

# Object-oriented Programming in Java

## Session: 3

Collections API



# Objectives



- ◆ Explain java.util package
- ◆ Explain List classes and interfaces
- ◆ Explain Set classes and interfaces
- ◆ Explain Map classes and interfaces
- ◆ Explain Queues and Arrays

For Aptech Centre Use Only



- ◆ The collection framework consist of collection interfaces which are primary means by which collections are manipulated.
- ◆ They also have wrapper and general purpose implementations. Adapter implementation helps to adapt one collection over other.
- ◆ Besides these, there are convenience implementations and legacy implementations.

For Aptech Centre Use Only



- ◆ The `java.util` package contains the definition of a number of useful classes providing a broad range of functionality.
  - ◆ The package mainly contains collection classes that are useful for working with groups of objects.
  - ◆ The package also contains the definition of classes that provides date and time facilities and many other utilities, such as calendar and dictionary.
  - ◆ It also contains a list of classes and interfaces to manage a collection of data in memory.
- ◆ The following figure displays some of the classes present in `java.util` package:





- ◆ A collection is a container that helps to group multiple elements into a single unit.
- ◆ Collections help to store, retrieve, manipulate, and communicate data.
- ◆ The Collections Framework represents and manipulates collections.
- ◆ It includes the following:
  - ◆ Algorithms
  - ◆ Implementations
  - ◆ Interfaces

For Aptech Centre Use Only



- ◆ Collections Framework consists of interfaces and classes for working with group of objects.
- ◆ At the top of the hierarchy, `Collection` interface lies.
- ◆ The `Collection` interface helps to convert the collection's type.
- ◆ The `Collection` interface is extended by the following sub interfaces:
  - ◆ `Set`
  - ◆ `List`
  - ◆ `Queue`
- ◆ Some of the `Collection` classes are as follows:
  - ◆ `HashSet`
  - ◆ `LinkedHashSet`
  - ◆ `TreeSet`

# Methods of Collection Interface



- ◆ `size`, `isEmpty`: Use these to inform about the number of elements that exist in the collection.
- ◆ `contains`: Use this to check if a given object is in the collection.
- ◆ `add`, `remove`: Use these to add and remove an element from the collection.
- ◆ `iterator`: Use this to provide an iterator over the collection.

For Apteck Centre Use Only



## Using the for-each construct:

- ◆ This helps to traverse a collection or array using a `for` loop.
- ◆ The following Code Snippet illustrates the use of the `for-each` construct to print out each element of a collection on a separate line:

### Code Snippet

```
for (Object obj : collection)
    System.out.println(obj);
```

## Using Iterator:

- ◆ These help to traverse through a collection.
- ◆ They also help to remove elements from the collection selectively.





- ◆ The `iterator()` method is invoked to obtain an `Iterator` for a collection.
- ◆ The `Iterator` interface includes the following methods:

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); //optional  
}
```

For Aptech Centre Use Only



- ◆ Bulk operations perform shorthand operations on an entire Collection using the basic operations.
- ◆ The following table describes the methods for bulk operations:

Method	Description
<code>containsAll</code>	This method will return true if the target Collection contains all elements that exist in the specified Collection.
<code>addAll</code>	This method will add all the elements of the specified Collection to the target Collection.
<code>removeAll</code>	This method will remove all the elements from the target Collection that exist in the specified Collection.
<code>retainAll</code>	This method will remove those elements from the target Collection that do not exist in the specified Collection.



- ◆ The `List` interface is an extension of the `Collection` interface.
- ◆ It defines an ordered collection of data and allows duplicate objects to be added to a list.
- ◆ Its advantage is that it adds position-oriented operations, enabling programmers to work with a part of the list.
- ◆ The `List` interface uses an index for ordering the elements while storing them in a list.
- ◆ `List` has methods that allow access to elements based on their position, search for a specific element, and return their position, in addition to performing arbitrary range operations.
- ◆ It also provides the `List` iterator to take advantage of its sequential nature.

# Methods of `List` Interface



- ◆ `add(int index, E element)`
- ◆ `addAll(int index, Collection<? extends E> c)`
- ◆ `get(int index)`
- ◆ `set(int index, E element)`
- ◆ `remove(int index)`
- ◆ `subList(int start, int end)`
- ◆ `indexOf(Object o)`
- ◆ `lastIndexOf(Object o)`

For Apteck Centre Use Only

# ArrayList Class [1-2]



- ◆ `ArrayList` class is an implementation of the `List` interface in the `Collections Framework`.
- ◆ The `ArrayList` class creates a variable-length array of object references.
- ◆ The `ArrayList` class includes all elements, including null.
- ◆ In addition to implementing the methods of the `List` interface, this class provides methods to change the size of the array that is used internally to store the list.
- ◆ Each `ArrayList` instance includes a capacity that represents the size of the array.
- ◆ A capacity stores the elements in the list and grows automatically as elements are added to an `ArrayList`.
- ◆ `ArrayList` class is best suited for random access without inserting or removing elements from any place other than the end.



- ◆ An instance of `ArrayList` can be created using any one of the following constructors:
  - ◆ `ArrayList()`
  - ◆ `ArrayList(Collection <? extends E> c)`
  - ◆ `ArrayList(int initialCapacity)`
- ◆ The following Code Snippet displays the creation of an instance of the `ArrayList` class:

## Code Snippet

