**UNIT – 2**

## Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

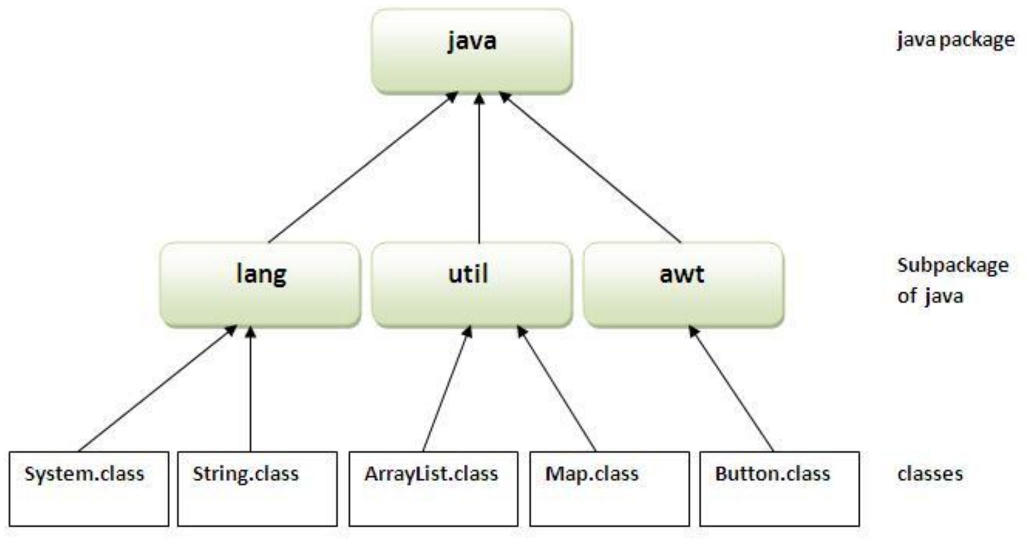
Here, we will have the detailed learning of creating and using user-defined packages.

### Advantages of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }  }
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

**To Compile:** javac -d . Simple.java

**To Run:** java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

**1) Using packagename.***

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

**2) Using packagename.classname**

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

**3) Using fully qualified name**

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output:Hello

***Note: If you import a package, subpackages will not be imported.***

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

*Note: Sequence of the program must be package then import then class.*

## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

*The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.*

### Example of Subpackage

```
package com.javatpoint.core;
```

```
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

**To Compile:** javac -d . Simple.java

**To Run:** java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

## How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

### To Compile:

**e:\sources> javac -d c:\classes Simple.java**

### To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class f

**e:\sources> set classpath=c:\classes;.;**

**e:\sources> java mypack.Simple**

## Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

**e:\sources> java -classpath c:\classes mypack.Simple**

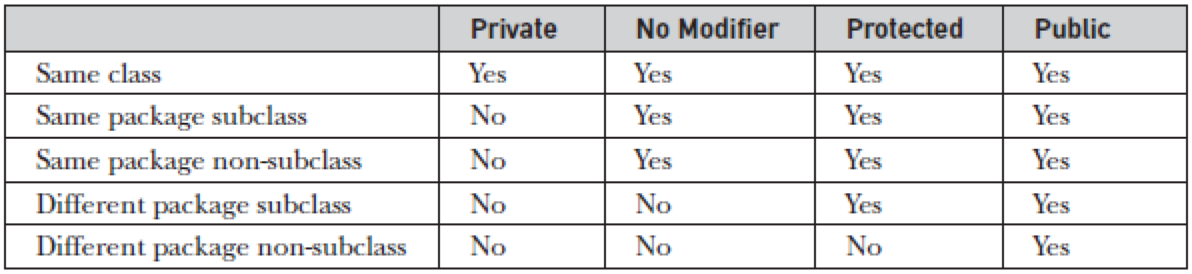
```
Output:Welcome to package
```

## Access Protection

• Subclasses in the same package
• Non-subclasses in the same package
• Subclasses in different packages
• Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.



| | Private | No Modifier | Protected | Public |
|---|---|---|---|---|
| Same class | Yes | Yes | Yes | Yes |
| Same package subclass | No | Yes | Yes | Yes |
| Same package non-subclass | No | Yes | Yes | Yes |
| Different package subclass | No | No | Yes | Yes |
| Different package non-subclass | No | No | No | Yes |

**Table 9-1** Class Member Access

Table 9-1 applies only to members of classes. A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

## Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

### Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.

- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

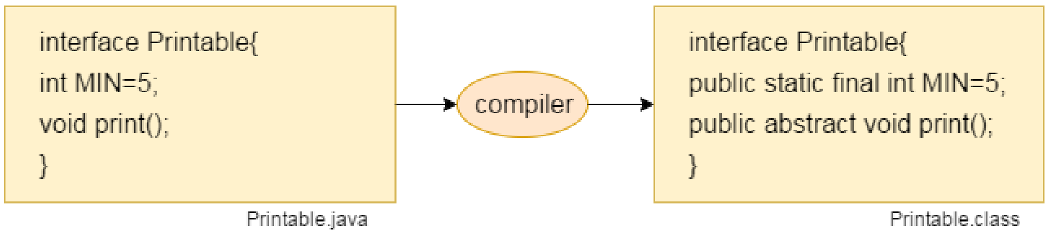
**Java 8 Interface Improvement**

Since Java 8, interface can have default and static methods which is discussed later.

Internal addition by compiler

> ***The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.***

In other words, Interface fields are public, static and final by default, and methods are public and abstract.

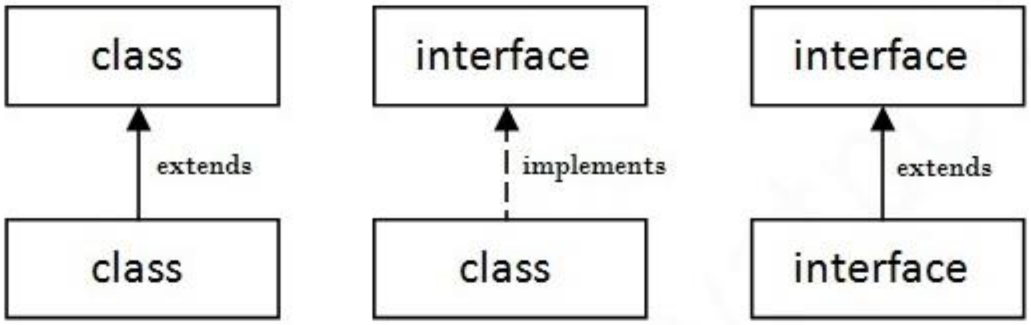


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

*Understanding relationship between classes and interfaces*

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



**Java Interface Example**

In this example, Printable interface has only one method, its implementation is provided in the A class.

```
interface printable{
void print();
}
class A6 implements printable{
```

```
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

*File: TestInterface1.java*

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Output:

```
drawing circle
```

**Java Interface Example: Bank**

Let's see another example of java interface which provides the implementation of Bank interface.

*File: TestInterface2.java*

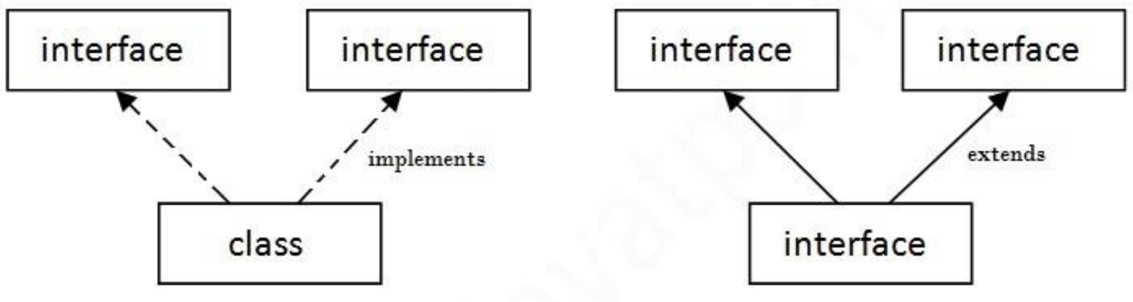
```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



**Multiple Inheritance in Java**

```
interface Printable{
void print();
}
```

```
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

```
Output:Hello
     Welcome
```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

**Interface inheritance**

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

## Java Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitely.

Syntax of nested interface which is declared within the interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

Syntax of nested interface which is declared within the class

```
class class_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

**Example of nested interface which is declared within the interface**

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

Output:hello nested interface

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection frameword, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below:.

