Java Collections

PlanetSoft

What n Why a Collection?

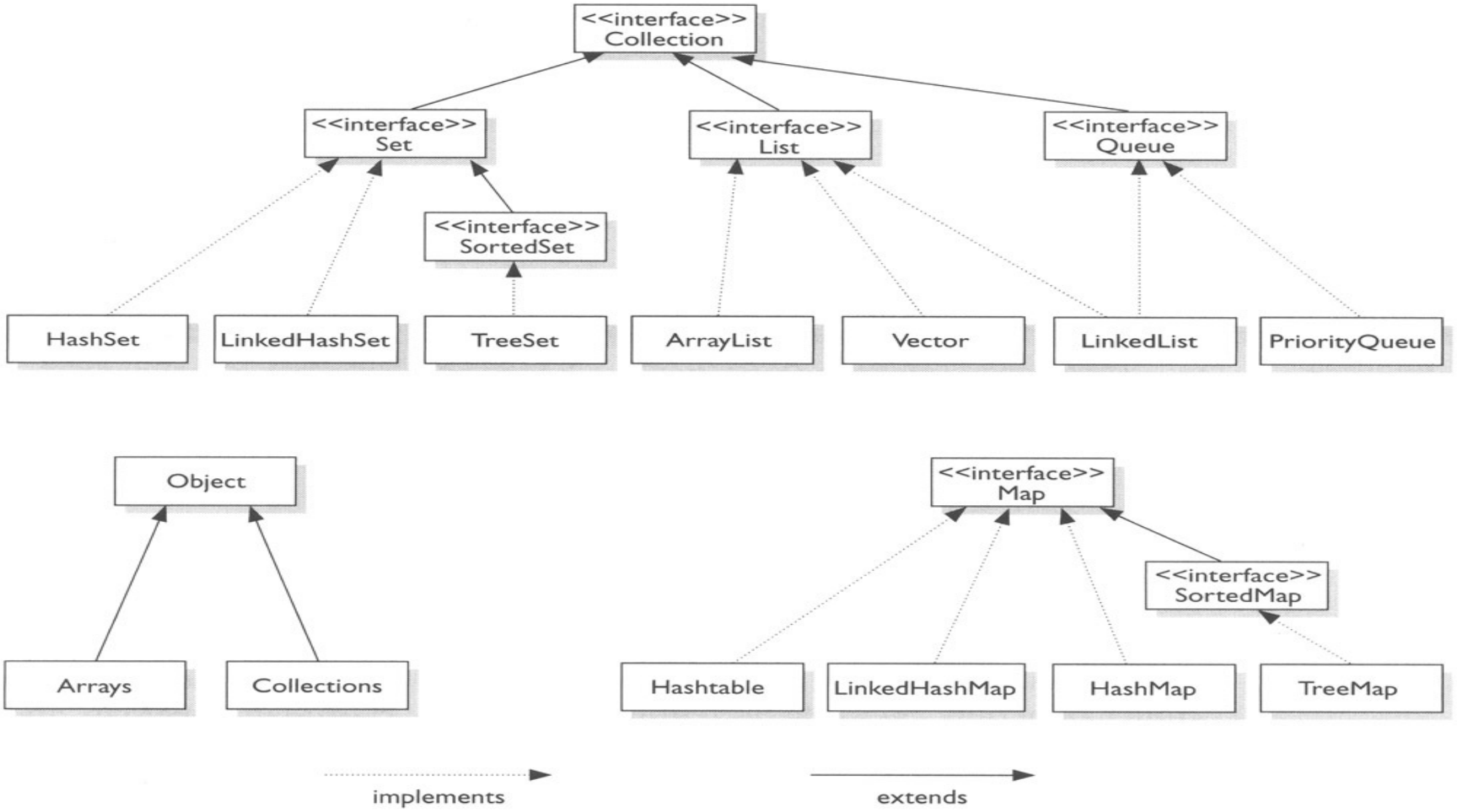
- What?
  - Is simply an object that groups multiple elements into a single unit
- Why?
  - store, retrieve, manipulate, and communicate aggregate data