```
. . .  
List<String> listObj = new ArrayList<String> ();  
System.out.println("The size is : " + listObj.size());  
for (int ctr=1; ctr <= 10; ctr++)  
{  
    listObj.add("Value is : " + new Integer(ctr));  
}  
. . .
```

# Methods of ArrayList Class



- ◆ `add(E obj)`
- ◆ `trimToSize()`
- ◆ `ensureCapacity(int minCap)`
- ◆ `clear()`
- ◆ `contains(Object obj)`
- ◆ `size()`

The following Code Snippet displays the use of ArrayList class:

## Code Snippet

```
. . .
List<String> listObj = new ArrayList<String> ();
System.out.println("The size is : " + listObj.size());
for (int ctr=1; ctr <= 10; ctr++)
{
    listObj.add("Value is : " + new Integer(ctr));
}
listObj.set(5, "Hello World");
System.out.println("Value is: " + (String)listObj.get(5));
. . .
```



- ◆ The `Vector` class is similar to an `ArrayList` as it also implements dynamic array.
- ◆ `Vector` class stores an array of objects and the size of the array can increase or decrease.
- ◆ The elements in the `Vector` can be accessed using an integer index.
- ◆ Each vector maintains a capacity and a `capacityIncrement` to optimize storage management.
- ◆ The vector's storage increases in chunks specified by the `capacityIncrement` as components are added to it.
- ◆ The constructors of this class are as follows:
  - ◆ `Vector()`
  - ◆ `Vector(Collection<? extends E> c)`
  - ◆ `Vector(int initCapacity)`
  - ◆ `Vector(int initCapacity, int capIncrement)`





The following Code Snippet displays the creation of an instance of the `Vector` class:

## Code Snippet

```
. . .  
Vector vecObj = new Vector();  
. . .
```

## Methods of Vector Class:

```
addElement(E obj)  
capacity()  
toArray()  
elementAt(int pos)  
removeElement(Object obj)  
clear()
```



The following Code Snippet displays the use of the Vector class:

## Code Snippet

```
. . .  
Vector<Object> vecObj = new Vector<Object>();  
vecObj.addElement(new Integer(5));  
vecObj.addElement(new Integer(7));  
vecObj.addElement(new Integer(45));  
vecObj.addElement(new Float(9.95));  
vecObj.addElement(new Float(6.085));  
System.out.println("The value is: "  
+ (Object) vecObj.elementAt(3));  
. . .
```

# LinkedList Class [1-2]



- ◆ `LinkedList` class implements the `List` interface.
- ◆ An array stores objects in consecutive memory locations, whereas a linked list stores object as a separate link.
- ◆ It provides a linked list data structure.
- ◆ A linked list is a list of objects having a link to the next object.
- ◆ There is usually a data element followed by an address element that contains the address of the next element in the list in a sequence.
- ◆ Each such item is referred as a node.
- ◆ Linked lists allow insertion and removal of nodes at any position in the list, but do not allow random access.
- ◆ There are several different types of linked lists - singly-linked lists, doubly-linked lists, and circularly-linked lists.

## LinkedList Class [2-2]



Java provides the `LinkedList` class in the `java.util` package to implement linked lists.

### **LinkedList () :**

The `LinkedList ()` constructor creates an empty linked list.

### **LinkedList(Collection <? extends E>c):**

The `LinkedList(Collection <? extends E>c)` constructor creates a linked list, which contains the elements of a specified collection, in the order they are returned by the collection's iterator.

The following Code Snippet displays the creation of an instance of the `LinkedList` class:

#### Code Snippet

