Collections

How to Represent Multiplicity

- Using heterogeneous array of objects

Potential issues ?

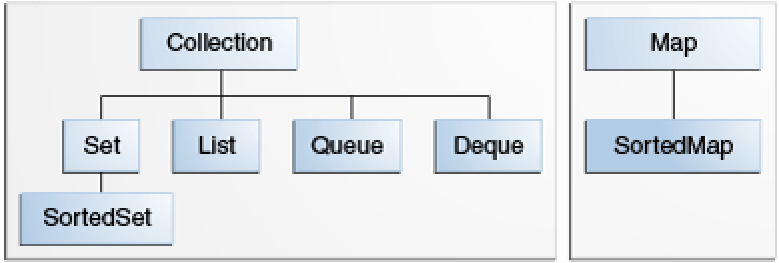
```
Employee[] staff = new Employee[3];

staff[0] = new Manager(…);
staff[1] = new Employee(…);
staff[2] = new Employee(…);

//print salary
for (int i = 0; i<3; i++){
    System.out.println(staff[i].getSalary());
}
```

Collections

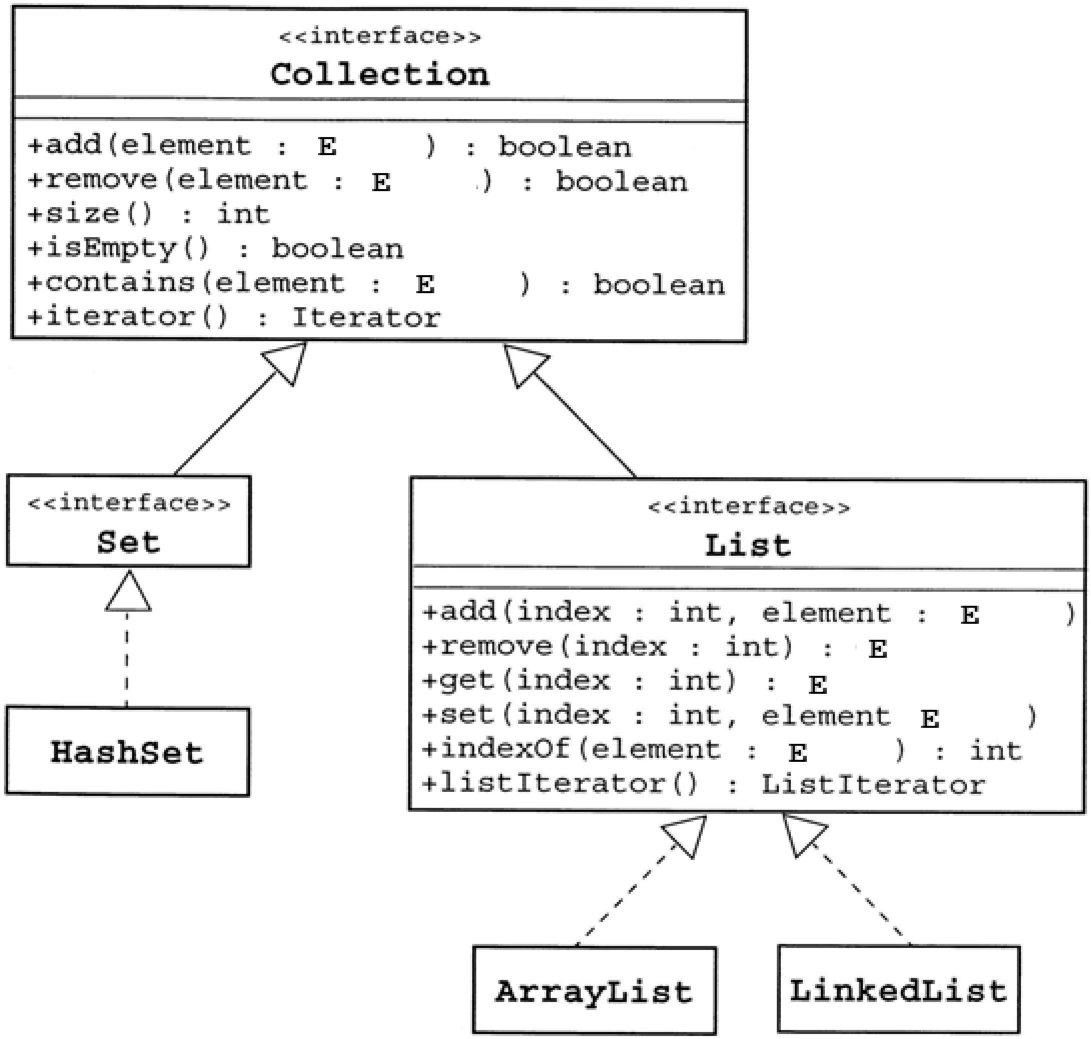
- A single object (data structure) to represent *a group of objects*
  - Like a container
    - Ordered or unordered
    - Duplicates permitted or not
  - Able to add more objects
  - Able to remove unwanted objects
  - No need to track index
  - Able to traverse the whole group

