JAVA COLLECTIONS

---Dilip Dedhia, Ph.D.

Collections Framework

- **Interoperability between unrelated APIs**
- **Reduces the effort required to learn APIs**
- **Reduces the effort required to design and implement APIs**
- **Fosters software reuse**

A collections framework is a unified architecture for representing and manipulating collections, allowing them to be manipulated independently of the details of their representation.

•Reduces programming effort by providing useful data structures and algorithms so you don't have to write them yourself.

•Increases performance by providing high-performance implementations of useful data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be easily tuned by switching implementations.

•Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.

•Reduces the effort required to learn APIs by eliminating the need to learn multiple ad hoc collection APIs.

•Reduces the effort required to design and implement APIs by eliminating the need to produce ad hoc collections APIs.

•Fosters software reuse by providing a standard interface for collections and algorithms to manipulate them.

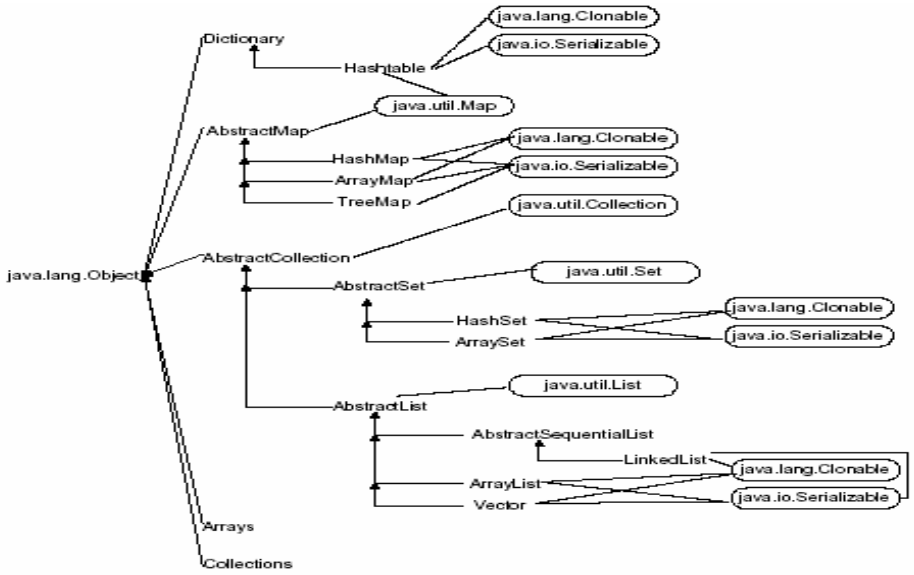
Collection Classes

- **The Java 2 platform contains a Collections API**
- **This group of classes represent various data structures used to store and manage objects**
- **Their underlying implementation is implied in the class names, such as** `ArrayList` **and** `LinkedList`
- **Several interfaces are used to define operations on the collections, such as** `List`**,** `Set`**,** `SortedSet`**,** `Map`**, and** `SortedMap`

Collections are objects that hold other objects that are accessed, placed, and maintained under some set of rules.

The most basic interface is Collection. The interfaces extend Collection: Set, List, and SortedSet. The other two collection interfaces, Map and SortedMap, do not extend Collection, as they represent mappings rather than true collections. However, these interfaces contain *collection-view* operations, which allow them to be manipulated as collections

The Collection Class



Overview: Utilities

- Utility Interfaces
  - Comparator
  - Iterator
- Utility Classes
  - Collections
  - Arrays

Collection Interface

- **A *collection* is a group of data manipulate as a single object.**
- **Corresponds to a *bag.***
- **Insulate client programs from the implementation.**
  - **array, linked list, hash table, balanced binary tree**
- **Can grow as necessary.**
- **Contain only Objects (reference types).**
- **Heterogeneous.**
- **Can be made thread safe (concurrent access).**
- **Can be made not-modifiable.**

The Collection interface is the interface that acts as the root of all collection classes. It defines the methods that all collections (except Map type collections)
must implement.