```
. . .  
LinkedList<String> lisObj = new LinkedList<List>();  
. . .
```

# Methods of LinkedList Class



- ◆ `addFirst(E obj)`
- ◆ `addLast(E obj)`
- ◆ `getFirst()`
- ◆ `getLast()`
- ◆ `removeFirst()`
- ◆ `removeLast()`

The following Code Snippet displays the use of the methods of the `LinkedList` class:

## Code Snippet

```
. . .
LinkedList<String> lisObj = new LinkedList<String>();
lisObj.add("John");
lisObj.add("Mary");
lisObj.add("Jack");
lisObj.add("Elvis");
lisObj.add("Martin");
System.out.println("Original content of the list: " + lisObj);
lisObj.removeFirst();
System.out.println("After removing content of the list: " + lisObj);
. . .
```



- ◆ The autoboxing and unboxing feature automates the process of using primitive value into a collection.
- ◆ Note that collections hold only object references.
- ◆ So, primitive values, such as `int` from `Integer`, have to be boxed into the appropriate wrapper class.
- ◆ If an `int` value is required, the integer value must be unboxed using the `intValue()` method.
- ◆ The autoboxing and unboxing feature helps to reduce the clutter in the code.

For Aptechnical Use Only



- ◆ The `Set` interface creates a list of unordered objects.
- ◆ It creates non-duplicate list of object references.
- ◆ The `Set` interface inherits all the methods from the `Collection` interface, except those methods that allow duplicate elements.
- ◆ The Java platform contains three general-purpose `Set` implementations. They are as follows:
  - ◆ `HashSet`
  - ◆ `TreeSet`
  - ◆ `Link`
- ◆ The `Set` interface is an extension of the `Collection` interface and defines a set of elements.
- ◆ The difference between `List` and `Set` is that, the `Set` does not permit duplication of elements.
- ◆ `Set` is used to create non-duplicate list of object references.
- ◆ Therefore, `add()` method returns false if duplicate elements are added.

# Methods of Set Interface



- ◆ `containsAll(Collection<?> obj)`
- ◆ `addAll(Collection<? extends E> obj)`
- ◆ `retainAll(Collection<?> obj)`
- ◆ `removeAll(Collection<?> obj)`

For Apteck Centre Use Only



# SortedSet Interface [1-2]



- ◆ The `SortedSet` interface extends the `Set` interface and its iterator traverses its elements in the ascending order.
- ◆ Elements can be ordered by natural ordering, or by using a `Comparator` that a user can provide while creating a sorted set.
- ◆ `SortedSet` is used to create sorted lists of non-duplicate object references.
- ◆ The ordering of a sorted set should be consistent with `equals()` method.
- ◆ A sorted set performs all element comparisons using the `compareTo()` or `compare()` method.

For Aptech Centre Use Only



- ◆ Typically, sorted set implementation classes provide the following standard constructors:
  - ◆ No argument (void) constructor
  - ◆ Single argument of type Comparator constructor
  - ◆ Single argument of type Collection constructor
  - ◆ Single argument of type SortedSet constructor
- ◆ Some of the methods in this interface are as follows:
  - ◆ `first()`
  - ◆ `last()`
  - ◆ `headSet (E endElement)`
  - ◆ `subSet (E startElement, E endElement)`
  - ◆ `tailSet (E fromElement)`



- ◆ HashSet class implements the Set interface and creates a collection that makes use of a hashtable for data storage.
- ◆ This HashSet class allows null element.
- ◆ The HashSet class provides constant time performance for the basic operations.
- ◆ The constructors of the HashSet class are as follows:
  - ◆ `HashSet()`
  - ◆ `HashSet(Collection<? extends E> c)`
  - ◆ `HashSet(int size)`
  - ◆ `HashSet(int size, float fillRatio)`
- ◆ The following Code Snippet displays the creation of an instance of HashSet class:

## Code Snippet

