## 1.Write about Java Streams.

**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.

Programs read inputs from data sources (e.g., keyboard, file, network, memory buffer, or another program) and write outputs to data sinks (e.g., display console, file, network, memory buffer, or another program). In Java standard I/O, inputs and outputs are handled by the so-called streams.

### 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. It is important to mention that Java does not differentiate between the various types of data sources or sinks (e.g., file or network) in stream I/O. They are all treated as a sequential flow of data. Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console or another program.

The Java program receives data from a source by opening an input stream, and sends data to a sink by opening an output stream. All Java I/O streams are one-way (except the RandomAccessFile, which will be discussed later). If your program needs to perform both input and output, you have to open two streams - an input stream and an output stream.

Following are various streams that Java can handle with java.io package

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- Scanning and Formatting allows a program to read and write formatted text.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

**Stream I/O operations involve three steps:**

1. Open an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
2. Read from the opened input stream until "end-of-stream" encountered, or write to the opened output stream (and optionally flush the buffered output).
3. Close the input/output stream.

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

1) System.out: standard output stream
2) System.in: standard input stream
3) System.err: standard error stream

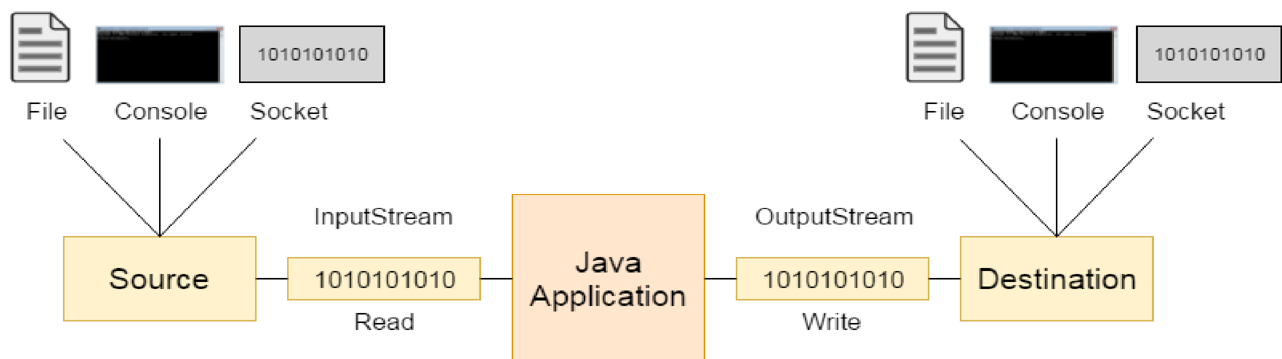
# OutputStream

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

# InputStream

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

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

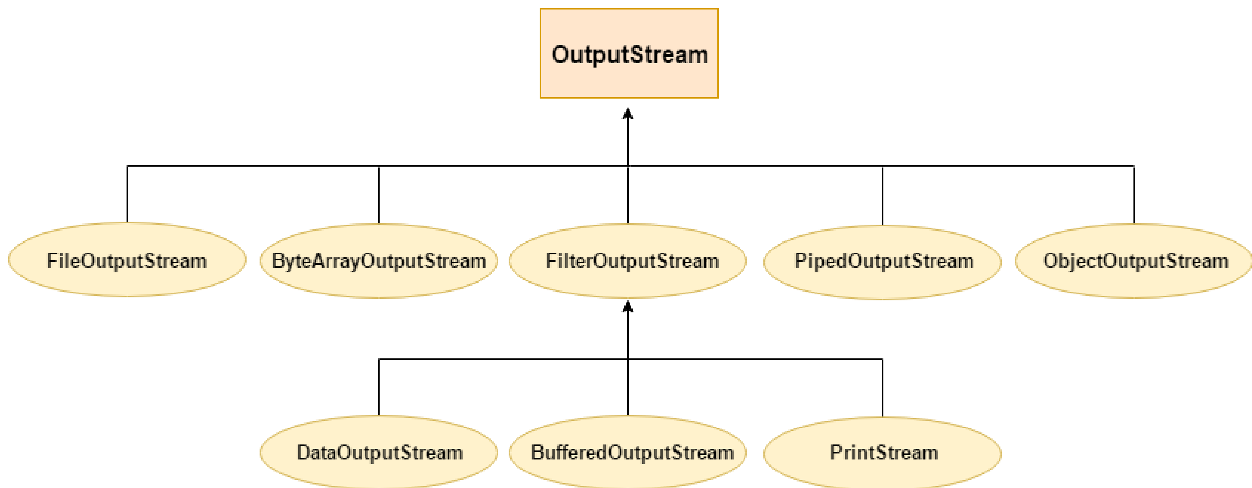



### OutputStream class

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.

