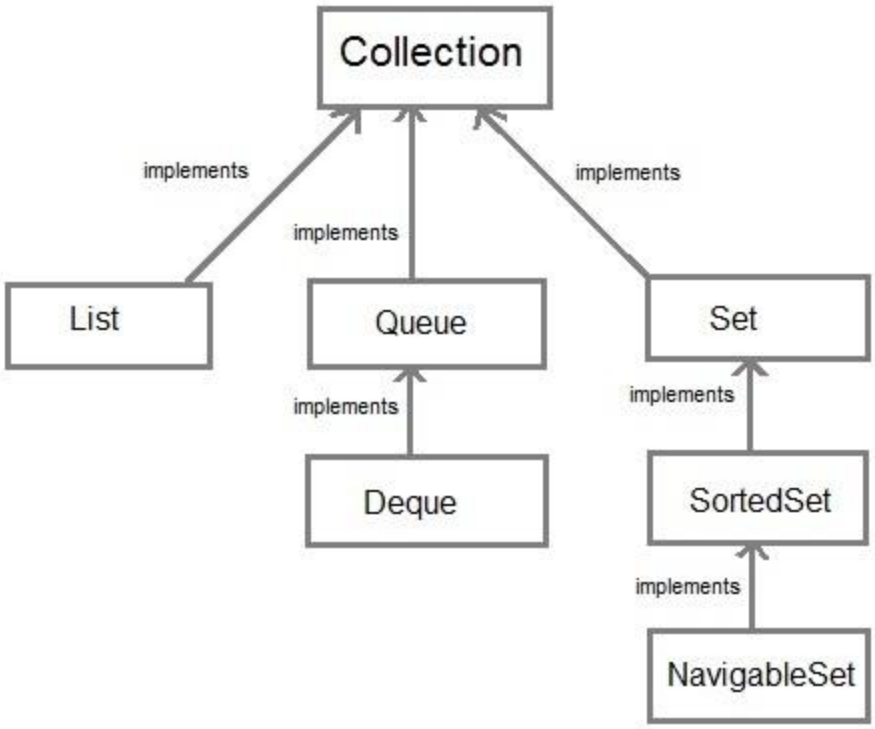
COLLECTION

Collection Framework

Collection framework was not part of original Java release. Collections was added to J2SE 1.2. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects. Collection framework provides many important classes and interfaces to collect and organize group of alike objects.

Important Interfaces of Collection API

Interface	Description
Collection	Enables you to work with groups of object; it is at the top of collection hierarchy
Deque	Extends queue to handle double ended queue.
List	Extends collection to handle sequences list of object.
Queue	Extends collection to handle special kind of list in which element are removed only from the head.
Set	Extends collection to handle sets, which must contain unique element.
SortedSet	Extends sets to handle sorted set.



All these Interfaces give several methods which are defined by collections classes which implement these interfaces.

Why Collections were made Generic ?

Generics added **type safety** to Collection framework. Earlier collections stored **Object class** references. Which means any collection could store any type of object. Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch. Hence Generics was introduced, now you can explicitly state the type of object being stored.

Collections and Autoboxing

We have studied that Autoboxing converts primitive types into Wrapper class Objects. As collections doesn't store primitive data types(stores only refrences), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

Most Commonly thrown Exceptions in Collection Framework

Exception Name	Description
UnSupportedOperationException	occurs if a Collection cannot be modified
ClassCastException	occurs when one object is incompatible with another
NullPointerException	occurs when you try to store null object in Collection
IllegalArgumentException	thrown if an invalid argument is used
IllegalStateException	thrown if you try to add an element to an already full Collection

Interfaces of Collection Framework

The collection framework has a lot of Interfaces, setting the fundamental nature of various collection classes. Lets study the most important Interfaces in the collection framework.

