Collection Framework

Java.util package

Legacy classes(**jdk1.0**) Collection Framework(**jdk1.2**) Generic types(**jdk1.5**)

Limitations with arrays:

1) Size of an array is **fixed** i.e., once an array is created, there is no way of increasing or decreasing its size.
2) Array can hold only **homogeneous** data elements.
   Example:

```
String[] str = new String[5];
Str[0] = new String("rishikesh");
Str[1]=new Integer(10); //C.E
```

3) No predefined data structures to manage array data.

To overcome above limitations, Collection Framework was introduced:

1) The size of a collection is not fixed i.e., the size of a collection can be increased or decreased.
2) Collection can hold heterogeneous (any type of) objects.
3) **Predefined data structures such as Stack, Queue, DeQue, LinkedList ,etc are defined to manage collection data.**

Difference between array and collection:

Array	**Collection**
Size is fixed	Collection size can grow or shrink
Contains only **homogeneous** elements.	Contains **heterogeneous** elements.
The elements type can be Primitive or Reference type.	**Allows only Reference types but not Primitive type.**
No predefined data structors to manage array data.	Predefined data structors are defined to manage collection data.

There are two root interfaces in **java.util** package:

Java.util.Collection

Java.util.Map

Java.util.Collection <<interface>>

It represents group of individual objects as a single entity (object).
It acts as a **root** interface for entire collection framework.
This interface defines the most common **methods**, **which can be applied to any collection object**.

Collection vs Collections vs collection

Collection is an interface. Where as Collections is a class, which contains utility methods such as **sort**(), **binarySearch**(), **copy**(), etc, and **collection represents group of objects as a single entity**.

Collection properties:

1. Some Collection implementation classes allow **duplicate** elements and others will not allow duplicate elements.
2. Some Collection implementation classes are **ordered** and others are unordered collections.
3. Some Collection implementation classes allow **null values** and others will not allow null values.
4. By default, some Collection implementation classes are **sorted** and others are not sorted.
5. By default, Some Collection implementation classes are **synchronized** and others are not synchronized.

Basic operations in collections are:

1. Add element(s) to the collection.
2. Remove element(s) from the collection.
3. Search element(s) in the collection.
4. Retrieve all elements from the underlying collection (Iterate through the collection).
5. Peform all above operations but based on index, hence it is applicable only for List implementation classes.

To perform above operations, the **Collection** interface have the following common methods:

1. boolean add(Object o);
2. boolean addAll(Collection c);
3. Boolean remove(Object o);
4. Boolean removeAll(Collection c);
5. Void clear();
6. Boolean contains(Object o);
7. Boolean containsAll(Collection c);
8. **Iterator iterator();**
9. Boolean isEmpty();
10. Int size();
11. Object[] toArray();

//Add object(s) to the collection.

//Remove object(s) from the collection.

//Search object(s) from the collection.

//Iterate through the collection.

There are three direct subinterfaces of Collection interface: List, Set, Queue.

Collection<<interface>>

List<<interface>> **Set<<interface>>** **Queue<<interface>**

List <<interface>>

It is inherited from a Collection interface.

The List is a **growable array** whose size can be increased or decreased.

It allows duplicate elements.

List is an ordered collection based on index but not insertion.

List allows more than one null value.

By default, none of the List implementation classes are sorted.

By default, some of the List implementation classes are synchronized.

Since List is an ordered collection based on index, List interface have index specific methods along with common methods inherited from Collection interface.

1. Boolean add(int index, Object o);
2. Boolean **addAll**(intindex, Collection c);
3. Object remove(int index);
4. Int indexOf(Object o); //Returns index of first occurance of specified object, otherwise returns -1
5. Int lastIndexOf(Object o); //Returns index of last occurance of specified object, otherwise returns -1
6. Object set(int index, Object new);//Replaces with new element
7. **ListIterator listIterator(); //Applicable only for List implementation classes.**
8. Object **get**(int index); //Retrieves an element from the specified index.

Vector

Vector is a growable array whose size can be grown or shrink.

Vector allows duplicate elements.

Vector is an ordered collection based on index.

Vector allows any number of null values.

By default, vector elements are not sorted.

Vector is a synchronized collection (**thread-safe**), since it is a legacy class.

Example:

```
//VectorDemo.java
import java.util.Enumeration;
import java.util.Vector; public class
VectorDemo{
    public static void main(String[] args) { Vector v
            = new Vector(); v.addElement(1); //
            auto boxing v.addElement(new
            Integer(5));
            v.add(new Integer(3));
            v.add(new Integer(3));   //allows duplicate element
            v.add(null);       //allows null value
            v.add(0, new Integer(4)); //adding element using index position
            v.add("hello");
            System.out.println(v);  //[4, 1, 5, 3, 3, null, hello]
    }
}
```

ArrayList

ArrayList is a growable array i.e., its size can be increased or decreased.

ArrayList allows duplicate elements.

ArrayList is an ordered collection based on index.

ArrayList allows duplicate null values.

By default, ArrayList elements are not sorted.

But, ArrayList elements can be explicitly sorting using following utility methods from Collections class:

Public static void sort(List) // Natural Sorting order

Public static void sort(List, Comparator) // Customized Sorting order

By default, ArrayList is non-synchronized collection (non thread-safe). But, explicitly we can covert to synchronized version using following utility method from Collections class.

