Overridding:

The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement.

```
class Animal{

public void move(){
System.out.println("Animals can move");
}
}
class Dog extendsAnimal{
public void move(){
System.out.println("Dogs can walk and run");
}
public void bark(){
System.out.println("Dogs can bark");
}
}

public class TestDog{

public static void main(String args[]){
Animal a =new Animal();// Animal reference and object
Animal b =new Dog();// Animal reference but Dog object

  a.move();// runs the method in Animal class
  b.move();//Runs the method in Dog class
  b.bark();
}
}
```

super Keyword:

When invoking a superclass version of an overridden method the super keyword is used.

```
class Animal{

public void move(){
System.out.println("Animals can move");
}
}
class Dog extends Animal{
public void move(){
super.move();// invokes the super class method
System.out.println("Dogs can walk and run");
}
}

public class TestDog{

public static void main(String args[]){

Animal b =new Dog();// Animal reference but Dog
 object
b.move();//Runs the method in Dog class
}
}
```

this is a **keyword in Java.** It can be used inside the method or constructor of a class. It(this) works as a reference to the current object, whose method or constructor is being invoked.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){
System.out.println(rollno+" "+name+" “+fee);
}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:
0 null 0.0
0 null 0.0

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000
112 sumit 6000
```

```
//this keyword to invoke a method
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

```
//this keyword to invoke current constructor
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

Polymorphism:

It is the ability of an object to take on many forms. The most common use of polymorphism in OOP, occurs when a parent class reference is used to refer to a child class object.

//Animal,java

```
public class Animal{
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
//Horse.java
class Horse extends Animal{
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){
      Animal obj = new Horse();
      obj.sound();
    }
}
//Cat.java
public class Cat extends Animal{
    @Override
    public void sound(){
        System.out.println("Meow");
    }
    public static void main(String args[]){
      Animal obj = new Cat();
      obj.sound();
    }
}
```

Compile Time Polymorphism: or Static Binding

```
class Overload
{
    void demo (int a)
    {
       System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
       System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
       System.out.println("double a: " + a);
       return a*a;
    }
}
class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
        double result;
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

Static and dynamic binding
Static Binding that happens at compile time and **Dynamic Binding** that happens at runtime.

Static Binding:
The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is **compile-time**. we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overriden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it.

Dynamic Binding or Late Binding
When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. **Method Overriding** is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed.

The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens.

Static Binding Example:

```
class Human{
    public static void walk()
    {
        System.out.println("Human walks");
    }
}
class Boy extends Human{
    public static void walk(){
        System.out.println("Boy walks");
    }
    public static void main( String args[]) {
        /* Reference is of Human type and object is
         * Boy type
         */
        Human obj = new Boy();
        /* Reference is of HUman type and object is
         * of Human type.
         */
        Human obj2 = new Human();
        obj.walk();
        obj2.walk();
    }
}
```

Dynamic Binding:

```
class Human{
    //Overridden Method
    public void walk()
    {
        System.out.println("Human walks");
    }
}
class Demo extends Human{
    //Overriding Method
    public void walk(){
        System.out.println("Boy walks");
    }
    public static void main( String args[]) {
        /* Reference is of Human type and object is
         * Boy type
         */
        Human obj = new Demo();
        /* Reference is of HUman type and object is
         * of Human type.
         */
        Human obj2 = new Human();
        obj.walk();
        obj2.walk();
    }
}
```

Abstraction:

An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclass. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
        System.out.println("Woof");
   }
   public static void main(String args[]){
        Animal obj = new Dog();
        obj.sound();
   }
}
```

```
abstract class AbstractDemo{
   public void myMethod(){
      System.out.println("Hello");
   }
   abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

   public void anotherMethod() {
       System.out.print("Abstract method");
   }
   public static void main(String args[])
   {
      //error: You can't create object of it
      AbstractDemo obj = new AbstractDemo();
      obj.anotherMethod();
   }
}
```

we cannot instantiate an abstract class. This program throws a compilation error.

Output:
Unresolved compilation problem: Cannot instantiate the type AbstractDemo

```java
//Example of Abstract class and method

abstract class MyClass{
   public void disp(){
     System.out.println("Concrete method of parent class");
   }
   abstract public void disp2();
}

class Demo extends MyClass{
   /* Must Override this method while extending
    * MyClas
    */
   public void disp2()
   {
      System.out.println("overriding abstract method");
   }
   public static void main(String args[]){
      Demo obj = new Demo();
      obj.disp2();
   }
}
```

Output:

```
Output:
overriding abstract method
```

```java
//abstract class
abstract class Sum{
   /* These two are abstract methods, the child class
    * must implement these methods
    */
   public abstract int sumOfTwo(int n1, int n2);
   public abstract int sumOfThree(int n1, int n2, int n3);

   //Regular method
   public void disp(){
      System.out.println("Method of class Sum");
   }
}
//Regular class extends abstract class
class Demo extends Sum{

   /* If I don't provide the implementation of these two methods, the
    * program will throw compilation error.
    */
   public int sumOfTwo(int num1, int num2){
      return num1+num2;
   }
```

