oSYLLABUS JAVA PROGRAMMING

Paper Code: ETCS-307	**L**	**T/P**	**C**
Paper: Java Programming	**3**	**1**	**4**

INSTRUCTIONS TO PAPER SETTERS: **MAXIMUM MARKS: 75**

Objective: To learn object oriented concepts and enhancing programming skills.

UNIT I

Overview and characteristics of Java, Java program Compilation and Execution Process Organization of the Java Virtual Machine, JVM as an interpreter and emulator, Instruction Set, class File Format, Verification, Class Area, Java Stack, Heap, Garbage Collection. Security Promises of the JVM, Security Architecture and Security Policy. Class loaders and security aspects, sandbox model [T1,R2][No. of Hrs.: 11]

UNIT II

Java Fundamentals, Data Types & Literals Variables, Wrapper Classes, Arrays, Arithmetic Operators, Logical Operators, Control of Flow, Classes and Instances, Class Member Modifiers Anonymous Inner Class Interfaces and Abstract Classes, inheritance, throw and throws clauses, user defined Exceptions, The String Buffer Class, tokenizer, applets, Life cycle of applet and Security concerns. [T1,T2][No. of Hrs.: 12]

UNIT III

Threads: Creating Threads, Thread Priority, Blocked States, Extending Thread Class, Runnable Interface, Starting Threads, Thread Synchronization, Synchronize Threads, Sync Code Block, Overriding Synced Methods, Thread Communication, wait, notify and notify all.AWT Components, Component Class, Container Class, Layout Manager Interface Default Layouts, Insets and Dimensions, Border Layout, Flow Layout, Grid Layout, Card Layout Grid Bag Layout AWT Events, Event Models, Listeners, Class Listener, Adapters, Action Event Methods Focus Event Key Event, Mouse Events, Window Event [T2][No. of Hrs.: 11]

UNIT IV

Input/Output Stream, Stream Filters, Buffered Streams, Data input and Output Stream, Print Stream Random Access File, JDBC (Database connectivity with MS-Access, Oracle, MS-SQL Server), Object serialization, Sockets, development of client Server applications, design of multithreaded server. Remote Method invocation, Java Native interfaces, Development of a JNI based application. Collection API Interfaces, Vector, stack, Hashtable classes, enumerations, set, List, Map, Iterators. [T1][R1][No. of Hrs.: 10]

Java IO Stream

Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

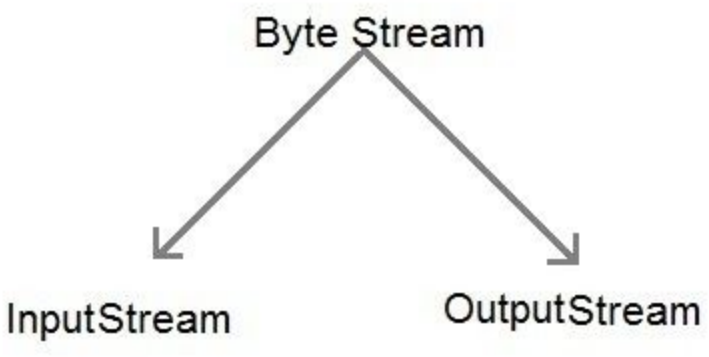
Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode

---

Java Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file

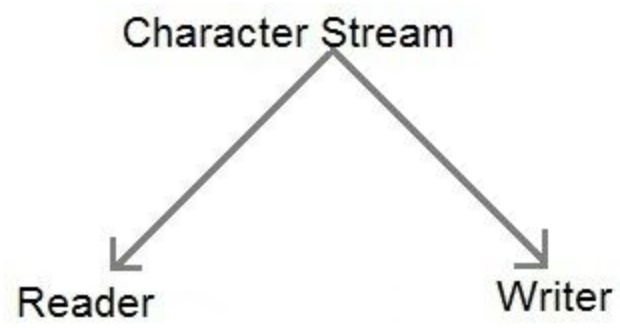
FileOutputStream	Output stream that writes to a file.
InputStream	Abstract class that describes stream input.
OutputStream	Abstract class that describes stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

These classes define several key methods. Two most important are

1. `read()` : reads byte of data.
2. `write()` : Writes byte of data.

Java Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



These two abstract classes have several concrete classes that handle unicode character (The Unicode standard uses hexadecimal to express a character. For example, the value 0x0041 represents the Latin character A.).

Differences :

Character Streams	Byte Streams
These handle data in 16 bit Unicode.	These handle data in bytes (8 bits).
Common classes are FileReader and FileWriter	Common classes are FileInputStream and FileOutputStream
Works with character data	Works with non-character data

Character Data: Read and write text data only.
Non character Data: store characters, audio, video and images.

1. **Character Data Copy**

```
import java.io.*;
class CopyFile
{
	public static void main(String[] args)
	{
		try
		{
			FileReader fr=new FileReader("Data.txt");
			FileWriter fw=new FileWriter("Copy.txt");
			int i=0;
			while((i=fr.read())!=-1)
			{
				fw.write(i);
			}
			fw.flush();
			fw.close();
			fr.close();
			System.out.println("Copied sucessfully..");

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

		}
	}
}
```

2. **Byte data Copy:**

```
import java.io.*;
class CopyPhotoS
{
	public static void main(String[] args)
	{
		try
		{
			FileInputStream fr=new FileInputStream("photo.jpg");
			FileOutputStream fw=new FileOutputStream("PhotoCopyS.jpg");
			int i=0;
			while((i=fr.read())!=-1)
			{
				fw.write(i);
			}
			fw.flush();
			fw.close();
```

```
                fr.close();
                System.out.println("Copied sucessfully..");

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

            }
        }
    }
```

Some important Character stream classes

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.

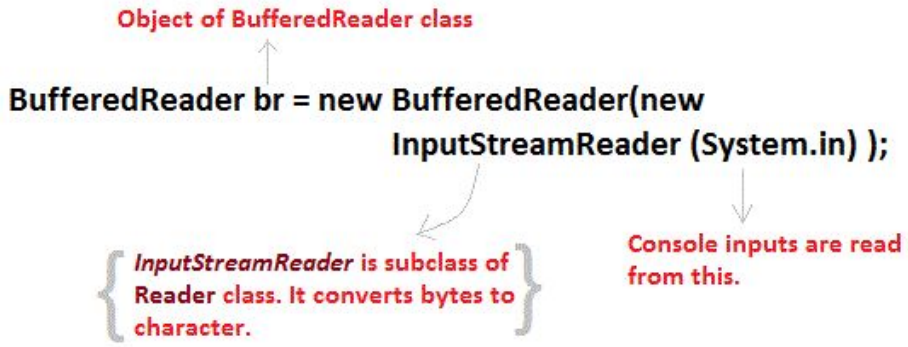
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.

InputStreamReader is= new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(is);

OR



Reading Characters

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

int read() throws IOException

Below is a simple example explaining character input.

```
class CharRead
{
  public static void main( String args[])
  {
    BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
    char c = (char)br.read();   //Reading character
  }
```

Ex-2: import java.io.*;

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

```
                System.out.println("Enter the number");
                InputStreamReader is= new InputStreamReader(System.in);

                BufferedReader br= new BufferedReader(is);
                //BufferedReader will not work without InputStreamReader
                int n= Integer.parseInt(br.readLine());
                //br.readLine will return string, so to get number, parsing to Integer is must
                System.out.println(n);

        }


}
```

Reading Strings in Java

To read string we have to use `readLine()` function with BufferedReader class's object.

String readLine() throws IOException

Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
  public static void main(String[] args)
  {
    String text;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    text = br.readLine();    //Reading String
    System.out.println(text);
  }

}
```

Program to read from a file using BufferedReader class

```
import java. Io *;
class ReadTest
{
  public static void main(String[] args)
  {
    try
    {
      File fl = new File("d:/myfile.txt");
      BufferedReader br = new BufferedReader(new FileReader(fl)) ;
      String str;
      while ((str=br.readLine())!=null)
      {
        System.out.println(str);
      }
      br.close();
      fl.close();
    }
    catch(IOException  e) {
      e.printStackTrace();
    }
  }

}
```

Program to write to a File using FileWriter class

```
import java. Io *;
class WriteTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   String str="Write this string to my file";
   FileWriter fw = new FileWriter(fl) ;
   fw.write(str);
   fw.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
```

Java - Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

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

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

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. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
public class CopyFile {

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
```

```
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Now let's have a file input.txt with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

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.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {

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

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

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

Now let's have a file input.txt with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

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

Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q" −

Example

```
import java.io.*;
public class ReadConsole {

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

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

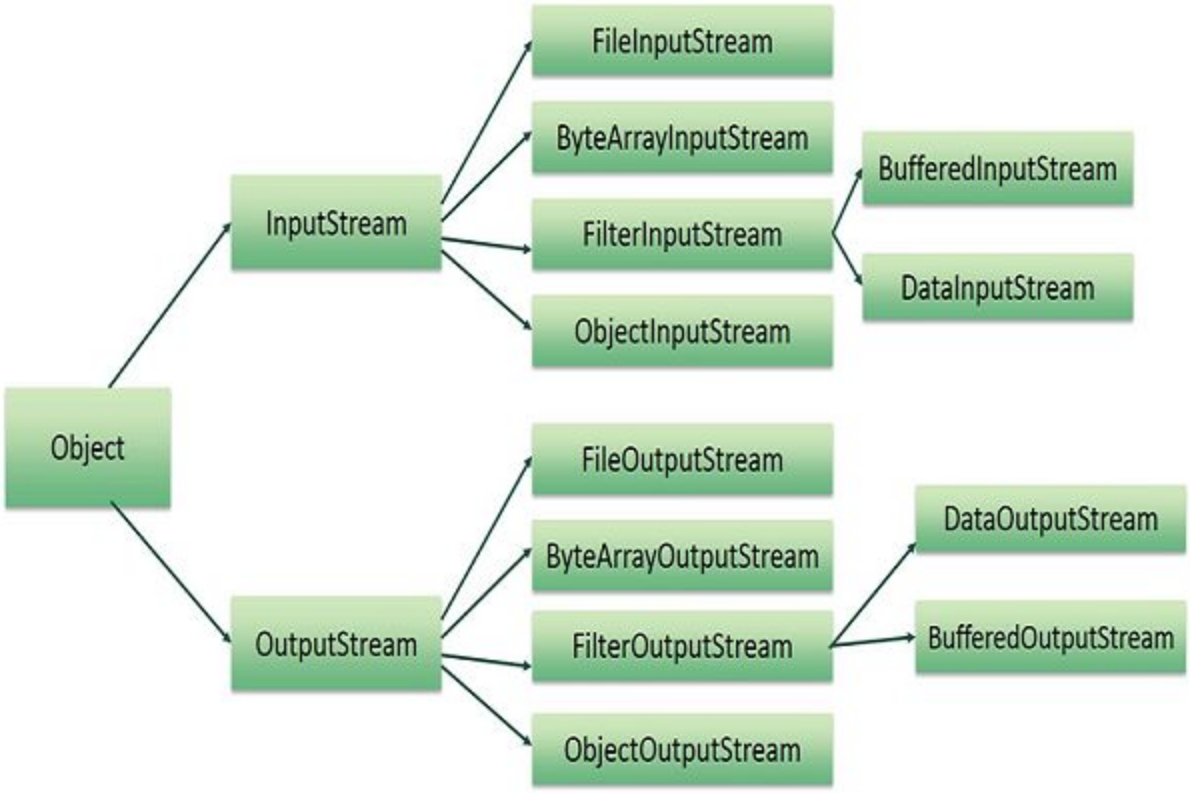
Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

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

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



The two important streams are FileInputStream and FileOutputStream, which would be discussed in this tutorial.

FileInputStream

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

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

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

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

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

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

Sr.No.	Method & Description

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

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

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

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

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

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

Sr.No.	Method & Description
1	public void close() throws IOException{} This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	protected void finalize()throws IOException {} This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.

3	public void write(int w)throws IOException{} This methods writes the specified byte to the output stream.
4	public void write(byte[] w) Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links −

- ByteArrayOutputStream
- DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;
public class fileStreamTest {

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      } catch (IOException e) {
         System.out.print("Exception");
```

```
        }
    }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Java I/O Tutorial

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

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

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

Stream

A stream is a sequence of data. 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 the 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 an error** message to the console.

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

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

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

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

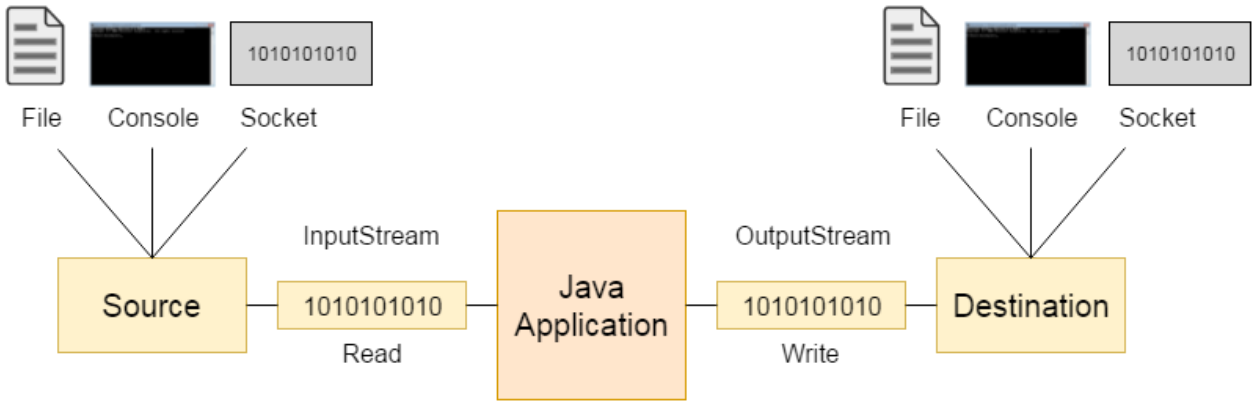
OutputStream

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 the working of Java OutputStream and InputStream by the figure given below.



OutputStream class

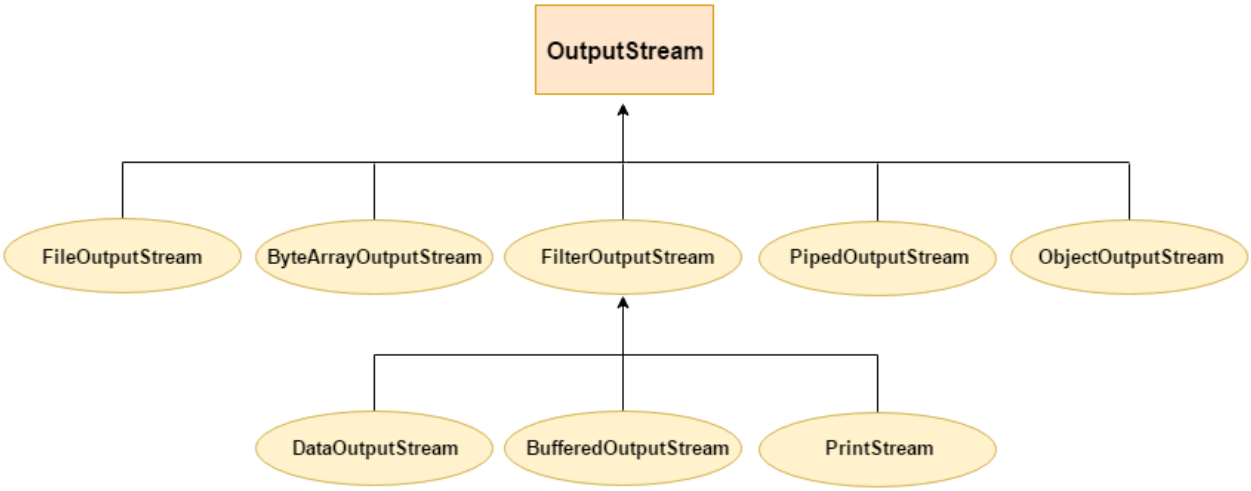
OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

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

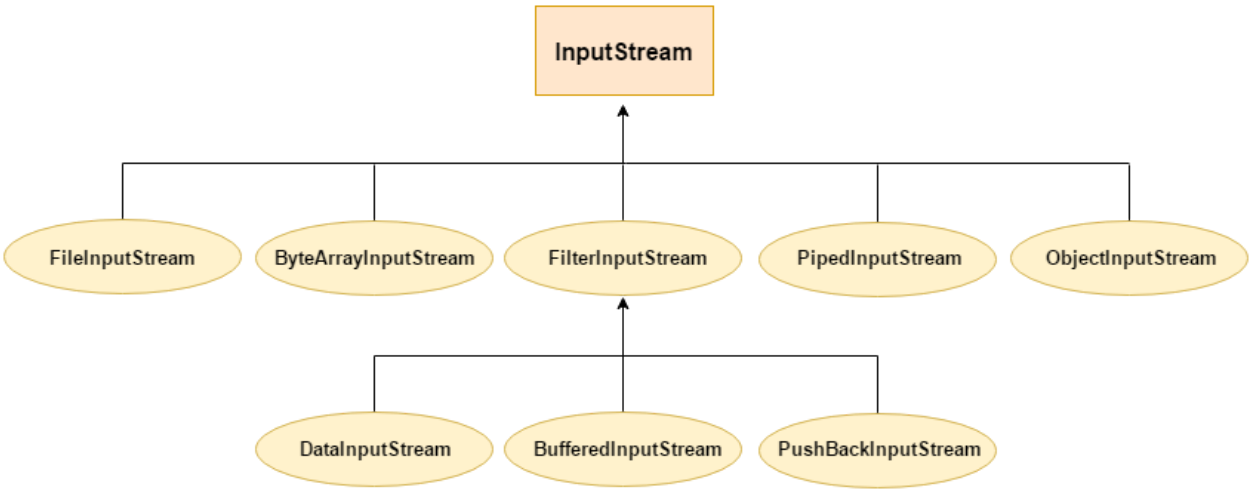
InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.

