CL309 - Object Oriented Analysis and Design Lab
Lab#11

Object-Oriented Programming: Collection classes

Objectives

Understanding the concepts of

- Collection Classes

Collections

The Java *Collections API* is a set of classes and interfaces designed to store multiple objects. Collection is the most basic generally useful interface that most, but not all, collection classes implement

- it specifies a number of useful methods, such as those to add and remove elements, ascertain the size of the collection, determine if a specific object is contained in the collection, or return an Iterator that can be used to loop through all elements of the collection (more on this later)
- methods include: add(Object o), remove(Object o), contains(Object o), iterator()
- note that removing an element removes an object that compares as equal to a specified object, rather than by position
- Collection extends the Iterable interface, which only requires one method, which supplies an object used to iterate through the collection
- There are a variety of classes that store objects in different ways
    - *Lists* store objects in a specific order
    - *Sets* reject duplicates of any objects already in the collection
    - *Maps* store objects in association with a key, which is later used to look up and retrieve the object (note that if an item with a duplicate key is put into a map, the new item will replace the old item)

The basic distinctions are defined in several interfaces in the java.util package

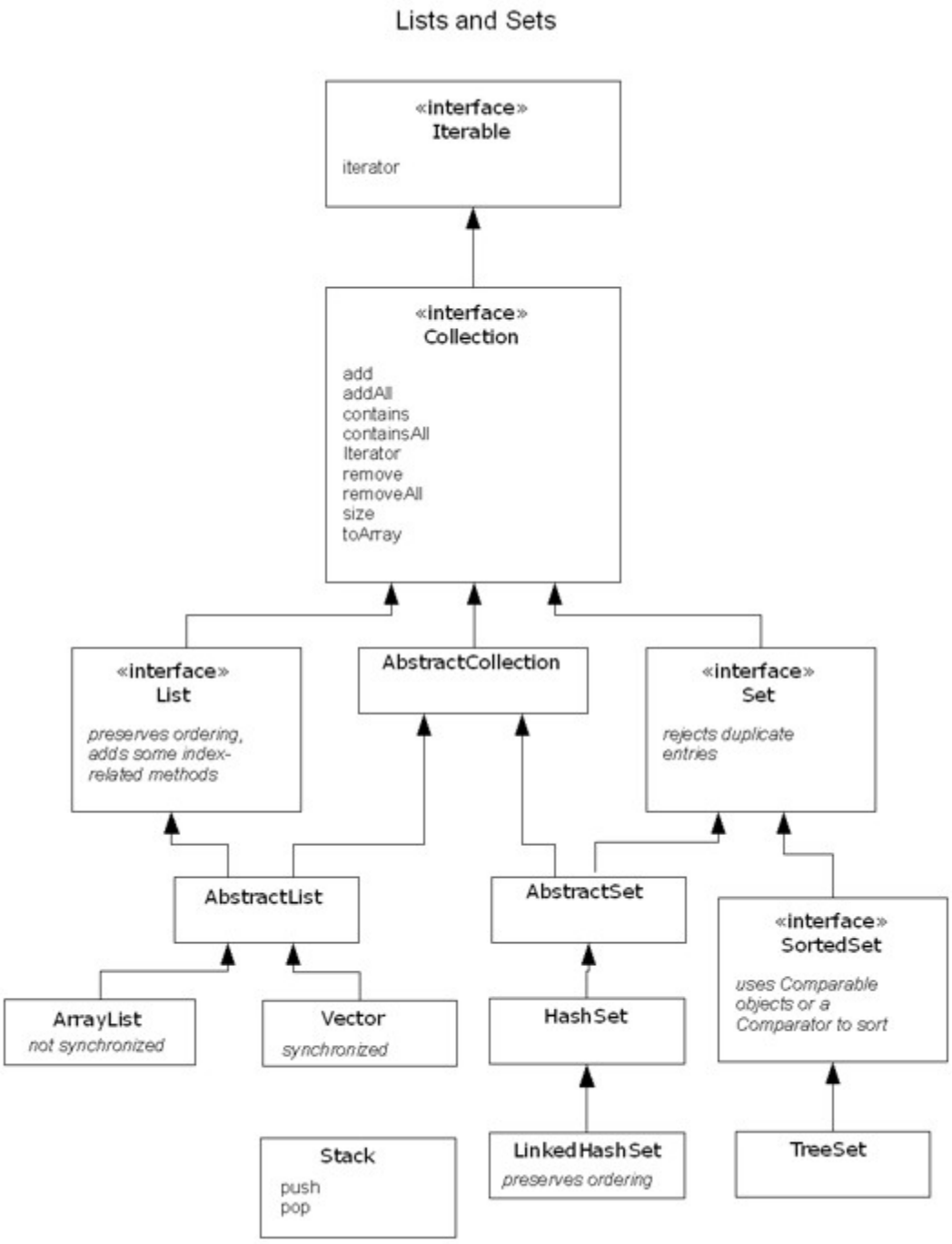
- the *List* interface adds the ability to insert and delete at a specified index within the collection, or retrieve from a specific position
- the *Set* interface expects that implementing classes will modify the adding methods to prevent duplicates, and also that any constructors will prevent duplicates (the add method returns a boolean that states whether the operation succeeded or not; i.e., if the object could be added because it was not already there)