```
public int sumOfThree(int num1, int num2, int num3){
        return num1+num2+num3;
    }
    public static void main(String args[]){
        Sum obj = new Demo();
        System.out.println(obj.sumOfTwo(3, 7));
        System.out.println(obj.sumOfThree(4, 3, 19));
        obj.disp();
    }
}
```

Output:

```
10
26
Method of class Sum
```

Encapsulation:

The whole idea behind encapsulation is to hide the implementation details from users. Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

```
//EncapsTest.java
class EncapsulationDemo{
    private int ssn;
    private String empName;
    private int empAge;

    //Getter and Setter methods
    public int getEmpSSN(){
        return ssn;
    }

    public String getEmpName(){
        return empName;
    }

    public int getEmpAge(){
        return empAge;
    }

    public void setEmpAge(int newValue){
        empAge = newValue;
    }
```

```
public void setEmpName(String newValue){
        empName = newValue;
    }

    public void setEmpSSN(int newValue){
        ssn = newValue;
    }
}

public class EncapsTest{
    public static void main(String args[]){
        EncapsulationDemo obj = new EncapsulationDemo();
        obj.setEmpName("Mario");
        obj.setEmpAge(32);
        obj.setEmpSSN(112233);
        System.out.println("Employee Name: " + obj.getEmpName());
        System.out.println("Employee SSN: " + obj.getEmpSSN());
        System.out.println("Employee Age: " + obj.getEmpAge());
    }
}
```

Output:

```
Employee Name: Mario
Employee SSN: 112233
Employee Age: 32
```

`Interface:`

An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: Java abstract method). Also, the variables declared in an interface are public, static & final by default.

The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

```java
interface MyInterface
{
    /* compiler will treat them as:
     * public abstract void method1();
     * public abstract void method2();
     */
    public void method1();
    public void method2();
}
class Demo implements MyInterface
{
    public void method1()
    {
    System.out.println("implementation of method1");
     }
     public void method2()
     {
    System.out.println("implementation of method2");
     }
     public static void main(String arg[])
     {
    MyInterface obj = new Demo();
    obj.method1();
     }
}
```

Interface and Inheritance

```
interface Inf1{
   public void method1();
}
interface Inf2 extends Inf1 {
   public void method2();
}
public class Demo implements Inf2{
   /* Even though this class is only implementing the
    * interface Inf2, it has to implement all the methods
    * of Inf1 as well because the interface Inf2 extends Inf1
    */
     public void method1(){
   System.out.println("method1");
     }
     public void method2(){
   System.out.println("method2");
     }
     public static void main(String args[]){
   Inf2 obj = new Demo();
  obj.method1();
   obj.method2();
     }
}
```

enum

Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

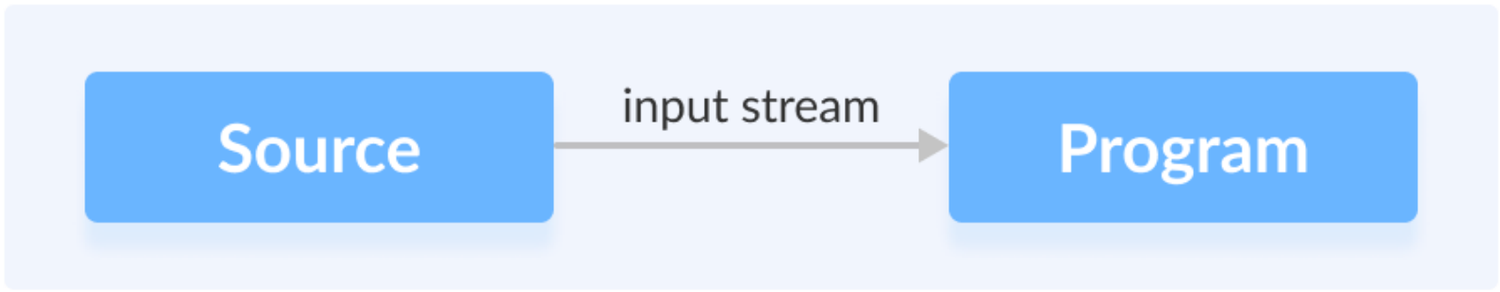
The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

```
import java.util.Vector;
import java.util.Enumeration;