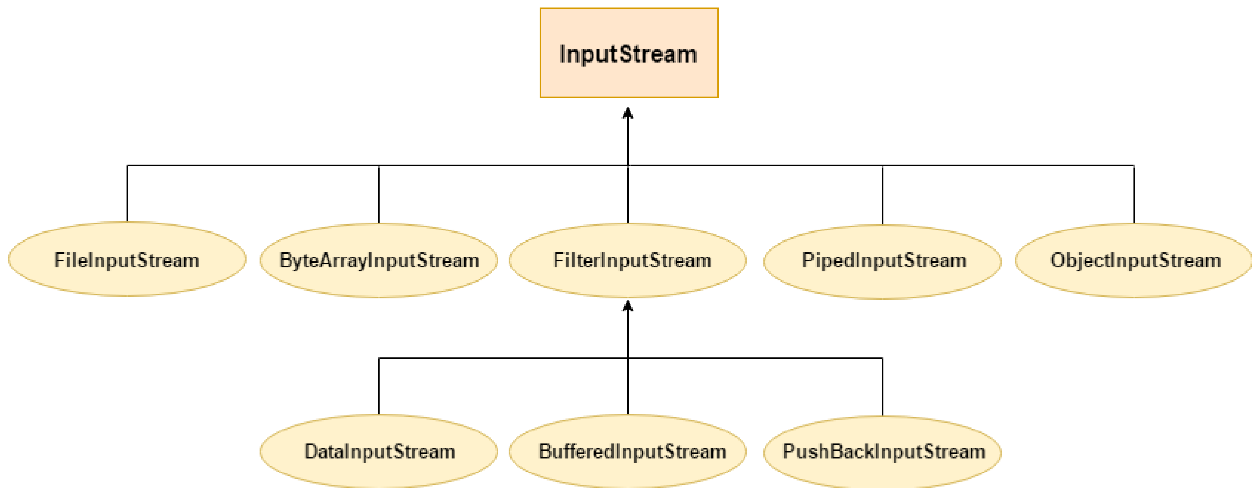
## Useful methods of OutputStream

| 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 strea |




## InputStream class

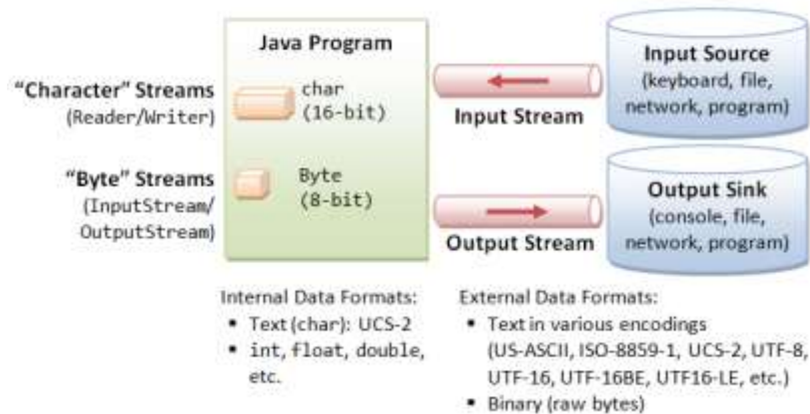
InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.




## Useful methods of InputStream

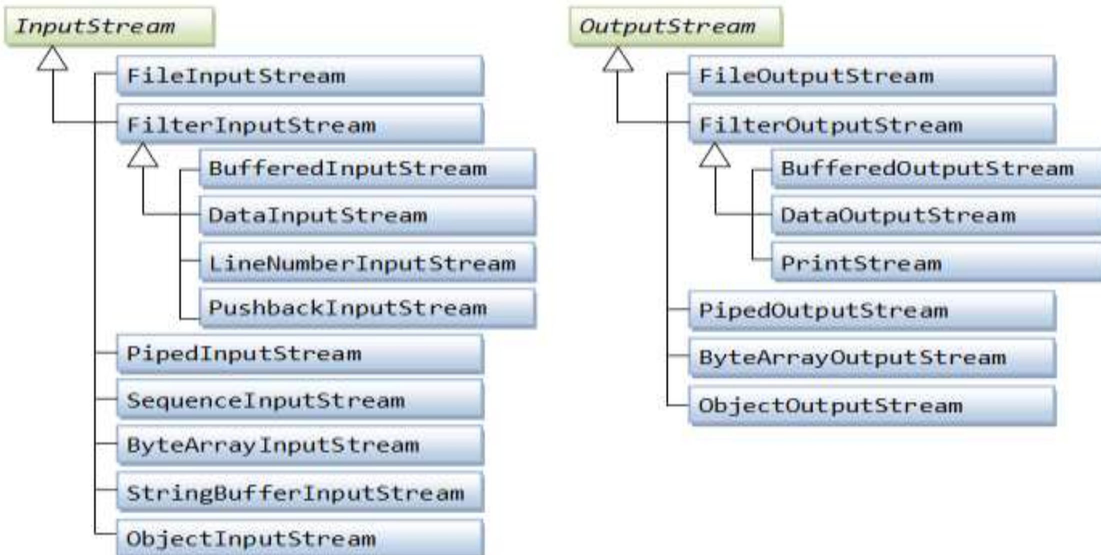
| Method | Description |
| --- | --- |
| 1) public abstract int read()throws IOException | reads the next byte of data from the input stream. It returns -1 at the end of the file. |

| 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 | |




**2.Discuss about Byte Oriented Streams in Java.**

Byte streams are used to read/write *raw bytes* serially from/to an external device. All the byte streams are derived from the `abstract` superclasses `InputStream` and `OutputStream`, as illustrated in the class diagram.



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 | |

## 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. |

## FileInputStream

A FileInputStream obtains input bytes from a file in a file system. FileInputStream is meant for reading streams of raw bytes such as image data.

## Constructors

FileInputStream(File file) Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.

FileInputStream(String name)
Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system.

### Methods:

| Modifier and Type | Method and Description |
|---|---|
| `int` | **`available`**`()`<br>Returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream. |
| `void` | **`close`**`()`<br>Closes this file input stream and releases any system resources associated with the stream. |
| `protected void` | **`finalize`**`()`<br>Ensures that the `close` method of this file input stream is called when there are no more references to it. |
| `int` | **`read`**`()`<br>Reads a byte of data from this input stream. |
| `int` | **`read`**`(byte[] b)`<br>Reads up to `b.length` bytes of data from this input stream into an array of bytes. |
| `int` | **`read`**`(byte[] b, int off, int len)`<br>Reads up to `len` bytes of data from this input stream into an array of bytes. |
| `long` | **`skip`**`(long n)`<br>Skips over and discards `n` bytes of data from the input stream. |

```
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=0;
7.              while((i=fin.read())!=-1){
8.               System.out.print((char)i);
9.              }
10.             fin.close();
11.           }catch(Exception e){System.out.println(e);}
12.          }
13.         }
```

### Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a file. A file output stream is an output stream for writing data to a File or to a FileDescriptor. Whether or not a file is available or may be created depends upon the underlying platform. FileOutputStream is meant for writing streams of raw bytes such as image data

| Constructor and Description |
| --- |
| `FileOutputStream(File file)`<br>Creates a file output stream to write to the file represented by the specified `File` object. |
| `FileOutputStream(File file, boolean append)`<br>Creates a file output stream to write to the file represented by the specified `File` object. |
| `FileOutputStream(String name)`<br>Creates a file output stream to write to the file with the specified name. |
| `FileOutputStream(String name, boolean append)`<br>Creates a file output stream to write to the file with the specified name. |

| Modifier and Type | Method and Description |
| --- | --- |
| `void` | `close()`<br>Closes this file output stream and releases any system resources associated with this stream. |
| `protected void` | `finalize()`<br>Cleans up the connection to the file, and ensures that the `close` method of this file output stream is called when there are no more references to this stream. |
| `void` | `write(byte[] b)`<br>Writes `b.length` bytes from the specified byte array to this file output stream. |
| `void` | `write(byte[] b, int off, int len)`<br>Writes `len` bytes from the specified byte array starting at offset `off` to this file output stream. |
| `void` | `write(int b)`<br>Writes the specified byte to this file output stream. |

```
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");
6.            String s="Welcome to javaTpoint.";
7.            byte b[]=s.getBytes();//converting string into byte array
8.            fout.write(b);
9.            fout.close();
10.           System.out.println("success...");
11.          }catch(Exception e){System.out.println(e);}
12.    }
13.}
```