```
public static interface Showable$Message
{
  public abstract void msg();
}
```

**Example of nested interface which is declared within the class**

Let's see how can we define an interface inside the class and how can we access it.

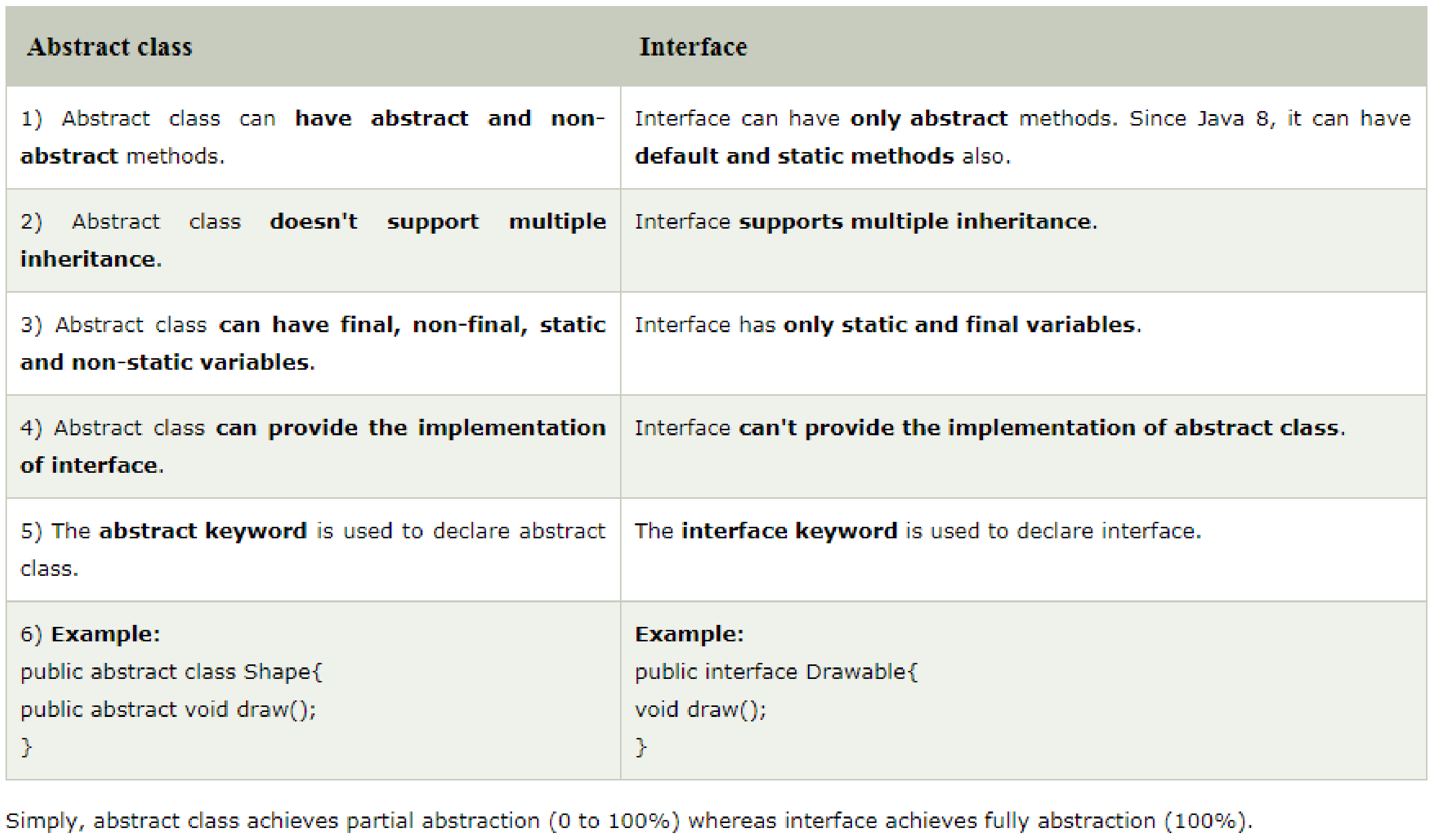
```
class A{
  interface Message{
   void msg();
  }
}

class TestNestedInterface2 implements A.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  A.Message message=new TestNestedInterface2();//upcasting here
  message.msg();
 }
}
```

```
Output:hello nested interface
```

**Difference between abstract class and interface**

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.



| Abstract class | Interface |
|---|---|
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance.** | Interface **supports multiple inheritance.** |
| 3) Abstract class **can have final, non-final, static and non-static variables.** | Interface has **only static and final variables.** |
| 4) Abstract class **can provide the implementation of interface.** | Interface **can't provide the implementation of abstract class.** |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) **Example:**<br>public abstract class Shape{<br>public abstract void draw();<br>} | **Example:**<br>public interface Drawable{<br>void draw();<br>} |

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

# Stream based I/O(java.io)

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

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

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

**Stream:**

A stream is a sequence of data.In Java 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 console.

**1) System.out:** standard output stream

**2) System.in:** standard input stream

**3) System.err:** standard error stream

Let's see the code to print **output and error** message to the console.

```
System.out.println("simple message");
System.err.println("error message");
```

Let's see the code to get **input** from console.

```
int i=System.in.read();//returns ASCII code of 1st character
System.out.println((char)i);//will print the character
```

**Byte Streams**

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

**Character Streams**

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

**OutputStream vs InputStream**

The explanation of OutputStream and InputStream classes are given below:

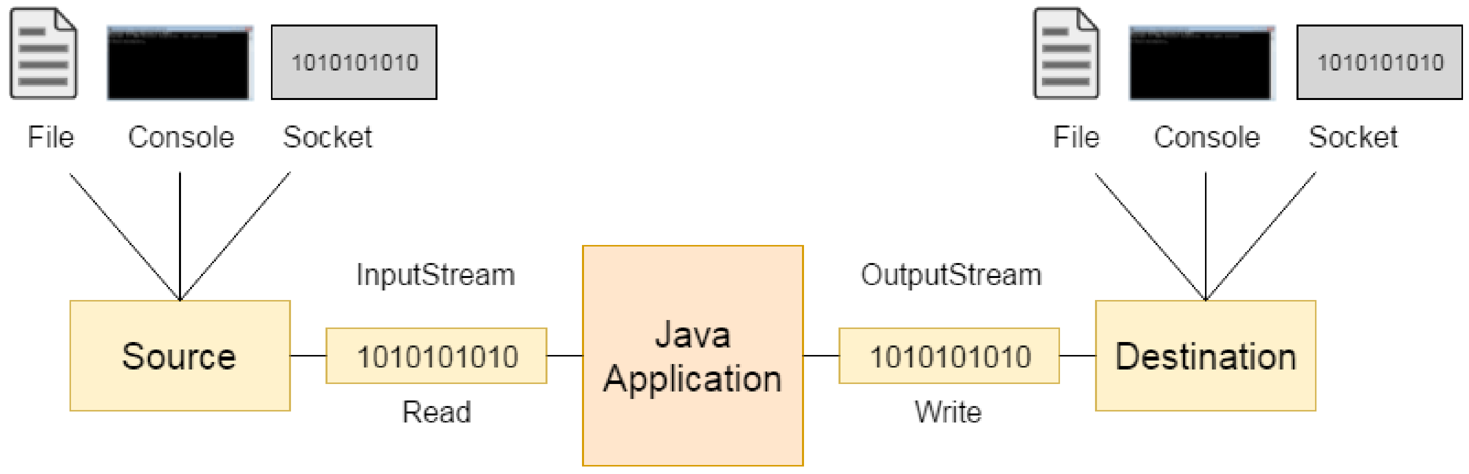
**OutputStream**

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.

Let's understand working of Java OutputStream and InputStream by the figure given below.



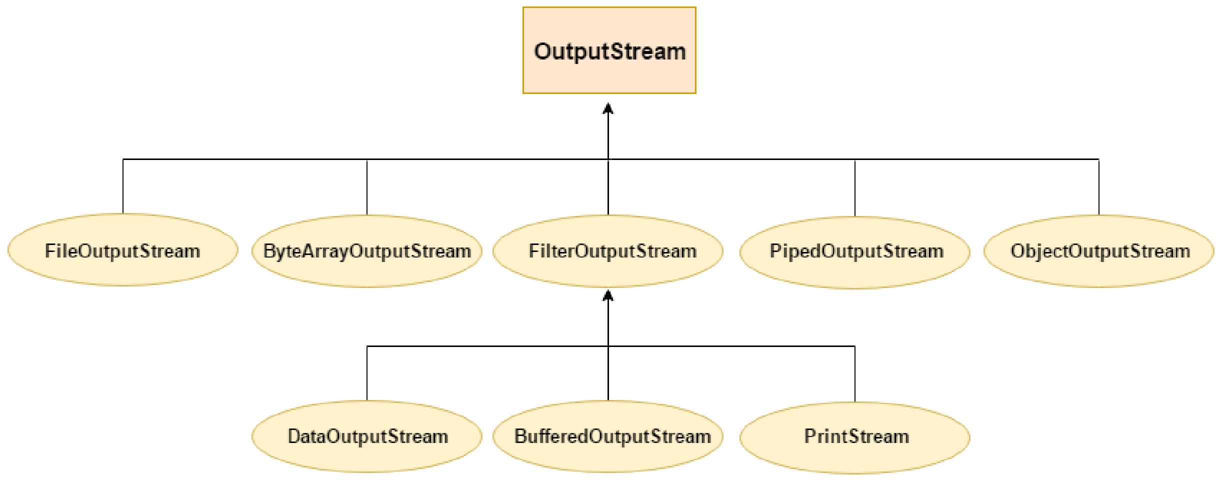
**OutputStream class**

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

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**



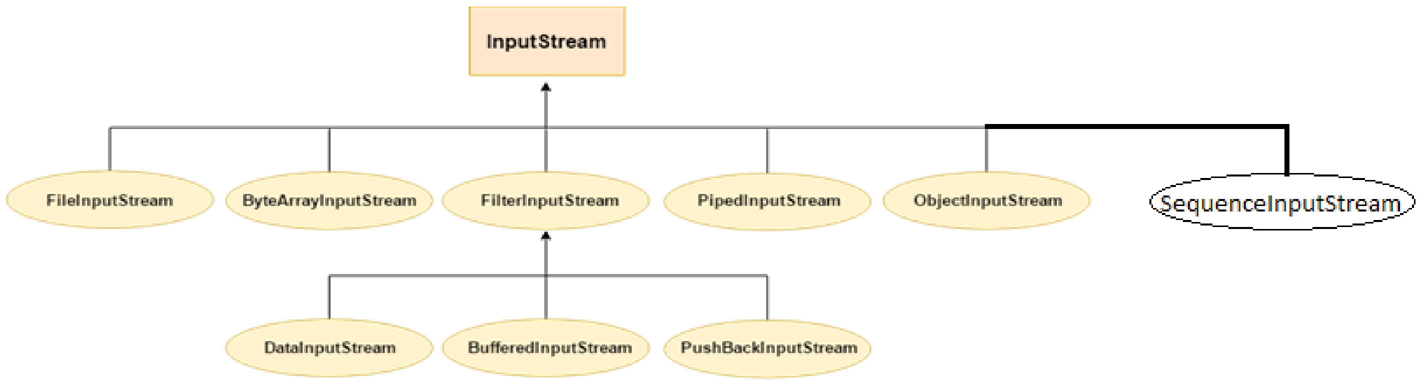
**InputStream class**

InputStream class is an abstract class. It is the super class 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 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**



## Java FileOutputStream Class

Java 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 FileOutStream.

FileOutputStream class declaration

Let's see the declaration for Java.io.FileOutputStream class:

```
public class FileOutputStream extends OutputStream
```

FileOutputStream class methods

| Method | Description |
|---|---|
| protected void finalize() | It is sued to clean up the connection with the file output stream. |
| void write(byte[] ary) | It is used to write **ary.length** bytes from the byte array to the file output stream. |
| void write(byte[] ary, int off, int len) | It is used to write **len** bytes from the byte array starting at offset **off** to the file output stream. |
| void write(int b) | It is used to write the specified byte to the file output stream. |
| FileChannel getChannel() | It is used to return the file channel object associated with the file output stream. |
| FileDescriptor getFD() | It is used to return the file descriptor associated with the stream. |
| void close() | It is used to closes the file output stream. |

**Java FileOutputStream Example 1: write byte**

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
   public static void main(String args[]){
        try{
          FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
          fout.write(65);
          fout.close();
          System.out.println("success...");
         }catch(Exception e){System.out.println(e);}
     }
}
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

```
A
```

**Java FileOutputStream example 2: write string**

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
        try{
          FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
          String s="Welcome to javaTpoint.";
          byte b[]=s.getBytes();//converting string into byte array
          fout.write(b);
          fout.close();
          System.out.println("success...");
         }catch(Exception e){System.out.println(e);}
      }
}
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **Welcome to javaTpoint.**

testout.txt

```
Welcome to javaTpoint.
```

## Java FileInputStream Class

Java 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. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

Java FileInputStream class declaration

Let's see the declaration for java.io.FileInputStream class:

1. **public class** FileInputStream **extends** InputStream

Java FileInputStream class methods

| Method | Description |
|---|---|
| int available() | It is used to return the estimated number of bytes that can be read from the input stream. |
| int read() | It is used to read the byte of data from the input stream. |
| int read(byte[] b) | It is used to read up to **b.length** bytes of data from the input stream. |
| int read(byte[] b, int off, int len) | It is used to read up to **len** bytes of data from the input stream. |
| long skip(long x) | It is used to skip over and discards x bytes of data from the input stream. |
| FileChannel getChannel() | It is used to return the unique FileChannel object associated with the file input stream. |
| FileDescriptor getFD() | It is used to return the FileDescriptor object. |
| protected void finalize() | It is used to ensure that the close method is call when there is no more reference to the file input stream. |
| void close() | It is used to closes the stream. |

**Java FileInputStream example 1: read single character**

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=fin.read();
          System.out.print((char)i);

          fin.close();
```

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

**Note:** Before running the code, a text file named as **"testout.txt"** is required to be created. In this file, we are having following content:

```
Welcome to javatpoint.
```

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output:

```
W
```

**Java FileInputStream example 2: read all characters**

```
import java.io.FileInputStream;
public class DataStreamExample {
     public static void main(String args[]){
          try{
            FileInputStream fin=new FileInputStream("D:\\testout.txt");
            int i=0;
            while((i=fin.read())!=-1){
             System.out.print((char)i);
            }
            fin.close();
          }catch(Exception e){System.out.println(e);}
         }
        }
```

Output:

```
Welcome to javaTpoint
```

**Java BufferedOutputStream Class**

Java 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.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:

1. OutputStream os= **new** BufferedOutputStream(**new** FileOutputStream("D:\\IO Package\\testout.txt"));

Java BufferedOutputStream class declaration

Let's see the declaration for Java.io.BufferedOutputStream class:

1. **public class** BufferedOutputStream **extends** FilterOutputStream

Java BufferedOutputStream class constructors

| Constructor | Description |
|---|---|
| BufferedOutputStream(OutputStream os) | It creates the new buffered output stream which is used for writing the data to the specified output stream. |
| BufferedOutputStream(OutputStream os, int size) | It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size. |

Java BufferedOutputStream class methods

| Method | Description |
|---|---|
| void write(int b) | It writes the specified byte to the buffered output stream. |
| void write(byte[] b, int off, int len) | It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset |
| void flush() | It flushes the buffered output stream. |

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("D:\\testout.txt");
      BufferedOutputStream bout=new BufferedOutputStream(fout);
      String s="Welcome to javaTpoint.";
      byte b[]=s.getBytes();
      bout.write(b);
      bout.flush();
      bout.close();
      fout.close();
      System.out.println("success");
    }
    }
```

Output:

```
Success
```

testout.txt

```
Welcome to javaTpoint.
```

**Java BufferedInputStream Class**

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

The important points about BufferedInputStream are:

- When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
- When a BufferedInputStream is created, an internal buffer array is created.

Java BufferedInputStream class declaration

Let's see the declaration for Java.io.BufferedInputStream class:

1. **public class** BufferedInputStream **extends** FilterInputStream

Java BufferedInputStream class constructors

| **Constructor** | **Description** |
|---|---|

| | |
|---|---|
| BufferedInputStream(InputStream IS) | It creates the BufferedInputStream and saves it argument, the input stream IS, for later use. |
| BufferedInputStream(InputStream IS, int size) | It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use. |

Java BufferedInputStream class methods

| **Method** | **Description** |
|---|---|
| int available() | It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream. |
| int read() | It read the next byte of data from the input stream. |
| int read(byte[] b, int off, int ln) | It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset. |
| void close() | It closes the input stream and releases any of the system resources associated with the stream. |
| void reset() | It repositions the stream at a position the mark method was last called on this input stream. |
| void mark(int readlimit) | It sees the general contract of the mark method for the input stream. |
| long skip(long x) | It skips over and discards x bytes of data from the input stream. |
| boolean markSupported() | It tests for the input stream to support the mark and reset methods. |

Example of Java BufferedInputStream

Let's see the simple example to read data of file using BufferedInputStream:

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

Here, we are assuming that you have following data in **"testout.txt"** file:

```
javaTpoint
```

Output:

```
javaTpoint
```

**Java SequenceInputStream Class**

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

Java SequenceInputStream Class declaration

Let's see the declaration for Java.io.SequenceInputStream class:

1. **public class** SequenceInputStream **extends** InputStream

Constructors of SequenceInputStream class

| **Constructor** | **Description** |
|---|---|
| SequenceInputStream(InputStream s1, InputStream s2) | creates a new input stream by reading the data of two input stream in order, first s1 and then s2. |

|  |  |
|---|---|
| SequenceInputStream(Enumeration e) | creates a new input stream by reading the data of an enumeration whose type is InputStream. |

Methods of SequenceInputStream class

| **Method** | **Description** |
|---|---|
| int read() | It is used to read the next byte of data from the input stream. |
| int read(byte[] ary, int off, int len) | It is used to read len bytes of data from the input stream into the array of bytes. |
| int available() | It is used to return the maximum number of byte that can be read from an input stream. |
| void close() | It is used to close the input stream. |

Java SequenceInputStream Example

In this example, we are printing the data of two files testin.txt and testout.txt.

```
import java.io.*;
class InputStreamExample {
  public static void main(String args[])throws Exception{
   FileInputStream input1=new FileInputStream("D:\\testin.txt");
   FileInputStream input2=new FileInputStream("D:\\testout.txt");
   SequenceInputStream inst=new SequenceInputStream(input1, input2);
   int j;
   while((j=inst.read())!=-1){
    System.out.print((char)j);
   }
   inst.close();
   input1.close();
   input2.close();
  }
}
```

Here, we are assuming that you have two files: testin.txt and testout.txt which have following information:

testin.txt:

```
Welcome to Java IO Programming.
```

testout.txt:

```
It is the example of Java SequenceInputStream class.
```

After executing the program, you will get following output:

Output:

```
Welcome to Java IO Programming. It is the example of Java SequenceInputStream class.
```

Example that reads the data from two files and writes into another file

In this example, we are writing the data of two files **testin1.txt** and **testin2.txt** into another file named **testout.txt.**

```
import java.io.*;
class Input1{
  public static void main(String args[])throws Exception{
   FileInputStream fin1=new FileInputStream("D:\\testin1.txt");
   FileInputStream fin2=new FileInputStream("D:\\testin2.txt");
   FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
   SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
   int i;
   while((i=sis.read())!=-1)
   {
     fout.write(i);
   }
   sis.close();
   fout.close();
   fin1.close();
   fin2.close();
   System.out.println("Success..");
  }
}
```

Output:

```
Succeess...
```

testout.txt:

1. Welcome to Java IO Programming. It is the example of Java SequenceInputStream **class**.

SequenceInputStream example that reads data using enumeration

If we need to read the data from more than two files, we need to use Enumeration. Enumeration object can be obtained by calling elements() method of the Vector class. Let's see the simple example where we are reading the data from 4 files: a.txt, b.txt, c.txt and d.txt.