Public static List synchronizedList(List)

Example: import java.util.*;

```
class ArrayListDemo{
    public static void main(String[]
            args){ List al=new
            ArrayList(); al.add("A");
            al.add(10);
            al.add("A");
            al.add(null);
            System.out.println(al);//[A,10, A, null]
            al.remove(2);
            System.out.println(al); //[A,10,
            null] al.add(2,"M");
            al.add("N");
            System.out.println(al);// [A, 10, M, null, N]

            al = Collections.synchronizedList(al);
    }
}
```

Difference between ArrayList and Vector:

ArrayList	Vector
Non synchronized collection i.e., **non thread-safe**.	By default, synchronized i.e**., thread-safe**.
High Performance.	Low Performance.
Non-legacy class and introduced in JDK1.2 version	Legacy class and introduced in JDK1.0 version.

LinkedList

1. The underlying data structure is doubly LinkedList.
2. LinkedList allows duplicate elements.
3. **LinkedList is an ordered collection based on index but not insertion**.
4. LinkedList allows duplicate null values.
5. By default, LinkedList elements are not sorted.
6. By default, LinkedList is non-synchronized class (non thread-safe).

Stack

It is a child class of Vector and contains only one constructor.

Stack s=new Stack();

Methods

1. Object **push**(Object o)To insert an element at the top of the stack.
2. Object **pop**();To remove and return the top of the stack.

Example: import java.util.*; class StackDemo{

```
    public static void main(String[] args){
            Stack s=new Stack();
            s.push("A");
            s.push("B");
            s.push("C");
            System.out.println(s);//[A, B, C]
            s.pop();
            System.out.println(s);//[A, B]
    }
}
```

Iterating through collection elements

Used to retrieve all elements but one at a time from the underlying collection. There are four ways in which objects can be retrieved from the underlying collection:

1. Enumeration:Applicable only for legacy classes.
2. Iterator :Applicable for all Collection implementation classes.
3. ListIterator :Applicable only for List implementation classes.
4. Enhanced for loop : Used in case of Generic types

Java.util.Enumeration <<Interface>>

Introduced in 1.0 version. Hence, it is applicable only for legacy classes.

The following method is applicable only for legacy classes, whose return type is an Enumeration. public Enumeration **elements()**

Enumeration interface defines the following two methods

1. Public Boolean hasMoreElements();
2. Public Object nextElement();

Example: Print even numbers import java.util.*;

class EnumerationDemo {

```
    public static void main(String[] args) {
            Vector v= new Vector();
            for(int i=1; i<=10; i++){
                    v.addElement(i);
            }
            System.out.println(v.toString());  //[1,2,3,4,5,6,7,8,9,10]

            Enumeration e=v.elements();
            while(e.hasMoreElements()){
                    int i=(Integer) e.nextElement();//unboxing
                    if(i%2==0){ //even numbers
                            System.out.println(i);
                    }
            }
    }
}
```

O/P: [2, 4,6,8,10]

Limitations with Enumeration:

1) Enumeration is applicable only for legacy classes.
2) While iteratering, we can perform only read operation, but not remove, modify, add operations in underlying collection.

Iterator <<interface>>

It is introduced in 1.2 version. Hence, it is used with both legacy and non-legacy classes.

The following method is declared in the Collection interface, hence it is applicable for all Collection implementation classes.

Public Iterator **iterator**()

Iterator interface defines the following three methods:

1. Public Boolean hasNext();
2. Public Object next();
3. Public void **remove()**;

Example: Remove Even numbers but print odd numbers

```
import java.util.*;
class IteratorDemo{
    public static void main(String[]
            args){ List v=new Vector();
            for(int i=1;i<=10; i++){
                    v.add(i);
            }
```

```
            System.out.println(v);//[1, 2, 3, 4, 5, 6, 7, 8,9, 10]
            Iterator itr=v.iterator();
            while(itr.hasNext()){
                    Integer i=(Integer)itr.next();
                    if(i%2 == 0){ //Even number
                            itr.remove();
                    }
            }
            System.out.println(v);//[1, 3, 5, 7, 9]
    }
}
```

Advantages with Iterator

1) Applicable for non-legacy classes along with legacy classes.
2) Can remove element from the underlying collection.
3) Method names are improved.

Limitations with Enumeration and Iterator:

1. Both Enumeration and Iterator supports iteration in forward direction.
2. We cannot perform modify, add operations.

ListIterator <<inteface>>

It is the child interface of Iterator, it has introduced in 1.2 version.

The ListIterator is only applicable for List implementation classes.

The List interface have the following method, whose return type is ListIterator:

Public ListIterator **listIterator**()

This interface defines the following 9 methods.

1. Public Boolean **hasNext();**
2. Public Boolean hasPrevious();
3. Public Object **next();**
4. Public Object previous();
5. Public int nextIndex();
6. Public int previousIndex();
7. Public void **remove();** //remove operation
8. Public void **set(Object O);** //modify operation
9. Public void **add(Object new);** // We can add new object to the underlying collection.

Example:

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

```
        List players=new Vector();
        players.add("Dhoni");
        players.add("Sachin");
        players.add("Sehwag");
        players.add("Sourav");
        System.out.println(players);//[Dhoni, Sachin, Sehwag, Sourav]

        ListIterator itr= players.listIterator();
        while(itr.hasNext()){
            String s=(String)itr.next();
            if(s.equals("Sourav")){
                itr.remove();
                itr.add("Yuvraj");
                //itr.set("Yuvraj");
            }
        }
        System.out.println(players);//[Dhoni, Sachin, Sehwag, Yuvraj]
    }
}
```

Note:

Among all three iterator classes, the ListIterator is most powerfull iterator class. But the only limitation with this interface is, it is applicable only for List implementation classes.