Collection Interface

- **Major methods:**
  - **int size();**
  - **boolean isEmpty();**
  - **boolean contains(Object);**
  - **Iterator iterator();**
  - **Object[] toArray();**
  - **boolean add(Object);**
  - **boolean remove(Object);**
  - **void clear();**

The interface defines methods for adding an element to a collection, accessing

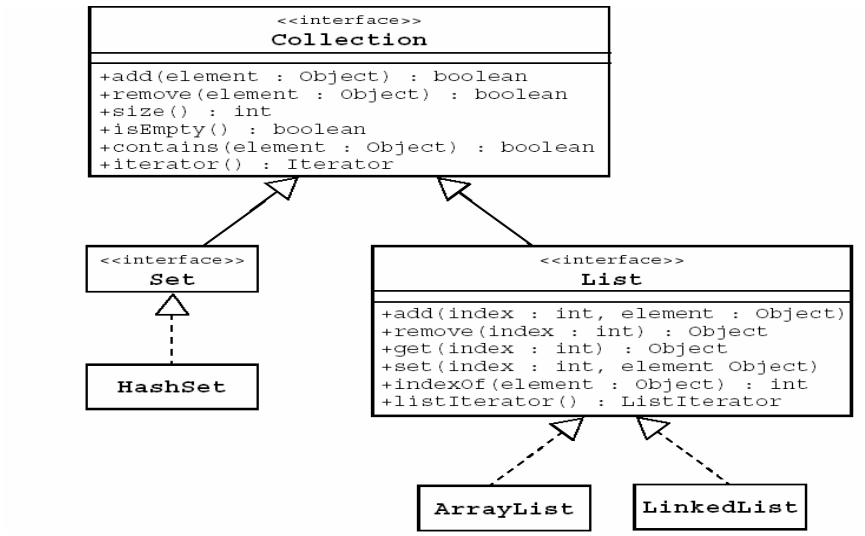
elements in a collection, removing an element from a collection and indicating the

size of the collection.

Collection Interface

- Advantages
  - Can hold different types of objects.
  - Resizable
- Disadvantages
  - Must cast to correct type
  - Cannot do compile-time type checking.

Collection API



List

- interface List extends Collection
- An ordered collection of objects
- Duplicates allowed

A list is an ordered collection of elements. That is a list has a very specific sequence

to the elements it contains. That order is determined by the order in which objects

are added to the ordered collection / List instance. An implementation of the List

interface can hold any type of object. Implementations of the List interface can be

used in situations where the order in which the objects were added to the instance

must be preserved. In general List implementations will allow duplicate objects.

List Interface

- Extensions compared to the Collection interface
  - Access to elements via indexes, like arrays
  - add (int, Object), get(int), remove(int),
  - set(int, Object) (note set = replace bad name for the method)
- Search for elements
  - indexOf(Object), lastIndexOf(Object)
- Specialized Iterator, call ListIterator
- Extraction of sublist
  - subList(int fromIndex, int toIndex)

List Details

- **Major additional methods:**
  - Object get(int);
  - Object set(int, Object);
  - int indexOf(Object);
  - int lastIndexOf(Object);
  - void add(int, Object);
  - Object remove(int);
  - List subList(int, int);
  - **add() inserts**
  - **remove() deletes**
- **Implemented by:**
  - **ArrayList, LinkedList, Vector**

List Example

```
import java.util.*;
public class ListExample {
  public static void main(String[] args) {
    List list = new ArrayList();
    list.add("one");
    list.add("second");
    list.add("3rd");
    list.add(new Integer(4));
    list.add(new Float(5.0F));
    list.add("second"); // duplicate, is added
    list.add(new Integer(4)); // duplicate, is added
    System.out.println(list);
  }
}
```

ArrayList

- **Similar to an array, but dynamic: you don't have to worry about running out of bounds.**
- **When your ArrayList becomes full, ArrayList automatically:**
  - **creates a new array 10% bigger**
  - **copies the data from the old to the new array**
- **Constructors**
  - **public ArrayList(int initialCapacity)**
  - **public ArrayList()**
  - **public ArrayList(Collection c)**

A concrete subclass of the AbstractList class.