The Collection Interface

1. It is at the top of collection heirarchy and must be implemented by any class that defines a collection. Its general declaration is,

   ```
   interface Collection < E >
   ```

2. Following are some of the commonly used methods in this interface.

Methods	Description
add(E obj)	Used to add objects to a collection. Doesn't add duplicate elements to the collection.
addAll(Collection C)	Add all elements of collection C to the invoking collection
remove(Object obj)	To remove an object from collection
removeAll(Collection C)	Removes all element of collection C from the invoking collection
contains(Object obj)	To determine whether an object is present in collection or not
isEmpty()	Returns true if collection is empty, else returns false
size()	returns number of elements present in collection

The List Interface

1. It extends the **Collection** Interface, and defines storage as sequence of elements. Following is its general declaration,

```
interface List < E >
```

2. Allows random access and insertion, based on position.
3. It allows Duplicate elements.
4. Apart from methods of Collection Interface, it adds following methods of its own.

Methods	Description
get(int index)	Returns object stored at the specified index
set(int index, E obj)	Stores object at the specified index in the calling collection
indexOf(Object obj)	Returns index of first occurence of obj in the collection
lastIndexOf(Object obj)	Returns index of last occurence of obj in the collection
subList(int start, int end)	Returns a list containing elements between start and end index in the collection

---

The Set Interface

1. This interface defines a Set. It extends **Collection** interface and doesn't allow insertion of duplicate elements. It's general declaration is,

```
interface Set < E >
```

2. It doesn't define any method of its own. It has two sub interfaces, **SortedSet** and **NavigableSet**.
3. **SortedSet** interface extends **Set** interface and arranges added elements in an ascending order.
4. **NavigabeSet** interface extends **SortedSet** interface, and allows retrieval of elements based on the closest match to a given value or values.

The Queue Interface

1. It extends **collection** interface and defines behaviour of queue, that is first-in, first-out. It's general declaration is,

```
interface Queue < E >
```

2. There are couple of new and interestin methods added by this interface. Some of them are mentioned in below table.

Methods	**Description**
poll()	removes element at the head of the queue and returns **null** if queue is empty
remove()	removes element at the head of the queue and throws**NoSuchElementException** if queue is empty
peek()	returns the element at the head of the queue without removing it. Returns **null** if queue is empty
element()	same as peek(), but throws **NoSuchElementException** if queue is empty
offer(E obj)	Adds object to queue.

The Dequeue Interface

1. It extends **Queue** interface and declares behaviour of a double-ended queue. Its general declaration is,

```
interface Dequeue < E >
```

2. Double ended queues can function as simple queues as well as like standard Stacks.

The Collection classes

Java provides a set of Collection classes that implements Collection interface. Some of these classes provide full implementations that can be used as it is and other abstract classes provides skeletal implementations that can be used as starting points for creating concrete collections.

ArrayList class

1. ArrayList class extends **AbstractList** class and implements the **List** interface.
2. ArrayList supports dynamic array that can grow as needed. ArrayList has three constructors.

```
ArrayList()

ArrayList( Collection C )

ArrayList( int capacity )
```

3. ArrayLists are created with an initial size, when this size is exceeded, it gets enlarged automatically.
4. It can contain Duplicate elements and maintains the insertion order.
5. ArrayLists are not synchronized.

Example of ArrayList

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

```
  ArrayList< String> al = new ArrayList< String>();
  al.add("ab");
  al.add("bc");
  al.add("cd");
  system.out.println(al);
 }
}
```

`Output : [ab,bc,cd]`

Getting Array from an ArrayList

`toArray()` method is used to get an araay containing all the contents of the list. Following are the reasons why you must obtain array from your ArrayList whenever required.

- To obtain faster processing.
- To pass array to methods who do not accept Collectionn as arguments.
- To integrate and use collections with legacy code.

Storing User-Defined classes

In the above example we are storing only string object in ArrayList collection. But You can store any type of object, including object of class that you create in Collection classes.