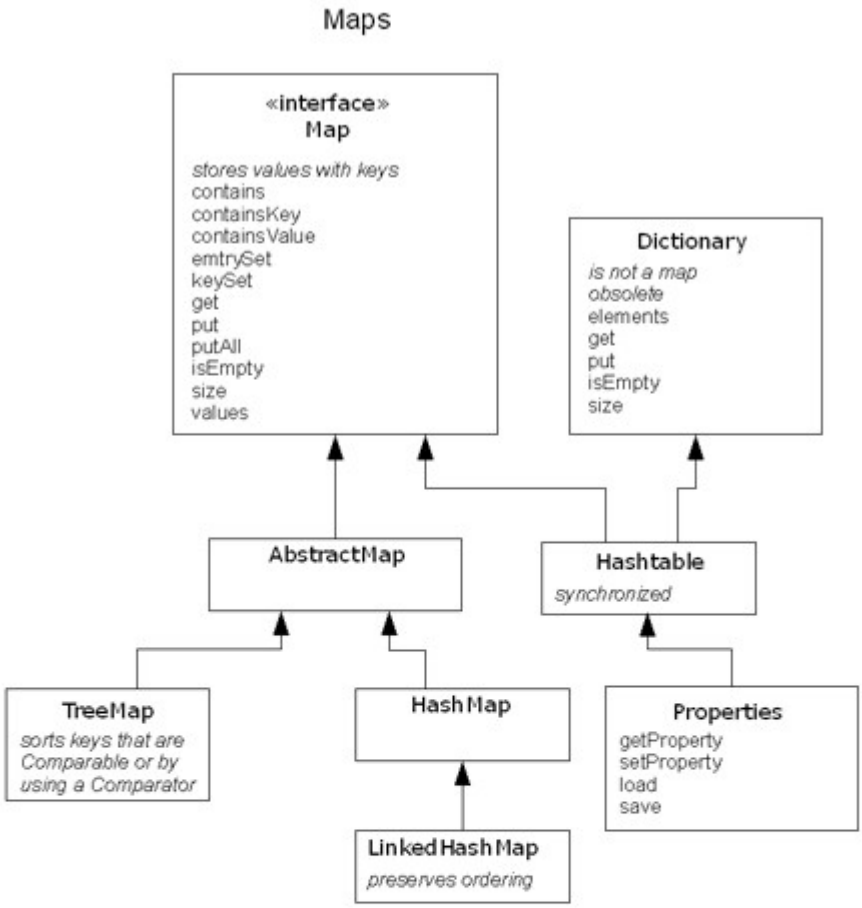
    - sets do not support any indexed retrieval
- the *Map* interface does not extend Collection, because its adding, removing, and retrieving methods use a key value to identify an element
    - methods include: get(Object key), put(Object key, Object value), remove(Object key), containsKey(Objectkey), containsValue(Objectvalue), keySet(), entrySet()
    - maps do not support any indexed retrieval

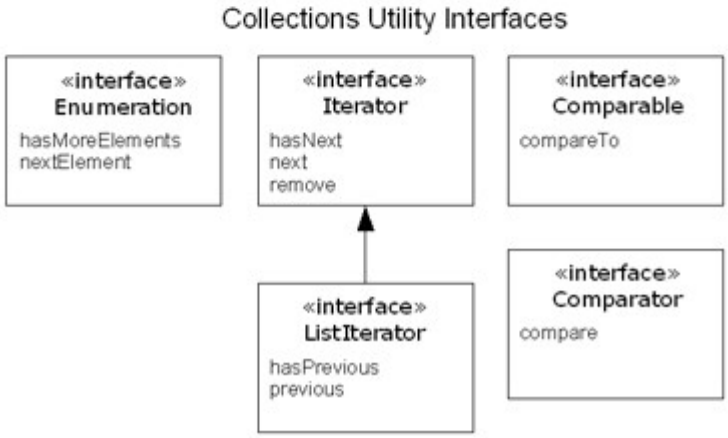
In addition to the List, Set, and Map categories, there are also several inferences you can make from some of the collection class names:

- a name beginning with *Hash* uses a hashing and mapping strategy internally, although the key values usually have no meaning (the hashing approach attempts to provide an efficient and approximately equal lookup time for any element)
- a name beginning with *Linked* uses a linking strategy to preserve the order of insertion, and is optimized for insertion/deletion as opposed to appending or iterating
- a name beginning with *Tree* uses a binary tree to impose an ordering scheme - either the *natural order* of the elements (as specified by the Comparable interface implemented by many classes including String and the numeric wrapper classes), or an order dictated by a special helper class object (a Comparator)

The following diagrams show some of the classes and interfaces that are available


Lists and Sets
«interface»
Iterable
iterator
«interface»
Collection
add
addAll
contains
containsAll
Iterator
remove
removeAll
size
toArray
«interface»
List
preserves ordering, adds some index-related methods
AbstractCollection
«interface»
Set
rejects duplicate entries
AbstractList
AbstractSet
«interface»
SortedSet
uses Comparable objects or a Comparator to sort
ArrayList
not synchronized
Vector
synchronized
HashSet
Stack
push
pop
LinkedHashSet
preserves ordering
TreeSet





Using the Collection Classes

The following are several examples of collection classes

Vector - stores objects in a linear list, in order of addition

- you can insert and delete at any location
- all methods are synchronized, so that the class is *thread-safe*
- predates the collections framework, but was marked to implement List when the collections API was developed
    - as a result, there are many duplicate methods, like add and addElement, since the names chosen for Collection methods didn't all match existing method names in Vector
- used extensively in the past, and therefore you will see it in a lot of code, but most recent code uses ArrayList instead

ArrayList and LinkedList - stores objects in a linear list, in order of addition

There are two general-purpose List implementations in the Collections Framework: ArrayList and LinkedList. Which of the two List implementations you use depends on your specific needs. If you need to support random access, then ArrayList offers the optimal collection. If, however, you need to only access the list elements sequentially, then LinkedList offers the better implementation.

TreeSet - stores objects in a linear sequence, sorted by a comparison, with no duplicates

- stores Comparable items or uses a separate Comparator object to determine ordering
- uses a *balanced tree* approach to manage the ordering and retrieval
- note that there is no tree list collection, since the concepts of insertion order and natural order are incompatible
- since sets reject duplicates, any comparison algorithm should include a guaranteed tiebreaker (for example, to store employees in last name, first name order: to allow for two Joe Smiths, we should include the employee id as the final level of comparison)

Maps

The Map interface is not an extension of the Collection interface. Instead, the interface starts off its own interface hierarchy for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by definition. The interface methods can be broken down into three sets of operations: altering, querying, and providing alternative views. The alteration operations allow you to add and remove key-value pairs from the map. Both the key and value can be null. However, you should not add a Map to itself as a key or value.

- Object put(Object key, Object value)
- Object remove(Object key)
- void putAll(Map mapping)
- void clear()

The query operations allow you to check on the contents of the map:

- Object get(Object key)

- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()

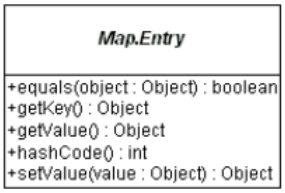
The last set of methods allows you to work with the group of keys or values as a collection.

- public Set keySet()
- public Collection values()
- public Set entrySet()

Because the collection of keys in a map must be unique, you get a Set back. Because the collection of values in a map may not be unique, you get a Collection back. The last method returns a Set of elements that implement the Map.Entry interface.

Map.Entry interface

The entrySet() method of Map returns a collection of objects that implement the Map.Entry interface. Each object in the collection is a specific key-value pair in the underlying Map. Iterating through this collection, you can get the key or value, as well as change the value of each entry.



TreeMap and HashMap

The Collections Framework provides two general-purpose Map implementations: *HashMap* and *TreeMap*. As with all the concrete implementations, which implementation you use depends on your specific needs. For inserting, deleting, and locating elements in a Map, the *HashMap* offers the best alternative. If, however, you need to traverse the keys in a sorted order, then *TreeMap* is your better alternative. Depending upon the size of your collection, it may be faster to add elements to a *HashMap*, then convert the map to a *TreeMap* for sorted key traversal.