```
. . .  
Set<String> words = new HashSet<String>();  
. . .
```



- ◆ The `LinkedHashSet` class creates a list of elements and maintains the order of the elements added to the `Set`.
- ◆ This class includes the following features:
  - ◆ It provides all of the optional `Set` operations.
  - ◆ It permits null elements.
  - ◆ It provides constant-time performance for the basic operations such as `add` and `remove`.
- ◆ The constructors of this class are as follows:
  - ◆ `LinkedHashSet()`
  - ◆ `LinkedHashSet(Collection<? extends E> c)`
  - ◆ `LinkedHashSet(int initial capacity)`

For Aptech Centre Use Only



- ◆ `TreeSet` class implements the `NavigableSet` interface and uses a tree structure for data storage.
- ◆ The elements can be ordered by natural ordering or by using a `Comparator` provided at the time of `Set` creation.
- ◆ Objects are stored in ascending order and therefore accessing and retrieving an object is much faster.
- ◆ `TreeSet` is used when elements need to be extracted quickly from the collection in a sorted manner.
- ◆ This class includes the following constructors:
  - ◆ `TreeSet()`
  - ◆ `TreeSet(Collection<? extends E> c)`
  - ◆ `TreeSet(Comparator<? super E> c)`
  - ◆ `TreeSet(SortedSet<E> s)`
- ◆ The following Code Snippet creates an instance of `TreeSet`:

## Code Snippet

```
. . .  
TreeSet tsObj = new TreeSet();  
. . .
```



- ◆ A `Map` object stores data in the form of relationships between keys and values.
- ◆ Each key will map to at least a single value.
- ◆ If key information is known, its value can be retrieved from the `Map` object.
- ◆ Keys should be unique but values can be duplicated.
- ◆ The `Map` interface does not extend the `Collection` interface.
- ◆ The interface describes a mapping from keys to values, without duplicate keys.
- ◆ The `Collections` API provides three general-purpose `Map` implementations:
  - ◆ `HashMap`
  - ◆ `TreeMap`
  - ◆ `LinkedHashMap`
- ◆ The important methods of a `Map` interface are as follows:
  - ◆ `put(K key, V value)`
  - ◆ `get(Object key)`
  - ◆ `containsKey(Object key)`
  - ◆ `containsValue(Object value)`
  - ◆ `size()`
  - ◆ `values()`



- ◆ The `HashMap` class implements the `Map` interface and inherits all its methods.
- ◆ An instance of `HashMap` has two parameters: initial capacity and load factor.
- ◆ Initial capacity determines the number of objects that can be added to the `HashMap` at the time of the `Hashtable` creation.
- ◆ The load factor determines how full the `Hashtable` can get, before its capacity is automatically increased.
- ◆ The constructors of this class are as follows:
  - ◆ `HashMap()`
  - ◆ `HashMap(int initialCapacity)`
  - ◆ `HashMap(int initialCapacity, float loadFactor)`
  - ◆ `HashMap(Map<? extends K,? extends V> m)`



- ◆ The following Code Snippet displays the use of the HashMap class:

## Code Snippet

```
. . .  
class EmployeeData  
{  
    public EmployeeData(String nm)  
    {  
        name = nm;  
        salary = 5600;  
    }  
    public String toString()  
    {  
        return "[name=" + name + ", salary=" + salary +  
        "]" ;  
    }  
}
```



# HashMap Class [3-4]



```
public String toString()
{
    return "[name=" + name + ", salary=" + salary +
"]";
}

. . .
}

public class MapTest
{
    public static void main(String[] args)
    {
        Map<String, EmployeeData> staffObj = new
HashMap<String,
        EmployeeData>();
        staffObj.put("101", new EmployeeData("Anna
John"));
    }
}
```

# HashMap Class [4-4]