Each ArrayList instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally.

public ArrayList(int initialCapacity) -Constructs an empty list with the specified initial capacity.

public ArrayList() -Constructs an empty list with an initial capacity of ten.

public ArrayList(Collection c)-Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator. The ArrayList instance has an initial capacity of 110% the size of the specified collection.

ArrayList Example

```
import java.util.*;
public class Test {
  public static void main(String args []) {
   ArrayList list = new ArrayList(10);
   list.add("John");
   list.add("Denise");
   list.add("Phoebe");
   list.add("John");
   Iterator it = list.iterator();
   while (it.hasNext()) {
   System.out.println(it.next());}
  }
}
```

The results obtained from running this example are :

C:>java Test

John

Phoebe

Denise

John

LinkedList

- **a doubly-linked list implementation**
- **May provide better performance than ArrayList if elements frequently inserted/deleted within the List**
- **For queues and double-ended queues (deques)**
  - **public LinkedList()**
  - **public LinkedList(Collection c)**

The LinkedList class is a concrete subclass of the AbstractSequentialList class. It

provides a doubly linked list implementation of the List interface. This class is

particularly good for sequential access oriented operations and for insertions and

deletions.

Implements all optional list operations, and permits all elements (including null). In addition to implementing the List interface, the LinkedList class provides uniformly named methods to get, remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, queue, or double-ended queue (deque).

public LinkedList() -Constructs an empty list.

public LinkedList(Collection c) -Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

LinkedList Example

```
import java.util.*;
public class MyStack {
  private LinkedList list = new LinkedList();
  public void push(Object o){
   list.addFirst(o);    }
  public Object top(){
   return list.getFirst();    }
  public Object pop(){
   return list.removeFirst(); }
  public static void main(String args[]) {
   Car myCar;
   MyStack s = new MyStack();
   s.push (new Car());
   myCar = (Car)s.pop();  }
}
```

Set

- interface Set extends Collection
- An unordered collection of objects
- No duplicate elements
- Same methods as Collection
  - Semantics are different, so different interface needed for design
- Implemented by:
  - HashSet, TreeSet

The interface Set is basically the same as the interface Collection, with the exception

that it does not allow duplicates. That is, it is only possible to hold a single reference

to an object in a set.

HashSet

- **A Set backed by a hash table**
  - **public HashSet()**
  - **public HashSet(Collection c)**
  - **public HashSet(int initialCapacit, float loadFactor)**
  - **public HashSet(int initialCapacity)**

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.

public HashSet() -Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).

public HashSet(Collection c) -Constructs a new set containing the elements in the specified collection. The HashMap is created with default load factor (0.75) and an initial capacity sufficient to contain the elements in the specified collection.

public HashSet(int initialCapacit, float loadFactor) Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

public HashSet(int initialCapacity) Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor, which is 0.75.

HashCode

- **Every object has a default hash code that is derived from the objects memory address.**
- **The hashCode method only gets invoked when one uses the object as the key to a Hashtable**

```
public int hashCode()  {
   int hash = 0;
   int len = char.length();
   for ( int i=0; i<len; i++ ) {
          hash = 31 * hash + val[i];
   }
   return hash;
}
```

If one wants to file something away for later retrieval, it can be faster if one files it numerically rather than by a long alphabetic key. A hashCode is a way of computing a small (32-bit) digest numeric key from a long String or even an arbitrary clump of bytes.