## Buffered Streams

*For unbuffered* I/O ,each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class.
Ex:-

```
inputStream = new BufferedInputStream((new FileInputStream("sample.txt"));

outputStream = new BufferedOutputStream(new
BufferedOuputStream("characteroutput.txt"));
```

## Formatted Data-Streams:

# Data streams

DataStream support binary I/O of primitive data type values
boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the DataInput interface or the DataOutput interface.

A **DataInputStream** lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

### Constructor

`DataInputStream(InputStream in)`
Creates a DataInputStream that uses the specified underlying InputStream.

**Methods:**

| | |
|---|---|
| boolean | **readBoolean**()<br>See the general contract of the `readBoolean` method of `DataInput`. |
| byte | **readByte**()<br>See the general contract of the `readByte` method of `DataInput`. |
| char | **readChar**()<br>See the general contract of the `readChar` method of `DataInput`. |
| double | **readDouble**()<br>See the general contract of the `readDouble` method of `DataInput`. |
| float | **readFloat**()<br>See the general contract of the `readFloat` method of `DataInput`.<br>Similarly we have readLong(),readByte(),readInt() methods are available. |

A **DataOutputStream is a stream** lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in. It is created by using constructor

`DataOutputStream(OutputStream out)`
Creates a new data output stream to write data to the specified underlying output stream.

**Methods**

| | |
|---|---|
| void | `write(int b)`<br>Writes the specified byte (the low eight bits of the argument `b`) to the underlying output stream. |
| void | `writeBoolean(boolean v)`<br>Writes a `boolean` to the underlying output stream as a 1-byte value. |
| void | `writeByte(int v)`<br>Writes out a `byte` to the underlying output stream as a 1-byte value. |
| void | `writeBytes(String s)`<br>Writes out the string to the underlying output stream as a sequence of bytes. |
| void | `writeChar(int v)`<br>Writes a `char` to the underlying output stream as a 2-byte value, high byte first. |
| void | `writeChars(String s)`<br>Writes a string to the underlying output stream as a sequence of characters. |
| void | `writeDouble(double v)`<br>Converts the double argument to a `long` using the `doubleToLongBits` method in class `Double`, and then writes that `long` value to the underlying output stream as an 8-byte quantity, high byte first. |
| void | `writeFloat(float v)`<br>Converts the float argument to an `int` using the `floatToIntBits` method in class `Float`, and then writes that `int` value to the underlying output stream as a 4-byte quantity, high byte first. |
| void | `writeInt(int v)`<br>Writes an `int` to the underlying output stream as four bytes, high byte first. |

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

ObjectOutputStream and ObjectInputStream can provide an application with persistent storage for graphs of objects when used with a FileOutputStream and FileInputStream respectively. ObjectInputStream is used to recover those objects previously serialized. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.

ObjectInputStream ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine. Classes are loaded as required using the standard mechanisms.Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams.

The method `readObject` is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during serialization. When read they need to be cast to the expected type.

Primitive data types can be read from the stream using the appropriate method on DataInput.

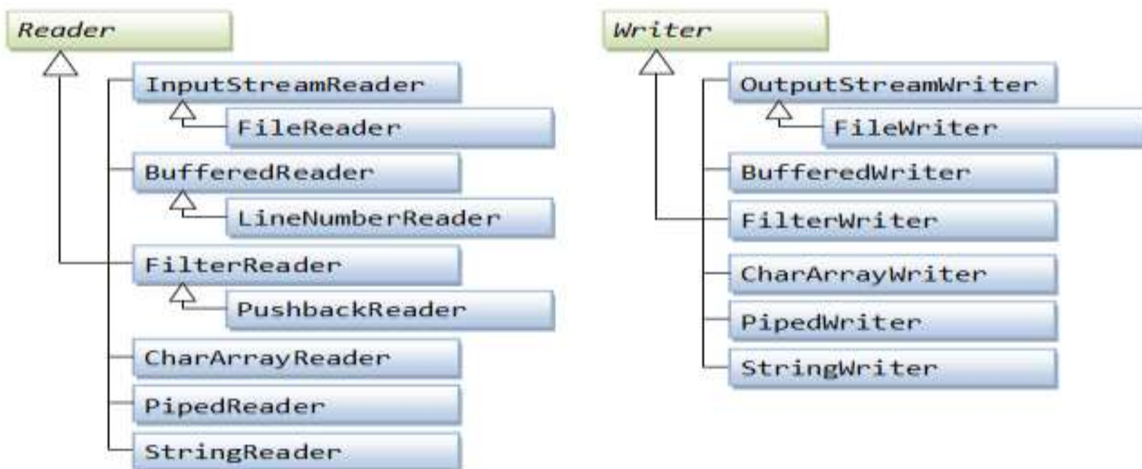
Reading an object is analogous to running the constructors of a new object. Memory is allocated for the object and initialized to zero (NULL). No-arg constructors are invoked for the non-serializable classes and then the fields of the serializable classes are restored from the stream starting with the serializable class closest to java.lang.object and finishing with the object's most specific class.

For example to read from a stream as written by the example in ObjectOutputStream:

```
FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
int i = ois.readInt();
String today = (String) ois.readObject();
Date date = (Date) ois.readObject();
ois.close();
```

3.Write about Character Oriented Streams in Java.

Java internally stores characters (char type) in 16-bit UCS-2 character set. But the external data source/sink could store characters in other character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes. Hence, Java has to differentiate between byte-based I/O for processing 8-bit raw bytes, and character-based I/O for processing texts.