PlanetSoft

Benefits of Collection Framework

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs
  - The collection interfaces are the vernacular by which APIs pass collections back and forth
- Reduce effort to learn and use new APIs
- Reduces effort to design new APIs

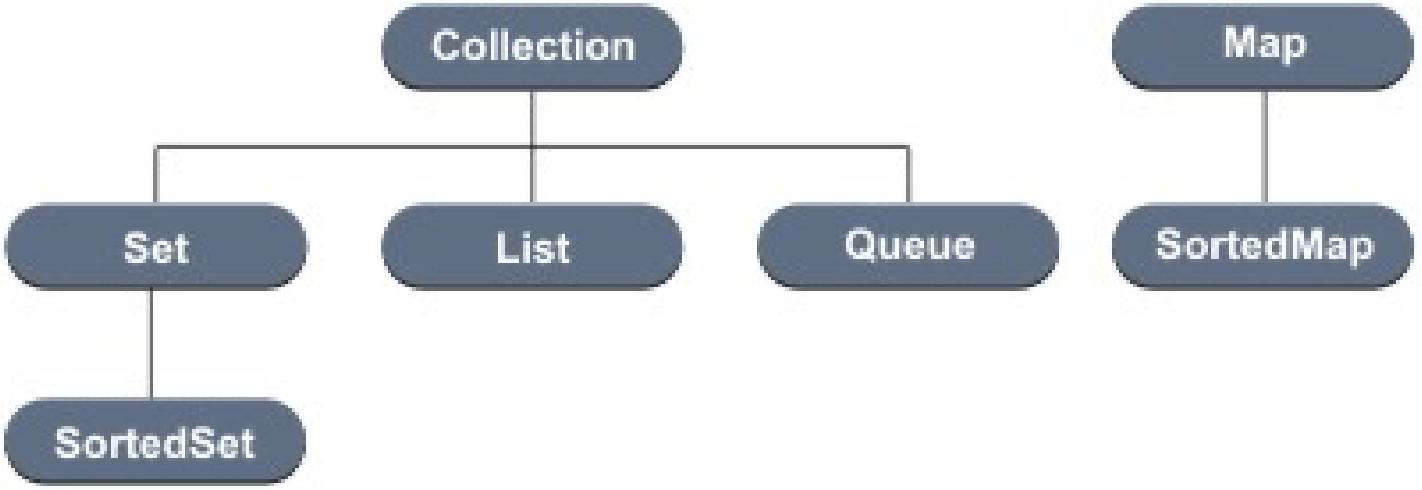
The Java Collections Framework



PlanetSoft

Collection Interfaces

- Collections are primarily defined through a set of interfaces
  - Supported by a set of classes that implement the interfaces



PlanetSoft

Collection Interfaces (Contd.)

- Interfaces are used of flexibility reasons
  - Programs that uses an interface is not tightened to a specific implementation of a collection
  - It is easy to change or replace the underlying with another (more efficient) class that implements the same interface

Collection Interface

- The root of the collection hierarchy
- Is the least common denominator that all collection implement
  - Every collection object is a type of Collection interface
- Is used to pass collection objects around and to manipulate them when maximum generality is desired
- JDK doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List

Collection Interface (Contd.)

```
public interface Collection<E> extends Iterable<E> {
        //Basic Operations
        int size();
        boolean isEmpty();
        boolean contains(Object element);
        boolean add(E element);               //optional
        boolean remove(Object element);   //optional
        Iterator<E> iterator();

        //Bulk Operations
        boolean containsAll(Collection<?> c);
        boolean addAll(Collection<? extends E> c);        //optional
        boolean removeAll(Collection<?> c);                                  //optional
        boolean retainAll(Collection<?> c);                  //optional
        void clear();               //optional

        //Array Operations
        Object[] toArray();
        <T> T[] toArray(T[] a);
}
```

PlanetSoft

Collection Interface (Contd.)

- The *add()* method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't
- It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call
  - *add()* method of Set interface follows "no duplicate" rule
- for-each:
  - The for-each construct allows you to concisely traverse a collection or array using a *for* loop
  - for (Object o : collection)
    - System.out.println(o);

The Iterator interface

- An Iterator is an object that enables through a collection and to remove elements from the collection selectively, if desired
- fail-fast – Exception thrown if collection is modified externally, i.e., not via the iterator (multi-threading)

```
public interface Iterator {
    boolean hasNext();
    Object next();     //Note – one way
    void remove();    //optional
}
```

PlanetSoft

The Iterator Interface (contd.)