public class EnumerationTester{

public static void main(String args[]){
Enumeration days;
Vector dayNames =newVector();
dayNames.add("Sunday");
 dayNames.add("Monday");
 dayNames.add("Tuesday");
 dayNames.add("Wednesday");
 dayNames.add("Thursday");
 dayNames.add("Friday");
 dayNames.add("Saturday");
 days = dayNames.elements();
while(days.hasMoreElements())
{ System.out.println(days.nextElement());
}
}
}
```

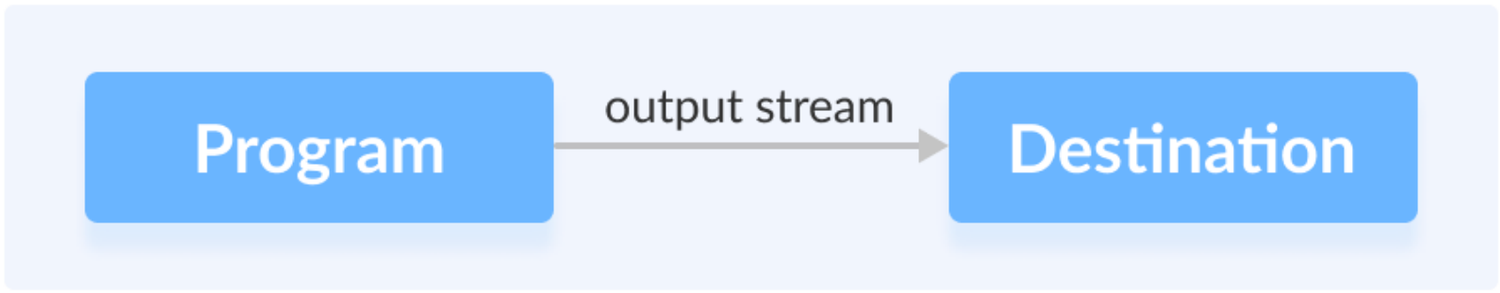
File, Stream and I/O

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.

Reading data from source



Writing data to destination



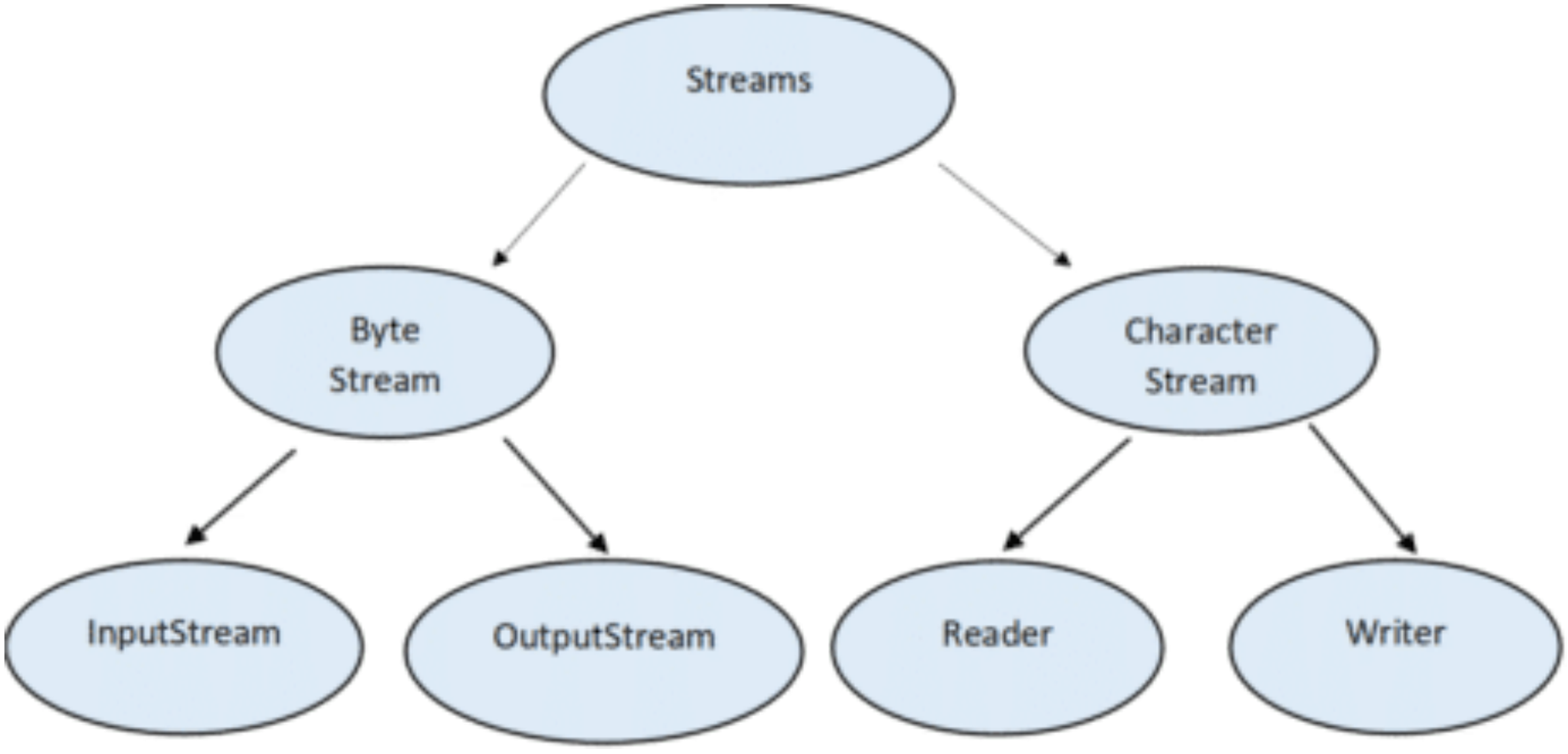
Streams are the sequence of bits(data).
There are two types of streams:

- Input Streams
- Output Streams

Input Streams:
Input streams are used to read the data from various input devices like keyboard, file, network, etc.
Output Streams:
Output streams are used to write the data to various output devices like monitor, file, network, etc.

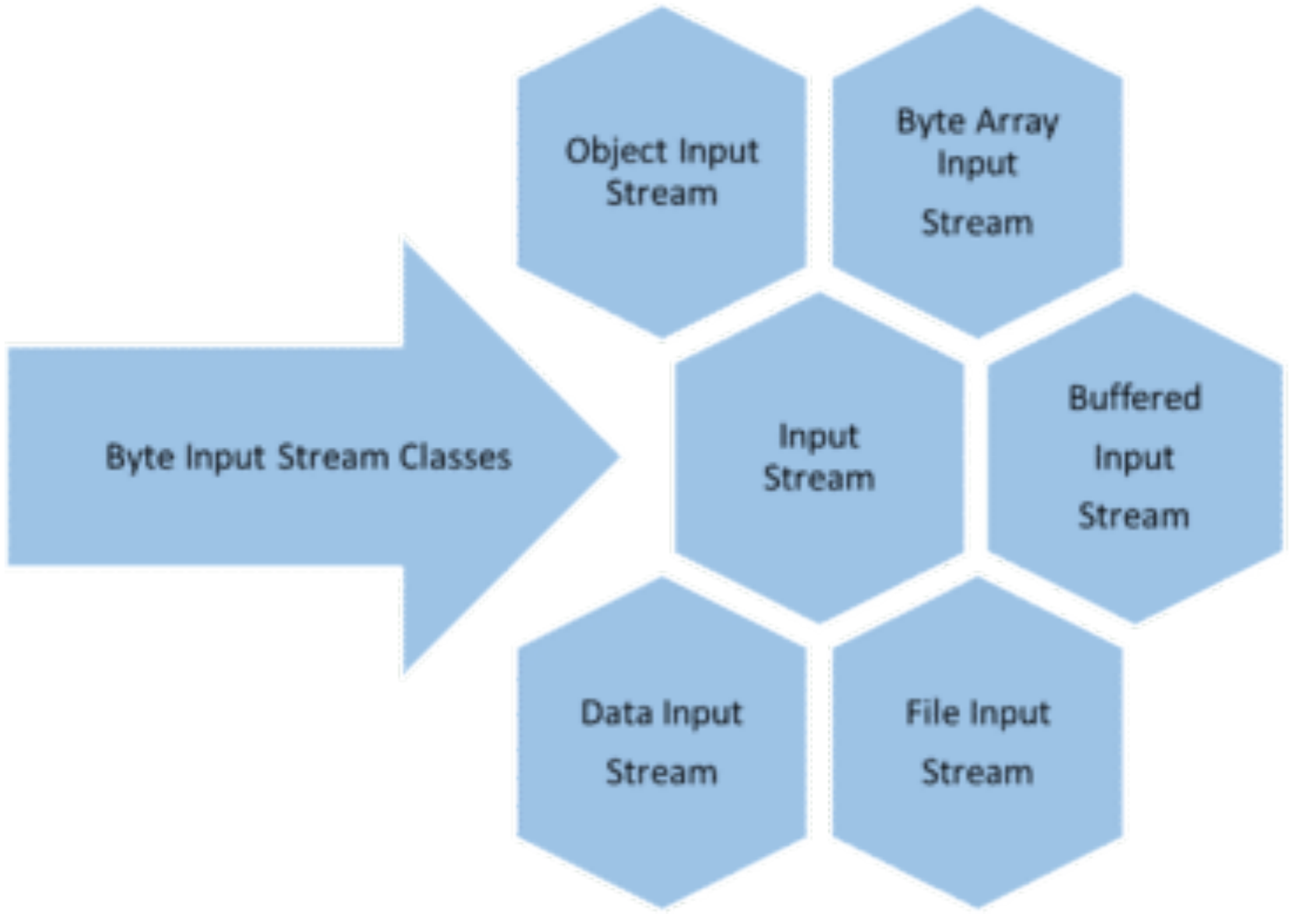
There are two types of streams based on data:

- **Byte Stream**: used to read or write byte data.
- **Character Stream**: used to read or write character data.

- 

Byte Input Stream:

- These are used to read byte data from various input devices.
- InputStream is an abstract class and it is the super class of all the input byte streams.

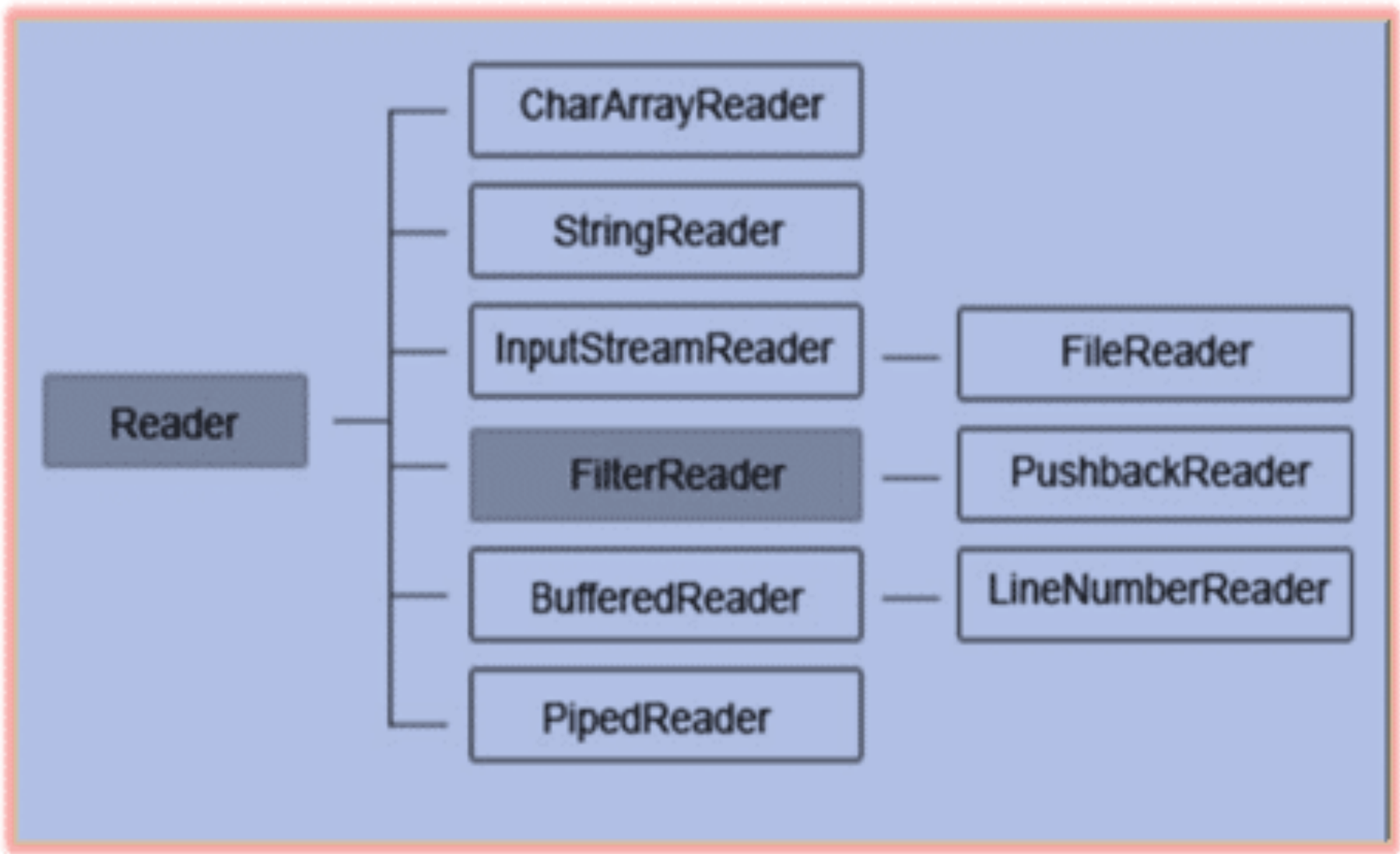


Byte Streams