2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

InputStream Hierarchy



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 FileOutputStream.

FileOutputStream class declaration

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

1. **public class** FileOutputStream **extends** OutputStream

FileOutputStream class methods

Method	Description
protected void finalize()	It is used 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 Ex- 1: **write content to a file**

```
1. import java.io.FileOutputStream;
2. public class FileOutputStreamExample {
3.     public static void main(String args[]){
4.           try{
5.             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

```
1. import java.io.FileInputStream;
2. public class DataStreamExample {
3.     public static void main(String args[]){
4.         try{
5.           FileInputStream fin=new FileInputStream("D:\\testout.txt");
6.           int i=fin.read();
7.           System.out.print((char)i);
8.
9.           fin.close();
10.        }catch(Exception e){System.out.println(e);}
11.       }
12.     }
```

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

```
1. package com.javatpoint;
2. 
3. import java.io.FileInputStream;
4. public class DataStreamExample {
5.     public static void main(String args[]){
6.         try{
7.           FileInputStream fin=new FileInputStream("D:\\testout.txt");
8.           int i=0;
9.           while((i=fin.read())!=-1){
10.           System.out.print((char)i);
11.          }
12.          fin.close();
13.        }catch(Exception e){System.out.println(e);}
14.      }
15.     }
```

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.

```
1. package com.javatpoint;
2. import java.io.*;
3. public class BufferedOutputStreamExample{
4. public static void main(String args[])throws Exception{
5.     FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.     BufferedOutputStream bout=new BufferedOutputStream(fout);
7.     String s="Welcome to javaTpoint.";
8.     byte b[]=s.getBytes();
9.     bout.write(b);
10.    bout.flush();
11.    bout.close();
12.    fout.close();
13.    System.out.println("success");
14. }
15. }
```

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:

```
1. package com.javatpoint;
2.  import java.io.*;
3. public class BufferedInputStreamExample{
4.  public static void main(String args[]){
5.   try{
6.     FileInputStream fin=new FileInputStream("D:\\testout.txt");
7.     BufferedInputStream bin=new BufferedInputStream(fin);
8.     int i;
9.     while((i=bin.read())!=-1){
10.     System.out.print((char)i);
11.    }
12.    bin.close();
13.    fin.close();
14.  }catch(Exception e){System.out.println(e);}
15. }
16.}
```

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

javaTpoint

Output:

javaTpoint

Java FilterOutputStream Class

Java FilterOutputStream class implements the OutputStream class. It provides different sub classes such as **BufferedOutputStream and DataOutputStream** to provide additional functionality.

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.

```
1. package com.javatpoint;
2.
3. import java.io.*;
4. public class OutputExample {
5.     public static void main(String[] args) throws IOException {
```

```
6.          FileOutputStream file = new FileOutputStream(D:\\testout.txt);
7.          DataOutputStream data = new DataOutputStream(file);
8.          data.writeInt(65);
9.          data.flush();
10.         data.close();
11.         System.out.println("Succcess...");
12.     }
13. }
```

Output:

```
Succcess...
```

testout.txt:

```
A
```

Ex-2 : **DataOutputStreamEx.java**

```
import java.io.*;

class DataOutputStreamEx

{

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

{

DataOutputStream dos=new DataOutputStream(new FileOutputStream("xyz.txt"));

dos.writeInt(10);

dos.writeChar('A');

dos.writeDouble(23.33);

dos.close();

}

}
```

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

Ex-1:

```
import java.io.*;
class DataInputStreamEx
{
public static void main(String[] args) throws IOException
{
DataInputStream dis=new DataInputStream(new FileInputStream("xyz.text"));
int a=dis.readInt();
char ch=dis.readChar();
double d=dis.readDouble();
System.out.println(a+" "+ch+ " "+d);
dis.close();
}
}
```

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

```
1. package com.javatpoint;
2. import java.io.*;
3. public class DataStreamExample {
4.   public static void main(String[] args) throws IOException {
5.     InputStream input = new FileInputStream("D:\\testout.txt");
6.     DataInputStream inst = new DataInputStream(input);
7.     int count = input.available();
8.     byte[] ary = new byte[count];
9.     inst.read(ary);
```

```
10.     for (byte bt : ary) {
11.       char k = (char) bt;
12.       System.out.print(k+"-");
13.     }
14.   }
15. }
```

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

JAVA

Output:

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

java.io.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.

Commonly used methods of PrintStream class

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line./li>
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.

- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example of java.io.PrintStream class

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

```
1. import java.io.*;
2. class PrintStreamTest{
3.  public static void main(String args[])throws Exception{
4.    FileOutputStream fout=new FileOutputStream("mfile.txt");
5.    PrintStream pout=new PrintStream(fout);
6.    pout.println(1900f);
7.    pout.println('A');
8.    pout.println("Hello Java");
9.    pout.println("Welcome to Java");
10.   pout.close();
11.   fout.close();
12. }
13.}
```

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.

```
1. package com.javatpoint;
2. 
3. import java.io.FileReader;
4. public class FileReaderExample {
5.     public static void main(String args[])throws Exception{
6.           FileReader fr=new FileReader("D:\\testout.txt");
7.           int i;
8.           while((i=fr.read())!=-1)
9.           System.out.print((char)i);
10.          fr.close();
11.    }
12. }
```

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.

```
1. package com.javatpoint;
2. import java.io.FileWriter;
```

```
3.  public class FileWriterExample {
4.      public static void main(String args[]){
5.          try{
6.            FileWriter fw=new FileWriter("D:\\testout.txt");
7.            fw.write("Welcome to javaTpoint.");
8.            fw.close();
9.           }catch(Exception e){System.out.println(e);}
10.          System.out.println("Success...");
11.     }
12. }
```

Output:

```
Success...
```

testout.txt:

```
Welcome to javaTpoint.
```

Java PrintWriter class

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

Class declaration

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

```
1.  public class PrintWriter extends Writer
```

Methods of PrintWriter class

Method	Description

void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.
PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
PrintWriter append(CharSequence ch, int start, int end)	It is used to append a subsequence of specified character to the writer.
boolean checkError()	It is used to flushes the stream and check its error state.
protected void setError()	It is used to indicate that an error occurs.
protected void clearError()	It is used to clear the error state of a stream.
PrintWriter format(String format, Object... args)	It is used to write a formatted string to the writer using specified arguments and format string.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.
void close()	It is used to close the stream.

Java PrintWriter Example

Let's see the simple example of writing the data on a **console** and in a **text file testout.txt** using Java PrintWriter class.

```
1.  package com.javatpoint;
2.
3.  import java.io.File;
4.  import java.io.PrintWriter;
5.  public class PrintWriterExample {
6.      public static void main(String[] args) throws Exception {
7.              //Data to write on Console using PrintWriter
8.        PrintWriter writer = new PrintWriter(System.out);
9.        writer.write("Javatpoint provides tutorials of all technology.");
10.  writer.flush();
11.       writer.close();
12. //Data to write in File using PrintWriter
13.       PrintWriter writer1 =null;
14.         writer1 = new PrintWriter(new File("D:\\testout.txt"));
15.         writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");
16.                         writer1.flush();
17.         writer1.close();
18.    }
19. }
```

Outpt

```
Javatpoint provides tutorials of all technology.
```

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

```
1. import java.io.*;
2. public class FilterExample {
3.     public static void main(String[] args) throws IOException {
4.         File data = new File("D:\\testout.txt");
5.         FileOutputStream file = new FileOutputStream(data);
6.         FilterOutputStream filter = new FilterOutputStream(file);
7.         String s="Welcome to javaTpoint.";
8.         byte b[]=s.getBytes();
```

```
9.          filter.write(b);
10.         filter.flush();
11.         filter.close();
12.         file.close();
13.         System.out.println("Success...");
14.     }
15. }
16.
```

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

```
1. import java.io.*;
2. public class FilterExample {
3.     public static void main(String[] args) throws IOException {
4.         File data = new File("D:\\testout.txt");
5.         FileInputStream  file = new FileInputStream(data);
6.         FilterInputStream filter = new BufferedInputStream(file);
7.         int k =0;
8.         while((k=filter.read())!=-1){
9.             System.out.print((char)k);
10.        }
11.        file.close();
12.        filter.close();
13.    }
14. }
```

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

```
Welcome to javatpoint
```

Output:

```
Welcome to javatpoint
```

Java - RandomAccessFile

This class is used for **reading and writing to random access file.** A random access file behaves like a **large array of bytes.** There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than **EOFException is thrown**. It is a type of IOException.

1. When we Perform file operation in **any location of file** is called RAF.
2. We have a class named **RandomAccessFile**, which allows to open a file in read-write mode so that we can perform read and write operations using some objects.
3. RAF is Byte oriented class.
4. Constructor RAF: **RandomAccessFile(String fileName, String fileMode)**
    a. Name of the file : a string class object
    b. Mode: a string class object: Read/Write

5. File mode specifies whether the file has to be opened in the **read mode ("r") or read-write mode("rw").**

Constructor

Constructor	Description
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Method

Modifier and Type	Method	Method
void	close()	It closes this random access file stream and releases any system resources associated with the stream.
FileChannel	getChannel()	It returns the unique FileChannel object associated with this file.
int	readInt()	It reads a signed 32-bit integer from this file.
String	readUTF()	It reads in a string from this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	writeDouble(double v)	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void	writeFloat(float v)	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.

void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

Example

```
1.  import java.io.IOException;
2.  import java.io.RandomAccessFile;
3.
4.  public class RandomAccessFileExample {
5.      static final String FILEPATH ="myFile.TXT";
6.      public static void main(String[] args) {
7.          try {
8.              System.out.println(new String(readFromFile(FILEPATH, 0, 18)));

              //0: start pos, till 18th pos.

9.              writeToFile(FILEPATH, "I love my country and my people", 31);

    // write at 31st location

10.         } catch (IOException e) {
11.             e.printStackTrace();
12.         }
13.     }
14.     private static byte[] readFromFile(String filePath, int position, int size)
15.             throws IOException {
16.         RandomAccessFile file = new RandomAccessFile(filePath, "r");
17.         file.seek(position);
18.         byte[] bytes = new byte[size];
19.         file.read(bytes);
20.         file.close();
21.         return bytes;
22.     }
23.     private static void writeToFile(String filePath, String data, int position)
24.             throws IOException {
```