- *hasNext()* returns true if the iteration has more elements
- *next()* method returns the next element in the iteration
- *remove()* is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress
  - When called and supported, the element returned by the last *next()* call is removed
  - The for-each construct hides the iterator, so you cannot call remove; so for-each not usable for filtering

The Iterator Interface (contd.)

Example:

```
Collection collection = …;
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    Object element = iterator.next();
    if (removalCheck(element))
       iterator.remove();
}
```

Collection Interface: Bulk Operations

- *containsAll()* – returns true if the target collection contains all of the elements in the specified collection
- *addAll()* – adds all of the elements in the specified collection to the target collection
- *removeAll()* – removes from the target collection all of its elements that are also contained in the specified collection
- *retainAll()* – retains only those elements in the target collection that are also contained in the specified collection
  - Kind of like intersection
- *clear()* – removes all elements from the collection

The Set Interface

- Corresponds to the mathematical definition of a set (no duplicates are allowed)
- Compared to the Collection interface:
  - Interface is identical
  - Every constructor must create a collection without duplicates
  - The operation **add** cannot add an element already in the set

The Set Interface (Contd.)

- Set Idioms
  - set1 ∪ set2 – set1.addAll(set2)
  - set1 ∩ set2 – set1.retainAll(set2)
  - set1 – set2 – set1.removeAll(set2)
- equals operation – Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ
  - Two Set instances are equal if they contain the same elements
- Implementations of Set interface – HashSet, TreeSet, and LinkedHashSet

HashSet

- Implemented using a hash table
- No ordering of elements
- *add*, *remove*, and *contains* methods constant time complexity O(c)
- Iteration is linear in the sum of the number of entries and its capacity
  - Choosing an initial capacity that's too high can waste both space and time
  - Choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity

HashSet (Contd.)

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected : " + args[i]);
        System.out.println(s.size() + " distinct words detected : " + s);
    }
}
```

LinkedHashSet

- Implemented as a hash table with a linked list running through it
- Provides insertion-ordered iteration (least recently inserted to most recently) and runs nearly as fast as HashSet
- Spares its clients from the unspecified, generally chaotic ordering provided by HashSet without incurring the increased cost associated with TreeSet

SortedSet Interface

- A Set that maintains its elements in ascending order
- Several additional operations are provided to take advantage of the ordering
- Sorted sets are used for naturally ordered sets, such as word lists and membership roll

SortedSet Interface (Contd.)

- Provides access to ends of the set as well as to subsets of the set
- Changes to the subset are reflected in the source set and vice-versa i.e., changes in the source set are reflected in the subset
- Elements added to a SortedSet must either implement Comparable or you must provide a Comparator to the constructor of its implementation class: TreeSet

TreeSet

- Implemented using a tree structure
- Guarantees ordering of elements
- add, remove, and contains methods logarithmic time complexity O(log(n)), where n is the number of elements in the set
- Used when there is a need to use the operations in the SortedSet interface, or if value-ordered iteration is required

TreeSet (Contd.)

```
Set ts = new TreeSet();

ts.add("one");
ts.add("two");
ts.add("three");
ts.add("four");

System.out.println("Members of the TreeSet : " + ts);
```

Output—

Members of the TreeSet : [four, one, three, two]

List Interface

- An ordered collection (sometimes called a sequence)
- Lists can contain duplicate elements
- The user of a list generally has precise control over where in the list each element is inserted and can access elements by their integer index (position)

List Interface (Contd.)

- Additional operations supported by List interface over Collection:
  - Positional Access – manipulates elements based on their numerical position in the list
  - Search – searches for a specified object in the list and returns its numerical position
  - Iteration – extends Iterator semantics to take advantage of the list's sequential nature
  - Range-view – performs arbitrary range operations on the list

List Interface (Contd.)

- Further requirements compared to the Collection interface:
  - add(Object) – Adds at the end of the list
  - remove(Object) – Removes at the start of the list
  - list1.equals(list2) – The ordering of the elements is taken into consideration
  - hashCode – Extra requirements to this method
    - list1.equals(list2) implies that list1.hashCode()==list2.hashCode()

List Interface (Contd.)