Java Collections

- Core **`Collection`** interface
    - A group of objects (elements); any specific ordering or lack of ordering and allowance of duplicates is specified by each implementation
  - **`Set`** interface
    - Unordered, no duplicates permitted
    - Unique things
  - **`List`** interface
    - Ordered, duplicates permitted
    - Lists of things
  - **`Queue`** interface
    - A collection with additional insertion, removal and inspection operations
  - **`Map`** interface
    - Mapping keys and values
    - Things with a unique ID
  - **`Deque`** interface
    - A collection with additional insertion, removal and inspection operations
    - A double-ended-queue (FIFO/LIFO)



Collections Interface and Class Hierarchy


<<interface>>
Collection
+add(element : E) : boolean
+remove(element : E) : boolean
+size() : int
+isEmpty() : boolean
+contains(element : E) : boolean
+iterator() : Iterator
<<interface>>
Set
HashSet
<<interface>>
List
+add(index : int, element : E)
+remove(index : int) : E
+get(index : int) : E
+set(index : int, element E)
+indexOf(element : E) : int
+listIterator() : ListIterator
ArrayList
LinkedList

Generics

- Similar to C++ templates; not expanded unlike a C++ template
- **Generic interface/class/method** takes ***type parameters***
- Abstract over types; compile-time type safety
- Eliminating the drudgery of casting
- Since 1.5

An arbitrary list

Prior to 1.5

```
List myList = new LinkedList();
myList.add(new Integer(0));
myList.add(“I am a String”);
Integer x = (Integer) myList.iterator().next();
```

Complier does not check the type

Cast is required.
Integer or String ?

Type declaration: generic instantiation

Since 1.5

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Complier checks the type correctness

No cast is required

Example: A List

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

    List<String> list = new ArrayList<String>();

    list.add("one");
    list.add("second");
    list.add("3rd");

    list.add("second");      //duplicate added

    System.out.println(list);
  }
}
```

You can print a collection

Output:

```
[one, second, 3rd, second]
```

Example: A Set

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

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

    words.add("one");
    words.add("second");
    words.add("3rd");

    words.add("second");      //duplicate added

    System.out.println(words);
  }
}
```

Output:

```
[second, one, 3rd]
```

Example: A Map

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

    Map<String, String> medals = new HashMap<String, String>();

    medals.put("one‟, “Gold”);
    medals.put(“second‟, “Light Grey”);
    medals.put(“3rd‟, “Bronze”);

    medals.put(“second‟, “Silver”);   //duplicate key added

    System.out.println(medals);

    System.out.println(medals.get("one"));
    System.out.println(medals.get("second"));
    System.out.println(medals.get("3rd"));
  }
}
```

Output:

```
{second=Silver, one=Gold, 3rd=Bronze}
Gold
Silver
Bronze
```

Example: A Tree Set

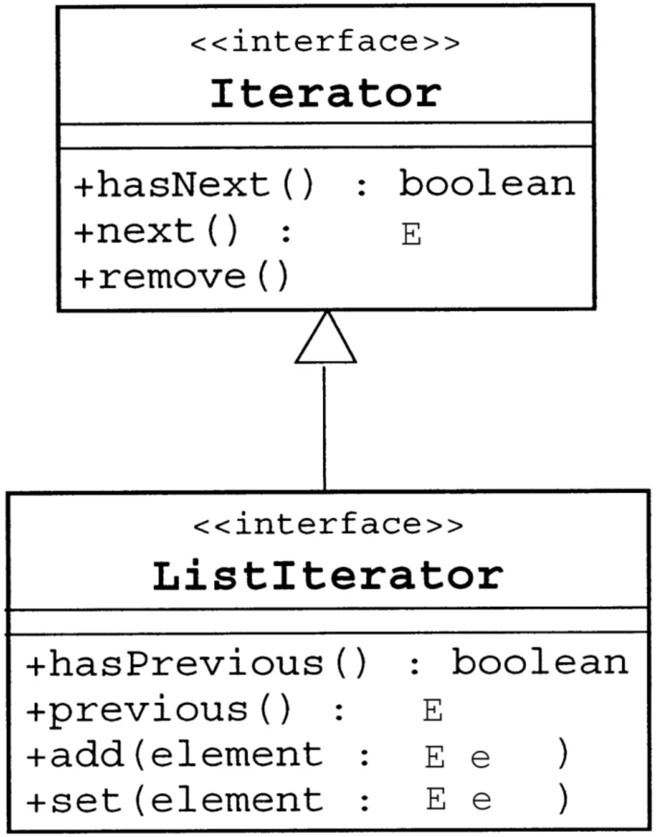
```
public class TreeSetExample {
  public static void main(String[] args) {

    SortedSet<String> words = new TreeSet<String>();

    words.add("one");
    wrods.add("second");
    words.add("3rd");

    words.add("second");       //duplicate added

    System.out.println(words);
  }
}
```

Output:

```
[3rd, one, second]
```

Iterators

- Retrieving every element in a collection
  - An **`Iterator`** of a **`Set`** is unordered
  - A **`ListIterator`** of a **`List`** can be scanned forwards (**`next()`**) or backwards (**`previous()`**)



Traversing Collections

- Using **`for-each`** Construct

```
List<String> list = new ArrayList<String>();
for (String s : list)         //for each string in list
     System.out.println(s);
```

- Using **`Iterator`**s

```
1 List<String> list = new ArrayList<String>();
  list.add(…);
  …
2 Iterator<String> iter = list.iterator();
  while (iter.hasNext() ){ 3
       String element = iter.next(); 4
       // do something with element
  }
```

Traversing a List: Idioms

- Iterating forward through a list

```
for (ListIterator<Type> iter = list.listIterator(); iter.hasNext(); )
{
    Type t = iter.next();
    ...
}
```

- Iterating backward through a list

```
for (ListIterator<Type> iter = list.listIterator(list.size()); iter.hasPrevious(); )
{
    Type t = iter.previous();
    ...
}
```

- Traversing using for-each

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

Bulk Operations

```
containAll()
addAll()
removeAll()
retainAll()
clear()
```

Array Operations

- **`toArray()`** method
  - A bridge between collections and APIs that expect arrays on input

```
Object[] a = c.toArray();
```

- If the collection is known to contain elements of a type

```
List<String> c = new ArrayList<String>();
String[ ] a = c.toArray();
```

- **`Array.asList()`** factory method
  - Allows an array to be viewed as a **`List`**
  - Resulting **`List`** is not a **`List`** implementation; no **`add/remove`** operations: arrays are not resizable