```
import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException
   {
      FileReader in = null; FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
      while ((c = in.read()) != -1) {
      out.write(c);
      }
}finally {
if (in != null) {
in.close();
      }
      if (out != null) { out.close();
      }
}
}
}
```

Character Input Stream:

- These are used to read char data from various input devices.
- Reader is an abstract class and is the super class for all the character input streams.

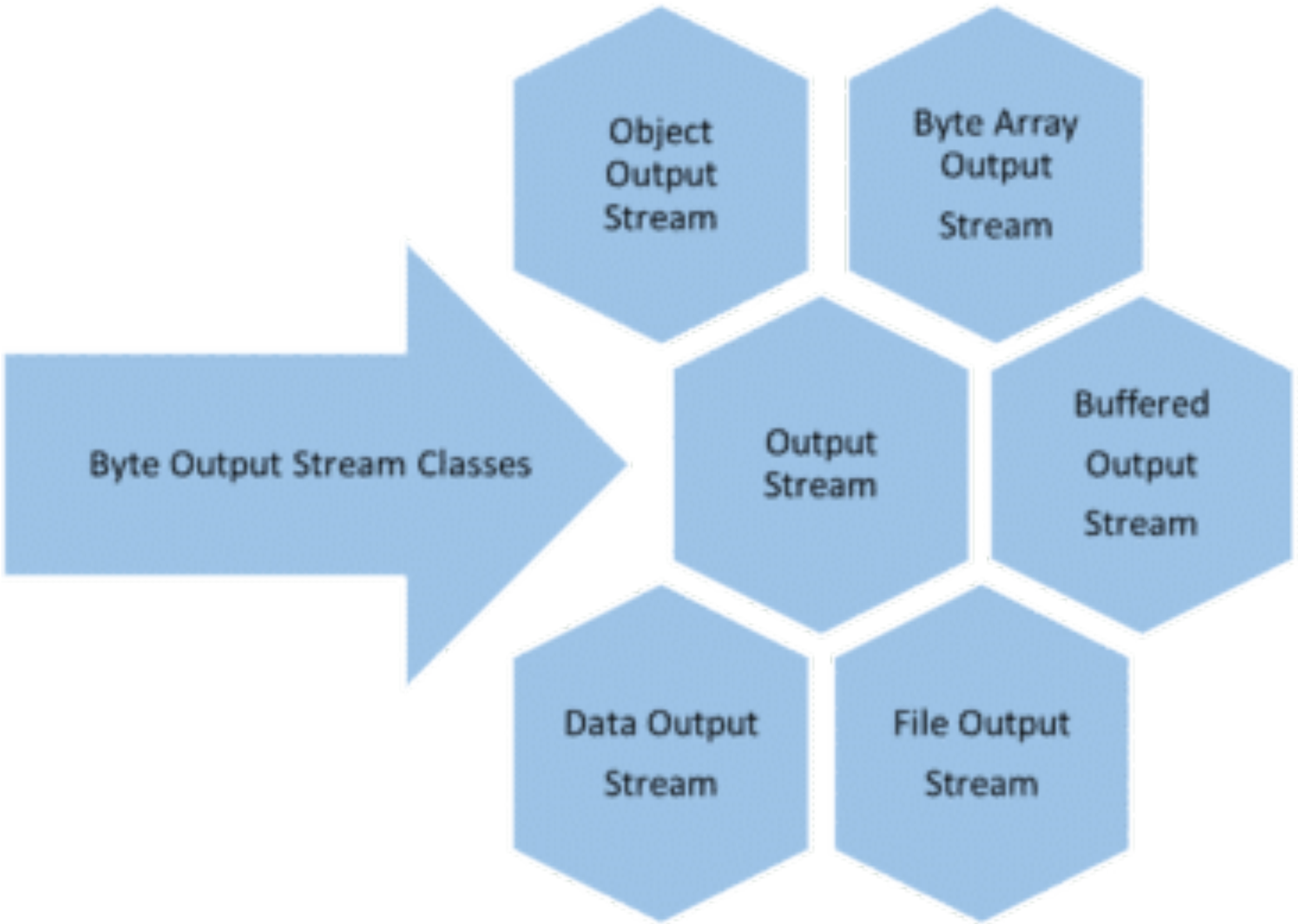


Character Stream