# Java Writer

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

**Constructor**

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

## Methods

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

| abstract void | write(char[] cbuf, int off, int len) | It writes a portion of an array of characters. |
|---|---|---|
| void | write(int c) | It writes a single character. |
| void | write(String str) | It writes a string. |
| void | write(String str, int off, int len) | It writes a portion of a string. |

```
1. import java.io.*;
2. public class WriterExample {
3.     public static void main(String[] args) {
4.         try {
5.             Writer w = new FileWriter("output.txt");
6.             String content = "I love my country";
7.             w.write(content);
8.             w.close();
9.             System.out.println("Done");
10.        } catch (IOException e) {
11.            e.printStackTrace();
12.        }
13.    }
14.}
```

## Java Reader

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

**protected Reader()** It creates a new character-stream reader whose critical sections will synchronize on the reader itself.

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

## Methods

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

```
1. import java.io.*;
2. public class ReaderExample {
3.     public static void main(String[] args) {
4.         try {
5.             Reader reader = new FileReader("file.txt");
6.             int data = reader.read();
7.             while (data != -1) {
8.                 System.out.print((char) data);
```

```
9.                 data = reader.read();
10.             }
11.             reader.close();
12.         } catch (Exception ex) {
13.             System.out.println(ex.getMessage());
14.         }
15.     }
16.}
```

## 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.

**Constructors of FileWriter class**

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.

| 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. |

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

## Java FileReader Class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

### 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. |

Example:

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

## BufferedReader `extends Reader`

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap

a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

```
BufferedReader in    = new BufferedReader(new FileReader("foo.in"));
```

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

## **BufferedWriter** extends Writer

Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

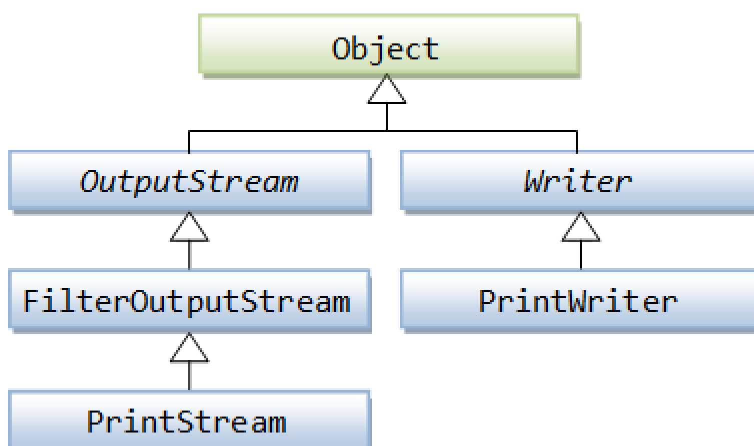
The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.

A newLine() method is provided, which uses the platform's own notion of line separator as defined by the system property line.separator. Not all platforms use the newline character ('\n') to terminate lines. Calling this method to terminate each output line is therefore preferred to writing a newline character directly.

In general, a Writer sends its output immediately to the underlying character or byte stream. Unless prompt output is required, it is advisable to wrap a BufferedWriter around any Writer whose write() operations may be costly, such as FileWriters and OutputStreamWriters. For example,

```
PrintWriter out
  = new PrintWriter(new BufferedWriter(new FileWriter("foo.out")));
```

## **4. Discuss about java.io.PrintStream & java.io.PrintWriter**



The byte-based java.io.printSteam supports convenient printing methods such as print() and println() for printing primitives and text string. Primitives are converted to their string representation for printing.

The printf() and format() were introduced in JDK 1.5 for formatting output with former specifiers. printf() and format() are identical.

A PrintStream never throws an IOException. Instead, it sets an internal flag which can be checked via the checkError() method. A PrintStream can also be created to flush the output automatically.

That is, the flush() method is automatically invoked after a byte array is written, one of the println() methods is invoked, or after a newline ('\n') is written.

The standard output and error streams (System.out and System.err) belong to PrintStream.

All characters printed by a PrintStream are converted into bytes using the default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes.

The character-stream PrintWriter is similar to PrintStream, except that it write in characters instead of bytes. The PrintWriter also supports all the convenient printing methods print(), println(), printf() and format(). It never throws an IOException and can optionally be created to support automatic flushing.

# Unit-3

## Wrapper Classes in Java

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

**Need of Wrapper Classes**

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayListand Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

**Primitive Data types and their Corresponding Wrapper class**

| Primitive Data Type | Wrapper Class |
|---|---|
| char | Character |
| byte | Byte |
| short | Short |
| long | Integer |
| float | Float |
| double | Double |
| boolean | Boolean |

### Autoboxing and Unboxing

**Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.
Int x=6;
Integer obj=new Integer(x); // converting int to Integer class

**Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double etc.
Int x = Integer.valueOf(obj);

**Methods**

| Sr.No. | Method & Description |
|---|---|
| 1 | **xxxValue()**<br><br>Converts the value of *this* Number object to the xxx data type and returns it. |
| 2 | **compareTo()**<br><br>Compares *this* Number object to the argument. |
| 3 | **equals()**<br><br>Determines whether *this* number object is equal to the argument. |
| 4 | **valueOf()**<br><br>Returns an Integer object holding the value of the specified primitive. |
| 5 | **toString()**<br><br>Returns a String object representing the value of a specified int or Integer. |
| 6 | **parseInt()** parseFloat(),parseDouble()<br><br>This method is used to get the primitive data type of a certain String. |

