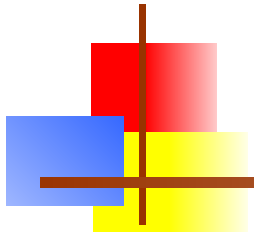
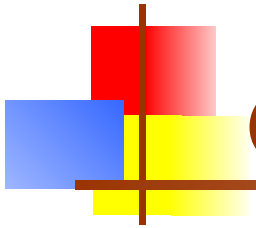
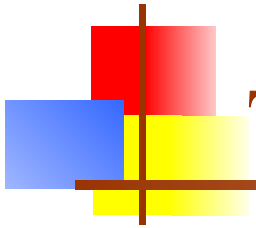


Introduction to Collections



Collections

- A collection is a structured group of objects
- Java 1.2 introduced the Collections Framework
  - Collections are defined in java.util
  - The Collections framework is mostly about *interfaces*
  - There are a number of predefined implementations
- Java 5 introduced generics and "genericized" all the existing collections
  - Vectors have been *redefined* to implement Collection
  - Trees, linked lists, stacks, hash tables, and other classes are implementations of Collection
  - Arrays do *not* implement the Collection interfaces

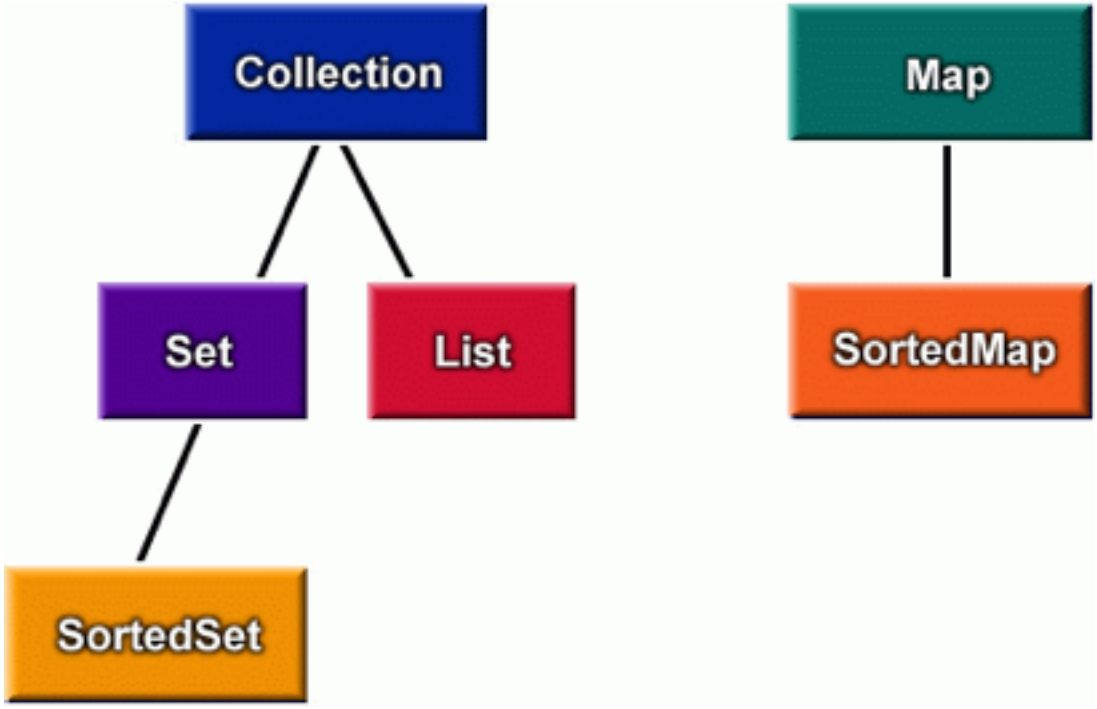


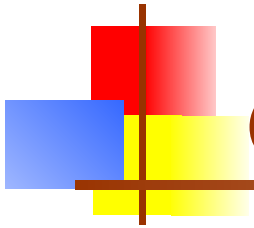
Types of Collection

- Java supplies several types of Collection:
  - Set: cannot contain duplicate elements, order is not important
  - SortedSet: like a Set, but order is important
  - List: may contain duplicate elements, order is important

- Java also supplies some "collection-like" things:
  - Map: a "dictionary" that associates *keys* with values, order is not important
  - SortedMap: like a Map, but order is important