Hashtable - stores objects in a Map

- all methods are synchronized, so that the class is thread-safe
- predates the collections framework, but was marked to implement Map when the collections API was developed
    - like Vector, has some duplicate methods
- extends an obsolete class called Dictionary

HashSet - uses hashing strategy to manage a Set

- rejects duplicate entries
- entries are managed by an internal HashMap that uses the entry hashCode values as the keys
- methods are not synchronized

Using the Iterator Interface

Iterators provide a standard way to loop through all items in a collection, regardless of the type of collection

- all collection classes implement an iterator() method that returns the object's iterator (declared as Iterator iterator())
- this method is specified by the Iterable interface, which is the base interface for Collection
- for maps, the collection itself is not Iterable, but the set of keys and the set of entries are
- you can use a for-each loop for any Iterable object

Without an iterator, you could still use an indexed loop to walk through a list using the size() method to find the upper limit, and the get(int index) method to retrieve an item

- but, other types of collections may not have the concept of an index, so an iterator is a better choice

There are two key methods: boolean hasNext(), which returns true until there are no more elements, and Object next(), which retrieves the next element and also moves the iterator's internal pointer to it

The Enumeration class is an older approach to this concept; it has methods hasMoreElements() and nextElement()

Example program 1

```
import java.util.*;
import java.util.Map.Entry;

public class CollectionsTest {

 public static void main(String[] args) {
  List<Integer> l = new ArrayList<Integer>();
  Map<Integer, String> m = new TreeMap<Integer, String>();
  Set<Integer> s = new TreeSet<Integer>();

  l.add(new Integer(1));
  l.add(new Integer(4));
  l.add(new Integer(3));
  l.add(new Integer(2));
  l.add(new Integer(3));
```

```
  m.put(new Integer(1), "A");
  m.put(new Integer(4), "B");
  m.put(new Integer(3), "C");
  m.put(new Integer(2), "D");
  m.put(new Integer(3), "E");

  s.add(new Integer(1));
  s.add(new Integer(4));
  s.add(new Integer(3));
  s.add(new Integer(2));
  s.add(new Integer(3));

  System.out.println("List");
  Iterator<Integer> i = l.iterator();
  while (i.hasNext())
        System.out.println(i.next());

  System.out.println("Map using keys");
  i = m.keySet().iterator();
  int key;
  while (i.hasNext())
  {
        key=i.next();
        System.out.println(m.get(key));
  }

  Iterator<Entry<Integer, String>> j= m.entrySet().iterator();
  System.out.println("Map using entries");

  while (j.hasNext())
        System.out.println(j.next());

  System.out.println("Set");
  i = s.iterator();
  while (i.hasNext())
        System.out.println(i.next());
 }
}
```

This program demonstrates the three types of collections. As is commonly done, the variables are typed as the most basic interfaces (List, Set, and Map).

We attempt to add the same sequence of values to each: 1, 4, 3, 2, and 3

- they are deliberately out of sequence and contain a duplicate
- for the Map, the numbers are the keys, and single-letter strings are used for the associated values

An iterator is then obtained for each and the series printed out (for the Map we try two different approaches: iterating through the keys and retrieving the associated entries, and iterating directly through the set of entries)

Note the order of the values for each, and also which of the duplicates was kept or not.

- the List object stores all the objects and iterates through them in order of addition
- the Map object stores by key; since a TreeMap is used, its iterator returns the elements sorted in order by the keys
- since one key is duplicated, the later of the two values stored under that key is kept
- the Set object rejects the duplicate, so the first item entered is kept (although in this case it would be hard to tell which one was actually stored)

Example program 2

To demonstrate the use of the concrete Set classes, the following program creates a HashSet and adds a group of names, including one name twice. The program then prints out the list of names in the set, demonstrating the duplicate name isn't present. Next, the program treats the set as a TreeSet and displays the list sorted.

```
import java.util.*;

public class SetExample {
public static void main(String args[]) {
      Set set = new HashSet();
      set.add("Bernadine");
      set.add("Elizabeth");
      set.add("Gene");
      set.add("Elizabeth");
      set.add("Clara");
      System.out.println(set);
      Set sortedSet = new TreeSet(set);
      System.out.println(sortedSet);
}
}
```

Running the program produces the following output. Notice that the duplicate entry is only present once, and the second list output is sorted alphabetically.

[Gene, Clara, Bernadine, Elizabeth]
[Bernadine, Clara, Elizabeth, Gene]