**Math Class:**

**Used to perform mathematical operations. Present in System.lang package.**

| | |
|---|---|
| **1** | **abs()**<br><br>**Returns the absolute value of the argument.** |
| **2** | **ceil()**<br><br>**Returns the smallest integer that is greater than or equal to the argument. Returned as a double.** |
| **3** | **floor()**<br><br>**Returns the largest integer that is less than or equal to the argument. Returned as a double.** |

| | |
|---|---|
| 4 | rint()<br><br>Returns the integer that is closest in value to the argument. Returned as a double. |
| 5 | round()<br><br>Returns the closest long or int, as indicated by the method's return type to the argument. |
| 6 | min(x,y)<br><br>Returns the smaller of the two arguments. |
| 7 | max(x,y)<br><br>Returns the larger of the two arguments. |
| 8 | exp(x)<br><br>Returns the base of the natural logarithms, e, to the power of the argument. |
| 9 | log(x)<br><br>Returns the natural logarithm of the argument. |
| 10 | pow(x,y)<br><br>Returns the value of the first argument raised to the power of the second argument. |
| 17 | sqrt(x)<br><br>Returns the square root of the argument. |
| 18 | sin(x)<br><br>Returns the sine of the specified double value. |
| 19 | cos(x)<br><br>Returns the cosine of the specified double value. |
| 20 | tan(x)<br><br>Returns the tangent of the specified double value. |
| 21 | asin() |

| | |
|---|---|
| | **Returns the arcsine of the specified double value.** |
| **22** | **acos()**<br><br>**Returns the arccosine of the specified double value.** |
| **23** | **atan()**<br><br>**Returns the arctangent of the specified double value.** |
| **24** | **atan2()**<br><br>**Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.** |
| **25** | **toDegrees()**<br><br>**Converts the argument to degrees.** |
| **26** | **toRadians()**<br><br>**Converts the argument to radians.** |
| **27** | **random()**<br><br>**Returns a random number.** |

### Collections Framework

The collections framework was designed to meet several goals, such as −

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following −

- Interfaces − These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- Implementations, i.e., Classes − These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

- Algorithms − These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

## The Collection Interfaces

The collections framework defines several interfaces. This section provides an overview of each interface −

| Sr.No. | Interface & Description |
|---|---|
| 1 | **The Collection Interface**<br><br>This enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| 2 | **The List Interface**<br><br>This extends **Collection** and an instance of List stores an ordered collection of elements. |
| 3 | **The Set**<br><br>This extends Collection to handle sets, which must contain unique elements. |
| 4 | **The SortedSet**<br><br>This extends Set to handle sorted sets. |
| 5 | **The Map**<br><br>This maps unique keys to values. |

## The Collection Classes

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table −

| Sr.No. | Class & Description |
|---|---|
| 1 | **AbstractCollection**<br><br>Implements most of the Collection interface. |
| 2 | **AbstractList**<br><br>Extends AbstractCollection and implements most of the List interface. |
| 3 | **AbstractSequentialList**<br><br>Extends AbstractList for use by a collection that uses sequential rather than random access of its elements. |
| 4 | **LinkedList**<br><br>Implements a linked list by extending AbstractSequentialList. |
| 5 | **ArrayList**<br><br>Implements a dynamic array by extending AbstractList. |
| 6 | **AbstractSet**<br><br>Extends AbstractCollection and implements most of the Set interface. |
| 7 | **HashSet**<br><br>Extends AbstractSet for use with a hash table. |
| 8 | **LinkedHashSet**<br><br>Extends HashSet to allow insertion-order iterations. |
| 9 | **TreeSet**<br><br>Implements a set stored in a tree. Extends AbstractSet. |
| 10 | **AbstractMap**<br><br>Implements most of the Map interface. |

| 11 | **HashMap**<br><br>Extends AbstractMap to use a hash table. |
|---|---|
| 12 | **TreeMap**<br><br>Extends AbstractMap to use a tree. |

**Vector Class**

Vector implements a dynamic array. It is similar to ArrayList, but with two differences −

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

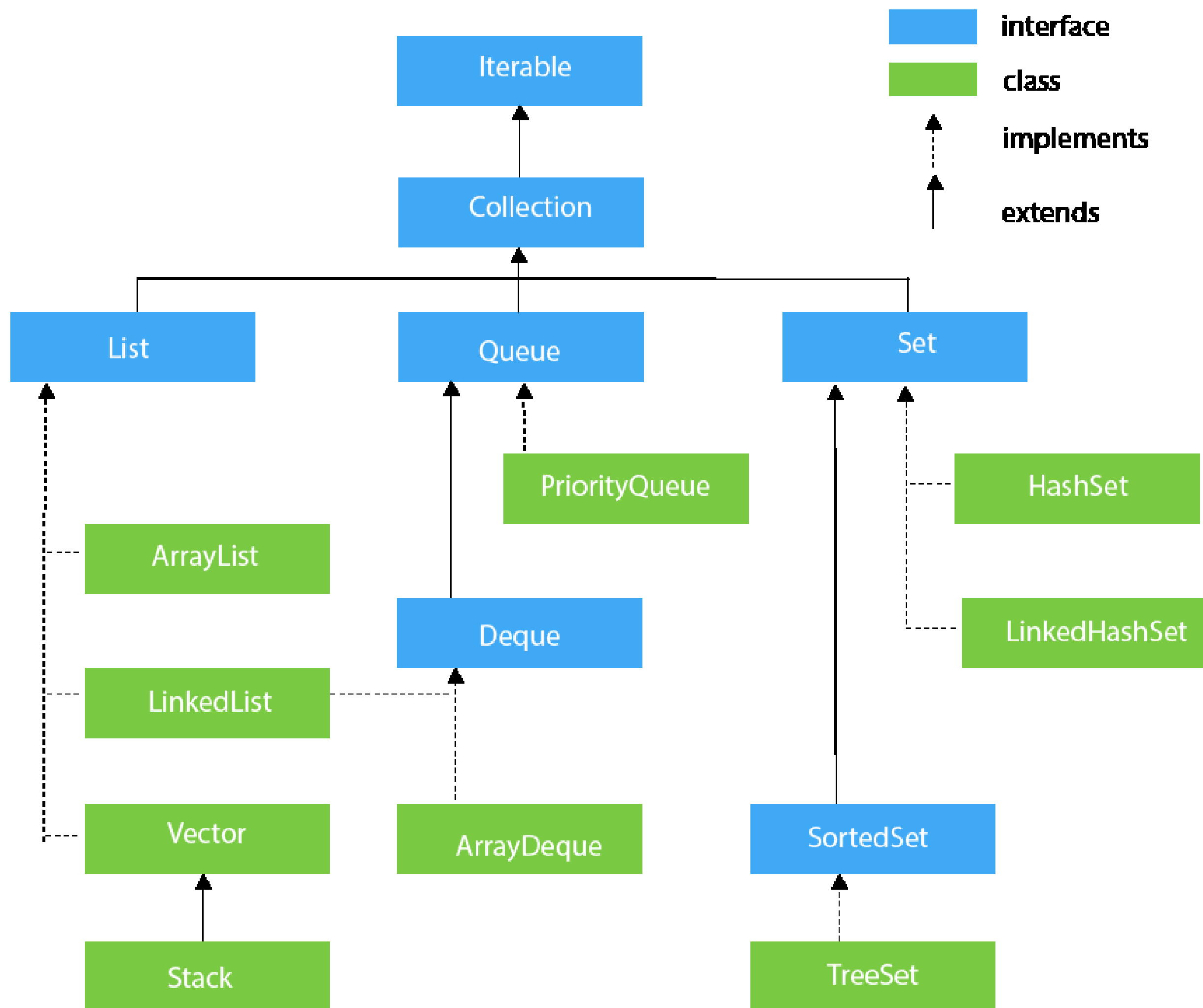
| Sr.No. | Constructor & Description |
|---|---|
| 1 | **Vector( )**<br><br>This constructor creates a default vector, which has an initial size of 10. |
| 2 | **Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size. |