Conotmparison table for Enumeration, Iterator, and ListIterator

Property	Enumeration	Iterator	ListIterator
Is legacy?	Yes	No	No
It is applicable for	Only legacy classes	All Collection implemented classes (both legacy & non-legacy)	Only for List implementation classes.
Navigation	Single direction (forward)	Single direction (forward)	Both directions (bi-directional).
How to get	Using **elements()** method	Using **iterator()** method	Using **listIterator()** method
Operations	Only read	Read and remove	Read, remove, modify, add
Methods	hasMoreElements(), nextElement()	hasNext() next() remove()	9 methods.

Sorting List Elements

The Collections class contains following overloaded sort() methods to sort List elements explicitly. Public static void **sort**(List) //natural sorting order

Public static void **sort**(List, Comparator)//customized sorting order

Example:

```
import java.util.Collections;
import java.util.List; import
java.util.Vector;
import java.util.Comparator;

public class VectorSortDemo {
        public static void main(String[] args){
                List fruits = new Vector();
                fruits.add("grapes");
                fruits.add("papaya");
                fruits.add("banana");
                fruits.add("pea");
                fruits.add("apple");
                System.out.println(fruits.toString());

                Collections.sort(fruits); // natural sorting order
                System.out.println(fruits.toString());

                Comparator c = Collections.reverseOrder();
                Collections.sort(fruits, c); // customized sorting
                order System.out.println(fruits);
        }
}
```

O/P:

```
[grapes, papaya, banana, pea, apple]
[apple, banana, grapes, papaya, pea]
[pea, papaya, grapes, banana, apple]
```

Set <<interface>>

It is inherited from Collection interface.

1. Set never allows duplicate elements.
2. The order of the Set elements depends on its implementation class.
    a. The elements in HashSet are unordered.
    b. The elements in LinkedHashSet are in insertion order.
    c. The elements in TreeSet are sorted (either Natural or Customized Sorting Order).
3. Set implementation classes allow at most one null value(Zero or One null value).
4. By default, some of the Set implementation classes are sorted.

5. By default, none of the Set implementation class is synchronized (non thread-safe). Use following utility method from Collections class to make our set elements as thread-safe.

   Public static Set synchronizedSet(Set)

6. The Set interface does not contain any new methods, hence we have to use Collection interface methods.
7. The HashSet uses hashing mechanism and LinkedHashSet uses both hashing and list mechanism.
8. Since both HashSet and LinkedHashSet uses hashing mechanism, hence overriding hashCode() method is mandatory.
9. Since both HashSet and LinkedHashSet are Set implementation classes, which never allows duplicate elements. To check duplicateness, overriding equals() is mandatory.
10. But, TreeSet elements are sorted using either compareTo() method from Comparable interface or compare() method from Comparator interface. The return value of the compareTo() or compare() method is used to check duplicate elements in TreeSet. Hence, overrding equals() and hashCode() methods for TreeSet elements is not required.

Overriding equals() and hashCode() methods

To avoid duplicate elements in HashSet and LinkedHashSet, overrding equals() method is mandatory.

Since both HashSet and LinkedHashSet uses hashing mechanism, overrding hashCode() method is mandatory.

Example:

```
class Employee{
    @Override
    public int hashCode() { }
    @Override
    public boolean equals(Object obj) {      }
}
```

HashSet

Internally uses hashing mechanism. Hence, overrding equals() and hashCode() methods are mandatory. Duplicate objects are not allowed.

HashSet is an unordered collection.
Allows almost one null value.

We never sort HashSet elements i.e., **explicitly also we cannot sort hashset** elements. By default, HashSet is not synchronized.

Example: Override both equals() and hashCode() methods for hashset elements.

```
class Employee{
    int eno; String
    ename; int
    sal;
    public Employee(int e, String name,int s ){
            eno = e;
            ename = name;
            sal = s;
    }
    @Override
    public int hashCode() {
            final int prime =
            31; int result = 1;
            result = prime * result + ((ename == null) ? 0 :
            ename.hashCode()); result = prime * result + eno;
            result = prime * result + sal;
            return result;
    }
    @Override
    public boolean equals(Object obj)
            { if (this == obj)
                    return true;
```

```
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            final Employee other = (Employee)
            obj; if (ename == null) {
                if (other.ename != null)
                    return false;
            } else if (!ename.equals(other.ename))
                return false;
            if (eno != other.eno)
                return false;
            if (sal != other.sal)
                return false;
            return true;
    }
}
public class HashSetDemo {
    public static void main(String[] args) {
            Employee e1 = new Employee(1,"abc", 1000);
            Employee e2 = new Employee(1,"abc", 1000);
            Set employees = new HashSet();
            employees.add(e1);
            employees.add(e2);
            System.out.println(employees.size()); //O/P: 1
    }
}
```

LinkedHashSet

This is exactly same as HashSet except the following differences:

HashSet	LinkedHashSet
HashSet uses Hashing mechanism to store and retrieve elements.	LinkedHashSet uses both hashing and linked list mechanism to store and retrieve element.
Unordered collection (Insertion order is not preserved).	**Ordered collection based on insertion order but not index order(Insertion order is preserved).**
Introduced in 1.2 version	Introduced in 1.4 version

Map <<interface>>

This is the **root interface** for all key and value based maps.
This is used to represent a group of **entries** as key and value pairs.
Map interface is not inhereited from Collection interface.
Keys are never duplicated, but values can be duplicated.
Some of the Map impl classes are ordered based on key.
Some of the Map impl classes allows both null key and null values.
By default, some of the Map impl classes are sorted based on key.
By default, some of the Map impl classes are synchronized.

