JAVA COLLECTIONS

---Dilip Dedhia, Ph.D.

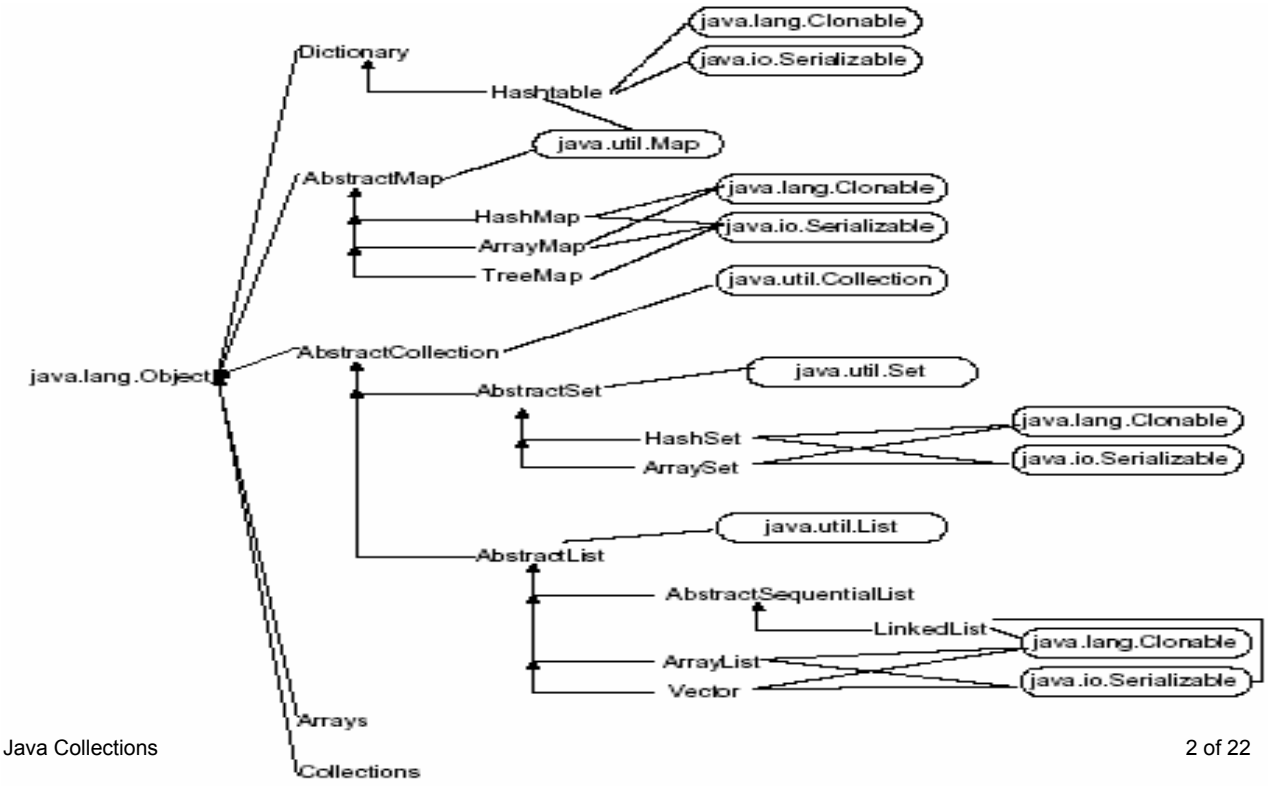
Collections Framework

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

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`

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.

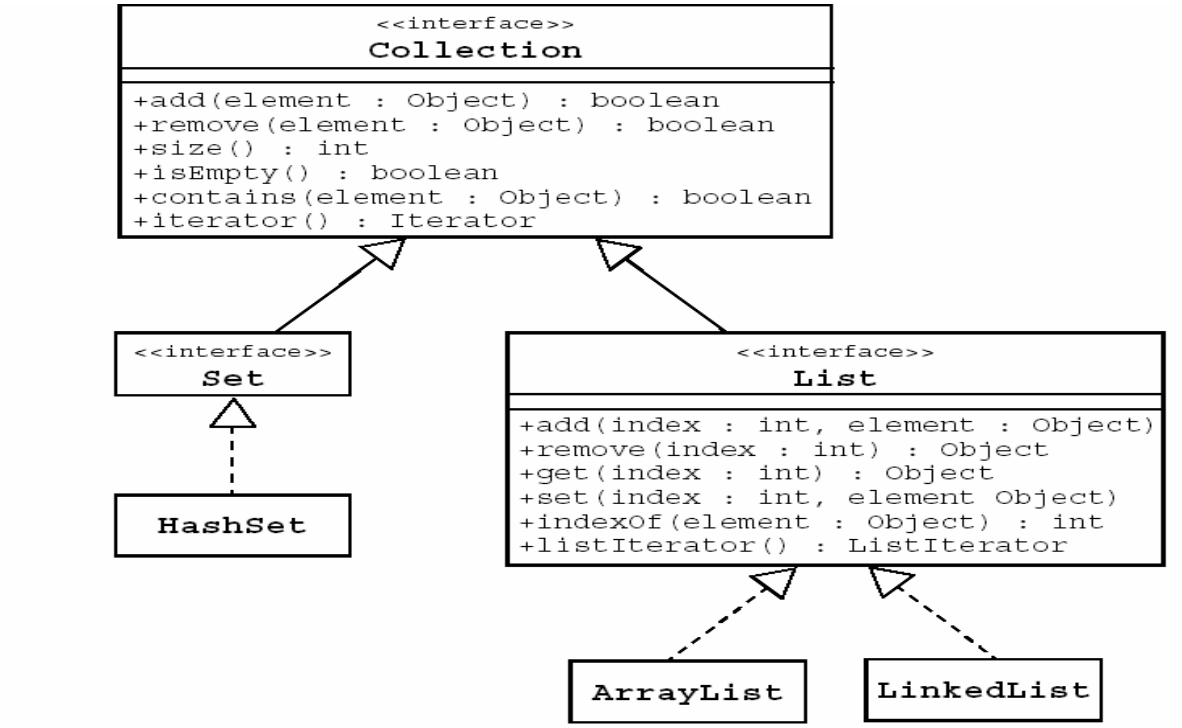
Collection Interface

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

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

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)

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());}
  }
}
```

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)

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

HashSet

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

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;
}
```

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()); }
  }
}
```

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.

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)

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);
  }
}
```

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); }

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

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

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)

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)

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)

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)

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.