```
import java.io.*;
import java.util.*;
class Input2{
public static void main(String args[])throws IOException{
//creating the FileInputStream objects for all the files
FileInputStream fin=new FileInputStream("D:\\a.txt");
FileInputStream fin2=new FileInputStream("D:\\b.txt");
FileInputStream fin3=new FileInputStream("D:\\c.txt");
FileInputStream fin4=new FileInputStream("D:\\d.txt");
//creating Vector object to all the stream
Vector v=new Vector();
v.add(fin);
v.add(fin2);
v.add(fin3);
v.add(fin4);
//creating enumeration object by calling the elements method
Enumeration e=v.elements();
//passing the enumeration object in the constructor
SequenceInputStream bin=new SequenceInputStream(e);
int i=0;
while((i=bin.read())!=-1){
System.out.print((char)i);
}
bin.close();
fin.close();
fin2.close();
}
}
```

The a.txt, b.txt, c.txt and d.txt have following information:

a.txt:

```
Welcome
```

b.txt:

```
to
```

c.txt:

```
java
```

d.txt:

```
programming
```

Output:

```
Welcometojavaprogramming
```

**Java ByteArrayOutputStream Class**

Java 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.

Java ByteArrayOutputStream class declaration

Let's see the declaration for Java.io.ByteArrayOutputStream class:

1. **public class** ByteArrayOutputStream **extends** OutputStream

Java ByteArrayOutputStream class constructors

| **Constructor** | **Description** |
|---|---|
| ByteArrayOutputStream() | Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary. |
| ByteArrayOutputStream(int size) | Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes. |

Java ByteArrayOutputStream class methods

| **Method** | **Description** |
|---|---|
| int size() | It is used to returns the current size of a buffer. |
| byte[] toByteArray() | It is used to create a newly allocated byte array. |
| String toString() | It is used for converting the content into a string decoding bytes using a platform default character set. |
| String toString(String charsetName) | It is used for converting the content into a string decoding bytes using a specified charsetName. |
| void write(int b) | It is used for writing the byte specified to the byte array output stream. |
| void write(byte[] b, int off, int len | It is used for writing **len** bytes from specified byte array starting from the offset **off** to the byte array output stream. |
| void writeTo(OutputStream out) | It is used for writing the complete content of a byte array output stream to the specified output stream. |
| void reset() | It is used to reset the count field of a byte array output stream to zero value. |
| void close() | It is used to close the ByteArrayOutputStream. |

Example of Java ByteArrayOutputStream

Let's see a simple example of java ByteArrayOutputStream class to write common data into 2 files: f1.txt and f2.txt.

```
import java.io.*;
public class DataStreamExample {
public static void main(String args[])throws Exception{
    FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
    FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");
```

```
        ByteArrayOutputStream bout=new ByteArrayOutputStream();
        bout.write(65);
        bout.writeTo(fout1);
        bout.writeTo(fout2);

        bout.flush();
        bout.close();//has no effect
        System.out.println("Success...");
       }
      }
```

Output:

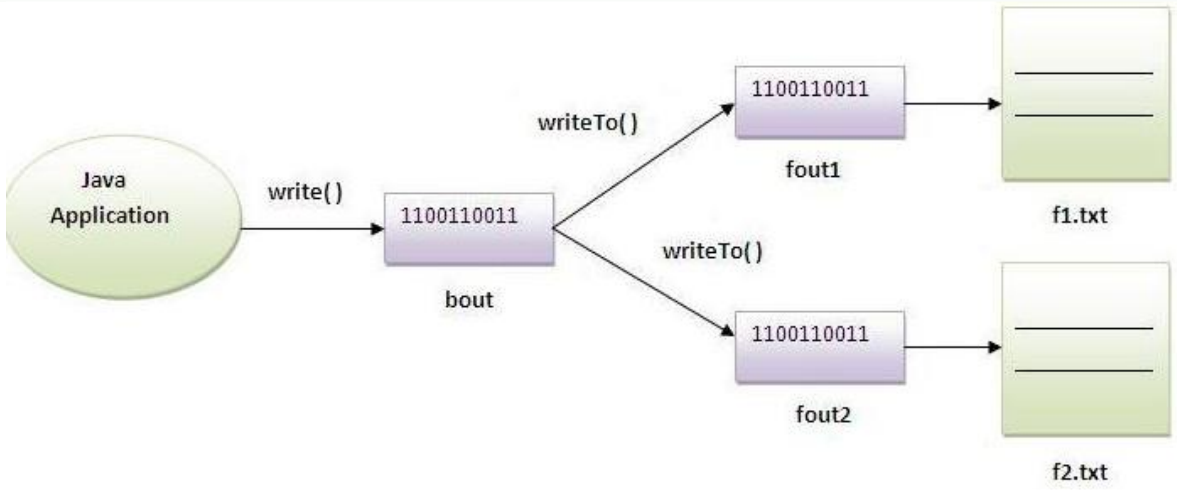
```
Success...
```

f1.txt:

```
A
```

f2.txt:

```
A
```



**Java ByteArrayInputStream Class**

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.

Java 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.

Java ByteArrayInputStream class declaration

Let's see the declaration for Java.io.ByteArrayInputStream class:

1. **public class** ByteArrayInputStream **extends** InputStream

Java ByteArrayInputStream class constructors

| **Constructor** | **Description** |
|---|---|
| ByteArrayInputStream(byte[] ary) | Creates a new byte array input stream which uses **ary** as its buffer arr |
| ByteArrayInputStream(byte[] ary, int offset, int len) | Creates a new byte array input stream which uses **ary** as its buffer a read up to specified **len** bytes of data from an array. |

Java ByteArrayInputStream class methods

| **Methods** | **Description** |
|---|---|
| int available() | It is used to return the number of remaining bytes that can be read from the input stream. |
| int read() | It is used to read the next byte of data from the input stream. |
| int read(byte[] ary, int off, int len) | It is used to read up to len bytes of data from an array of bytes in the input stream. |
| boolean markSupported() | It is used to test the input stream for mark and reset method. |
| long skip(long x) | It is used to skip the x bytes of input from the input stream. |
| void mark(int readAheadLimit) | It is used to set the current marked position in the stream. |
| void reset() | It is used to reset the buffer of a byte array. |
| void close() | It is used for closing a ByteArrayInputStream. |

Example of Java ByteArrayInputStream

Let's see a simple example of java ByteArrayInputStream class to read byte array as input stream.

```
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);
    }
  }
}
```

Output:

```
ASCII value of Character is:35; Special character is: #
ASCII value of Character is:36; Special character is: $
ASCII value of Character is:37; Special character is: %
ASCII value of Character is:38; Special character is: &
```

**Java DataOutputStream Class**

Java DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Java DataOutputStream class declaration

Let's see the declaration for java.io.DataOutputStream class:

1. **public class** DataOutputStream **extends** FilterOutputStream **implements** DataOutput

Java DataOutputStream class methods

| **Method** | **Description** |
|---|---|

| | |
|---|---|
| int size() | It is used to return the number of bytes written to the data output stream. |
| void write(int b) | It is used to write the specified byte to the underlying output stream. |
| void write(byte[] b, int off, int len) | It is used to write len bytes of data to the output stream. |
| void writeBoolean(boolean v) | It is used to write Boolean to the output stream as a 1-byte value. |
| void writeChar(int v) | It is used to write char to the output stream as a 2-byte value. |
| void writeChars(String s) | It is used to write string to the output stream as a sequence of characters. |
| void writeByte(int v) | It is used to write a byte to the output stream as a 1-byte value. |
| void writeBytes(String s) | It is used to write string to the output stream as a sequence of bytes. |
| void writeInt(int v) | It is used to write an int to the output stream |
| void writeShort(int v) | It is used to write a short to the output stream. |
| void writeShort(int v) | It is used to write a short to the output stream. |
| void writeLong(long v) | It is used to write a long to the output stream. |
| void writeUTF(String str) | It is used to write a string to the output stream using UTF-8 encoding in portable manner. |
| void flush() | It is used to flushes the data output stream. |

Example of DataOutputStream class

In this example, we are writing the data to a text file **testout.txt** using DataOutputStream class.