Property	Key	Value
Duplicate	**Never**	Yes
Ordered	Depends on Map impl classes.	N/A
Null values	Zero/One	Zero/More
Sorted	Depends on Map impl class.	N/A
Synchronized	Some of the Map impl classes are synchronized	

Example:

Key	Value
"A"	Apple
"B"	Ball
"C"	Cat
"D"	Dog

Methods

1. **Public Object put(Object key, Object value);**

To insert one Key & value pair in the Map. If the specified key is already present then old value is replaced with new value and old value will be returned. If the key is not present, then null will be returned.

2. **Public Object get(Object key);**
   Return the value associated with the key. If the specified key is not available then it returns null.
3. **Public Object remove(Object key);**
   To remove the key and value entry and return value or null if the key does not exist.
4. **Public Void clear();**
   Remove all entries.
5. **Public Int size();**
   Number of entries in the Map.
6. **Public Set keySet();**Returns the Set view of the keys, since keys cannot be duplicated.
7. **Public Collection values();**Returns Collection view of the values

Except for TreeMap, for all other Map implementation classes, overriding both hashCode() and equals() methods required for keys.

```
class Employee2 {
        @Override
        public int hashCode() {  }

        @Override
        public boolean equals(Object obj) {   }
}
```

Hashtable

Underlying data structure is hashing mechanism.

Keys cannot be duplicated but values can be duplicated.

Both keys and values are unordered i.e., the entries in Hashtable are unordered.

Neither key nor value is null.

The Hashtable entries are not sorted.

By default, Hashtable is a synchronized Map.

Two parameters which affects the hashtable performance are **Capacity** and **Load factor**. **The capacity is the number of buckets in the hash table. The load factor is a measure of when hash table is automatically increased. The default load factor is 0.75.**

The following methods are used to retrieve values from the hashtable.

Enumeration elements();
Collection values(); } //values

The following methods are used to retrieve keys from the hashtable.

Enumeration keys();
Set keySet(); } //Keys

Example: Use built-in class String as Key.

```
import java.util.Hashtable;
public class HashtableDemo {
        public static void main(String[] args) {
                Hashtable ht = new Hashtable();
                ht.put("one", 1);//autoboxing
                ht.put("two", 2); ht.put("three",
                3);
                System.out.println(ht);  //{two=2, one=1, three=3}

                Enumeration keys = ht.keys();
                //Set keys = ht.keySet();
                while(keys.hasMoreElements()){
                String str = (String)keys.nextElement();
                        System.out.print(str+"\t");
                }

                System.out.println();

                //Retrieve values from hashtable.
                Enumeration values = ht.elements();
                //Collection values = ht.values();
                while(values.hasMoreElements()){
                        Integer iRef = (Integer)values.nextElement();
                        System.out.print(iRef + "\t");
                }
        }
}
```

HashMap

The underlying data structure is hashing mechanism.

Keys cannot be duplicated but values can be duplicated. The HashMap entries are unordered.

Allows both null key and null values.
HashMap entries are not sorted.

By default, HashMap is not synchronized. But, we can explicitly synchronize HashMap by using following utility method from Collections class:

Public static Map synchronizedMap(Map m);

Example:

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

```
class HashMapDemo{
        public static void main(String[] args){
                HashMap m=new HashMap();
                m.put("Laptops",10);
                m.put("Memory sticks",20);
                m.put("Pens",30); m.put(null,
                null);
                System.out.println(m); //{null=null, Pens=30, Laptops=10, Memory sticks=20}
        }
}
```

Difference between HashMap and Hashtable

HashMap	Hashtable
By default, it is not synchronized (non thread-safe)	By default, it is synchronized (thread-safe).
Performance is high	Performance is low
Introduced in 1.2 version	Legacy, introduced in 1.0 version.
Both key and values can be null.	Neither key nor value is null.

Properties

Application properties and Server or System properties are usually configured in property file with .properties extension.
The entries in property file are key, values and separated with equal (=) symbol.

Though the Properties class is inherited from Hashtable, but both keys and values must be string objects only.
The advantages of using properties file is, the changes can be made in one place, which are reflected across all other placess in our code.

Methods:

1. String getProperty(String Propertyname)
   Return the value associated with the specified property or null if the property does not exist.
2. String setProperty(String popertyname, String propertyvalue);
3. Void load(InputStream is);
   To load the properties from properties file into java properties object.

Example:

```
#connection.properties
hostname=localhost
username=scott
password=tiger

//PropertiesDemo.java import
java.io.FileInputStream; import
java.util.Properties; public
class PropertiesDemo {
        public static void main(String[] args) throws Exception{
```

```
                Properties props = new Properties();
                props.load(new
                FileInputStream("connection.properties"));
                System.out.println(props);
                String s = props.getProperty("hostname");
                props.setProperty("port", "1521");
                System.out.println(props);
        }
}
```

O/P:

```
{hostname=localhost, password=tiger, username=scott}
{port=1521, hostname=localhost, password=tiger, username=scott}
```

Generic types (type safe)

Generic types (type safe) were introduced in **jdk1.5**.

To insert only string elements into ArrayList, we can create generic (type safe) version of ArrayList as follows: ArrayList**<String>** names = new ArrayList**<String>**();

For the above ArrayList, we can add only string elements. If we are trying to insert other than string elements, we will get compilation error itself. Hence, Generics are type safe. Generic types are resolved at compile time.

Example:

Base type Generic or Parameter type

```
ArrayList<String> fruits = new ArrayList<String>(); //ArrayList holds only String objects.
names.add("apple");
names.add("grapes");
names.add("banana");
//names.add(new Integer(10));//C.E. Cannot find symbol add(Integer)
```

Hence, it is guaranteed that all elements in generic collection are similar types.