```
25.         RandomAccessFile file = new RandomAccessFile(filePath, "rw");
26.         file.seek(position);
27.         file.write(data.getBytes());
28.         file.close();
29.     }
30. }
```

The myFile.TXT contains text **"This class is used for reading and writing to random access file."**

after running the program it will contains

Output:

On Console: This class is used

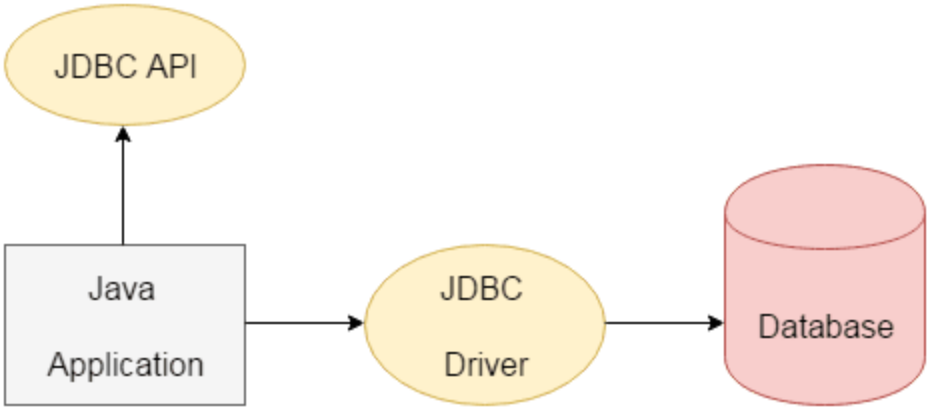
On file: This class is used for reading I love my country and my peoplele.

Java JDBC Tutorial

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database.

What is JDBC Driver in Java?

A driver is nothing but software required to connect to a database from Java program. JDBC is just an API, which Java has designed and to implementation of these APIs lies on different vendor because different database works in a different way, they internally use different protocols.

So MySQL gives its own implementation of JDBC, we call it MySQL JDBC driver and we use it when we want to connect to MySQL database from Java program. Similarly Oracle, SQL SERVER, Sybase and PostgreSQL have provided their own implementation of JDBC API to connect them. Since Java program uses JDBC API, they are portable across different database, all you need to do is change the JDBC driver, which is just a JAR file if you are using type 4 JDBC driver.

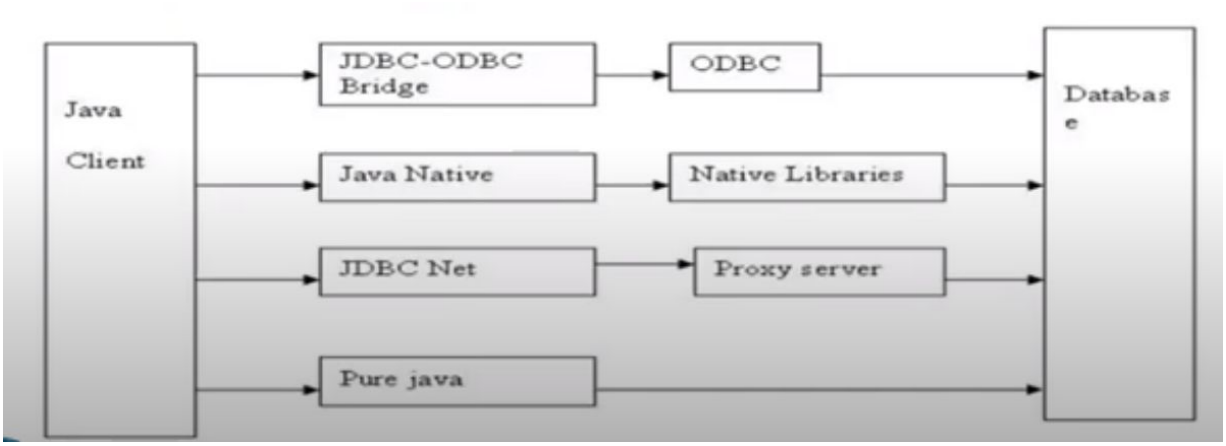
There are 4 types of JDBC drivers:

1. **Type1**: JDBC-ODBC bridge driver
2. **Type 2**: Native-API driver (partially java driver)
3. **Type 3**: Network Protocol driver (fully java driver)
4. **Type 4**: Thin driver (fully java driver)

Why many types of JDBC Drivers

The different types of JDBC driver comes from the fact how they work, which is basically driven by two factors, portability, and performance. Type 1 JDBC driver is the poorest in terms of portability and performance while type 4 JDBC drivers are highly portable and gives the best performance.

Since the database is very important and almost all Java application uses the database in some form or other, it's important to learn JDBC well.



1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver **converts JDBC method calls into the ODBC function calls.** This is now discouraged because of thin driver. (Vendor based database libraries means you need to use libraries provided by vendors to connect to database. For example You need to use ojdbc10.jar for Oracle , postgresql-9.4.1207.jar for postgresql etc.)

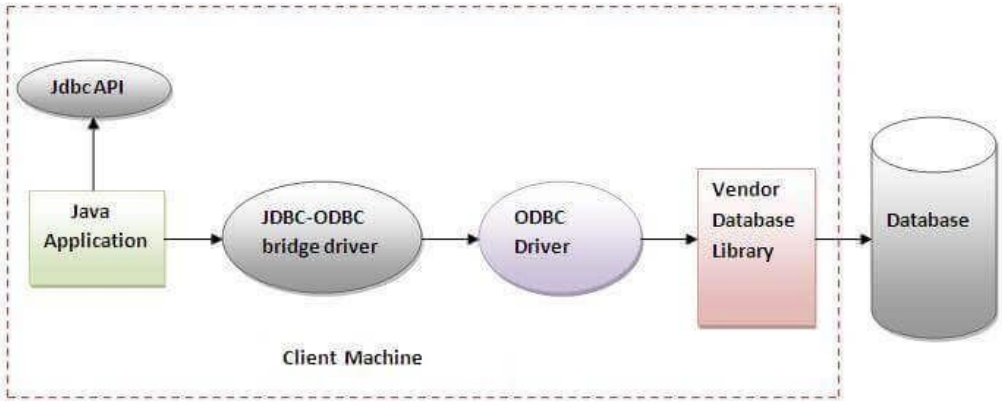


Figure- JDBC-ODBC Bridge Driver

In Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

---

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts **JDBC method calls into native calls** (In rough terms, a "**native** function **call**" means **calling** a function that is implemented by, and specific to, the operating system (or family of operating systems). The functions in the win32 API could be described by windows programmers as **native** functions.) of the database API. It is not written entirely in java.

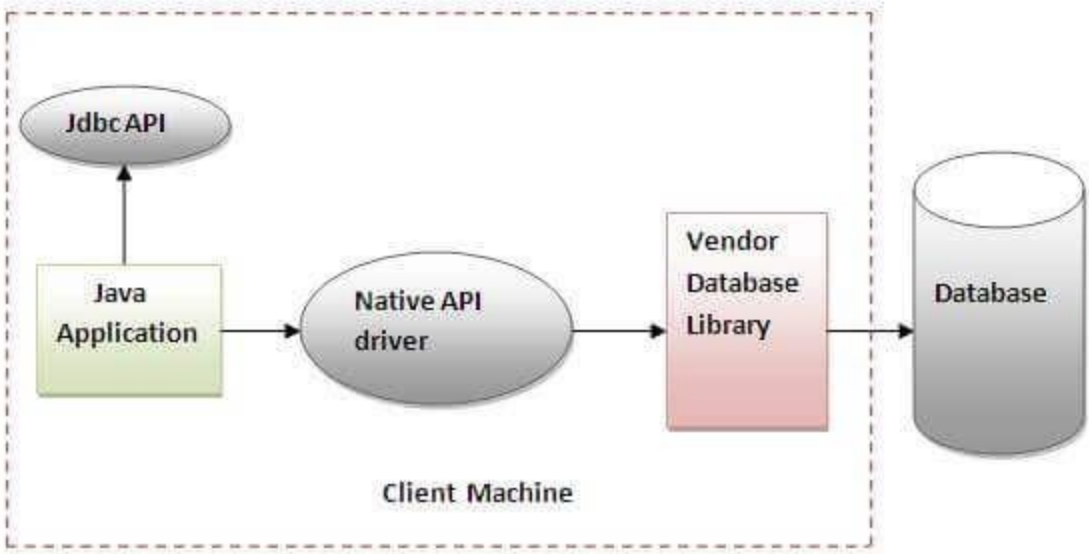


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

---

3) Network Protocol driver

The Network Protocol driver uses **middleware (application server)** that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

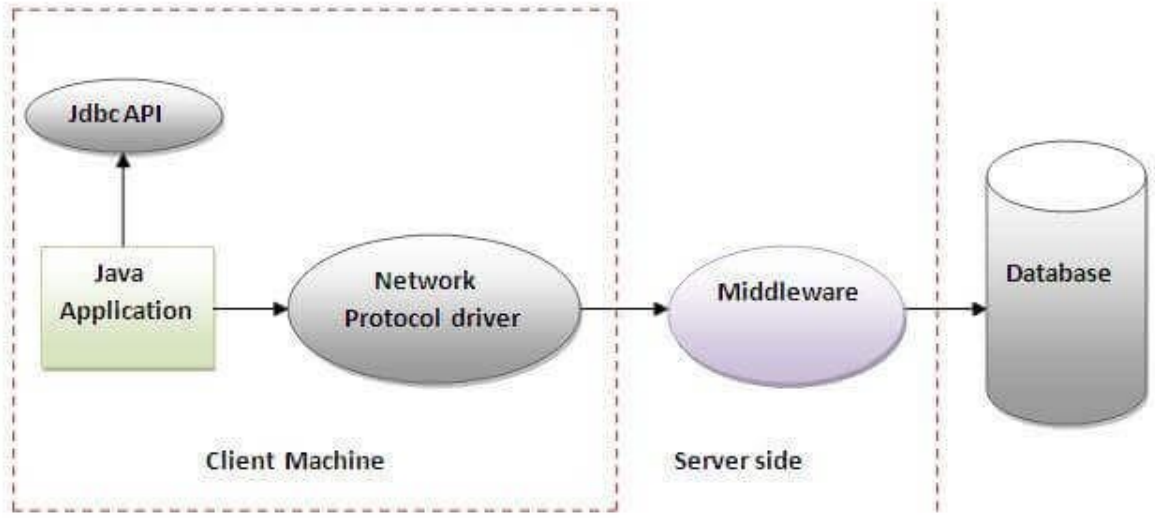


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

---

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol.

That is why it is known as **thin driver.** It is fully written in Java language.

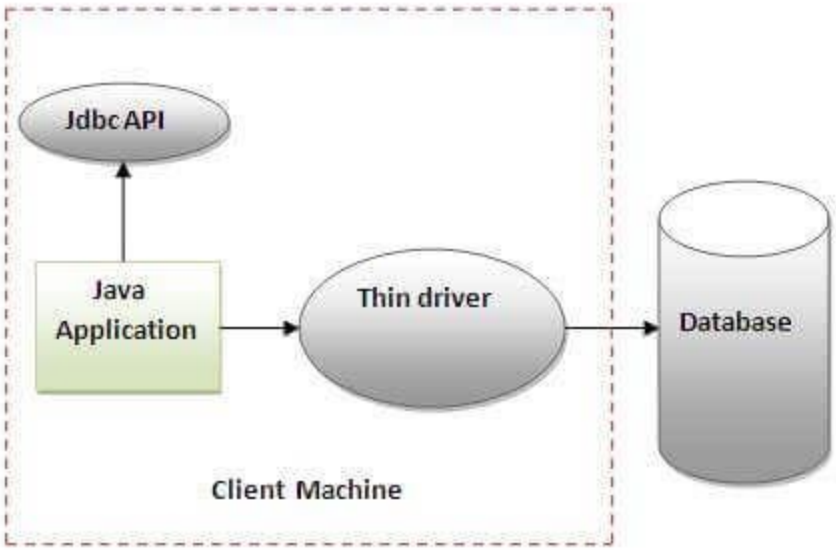


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

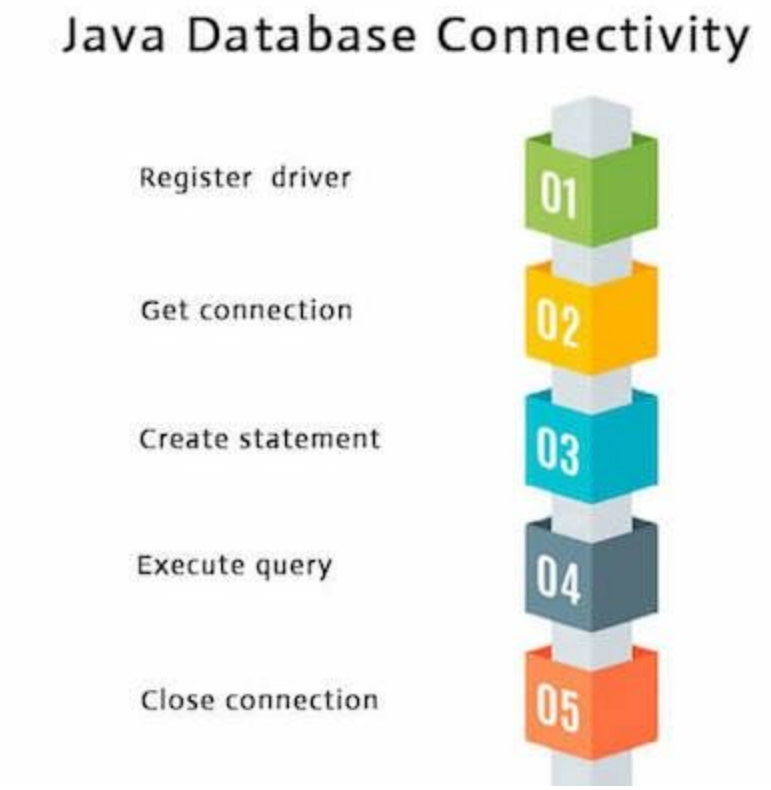
Disadvantage:

- Drivers depend on the Database.

Java Database Connectivity with 5 Steps

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Step-1 : Register the Driver class
- Step-2: Create connection
- Step-3: Create statement
- Step4: Execute queries
- Step5: Close connection



---

1) Register the driver class

The **forName()** method of **Class** class is used to **register the driver class**. This method is used to dynamically load the driver class.

Syntax of forName() method

```
1. public static void forName(String className)throws ClassNotFoundException
```

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

Example to register the OracleDriver class

Here, Java program is loading oracle driver to establish database connection.

```
1. Class.forName("oracle.jdbc.driver.OracleDriver");
```

---

2) Create the connection object

The **getConnection()** method of **DriverManager** class is used to establish connection with the database.

Syntax of getConnection() method

```
1. 1) public static Connection getConnection(String url)throws SQLException
2. 2) public static Connection getConnection(String url,String name,String password)
      throws SQLException
```

Example to establish connection with the Oracle database

```
1. Connection con=DriverManager.getConnection(
   "jdbc:oracle:thin:@localhost:1521:xe","system","password");
```

---

3) Create the Statement object

The createStatement() method of **Connection interfac**e is used to **create statement.** The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

```
1. public Statement createStatement()throws SQLException
```

Example to create the statement object

```
1. Statement stmt=con.createStatement();
```

---

4) Execute the query

The **executeQuery()** method of **Statement interface** is used to execute queries to the database. This method returns the **object of ResultSet** that can be used to get all the records of a table.

Syntax of executeQuery() method

```
1. public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

```
1. ResultSet rs=stmt.executeQuery("select * from emp");
2. while(rs.next()){
3. System.out.println(rs.getInt(1)+" "+rs.getString(2));
4. }
```

---

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
1. public void close()throws SQLException
```

Example to close connection

```
1. con.close();
```

Note: Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement.

It avoids explicit connection closing step.

More Statements (interfaces)

Once a connection is obtained we can interact with the database. **The JDBC *Statement, CallableStatement,* and *PreparedStatement* interfaces** define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

JDBC API provides 3 different interfaces to execute the different types of SQL queries. They are,

1) ***Statement*** – Used to execute normal SQL queries.

2) ***PreparedStatement*** – Used to execute dynamic or parameterized SQL queries.

3) ***CallableStatement*** – Used to execute the stored procedures.

These three interfaces look very similar but they differ significantly from one another in the functionalities they provide and the performance they give. In this post, we will discuss about the differences between Statement, PreparedStatement and CallableStatement in detail.

1) Statement

Statement interface is used to execute normal SQL queries. You can't pass the parameters to SQL query at run time using this interface. This interface is preferred over other two interfaces if you are executing a particular SQL query only once. The performance of this interface is also very less compared to other two interfaces. In most of time, Statement interface is used for DDL statements like **CREATE**, **ALTER**, **DROP** etc. For example,

```
1
    //Creating The Statement Object
2
3
    Statement stmt = con.createStatement();
4
5
    //Executing The Statement
6
7
    stmt.executeUpdate("CREATE TABLE STUDENT(ID NUMBER NOT
    NULL, NAME VARCHAR)");
```

2) PreparedStatement

PreparedStatement is used to execute dynamic or parameterized SQL queries. PreparedStatement extends Statement interface. You can pass the parameters to SQL query at run time using this interface. It is recommended to use PreparedStatement if you are executing a

particular SQL query multiple times. It gives better performance than Statement interface. Because, PreparedStatement are precompiled and the query plan is created only once irrespective of how many times you are executing that query. This will save lots of time.

```
1   //Creating PreparedStatement object
2
3
4   PreparedStatement pstmt = con.prepareStatement("update
    STUDENT set NAME = ?, where ID = ?");
5
6
7   //Setting values to place holders using setter methods of
8   PreparedStatement object
9
10
    pstmt.setString(1, "MyName");   //Assigns "MyName" to first
11  place holder
12
13
    pstmt.setInt(2, 111);    //Assigns "111" to second place
    holder


    //Executing PreparedStatement


    pstmt.executeUpdate();
```

Java Database Connectivity with Oracle

To connect java application with the oracle database, we need to follow 5 following steps. In this example, we are using Oracle 10g as the database. So we need to know following information for the oracle database:

1. **Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.
2. **Connection URL:** The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, localhost is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.
3. **Username:** The default username for the oracle database is **system**.
4. **Password:** It is the password given by the user at the time of installing the oracle database.

Create a Table

Before establishing connection, let's first create a table in oracle database. Following is the SQL query to create a table.

```
create table emp(id number(10),name varchar2(40),age number(3));
```

Example to Connect Java Application with Oracle database

In this example, we are connecting to an Oracle database and getting data from **emp** table. Here, **system** and **oracle** are the username and password of the Oracle database.

```
1. import java.sql.*;
2. class OracleCon{
3. public static void main(String args[]){
4. try{
5. //step1 load the driver class
6. Class.forName("oracle.jdbc.driver.OracleDriver");
7.
8. //step2 create  the connection object
9. Connection con=DriverManager.getConnection(
10. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
11.
12. //step3 create the statement object
13. Statement stmt=con.createStatement();
14.
15. //step4 execute query
16. ResultSet rs=stmt.executeQuery("select * from emp");
17. while(rs.next())
18. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
19.
20. //step5 close the connection object
21. con.close();
22.
23. }catch(Exception e){ System.out.println(e);}
24.
25. }
26. }
```

The above example will fetch all the records of emp table.

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

Two ways to load the jar file:

1. paste the ojdbc14.jar file in jre/lib/ext folder
2. set classpath

1) paste the ojdbc14.jar file in JRE/lib/ext folder:

Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath:

Firstly, search the ojdbc14.jar file then open command prompt and write:

1. C:>set classpath=c:\folder\ojdbc14.jar;.;

How to set the permanent classpath:

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;.; as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;

Java Database Connectivity with MySQL

To connect a Java application with the MySQL database, we need to follow 5 following steps.

In this example we are using **MySql** as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/dbase** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and dbase is the database name. We may use any database, in such case, we need to replace the dbase with our database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

1. create database dbase;
2. use dbase; //Database name
3. create table emp(id **int**(10),name varchar(40),age **int**(3));

Example to Connect Java Application with mysql database

In this example, dbase is the database name, root is the username and password both.

```
1. import java.sql.*;
2. class MysqlCon{
3. public static void main(String args[]){
4. try{
5. Class.forName("com.mysql.jdbc.Driver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:mysql://localhost:3306/dbase","root","root");
8. //here dbase is database name, root is username and password
9. Statement stmt=con.createStatement();
```

```
10. ResultSet rs=stmt.executeQuery("select * from emp");
11. while(rs.next())
12. System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
13. con.close();
14. }catch(Exception e){ System.out.println(e);}
15. }
16. }
```

The above example will fetch all the records of emp table.

To connect java application with the mysql database, **mysqlconnector.jar** file is required to be loaded.

Two ways to load the jar file:

1. Paste the mysqlconnector.jar file in jre/lib/ext folder
2. Set classpath

1) Paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2) Set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath

open command prompt and write:

1. C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;

Java JDBC Example Connect to Microsoft Access Database

This JDBC tutorial guides you how to develop a Java program that connects to a Microsoft Access Database. In the early days of JDBC, you can connect to an Access database via JDBC ODBC driver provided by JDK. However JDBC ODBC driver is no longer supported so you need to use a third-party JDBC driver for Microsoft Access. And your Java code still uses JDBC API as normal.

Connectivity with Access without DSN

There are two ways to connect java application with the access database.

1. Without DSN (Data Source Name)
2. With DSN

Java is mostly used with Oracle, mysql, or DB2 database. So you can learn this topic only for knowledge.

DSN: It is the name that applications use to request a connection to an ODBC Data Source. In other words, it is a symbolic name that represents the ODBC connection. It stores the connection details like database name, directory, database driver, UserID, password, etc. when making a connection to the ODBC.

Example to Connect Java Application with access without DSN

In this example, we are going to connect the java program with the access database. In such case, we have created the login table in the access database. There is only one column in the table named name. Let's get all the name of the login table.

```
1. import java.sql.*;
2. class Test{
3. public static void main(String ar[]){
4.  try{
5.    String database="student.mdb";//Here database exists in the current directory
6.
7.    String url="jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};
8.                  DBQ=" + database + ";DriverID=22;READONLY=true";
9.
10.   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
11.   Connection c=DriverManager.getConnection(url);
```

```
12.    Statement st=c.createStatement();
13.    ResultSet rs=st.executeQuery("select * from login");
14.
15.    while(rs.next()){
16.     System.out.println(rs.getString(1));
17.    }
18.
19. }catch(Exception ee){System.out.println(ee);}
20.
21. }}
```

Example to Connect Java Application with access with DSN

Connectivity with type1 driver is not considered good. To connect java application with type1 driver, create DSN first, (It is the name that applications use to request a connection to an ODBC Data Source. In other words, it is a symbolic name that represents the ODBC connection. It stores the connection details like database name, directory, database driver, UserID, password, etc. when making a connection to the ODBC.)here we are assuming your dsn name is mydsn.