The Collections hierarchy

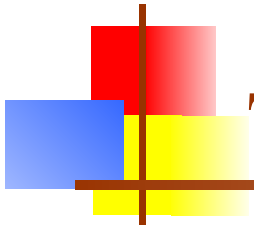




Collections are ADTs

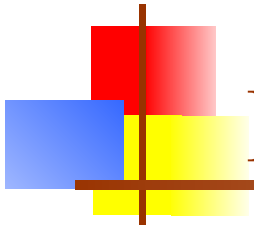
- Here's the *good news* about collections:
  - They are *elegant:* they combine maximum power with maximum simplicity
  - They are *uniform:* when you know how to use one, you almost know how to use them all
  - You can easily convert from one to another

- And the *bad news:*
  - Because there is no special syntax for them (as there is for lists, sets, and dictionaries in Python), you have to work with them using object notation



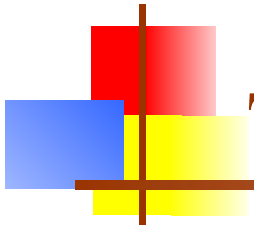
The Collection interface

- Much of the elegance of the Collections Framework arises from the intelligent use of interfaces
- The Collection interface specifies (among many other operations):
  - boolean add(E o)
  - boolean contains(Object o)
  - boolean remove(Object o)
  - boolean isEmpty()
  - int size()
  - Object[] toArray()
  - Iterator<E> iterator()



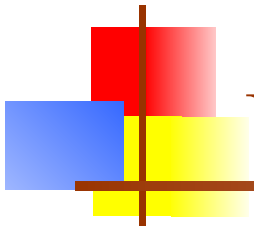
Lists

```
import java.util.*;
public static void main(String args[]) {
    String[] array = {"Phil", "Mary", "Betty", "bob"};
    List<String> myList = Arrays.asList(array);
    Collections.sort(myList);
    System.out.println("Sorted: " + myList);
    int where = Collections.binarySearch(myList, "bob");
    System.out.println("bob is at " + where);
    Collections.shuffle(myList);
    System.out.println("Shuffled: " + myList);
    printAll(myList);
}
```



The Iterator interface

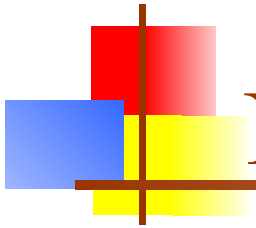
- An iterator is an object that will return the elements of a collection, one at a time
- interface Iterator<E>
  - boolean hasNext()
    - Returns true if the iteration has more elements
  - E next()
    - Returns the next element in the iteration
  - void remove()
    - Removes from the underlying collection the last element returned by the iterator (optional operation)



Using an Iterator

```
static void printAll (Collection coll) {
    Iterator iter = coll.iterator( );
    while (iter.hasNext( )) {
       System.out.println(iter.next( ) );
    }
}
```

- hasNext() just checks if there are any more elements
- next() returns the next element and advances in the collection
- Note that this code is polymorphic—it will work for *any* collection



New for statement

- The syntax of the new statement is

  for(*type var* : *array*) {…}

  or for(*type var* : *collection*) {…}

- Example:

```
for(float x : myRealArray) {
    myRealSum += x;
}
```

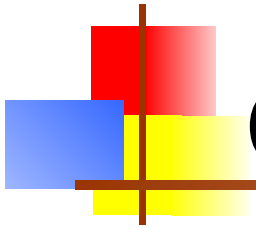
- For a collection class that has an Iterator, instead of

```
for (Iterator iter = c.iterator(); iter.hasNext(); )
    ((TimerTask) iter.next()).cancel();
```

  you can now say

```
for (TimerTask task : c)
    task.cancel();
```

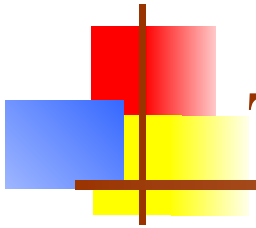
- Note that this for loop is implemented with an Iterator!



ConcurrentModificationException

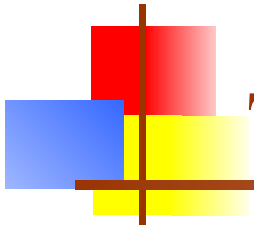
```
static void printAll (Collection coll) {
    Iterator iter = coll.iterator( );
    // When you create an iterator, a "fingerprint"
    // of the collection (list or array) is taken
    while (iter.hasNext( )) {
       System.out.println(iter.next( ) );
       // Both hasNext and next check to make sure
       // the collection hasn't been altered, and will
       // throw a ConcurrentModificationException
       // if it has
    }
}
```

- This means you cannot add or remove elements from the collection within the loop, or any method called from within the loop, *or from some other Thread* that has nothing to do with the loop