```
staffObj.put("102", new EmployeeData("Harry Hacker"));
staffObj.put("103", new EmployeeData("Joby Martin"));
System.out.println(staffObj);
staffObj.remove("103");
staffObj.put("106", new EmployeeData("Joby Martin"));
System.out.println(staffObj.get("106"));
System.out.println(staffObj);
. . .
}
}
```

For Aptech Centre Use Only



- ◆ The `Hashtable` class implements the `Map` interface but stores elements as a key/value pairs in the hashtable.
- ◆ While using a `Hashtable`, a key is specified to which a value is linked.
- ◆ The key is hashed and then the hash code is used as an index at which the value is stored.
- ◆ The class inherits all the methods of the `Map` interface.
- ◆ To retrieve and store objects from a hashtable successfully, the objects used as keys must implement the `hashCode()` and `equals()` method.
- ◆ The constructors of this class are as follows:
  - ◆ `Hashtable()`
  - ◆ `Hashtable(int initCap)`
  - ◆ `Hashtable(int initCap, float fillRatio)`
  - ◆ `Hashtable(Map<? extends K,? extends V> m)`

# Hashtable Class [2-2]



The following Code Snippet displays the use of the Hashtable class:

## Code Snippet

```
. . .
Hashtable<String, String> bookHash = new Hashtable<String, String>();
bookHash.put("115-355N", "A Guide to Advanced Java");
bookHash.put("116-455A", "Learn Java by Example");
bookHash.put("116-466B", "Introduction to Solaris");
String str = (String) bookHash.get("116-455A");
System.out.println("Detail of a book " + str);
System.out.println("Is table empty " + bookHash.isEmpty());
System.out.println("Does table contains key? " +
bookHash.containsKey("116- 466B"));
Enumeration name = bookHash.keys();
while (name.hasMoreElements())
{
    String bkCode = (String)name.nextElement();
    System.out.println( bkCode +": " + (String)bookHash.get(bkCode));
}
. . .
```

# TreeMap Class [1-2]



- ◆ The `TreeMap` class implements the `NavigableMap` interface but stores elements in a tree structure.
- ◆ The `TreeMap` returns keys in sorted order.
- ◆ If there is no need to retrieve `Map` elements sorted by key, then the `HashMap` would be a more practical structure to use.
- ◆ The constructors of this class are as follows:
  - ◆ `TreeMap()`
  - ◆ `TreeMap(Comparator<? super K> c)`
  - ◆ `TreeMap(Map<? extends K, ? extends V> m)`
  - ◆ `TreeMap(SortedMap<K, ? extends V> m)`
- ◆ The important methods of the `TreeMap` class are as follows:
  - ◆ `firstKey()`
  - ◆ `lastKey()`
  - ◆ `headMap(K toKey)`
  - ◆ `tailMap(K fromKey)`

# TreeMap Class [2-2]



The following Code Snippet displays the use of the TreeMap class:

## Code Snippet