Example of storing User-Defined object

Contact class

```
class Contact
{
    String first_name;
    String last_name;
```

```
    String phone_no;


    public Contact(String fn,String ln,String pn)
    {
    first_name = fn;
    last_name = ln;
    phone_no = pn;
    }


    public String toString()
    {
        return first_name+" "+last_name+"("+phone_no+")";
    }
}
```

Storing Contact class

```
public class PhoneBook
{


   public static void main(String[] args)
   {
       Contact c1 = new Contact("Ricky", "Pointing","999100091");
       Contact c2 = new Contact("David", "Beckham","998392819");
       Contact c3 = new Contact("Virat", "Kohli","998131319");


    ArrayList< Contact> al = new ArrayList< Contact>();
     al.add(c1);
     al.add(c2);
     al.add(c3);
     System.out.println(al);
```

```
   }


}
```

Output

```
[Ricky Pointing(999100091), David Beckham(998392819), Virat Kohli(998131319)]
```

LinkedList class

1. LinkedList class extends **AbstractSequentialList** and implements **List**,**Deque** and **Queue** inteface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It can contain duplicate elements and is not synchronized.

Example of LinkedList class

```
import java.util.* ;

class Test

{

 public static void main(String[] args)

 {

  LinkedList< String> ll = new LinkedList< String>();

  ll.add("a");

  ll.add("b");

  ll.add("c");

  ll.addLast("z");

  ll.addFirst("A");

  System.out.println(ll);

 }
```

```
}
```

```
Output : [A, a, b,c, z]
```

HashSet class

1. HashSet extends **AbstractSet** class and implements the **Set** interface.
2. It creates a collection that uses hash table for storage.
3. HashSet does not maintain any order of elements.

Example of HashSet class

```
import java.util.*;

class HashSetDemo

{

 public static void main(String args[])

 {

  HashSet< String> hs = new HashSet< String>();

  hs.add("B");

  hs.add("A");

  hs.add("D");

  hs.add("E");

  hs.add("C");

  hs.add("F");

  System.out.println(hs);

 }

}
```

```
Output: [D, E, F, A, B, C]
```

LinkedHashSet Class

1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet maintains a linked list of entries in the set.
3. LinkedHashSet stores elements in the order in which elements are inserted.

Example of LinkedHashSet class

```
import java.util.*;

class LinkedHashSetDemo

{

 public static void main(String args[])

 {

  LinkedHashSet< String> hs = new LinkedHashSet< String>();

  hs.add("B");

  hs.add("A");

  hs.add("D");

  hs.add("E");

  hs.add("C");

  hs.add("F");

  System.out.println(hs);

 }

}
```

```
Output : [B, A, D, E, C, F]
```

TreeSet Class

1. It extends **AbstractSet** class and implements the **NavigableSet** interface.

2. It stores elements sorted ascending order.
3. Uses a Tree structure to store elements.
4. Access and retrieval times are quite fast.
5. It has four Constructors.

```
TreeSet()

TreeSet( Collection C )

TreeSet( Comparator comp )

TreeSet( SortedSet ss )
```

Accessing a Collection

To access, modify or remove any element from any collection we need to first find the element, for which we have to cycle throught the elements of the collection. There are three possible ways to cycle through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

Accessing elements using Iterator

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide **iterator()** method to return an iterator.

```
import java.util.*;

class Test_Iterator

{

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

```
 {
  ArrayList< String> ar = new ArrayList< String>();
  ar.add("ab");
  ar.add("bc");
  ar.add("cd");
  ar.add("de");

 Iterator it = ar.iterator();     //Declaring Iterator
  while(it.hasNext())
  {
   System.out.print(it.next()+" ");
  }
 }
}
```

```
Output : ab bc cd de
```

Accessing element using ListIterator

ListIterator Interface is used to traverse a list in both forward and backward direction. It is available to only those collections that implement the **List** Interface.

```
import java.util.*;
class Test_Iterator
{
 public static void main(String[] args)
 {
  ArrayList< String> ar = new ArrayList< String>();
  ar.add("ab");
  ar.add("bc");
  ar.add("cd");
```

```
  ar.add("de");


  ListIterator litr = ar.listIterator();
  while(litr.hasNext())                 //In forward direction
  {
   System.out.print(litr.next()+" ");
  }


  while(litr.hasPrevious())             //In backward direction
  {
   System.out.print(litr.next()+" ");
  }
 }
}
```

Output :

```
  ab bc cd de
  de cd bc ab
```

Using for-each loop

`for-each` version of for loop can also be used for traversing each element of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any reverse access. `for-each` loop can cycle through any collection of object that implements Iterable interface.