```
import java.io.*;

public class ReadConsole {
   public static void main(String args[]) throws IOException
   {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Press Enter to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         }
         while(c != '\n');
      }
      finally {
         if (cin != null) {
         cin.close();

         }
      }
   }
}
```

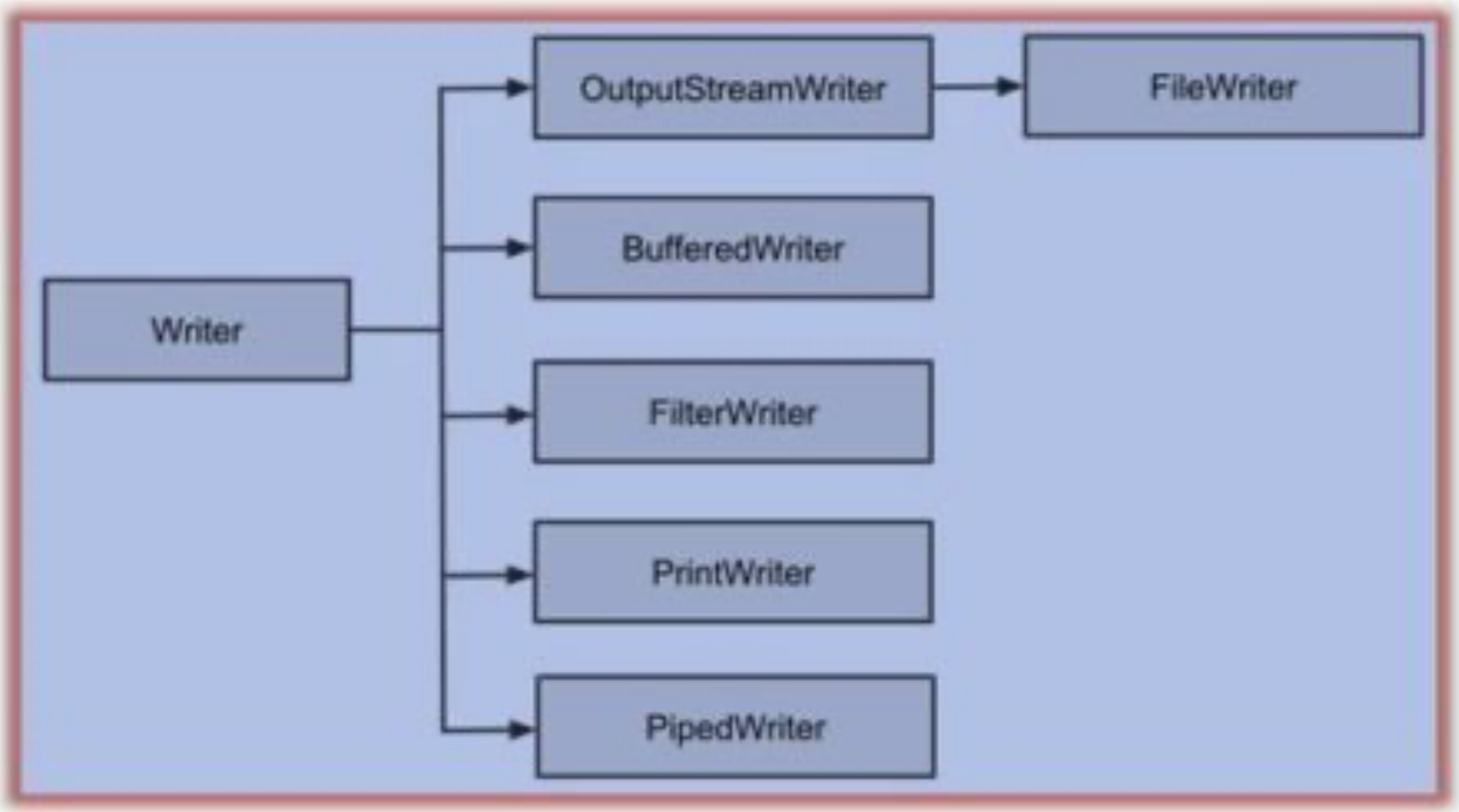
Byte Output Stream:

- These are used to write byte data to various output devices.
- Output Stream is an abstract class and it is the superclass for all the output byte streams.



Character Output Stream:

- These are used to write char data to various output devices.
- Writer is an abstract class and is the super class of all the character output streams.



File Navigation and I/O:

File Class:

This class is used for creation of files and directories, file searching, file deletion etc.

SN	Methods with Description
1	**public String getName()** Returns the name of the file or directory denoted by this abstract pathname.
2	**public String getParent()** Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
3	**public File getParentFile()** Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
4	**public String getPath()** Converts this abstract pathname into a pathname string.