```
String[] ss = new String(10);
List<String> list = Arrays.asList(ss);
```

Positional Access and Search of a List

```
public interface List<E> extends Collection<E> {

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

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

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

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

Object Comparison

- **`Comparable`** interface
  - Defining sort order

```
public interface Comparable<T>
{
     int compareTo(T other);
     // 0 if a equals b;
     // <0 (>0) if a comes before (after) b;
}
```

```
public Item implements Comparable<Item>
{
  public int compareTo(Item other)
  {
    return partNumber - other.partNumber;
  }
}
```

- **`Comparator`** interface
  - Defining order other than its natural ordering by passing a **`Comparator`** to the constructor of an collection

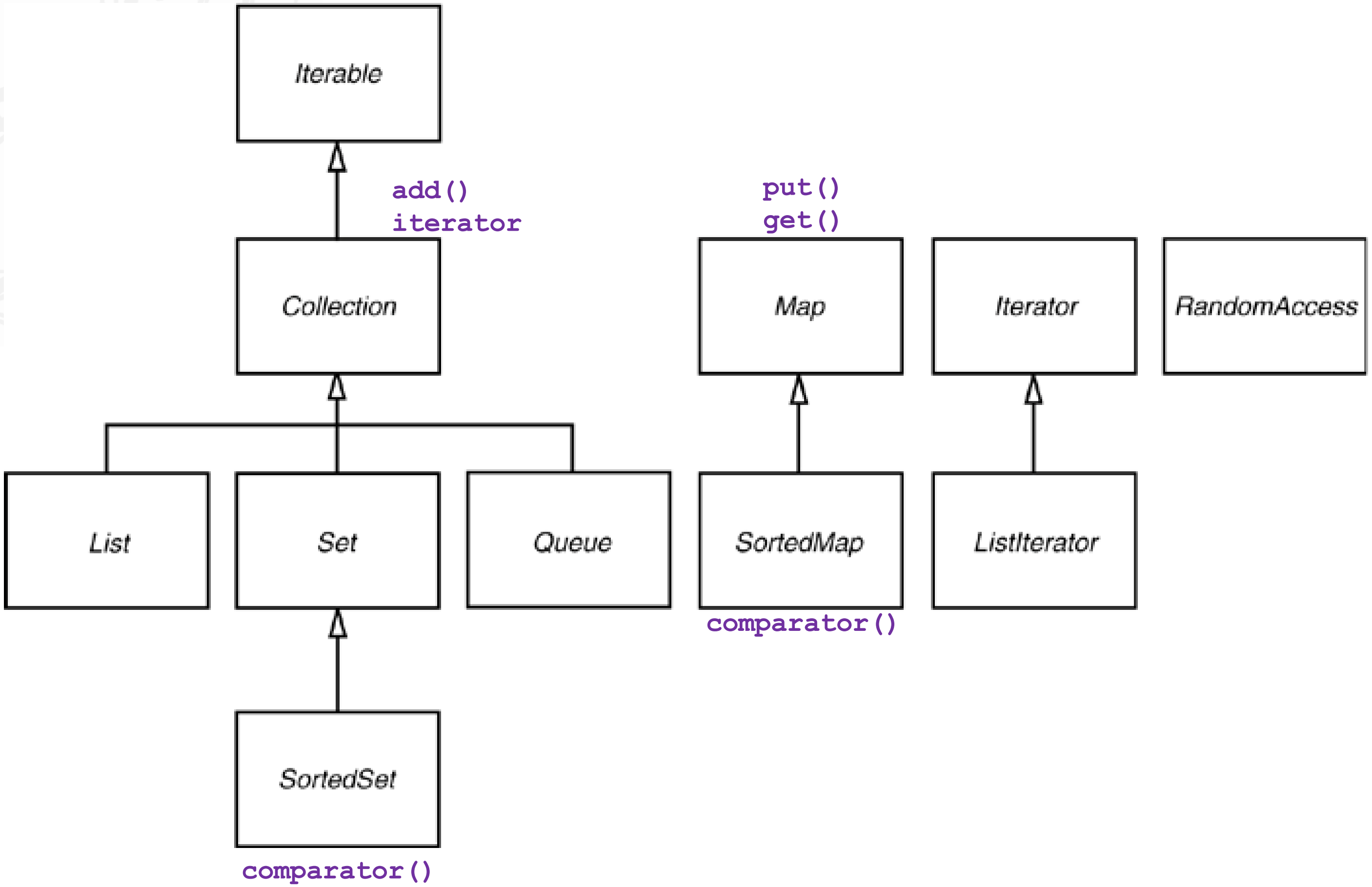
```
public interface Comparator<T>
{
     int compare(T a, T b);
     // 0 if a equals b;
     // <0 (>0) if a comes before (after) b;
}
```

```
SortedSet<Item> sortByDescr = new TreeSet<>(
  new Compartor<Item>()
    {
      public int compare(Item a, Item b)
      {
        String descrA = a.getDescription();
        String descrB = b.getDescription();

        return DescrA.compareTo(descrB);
      }
    }
});
```