```
import java.util.*;

class ForEachDemo

{

 public static void main(String[] args)

 {
```

```
  LinkedList< String> ls = new LinkedList< String>();
  ls.add("a");
  ls.add("b");
  ls.add("c");
  ls.add("d");

  for(String str : ls)
  {
   System.out.print(str+" ");
  }
 }
}
```

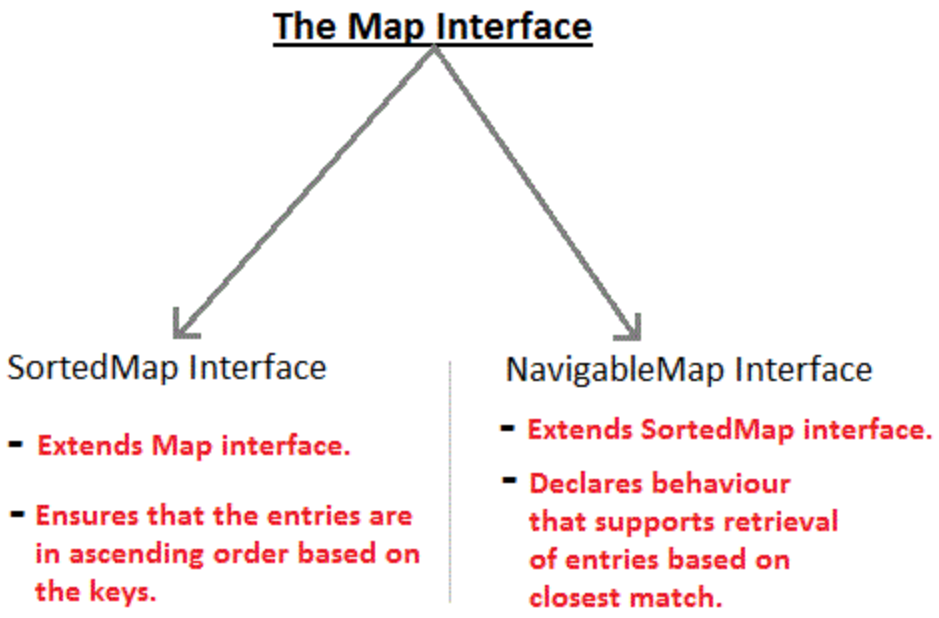
```
Output : a b c d
```

Map Interface

A Map stores data in key and value association. Both key and values are objects. The key must be unique but the values can be duplicate. Although Maps are a part of Collection Framework, they can not actually be called as collections because of some properties that they posses. However we can obtain a **collection-view** of maps.

Interface	Description
Map	Maps unique key to value.
Map.Entry	Describe an element in key and value pair in a map. This is an inner class of map.
NavigableMap	Extends SortedMap to handle the retrienal of entries based on closest match searches

SortedMap	Extends Map so that key are maintained in an ascending order.



Commonly used Methods defined by Map

- boolean **containsKey**(Object *k*): returns true if map contain *k* as key. Otherwise false.
- Object **get**(Object *k*) : returns values associated with the key *k*.
- Object **put**(Object *k*, Object *v*) : stores an entry in map.
- Object **putAll**(Map *m*) : put all entries from *m* in this map.
- Set **keySet**() : returns **Set** that contains the key in a map.
- Set **entrySet**() : returns **Set** that contains the entries in a map.

HashMap class

1. HashMap class extends **AbstractMap** and implements **Map** interface.

2. It uses a **hashtable** to store the map. This allows the execution time of `get()` and `put()` to remain same.
3. HashMap has four constructor.