```
public interface List<E> extends Collection<E> {
        //Positional access
        E get(int index);
        E set(int index, E element);            //optional
        boolean add(E element);                 //optional
        void add(int index, E element);         //optional
        E remove(int index);   //optional
        boolean addAll(int index, Collection<? extends E> c);              //optional

        //Search
        int indexOf(Object o);
        int lastIndexOf(Object o);

        //Iteration
        ListIterator<E> lsitIterator();
        ListIterator<E> listIterator(int index);

        //Range-view
        List<E> subList(int from, int to);
}
```

PlanetSoft

ListIterator Interface

- ListIterator extends Iterator
  - boolean hasNext(); Object next();
  - boolean hasPrevious(); Object previous();
  - int nextIndex(); int previousIndex();
  - void remove(); void set(Object o); void add(Object o) – All three optional
- The ListIterator is originally positioned beyond the end of the list i.e. list.size(), as the index of the first element is 0

Example:

```
List list = …;
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
    Object element = iterator.previous
    //Process element
}
```

PlanetSoft

ListIterator Interface (Contd.)

- Normally, ListIterator is not used to alternate between going forward and backward in one iteration through the elements of a collection, though technically possible
- *add()* – Adding an element results in the new element being added immediately prior to the implicit cursor
  - calling previous() after adding an element would return the new element
  - calling next would have no effect after adding an element

Range-view of List Interface

- subList():
  - Element at fromIndex is in the sublist, element at toIndex is not
  - Loosely maps to the following for-loop:

    for (int i=fromIndex; i<toIndex; i++)
- Changes to the sublist (like *add()*, *remove()* and *set()* calls) have an effect on the underlying list

ArrayList

- Is an array based implementation where elements can be accessed directly via the *get* and *set* methods
  - Default choice for simple sequence
  - Offers constant-time positional access
  - Fast
  - Think of ArrayList as Vector without synchronization overhead
  - Use it if you need to support random access, without inserting or removing elements from any place other than the end

LinkedList

- LinkedList is based on a double linked list
  - Gives better performance on *add* and *remove* compared to ArrayList
  - Gives poorer performance on *get* and *set* methods compared to ArrayList
  - Use it if you need to frequently add and remove elements from the middle of the list and only access the list elements sequentially

LinkedList (Contd.)

- New methods of LinkedList:
  - void addFirst(element : Object);
  - void addLast(element : Object);
  - Object getFirst();
  - Object getLast();
  - Object removeFirst();
  - Object removeLast();
- Using these methods, it can be treated as a stack, or queue, or other end-oriented data structure
  - LinkedList as queue – Use addFirst() and removeLast() methods
  - LinkedList as stack – Use addLast() and removeFirst() methods

ArrayList and LinkedList Example

```
List arrList = new ArrayList();
arrList.add(“Naren”);
arrList.add(“Uday”);
arrList.add(“Suresh”);
arrList.add(“Bharath”);
arrList.add(“Naren”);
System.out.println(arrList);
System.out.println(“1 : ” + arrList.get(1));
System.out.println(“0 : ” + arrList.get(0));

LinkedList queue = new LinkedList();
queue.addFirst(“Naren”);
queue.addFirst(“Uday”);
queue.addFirst(“Suresh”);
queue.addFirst(“Bharath”);
queue.addFirst(“Naren”);
System.out.println(queue);
queue.removeLast();
queue.removeLast();
System.out.println(queue);
```

The Map Interface

- Maps keys to values, handles key/value pairs
- A Map cannot contain duplicate keys; each key can map to atmost one value
- Methods for adding and deleting:
  - put(Object key, Object value)
  - remove(Object key)
- Methods for extraction objects:
  - get(Object key)
- Methods to retrieve the keys, the values, and (key, value) pairs:
  - keySet() //returns a Set
  - values() //returns a Collection
  - entrySet() //returns a Set

The Map Interface (Contd.)

```
public interface Map<K,V>
    //Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    //Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
```

The Map Interface (Contd.)

