Collections

COLLECTIONS

What is the Collections framework?

Collections framework provides two things:

- implementations of common **high-level *data structures***: e.g. Maps, Sets, Lists, etc.
- An organized class hierarchy with rules/ formality for adding. new implementations

Definition of collection

A ***collection*** — **sometimes called a container** — is simply an object that groups multiple elements into a single unit.

Collections are used to store, retrieve, manipulate, and communicate aggregate data.

They typically represent data items that form a natural group, e.g.

- poker hand (**a collection of cards**), a mail folder (a collection of letters), or a telephone directory (a mapping from names to phone numbers).

General comments about data structures

"Containers" for storing data.

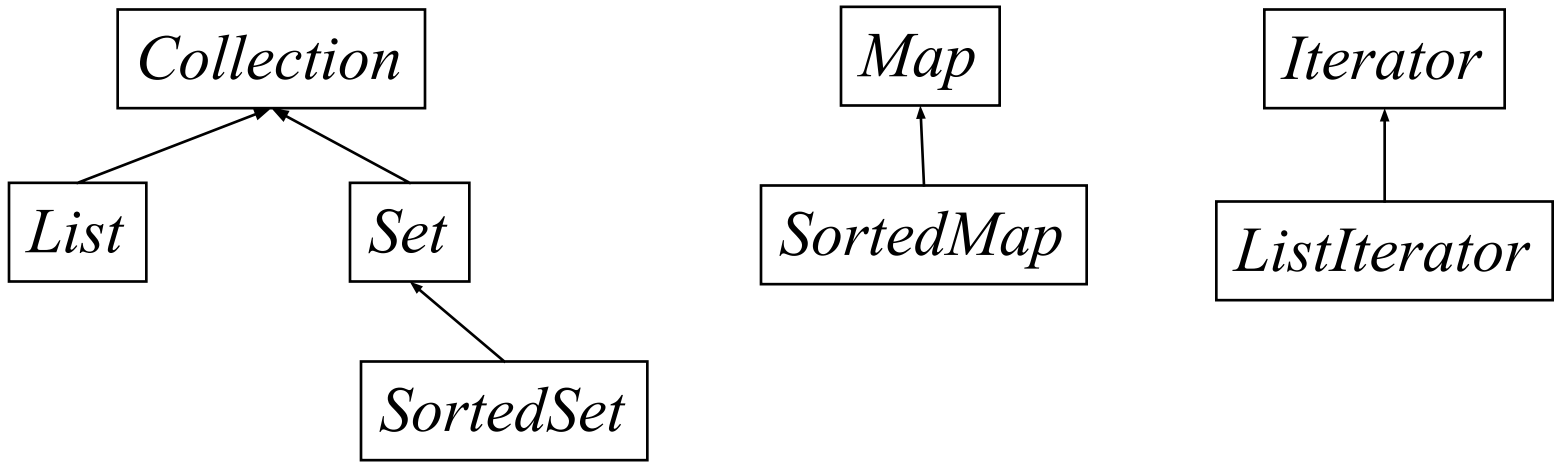
Different data structures provide different abstractions for getting/setting elements of data.

- linked lists
- hashtables
- vectors
- arrays

Same data structures can even be implemented in different ways for performance/memory:

- queue over linked list
- queue over arrays

Collections-related Interface hierarchy



- The *Collection* inteface stores groups of Objects, with duplicates allowed
- The *Set* interface extends *Collection* but forbids duplicates
- The *List* interface extends *Collection*, allows duplicates, and introduces positional indexing.
- *Map* is a separate hierarchy

Collection implementations

Note that Java does not provide any direct implementations of *Collection*.

Rather, concrete implementations are based on other interfaces which extend Collection, such as Set, List, etc.

Still, the most general code will be written using Collection to type variables.

Collection Interface

```
boolean add(Object o);
boolean addAll(Collection c);
void clear();
boolean contains(Object o);
boolean containsAll(Collection c);
boolean equals(Object o);
int hashCode();
boolean isEmpty();
Iterator iterator();
boolean remove(Object o);
boolean removeAll(Collection c);
boolean retainAll(Collection c);
int size();
Object[] toArray();
Object[] toArray(Object[] a);
```

Optional operation, throw UnsupportedOperationException

Comments on Collection methods

Note the iterator() method, which returns an Object which implements the *Iterator* interface.