```
1.  import java.sql.*;
2.  class Test{
3.  public static void main(String ar[]){
4.   try{
5.     String url="jdbc:odbc:mydsn";
6.     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
7.     Connection c=DriverManager.getConnection(url);
8.     Statement st=c.createStatement();
9.     ResultSet rs=st.executeQuery("select * from login");
10.
11.    while(rs.next()){
12.     System.out.println(rs.getString(1));
13.    }
14.
15. }catch(Exception ee){System.out.println(ee);}
16.
17. }}
```

Steps to ADD DSN:

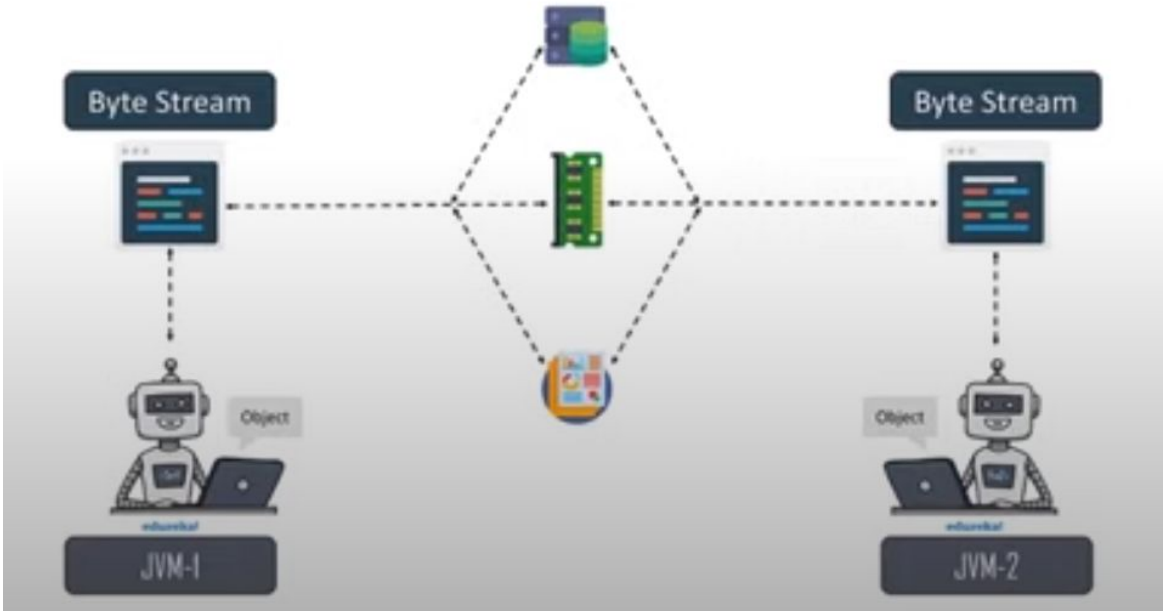
1. Click **Start**, and then click **Control Panel**.
1. In the Control Panel, double-click **Administrative Tools**.
2. In the Administrative Tools dialog box, double-click **Data Sources (ODBC)**.
   The **ODBC Data Source Administrator** dialog box appears.
3. Click **User DSN**, **System DSN**, or **File DSN**, depending on the type of data source you want to add.
4. Click **Add**.
5. Select the driver that you want to use, and then click **Finish** or **Next**.
   If the driver you want is not listed, contact the administrator of the database you are connecting to for information about how to obtain the correct driver.

Object Serialization and De-Serialization in Java

When you create a class, you may create an object for that particular class and once we execute/terminate the program, the object is destroyed by itself via the garbage collector thread.

what happens if you want to call that class without re-creating the object? In those cases, what you do is use the serialization concept by converting **data into a byte stream.**

object serialization is a process used to convert the state of an object into a byte stream, which can be persisted into disk/file or sent over the network to any other running java virtual machine. The reverse process of creating an object from the byte stream is called **deserialization** . The byte stream created is platform independent. so, the object serialized on one platform can be deserialized on a different platform.



Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out −

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object −

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

Emp.java

```
import java.io.Serializable;

public class Emp implements Serializable
{
  public String name;
  public String address;

}
```

SerialDemo.java

```
import java.io.*;

public class SerialDemo {

   public static void main(String [] args) {
      Emp e = new Emp();
      e.name = "Vaishali";
      e.address = "ADGITM";

      try {
         FileOutputStream fileOut =new FileOutputStream("Employee.txt");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
         fileOut.close();
         System.out.printf("Serialized data is saved in Employee.txt");
      } catch (IOException i) {
         i.printStackTrace();
      }
   }
}
```

Contents Of Employee.txt:

¬í sr Emp_ Ò¤¿¶¬› L addresst Ljava/lang/String;L nameq ~ xpt ADGITMt Vaishali

DeserialDemo.java

```
import java.io.*;
public class DeserialDemo {

   public static void main(String [] args) {
      Emp e = null;
      try {
         FileInputStream fileIn = new FileInputStream("Employee.txt");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Emp) in.readObject();
         in.close();
         fileIn.close();
      } catch (IOException i) {
         i.printStackTrace();
         return;
      } catch (ClassNotFoundException c) {
         System.out.println("Employee class not found");
         c.printStackTrace();
         return;
      }

      System.out.println("Deserialized Employee...");
      System.out.println("Name: " + e.name);
      System.out.println("Address: " + e.address);
   }
}
```

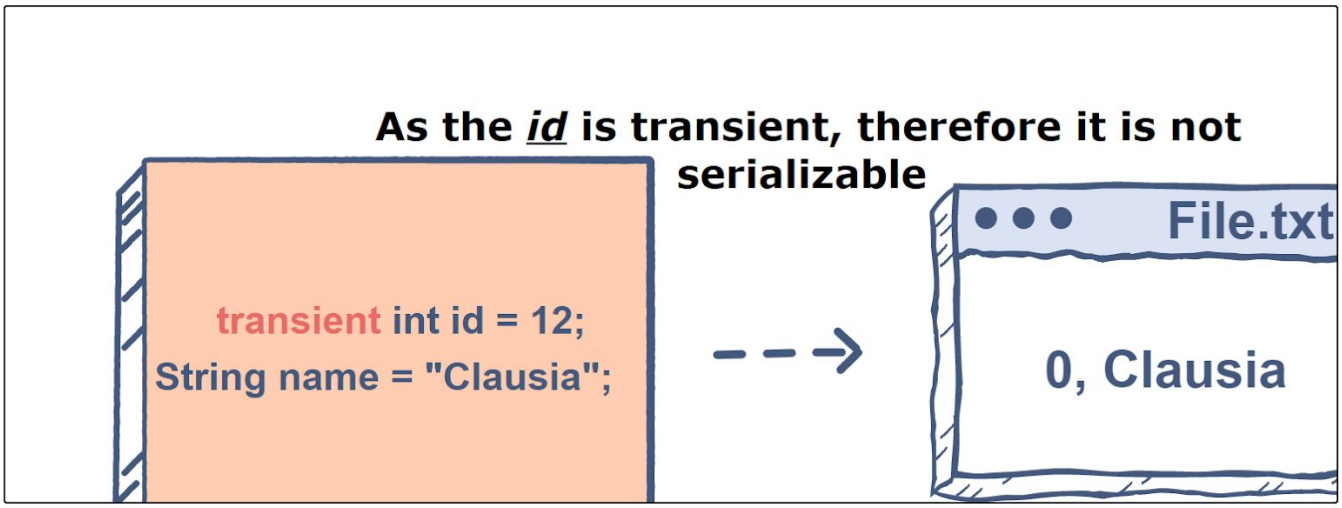
Output:

```
Deserialized Employee...
Name: Vaishali
Address: ADGITM
```

Notice that for a class to be **serialized successfully,** two conditions must be met −

- The class must implement **the java.io.Serializable interface.**
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient.**

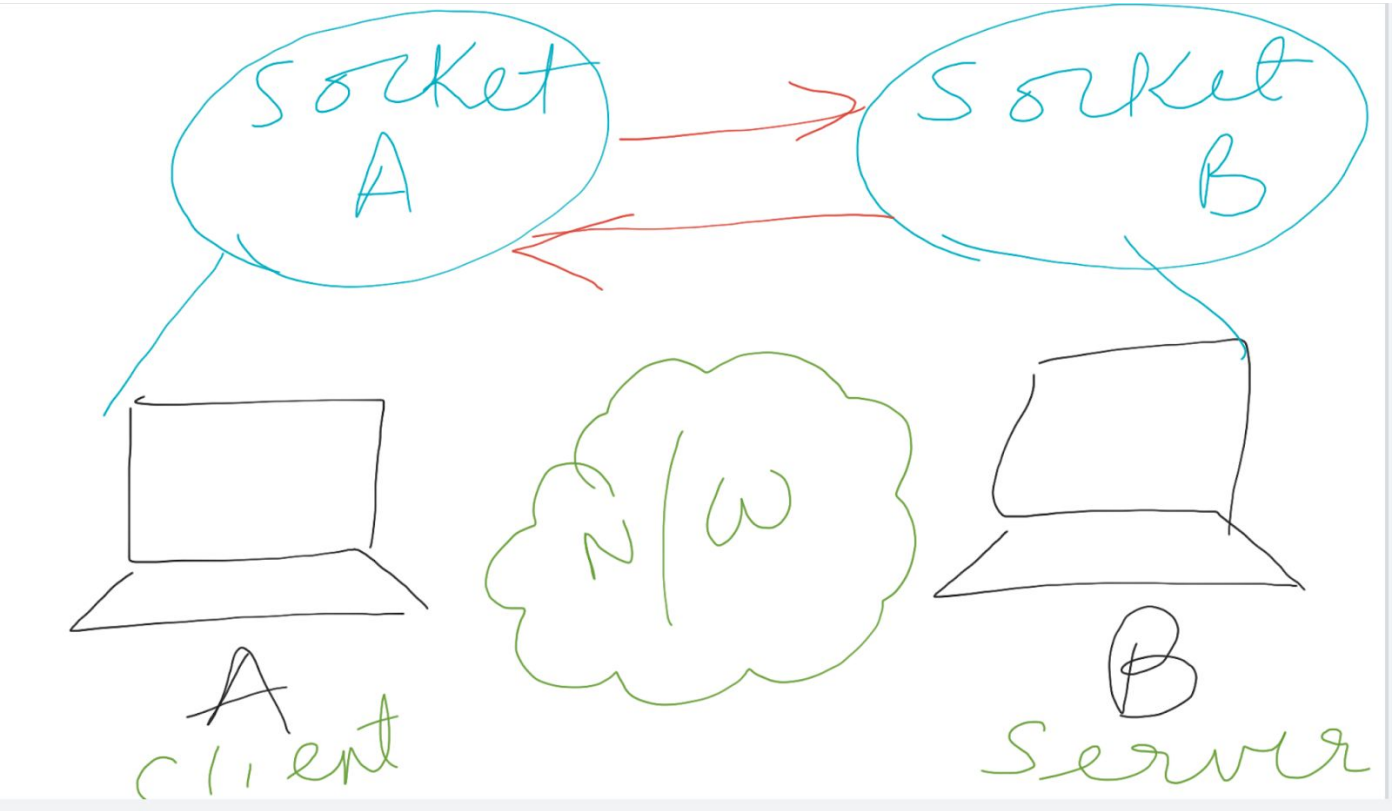
The `transient` **keyword** in Java is used to avoid serialization. If any object of a data structure is defined as a `transient`, then it will not be serialized.



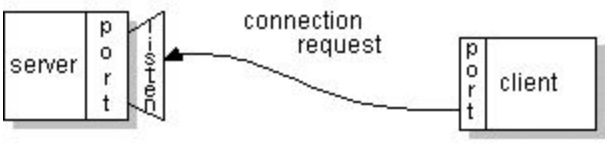
SOCKET

What Is a Socket?

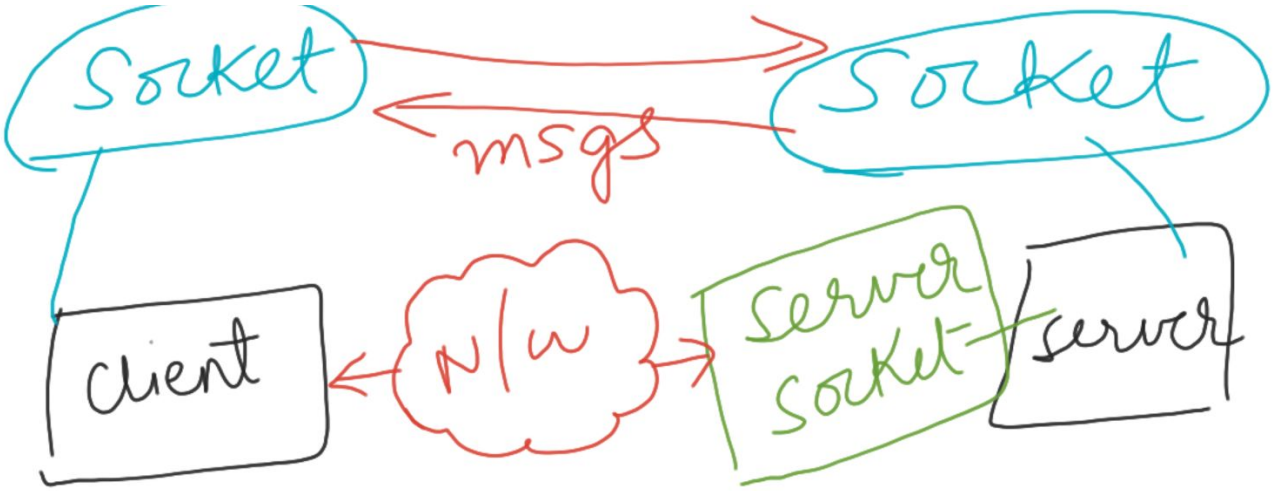
Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.



On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to connect with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it



can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The `java.net` package in the Java platform provides a class, `Socket`**,** that implements one side of a two-way connection between your Java program and another program on the network. The `Socket` class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the `java.net.Socket`

class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

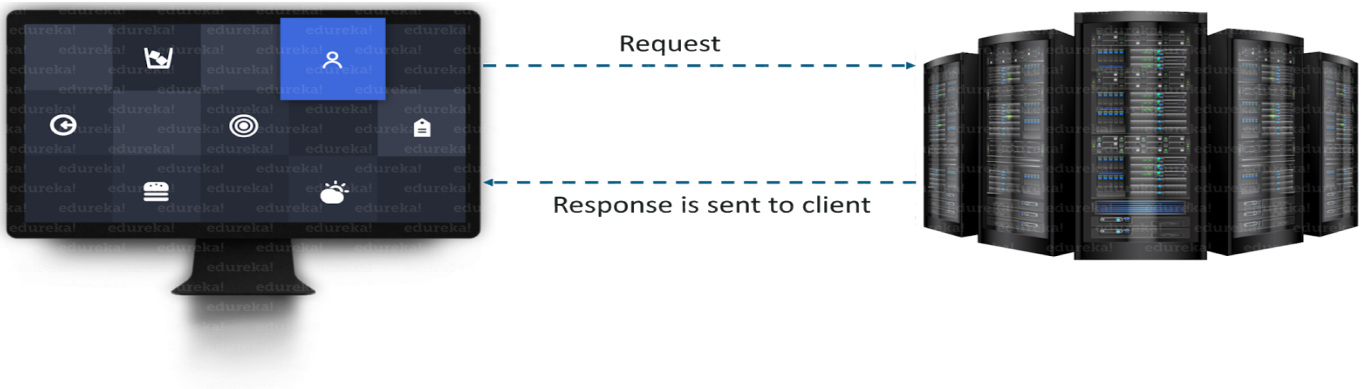
Additionally, `java.net` includes the `ServerSocket` class, which implements a socket that servers can use to listen for and accept connections to clients.

If you are trying to connect to the Web, the `URL` class and related classes (`URLConnection`, `URLEncoder`) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation.

Socket programming in Java is used for communication between the applications that are running on different JRE. It can be either connection-oriented or connectionless. On the whole, **a socket is a way to establish a connection between a client and a server.**

What is Socket Programming in Java?

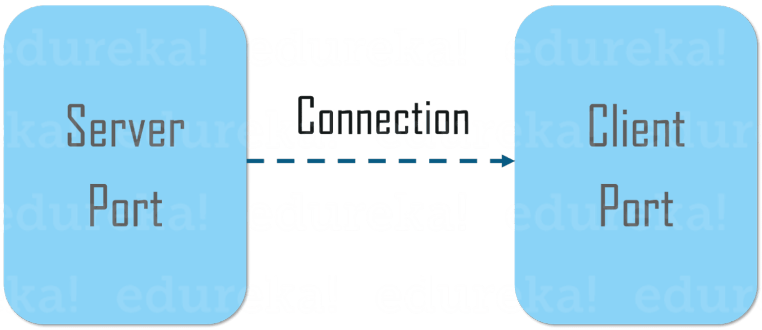
Socket programming is a way of connecting two nodes on a network to communicate with each other. One *socket* (node) listens on a particular port at an IP, while other *socket* reaches out to the other in order to form a connection.



The server forms the listener ***socket*** *while* the client reaches out to the server. Socket and Server Socket classes are used for connection-oriented socket programming.

What is a Socket in Java?

A socket in Java is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

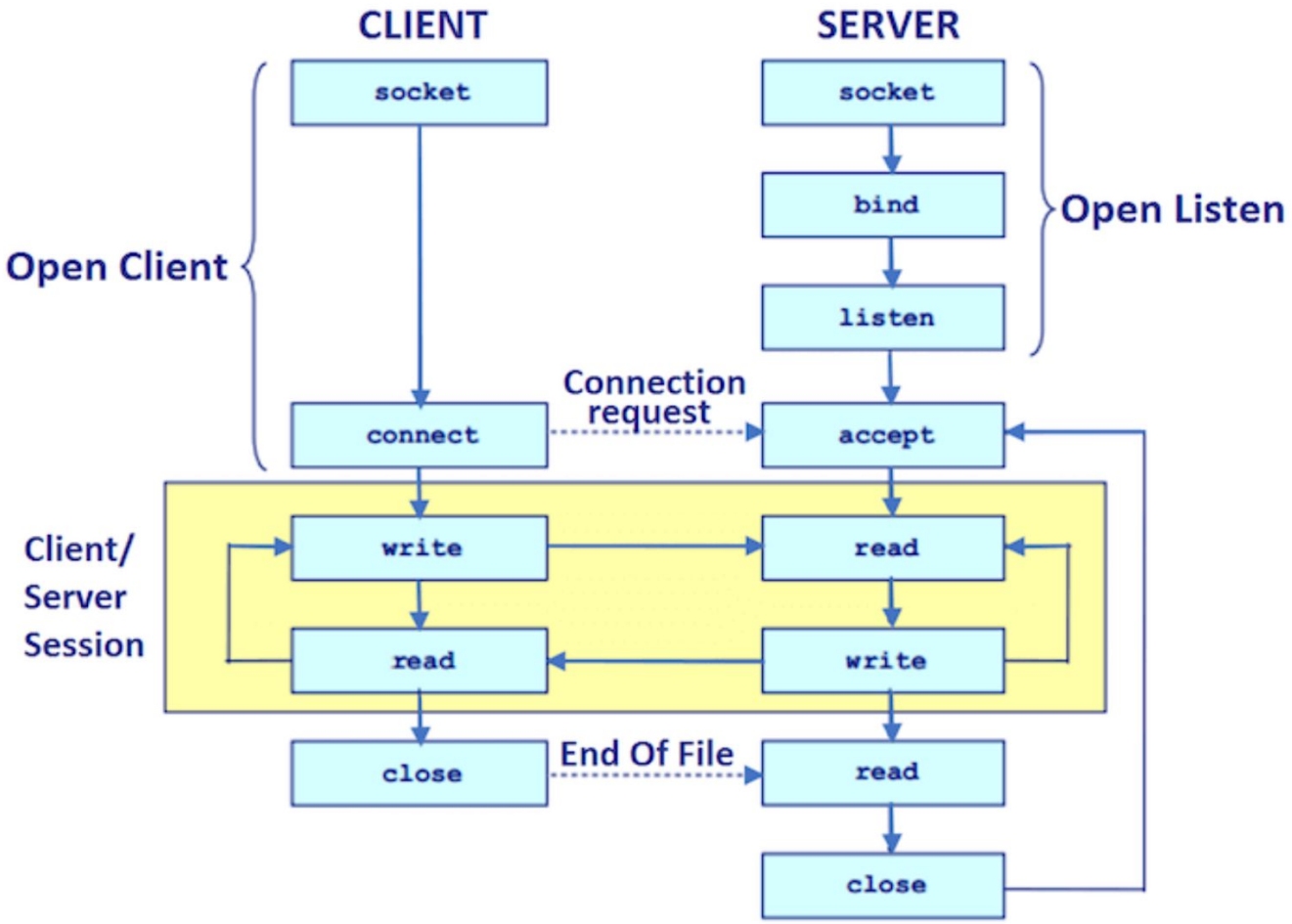


An endpoint is a combination of an IP address and a port number. The package in the Java platform provides a class, Socket that implements one side of a two-way connection between your Java program and another program on the network. The class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Client Side Programming

In the case of client-side programming, the client will first wait for the server to start. Once the server is up and running, it will send the requests to the server. After that, the client will wait for the response from the server. So, this is the whole logic of client and server communication.

In order to initiate a clients request, you need to follow the below-mentioned steps:



1. **Establish a Connection**

The very first step is to **establish a socket connection. A socket connection implies that the two machines have information about each other's network location (IP Address) and TCP port.** You can create a Socket with the help of a below statement:

Socket socket = new Socket("127.0.0.1", 5000)

Or

Socket socket = new Socket("localhost", 5000)

- Here, the first argument represents the IP address of Server.
- The second argument represents the TCP Port. (It is a number that represents which application should run on a server.)

2**. Communication**

In order to communicate over a socket connection, **streams** are used for both input and output the data. After establishing a connection and sending the requests, you need to close the connection.

3**. Closing the connection**

The socket connection is closed explicitly once the message to the server is sent.

Now let’s see how to write a Java program to implement socket connection at client side.