```
//Collection Views
public Set<K> keySet();
public Collection<V> values();
public Set<Map.Entry<K,V>> entrySet();

//Interface for entrySet elements
public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
}
```

Implementations of the Map Interface

- HashMap
  - Implementation is based on a hash table
  - Use it if you want maximum speed and don't care about iteration order
  - Most commonly used implementation
- TreeMap
  - Implementation is based on red-black tree structure
  - Use it when you need SortedMap operations or key-ordered Collection-view iteration
- LinkedHashMap
  - Use it if you want near–HashMap performance and insertion-order iteration

Queue Interface

- A collection used to hold multiple elements prior to processing
- Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations
- Typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner
- Also check Deque interface which is a sub-interface of Queue
  - It's a Queue which can be used both forwards or backwards, or both at once
  - Implementations – ArrayDeque and LinkedList

Implementations of Queue Interface

- LinkedList
  - You already know
- ArrayDeque
  - Resizable-array implementation of the Deque interface
  - No capacity restrictions, not thread-safe in the absence of external synchronization
  - Null elements prohibited
  - Faster than Stack when used as stack, and faster than LinkedList when used as a queue

Implementations of Queue Interface (Contd.)

- PriorityQueue
  - PriorityQueue class is a priority queue based on the heap data structure
  - This queue orders elements according to an order specified at construction time, which can be the elements' natural ordering or the ordering imposed by an explicit Comparator

Static Methods on Collections

- Search and sort:
  - int binarySearch(List list, Object key);
  - int binarySearch(List list, Object key, Comparator c);
  - void sort(List list);
  - void sort(List list, Comparator c);
- Reorganization:
  - void reverse(List list);
  - shuffle(List list);
  - shuffle(List list, Random rnd);

Static Methods on Collections (Contd.)

- Wrappings:
  - Collection unmodifiableCollection(Collection c);
  - List unmodifiableList(List list);
  - Map unmodifiableMap(Map m);
  - Set unmodifiableSet(Set s);
  - SortedMap unmodifiableSortedMap(SortedMap m);
  - SortedSet unmodifiableSortedSet(SortedSet s)
  - Similar to all unmodifialbe, synchronized methods too
    - Ex : synchronizedCollection(Collection c);
    - Similarly with List, Map, Set, SortedMap, and SortedSet

Static Methods on Collections (Contd.)

- Others:
  - void copy(List dest, List src);
  - Enumeration enumeration(Collection c);
  - void fill(List list, Object obj);
  - int indexOfSubList(List source, List target);
  - int lastIndexOfSubList(List source, List target);
  - ArrayList list(Enumeration e);
  - Object max(Collection coll);
  - Object max(Collection coll, Comparator comp);
  - Object min(Collection coll);
  - Object min(Collection coll, Comparator comp);
  - List nCopies(int n, Object o); //Immutable List
  - boolean replaceAll(List list, Object oldVal, Object newVal);
  - Comparator reverseOrder();
  - Set singleton(Object o); //Immutable set returned
  - List singletonList(Object o); //Immutable
  - Map singletonMap(Object key, Object value); //Immutable

Abstract Classes

- Includes abstract implementations
  - AbstractCollection
  - AbstractSet
  - AbstractList
  - AbstractSequentialList
  - AbstractMap
- To aid in custom implementations
  - Reduces code you have to write

Historical Collection Classes

- Arrays
- Vector
- Stack
- Enumeration Interface
- Hashtable
- Properties

PlanetSoft

Arrays

- Fixed-size collections of the same datatype
- Only collection that supports storing primitive datatypes
- You specify both the number and type of object you wish to store while creation
- It can neither grow nor store a different type (unless it extends the first type)
- Size – array.length
- Accessing beyond either end of the array will throw ArrayIndexOutOfBoundsException at runtime

Arrays (Contd.)

- When created, arrays are automatically initialized, either to false for a boolean array, null for an Object array, or the numerical equivalent of 0 for everything else
- To make a copy of an array, perhaps to make it larger, use arraycopy() method of System
  - System.arraycopy(Object sourceArray, int sourceStartPosition, Object destinationArray, int destinationStartPosition, int length)
- Helper class – java.util.Arrays
  - Search and sort – binarySearch(), sort()
  - Comparison – equals()
  - Instantiation – fill()
  - Conversion – asList()

Vector and Stack

- Vector acts like a growable array, but can store heterogeneous dataelements
- Instead, use ArrayList
- Stack class extends Vector to implement LIFO with push() and pop() methods
  - Still, you can access or modify a Stack with the inherited Vector methods

Enumeration Interface

- Allows you to iterate through all the elements of a collection
- Enumeration:
  - boolean hasMoreElements();
  - Object nextElement();
- No removal support

PlanetSoft

Hashtable and Properties

- Hashtable – Permits storing any object as its key or value (besides null)
  - **Similar to HashMap**
  - **If synchronized Map is needed, using Hashtable is slightly faster than using a synchronized HashMap**
- Properties – Specialized Hashtable for working with text strings
  - **While you have to cast values retrieved from a Hashtable to your desired class, the Properties class allows you to get text values without casting**
  - **Supports loading and saving property settings from an input stream or to an output stream**
  - **Similar to TreeMap**

Algorithms

- Sorting
- Shuffling
- Routine data manipulation
- Searching
- Composition
- Find extreme values

Algorithms (Contd.)

- Provided as static methods of Collections class
- The first argument is the collection on which the operation is to be performed
- The majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on arbitrary Collection instances

Sorting

- The sort algorithm reorders a List so that its elements are in ascending order according to an ordering relationship
- Two forms of the operations
  - Takes a List and sorts it according to its elements' natural ordering
  - Takes a Comparator in addition to a List and sorts the elements with the Comparator

Natural Ordering

- The type of the elements in a List already implements Comparable interface
- Examples:
  - If the List consists of String elements, it will be sorted into alphabetical order
  - If it consists of Date elements, it will be sorted into chronological order
  - String and Date both implement the Comparable interface
- If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a ClassCastException

Sorting by Natural Order of List