| Sr.No. | Method & Description |
|---|---|
| 1 | **void add(int index, Object element)**<br><br>Inserts the specified element at the specified position in this Vector. |
| 2 | **boolean add(Object o)**<br><br>Appends the specified element to the end of this Vector. |
| 3 | **boolean addAll(Collection c)**<br><br>Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator. |

| | |
|---|---|
| 4 | **boolean addAll(int index, Collection c)**<br><br>Inserts all of the elements in in the specified Collection into this Vector at the specified position. |
| 5 | **void addElement(Object obj)**<br><br>Adds the specified component to the end of this vector, increasing its size by one. |
| 6 | **int capacity()**<br><br>Returns the current capacity of this vector. |
| 7 | **void clear()**<br><br>Removes all of the elements from this vector. |
| 8 | **Object clone()**<br><br>Returns a clone of this vector. |
| 9 | boolean contains(Object elem)<br><br>Tests if the specified object is a component in this vector. |
| 10 | boolean containsAll(Collection c)<br><br>Returns true if this vector contains all of the elements in the specified Collection. |
| 11 | void insertElementAt(Object obj, int index)<br><br>Inserts the specified object as a component in this vector at the specified index. |
| 12 | boolean isEmpty()<br><br>Tests if this vector has no components. |
| 13 | Object lastElement()<br><br>Returns the last component of the vector. |
| 14 | int lastIndexOf(Object elem)<br><br>Returns the index of the last occurre |

## 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 parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator 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. |
|---|---|---|

**ArrayList**

The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold

Following is the list of the constructors provided by the ArrayList class.

| Sr.No. | Constructor & Description |
|---|---|
| 1 | **ArrayList( )**<br>This constructor builds an empty array list. |
| 2 | **ArrayList(Collection c)**<br>This constructor builds an array list that is initialized with the elements of the collection **c**. |
| 3 | **ArrayList(int capacity)**<br>This constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list. |

Apart from the methods inherited from its parent classes, ArrayList defines the following methods –

| Sr.No. | Method & Description |
|---|---|
| 1 | **void add(int index, Object element)** |

| | |
|---|---|
| | Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 \|\| index > size()). |
| 2 | **boolean add(Object o)**<br><br>Appends the specified element to the end of this list. |
| 3 | **boolean addAll(Collection c)**<br><br>Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException, if the specified collection is null. |
| 4 | **boolean addAll(int index, Collection c)**<br><br>Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null. |
| 5 | **void clear()**<br><br>Removes all of the elements from this list. |
| 6 | **Object clone()**<br><br>Returns a shallow copy of this ArrayList. |
| 7 | **boolean contains(Object o)**<br><br>Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element **e** such that (o==null ? e==null : o.equals(e)). |
| 8 | **void ensureCapacity(int minCapacity)**<br><br>Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument. |
| 9 | **Object get(int index)**<br><br>Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 \|\| index >= size()). |

### Linked List

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

Following are the constructors supported by the LinkedList class.

| Sr.No. | Constructor & Description |
|---|---|
| 1 | **LinkedList( )**<br><br>This constructor builds an empty linked list. |
| 2 | **LinkedList(Collection c)**<br><br>This constructor builds a linked list that is initialized with the elements of the collection **c**. |

Apart from the methods inherited from its parent classes, LinkedList defines following methods −

| Sr.No. | Method & Description |
|---|---|
| 1 | **void add(int index, Object element)**<br><br>Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 \|\| index > size()). |
| 2 | **boolean add(Object o)**<br><br>Appends the specified element to the end of this list. |
| 3 | **boolean addAll(Collection c)**<br><br>Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null. |
| 4 | **boolean addAll(int index, Collection c)**<br><br>Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null. |
| 5 | **void addFirst(Object o)**<br><br>Inserts the given element at the beginning of this list. |
| 6 | **void addLast(Object o)** |

| | Appends the given element to the end of this list. |
|---|---|
| 7 | **void clear()**<br><br>Removes all of the elements from this list. |
| 8 | **Object clone()**<br><br>Returns a shallow copy of this LinkedList. |
| 9 | **boolean contains(Object o)**<br><br>Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)). |
| 10 | **Object get(int index)**<br><br>Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 \|\| index >= size()). |
| 11 | **Object getFirst()**<br><br>Returns the first element in this list. Throws NoSuchElementException if this list is empty. |
| 12 | **Object getLast()**<br><br>Returns the last element in this list. Throws NoSuchElementException if this list is empty. |

## HashSet

HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage.

A hash table stores information by using a mechanism called **hashing**. In hashing, the informational content of a key is used to determine a unique value, called its hash code.