```
HashMap()
HashMap(Map< ? extends k, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillratio)
```

4. HashMap does not maintain order of its element.

Example

```
import java.util.*;
class HashMapDemo
{
 public static void main(String args[])
 {
  HashMap< String,Integer> hm = new HashMap< String,Integer>();
  hm.put("a",new Integer(100));
  hm.put("b",new Integer(200));
  hm.put("c",new Integer(300));
  hm.put("d",new Integer(400));

  Set< Map.Entry< String,Integer> > st = hm.entrySet();    //returns Set view
  for(Map.Entry< String,Integer> me:st)
  {
   System.out.print(me.getKey()+":");
   System.out.println(me.getValue());
  }
 }
```

```
}
```

Output:

```
c 300

a 100

d 400

b 200
```

TreeMap class

1. TreeMap class extends **AbstractMap** and implements **NavigableMap** interface.
2. It creates Map, stored in a tree structure.
3. A **TreeMap** provides an efficient means of storing key/value pair in efficient order.
4. It provides key/value pairs in sorted order and allows rapid retrieval.

Example

```
import java.util.*;

class TreeMapDemo

{

 public static void main(String args[])

 {

  TreeMap< String,Integer> tm = new TreeMap< String,Integer>();

  tm.put("a",new Integer(100));

  tm.put("b",new Integer(200));

  tm.put("c",new Integer(300));

  tm.put("d",new Integer(400));


  Set< Map.Entry< String,Integer> > st = tm.entrySet();

  for(Map.Entry me:st)

  {
```

```
   System.out.print(me.getKey()+":");
   System.out.println(me.getValue());
  }
 }
}
```

Output:

```
a 100
b 200
c 300
d 400
```

LinkedHashMap class

1. **LinkedHashMap** extends **HashMap** class.
2. It maintains a linked list of entries in map in order in which they are inserted.
3. **LinkedHashMap** defines the following constructor

```
LinkedHashMap()

LinkedHashMap(Map< ? extends k, ? extends V> m)

LinkedHashMap(int capacity)

LinkedHashMap(int capacity, float fillratio)

LinkedHashMap(int capacity, float fillratio, boolean order)
```

4. It adds one new method `removeEldestEntry()`. This method is called by `put()` and `putAll()` By default this method does nothing. However we can override this method to remove oldest element in the map. **Syntax**

```
protected boolean removeEldestEntry(Map.Entry e)
```

EnumMap class

1. **EnumMap** extends **AbstractMap** and implements **Map** interface.
2. It is used for key as enum

Comparator Interface

In Java, Comparator interface is used to order the object in your own way. It gives you ability to decide how element are stored within sorted collection and map.

Comparator Interface defines `compare()` method. This method compare two object and return 0 if two object are equal. It returns a positive value if object1 is greater than object2. Otherwise a negative value is return. The method can throw a **ClassCastException** if the type of object are not compatible for comparison.

Example

Student class

```
class Student

int roll;

  String name;

  Student(int r,String n)

  {

      roll = r;
```

```
      name = n;
  }
  public String toString()
  {
      return roll+" "+name;
  }
```

MyComparator class

This class defines the comparison logic for Student class based on their roll. Student object will be sotred in ascending order of their roll.

```
class MyComparator implements Comparator
{
  public int compare(Student s1,Student s2)
    {
        if(s1.roll == s2.roll) return 0;
        else if(s1.roll > s2.roll) return 1;
        else return -1;
    }
}
public class Test
{

   public static void main(String[] args)
   {
       TreeSet< Student> ts = new TreeSet< Student>(new MyComparator());
       ts.add(new Student(45, "Rahul"));
       ts.add(new Student(11, "Adam"));
       ts.add(new Student(19, "Alex"));
       System.out.println(ts);
   }
```