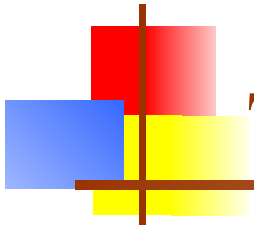
The Set interface

- A set is a collection in which:
  - There are no duplicate elements (according to equals), and
  - Order is not important
- interface Set<E> implements Collection, Iterable
- The methods of Set are exactly the ones in Collection
- The following methods are especially interesting:
  - boolean contains(Object o) // membership test
  - boolean containsAll(Collection<?> c) //subset test
  - boolean addAll(Collection<? extends E> c) // union
  - boolean retainAll(Collection<?> c) // intersection
  - boolean removeAll(Collection<?> c) // difference
- addAll, retainAll, and removeAll return true if the receiving set is changed, and false otherwise



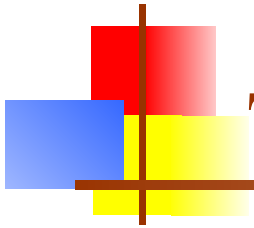
The List interface

- A list is an *ordered* sequence of elements
- interface List<E> extends Collection, Iterable
- Some important List methods are:
  - void add(int index, E element)
  - E remove(int index)
  - boolean remove(Object o)
  - E set(int index, E element)
  - E get(int index)
  - int indexOf(Object o)
  - int lastIndexOf(Object o)
  - ListIterator<E> listIterator()
    - A ListIterator is like an Iterator, but has, in addition, hasPrevious and previous methods



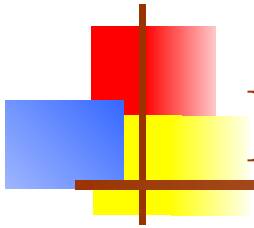
The SortedSet interface

- A SortedSet is a Set for which the order of elements *is* important
- interface SortedSet<E>
  implements Set, Collection, Iterable
- Two of the SortedSet methods are:
  - E first()
  - E last()
- More interestingly, only Comparable elements can be added to a SortedSet, and the set's Iterator will return these in sorted order
- The Comparable interface is covered in a separate lecture



The Map interface

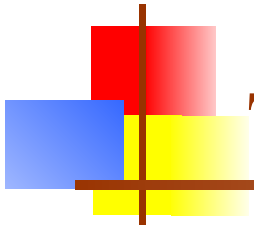
- A map is a data structure for associating keys and values
- Interface Map<K,V>
- The two most important methods are:
  - V put(K key, V value) // adds a key-value pair to the map
  - V get(Object key) // given a key, looks up the associated value
- Some other important methods are:
  - Set<K> keySet()
    - Returns a set view of the keys contained in this map.
  - Collection<V> values()
    - Returns a collection view of the values contained in this map



Dictionary -> HashMap

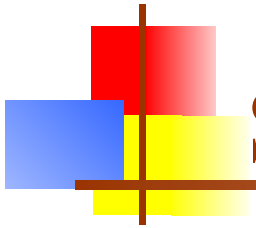
- hash = { ’one’: ’un’, ’two’: ’deux’, ’three’: ’trois’ }
  print ’two ->’, hash[’two’]
  print ’three ->’, hash[’three’]

- ```
  Hashtable<String, String> table = new Hashtable<String, String>();
  table.put("one", "un");
  table.put("two", "deux");
  table.put("three", "trois");
  System.out.println("two -> " + table.get("two"));
  System.out.println("deux -> " + table.get("deux"));
  ```
  ```



The SortedMap interface

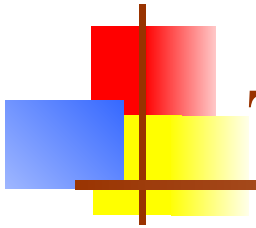
- A sorted map is a map that keeps the *keys* in sorted order
- Interface SortedMap<K,V>
- Two of the SortedMap methods are:
  - K firstKey()
  - K lastKey()
- More interestingly, only Comparable elements can be used as keys in a SortedMap, and the method Set<K> keySet() will return a set of keys whose iterator will return them sorted order
- The Comparable interface is covered in a separate lecture



Some implementations

- class HashSet<E> implements Set
- class TreeSet<E> implements SortedSet
- class ArrayList<E> implements List
- class LinkedList<E> implements List
- class Vector<E> implements List
  - class Stack<E> extends Vector
    - Important methods: push, pop, peek, isEmpty
- class HashMap<K, V> implements Map
- class TreeMap<K, V> implements SortedMap

- All of the above provide a no-argument constructor



The End

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

— Linus Torvalds