```
1
2     // A Java program for a ClientSide
3
4     import java.net.*;
5
6     import java.io.*;
7
8     public class ClientProgram
9
10    {
11
12    // initialize socket and input output streams
13
14    private Socket socket = null;
15
16    private DataInputStream input = null;
17
18    private DataOutputStream out = null;
19
20    // constructor to put ip address and port
21
22    public Client(String address, int port)
23
24    {
25
26    // establish a connection
27
28    try
29
30    {
31
32    socket = new Socket(address, port);
33
34    System.out.println("Connected");
35
36    // takes input from terminal
37
38    input = new DataInputStream(System.in);
39
40    // sends output to the socket
41
42    out = new DataOutputStream(socket.getOutputStream());
43
44    }
    catch(UnknownHostException u)
    {
```

```
45
46      System.out.println(u);
47
48      }
49
50      catch(IOException i)
51
52      {
53
54      System.out.println(i);
55
56      }// string to read message from input
57
58      String line = "";
59
60      // keep reading until "Over" is input
        while (!line.equals("Over"))
        {
        try
        {
        line = input.readLine();
        out.writeUTF(line);
        }
        catch(IOException i)
        {
        System.out.println(i);
        }
        }
        // close the connection
        try
        {
        input.close();
        out.close();
```

```
socket.close();

}

catch(IOException i)

{

System.out.println(i);

}

}

public static void main(String args[]) {

Client client = new Client("127.0.0.1", 5000);

}

}
```

Now, let’s implement server-side programming and then arrive at the output.

Server Side Programming

Basically, the server will instantiate its object and wait for the client request. Once the client sends the request, the server will communicate back with the response.

In order to code the server-side application, you need two sockets and they are as follows:

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old socket for communication with the client.

Communication

getOutputStream() method is used to send the output through the socket.

Close the Connection

It is important to close the connection by closing the socket as well as input/output streams once everything is done.

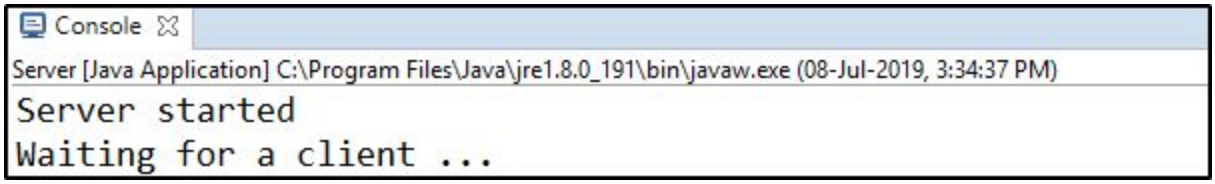
Now let’s see how to write a Java program to implement socket connection at server side.

```
1
2     // A Java program for a Serverside
3
4     import java.net.*;
5
6     import java.io.*;
7
8     public class Server
9
10    {//initialize socket and input stream
11
12    private Socket socket = null;
13
14    private ServerSocket server = null;
15
16    private DataInputStream in = null;
17
18    // constructor with port
19
20    public Server(int port)
21
22    {
23
24    // starts server and waits for a connection
25
26    try{
27
28    server = new ServerSocket(port);//create a socket object
29
30    System.out.println("Server started");
31
32    System.out.println("Waiting for a client ...");
33
34    socket = server.accept();
35
36    System.out.println("Client accepted");
37
38    // takes input from the client socket
39
40    in = new DataInputStream(new
41    BufferedInputStream(socket.getInputStream()));
42
43    String line = "";
44
45    // reads message from client until "Over" is sent
46
47    while (!line.equals("Over"))
48
49    {
50
```

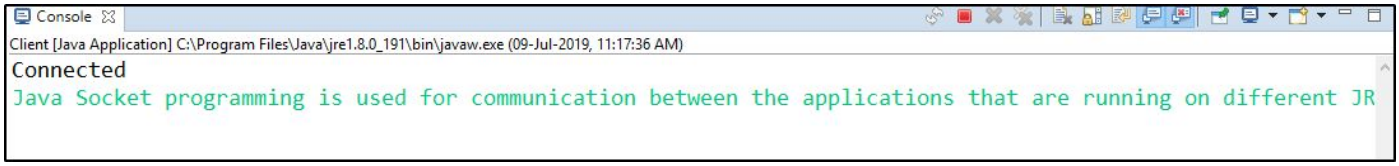
```
45
46      try
47
48      {
49
50      line = in.readUTF();
51
52      System.out.println(line);
        }
        catch(IOException i)
        {
        System.out.println(i);
        }}
        System.out.println("Closing connection");
        // close connection
        socket.close();
        in.close();
        }
        catch(IOException i){
        System.out.println(i);
        }}
        public static void main(String args[]){
        Server server = new Server(5000);
        }}
```

After configuring both client and server end, you can execute the server side program first. After that, you need to run client side program and send the request. As soon as the request is sent from the client end, server will respond back. Below snapshot represents the same.

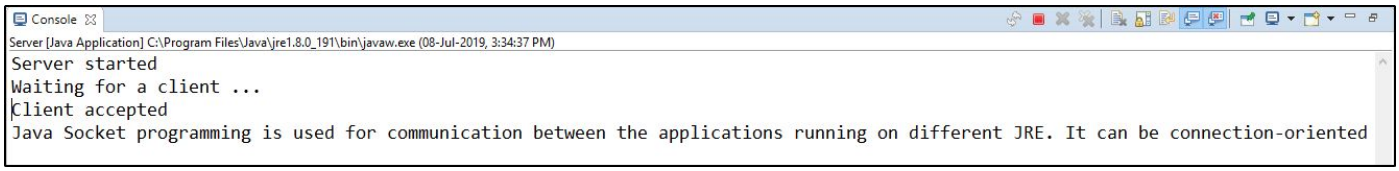
1. When you run the server side script, it will start and wait for the client to get started.



2. Next, the client will get connected and inputs the request in the form of a string.



3. When the client sends the request, the server will respond back.



One-way Communication

ClientSide:

```
package socketdemo;

import java.io.*;
import java.net.*;


public class ClientSocket {

	public static void main(String[] args) throws Exception
	{
		String ip="localhost";
		int port= 9955;; //0-1023 to 65535
		Socket s=new Socket(ip,port);

		String str="Vaishali";
```

```
                OutputStreamWriter os= new OutputStreamWriter(s.getOutputStream());
                PrintWriter out=new PrintWriter(os);
                out.write(str);
                os.flush();
                s.close();
        }
}
```

Serverside:

```
package socketdemo;

import java.io.*;
import java.net.*;

public class ServerSocketDemo {

        public static void main(String[] args) throws Exception
        {
                // TODO Auto-generated method stub
                System.out.println("Server is started");
                ServerSocket ss=new ServerSocket(9955);

                System.out.println("Server is waiting for client request");
                Socket s= ss.accept();

                System.out.println("Client Connected");

                InputStreamReader is= new InputStreamReader(s.getInputStream());
                BufferedReader br= new BufferedReader(is);
                String str= br.readLine();

                System.out.println("Client data:  " +str);
                ss.close();


        }

}
```

Output:
Serverside:

Server is started
Server is waiting for client request

ClientSide:

Server is started
Server is waiting for client request
Client Connected
Client data: Vaishali

Two_way Communication:

ServerSide:

```
package socketdemo;

import java.io.*;
import java.net.*;

public class ServerTwoWaySocket {

	public static void main(String[] args) throws Exception
	{
		// TODO Auto-generated method stub
		System.out.println("S: Server is started");
		ServerSocket ss=new ServerSocket(9950);

		System.out.println("S: Server is waiting for client request");
		Socket s= ss.accept();

		System.out.println("S: Client Connected");

		InputStreamReader is= new InputStreamReader(s.getInputStream());
		BufferedReader br= new BufferedReader(is);
		String str= br.readLine();

		System.out.println("S: Client data:  " +str);
		String nickName= str.substring(0,4);

		OutputStreamWriter os= new OutputStreamWriter(s.getOutputStream());
		PrintWriter out=new PrintWriter(os);
```

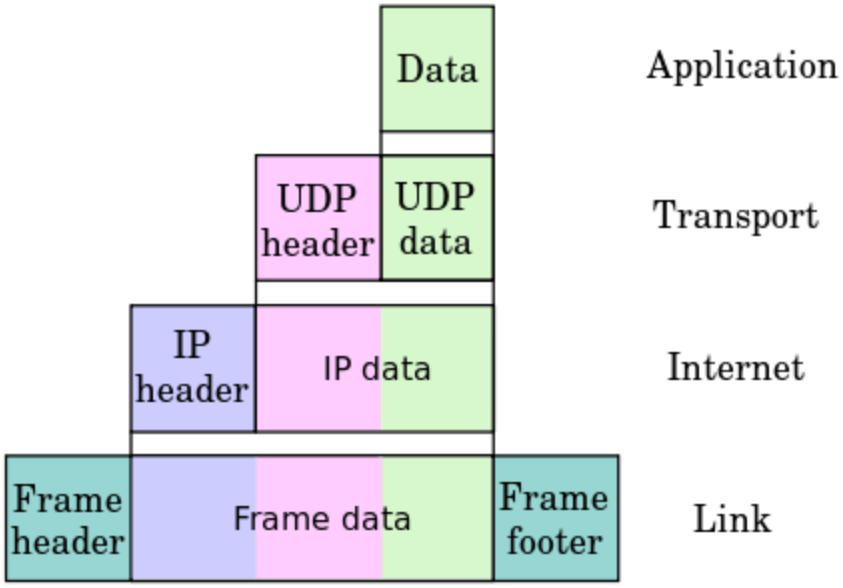
```
                out.println(nickName);
                out.flush();
                System.out.println("S: Data sent from Server to Client");



        }

}
```

Client side:

```
package socketdemo;
import java.io.*;
import java.net.*;
public class ClientTwoWaySocket {

                public static void main(String[] args) throws Exception
                {
                        String ip="localhost";
                        int port= 9950;; //0-1023 to 65535
                        //System.out.println("hi");
                        Socket s=new Socket(ip,port);
                        //System.out.println("hi1");

                        String str="Vaishali";

                        OutputStreamWriter os= new OutputStreamWriter(s.getOutputStream());
                        PrintWriter out=new PrintWriter(os);
                        out.write(str);
                        os.flush();

                        InputStreamReader is= new InputStreamReader(s.getInputStream());
                        BufferedReader br= new BufferedReader(is);
                        String nickName = br.readLine();

                        System.out.println("C: Data from server" +nickName);

                        s.close();
                }
        }
```

Differences between HTTP and Socket?

HTTP is an application protocol. It basically means that HTTP itself can't be used to transport information to/from a remote end point. Instead it relies on an underlying protocol which in HTTP's case is TCP.



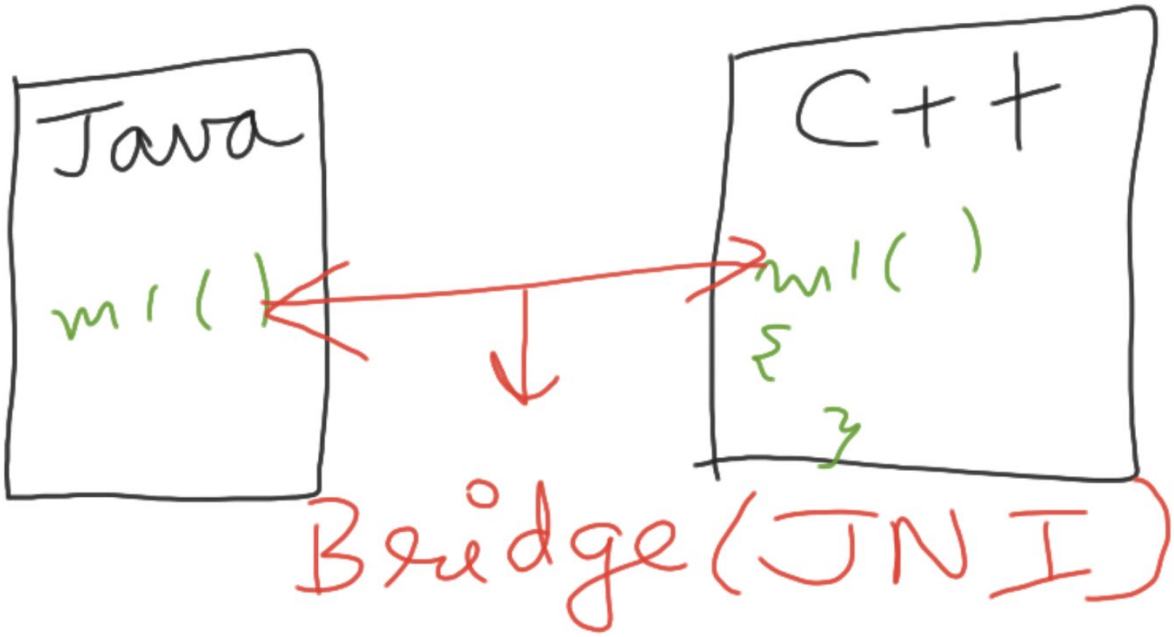
Sockets on the other hand are an API that most operating systems provide to be able to talk with the network. The socket API supports different protocols from the transport layer and down.
That means that if you would like to use TCP you use sockets. But you can also use sockets to communicate using HTTP, but then you have to decode/encode messages according to the HTTP specification (RFC2616). Since that can be a huge task for most developers we also got ready clients in our developer frameworks (like .NET), for instance the `WebClient` or the `HttpWebRequest` classes.

Java Native Interface Programming

> The JDK1.1 supports the Java Native Interface (JNI). On one hand, the JNI defines a standard naming and calling convention so that the Java Virtual Machine (VM) can locate and invoke your native methods. On the other hand, the JNI offers a set of standard interface functions. You call JNI functions from your native method code to do such things as access and manipulate Java objects, release Java objects, create new objects, call Java methods, and so on.

Java Native Interface (JNI) is used to support native codes. (Native codes are non java codes i.e C or C++)

JNI is like a bridge between byte code running in JVM and native code.



Components needed for JNI application:

1. Java Code: It should include at least one native method.
2. Native code: Actual logic of the native methods.
3. JNI header file: which is being included in C/C++ code.”javah” or “java -h”
4. C/C++ compiler

JNI Overview

- An interface that allows Java to interact with code written in another language
- Motivation for JNI
    - Code reusability
        - Reuse existing/legacy code with Java (mostly C/C++)
    - Performance
        - Native code used to be up to 20 times faster than Java, when running in interpreted mode
        - Modern JIT compilers (HotSpot) make this a moot point
    - Allow Java to tap into low level O/S, H/W routines
- JNI code is not portable!

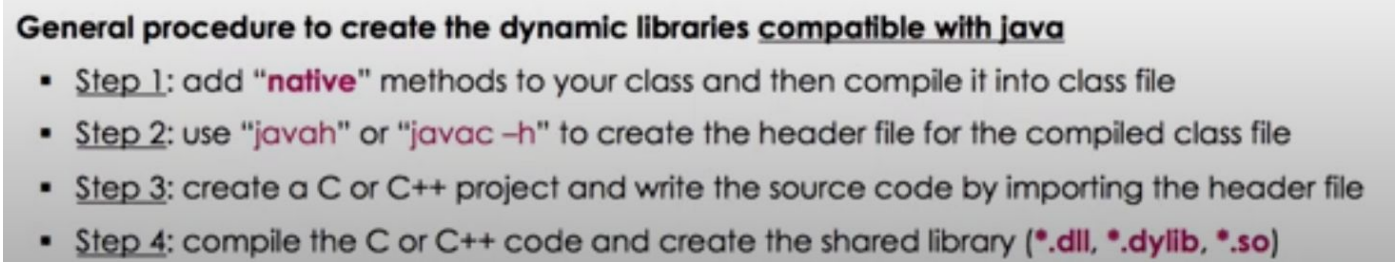
Note

JNI can also be used to invoke Java code from within natively-written applications - such as those written in C/C++.

In fact, the java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

16.2. JNI Components

- `javah` - JDK tool that builds C-style header files from a given Java class that includes `native` methods
  - Adapts Java method signatures to native function prototypes
- `jni.h` - C/C++ header file included with the JDK that maps Java types to their native counterparts
  - `javah` automatically includes this file in the application header files



16.3. JNI Development (Java)

- Create a Java class with native method(s): `public native void sayHi(String who, int times);`
- Load the library which implements the method: `System.loadLibrary("HelloImpl");`
- Invoke the native method from Java

For example, our Java code could look like this:

```
public class Hello {

  public native void sayHi(String who, int times); //❶


  static { System.loadLibrary("HelloImpl"); } //❷


  public static void main (String[] args) {

    Hello hello = new Hello();
```

```
    hello.sayHi(args[0], Integer.parseInt(args[1])); //❸

  }

}
```

❶

❸ The method `sayHi` will be implemented in C/C++ in separate file(s), which will be compiled into a library.

❷ The library filename will be called `libHelloImpl.so` (on Unix), `HelloImpl.dll` (on Windows) and `libHelloImpl.jnilib` (Mac OSX), but when loaded in Java, the library has to be loaded as `HelloImpl`.

Collections in Java

The Java collections framework provides a set of interfaces and classes to implement various data structures and algorithms.The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

Or

An objects that holds a collection (or a group, container) of objects

- Each item in a container is called an element
- Import java.util package

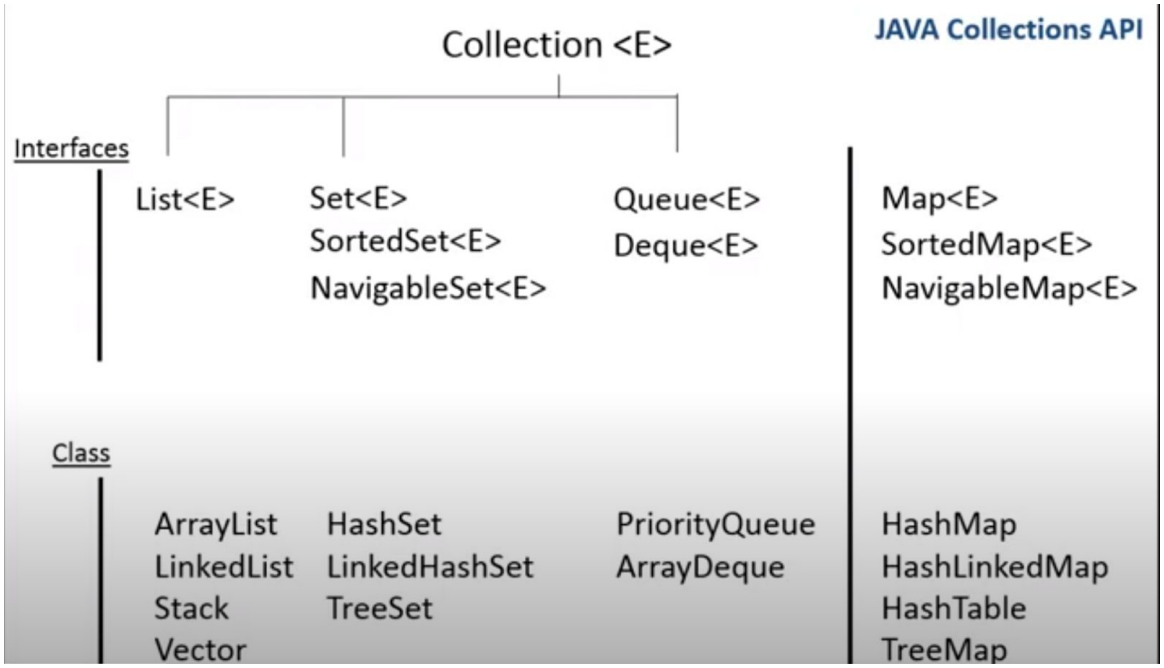
What is a framework in Java

- It provides readymade architecture.
- It represents a set of classes and interfaces.
- It is optional.

What is Collection framework

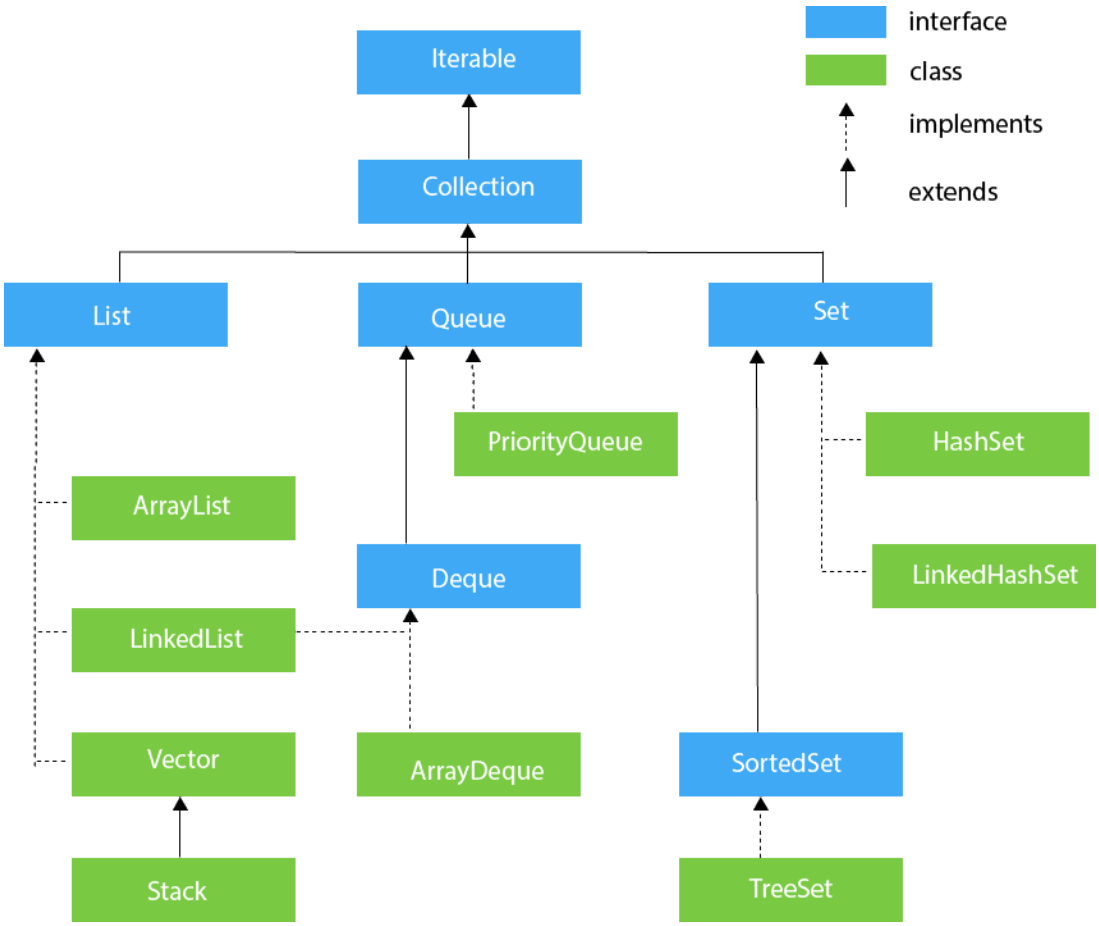
The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm



Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.

4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public boolean isEmpty()	It checks if collection is empty.
15	default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.

18	public boolean equals(Object element)	It matches two collections.
19	public int hashCode()	It returns the hash code number of the collection.

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList<data-type>();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list. The classes that implement the List interface are given below.

---

1. ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

Ex-1

```
import java.util.*;

public class TestJavaCollection1{

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

```
Collection list=new ArrayList();//Creating arraylist

list.add("Hello");//Adding object in arraylist

list.add("Hey");

list.add("Namastey");

list.add(9);

System.out.println(list);

}

}
```

Output:

```
[Hello, Hey, Namastey, Hi]
```

Ex-2

```
1. import java.util.*;
2. public class TestJavaCollection1{
3. public static void main(String args[]){
4. Collection list=new ArrayList();//Creating arraylist
5. list.add("Hello");//Adding object in arraylist
6. list.add("Hey");
7. list.add("Namastey");
8. list.add("Hi");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. } }
```

Output:

```
Hello

Hey

Namastey

Hi
```

Ex-3

```
import java.util.*;

public class TestJavaCollection1{
```

```
public static void main(String args[]){
List list=new ArrayList();//Creating arraylist
//Adding object in arraylist
list.add(4); //Integer v=new Integer(4), this is an object of Integer
list.add(6);
list.add(9);
list.add(2,2);
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
4
6
2
9
```

Ex-4

```
import java.util.*;
public class TestJavaCollection1{
public static void main(String args[]){
List list=new ArrayList();//Creating arraylist
list.add(4);//Adding object in arraylist
list.add(6);
list.add(9);
list.add(2,2);
//Traversing list through for loop

for (int i=0;i<list.size();i++)
{
   System.out.println(list.get(i));
}
}
}
```

Output:

```
4
6
2
9
```

Ex-5

```
import java.util.*;
public class TestJavaCollection1{
public static void main(String args[]){
List list=new ArrayList();//Creating arraylist
list.add(4);//Adding object in arraylist
list.add(6);
list.add(9);
list.add(2,2);
//Traversing list through for loop
for(Object o: list)
{
   System.out.println(o);
}
}
}
```

Output:

```
4
6
2
9
```

Concept of Generics:

```
List <E>list=new ArrayList<E>();
```

Ex

```
import java.util.*;
public class TestJavaCollection1{
```

```
public static void main(String args[]){
ArrayList <Integer> list=new ArrayList<Integer>();//Creating arraylist
list.add(4);//Adding object in arraylist
list.add(6);
list.add(9);
list.add(2,2);
//Traversing list through for loop
for(Integer o: list)
{
   System.out.println(o);
}
}
}
```

Output:

```
4
6
2
 9
```

Sorting in List:

```
Collections.sort(list);
```

Ex:

```
import java.util.*;
public class TestJavaCollection1{
public static void main(String args[]){
ArrayList <Integer> list=new ArrayList<Integer>();//Creating arraylist
list.add(4);//Adding object in arraylist
list.add(6);
list.add(2);
list.add(9);

Collections.sort(list);

//Traversing list through for loop
for(Integer o: list)
```

```
{
   System.out.println(o);
}
}
}
```

Output:

```
2
4
6
9
```

Reverse of a sorted List:

```
import java.util.*;
public class TestJavaCollection1{
public static void main(String args[]){
ArrayList <Integer> list=new ArrayList<Integer>();//Creating arraylist
list.add(4);//Adding object in arraylist
list.add(6);
list.add(2);
list.add(9);
Collections.sort(list);
Collections.reverse(list);
for(Integer o: list)
{
   System.out.println(o);
}
}
}
```

Output:

```
9
6
4
 2
```

Note: Original list is changed, so we can say that list is mutable.

LinkedList

LinkedList implements the **Collection interface.** It uses a **doubly linked list** internally to store the elements. It can **store the duplicate elements.** It maintains the **insertion order** and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Ex-1

```
import java.util.*;

public class LinkedListDemo{
public static void main(String args[]){
 LinkedList<String> ll = new LinkedList<String>();
 //add
 ll.add("Test");
 ll.add("java");
 ll.add("Link");
 ll.add("list");
 //print
 System.out.println("content of linkedlist" + ll);
}
}
```

Output:

```
 content of linkedlist[Test, java, Link, list]
```

Ex-3

```
import java.util.*;
public class LinkedListDemo{
public static void main(String args[]){
 LinkedList<String> ll = new LinkedList<String>();
 //add
 ll.add("Test");
 ll.add("java");
 ll.add("Link");
 ll.add("list");
//print
 System.out.println("content of linkedlist" + ll);
 //add first
```

```
 ll.addFirst("hey");
 //add Last
 ll.addLast("end");
 System.out.println("content of linkedlist" + ll);
}
}
```

Output:

```
content of linkedlist[Test, java, Link, list]
 content of linkedlist[hey, Test, java, Link, list, end]
```

Ex-4

```
import java.util.*;
 public class LinkedListDemo{
 public static void main(String args[]){
  LinkedList<String> ll = new LinkedList<String>();
  //add
  ll.add("Test");
  ll.add("java");
  ll.add("Link");
  ll.add("list");
  //print
  System.out.println("content of linkedlist" + ll);
  //add first
  ll.addFirst("hey");
  //add Last
  ll.addLast("end");
  System.out.println("content of linkedlist" + ll);
  //get
  System.out.println(ll.get(0));
  ll.set(0,"hello");
   System.out.println(ll.get(0));
 }
 }
```

Output:

```
content of linkedlist[Test, java, Link, list]
content of linkedlist[hey, Test, java, Link, list, end]
```

hey
hello

Ex-5

```
import java.util.*;
public class LinkedListDemo{
public static void main(String args[]){
 LinkedList<String> ll = new LinkedList<String>();
 //add
 ll.add("Test");
 ll.add("java");
 ll.add("Link");
 ll.add("list");
 //print
 System.out.println("content of linkedlist" + ll);
 //add first
 ll.addFirst("hey");
 //add Last
 ll.addLast("end");
 System.out.println("content of linkedlist" + ll);
 //get
 System.out.println(ll.get(0));
 ll.set(0,"hello");
  System.out.println(ll.get(0));
 //remove first  and last element
 ll.removeFirst();
 ll.removeLast();
 System.out.println("content of linkedlist" + ll);
 ll.remove(2);
 System.out.println("content of linkedlist" + ll);
}
}
```

Output:

content of linkedlist[Test, java, Link, list]
content of linkedlist[hey, Test, java, Link, list, end]
hey
hello

```
content of linkedlist[Test, java, Link, list]
content of linkedlist[Test, java, list]
```

Ex-6

```
import java.util.*;
public class LinkedListDemo{
public static void main(String args[]){
 LinkedList<String> ll = new LinkedList<String>();
 //add
 ll.add("Test");
 ll.add("java");
 ll.add("Link");
 ll.add("list");
 //print
 System.out.println("content of linkedlist" + ll);
 //add first
 ll.addFirst("hey");
 ll.add(2,"exampple");
 //add Last
 ll.addLast("end");
 System.out.println("content of linkedlist" + ll);
 //loop
 System.out.println("***For Loop***");
 for(int n=0;n<ll.size();n++)
 {
    System.out.println(ll.get(n));
 }

 System.out.println("***For each/advance for Loop***");
 for(String str:ll)
 {
    System.out.println(str);
 }
 System.out.println("***using iterator***");
 Iterator<String> it=ll.iterator();
 while(it.hasNext())
 {
    System.out.println(it.next());
```

```
 }
 int num=0;
 System.out.println("***while loop***");
 while(ll.size()>num)
 {
    System.out.println(ll.get(num));
    num++;
 }
}
}
```

Output:

```
content of linkedlist[Test, java, Link, list]
content of linkedlist[hey, Test, exampple, java, Link, list, end]
***For Loop***
hey
Test
exampple
java
Link
list
end
***For each/advance for Loop***
hey
Test
exampple
java
Link
list
end
***using iterator***
hey
Test
exampple
java
Link
list
end
***while loop***
hey
```

Test
exampple
java
Link
list
end

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Vector	ArrayList
Present since the initial version of Java(JDK 1.0 version).	Introduced in Java since JDK 1.2
Vector is a legacy class of Java.	ArrayList is a part of the Java Collections Framework.
Vector grows double its size when its capacity is reached.	ArrayList grows by half the size when its capacity is reached.
Vector methods are synchronized.	ArrayList is not synchronized.
Vector uses Enumerator and Iterator for traversing.	ArrayList uses only Iterator.
Vector operations are slower.	ArrayList is faster.
Vector has increment size using which vector size can be increased.	ArrayList does not provide increment size.
Vector is thread-safe which means using Vector from multiple threads is permitted and is safe.	ArrayList is not thread-safe.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection3{
3. public static void main(String args[]){
4. Vector<String> v=new Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
```

```
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Ex-2

```
import java.util.*;
public class TestJavaCollection1{
public static void main(String args[]){
Vector<Integer> myVector= new Vector<Integer>();
myVector.add(10);
myVector.add(4);
myVector.add(12);
myVector.add(8);
myVector.add(5);

System.out.println(myVector);
System.out.println(myVector.get(2));
myVector.remove(3);
System.out.println(myVector);

Vector<Integer> yourVector= new Vector<Integer>();
yourVector.add(10);
yourVector.add(14);
myVector.addAll(yourVector);
System.out.println(myVector);
}
}
```

Output:

```
[10, 4, 12, 8, 5]
12
[10, 4, 12, 5]
 [10, 4, 12, 5, 10, 14]
```

Stack

The stack is the subclass of Vector. It implements the **last-in-first-out** data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection4{
3. public static void main(String args[]){
4. Stack<String> stack = new Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the **first-in-first-out order.** It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like **PriorityQueue, Deque, and ArrayDeque** which implements the Queue interface.