```
. . .  
TreeMap<String, EmployeeData> staffObj = new TreeMap<String,  
EmployeeData>();  
staffObj.put("101", new EmployeeData("Anna John"));  
staffObj.put("102", new EmployeeData("Harry Hacker"));  
staffObj.put("103", new EmployeeData("Joby Martin"));  
System.out.println(staffObj);  
staffObj.remove("103");  
staffObj.put("104", new EmployeeData("John Luther"));  
System.out.println(staffObj.get("104"));  
Object firstKey = staffObj.firstKey();  
System.out.println(firstKey.toString());  
System.out.println((String) staffObj.firstKey());  
System.out.println((String) (staffObj.lastKey()));  
. . .
```



- ◆ LinkedHashMap class implements the concept of hashtable and the linked list in the Map interface.
- ◆ A LinkedHashMap maintains the values in the order they were inserted, so that the key/values will be returned in the same order that they were added to this Map.
- ◆ The constructors of this class are as follows:
  - ◆ `LinkedHashMap()`
  - ◆ `LinkedHashMap(int initialCapacity)`
  - ◆ `LinkedHashMap(int initialCapacity, float loadFactor)`
  - ◆ `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`
  - ◆ `LinkedHashMap(Map<? extends K,? extends V> m)`
- ◆ The important methods in LinkedHashMap class are as follows:
  - ◆ `clear()`
  - ◆ `containsValue(Object value)`
  - ◆ `get(Object key)`
  - ◆ `removeEldestEntry(Map.Entry<K,V> eldest)`



- ◆ In the `Stack` class, the stack of objects results in a Last-In-First-Out (LIFO) behavior.
- ◆ It extends the `Vector` class to consider a vector as a stack.
- ◆ `Stack` only defines the default constructor that creates an empty stack.
- ◆ It includes all the methods of the vector class.
- ◆ This interface includes the following five methods:
  - ◆ `empty()`
  - ◆ `peek()`
  - ◆ `pop()`
  - ◆ `push(E item)`
  - ◆ `int search(Object o)`





- ◆ A `Queue` is a collection for holding elements that needs to be processed.
- ◆ In `Queue`, the elements are normally ordered in First-In-First-Out (FIFO) manner.
- ◆ A queue can be arranged in other orders too.
- ◆ Every `Queue` implementation defines ordering properties.
- ◆ In a FIFO queue, new elements are inserted at the end of the queue.
- ◆ LIFO queues or stacks order the elements in LIFO pattern.
- ◆ However, in any form of ordering, a call to the `poll()` method removes the head of the queue.



- ◆ A double ended queue is commonly called deque.
- ◆ It is a linear collection that supports insertion and removal of elements from both ends.
- ◆ Usually, `Deque` implementations have no restrictions on the number of elements to include.
- ◆ A deque when used as a queue results in FIFO behavior.
- ◆ The `Deque` interface and its implementations when used with the `Stack` class provides a consistent set of LIFO stack operations.
- ◆ The following Code Snippet displays `Deque`:

## Code Snippet

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

- ◆ Some of the important methods supported by this class are as follows:
  - ◆ `poll()`
  - ◆ `peek()`
  - ◆ `remove()`
  - ◆ `offer(E obj)`
  - ◆ `element()`

# PriorityQueue Class [1-3]



- ◆ Priority queues are similar to queues but the elements are not arranged in FIFO structure.
- ◆ They are arranged in a user-defined manner.
- ◆ The elements are ordered either by natural ordering or according to a comparator.
- ◆ A priority queue neither allows adding of non-comparable objects nor allows null elements.
- ◆ A priority queue is unbound and allows the queue to grow in capacity.
- ◆ When the elements are added to a priority queue, its capacity grows automatically.



- ◆ The constructors of this class are as follows:
  - ◆ `PriorityQueue()`
  - ◆ `PriorityQueue(Collection<? extends E> c)`
  - ◆ `PriorityQueue(int initialCapacity)`
  - ◆ `PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`
  - ◆ `PriorityQueue(PriorityQueue<? extends E> c)`
  - ◆ `PriorityQueue(SortedSet<? extends E> c)`
- ◆ The `PriorityQueue` class inherits the method of the `Queue` class.
- ◆ The other methods supported by the `PriorityQueue` class are as follows:
  - ◆ `add(E e)`
  - ◆ `clear()`
  - ◆ `comparator()`
  - ◆ `contains(Object o)`
  - ◆ `iterator()`
  - ◆ `toArray()`

# PriorityQueue Class [3-3]



The following Code Snippet displays the use of the `PriorityQueue` class:

## Code Snippet

```
. . .
PriorityQueue<String> queue = new PriorityQueue<String>();
queue.offer("New York");
queue.offer("Kansas");
queue.offer("California");
queue.offer("Alabama");
System.out.println("1. " + queue.poll()); // removes
System.out.println("2. " + queue.poll()); // removes
System.out.println("3. " + queue.peek());
System.out.println("4. " + queue.peek());
System.out.println("5. " + queue.remove()); // removes
System.out.println("6. " + queue.remove()); // removes
System.out.println("7. " + queue.peek());
System.out.println("8. " + queue.element()); // Throws Exception
. . .
```



- ◆ `Arrays` class provides a number of methods for working with arrays such as searching, sorting, and comparing arrays.
- ◆ The class has a static factory method that allows the array to be viewed as lists.
- ◆ The methods of this class throw an exception, `NullPointerException` if the array reference is null.
- ◆ Some of the important methods of this class are as follows:
  - ◆ `equals(<type> arrObj1, <type> arrObj2)`
  - ◆ `fill(<type>[] array, <type> value)`
  - ◆ `fill(type[] array, int fromIndex, int toIndex, type value)`
  - ◆ `sort(<type>[] array)`
  - ◆ `sort(<type> [] array, int startIndex, int endIndex)`
  - ◆ `toString(<type>[] array)`



Collection API provides the following two interfaces for ordering interfaces:

- ◆ **Comparable**: The `Comparable` interface imposes a total ordering on the objects of each class which implements it. Lists of objects implementing this interface are automatically sorted. It is sorted using **`Collection.sort`** or **`Arrays.sort`** method.
- ◆ **Comparator**: This interface provides multiple sorting options and imposes a total ordering on some collection of objects.

For Apteck Centre Use Only

# Enhancements in Collection Classes [1-11]



- ◆ The `ArrayDeque` class implements the `Deque` interface.
- ◆ This class is faster than stack and linked list when used as a queue.
- ◆ It does not put any restriction on capacity and does not allow null values.
- ◆ The following Code Snippet shows the use of some of the methods available in the `ArrayDeque` class:

## Code Snippet