Collections Framework

- Interfaces
  - Abstract data types representing collections
  - **`Collection, List, Set, Map`** …
- Implementations
  - Concrete implementations of collection interfaces
    - **`ArrayList, LinkedList, HashSet, TreeSet, HashMap`** …
- Algorithms
  - Polymorphic methods to perform useful computations
    - **`sort(), search(), shuffle(), rotate(), reverse(), swap(), max(), min(), frequency()`** …

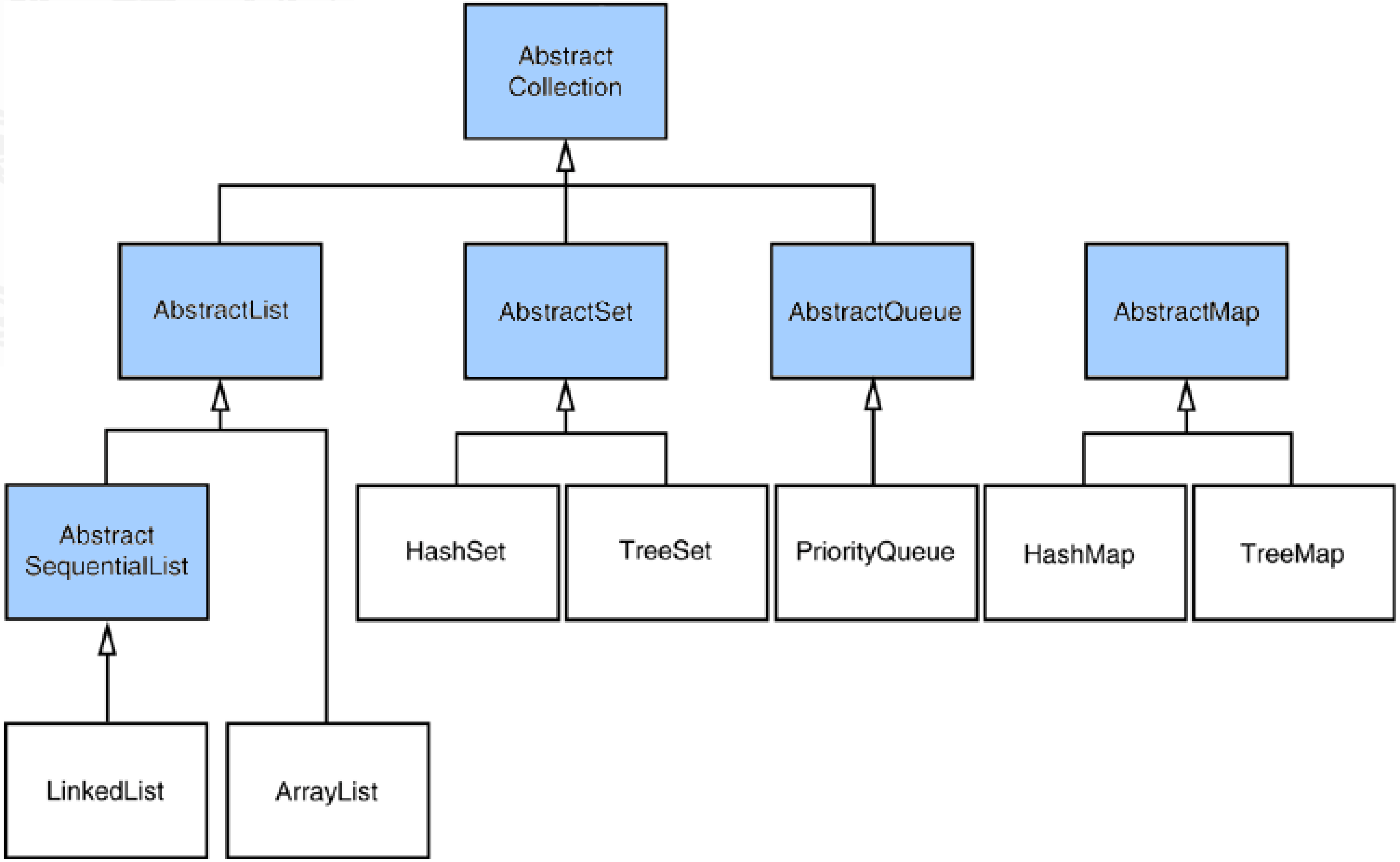
Interfaces of Collection Framework



Implementations

- **General-purpose implementations**
  - the most commonly used implementations, designed for everyday use.
- **Special-purpose implementations**
  - designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations**
  - designed to support high concurrency, typically at the expense of single-threaded performance. These implementations are part of the java.util.concurrent package.
- **Wrapper implementations**
  - used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
- **Convenience implementations**
  - mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
- **Abstract implementations**
  - skeletal implementations that facilitate the construction of custom implementations

Classes in Collection Framework



General-Purpose Implementations

Interfaces	Hash table implementation	Resizable array implementation	Tree implementation	Linked list implementation	Hash table + Linked list implementation
Set	HashSet		TreeSet (*Sorted Set*)		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap (*Sorted Map*)		LinkedHashMap