```
import java.io.*;
public class OutputExample {
    public static void main(String[] args) throws IOException {
        FileOutputStream file = new FileOutputStream(D:\\testout.txt);
        DataOutputStream data = new DataOutputStream(file);
        data.writeInt(65);
        data.flush();
        data.close();
        System.out.println("Succcess...");
    }
}
```

Output:

```
Succcess...
```

testout.txt:

```
A
```

Java DataInputStream Class

Java DataInputStream class allows an application to read primitive data from the input stream in a machine-independent way.

Java application generally uses the data output stream to write data that can later be read by a data input stream.

Java DataInputStream class declaration

Let's see the declaration for java.io.DataInputStream class:

1. **public class** DataInputStream **extends** FilterInputStream **implements** DataInput

Java DataInputStream class Methods

| **Method** | **Description** |
|---|---|
| int read(byte[] b) | It is used to read the number of bytes from the input stream. |
| int read(byte[] b, int off, int len) | It is used to read **len** bytes of data from the input stream. |

| | |
|---|---|
| int readInt() | It is used to read input bytes and return an int value. |
| byte readByte() | It is used to read and return the one input byte. |
| char readChar() | It is used to read two input bytes and returns a char value. |
| double readDouble() | It is used to read eight input bytes and returns a double value. |
| boolean readBoolean() | It is used to read one input byte and return true if byte is non zero, false if byte is zero. |
| int skipBytes(int x) | It is used to skip over x bytes of data from the input stream. |
| String readUTF() | It is used to read a string that has been encoded using the UTF-8 format. |
| void readFully(byte[] b) | It is used to read bytes from the input stream and store them into the buffer array. |
| void readFully(byte[] b, int off, int len) | It is used to read **len** bytes from the input stream. |

Example of DataInputStream class

In this example, we are reading the data from the file testout.txt file.

```
import java.io.*;
public class DataStreamExample {
  public static void main(String[] args) throws IOException {
    InputStream input = new FileInputStream("D:\\testout.txt");
    DataInputStream inst = new DataInputStream(input);
    int count = input.available();
    byte[] ary = new byte[count];
    inst.read(ary);
    for (byte bt : ary) {
      char k = (char) bt;
      System.out.print(k+"-");
```

```
        }
      }
    }
```

Here, we are assuming that you have following data in **"testout.txt"** file:

```
JAVA
```

Output:

```
J-A-V-A
```

**Java FilterOutputStream Class**

Java FilterOutputStream class implements the OutputStream class. It provides different sub classes such as BufferedOutputStream and DataOutputStream to provide additional functionality. So it is less used individually.

Java FilterOutputStream class declaration

Let's see the declaration for java.io.FilterOutputStream class:

1. **public class** FilterOutputStream **extends** OutputStream

Java FilterOutputStream class Methods

| **Method** | **Description** |
|---|---|
| void write(int b) | It is used to write the specified byte to the output stream. |
| void write(byte[] ary) | It is used to write ary.length byte to the output stream. |
| void write(byte[] b, int off, int len) | It is used to write len bytes from the offset off to the output stream. |
| void flush() | It is used to flushes the output stream. |
| void close() | It is used to close the output stream. |

Example of FilterOutputStream class

```
import java.io.*;
public class FilterExample {
        public static void main(String[] args) throws IOException {
            File data = new File("D:\\testout.txt");
            FileOutputStream file = new FileOutputStream(data);
            FilterOutputStream filter = new FilterOutputStream(file);
            String s="Welcome to javaTpoint.";
            byte b[]=s.getBytes();
            filter.write(b);
            filter.flush();
            filter.close();
            file.close();
            System.out.println("Success...");
        }
    }
```

Output:

```
Success...
```

testout.txt

```
Welcome to javaTpoint.
```

**Java FilterInputStream Class**

Java FilterInputStream class implements the InputStream. It contains different sub classes as BufferedInputStream, DataInputStream for providing additional functionality. So it is less used individually.

Java FilterInputStream class declaration

Let's see the declaration for java.io.FilterInputStream class

```
1. public class FilterInputStream extends InputStream
```

Java FilterInputStream class Methods

| **Method** | **Description** |
|---|---|
| int available() | It is used to return an estimate number of bytes that can be read from the input stream. |
| int read() | It is used to read the next byte of data from the input stream. |

| | |
|---|---|
| int read(byte[] b) | It is used to read up to byte.length bytes of data from the input stream. |
| long skip(long n) | It is used to skip over and discards n bytes of data from the input stream. |
| boolean markSupported() | It is used to test if the input stream support mark and reset method. |
| void mark(int readlimit) | It is used to mark the current position in the input stream. |
| void reset() | It is used to reset the input stream. |
| void close() | It is used to close the input stream. |

Example of FilterInputStream class

```
import java.io.*;
public class FilterExample {
   public static void main(String[] args) throws IOException {
      File data = new File("D:\\testout.txt");
      FileInputStream  file = new FileInputStream(data);
      FilterInputStream filter = new BufferedInputStream(file);
      int k =0;
      while((k=filter.read())!=-1){
         System.out.print((char)k);
      }
      file.close();
      filter.close();
   }
}
```

Here, we are assuming that you have following data in **"testout.txt"** file:

```
Welcome to javatpoint
```

Output:

```
Welcome to javatpoint
```

## Character Stream Classes:

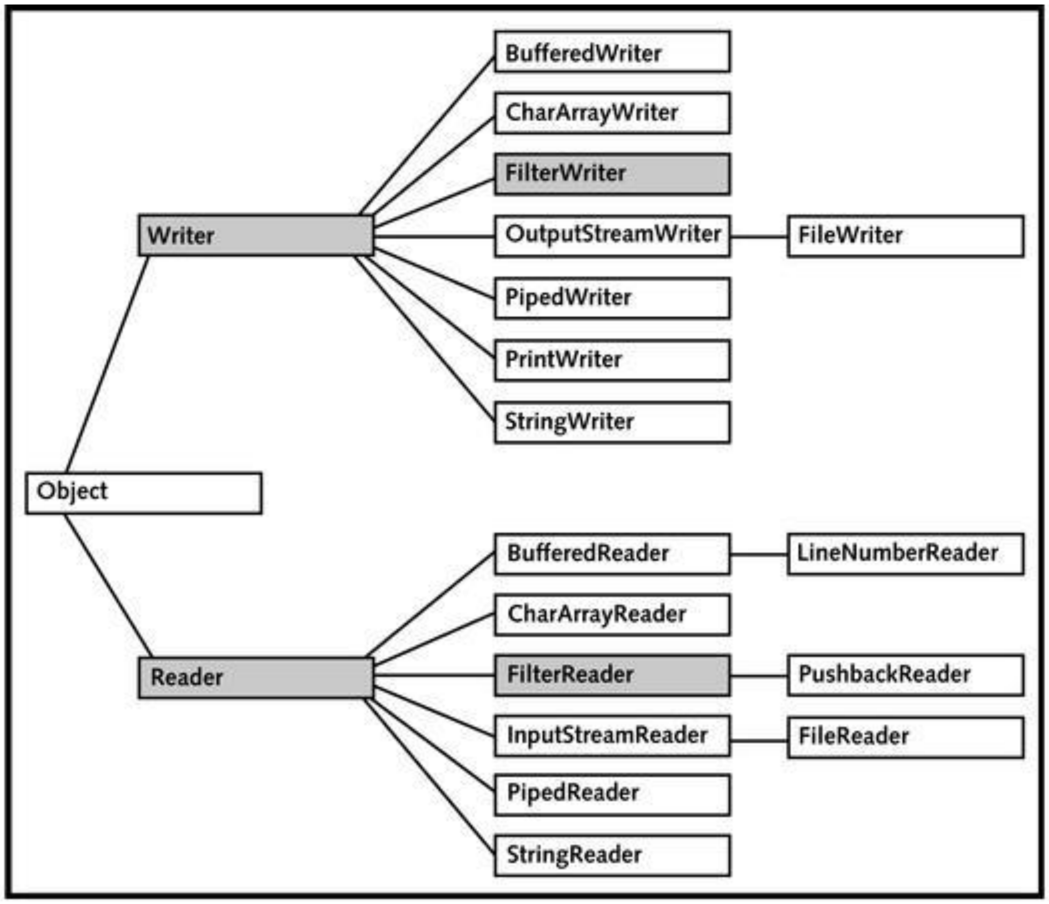


**Figure 8-2** Character stream classes

**Java Reader**

Java Reader is an abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods to provide higher efficiency, additional functionality, or both.

Some of the implementation class are BufferedReader, CharArrayReader, FilterReader, InputStreamReader, PipedReader, StringReader

Fields

| **Modifier and Type** | **Field** | **Description** |
|---|---|---|
| protected Object | lock | The object used to synchronize operations on this stream. |

**Constructor**

| Modifier | Constructor | Description |
|---|---|---|
| protected | Reader() | It creates a new character-stream reader whose critical sections will synchronize on the reader itself. |
| protected | Reader(Object lock) | It creates a new character-stream reader whose critical sections will synchronize on the given object. |

Methods

| Modifier and Type | Method | Description |
|---|---|---|
| abstract void | close() | It closes the stream and releases any system resources associated with it. |
| void | mark(int readAheadLimit) | It marks the present position in the stream. |
| boolean | markSupported() | It tells whether this stream supports the mark() operation. |
| int | read() | It reads a single character. |
| int | read(char[] cbuf) | It reads characters into an array. |
| abstract int | read(char[] cbuf, int off, int len) | It reads characters into a portion of an array. |
| int | read(CharBuffer target) | It attempts to read characters into the specified character buffer. |
| boolean | ready() | It tells whether this stream is ready to be read. |
| void | reset() | It resets the stream. |

|  |  |  |
|---|---|---|
| long | skip(long n) | It skips characters. |

Example

```
import java.io.*;
public class ReaderExample {
    public static void main(String[] args) {
        try {
            Reader reader = new FileReader("file.txt");
            int data = reader.read();
            while (data != -1) {
                System.out.print((char) data);
                data = reader.read();
            }
            reader.close();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

file.txt:

```
I love my country
```

Output:

```
I love my country
```

Java Writer

It is an abstract class for writing to character streams. The methods that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses will override some of the methods defined here to provide higher efficiency, functionality or both.