```
import java.util.ArrayDeque;
import java.util.Iterator;
...
...
public static void main(String args[]) {
    ArrayDeque arrDeque = new ArrayDeque();
    arrDeque.addLast("Mango");
}
```



# Enhancements in Collection Classes [2-11]



```
arrDeque.addLast("Apple");
arrDeque.addFirst("Banana");
for (Iterator iter = arrDeque.iterator(); iter.hasNext();) {
    System.out.println(iter.next());
}
for (Iterator descendingIter = arrDeque.descendingIterator();
descendingIter.hasNext();) {
    System.out.println(descendingIter.next());
}
System.out.println("First Element : " + arrDeque.getFirst());
System.out.println("Last Element : " + arrDeque.getLast());
System.out.println("Contains \"Apple\" : " + arrDeque.
contains("Apple"));
}
...
```

- ◆ The `ConcurrentSkipListSet` class implements the `NavigableSet` interface.
- ◆ The elements are sorted based on natural ordering or by a `Comparator`. The `Comparator` is an interface that uses the `compare()` method to sort objects that don't have a natural ordering.

# Enhancements in Collection Classes [3-11]



- ◆ The following Code Snippet shows the use of some of the methods available in `ConcurrentSkipListSet` class:

## Code Snippet

```
import java.util.Iterator;
import java.util.concurrent.ConcurrentSkipListSet;
...
public static void main(String args[]) {
    ConcurrentSkipListSet fruitSet = new ConcurrentSkipListSet();
    fruitSet.add("Banana");
    fruitSet.add("Peach");
    fruitSet.add("Apple");
    fruitSet.add("Mango");
}
```

# Enhancements in Collection Classes [4-11]



```
fruitSet.add("Orange");  
// Displays in ascending order  
Iterator iterator = fruitSet.iterator();  
System.out.print("In ascending order :");  
while (iterator.hasNext())  
System.out.print(iterator.next() + " ");  
// Displays in descending order  
System.out.println("In descending order: " +  
fruitSet.descendingSet() + "\n");  
System.out.println("Lower element: " + fruitSet.lower("Mango"));  
System.out.println("Higher element: " + fruitSet.higher("Apple"));  
}  
...
```

- ◆ The `ConcurrentSkipListMap` class implements `ConcurrentNavigableMap` interface.
- ◆ It belongs to `java.util.concurrent` package.

# Enhancements in Collection Classes [5-11]



The following Code Snippet shows the use of some of the methods available in `ConcurrentSkipListMap` class:

## Code Snippet

```
import java.util.concurrent.ConcurrentSkipListMap;
...
...
public static void main(String args[]) {
    ConcurrentSkipListMap fruits = new ConcurrentSkipListMap();
    fruits.put(1, "Apple");
    fruits.put(2, "Banana");
    fruits.put(3, "Mango");
    fruits.put(4, "Orange");
    fruits.put(5, "Peach");
    // Retrieves first data
    System.out.println("First data: " + fruits.firstEntry() + "\n");
    // Retrieves last data
    System.out.println("Last data: " + fruits.lastEntry() + "\n");
    // Displays all data in descending order
```

# Enhancements in Collection Classes [6-11]