Queue	LinkedList	Priority Queue
Deque	ArrayDeque	LinkedList

OO Design Principle

Program to an interface, not an implementation

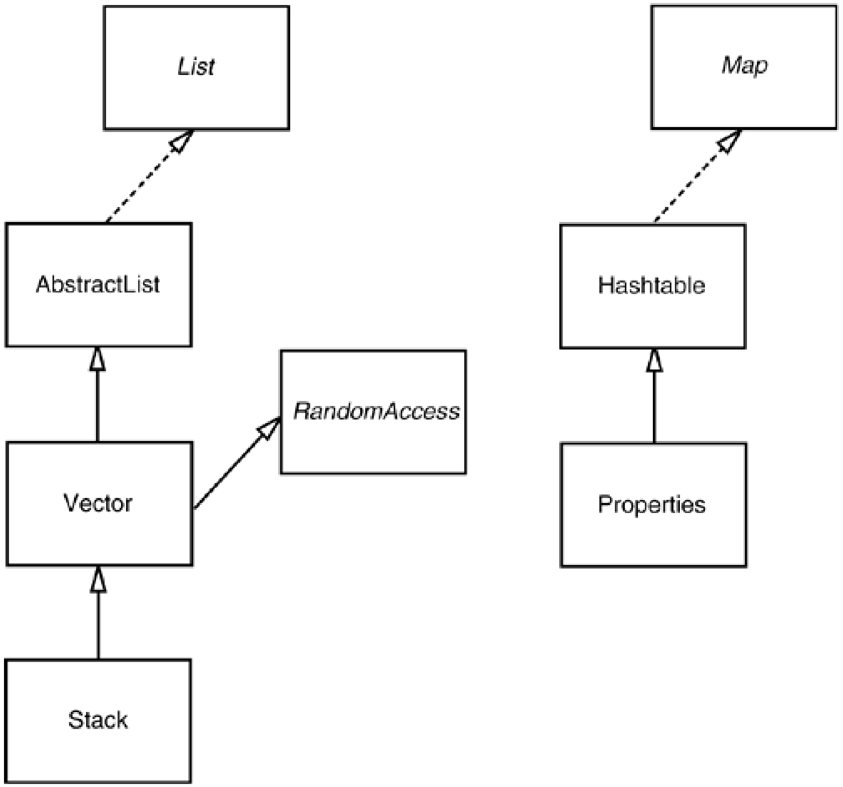
Legacy Container Classes

- Classes to represent collections (since 1.0) before Collections Framework (since 1.2)

- Thread-safe, therefore heavyweight

- Integrated into the Collections Framework

```
Vector
Stack
Hashtable
Properties
```



Wrapper Implementations

- Synchronization Wrappers
  - add automatic synchronization (thread-safety) to an arbitrary collection using following factory methods:

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

- Unmodifiable Wrappers
  - Make a collection immutable once built; allows certain clients read-only access to a collection

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
public static <T> List<T> unmodifiableList(List<? extends T> list);
public static <K,V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m);
public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<? extends T> s);
public  static <K,V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K,? extends V> m);
```

Convenience Implementations

- List view of an array

  eg.

```
//create a fixed-size List
List<String> list = Arrays.asList(new String[size]);
```

- Immutable multiple-copy list

  eg.

```
//add 69 copies of the string “fruit bat” to the end of a List<String>
lovablePets.addAll(Collections.nCopies(69, "fruit bat"));
```

- Immutable singleton set

  eg.

```
//remove all occurences of e from the collection
c.removeAll(Collections.singleton(e));