Fields

| **Modifier and Type** | **Field** | **Description** |
|---|---|---|
| protected Object | lock | The object used to synchronize operations on this stream. |

**Constructor**

| **Modifier** | **Constructor** | **Description** |
|---|---|---|
| protected | Writer() | It creates a new character-stream writer whose critical sections will synchronize on the writer itself. |
| protected | Writer(Object lock) | It creates a new character-stream writer whose critical sections will synchronize on the given object. |

Methods

| Modifier and Type | **Method** | **Description** |
|---|---|---|
| Writer | append(char c) | It appends the specified character to this writer. |
| Writer | append(CharSequence csq) | It appends the specified character sequence to this writer |
| Writer | append(CharSequence csq, int start, int end) | It appends a subsequence of the specified character sequence to this writer. |
| abstract void | close() | It closes the stream, flushing it first. |
| abstract void | flush() | It flushes the stream. |
| void | write(char[] cbuf) | It writes an array of characters. |
| abstract void | write(char[] cbuf, int off, int len) | It writes a portion of an array of characters. |
| void | write(int c) | It writes a single character. |
| void | write(String str) | It writes a string. |

|  |  |  |
|---|---|---|
| void | write(String str, int off, int len) | It writes a portion of a string. |

**Java Writer Example**

```
import java.io.*;
public class WriterExample {
    public static void main(String[] args) {
        try {
            Writer w = new FileWriter("output.txt");
            String content = "I love my country";
            w.write(content);
            w.close();
            System.out.println("Done");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

```
Done
```

output.txt:

```
I love my country
```

**Java FileReader Class**

Java 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.

Java FileReader class declaration

Let's see the declaration for Java.io.FileReader class:

1. **public class** FileReader **extends** InputStreamReader

Constructors of FileReader class

| Constructor | Description |
|---|---|
| FileReader(String file) | It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException. |
| FileReader(File file) | It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException. |

Methods of FileReader class

| Method | Description |
|---|---|
| int read() | It is used to return a character in ASCII form. It returns -1 at the end of file. |
| void close() | It is used to close the FileReader class. |

**Java FileReader Example**

In this example, we are reading the data from the text file **testout.txt** using Java FileReader class.

```
import java.io.FileReader;
public class FileReaderExample {
   public static void main(String args[])throws Exception{
         FileReader fr=new FileReader("D:\\testout.txt");
         int i;
         while((i=fr.read())!=-1)
         System.out.print((char)i);
         fr.close();
   }
}
```

Here, we are assuming that you have following data in "testout.txt" file:

```
Welcome to javaTpoint.
```

Output:

```
Welcome to javaTpoint.
```

**Java FileWriter Class**

Java 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.

Java FileWriter class declaration

Let's see the declaration for Java.io.FileWriter class:

1. **public class** FileWriter **extends** OutputStreamWriter

Constructors of FileWriter class

| **Constructor** | **Description** |
|---|---|
| FileWriter(String file) | Creates a new file. It gets file name in string. |
| FileWriter(File file) | Creates a new file. It gets file name in File object. |

Methods of FileWriter class

| **Method** | **Description** |
|---|---|
| void write(String text) | It is used to write the string into FileWriter. |
| void write(char c) | It is used to write the char into FileWriter. |
| void write(char[] c) | It is used to write char array into FileWriter. |
| void flush() | It is used to flushes the data of FileWriter. |
| void close() | It is used to close the FileWriter. |

Java FileWriter Example

In this example, we are writing the data in the file testout.txt using Java FileWriter class.

```
import java.io.FileWriter;
public class FileWriterExample {
    public static void main(String args[]){
         try{
           FileWriter fw=new FileWriter("D:\\testout.txt");
           fw.write("Welcome to javaTpoint.");
           fw.close();
          }catch(Exception e){System.out.println(e);}
          System.out.println("Success...");
     }
}
```

Output:

```
Success...
```

testout.txt:

```
Welcome to javaTpoint.
```

**Java BufferedWriter Class**

Java 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.

Class declaration

Let's see the declaration for Java.io.BufferedWriter class:

1. **public class** BufferedWriter **extends** Writer

Class constructors

| **Constructor** | **Description** |
|---|---|
| BufferedWriter(Writer wrt) | It is used to create a buffered character output stream that uses the default size for an output buffer. |

| | |
|---|---|
| BufferedWriter(Writer wrt, int size) | It is used to create a buffered character output stream that uses the specified size for an output buffer. |

Class methods

| Method | Description |
|---|---|
| void newLine() | It is used to add a new line by writing a line separator. |
| void write(int c) | It is used to write a single character. |
| void write(char[] cbuf, int off, int len) | It is used to write a portion of an array of characters. |
| void write(String s, int off, int len) | It is used to write a portion of a string. |
| void flush() | It is used to flushes the input stream. |
| void close() | It is used to closes the input stream |

Example of Java BufferedWriter

Let's see the simple example of writing the data to a text file **testout.txt** using Java BufferedWriter.

```
import java.io.*;
public class BufferedWriterExample {
public static void main(String[] args) throws Exception {
    FileWriter writer = new FileWriter("D:\\testout.txt");
    BufferedWriter buffer = new BufferedWriter(writer);
    buffer.write("Welcome to javaTpoint.");
    buffer.close();
    System.out.println("Success");
    }
}
```

Output:

```
success
```

testout.txt:

```
Welcome to javaTpoint.
```

**Java BufferedReader Class**

Java 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.

Java BufferedReader class declaration

Let's see the declaration for Java.io.BufferedReader class:

1. **public class** BufferedReader **extends** Reader

Java BufferedReader class constructors

| **Constructor** | **Description** |
|---|---|
| BufferedReader(Reader rd) | It is used to create a buffered character input stream that uses the default size for an input buffer. |
| BufferedReader(Reader rd, int size) | It is used to create a buffered character input stream that uses the specified size for an input buffer. |

Java BufferedReader class methods

| **Method** | **Description** |
|---|---|
| int read() | It is used for reading a single character. |
| int read(char[] cbuf, int off, int len) | It is used for reading characters into a portion of an array. |
| boolean markSupported() | It is used to test the input stream support for the mark and reset method. |
| String readLine() | It is used for reading a line of text. |

|  |  |
|---|---|
| boolean ready() | It is used to test whether the input stream is ready to be read. |
| long skip(long n) | It is used for skipping the characters. |
| void reset() | It repositions the stream at a position the mark method was last called on this input stream. |
| void mark(int readAheadLimit) | It is used for marking the present position in a stream. |
| void close() | It closes the input stream and releases any of the system resources associated with the stream. |

**Java BufferedReader Example**

In this example, we are reading the data from the text file **testout.txt** using Java BufferedReader class.

```
import java.io.*;
public class BufferedReaderExample {
   public static void main(String args[])throws Exception{
         FileReader fr=new FileReader("D:\\testout.txt");
         BufferedReader br=new BufferedReader(fr);

         int i;
         while((i=br.read())!=-1){
         System.out.print((char)i);
         }
         br.close();
         fr.close();
   }
}
```

Here, we are assuming that you have following data in "testout.txt" file:

```
Welcome to javaTpoint.
```

Output:

```
Welcome to javaTpoint.
```

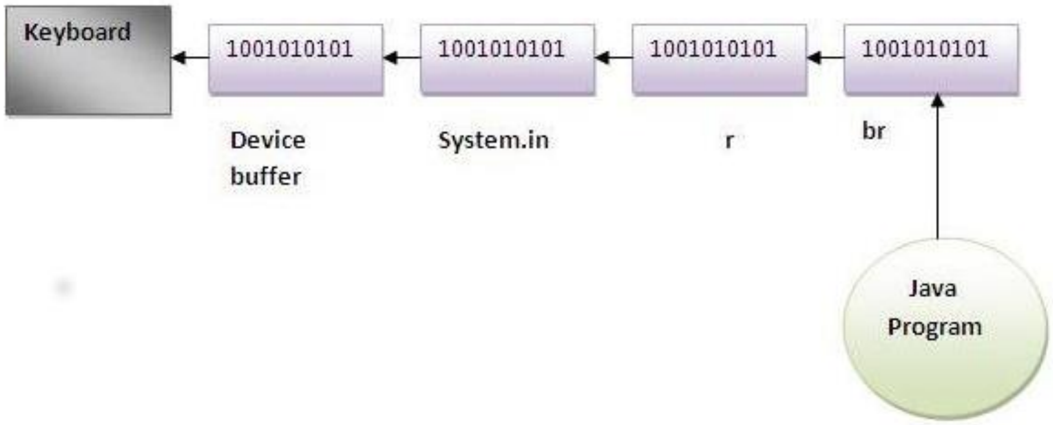
Reading data from console by InputStreamReader and BufferedReader

In this example, we are 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();
   System.out.println("Welcome "+name);
}
}
```

Output:

```
Enter your name
Nakul Jain
Welcome Nakul Jain
```



**Java 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.

Java CharArrayReader class declaration

Let's see the declaration for Java.io.CharArrayReader class:

1. **public class** CharArrayReader **extends** Reader

Java CharArrayReader class methods

| **Method** | **Description** |
|---|---|
| int read() | It is used to read a single character |
| int read(char[] b, int off, int len) | It is used to read characters into the portion of an array. |
| boolean ready() | It is used to tell whether the stream is ready to read. |
| boolean markSupported() | It is used to tell whether the stream supports mark() operation. |
| long skip(long n) | It is used to skip the character in the input stream. |
| void mark(int readAheadLimit) | It is used to mark the present position in the stream. |
| void reset() | It is used to reset the stream to a most recent mark. |
| void close() | It is used to closes the stream. |

Example of CharArrayReader Class:

Let's see the simple 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 = { 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't' };
    CharArrayReader reader = new CharArrayReader(ary);
    int k = 0;
    // Read until the end of a file
    while ((k = reader.read()) != -1) {
      char ch = (char) k;
      System.out.print(ch + " : ");
      System.out.println(k);
    }
  }
}
```

Output

```
j : 106
a : 97
v : 118
a : 97
t : 116
p : 112
o : 111
i : 105
n : 110
t : 116
```

**Java 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. Calling the close() method on this object has no effect.

Java CharArrayWriter class declaration

Let's see the declaration for Java.io.CharArrayWriter class:

1. **public class** CharArrayWriter **extends** Writer

Java CharArrayWriter class Methods

| Method | Description |
|---|---|
| int size() | It is used to return the current size of the buffer. |
| char[] toCharArray() | It is used to return the copy of an input data. |
| String toString() | It is used for converting an input data to a string. |
| CharArrayWriter append(char c) | It is used to append the specified character to the writer. |
| CharArrayWriter append(CharSequence csq) | It is used to append the specified character sequence to the writer. |

| | |
|---|---|
| CharArrayWriter append(CharSequence csq, int start, int end) | It is used to append the subsequence of a specified character to the writer. |
| void write(int c) | It is used to write a character to the buffer. |
| void write(char[] c, int off, int len) | It is used to write a character to the buffer. |
| void write(String str, int off, int len) | It is used to write a portion of string to the buffer. |
| void writeTo(Writer out) | It is used to write the content of buffer to different character stream. |
| void flush() | It is used to flush the stream. |
| void reset() | It is used to reset the buffer. |
| void close() | It is used to close the stream. |

Example of CharArrayWriter Class:

In this example, we are writing a common data to 4 files a.txt, b.txt, c.txt and d.txt.

```
package com.javatpoint;

import java.io.CharArrayWriter;
import java.io.FileWriter;
public class CharArrayWriterExample {
public static void main(String args[])throws Exception{
        CharArrayWriter out=new CharArrayWriter();
        out.write("Welcome to javaTpoint");
        FileWriter f1=new FileWriter("D:\\a.txt");
        FileWriter f2=new FileWriter("D:\\b.txt");
        FileWriter f3=new FileWriter("D:\\c.txt");
        FileWriter f4=new FileWriter("D:\\d.txt");
        out.writeTo(f1);
        out.writeTo(f2);
        out.writeTo(f3);
        out.writeTo(f4);
```

```
                  f1.close();
                  f2.close();
                  f3.close();
                  f4.close();
                  System.out.println("Success...");
                 }
                }
```

Output

```
Success...
```

After executing the program, you can see that all files have common data: Welcome to javaTpoint.

a.txt:

```
Welcome to javaTpoint
```

b.txt:

```
Welcome to javaTpoint
```

c.txt:

```
Welcome to javaTpoint
```

d.txt:

```
Welcome to javaTpoint
```

**Java PrintStream Class**

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Class declaration

Let's see the declaration for Java.io.PrintStream class:

1. **public class** PrintStream **extends** FilterOutputStream **implements** Closeable. Appendable

Methods of PrintStream class

| **Method** | **Description** |
|---|---|

| | |
|---|---|
| void print(boolean b) | It prints the specified boolean value. |
| void print(char c) | It prints the specified char value. |
| void print(char[] c) | It prints the specified character array values. |
| void print(int i) | It prints the specified int value. |
| void print(long l) | It prints the specified long value. |
| void print(float f) | It prints the specified float value. |
| void print(double d) | It prints the specified double value. |
| void print(String s) | It prints the specified string value. |
| void print(Object obj) | It prints the specified object value. |
| void println(boolean b) | It prints the specified boolean value and terminates the line. |
| void println(char c) | It prints the specified char value and terminates the line. |
| void println(char[] c) | It prints the specified character array values and terminates the line. |
| void println(int i) | It prints the specified int value and terminates the line. |
| void println(long l) | It prints the specified long value and terminates the line. |
| void println(float f) | It prints the specified float value and terminates the line. |
| void println(double d) | It prints the specified double value and terminates the line. |
| void println(String s) | It prints the specified string value and terminates the line. |

| | |
|---|---|
| void println(Object obj) | It prints the specified object value and terminates the line. |
| void println() | It terminates the line only. |
| void printf(Object format, Object... args) | It writes the formatted string to the current stream. |
| void printf(Locale l, Object format, Object... args) | It writes the formatted string to the current stream. |
| void format(Object format, Object... args) | It writes the formatted string to the current stream using specified format. |
| void format(Locale l, Object format, Object... args) | It writes the formatted string to the current stream using specified format. |

Example of java PrintStream class

In this example, we are simply printing integer and string value.

```
import java.io.FileOutputStream;
import java.io.PrintStream;
public class PrintStreamTest{
 public static void main(String args[])throws Exception{
   FileOutputStream fout=new FileOutputStream("D:\\testout.txt ");
   PrintStream pout=new PrintStream(fout);
   pout.println(2016);
   pout.println("Hello Java");
   pout.println("Welcome to Java");
   pout.close();
   fout.close();
  System.out.println("Success?");
 }
}
```

Output

```
Success...
```

The content of a text file **testout.txt** is set with the below data

```
2016
Hello Java
Welcome to Java
```

Example of printf() method using java PrintStream class:

Let's see the simple example of printing integer value by format specifier using **printf()** method of **java.io.PrintStream** class.

```
class PrintStreamTest{
 public static void main(String args[]){
   int a=19;
   System.out.printf("%d",a); //Note: out is the object of printstream
 }
}
```

Output

```
19
```

**Java StringWriter Class**

Java StringWriter class is a character stream that collects output from string buffer, which can be used to construct a string. The StringWriter class inherits the Writer class.

In StringWriter class, system resources like network sockets and files are not used, therefore closing the StringWriter is not necessary.

Java StringWriter class declaration

Let's see the declaration for Java.io.StringWriter class:

1. **public class** StringWriter **extends** Writer

Methods of StringWriter class

| Method | Description |
|---|---|
| void write(int c) | It is used to write the single character. |
| void write(String str) | It is used to write the string. |
| void write(String str, int off, int len) | It is used to write the portion of a string. |

| | |
|---|---|
| void write(char[] cbuf, int off, int len) | It is used to write the portion of an array of characters. |
| String toString() | It is used to return the buffer current value as a string. |
| StringBuffer getBuffer() | It is used t return the string buffer. |
| StringWriter append(char c) | It is used to append the specified character to the writer. |
| StringWriter append(CharSequence csq) | It is used to append the specified character sequence to the writer. |
| StringWriter append(CharSequence csq, int start, int end) | It is used to append the subsequence of specified character sequence to the writer. |
| void flush() | It is used to flush the stream. |
| void close() | It is used to close the stream. |

Java StringWriter Example

Let's see the simple example of StringWriter using BufferedReader to read file data from the stream.

```
import java.io.*;
public class StringWriterExample {
   public static void main(String[] args) throws IOException {
      char[] ary = new char[512];
      StringWriter writer = new StringWriter();
      FileInputStream input = null;
      BufferedReader buffer = null;
      input = new FileInputStream("D://testout.txt");
      buffer = new BufferedReader(new InputStreamReader(input, "UTF-8"));
      int x;
      while ((x = buffer.read(ary)) != -1) {
               writer.write(ary, 0, x);
      }
      System.out.println(writer.toString());
      writer.close();
      buffer.close();
```

```
        }
    }
```

testout.txt:

```
Javatpoint provides tutorial in Java, Spring, Hibernate, Android, PHP etc.
```

Output:

```
Javatpoint provides tutorial in Java, Spring, Hibernate, Android, PHP etc.
```

**Java StringReader Class**

Java StringReader class is a character stream with string as a source. It takes an input string and changes it into character stream. It inherits Reader class.

In StringReader class, system resources like network sockets and files are not used, therefore closing the StringReader is not necessary.

Java StringReader class declaration

Let's see the declaration for Java.io.StringReader class:

1. **public class** StringReader **extends** Reader

Methods of StringReader class

| **Method** | **Description** |
|---|---|
| int read() | It is used to read a single character. |
| int read(char[] cbuf, int off, int len) | It is used to read a character into a portion of an array. |
| boolean ready() | It is used to tell whether the stream is ready to be read. |
| boolean markSupported() | It is used to tell whether the stream support mark() operation. |
| long skip(long ns) | It is used to skip the specified number of character in a stream |
| void mark(int readAheadLimit) | It is used to mark the mark the present position in a stream. |

| | |
|---|---|
| void reset() | It is used to reset the stream. |
| void close() | It is used to close the stream. |

Java StringReader Example