While retrieving elements, we can directly assign to String variable without explicit type casting. String fruit = fruits.get(0);

It is always recommended to use enhanced-for loop to retrieve elements from Generic collection as follows:

```
for(String fruit : fruits)
    SOP(fruit);
```

IO STREAMS and SERIALIZATION

[java.io package]

File
InputStream
OutputStream
Reader
Writer
Serialization

Java.io.File<<class>>

This class is used to create New Files (Empty files), New Directories, Setting file or directory permissions. The File class is not meant for reading or writing file content.

Example:

```
import java.io.File;
import java.io.IOException;
public class FileDemo {
        public static void main(String[] args)throws IOException
                { File file = new File("abc.txt");
                If(!file.exists()){
                        file.createNewFile();// empty file is created.
                }
                System.out.println(file.length()); //0
        }
}
```

Byte Streams: InputStream and OutputStream

The abstract classes InputStream and OutputStream are the **root** of the inheritance hierarchies for **reading and writing of bytes** (**Image data**).

Note: There are different input and output sources: file, keyboard (only input source), monitor (only output source), socket, byte array (byte[]), character array (char[]), browser, etc.

The subclasses of the InputStream are used to read one byte at a time from one of the corresponding input source. The following diagram detailing the InputStream subclass hierarchy:

<<interface>>
DataInput

<<interface>>
ObjectInput

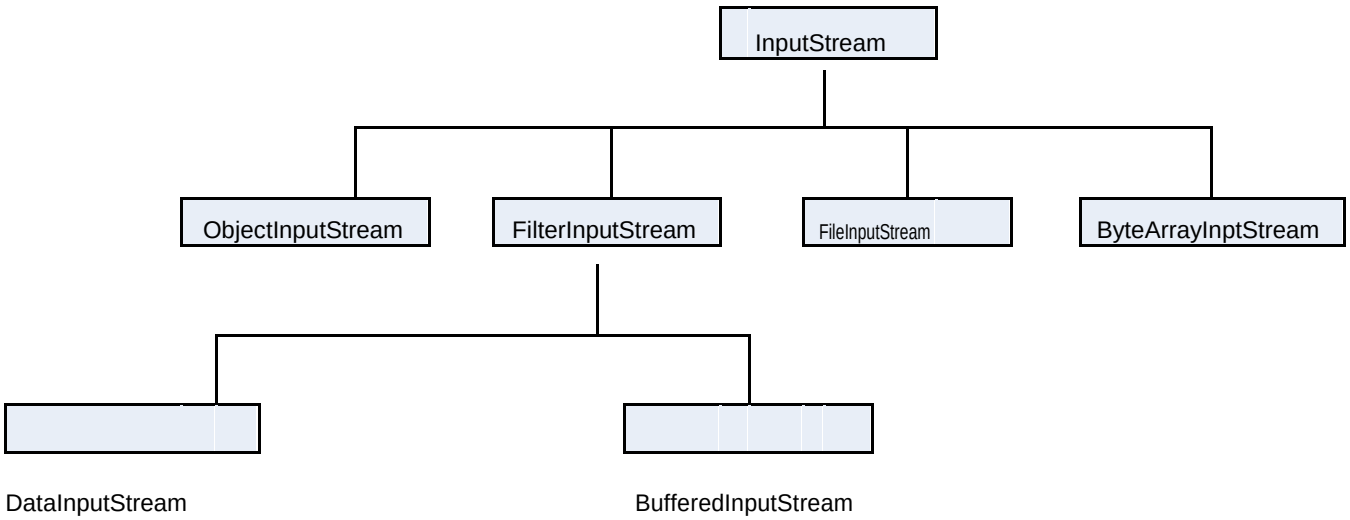


Figure 1: InputStream Inheritance hierarchy

The following methods from InputStream are used to read once byte at a time from the underlying input source.

int read() throws IOException

int read(byte[]) throws IOException

int read(byte[], int offset, int len)throws IOException

The read() method reads a byte but returns an int value. The byte read resides in the eight least significant bits of the int value, while the remaining 24 bits in the int value are **zeroed out**. **The read() method returns -1 when the end of the stream is reached.**

The following diagram detailing the OutputStream subclasses:

<<interface>>
DataOutput

<<interface>>
ObjectOutput

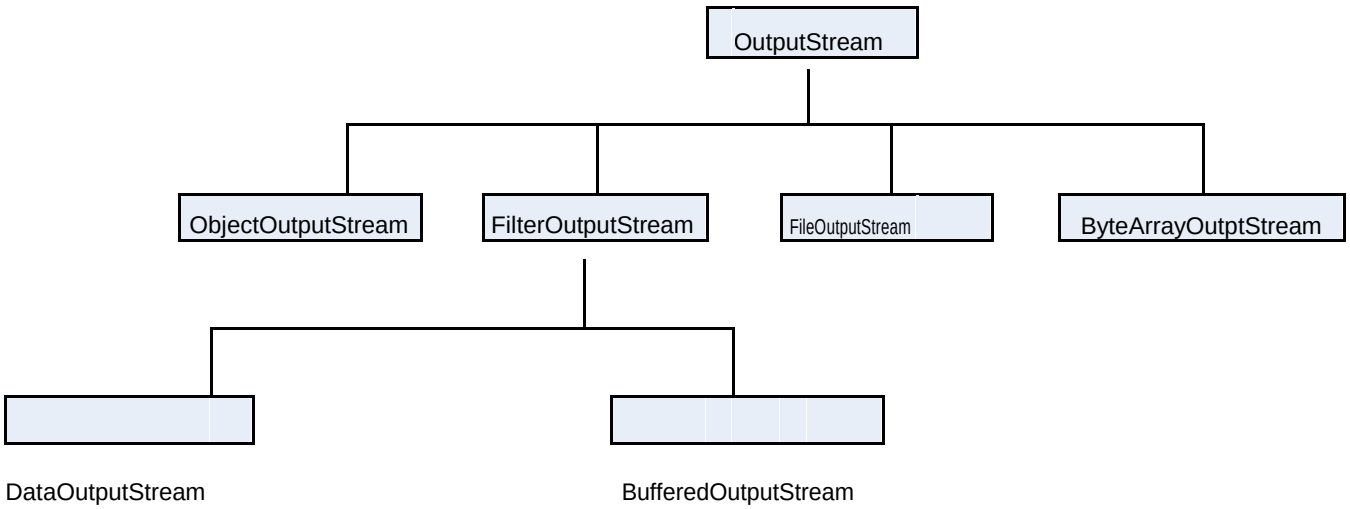


Figure 2: OutputStream Inheritance hierarchy

The following methods from OutputStream are used to write one byte at a time to the underlying output source.

void write(int) throws IOException

void write(byte[]) throws IOException

void write(byte[], int offset, int len)throws IOException

The write() method takes an int as argument, but **truncates** it down to the eight least significant bits before writing it out as a byte.

void flush() throws IOException Only for OutputStream void close() throws IOException

A stream should be closed when no longer needed, to free system resources. Closing an output stream automatically flushes the stream, meaning that any data in its internal buffer is written out. An output stream can also be manually flushed by calling flush() method.

Note:

1. Read and write operations on streams are synchronous operations.
2. FileNotFoundException, EOFException, and IOException are checked exceptions.

FileStreams: FileInputStream & FileOutputStream

FileInputStream and FileOutputStream classes are used to read and write one byte at a time from/to a file. The following are the FileInputStream constructors:

FileInputStream(String fileName)throws FileNotFoundException

FileInputStream(File fileobject)throws FileNotFoundException

If the file does not exists, a FileNotFoundException is thrown. A SecurityException is thrown if the file does not have read access.

The following are the FileOutputStream constructors:

FileOutputStream(String fileName)throws FileNotFoundException
FileOutputStream(String filename, boolean append)throws
FileNotFoundException FileOutputStream(File fileobject)throws
FileNotFoundException

If the file does not exist, it is created. If it exists, its contents are reset, unless the appropriate constructor is used to indicate that output should be appended to the file. A SecurityException is thrown if the file does not have write access or it cannot be created. FileNotFoundException is thrown if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason.

Example: Copy image file

```
import java.io.FileInputStream;
import
java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class CopyImageFile {
        public static void main(String[] args) {
                FileInputStream fis = null;
                FileOutputStream fos = null;
                if(args.length !=2 ){
                        System.out.println("Usage: java CopyFile <from_file> <to_file>");
                        return;
                }
                try {
                        fis = new FileInputStream(args[0]);
                        fos = new FileOutputStream(args[1]);
                        int b;
                        while((b=fis.read())!=-
                                1){ fos.write(b);
                        }
                } catch (FileNotFoundException e)
                        { e.printStackTrace();
                }catch (IOException e) {
                        e.printStackTrace();
                }finally{
                        try {
                                fos.close();
                                fis.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }
```

```
}
```

Filter Streams

A filter stream is a high-level stream that provides **additional functionality** to the underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter.

The **BufferedInputStream** is a filter class, which provides buffer(or cache) to read large data from the input source (such as file) at once into the buffer. When we ask for next character or line of data,it is retrieved from the buffer, which minimizes the number of hits with file.

The **BufferedOutputStream** is a filter class, which provides buffer(or cache) to write large data to the output source (such as file) at once from the buffer. Hence, every write operation not necessarily write directly to hard disk file, rather, it is initially written to the buffer. Once buffer is full, then the complete data will be written once into the file.

The main advantage of using above two classes is to **enhance performance**.

Example:

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import
java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
public class FilterStreamsDemo {
        public static void main(String[] args) {
                BufferedInputStream bis = null;
                BufferedOutputStream bos = null;
                if(args.length !=2 ){
                        System.out.println("Usage: java FilterStreamDemo <from_file> <to_file>");
                        return;
                }
                try {
                        bis = new BufferedInputStream(new FileInputStream(args[0])); bos
                        = new BufferedOutputStream(new FileOutputStream(args[1])); int
                        b;
                        while((b=bis.read())!=-1){
                                bos.write(b);
                        }
                } catch (FileNotFoundException e)
                        { e.printStackTrace();
                }catch (IOException e) {
                        e.printStackTrace();
                }finally{
```

```
                try {
                        bos.close();
                        bis.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
    }
}
```

The **DataInputStream** is a filter class, which converts binary representation of java primitive values into actual primitive values.

The **DataOutputStream** is a filter class, which converts actual primitive values into equivalent binary format.

Indeed, all the examples we have looked at thus far have dealt with streams of text data that required nothing more than casting each int returned to a character before it was used.