Queue interface can be instantiated as:

```
1. Queue<String> q1 = new PriorityQueue();
2. Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection5{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
```

```
20. while(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

Ex-2

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
LinkedList<Integer> queue=new LinkedList<Integer>();
queue.add(10);
queue.add(7);
queue.add(2);
queue.add(5);
System.out.println(queue);
System.out.println("head:"+queue.peek());
queue.poll();
System.out.println(queue);

PriorityQueue<Integer> pq=new PriorityQueue<Integer>();
pq.add(10);
pq.add(7);
pq.add(2);
pq.add(5);
System.out.println(pq);
pq.poll();
System.out.println("head:"+pq.peek());
System.out.println(pq);

}
}
```

Output:

```
[10, 7, 2, 5]
head:10
[7, 2, 5]
[2, 5, 7, 10]
head:5
[5, 10, 7]
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

```
1. Deque d = new ArrayDeque();
```

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection6{
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. for (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }
```

Output:

Gautam
Karan
Ajay

Ex-2:

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      ArrayDeque<String> animals= new ArrayDeque<>();
      animals.add("Dog");
      animals.add("Cat");
      animals.add("Horse");
      System.out.println("ArrayDeque: " + animals);

      // Get the first element
      String firstElement = animals.getFirst();
      System.out.println("First Element: " + firstElement);

      // Get the last element
      String lastElement = animals.getLast();
      System.out.println("Last Element: " + lastElement);
   }
}
```

Output:

```
ArrayDeque: [Dog, Cat, Horse]
First Element: Dog
 Last Element: Horse
```

Ex-3

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      ArrayDeque<String> animals= new ArrayDeque<>();
      animals.add("Dog");
      animals.add("Cat");
      animals.add("Cow");
      animals.add("Horse");
      System.out.println("ArrayDeque: " + animals);

      // Using remove()
      String element = animals.remove();
      System.out.println("Removed Element: " + element);

      System.out.println("New ArrayDeque: " + animals);
```

```
        // Using removeFirst()
        String firstElement = animals.removeFirst();
        System.out.println("Removed First Element: " + firstElement);
         System.out.println("New ArrayDeque: " + animals);

        // Using removeLast()
        String lastElement = animals.removeLast();
        System.out.println("Removed Last Element: " + lastElement);
         System.out.println("New ArrayDeque: " + animals);
    }
}
```

Output:
ArrayDeque: [Dog, Cat, Cow, Horse]
Removed Element: Dog
New ArrayDeque: [Cat, Cow, Horse]
Removed First Element: Cat
New ArrayDeque: [Cow, Horse]
Removed Last Element: Horse
New ArrayDeque: [Cow]

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by **HashSet, LinkedHashSet, and TreeSet.**

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

Set Operations

The Java `Set` interface allows us to perform basic mathematical set operations like union, intersection, and subset.

- **Union** - to get the union of two sets `x` and `y`, we can use `x.addAll(y)`
- **Intersection** - to get the intersection of two sets `x` and `y`, we can use `x.retainAll(y)`
- **Subset** - to check if `x` is a subset of `y`, we can use `y.containsAll(x)`

HashSet

HashSet class implements Set Interface. It represents the collection that uses a **hash table** for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Ex-1

```
import java.util.*;

public class TestJavaCollection7{

public static void main(String args[]){

//Creating HashSet and adding elements

HashSet<Integer> set=new HashSet<Integer>();

set.add(4);

set.add(8);

set.add(9);

set.add(2);

//Traversing elements

Iterator<Integer> itr=set.iterator();

while(itr.hasNext()){

System.out.println(itr.next());

}

}

}
```

Output:

```
2
4
8
9
```

Ex-2

```
import java.util.*;
public class TestJavaCollection7{
```

```
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<Integer> set=new HashSet<Integer>();
set.add(4);
set.add(8);
set.add(9);
set.add(2);
set.add(9); //adding redundant data
//Traversing elements
Iterator<Integer> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
2
4
8
9
//But only one 9 has been added in the output.
```

Ex-3 //TreeSet

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
TreeSet<Integer> set=new TreeSet<Integer>();
set.add(4);
set.add(8);
set.add(9);
set.add(2);
set.add(9);
//Traversing elements
Iterator<Integer> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

2
4
8
9

Ex-

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
Set<Integer> set=new HashSet<Integer>();
set.add(4);
set.add(8);
set.add(9);
set.add(8);
set.add(2);
System.out.println(set);
//Traversing elements
Iterator<Integer> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
Set<Integer> tset=new TreeSet<Integer>();
tset.add(4);
tset.add(8);
tset.add(9);
tset.add(8);
tset.add(2);
System.out.println(tset);
//Traversing elements
Iterator<Integer> itr1=set.iterator();
while(itr1.hasNext()){
System.out.println(itr1.next());
}
```

```
}
}
```

Output:

```
[2, 4, 8, 9]
2
4
8
9
[2, 4, 8, 9]
2
4
8
9
```

Ex: //Union of two sets

```
import java.util.Set;
import java.util.HashSet;

public class Main {

    public static void main(String[] args) {
        // Creating a set using the HashSet class
        Set<Integer> set1 = new HashSet<>();

        // Add elements to the set1
        set1.add(10);
        set1.add(2);
        set1.add(3);
        System.out.println("Set1: " + set1);

        // Creating another set using the HashSet class
        Set<Integer> set2 = new HashSet<>();
```

```
        // Add elements
        set2.add(1);
        set2.add(6);
        set2.add(2);
        System.out.println("Set2: " + set2);

        // Union of two sets
        set2.addAll(set1);
        System.out.println("Union is: " + set2);
    }
}
```

Output:

Set1: [2, 3, 10]

Set2: [1, 2, 6]

Union is: [1, 2, 3, 6, 10]//S2

Ex- //Intersection of two sets using RetainAll

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<Integer> primeNumbers = new HashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        System.out.println("HashSet1: " + primeNumbers);

        HashSet<Integer> evenNumbers = new HashSet<>();
        evenNumbers.add(2);
        evenNumbers.add(4);
        System.out.println("HashSet2: " + evenNumbers);

        // Intersection of two sets
        evenNumbers.retainAll(primeNumbers);
```

```
        System.out.println("Intersection is: " + evenNumbers);
    }
}
```

Output:

```
HashSet1: [2, 3]
HashSet2: [2, 4]
Intersection is: [2]
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
1. import java.util.*;
2. public class TestJavaCollection8{
3. public static void main(String args[]){
4. LinkedHashSet<String> set=new LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

```
1. SortedSet<data-type> set = new TreeSet();
```

TreeSet

Java TreeSet class implements the Set interface that uses a **tree** for storage. Like HashSet, TreeSet also contains **unique elements.** However, the access and retrieval time of TreeSet is quite fast. **The elements in TreeSet stored in ascending order.**

Consider the following example:

```
1. import java.util.*;
2. public class TestJavaCollection9{
3. public static void main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=new TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Output:

```
Ajay
Ravi
```

Vijay

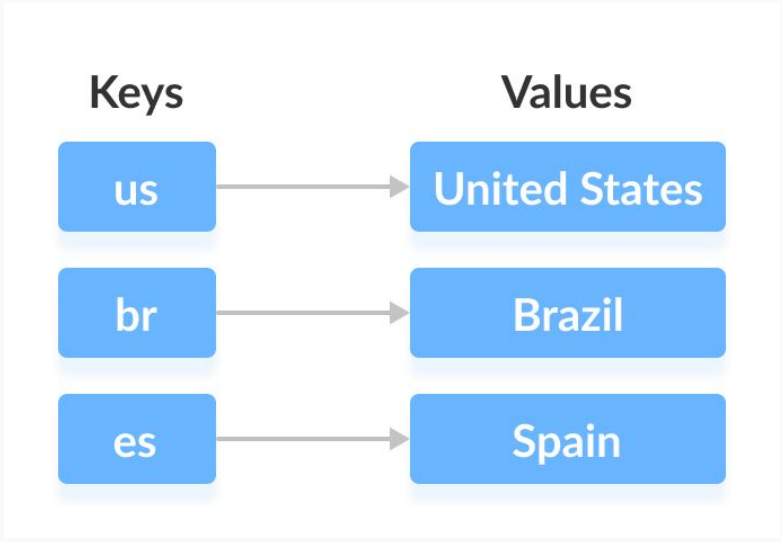
```
import java.util.HashSet;

class Main {
    public static void main(String[] args) {
        HashSet<Integer> primeNumbers = new HashSet<>();
        primeNumbers.add(2);
        primeNumbers.add(3);
        primeNumbers.add(5);
        System.out.println("HashSet1: " + primeNumbers);

        HashSet<Integer> oddNumbers = new HashSet<>();
        oddNumbers.add(1);
        oddNumbers.add(3);
        oddNumbers.add(5);
        System.out.println("HashSet2: " + oddNumbers);

        // Difference between HashSet1 and HashSet2
        primeNumbers.removeAll(oddNumbers);
        System.out.println("Difference : " + primeNumbers);
    }
}
```

Java Map Interface

A map contains values on the basis of key, i.e. **key and value pair**. Each key and value pair is known as an **entry**. A Map contains unique keys.

{0=”abc”,1=”xyz”, 2=”mnp”, 3=”mnp”}

A Map is useful if you have to search, update or delete elements on the basis of a key. A map cannot contain duplicate keys. And, each key is associated with a single value.



We can access and modify values using the keys associated with them.

In the above diagram, we have values: `United States`, `Brazil`, **and** `Spain`. And we have corresponding keys: `us`, `br`, **and** `es`.

Now, we can access those values using their corresponding keys.

Note: The `Map` interface maintains 3 different sets:

- the set of keys
- the set of values
- the set of key/value associations (mapping).

How to use Map?

In Java, we must import the `java.util.Map` package in order to use `Map`. Once we import the package, here's how we can create a map.

```
// Map implementation using HashMap
Map<Key, Value> numbers = new HashMap<>();
Map<String, String> numbers = new HashMap<>();
```

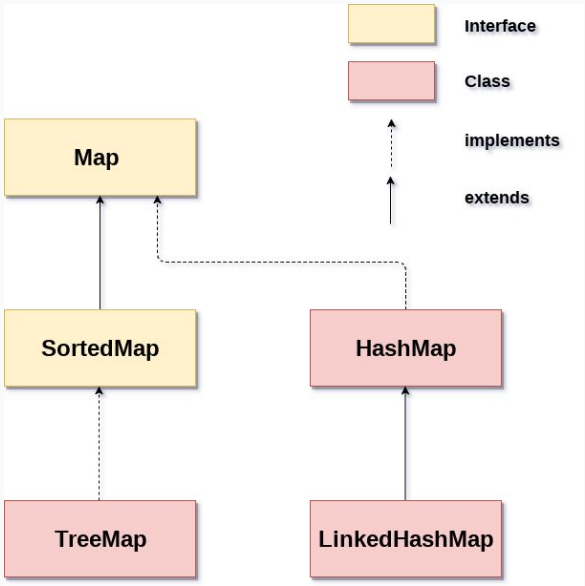
In the above code, we have created a `Map` named `numbers`. We have used the `HashMap` class to implement the `Map` interface.

Here,

- `Key` - a unique identifier used to associate each element (value) in a map
- `Value` - elements associated by keys in a map

Map Properties:

1. They contain values based on the key
2. They are not ordered
3. “Key” should be unique
4. “Value” can be duplicate

5. 

Useful methods of Map interface

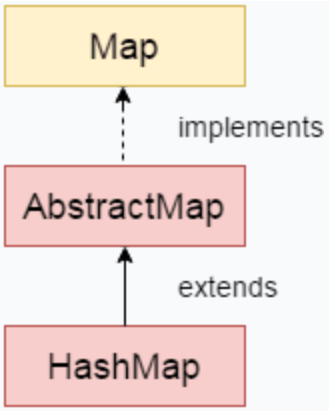
Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V putIfAbsent(K key, V value)	It inserts the specified value with the specified key in the map only if it is not already specified.
V remove(Object key)	It is used to delete an entry for the specified key.

boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
Set<Map.Entry<K,V>> entrySet()	It returns the Set view containing all the keys and values.
void clear()	It is used to reset the map.
V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)	It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null.
V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)	It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null.

boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
void forEach(BiConsumer<? super K,? super V> action)	It performs the given action for each entry in the map until all entries have been processed or the action throws an exception.
V get(Object key)	This method returns the object that contains the value associated with the key.
V getOrDefault(Object key, V defaultValue)	It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key.
int hashCode()	It returns the hash code value for the Map

boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
void replaceAll(BiFunction<? super K,? super V,? extends V> function)	It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
Collection values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

Java HashMap



Java **HashMap** class implements the Map interface which allows us *to* ***store key and value pair***, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the **java.util** package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as `HashMap<K,V>`, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

Points to remember

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.

- Java HashMap is non synchronized.
- Java HashMap maintains no order.

Map Interface
Ex-1: //old style

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
Map<String, String> map=new HashMap<>();
map.put("name","vaishali");
map.put("college","adgitm");
map.put("subject","java");
System.out.println(map);

Set<String> keys= map.keySet();
for(String key: keys)
{
   System.out.println(key +" "+ map.get(key));
}}
```

Output:

```
{college=adgitm, subject=java, name=vaishali}
college adgitm
subject java
name vaishali
```

Ex-2 :// new style

```
import java.util.*;
public class MapExample2{
 public static void main(String args[]){
  Map<Integer,String> map=new HashMap<Integer,String>();
  map.put(100,"Amit");
  map.put(101,"Vijay");
  map.put(102,"Rahul");
  //Elements can traverse in any order
```

```
  for(Map.Entry m:map.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Output:

```
100 Amit
101 Vijay
102 Rahul
```

Ex-3:

```
import java.util.Map;
import java.util.HashMap;

public class Main {

    public static void main(String[] args) {
        // Creating a map using the HashMap
        Map<String, Integer> numbers = new HashMap<>();

        // Insert elements to the map
        numbers.put("One", 1);
        numbers.put("Two", 2);
        System.out.println("Map: " + numbers);

        // Access keys of the map
        System.out.println("Keys: " + numbers.keySet());

        // Access values of the map
        System.out.println("Values: " + numbers.values());

        // Access entries of the map
        System.out.println("Entries: " + numbers.entrySet());

        // Remove Elements from the map
        int value = numbers.remove("Two");
        System.out.println("Removed Value: " + value);
    }
}
```

Output:

```
Map: {One=1, Two=2}
```

```
Keys: [One, Two]
Values: [1, 2]
Entries: [One=1, Two=2]
Removed Value: 2
```

Ex-3: //Entry set

```
import java.util.Map;
import java.util.HashMap;

public class Main {

    public static void main(String[] args) {
        // Creating a map using the HashMap
        Map<String, Integer> numbers = new HashMap<>();

        // Insert elements to the map
        numbers.put("Two", 2);
        numbers.put("Three",3);
        numbers.put("One", 1);

        System.out.println("Map: " + numbers);

        // Access keys of the map
        System.out.println("Keys: " + numbers.keySet());

        // Access values of the map
        System.out.println("Values: " + numbers.values());

        // Access entries of the map
        System.out.println("Entries: " + numbers.entrySet());

        // Remove Elements from the map
        int value = numbers.remove("Two");//2
        System.out.println("Removed Value: " + value);
    }
```

```
}
```

Output:

```
Map: {One=1, Two=2, Three=3}
Keys: [One, Two, Three]
Values: [1, 2, 3]
Entries: [One=1, Two=2, Three=3]
 Removed Value: 2
```

Ex-4://Search in Entry Set

```
import java.util.HashMap;
import java.util.Map.Entry;

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

    // create a hashmap
    HashMap<String, Integer> numbers = new HashMap<>();
    numbers.put("One", 1);
    numbers.put("Two", 2);
    numbers.put("Three", 3);
    System.out.println("HashMap: " + numbers);

    // value whose key is to be searched
    Integer value = 3;

    // iterate each entry of hashmap
    for(Entry<String, Integer> entry: numbers.entrySet()) {

      // if give value is equal to value from entry
      // print the corresponding key
      if(entry.getValue() == value) {
        System.out.println("The key for value " + value + " is " + entry.getKey());
        break;
      }
    }
  }
}
```

Output:

HashMap: {One=1, Two=2, Three=3}
The key for value 3 is Three

In the above example, we have created a hashmap named **`numbers`**. Here, we want to get the key for the value 3. Notice the line,

```
Entry<String, Integer> entry : numbers.entrySet()
```

Here, the `entrySet()` method returns a set view of all the entries.

- entry.getValue() - get value from the entry
- entry.getKey() - get key from the entry

Inside the if statement we check if the value from the entry is the same as the given value. And, for matching value, we get the corresponding key.