```
import java.io.StringReader;
public class StringReaderExample {
    public static void main(String[] args) throws Exception {
        String srg = "Hello Java!! \nWelcome to Javatpoint.";
        StringReader reader = new StringReader(srg);
        int k=0;
            while((k=reader.read())!=-1){
                System.out.print((char)k);
            }
        }
}
```

Output:

```
Hello Java!!
Welcome to Javatpoint.
```

**Java 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.

Fields

| Modifier | Type | Field | Description |
|---|---|---|---|
| static | String | pathSeparator | It is system-dependent path-separator character, represented as a string for convenience. |
| static | char | pathSeparatorChar | It is system-dependent path-separator character. |

| | | | |
|---|---|---|---|
| static | String | separator | It is system-dependent default name-separator character, represented as a string for convenience. |
| static | char | separatorChar | It is system-dependent default name-separator character. |

Constructors

| Constructor | Description |
|---|---|
| File(File parent, String child) | It creates a new File instance from a parent abstract pathname and a child pathname string. |
| File(String pathname) | It creates a new File instance by converting the given pathname string into an abstract pathname. |
| File(String parent, String child) | It creates a new File instance from a parent pathname string and a child pathname string. |
| File(URI uri) | It creates a new File instance by converting the given file: URI into an abstract pathname. |

Useful Methods

| Modifier and Type | Method | Description |
|---|---|---|
| static File | createTempFile(String prefix, String suffix) | It creates an empty file in the default temporary-file directory, using the given prefix and suffix to generate its name. |
| boolean | createNewFile() | It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. |

| | | |
|---|---|---|
| boolean | canWrite() | It tests whether the application can modify the file denoted by this abstract pathname.String[] |
| boolean | canExecute() | It tests whether the application can execute the file denoted by this abstract pathname. |
| boolean | canRead() | It tests whether the application can read the file denoted by this abstract pathname. |
| boolean | isAbsolute() | It tests whether this abstract pathname is absolute. |
| boolean | isDirectory() | It tests whether the file denoted by this abstract pathname is a directory. |
| boolean | isFile() | It tests whether the file denoted by this abstract pathname is a normal file. |
| String | getName() | It returns the name of the file or directory denoted by this abstract pathname. |
| String | getParent() | It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory. |
| Path | toPath() | It returns a java.nio.file.Path object constructed from the this abstract path. |
| URI | toURI() | It constructs a file: URI that represents this abstract pathname. |
| File[] | listFiles() | It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname |
| long | getFreeSpace() | It returns the number of unallocated bytes in the partition named by this abstract path name. |

|  |  |  |
|---|---|---|
| String[] | list(FilenameFilter filter) | It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. |
| boolean | mkdir() | It creates the directory named by this abstract pathname. |

Java File Example 1

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

        try {
            File file = new File("javaFile123.txt");
            if (file.createNewFile()) {
                System.out.println("New File is created!");
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

Output:

```
New File is created!
```

## Random Access File:

The **Java.io.RandomAccessFile** class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.

### Class declaration

Following is the declaration for **Java.io.RandomAccessFile** class −

```
public class RandomAccessFile
   extends Object
      implements DataOutput, DataInput, Closeable
```

Class constructors

| S.N. | Constructor & Description |
|---|---|
| 1 | **RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, the file specified by the File argument. |
| 2 | **RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, a file with the specified name. |

The following example shows the usage of **java.io.RandomAccessFile.readLine()** method.

```
import java.io.*;
public class RandomAccessFileDemo {
   public static void main(String[] args) {
         try {
              // create a new RandomAccessFile with filename test
         RandomAccessFile raf = new RandomAccessFile("c:/test.txt", "rw");
         // write something in the file
         raf.writeUTF("Hello World");

         // set the file pointer at 0 position
         raf.seek(0);
         // print the line
         System.out.println("" + raf.readLine());
         // set the file pointer at 0 position
         raf.seek(0);
         // write something in the file
         raf.writeUTF("This is an example \n Hello World");
         raf.seek(0);
         // print the line
         System.out.println("" + raf.readLine());
```

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

Assuming we have a text file **c:/test.txt**, which has the following content. This file will be used as an input for our example program −

```
ABCDE
```

Let us compile and run the above program, this will produce the following result −

```
Hello World
This is an example
```

**Java Console Class**

The Java 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. The Console class is introduced since 1.5.

Let's see a simple example to read text from console.

1. String text=System.console().readLine();
2. System.out.println("Text is: "+text);

Java Console class declaration

Let's see the declaration for Java.io.Console class:

1. **public final class** Console **extends** Object **implements** Flushable

Java Console class methods

| **Method** | **Description** |
|---|---|
| Reader reader() | It is used to retrieve the reader object associated with the console |
| String readLine() | It is used to read a single line of text from the console. |

| | |
|---|---|
| String readLine(String fmt, Object... args) | It provides a formatted prompt then reads the single line of text from the console. |
| char[] readPassword() | It is used to read password that is not being displayed on the console. |
| char[] readPassword(String fmt, Object... args) | It provides a formatted prompt then reads the password that is not being displayed on the console. |
| Console format(String fmt, Object... args) | It is used to write a formatted string to the console output stream. |
| Console printf(String format, Object... args) | It is used to write a string to the console output stream. |
| PrintWriter writer() | It is used to retrieve the PrintWriter object associated with the console. |
| void flush() | It is used to flushes the console. |

How to get the object of Console

System class provides a static method console() that returns the singleton instance of Console class.

1. **public static** Console console(){}

Let's see the code to get the instance of Console class.

1. Console c=System.console();

Java Console Example

```
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);
}
}
```

Output

```
Enter your name: Nakul Jain
Welcome Nakul Jain
```

Java 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);
}
}
```

Output

```
Enter password:
Password is: 123
```

## Serialization in Java

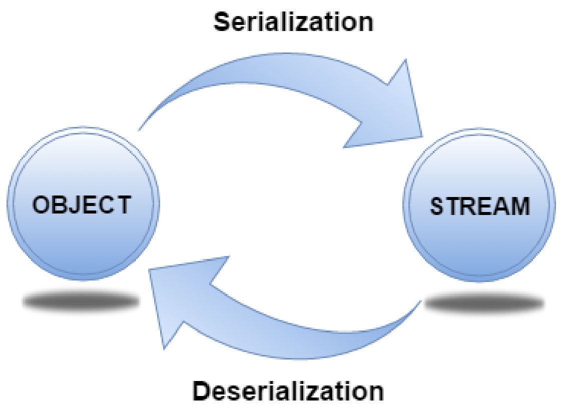
**Serialization in java** is a mechanism of *writing the state of an object into a byte stream*.

It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called *deserialization*.

Advantage of Java Serialization

It is mainly used to travel object's state on the network (known as marshaling).



java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implements *java.io.Serializable* interface by default.

Let's see the example given below:

```
import java.io.Serializable;
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

| **Method** | **Description** |
|---|---|
| 1) public final void writeObject(Object obj) throws IOException {} | writes the specified object to the ObjectOutputStream. |
| 2) public void flush() throws IOException {} | flushes the current output stream. |
| 3) public void close() throws IOException {} | closes the current output stream. |

Example of Java Serialization

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

```
import java.io.*;
class Persist{
 public static void main(String args[])throws Exception{
  Student s1 =new Student(211,"ravi");

  FileOutputStream fout=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(fout);

  out.writeObject(s1);
  out.flush();
  System.out.println("success");
 }
}
```

```
success
```

Deserialization in java

Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

| | |
|---|---|
| **1) public ObjectInputStream(InputStream in) throws IOException {}** | creates an ObjectInputStream that reads from the specified InputStream. |

Important Methods

| Method | Description |
|---|---|
| 1) public final Object readObject() throws IOException, ClassNotFoundException{} | reads an object from the input stream. |
| 2) public void close() throws IOException {} | closes ObjectInputStream. |

Example of Java Deserialization

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

  ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
  Student s=(Student)in.readObject();
  System.out.println(s.id+" "+s.name);

  in.close();
 }
}
```

```
211 ravi
```

## Java Enum

**Enum in java** is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

Points to remember for Java Enum

- enum improves type safety

- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple example of java enum

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args) {
for (Season s : Season.values())
System.out.println(s);

}}
```

```
Output:WINTER
    SPRING
    SUMMER
    FALL
```

## Autoboxing and Unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

Advantage of Autoboxing and Unboxing:

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

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);
```

```
 }
}
```

```
Output:50 5
```

download this example

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);
 }
}
```

```
Output:50
```

## Generics in Java

The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects.

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

*Advantage of Java Generics*

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. String s = list.get(0);

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error

**Syntax** to use generic collection

1. ClassOrInterface<Type>

**Example** to use Generics in java

1. ArrayList<String>

Full Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

```
Output:element is: jai
    rahul
    jai
```

Generic class

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

Let’s see the simple example to create and use the generic class.

**Creating generic class:**

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

The T type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

**Using generic class:**

Let’s see the code to use the generic class.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

```
Output:2
```

**Type Parameters**

The type parameters naming conventions are important to learn generics thoroughly. The commonly type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

**Generic Method**

Like generic class, we can create generic method that can accept any type of argument.

Let’s see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenerics4{

   public static < E > void printArray(E[] elements) {
       for ( E element : elements){
          System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
         System.out.println( "Printing Integer Array" );
        printArray( intArray  );
         System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

```
Output:Printing Integer Array
       10
       20
       30
       40
       50
       Printing Character Array
       J
       A
       V
       A
       T
       P
       O
       I
       N
       T
```