Hash codes are used to organize elements in the collections, calculated from the state of an object; not necessarily unique

Hash code is calculated recursively on every *significant* field and referenced object, and then combined into one **int** result

HashSet Example

```
import java.util.*;
public class Test {
  public static void main(String args []) {
   HashSet set = new HashSet(10);
   set.add("John");
   set.add("Denise");
   set.add("Phoebe");
   set.add("John");
   Iterator it = set.iterator();
   while (it.hasNext()) {
         System.out.println(it.next()); }
  }
}
```

This class creates a simple set and adds four strings to it (one of which is a duplicate).

The result of executing this class is:

C:>java Test

John

Phoebe

Denise

equals() Method

Reflexive: for any reference value x, x.equals(x) should return true.

Symmetric: for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.

Transitive: for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

Consistent: for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.

For any non-null reference value x, x.equals(null) should return false.

If two objects are equal, then they must have the same hash code, however the opposite is NOT true.

http://www.geocities.com/technofundo/tech/java/equalhash.html

Tree Set

- **A balanced binary tree implementation**
- **Imposes an ordering on its elements**
  - **public TreeSet()**
  - **public TreeSet(Comparator c)**
  - **public TreeSet(Collection c)**
  - **public TreeSet(SortedSet s)**

This class implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the *natural order* of the elements (see Comparable), or by the comparator provided at set creation time, depending on which constructor is used.

public TreeSet() -Constructs a new, empty set, sorted according to the elements' natural order. All elements inserted into the set must implement the Comparable interface. Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the set. If the user attempts to add an element to the set that violates this constraint (for example, the user attempts to add a string element to a set whose elements are integers), the add(Object) call will throw a ClassCastException.

public TreeSet(Comparator c)-Constructs a new, empty set, sorted according to the specified comparator. All elements inserted into the set must be *mutually comparable* by the specified comparator: comparator.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the set. If the user attempts to add an element to the set that violates this constraint, the add(Object) call will throw a ClassCastException.

public TreeSet(Collection c) Constructs a new set containing the elements in the specified collection, sorted according to the elements' *natural order*. All keys inserted into the set must implement the Comparable interface. Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) must not throw a ClassCastException for any elements k1 and k2 in the set.

public TreeSet(SortedSet s) Constructs a new set containing the same

TreeSet Example

```
import java.util.*;
public class HashTreeSetEx{
  public static void main (String args[]){
   Set set = new HashSet();
   set.add("one");
   set.add("two");
   set.add("three");
   set.add("four");
   set.add("one");
   System.out.println(set);
   Set sortedSet= new TreeSet(set);
   System.out.println(sortedSet);
  }
}
```

The program produces the following output.
[one, two, three, four]
[four, one, three, two]

Comparable

- **An interface imposes a total ordering on the objects of each class that implements it.**
- **the class's compareTo method is referred to as its natural comparison method.**
  - **public interface Comparable { public int compareTo(Object o); }**

The compareTo method compares the receiving object with the specified object, and returns a negative integer, zero, or a positive integer as the receiving object is less than, equal to, or greater than the specified Object.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

This method should allow one object to be compared to another. The result of this comparison is referred to as the natural ordering of the objects. Arrays of Objects that implement this interface can be sorted automatically by List.sort.The interface assumes that concrete implementations of this interface will implement the int compareTo(Object o) method such that the method return negative if the receiver is less than the object passed to the method, zero if they are equal and positive if the receiver is greater.

Comparable

- the relation that defines the natural ordering on a given class C is:

  {(x, y) such that x.compareTo((Object)y) <= 0}.

- The quotient for this total order is:

  {(x, y) such that x.compareTo((Object)y) == 0}.

Comparable Example

```
public class Name implements Comparable {
  private String firstName, lastName;
  ...
  public int compareTo(Object o)
  {
    Name n = (Name)o;
    int lastCmp = lastName.compareTo(n.lastName);
    return lastCmp != 0 ? lastCmp :
           firstName.compareTo(n.firstName);
  }
}
```

Vector

- **a synchronized resizable-array implementation of a List with additional "legacy" methods.**
  - **protected Object[] elementData**
  - **protected int elementCount**
  - **protected int capacityIncrement**

A Vector is an historical collection class that acts like a growable array, but can store heterogeneous data elements.

Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement. The capacity is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement.

protected Object[] elementData-The array buffer into which the components of the vector are stored. The capacity of the vector is the length of this array buffer, and is at least large enough to contain all the vector's elements. Any array elements following the last element in the Vector are null.

protected int elementCount-The number of valid components in this Vector object. Components elementData[0] through elementData[elementCount-1] are the actual items

protected int capacityIncrement-The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity. If the capacity increment is less than or equal to zero, the capacity of the vector is doubled each time it needs to grow.

Map

- interface Map (does not extend Collection)
- An object that maps keys to values
- Each key can have at most one value
- Replaces java.util.Dictionary interface
- Ordering may be provided by implementation class, but not guaranteed

The Map interface is not an extension of Collection interface. Instead the interface starts of it's own interface hierarchy, for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by defination.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Map Details

- **Major methods:**
  - **int size();**
  - **boolean isEmpty();**
  - **boolean containsKey(Object);**
  - **boolean containsValue(Object);**
  - **Object get(Object);**
  - **Object put(Object, Object);**
  - **Object remove(Object);**
  - **void putAll(Map);**
  - **void clear();**
- **Implemented by:**
  - **HashMap, Tree Map, Hashtable, WeakHashMap, Attributes**

HashMap

- **A hash table implementation of Map**
- **Like Hashtable, but supports null keys & values**
  - **public HashMap(int initialCapacity, float loadFactor)**
  - **public HashMap(int initialCapacity)**
  - **public HashMap()**
  - **public HashMap(Map m)**

This class is a concrete subclass of the AbstractMap class. **For inserting, deleting and locating elements in a Map the HashMap offers best alternatively. Using a HashMap requires that the class of key added have a well-defined hashCode() implementation.**

An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the rehash method. This implementation provides all of the optional map operations, and permits null values and the null key.

public HashMap(int initialCapacity, float loadFactor)-Constructs an empty HashMap with the specified initial capacity and load factor.

public HashMap(int initialCapacity)-Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

public HashMap()-Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

Constructs a new HashMap with the same mappings as the specified Map. The HashMap is created with default load factor (0.75) and an initial capacity sufficient to hold the mappings in the specified Map

HaspMap Example

```
import java.util.*;
public class Freq {
  private static final Integer ONE = new Integer(1);
  public static void main(String args[]) {
      Map m = new HashMap();
      // Initialize frequency table from command line
      for (int i=0; i < args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
            new Integer(freq.intValue() + 1)));
      }
      System.out.println(m.size()+" distinct words
  found:");
      System.out.println(m);
  }
}
```

TreeMap

- **A balanced binary tree implementation**
- **Imposes an ordering on its elements**
- **Elements are always stored in sorted order**
  - **public TreeMap()**
  - **public TreeMap(Comparator c)**
  - **public TreeMap(Map m)**
  - **public TreeMap(SortedMap m)**

This class is a concrete subclass of the AbstractMap class and implements a binary

tree as the data structure for holding the mappings from keys to values. the binary

tree is ordered on the keys and thus guarantees that the key-value pairs are in key

ordered. This class thus provides guaranteed log(n) time cost for the containsKey,

get, put and remove operations.

public TreeMap()- Constructs a new, empty map, sorted according to the keys' natural order. All keys inserted into the map must implement the Comparable interface. Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) must not throw a ClassCastException for any elements k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint (for example, the user attempts to put a string key into a map whose keys are integers), the put(Object key, Object value) call will throw a ClassCastException.

public TreeMap(Comparator c) Constructs a new, empty map, sorted according to the given comparator. All keys inserted into the map must be *mutually comparable* by the given comparator: comparator.compare(k1, k2) must not throw a ClassCastException for any keys k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint, the put(Object key, Object value) call will throw a ClassCastException.