```java
import java.io.File;

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

    File f = null;
    String[] strs = {"test1.txt", "test2.txt"};
    try{
      // for each string in string array
      for(String s:strs )
      {
        // create new file
        f= new File(s);

        // true if the file is executable
        boolean bool = f.canExecute();

        // find the absolute path
        String a = f.getAbsolutePath();

        // prints absolute path
        System.out.print(a);

        // prints
        System.out.println(" is executable: "+ bool);
      }
    }catch(Exception e){
      // if any I/O error occurs
      e.printStackTrace();
    }
```

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.

```
import java.io.*;

public class FileRead{

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

  File file = new File("Hello1.txt");
  // creates the file
  file.createNewFile();

  // creates a FileWriter Object
  FileWriter writer = new FileWriter(file);

  // Writes the content to the file
  writer.write("This\n is\n an\n example\n");
  writer.flush();
  writer.close();

  //Creates a FileReader Object
  FileReader fr = new FileReader(file);
  char [] a = new char[50];
  fr.read(a); // reads the content to the array
  for(char c : a)
    System.out.print(c); //prints the characters one by one
    fr.close();
 }
}
```

FileWriterClass:

This class inherits from the OutputStreamWriter class. The class is used for writing streams of characters.

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

   File file = new File("outputstream.txt");
   // creates the file
   file.createNewFile();

   // creates a FileWriter Object
   FileWriter writer = new FileWriter(file);
   // Writes the content to the file
   writer.write("This\n is\n an\n example\n");
   writer.flush();
   writer.close();

   //Creates a FileReader Object
   FileReader fr = new FileReader(file);
   char [] a = new char[50];
   fr.read(a); // reads the content to the array
   for(char c : a)
     System.out.print(c); //prints the characters one by one
     fr.close();
 }
}
```

Directories :

Creating Directories:

- **The mkdir( ) method creates a directory, returning true on success and false on failure.**
- **The mkdirs() method creates both a directory and all the parents of the directory.**

```
import java.io.File;

public class CreateDir {
  public static void main(String args[]) {
  String dirname = "/tmp/user/java/bin";
  File d = new File(dirname);
    // Create directory now.
    d.mkdirs();
  }
}
```