The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Following is the list of constructors provided by the HashSet class.

| Sr.No. | Constructor & Description |
|---|---|
| 1 | **HashSet( )**<br><br>This constructor constructs a default HashSet. |
| 2 | **HashSet(Collection c)**<br><br>This constructor initializes the hash set by using the elements of the collection **c**. |

Apart from the methods inherited from its parent classes, HashSet defines following methods −

| Sr.No. | Method & Description |
|---|---|
| 1 | **boolean add(Object o)**<br><br>Adds the specified element to this set if it is not already present. |
| 2 | **void clear()**<br><br>Removes all of the elements from this set. |
| 3 | **Object clone()**<br><br>Returns a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| 4 | **boolean contains(Object o)**<br><br>Returns true if this set contains the specified element. |
| 5 | **boolean isEmpty()**<br><br>Returns true if this set contains no elements. |
| 6 | **Iterator iterator()**<br><br>Returns an iterator over the elements in this set. |
| 7 | **boolean remove(Object o)**<br><br>Removes the specified element from this set if it is present. |

| 8 | **int size()**<br><br>Returns the number of elements in this set (its cardinality). |
|---|---|

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

   public static void main(String args[]) {
      // create a hash set
      HashSet hs = new HashSet();

      // add elements to the hash set
      hs.add("B");
      hs.add("A");
      hs.add("D");
      hs.add("E");
      hs.add("C");
      hs.add("F");
      System.out.println(hs);
   }
}
```

**HashMap**

The HashMap class uses a hashtable to implement the Map interface. This allows the execution time of basic operations, such as get( ) and put( ), to remain constant even for large sets.

Following is the list of constructors supported by the HashMap class.

| Sr.No. | Constructor & Description |
|---|---|
| 1 | **HashMap( )**<br><br>This constructor constructs a default HashMap. |

| 2 | **HashMap(Map m)**<br><br>This constructor initializes the hash map by using the elements of the given Map object **m**. |
|---|---|
| 3 | **HashMap(int capacity)**<br><br>This constructor initializes the capacity of the hash map to the given integer value, capacity. |

Apart from the methods inherited from its parent classes, HashMap defines the following methods −

| Sr.No. | Method & Description |
|---|---|
| 1 | **void clear()**<br><br>Removes all mappings from this map. |
| 2 | **Object clone()**<br><br>Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| 3 | **boolean containsKey(Object key)**<br><br>Returns true if this map contains a mapping for the specified key. |
| 4 | **boolean containsValue(Object value)**<br><br>Returns true if this map maps one or more keys to the specified value. |
| 5 | **Set entrySet()**<br><br>Returns a collection view of the mappings contained in this map. |
| 6 | **Object get(Object key)** |

|  |  |
|---|---|
|  | Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key. |
| 7 | **boolean isEmpty()**<br><br>Returns true if this map contains no key-value mappings. |
| 8 | **Set keySet()**<br><br>Returns a set view of the keys contained in this map. |
| 9 | **Object put(Object key, Object value)**<br><br>Associates the specified value with the specified key in this map. |
| 10 | **putAll(Map m)**<br><br>Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| 11 | **Object remove(Object key)**<br><br>Removes the mapping for this key from this map if present. |
| 12 | **int size()**<br><br>Returns the number of key-value mappings in this map. |
| 13 | **Collection values()**<br><br>Returns a collection view of the values contained in this map. |

```
import java.util.*;
class HashMap1{
 public static void main(String args[]){
   HashMap<Integer,String> hm=new HashMap<Integer,String>();
    System.out.println("Initial list of elements: "+hm);
      hm.put(100,"Amit");
```

```
        hm.put(101,"Vijay");
        hm.put(102,"Rahul");

        System.out.println("After invoking put() method ");
        for(Map.Entry m:hm.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }

        hm.putIfAbsent(103, "Gaurav");
        System.out.println("After invoking putIfAbsent() method ");
        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
          }
        HashMap<Integer,String> map=new HashMap<Integer,String>();
        map.put(104,"Ravi");
        map.putAll(hm);
        System.out.println("After invoking putAll() method ");
        for(Map.Entry m:map.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
          }
    }
   }
```

## BitSet

The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

The BitSet defines the following two constructors.

| Sr.No. | Constructor & Description |
|---|---|
| 1 | **BitSet( )**<br>This constructor creates a default object. |

| 2 | **BitSet(int size)**<br><br>This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero. |
|---|---|

itSet implements the Cloneable interface and defines the methods listed in the following table −

| Sr.No. | Method & Description |
|---|---|
| 1 | **void and(BitSet bitSet)**<br><br>ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object. |
| 2 | **void andNot(BitSet bitSet)**<br><br>For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared. |
| 3 | **int cardinality( )**<br><br>Returns the number of set bits in the invoking object. |
| 4 | **void clear( )**<br><br>Zeros all bits. |
| 5 | **void clear(int index)**<br><br>Zeros the bit specified by index. |
| 6 | **void clear(int startIndex, int endIndex)**<br><br>Zeros the bits from startIndex to endIndex. |
| 7 | **Object clone( )**<br><br>Duplicates the invoking BitSet object. |
| 8 | **boolean equals(Object bitSet)**<br><br>Returns true if the invoking bit set is equivalent to the one passed in bitSet. Otherwise, the method returns false. |
| 9 | **void flip(int index)**<br><br>Reverses the bit specified by the index. |

| | |
|---|---|
| 10 | **void flip(int startIndex, int endIndex)**<br><br>Reverses the bits from startIndex to endIndex. |
| 11 | **boolean get(int index)**<br><br>Returns the current state of the bit at the specified index. |
| 12 | **BitSet get(int startIndex, int endIndex)**<br><br>Returns a BitSet that consists of the bits from startIndex to endIndex. The invoking object is not changed. |
| 13 | **int hashCode( )**<br><br>Returns the hash code for the invoking object. |
| 14 | **boolean intersects(BitSet bitSet)**<br><br>Returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1. |
| 15 | **boolean isEmpty( )**<br><br>Returns true if all bits in the invoking object are zero. |
| 16 | **int length( )**<br><br>Returns the number of bits required to hold the contents of the invoking BitSet. This value is determined by the location of the last 1 bit. |
| 17 | **int nextClearBit(int startIndex)**<br><br>Returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex. |
| 18 | **int nextSetBit(int startIndex)**<br><br>Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned. |
| 19 | **void or(BitSet bitSet)**<br><br>ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object. |
| 20 | **void set(int index)** |

| | |
|---|---|
| | Sets the bit specified by index. |
| 21 | **void set(int index, boolean v)**<br><br>Sets the bit specified by index to the value passed in v. True sets the bit, false clears the bit. |
| 22 | **void set(int startIndex, int endIndex)**<br><br>Sets the bits from startIndex to endIndex. |
| 23 | **void set(int startIndex, int endIndex, boolean v)**<br><br>Sets the bits from startIndex to endIndex, to the value passed in v. true sets the bits, false clears the bits. |
| 24 | **int size( )**<br><br>Returns the number of bits in the invoking BitSet object. |
| 25 | **String toString( )**<br><br>Returns the string equivalent of the invoking BitSet object. |
| 26 | **void xor(BitSet bitSet)**<br><br>XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object. |