Iterator objects are used to traverse elements of the collection in their natural order.

Iterator has the following methods:

- boolean hasNext(); // are there any more elements?
- Object next();   // return the next element
- void remove();   // remove the element returned after lest next()

AbstractCollection Class

java.util.*AbstractCollection*

- Abstract class which is partial implementation of of Collection interface

- Implements all methods except iterator() and size()

- Makes it much less work to implement Collections Interface

List interface

An interface that extends the Collections interface.

An ordered collection (also known as a *sequence*).

- The user of this interface has precise control over where in the list each element is inserted.
- The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike *Set*, allows duplicate elements.

Provides a special *Iterator* called *ListIterator* for looping through elements of the List.

Additional Methods in *List* Interface

List extends *Collection* with additional methods for performing index-based operations:

- void add(int index, Object element)
- boolean addAll(int index, Collection collection)
- Object get(int index)
- int indexOf(Object element)
- int lastIndexOf(Object element)
- Object remove(int index)
- Object set(int index, Object element)

List/ListIterator Interface

The List interface also provides for working with a subset of the collection, as well as iterating through the entire list in a position friendly manner:

- ListIterator listIterator()
- ListIterator listIterator(int startIndex)
- List subList(int fromIndex, int toIndex)

ListIterator extends *Iterator* and adds methods for bi-directional traversal as well as adding/removing elements from the underlying collection.

Randomly shuffling a List

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

Concrete List Implementations

There are two concrete implementations of the *List* interface

- LinkedList
- ArrayList

Which is best to use depends on specific needs.

Linked lists tend to be optimal for inserting/removing elements.

ArrayLists are good for traversing elements sequentilly

Note that LinkedList and ArrayList both extend abstract partial implementations of the *List* interface.

LinkedList Class

The LinkedList class offeres a few additional methods for directly manipulating the ends of the list:

- void addFirst(Object)
- void addLast(Object);
- Object getFirst();
- Object getLast();
- Object removeFirst();
- Object removeLast();

These methods make it natural to implement other simpler data structures, like Stacks and Queues.

LinkedList examples

See heavily commented LinkedList Example in course notes

A few things to be aware of:

- it is really bad to use the positional indexing features copiously of LinkedList if you care at all about performance. This is because the LinkedList has no memory and must always traverse the chain from the beginning.
- Elements can be changed both with the List and ListIterator objects. That latter is often more convenient.
- You can create havoc by creating several iterators that you use to mutate the List. There is some protection built-in, but best is to have only one iterator that will actually mutate the list structure.

ArrayList Class

Also supports the List interface, so top-level code can pretty much invisibly use this class or LinkedList (minus a few additional operations in LinkedList).

However, ArrayList is much better for using positional index access methods.

At the same time, ArrayList is much worse at inserting elements.

This behavior follows from how ArrayLists are structured: they are just like Vectors.

More on ArrayList

Additional methods for managing size of underlying array

size, *isEmpty*, *get*, *set*, *iterator*, and *listIterator* methods all run in constant time.

Adding n elements take O[n] time.

Can explicitly grow capacity in anticipation of adding many elements.

Note: legacy Vector class almost identical. Main differences are naming and synchronization.

See short **ArrayList** example.

Vector class

Like an ArrayList, but synchronized for multithreaded programming.

Mainly for backwards-compatibility with old java.

Used also as base class for *Stack* implementation.

Stack class

Stack()
Creates an empty Stack. **Method**

boolean **empty**()
Tests if this stack is empty.

E **peek**()
Looks at the object at the top of this stack without removing it from the stack.

E **pop**()
Removes the object at the top of this stack and returns that object as the value of this function.