HashMap / TreeMap

- For storing key / value pairs
- Each key has at most one associated value
- e.g
  - drivers license number / driver
  - username / password
  - ISBN / book title
- HashMap keys are stored in a hash table
- TreeMap keys are stored in a binary tree

Hashtable

- **Synchronized hash table implementation of Map interface, with additional "legacy" methods.**
  - **public Hashtable(int initialCapacity, float loadFactor)**
  - **public Hashtable(int initialCapacity)**
  - **public Hashtable()**
  - **public Hashtable(Map t)**

This class implements a hashtable, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

An instance of Hashtable has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of *buckets* in the hash table, and the *initial capacity* is simply the capacity at the time the hash table is created. Note that the hash table is *open*: in the case of a "hash collision", a single bucket stores multiple entries, which must be searched sequentially. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the rehash method.

public Hashtable(int initialCapacity, float loadFactor)-Constructs a new, empty hashtable with the specified initial capacity and the specified load factor.

public Hashtable(int initialCapacity)-Constructs a new, empty hashtable with the specified initial capacity and default load factor, which is 0.75.

public Hashtable() Constructs a new, empty hashtable with a default initial capacity (11) and load factor, which is 0.75.

public Hashtable(Map t)- Constructs a new hashtable with the same mappings as the given Map. The hashtable is created with an initial capacity sufficient to hold the mappings in the given Map and a default load factor, which is 0.75.

Iterator

- Represents a loop
- Created by Collection.iterator()
- Similar to Enumeration
  - Improved method names
  - Allows a remove() operation on the current item

Iterator Methods

- **boolean hasNext()**
  - **Returns `true` if the iteration has more elements**
- **Object next()**
  - **Returns next element in the iteration**
- **void remove()**
  - **Removes the current element from the underlying Collection**

Iterator Example

```
static void filter(Collection c)
{
  for (Iterator it = c.iterator() ; it.hasNext(); )
    if (!cond(it.next()))
      it.remove();
}

static void filter(Collection c)
{
  Iterator it = c.iterator();
  while (it.hasNext())
    if (!cond(it.next()))
      it.remove();
}
```

ListIterator

- Interface ListIterator extends Iterator
- Created by List.listIterator()
- Adds methods to
  - traverse the List in either direction
  - modify the List during iteration
- Methods added:
  - hasPrevious(), previous()
  - nextIndex(), previousIndex()
  - set(Object), add(Object)

The ListIterator interface extends the Iterator interface to support bi-directional access as well as adding or removing or changing elements in the underlying collection.
An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next(). In a list of length n, there are n+1 valid index values, from 0 to n, inclusive.

Sorting

- Collections.sort() static method
- SortedSet, SortedMap interfaces
  - Collections that keep their elements sorted
  - Iterators are guaranteed to traverse in sorted order
- Ordered Collection Implementations
  - TreeSet, TreeMap

Sorting

- **Comparable interface**
  - **Must be implemented by all elements in SortedSet**
  - **Must be implemented by all keys in SortedMap**
  - **Method: int compareTo(Object o)**
  - **Defines "natural order" for that object class**
- **Comparator interface**
  - **Defines a function that compares two objects**
  - **Can design custom ordering scheme**
  - **Method: int compare(Object o1, Object o2)**

Sorting

- Total vs. Partial Ordering
  - Technical, changes behavior per object class
- Sorting Arrays
  - Use Arrays.sort(Object[])
  - Equivalent methods for all primitive types
    - Arrays.sort(int[]), etc.

Summary

- The various collection API interfaces and classes forms the basis of the data structures and is the corner stone of most implementations.
- Collections
  - Generalization of the array concept.
  - Set of interfaces defined in Java for storing object.
  - Multiple types of objects.
  - Resizable.
- Queue, Stack, Deque classes absent
  - Use LinkedList.