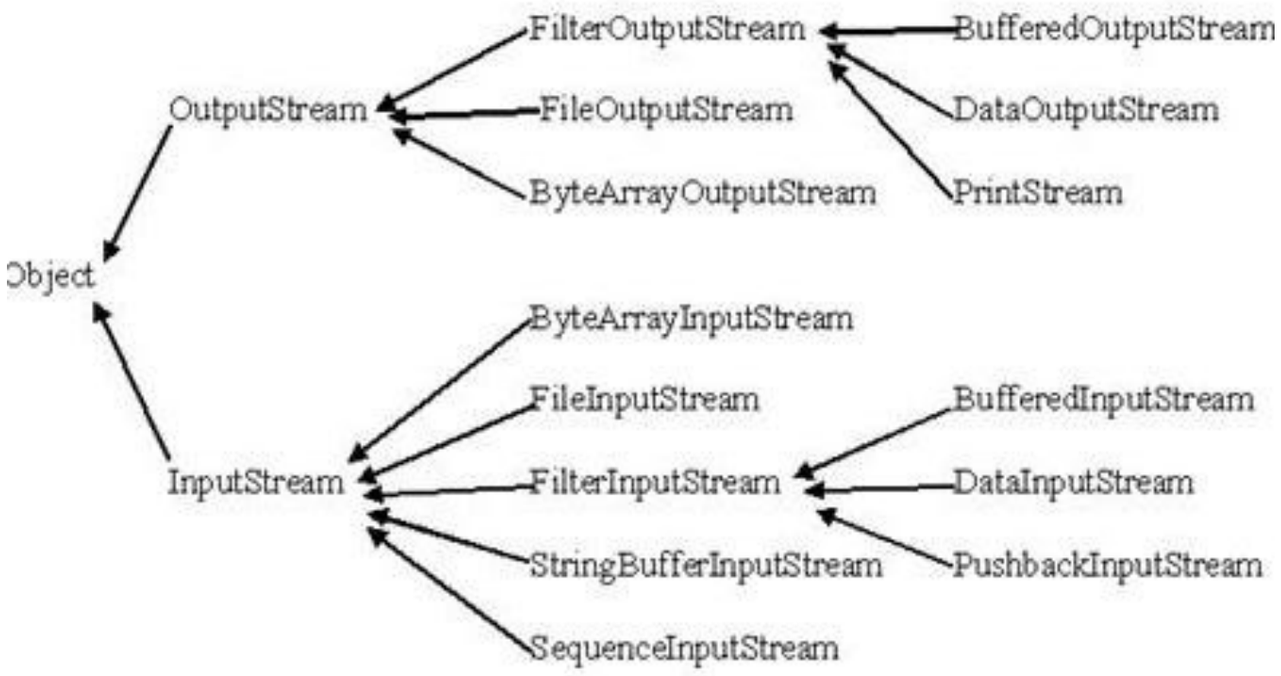
Listing Directories:

```
   File file = null; String[] paths;
   try{
     // create new file object
     file = new File("/tmp");
     // array of files and directory
     paths = file.list();
     // for each name in the path array
     for(String path:paths)
     {
       // prints filename and directory name
       System.out.println(path);
     }
   }catch(Exception e){
     // if any error occurs
     e.printStackTrace();
    }
  }
}
```

Standard Stream:

- 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

Reading and writing Files



FileInputStream

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

SN	Methods with Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

ByteArrayInputStream

```
import java.io.*;

public class ByteStreamTest {
  public static void main(String args[])throws IOException {
  ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);
  while( bOutput.size()!= 10 ) {
     // Gets the inputs from the user
     bOutput.write(System.in.read());
    }

    byte b [] = bOutput.toByteArray();
    System.out.println("Print the content");
    for(int x= 0 ; x < b.length; x++) {
     // printing the characters
     System.out.print((char)b[x]  + "   ");
    }
    System.out.println(" "); int c;

    ByteArrayInputStream bInput = new ByteArrayInputStream(b);

    System.out.println("Converting characters to Upper case " );
    for(int y = 0 ; y < 1; y++ ) {
     while(( c= bInput.read())!= -1)
       { System.out.println(Character.toUpperCase((char)c));
     }
     bInput.reset();
    }
  }
}
```

DataInputStream

The DataInputStream is used in the context of DataOutputStream and can be used to read primitives.

```
import java.io.*;

public class Test{
  public static void main(String args[])throws IOException{

    DataInputStream d = new DataInputStream(new
                FileInputStream("test.txt"));

    DataOutputStream out = new DataOutputStream(new
                FileOutputStream("test1.txt"));

    String count;
    while((count = d.readLine()) != null){
    String u = count.toUpperCase();
    System.out.println(u); out.writeBytes(u + "   ,");
    }
    d.close();
    out.close();
  }
}
```

FileOutputStream:

FileOutputStream is used to create a file and write data into it.

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

ByteArrayOutputStream - The ByteArrayOutputStream class stream creates a buffer in memory and all the data sent to the stream is stored in the buffer.

```java
import java.io.*;

public class ByteStreamTest {
  public static void main(String args[])throws IOException {
  ByteArrayOutputStream bOutput = new ByteArrayOutputStream(12);
  while( bOutput.size()!= 10 ) {
     // Gets the inputs from the user
     bOutput.write(System.in.read());
    }

    byte b [] = bOutput.toByteArray();
    System.out.println("Print the content");
    for(int x= 0 ; x < b.length; x++) {
     //printing the characters
     System.out.print((char)b[x]  + "   ");
    }
System.out.println(" "); int c;

ByteArrayOutputStream bInput = new ByteArrayOutputStream(b);

System.out.println("Converting characters to Upper case " );
for(int y = 0 ; y < 1; y++ ) {
  while(( c= bInput.read())!= -1) {
  System.out.println(Character.toUpperCase((char)c));
  }
  bInput.reset();
}
```

DataOutputStream:

The DataOutputStream stream let you write the primitives to an output source.

```
import java.io.*;

public class Test{
  public static void main(String args[])throws IOException{

    DataInputStream d = new DataInputStream(new
                FileInputStream("test.txt"));

    DataOutputStream out = new DataOutputStream(new
                FileOutputStream("test1.txt"));

    String count;
    while((count = d.readLine()) != null){
    String u = count.toUpperCase();
    System.out.println(u);
    out.writeBytes(u + " ,");
    }
    d.close();
    out.close();
  }
}
```