```
}
```

Output

```
[ 11 Adam, 19 Alex, 45 Rahul ]
```

As you can see in the ouput Student object are stored in ascending order of their **roll**.

Legacy Classes

Early version of java did not include the **Collection** framework. It only defined several classes and interface that provide method for storing objects. When **Collection** framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes. All legacy claases and interface were redesign by JDK 5 to support Generics.

The following are the legacy classes defined by **java.util** package

1. Dictionary
2. HashTable
3. Properties
4. Stack
5. Vector

There is only one legacy interface called **Enumeration**

NOTE: All the legacy classes are syncronized

Enumeration interface

1. **Enumeration** interface defines method to enumerate through collection of object.
2. This interface is suspended by **Iterator** interface.
3. However some legacy classes such as **Vector** and **Properties** defines several method in which**Enumeration** interface is used.
4. It specifies the following two methods

```
boolean hasMoreElements()


Object nextElement()
```

Vector class

1. **Vector** is similar to **ArrayList** which represents a dynamic array.
2. The only difference between **Vector** and **ArrayList** is that Vector is synchronised while Array is not.
3. Vector class has following four constructor

```
Vector()


Vector(int size)


Vector(int size, int incr)


Vector(Collection< ? extends E> c)
```

Vector defines several legacy method. Lets see some important legacy method define by **Vector** class.

Method	**Description**
addElement()	add element to the Vector
elementAt()	return the element at specified index
elements	return an enumeration of element in vector

firstElement()	return first element in the Vector
lastElement()	return last element in the Vector
removeAllElement()	remove all element of the Vector

Example of Vector

```
import java.util.*;

public class Test

{

  public static void main(String[] args)

   {

      Vector ve = new Vector();

       ve.add(10);

       ve.add(20);

       ve.add(30);

       ve.add(40);

       ve.add(50);

       ve.add(60);


       Enumeration en = ve.elements();


       while(en.hasMoreElements())

       {

           System.out.println(en.nextElement());

       }

   }
```

```
}
```

Output

```
10
20
30
40
50
60
```

Hashtable class

1. Like HashMap, Hashtable also stores key/value pair in hashtable. However neither **keys** nor **values** can be **null**.
2. There is one more difference between **HashMap** and **Hashtable** that is Hashtable is synchronized while HashMap is not.
3. Hashtable has following four constructor

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillratio)
Hashtable(Map< ? extends K, ? extends V> m)
```

Example of Hashtable

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

```
 {
  Hashtable< String,Integer> ht = new Hashtable< String,Integer>();
  ht.put("a",new Integer(100));
  ht.put("b",new Integer(200));
  ht.put("c",new Integer(300));
  ht.put("d",new Integer(400));

  Set st = ht.entrySet();
  Iterator itr=st.iterator();
  while(itr.hasNext())
  {
   Map.Entry m=(Map.Entry)itr.next();
   System.out.println(itr.getKey()+" "+itr.getValue());
  }
 }
}
```

Output:

```
a 100
b 200
c 300
d 400
```

Difference between HashMap and Hashtable

Hashtable	HashMap
Hashtable class is synchronized.	HastMap is not synchronize.

Because of Thread-safe, Hashtable is slower than HashMap	HashMap works faster.
Neither **key** nor **values** can be null	Both **key** and **values** can be null
Order of table remain constant over time.	does not guarantee that order of map remain constant over time.

Properties class

1. **Properties** class extends **Hashtable** class.
2. It is used to maintain list of value in which both key and value are **String**
3. **Properties** class define two constructor

```
Properties()

Properties(Properties default)
```

4. One advantage of **Properties** over **Hashtable** is that we can specify a default property that will be useful when no value is associated with a certain key.

Example of Properties class

```
import java.util.*;


public class Test
{


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

```
     Properties pr = new Properties();
     pr.put("Java", "James Ghosling");
     pr.put("C++", "Bjarne Stroustrup");
     pr.put("C", "Dennis Ritchie");
     pr.put("C#", "Microsoft Inc.");
     Set< ?> creator = pr.keySet();

     for(Object ob: creator)
     {
         System.out.println(ob+" was created by "+ pr.getProperty((String)ob) );
     }

   }

}
```

Output

```
Java was created by James Ghosling
C++ was created by Bjarne Stroustrup
C was created by Dennis Ritchie
C# was created by Microsoft Inc
```