List Vs Set Vs Map

1) **Duplicity:** List allows duplicate elements. Any number of duplicate elements can be inserted into the list without affecting the same existing values and their indexes.

Set doesn't allow duplicates. Set and all of the classes which implements Set interface should have unique elements.

Map stored the elements as key & value pair. Map doesn't allow duplicate keys while it allows duplicate values.

2) **Null values:** List allows any number of null values.

Set allows single null value at most.

Map can have single null key at most and any number of null values.

3) **Order:** List and all of its implementation classes maintains the insertion order.

Set doesn't maintain any order; still few of its classes sort the elements in an order such as LinkedHashSet maintains the elements in insertion order.

Similar to Set, Map also doesn't stores the elements in an order, however few of its classes does the same. For e.g. TreeMap sorts the map in the ascending order of keys and LinkedHashMap sorts the elements in the insertion order, the order in which the elements got added to the LinkedHashMap.

4) **Commonly used classes:**

List: ArrayList, LinkedList etc.

Set: HashSet, LinkedHashSet, TreeSet, SortedSet etc.

Map: HashMap, TreeMap etc

When to use List, Set and Map in Java?

1) If you do not want to have duplicate values in the database then **Set** should be your first choice as all of its classes do not allow duplicates.

2) If there is a need of frequent search operations based on the index values then List (ArrayList) is a better choice.

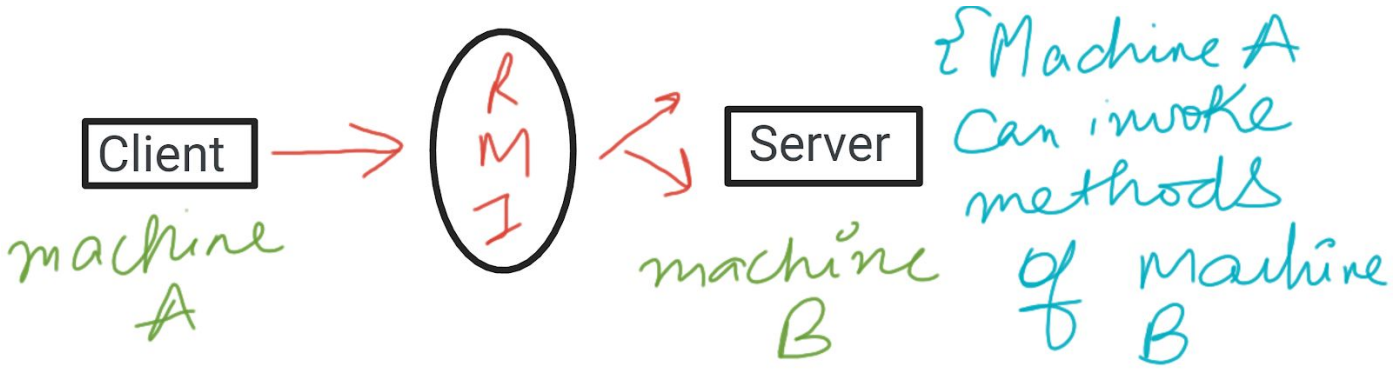
3) If there is a need of maintaining the insertion order then also the **List** is a preferred collection interface.

4) If the requirement is to have the key & value mappings in the database then **Map** is your best bet.

Java RMI - Introduction

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

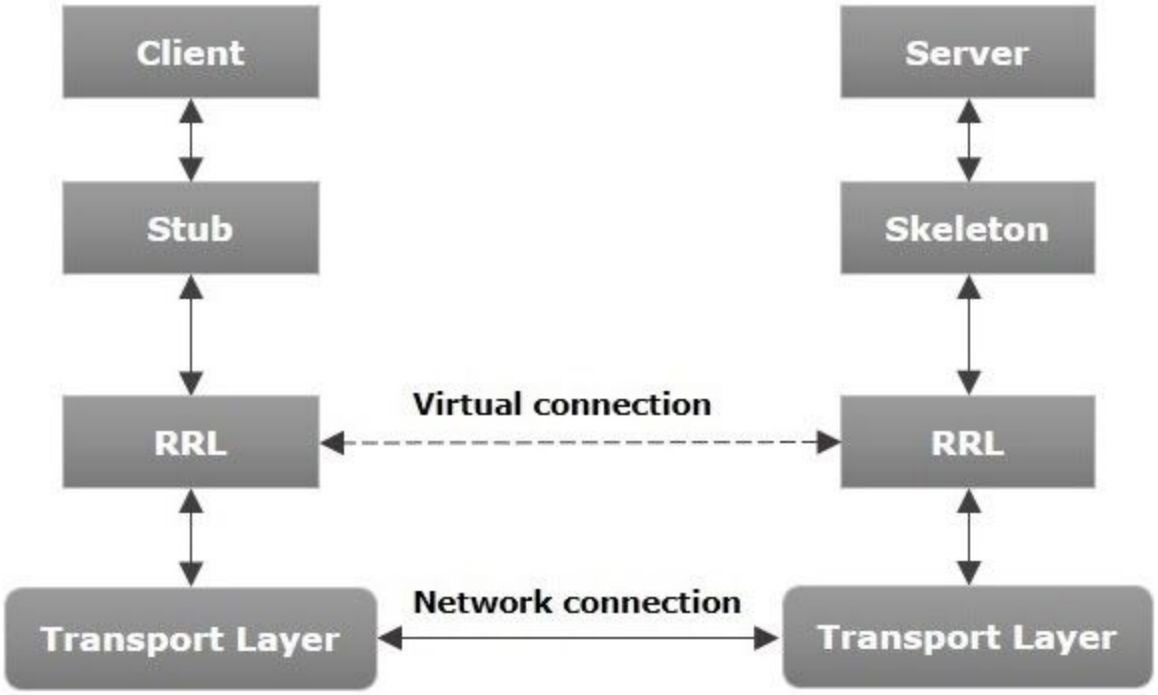


Architecture of an RMI Application

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

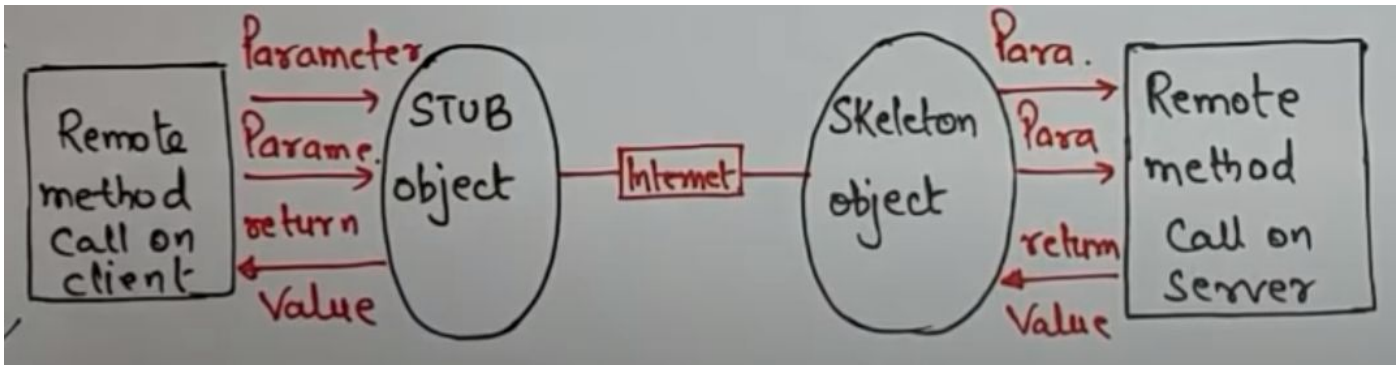
The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

- Transport Layer − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- Stub − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- Skeleton − This is the object which resides on the server side. stub communicates with this skeleton to pass request to the remote object.
- RRL(Remote Reference Layer) − It is the layer which manages the references made by the client to the remote object.

Working of an RMI Application



The following points summarize how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshalling.

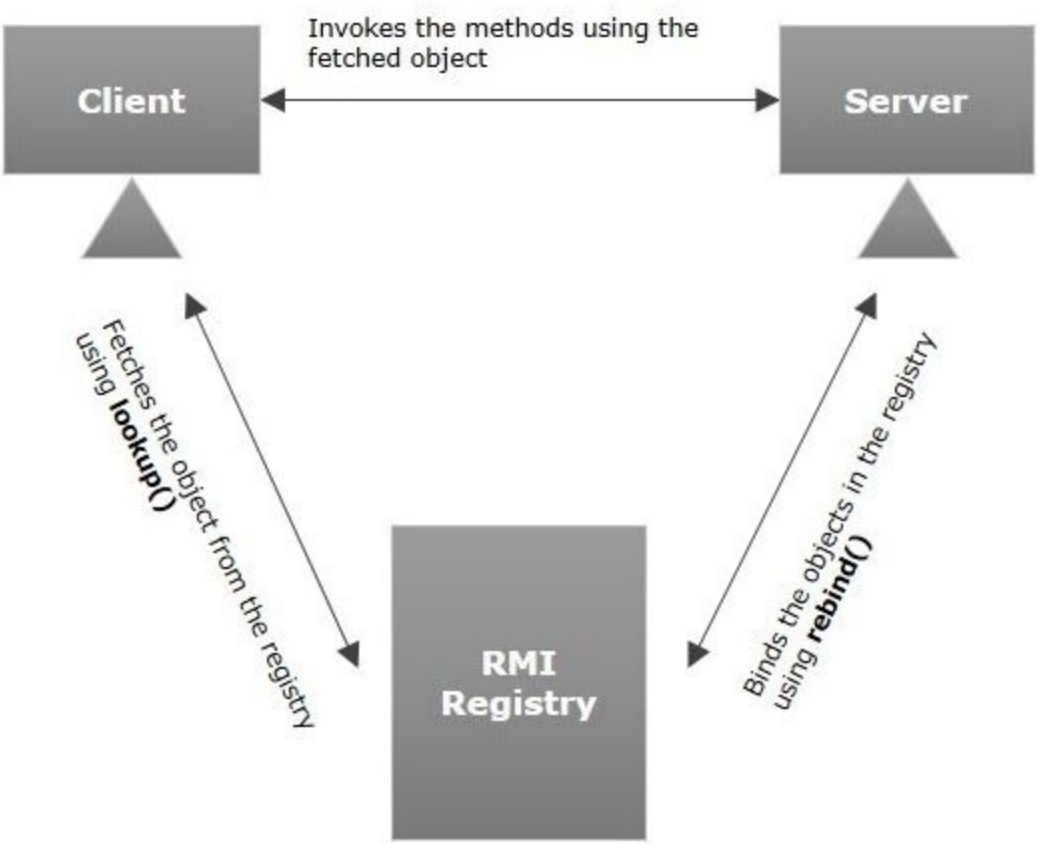
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method).

The following illustration explains the entire process −



Goals of RMI

Following are the goals of RMI −

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

Java Packages & API

A package in Java is used to group related classes. Think of it as a **folder** in a file directory. **We use packages to avoid name conflicts, and to write a better maintainable code.** Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

The library is divided into **packages and classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the `Scanner` class, which is used to get user input, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Example

Using the `Scanner` class to get user input:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

Import a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

Example

```
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system **directory** to store them. Just like folders on your computer:

Example

```
└── root
    └── mypack
        └── MyPackageClass.java
```

To create a package, use the `package` keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Save the file as MyPackageClass.java, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package.

The `-d` keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign "`.`", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the MyPackageClass.java file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

Miscellaneous (Extra)

Java Swing Tutorial

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides **platform-independent** and **lightweight components.**

The **javax.swing** package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller	Swing **follows MVC**.

	acts as an interface between model and view.	

What is JFC

The **Java Foundation Classes (JFC)** are a set of GUI components which simplify the development of desktop applications.

JFrame – A frame is an instance of JFrame. Frame is a window that can have title, border, menu, buttons, text fields and several other components. A Swing application must have a frame to have the components added to it.

JPanel – A panel is an instance of JPanel. A frame can have more than one panels and each panel can have several components. You can also call them parts of Frame. Panels are useful for grouping components and placing them to appropriate locations in a frame.

JLabel – A label is an instance of JLabel class. A label is unselectable text and images. If you want to display a string or an image on a frame, you can do so by using labels. In the above example we wanted to display texts “User” & “Password” just before the text fields , we did this by creating and adding labels to the appropriate positions.

JTextField – Used for capturing user inputs, these are the text boxes where user enters the data.

JPasswordField – Similar to text fields but the entered data gets hidden and displayed as dots on GUI.

JButton – A button is an instance of JButton class. In the above example we have a button “Login”.

Java vs Javax:

Originally, everything that was part of the standard API was part of the java package, whereas everything that was not part of the standard API was released under the package name **javax.** Hence, packages essential to the API was java, while javax contained the extensions to the API. It can even be said that javax, is just java with an **x, which stands for extension.**

Java Swing Examples

There are two ways to create a frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

We can write the code of swing inside the main(), constructor or any other method.

Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```
1. import javax.swing.*;
2. public class FirstSwingExample {
3. public static void main(String[] args) {
4. JFrame f=new JFrame();//creating instance of JFrame
5. 
6. JButton b=new JButton("click");//creating instance of JButton
7. b.setBounds(130,100,100, 40);//x axis, y axis, width, height
8. 
9. f.add(b);//adding button in JFrame
10. 
11. f.setSize(400,500);//400 width and 500 height
12. f.setLayout(null);//using no layout managers
13. f.setVisible(true);//making the frame visible
14. }
15. }
```

Example of Swing by Association inside constructor

We can also write all the codes of creating JFrame, JButton and method call inside the java constructor.

File: Simple.java

```
1. import javax.swing.*;
2. public class Simple {
3. JFrame f;
4. Simple(){
5. f=new JFrame();//creating instance of JFrame
6.
7. JButton b=new JButton("click");//creating instance of JButton
8. b.setBounds(130,100,100, 40);
```

```
9.
10. f.add(b);//adding button in JFrame
11.
12. f.setSize(400,500);//400 width and 500 height
13. f.setLayout(null);//using no layout managers
14. f.setVisible(true);//making the frame visible
15. }
16.
17. public static void main(String[] args) {
18. new Simple();
19. }
20. }
```

The setBounds(int xaxis, int yaxis, int width, int height)is used in the above example that sets the position of the button.

Simple example of Swing by inheritance

We can also inherit the JFrame class, so there is no need to create the instance of JFrame class explicitly.

File: Simple2.java

```
1. import javax.swing.*;
2. public class Simple2 extends JFrame{//inheriting JFrame
3. JFrame f;
4. Simple2(){
5. JButton b=new JButton("click");//create button
6. b.setBounds(130,100,100, 40);
7.
8. add(b);//adding button on frame
9. setSize(400,500);
10. setLayout(null);
11. setVisible(true);
12. }
13. public static void main(String[] args) {
14. new Simple2();
```

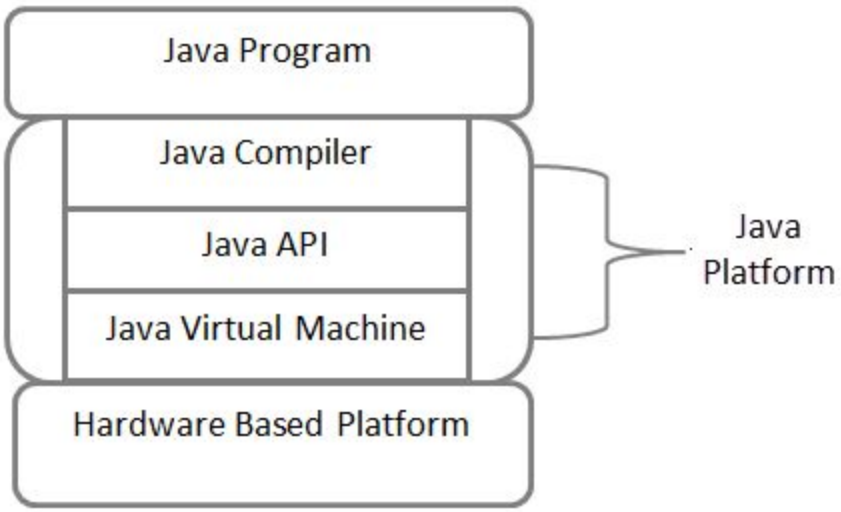
```
15. }}
```

Introduction to API in Java

An API can be described as a way to enable computers to possess a common interface, to allow them to communicate with each other. Java Application Programming Interface (API) is the area of Java development kit (JDK). An API includes classes, interfaces, packages and also their methods, fields, and constructors.

All these built-in classes give benefits to the programmer. Only programmers understand how to apply that class. A user interface offers the basic user interaction among user and computer, in the same manner, the API works as an application program interface which gives connection amongst the software as well as the consumer. API includes classes and packages which usually assist a programmer to minimize the lines of a program.

Understanding API in Java



Java Development Kit (JDK) is usually made up of three fundamental components, as per below:

1. Java compiler

2. Java Virtual Machine (JVM)

3. Java Application Programming Interface (API)

- The Java API added to the JDK, explains the function of every element. In Java programming, several components are pre-created as well as, widely used.
- Hence, the programmer can make use of a prewritten program with the Java API. After mentioning the obtainable API classes as well as, packages, the programmer quickly creates the required program classes and packages to get executed.
- The Java API is a vital element of the JDK and identifies features of every element. Although Programming in Java, the component is already produced and done it. Through Java, API coder can simply make use of the pre-written program.
- The programmers initially declare the classes and packages, then this coder can simply use the application program of classes and packages to get executed.