E **push**(E item)
Pushes an item onto the top of this stack.

int **search**(Object o)
Returns the 1-based position where an object is on this stack.

```
public class Deal {
  public static void main(String[] args) {
    if (args.length < 2) {
      System.out.println("Usage: Deal hands cards");
      return;
    }
    int numHands = Integer.parseInt(args[0]);
    int cardsPerHand = Integer.parseInt(args[1]);

    // Make a normal 52-card deck.
    String[] suit = new String[] {
        "spades", "hearts", "diamonds", "clubs" };
    String[] rank = new String[] {
        "ace","2","3","4","5","6","7","8",
        "9","10","jack","queen","king" };
    List<String> deck = new ArrayList<String>();
    for (int i = 0; i < suit.length; i++)
      for (int j = 0; j < rank.length; j++)
        deck.add(rank[j] + " of " + suit[i]);

    // Shuffle the deck.
    Collections.shuffle(deck);

    if (numHands * cardsPerHand > deck.size()) {
      System.out.println("Not enough cards.");
      return;}

    for (int i=0; i < numHands; i++)
      System.out.println(dealHand(deck, cardsPerHand));}

  public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;}
}
```

Set Interface

Set also extends *Collection*, but it prohibits duplicate items (this is what defines a Set).

No new methods are introduced; specifically, none for index-based operations (elements of Sets are not ordered).

Concrete Set implementations contain methods that forbid adding two equal Objects.

More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element

Java has two implementations: HashSet, TreeSet

Using Sets to find duplicate elements

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);

        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

HashSets and hash tables

Lists allow for ordered elements, but searching them is very slow.

Can speed up search tremendously if you don’t care about ordering.

Hash tables let you do this. Drawback is that you have no control over how elements are ordered.

hashCode() computes integer (quickly) which corresponds to position in hash table.

Independent of other objects in table.

HashSet Class

Hashing can be used to implement several important data structures.

Simplest of these is HashSet

- add elements with add(Object) method
- contains(Object) is redefined to first look for duplicates.
- if duplicate exists, Object is not added

What determines a duplicate?

- careful here, must redefine both hashCode() and equals(Object)!

HashSet

Look HashSetExample.java

Play around with some additional methods.

Try creating your own classes and override hashCode method.

Do Some timings.

Tree Sets

Another concrete set implementation in Java is TreeSet.

Similar to HashSet, but one advantage:

- While elements are added with no regard for order, they are returned (via iterator) in sorted order.
- What is sorted order?
  - this is defined either by having class implement *Comparable* interface, or passing a *Comparator* object to the TreeSet Constructor.
  - Latter is more flexible: doesn’t lock in specific sorting rule, for example. Collection could be sorted in one place by name, another by age, etc.

Comparable interface

Many java classes already implement this. Try String, Character, Integer, etc.

Your own classes will have to do this explicitly:

- *Comparable* defines the method

  public int compareTo(Object other);
- *Comparator* defines the method

  public int compare(Object a, Object b);

As we discussed before, be aware of the general contracts of these interfaces.

See TreeSetExample.java

Maps

Maps are similar to collections but are actually represented by an entirely different class hierarchy.

Maps store objects by key/value pairs:

- map.add(“1234”, “Andrew”);
- ie Object Andrew is stored by Object key 1234

Keys may not be duplicated

Each key may map to only one value

Java *Map* interface

Methods can be broken down into three groups:

- querying
- altering
- obtaining different views

Fairly similar to Collection methods, but Java designers still thought best to make separate hierarchy – no simple answers here.

Map methods

Here is a list of the Map methods:

- void clear()
- boolean containsKey(Object)
- boolean containsValue(Object)
- Set entrySet()
- boolean get(Object)
- boolean isEmpty()
- Set keySet()
- Object put(Object, Object)
- void putall(Map)
- Object remove(Object)
- int size()
- Collection values()

Map Implementations

We won’t go into too much detail on Maps.

Java provides several common class implementations:

- HashMap
  - a hashtable implementation of a map
  - good for quick searching where order doesn’t matter
  - must override hashCode and equals
- TreeMap
  - A tree implementation of a map
  - Good when natural ordering is required
  - Must be able to define ordering for added elements.