```
import java.util.BitSet;
public class BitSetDemo {

  public static void main(String args[]) {
     BitSet bits1 = new BitSet(16);
     BitSet bits2 = new BitSet(16);

     // set some bits
     for(int i = 0; i < 16; i++) {
        if((i % 2) == 0) bits1.set(i);
        if((i % 5) != 0) bits2.set(i);
```

```
}


System.out.println("Initial pattern in bits1: ");
System.out.println(bits1);
System.out.println("\nInitial pattern in bits2: ");
System.out.println(bits2);


// AND bits
bits2.and(bits1);
System.out.println("\nbits2 AND bits1: ");
System.out.println(bits2);
```

# 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. }
```

**Searching and Sorting**

The collections framework defines several algorithms that can be applied to collections and maps.

The methods defined in collection framework's algorithm are summarized in the following table −

| Sr.No. | Method & Description |
| --- | --- |
| 1 | **static int binarySearch(List list, Object value, Comparator c)**<br><br>Searches for value in the list ordered according to **c**. Returns the position of value in list, or -1 if value is not found. |
| 2 | **static int binarySearch(List list, Object value)**<br><br>Searches for value in the list. The list must be sorted. Returns the position of value in list, or -1 if value is not found. |
| 3 | **static Object max(Collection c, Comparator comp)**<br><br>Returns the maximum element in **c** as determined by comp. |
| 4 | **static Object max(Collection c)**<br><br>Returns the maximum element in **c** as determined by natural ordering. The collection need not be sorted. |
| 5 | **static Object min(Collection c, Comparator comp)**<br><br>Returns the minimum element in **c** as determined by comp. The collection need not be sorted. |
| 6 | **static Object min(Collection c)**<br><br>Returns the minimum element in **c** as determined by natural ordering. |
| 7 | s**tatic void sort(List list, Comparator comp)**<br><br>Sorts the elements of list as determined by comp. |
| 8 | static void sort(List list) |

|  |  |
| --- | --- |
|  | Sorts the elements of the list as determined by their natural ordering. |
| 9 | **static void swap(List list, int idx1, int idx2)**<br>Exchanges the elements in the list at the indices specified by idx1 and idx2. |

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

   public static void main(String args[]) {

      // Create and initialize linked list
      LinkedList ll = new LinkedList();
      ll.add(new Integer(-8));
      ll.add(new Integer(20));
      ll.add(new Integer(-20));
      ll.add(new Integer(8));

      // Create a reverse order comparator
      Comparator r = Collections.reverseOrder();

      // Sort list by using the comparator
      Collections.sort(ll, r);

      // Get iterator
      Iterator li = ll.iterator();
      System.out.print("List sorted in reverse: ");

      while(li.hasNext()) {
         System.out.print(li.next() + " ");
      }
}
```

## Calendar Class in Java

Calendar class in Java is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable, Serializable, Cloneable interfaces.

As it is an Abstract class, so we cannot use a constructor to create an instance. Instead, we will have to use the static method Calendar.getInstance() to instantiate and implement a sub-class.

- Calendar.getInstance(): return a Calendar instance based on the current time in the default time zone with the default locale.
- Calendar.getInstance(TimeZone zone)
- Calendar.getInstance(Locale aLocale)
- Calendar.getInstance(TimeZone zone, Locale aLocale)

**Java program to demonstrate getInstance() method**:

```
// Date getTime(): It is used to return a

// Date object representing this

// Calendar's time value.

 import java.util.*;

public class Calendar1 {

   public static void main(String args[])

   {

      Calendar c = Calendar.getInstance();

      System.out.println("The Current Date is:" + c.getTime());

   } }
```

| Methods | |
|---|---|
| abstract void add(int field, int amount) | It is used to add or subtract the specified amount of time to the given calendar field, based on the calendar’s rules. |
| int get(int field) | It is used to return the value of the given calendar field. |
| abstract int getMaximum(int field) | It is used to return the maximum value for the given calendar field of this Calendar instance. |
| abstract int getMinimum(int field) | It is used to return the minimum value for the given calendar field of this Calendar instance. |
| Date getTime() | It is used to return a Date object representing this Calendar’s time value.</td |

// creating Calendar object

Calendar calendar = Calendar.getInstance();

// Demonstrate Calendar's get()method

System.out.println("Current Calendar's Year: " + calendar.get(Calendar.YEAR));

System.out.println("Current Calendar's Day: " + calendar.get(Calendar.DATE));

System.out.println("Current MINUTE: " + calendar.get(Calendar.MINUTE));

System.out.println("Current SECOND: " + calendar.get(Calendar.SECOND));

# Date class in Java

The class Date represents a specific instant in time, with millisecond precision. The Date class of java.util package implements Serializable, Cloneable and Comparable interface. It provides constructors and methods to deal with date and time with java.

**Constructors**

- **Date()** : Creates date object representing current date and time.
- **Date(long milliseconds)** : Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
- **Date(int year, int month, int date)**
- **Date(int year, int month, int date, int hrs, int min)**
- **Date(int year, int month, int date, int hrs, int min, int sec)**
- **Date(String s)**

\

**Methods**

- **boolean after(Date date) :** Tests if current date is after the given date.
- **boolean before(Date date) :** Tests if current date is before the given date.
- **int compareTo(Date date) :** Compares current date with given date. Returns 0 if the argument Date is equal to the Date; a value less than 0 if the Date is before the Date argument; and a value greater than 0 if the Date is after the Date argument.
- **long getTime()** : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
- **void setTime(long time)** : Changes the current date and time to given time.

```
 Date d1 = new Date(2000, 11, 21);
Date d2 = new Date();  // Current date
Date d3 = new Date(2010, 1, 3);

boolean a = d3.after(d1);
System.out.println("Date d3 comes after " +
                   "date d2: " + a);

boolean b = d3.before(d2);
System.out.println("Date d3 comes before "+
                   "date d2: " + b);
```