```

- Empty Set, List and Map constants

  eg.

```
//return a empty set as input to a method exepcting a Collection of values
tourist.declarePurchases(Collections.emptySet());
```

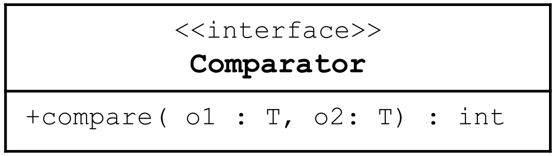
Sort Algorithm

- Reorders a `List` in ascending order according to its elements' *natural ordering*

Class	Natural Ordering
`Byte`	Signed numerical
`Character`	Unsigned numerical
`Long`	Signed numerical
`Integer`	Signed numerical
`Short`	Signed numerical
`Double`	Signed numerical
`Float`	Signed numerical
`BigInteger`	Signed numerical
`BigDecimal`	Signed numerical
`Boolean`	Boolean.FALSE < Boolean.TRUE
`File`	System-dependent lexicographic on path name
`String`	Lexicographic
`Date`	Chronological
`CollationKey`	Locale-specific lexicographic

Sort Algorithm

- Sorting elements with the **`Comparator`** provided to the **`sort()`** method



```
<<interface>>
Comparator
+compare( o1 : T, o2: T) : int
```

```
// An anagram group is a bunch of words, all of which contain
//    exactly the same letters but in a different order
// Read words from file and put into a simulated multimap
Map<String, List<String>> m = new HashMap<String, List<String>>();
…
// Make a List of all anagram groups above size threshold.
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);

// Sort anagram groups according to size
Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
        return o2.size() - o1.size();
    }});
```

It is a common idiom using an anonymous inner class to provide a **`Comparator`** here

Static method !