```
System.out.println("Data in reverse order: " +  
fruits.descendingMap());  
}  
...
```

- ◆ The `LinkedBlockingDeque` class implements the `BlockingDeque` interface.
- ◆ The class belongs to `java.util.concurrent` package.
- ◆ The class contains linked nodes that are dynamically created after each insertion.
- ◆ The following Code Snippet shows the implementation of `LinkedBlockingDeque` class and use of some of its available methods:

## Code Snippet

```
/* ProducerDeque.Java */  
import java.util.concurrent.BlockingDeque;  
class ProducerDeque implements Runnable {  
    private String name;  
    private BlockingDeque blockDeque;
```

# Enhancements in Collection Classes [7-11]



```
public ProducerDeque(String name, BlockingDeque blockDeque) {
    this.name = name;
    this.blockDeque = blockDeque;
}

public void run() {
    for (int i = 1; i < 10; i++) {
        try {
            blockDeque.addFirst(i);
            System.out.println(name + " puts " + i);
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (IllegalStateException ex) {
            System.out.println("Deque filled upto the maximum capacity");
            System.exit(0);
        }
    }
}
```

# Enhancements in Collection Classes [8-11]



```
    }  
}  
/* ConsumerDeque.Java */  
import java.util.concurrent.BlockingDeque;  
import java.util.concurrent.LinkedBlockingDeque;  
class ConsumerDeque implements Runnable {  
    private String name;  
    private BlockingDeque blockDeque;  
    public ConsumerDeque(String name, BlockingDeque blockDeque) {  
        this.name = name;  
        this.blockDeque = blockDeque;  
        public void run() {  
            for (int i = 1; i < 10; i++) {  
                try {  
                    int j = (Integer) blockDeque.peekFirst();  
                    System.out.println(name + " takes " + j);  
                    Thread.sleep(100);  
                } catch (InterruptedException e) {
```

# Enhancements in Collection Classes [9-11]



```
e.printStackTrace();  
}  
}  
}  
}  
/* LinkedBlockingDequeClass.Java */  
import java.util.concurrent.BlockingDeque;  
import java.util.concurrent.LinkedBlockingDeque;  
public class LinkedBlockingDequeClass {  
    public static void main(String[] args) {  
        BlockingDeque blockDeque = new LinkedBlockingDeque(5);  
        Runnable produce = new ProducerDeque("Producer", blockDeque);  
        Runnable consume = new ConsumerDeque("Consumer", blockDeque);  
        new Thread(produce).start();  
        new Thread(consume).start();  
    }  
}
```



# Enhancements in Collection Classes [10-11]



- ◆ The `AbstractMap.SimpleEntry` is static class nested inside `AbstractMap` class.
- ◆ This class is used to implement custom map.
- ◆ An instance of this class stores key-value pair of a single entry in a map.
- ◆ The value of the entry can be changed.
- ◆ The `getKey()` method returns the key of an entry in the instance.
- ◆ The following Code Snippet shows the implementation of `AbstractMap.SimpleEntry` static class and the use of some of its available methods:

## Code Snippet

```
AbstractMap.SimpleEntry<String,String> se = new  
AbstractMap.SimpleEntry<String,String>("1","Apple");  
System.out.println(se.getKey());  
System.out.println(se.getValue());  
se.setValue("Orange");  
System.out.println(se.getValue());
```

# Enhancements in Collection Classes [11-11]



- ◆ The `AbstractMap.SimpleImmutableEntry` class is a static class nested inside the `AbstractMap` class.
- ◆ As the name suggests, it does not allow modifying a value in an entry.
- ◆ If any attempt to change a value is made, it results in throwing `UnsupportedOperationException`.

For Aptech Centre Use Only



- ◆ The java.util package contains the definition of number of useful classes providing a broad range of functionality.
- ◆ The List interface is an extension of the Collection interface.
- ◆ The Set interface creates a list of unordered objects.
- ◆ A Map object stores data in the form of relationships between keys and values.
- ◆ A Queue is a collection for holding elements before processing.
- ◆ ArrayDeque class does not put any restriction on capacity and does not allow null values.
- ◆ AbstractMap.SimpleEntry is used for implementation of custom map.
- ◆ AbstractMap.SimpleImmutableEntry class is a static class and does not allow modification of values in an entry.