```
String n[] = {
    new String(“One”);
    new String(“Two”);
    new String(“Three”);
    new String(“Four”);
}
List l = Arrays.asList(n);
Collections.sort(l);
```

Output – [Four, One, Three, Two]

Sorting by Comparator Interface

- A comparison function, which imposes a total ordering on some collection of objects
- Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order
- Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering

Sorting by Natural Order of List

```
ArrayList arrList = new ArrayList();
arrList.add(“One”);
arrList.add(“Two”);
arrList.add(“Three”);
arrList.add(“Four”);

Comparator comp = Comparators.stringComparator();
Collections.sort(arrList, comp);
System.out.println(arrList);
```

Shuffling

- The shuffle algorithm does the opposite of what the sort does, destroying any trace of order that may have been present in a List
  - That is, this algorithm reorders the List based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness
- Useful in implementing games of chance
  - Could be used to shuffle a List of card objects representing a deck
  - Generating test cases

Routine Data Manipulation

- The Collections class provides five algorithms for doing routine data manipulation on List objects
  - reverse – reverses the order of the elements in a list
  - fill – overwrites every element in a List with the specified value. This operation is useful for reinitializing a List
  - copy – takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents
    - The destination List must be atleast as long as the source
    - If it is longer, the remaining elements in the destination List are unaffected
  - swap – swaps the elements at the specified positions in a List
  - addAll – adds all the specified elements to a Collection
    - The elements to be added may be specified individually or as an array

Searching

- The Collections class has binarySearch() method for searching a specified element in a sorted List

```
String n[] = {
    new String(“One”);
    new String(“Two”);
    new String(“Three”);
    new String(“Four”);
}
List l = Arrays.asList(n);
int position = Collections.binarySearch(l, “One”);
System.out.println(“Position of the searched item = ” + position);
```

Composition

- Collections.frequency(l,el) – counts the number of times the specified element occurs in the specified collection
- Collections.disjoint(l1, l2) – determines whether two collections are disjoint; i.e., whether they contain no elements in common

Good Bye Slide

- Use the new collections in your new work
- Tell your friends who still use Vector and Hashtable to get a life!!

PlanetSoft

Dedicated to this guy!

PlanetSoft

Chillaxxx! Its over, finally!!
ENJOY!!!