```
char c = (char)someinputstream.read();
```

And for most primitive data types such as long, float, and double, casting is not appropriate. The task of converting data to its native type is the perfect job for a filter stream.

Methods:

The **writeX()**, where X is any primitive type, is used to write actual primitive value into equivalent binary format.
The **readX()**, where X is any primitive type, is used to read binary representation of primitive value and convert it into actual primitive value.
The readX() method throws **EOFException** if this input stream reaches the end rather than -1.

Constructors:

The following constructors can be used for reading and writing java primitive values:

```
DataInputStream(InputStream)
DataOutputStream(OutputStream)
```

Example:

```
import java.io.*;
public class DataInputOutputStreamDemo {
        public static void main(String[] args) {
                try {
                        DataOutputStream dos = new DataOutputStream(new FileOutputStream("info.dat"));
                        dos.writeByte(10);
                        dos.writeInt(20);
                        dos.writeFloat(30.0F);
```

```
                dos.writeDouble(40.0);
                dos.close();

                DataInputStream dis = new DataInputStream(new
                FileInputStream("info.dat")); byte b = dis.readByte();
                int i = dis.readInt();
                float f = dis.readFloat();
                double d = dis.readDouble();

                System.out.println(b); //10
                System.out.println(i); //20
                System.out.println(f);//30.0
                System.out.println(d);//40.0
            } catch (FileNotFoundException e) {
                System.out.println("File not found");
            } catch (EOFException e) {
                System.out.println("End of stream");
            } catch (IOException e) {
                System.out.println("Input error");
            }
        }
}
```

Note: Read java primitive values in the same order in which they were written.

Conclusion: The readX() methods, where X is one of the primitive data type, returns primitive value or EOFException if it reaches end of the stream. Whereas read() method returns int value equivalent to read byte0020or -1 in case of end of the stream.

ByteArrayInputStream & ByteArrayOutputStream

These classes are used to store and retrieve data temparary.

Example:

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream; import
java.io.DataOutputStream;
public class ByteArrayOutputStreamDemo {
    public static void main(String[] args) throws Exception{
        ByteArrayOutputStream baos=new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);//writing to byte[] buffer
        dos.writeInt(10);
        dos.writeInt(20);
        ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
```

```
            DataInputStream dis = new DataInputStream(bais); //reading from above byte[]
            System.out.println(dis.readInt()); //10
            System.out.println(dis.readInt()); //20
        }
}
```

Note: We can write our own I/O stream classes to access special storage devices such as tape, zip, etc from java application.

Here are some additional stream classes available in the java.util.zip and java.util.jar packages:

GZIPInputStream	GZIPOutputStream
ZipInputStream	ZipOutputStream
JarInputStream	JarOutputStream

Character Streams: Readers and Writers

The abstract classes Reader and Writer are root classes used to read and write one character at a time. The following diagram detailing the Reader subclasses:

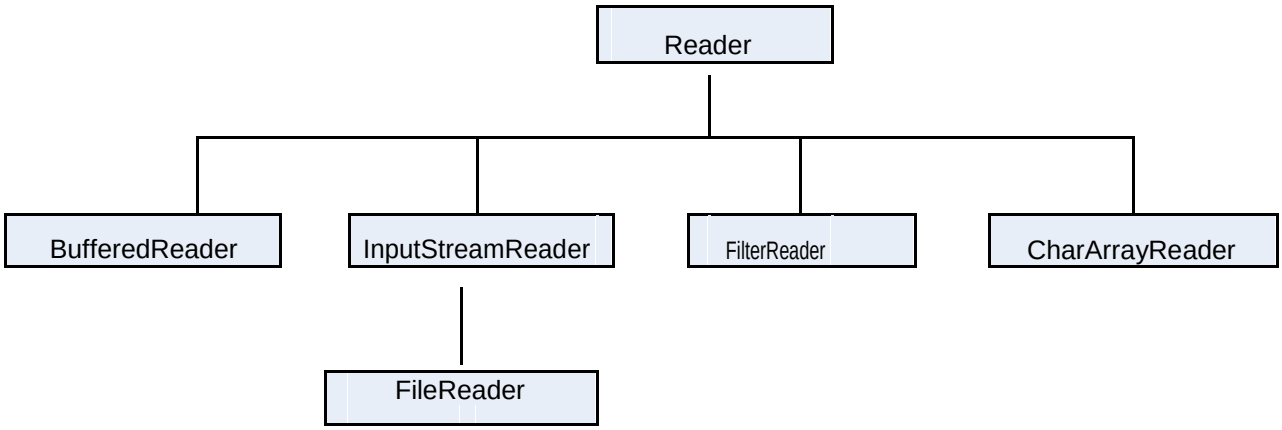


Figure 3: Reader Inheritance hierarchy

The following methods from Reader are used to read from the underlying input source **one character (two bytes) at a time**:

```
int read() throws IOException
int read(char[]) throws IOException
int read(char[], int offset, int len)throws IOException
```

The read() method reads the character as an int in the range 0 to 65535. The 2 bytes read resides in the 16 least significant bits of the int value, while the remaining 16-bits in the int value are zeroed out. The read() method reurns **-1** when the end of the stream has been reached.

The following diagram detailing the Writer subclasses:

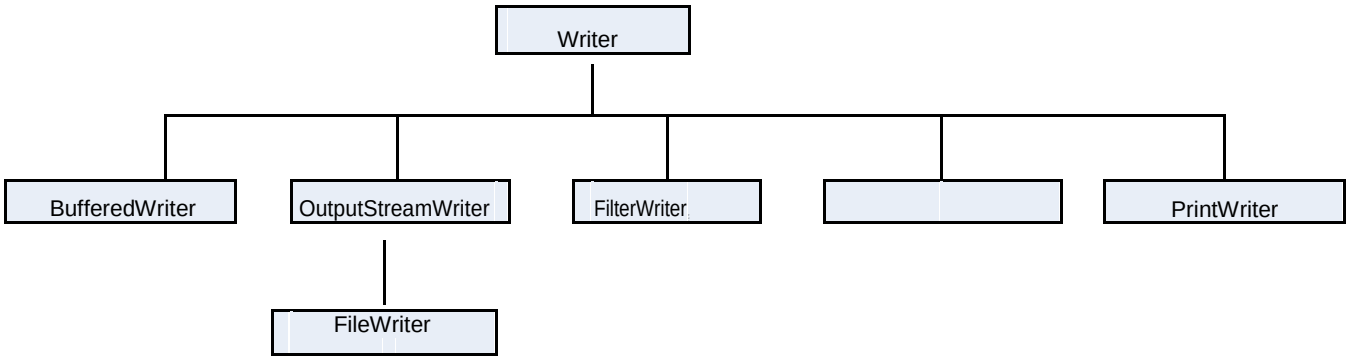


Figure 4: Writer inheritance hierarchy

The following methods from Writer class are used to write to the underlying output source one character (two bytes) at a time:

void write(int) throws IOException void
write(char[]) throws IOException

void write(char[], int offset, int len)throws IOException
void write(String str)throws IOException

void write(String str, int off, int length)throws IOException

The write() method takes an int as argument, but **truncates** it down to the 16 least significant bits before writing it out as a character.

void flush() throws IOException Only for Writer void
close() throws IOException

A stream should be closed when no longer needed, to free system resources. **Closing an output stream automatically flushes the stream, meaning that any data in its internal buffer is written out. An output stream can also be manually flushed by calling flush() method.**

FileReader & FileWriter

These classes are used to read or write character by character from the underlying files.

Example: Copying text file

```
import java.io.FileReader;
import java.io.FileWriter;
public class CopyTextFile {
    public static void main(String[] args)throws Exception {
        FileReader fr = new
        FileReader(args[0]); FileWriter fw =
        new FileWriter(args[1]); int ch;
        while((ch=fr.read())!=-1){
            fw.write(ch);
        }
        fr.close();
        fw.close();
```

```
	}
}
```

BufferedReader & BufferedWriter

The main advantage of using these classes is to enhance performance.

Use **readLine()** method from BufferedReader class to read one line at a time.
Use **newline()** method from BufferedWriter class to include new line.

Example:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
public class BufferedWriterDemo {
	public static void main(String[] args)throws Exception {
		BufferedWriter bw = new BufferedWriter(new
		FileWriter("abc2.txt")); bw.write(100);
		bw.newLine();
		char[] buff = {'a','b','c'};
		bw.write(buff);
		bw.newLine();
		bw.write("aspire");
		bw.newLine();
		bw.write("technologies");
		bw.flush();
		bw.close();

		BufferedReader br = new BufferedReader(new
		FileReader("abc2.txt")); String str=null;
		while((str=br.readLine())!=null)
			System.out.println(str);
	}
}
```

PrintWriter

The following method are defined in PrintWriter class.

print(int)	println(int)
print(char)	println(char);
print(boolean)	println(boolean)
print(double)	println(double)
print(String)	println(String)

Read From Keyboard

The **InputStreamReader** is a bridge class between binary data & text data. import java.io.BufferedReader;

import java.io.IOException; import java.io.InputStreamReader; public class ReadFromKB {

```
    public static void main(String[] args)throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Read x:");
        int x = Integer.parseInt(br.readLine());

        System.out.println("Read y:");
        int y = Integer.parseInt(br.readLine());

        int sum = x + y;

        System.out.println("The total="+sum);
    }
}
```

Serialization

Serialization is the process of **producing Stream of bytes** from the object and writing it into the output source such as file, socket, etc.

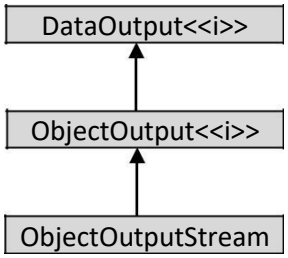
The serialization also keeps sufficient information to restore the fields during deserialization process.

To produce stream of bytes, the class should be inherited either from **java.io.Serializable** interface.

To store/save/persist an object in a stream (file, socket, etc) which is outside the JVM, the class must be inherited from java.io.Serializable interface, which is marker interface.

ObjectOutputStream

The class **ObjectOutputStream** provides both object as well as primitive **serialization** since it is inherited from ObjectOutput (contains methods to serialize objects) and ObjectOutput intern inherited from DataOutput (contains methods to serialize primitive data) interfaces.



Objects are serialized using **writeObject()** method and primitive data is serialized using **writeType()** methods, where 'type' is one of the primitive type.

ObjectInputStream

The class **ObjectInputStream** provides both object and primitive data **deserialization** since it is inherited from ObjectInput (contains methods to deserialize objects) and ObjectInput intern inherits DataInput (contains methods to deserialize primitive data) interfaces.

DataInput<<>i>

ObjectInput<<i>>

ObjectInputStream

Objects are deserialized using **readObject()** method and primitives values are deserialized using **readType()** methods, the 'type' is primitive type.

Example:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
class Authentication implements Serializable{
        private String name;
        transient private String password;
        Authentication(String name, String password){
                this.name = name;
                this.password = password;
        }
        public String toString(){
                return "Name:"+this.name+"    Passowrd:"+this.password;
        }
}
public class ObjectSerialization{
        public static void main(String[] args)throws Exception{
                Authentication a1 = new Authentication("aspire", "aspire123");
```

```
                    ObjectOutputStream oos = new
        ObjectOutputStream(new FileOutputStream("store.ser"));
                oos.writeObject(a1);
                oos.close();

                ObjectInputStream ois = new ObjectInputStream(new FileInputStream("store.ser"));
                Authentication a = (Authentication)ois.readObject();
                System.out.println(a.toString());
        }
}
```

O/P:

Name:aspire Passowrd:null

Note:

1. The object which is being serializable must be inherited from java.io.Serializable interface, otherwise, **java.io.NotSerializableException** will be thrown.
2. The **transient** modifier is used to not to save during serialization.