, f3.txt ,f4.txt and f5.txt.

```
import java.io.CharArrayWriter;
import java.io.FileWriter;
class CharArrayWriterEx{
public static void main (String args[])throws Exception{
CharArrayWriter caw=new CharArrayWriter();
String s="Hello CharArrayWriter class.";
   caw.write(s);
   FileWriter fw1=new FileWriter("c:\\javaprograms\\io\\f1.txt");
   FileWriter fw2=new FileWriter("c:\\javaprograms\\io\\f2.txt");
   FileWriter fw3=new FileWriter("c:\\javaprograms\\io\\f3.txt");
   FileWriter fw4=new FileWriter("c:\\javaprograms\\io\\f4.txt");
   FileWriter fw5=new FileWriter("c:\\javaprograms\\io\\f5.txt");
   caw.writeTo(fw1);
    caw.writeTo(fw2);
   caw.writeTo(fw3);
   caw.writeTo(fw4);
   caw.writeTo(fw5);
  fw1.close();
    fw2.close();
         fw3.close();
         fw4.close();
         fw5.close();
      System.out.println("Files write successfully...");
         }

}
```

PrintWriter class:

PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

PrintWriter Example

example of writing the data on a **console** and in a **text file** **a.txt** using Java PrintWriter class.

```
 import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample {
   public static void main(String[] args) throws Exception {
    //Data to write on Console using PrintWriter
     PrintWriter pw = new PrintWriter(System.out);
String s="Welcome to PrintWriter class."
     pw.write();
     pw.flush();
     pw.close();
//Data to write in File using PrintWriter
      File f= new File("c:\\javaprograms\\io\m.txt")
     PrintWriter   pw1 = new PrintWriter();

 pw1.write(" Java, Spring, Hibernate, Android, PHP etc.");
       pw1.flush();
       pw1.close();
f.close();
   }
}
```

OutputStreamWriter:

OutputStreamWriter is a class which is used to convert character stream to byte stream, the characters are encoded into byte using a specified charset. write() method calls the encoding converter which converts the character into bytes. The resulting bytes are then accumulated in a buffer before being written into the underlying output stream. The characters passed to write() methods are not buffered. We optimize the performance of OutputStreamWriter by using it with in a BufferedWriter so that to avoid frequent converter invocation.

```
import java.io.*;
  public class OutputStreamWriterEx {
      public static void main(String[] args) {

          try {
             OutputStream os = new FileOutputStream("c:\\javaprograms\\io\\out.txt");
             Writer osw = new OutputStreamWriter(os);
     String s="Hello OutputStreamWriter";
             osw.write(s);

             osw.close();
          } catch (Exception e) {
```

```
            e.getMessage();
        }
                    System.out.println("Write successfully..");
    }

  }
```

InputStreamReader:

An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

```
public class InputStreamReaderExample {
   public static void main(String[] args) {
      try  {
         InputStream stream = new FileInputStream("file.txt");
         Reader reader = new InputStreamReader(stream);
         int data = reader.read();
         while (data != -1) {
            System.out.print((char) data);
            data = reader.read();
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

File Class

The File class is an abstract representation of file and directory pathname. A pathname can be either absolute or relative.

The File class have several methods for working with directories and files such as creating new directories or files, deleting and renaming directories or files, listing the contents of a directory etc.

```
import java.io.*;
public class FileEx {
   public static void main(String[] args) {

      try {
         File file = new File("File.txt");
         if (file.createNewFile()) {
            System.out.println("New File is created!");
         } else {
            System.out.println("File already exists.");
```

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

    }
}
```

Ex:

```
import java.io.*;
public class FileExample {
public static void main(String[] args) {
    File f=new File("c:\\javaprograms");
    String filenames[]=f.list();
    for(String filename:filenames){
        System.out.println(filename);
    }
}
}
```