CSC143 Java

Overview of Collections

Java Collection Interfaces

- Collection
- Set (extends Collection) – unordered collection of objects with no duplicates
- List (extends Collection) – ordered sequence of objects (first, second, third, …); duplicates allowed
- Map – collection of <key, value> pairs; each key may appear only once in the collection; item lookup is via key values
  (Think of pairs like <word, definition>, <id#, student record>, <book ISBN number, book catalog description>, etc.)
- Iterator – Provides element-by-element access to collection items

New in Java 5.0: Generics

- Before Java 5.0, the static type of the elements of a Collection or of the keys and values of a Map was Object.
  - Usually required a type cast upon removal of an element
  - Allowed any type of reference to be added to a collection, potentially leading to errors
- Beginning with Java 5.0, the static type of the elements may be specified using a type parameter.
  - Allows compiler to check for type conformance
  - Warnings generated if type parameter not included

Java 5.0: Autoboxing and Auto-Unboxing

- Before Java 5.0, a primitive type could not be directly added to a collection. Instead, it needed to be manually 'boxed' using a wrapper class:

```
myIntegerList.add( new Integer( num ) );
```

- In addition, it was necessary to manually extract the primitive value from the wrapper object to use it:

```
int x = ( (Integer)myIntegerList.get( 0 ) ).intValue( ); // ugly!
```

- *Careful:* The distinction between primitive types and reference types is still present.

Examples

```
ArrayList numList = new ArrayList( );
int num = 2;
// numList.add( num ); // syntax error
numList.add( new Integer( num ) );
int x =
    ( (Integer)numList.get( 0 ) ).intValue( );

ArrayList stringList = new ArrayList( );
stringList.add( "Hello" );
stringList.add( Color.RED ); // legal –
                    /* potential bug */
```

```
ArrayList<Integer> numList =
                    new ArrayList<Integer>( );
int num = 2;
numList.add( num ); // autoboxing
int x = numList.get( 0 ); // auto-unboxing


ArrayList<String> stringList =
                    new ArrayList<String>( );
stringList.add( "Hello" );
// stringList.add( Color.RED );  // syntax error --
                    /* compiler helps catch bugs */
```


interface *Collection<E>*

Basic methods available on most collections

- **int size( )** – # of items currently in the collection
- **boolean isEmpty( )** – (size( ) == 0)
- **boolean contains(Object o)** – true if o.equals(element) for an element in the collection
- **boolean add(E o)** – ensure that o is in the collection by adding it if needed; return true if collection altered; false if not.
- **boolean remove(Object o)** – remove a single instance of o from the collection; return true if something was actually removed
- **void clear( )** – remove all elements
- **Iterator<E> iterator( )** – return an iterator object for this collection

E is the *type* parameter

interface *Iterator<E>*

Provides access to elements of any collection one-by-one, even if the collection has no natural ordering (sets, maps)

boolean hasNext( ) – true if the iteration has more elements

E next( ) – next element in the iteration; precondition: hasNext( ) == true

void remove( ) – remove from the underlying collection the element last returned by the iteration. Preconditions: next( ) has been called and remove( ) has not been called since the previous call to next( ). [Optional]

What is standard loop pattern to use an Iterator?

interface *List<E>* extends *Collection<E>*

Includes all Collection methods, plus ones that make sense if the collection is ordered. Following is a subset…

E get(int pos) – return element at position pos
E set(int pos, E elem) – store elem at position pos, returns value previously stored.
void add(int pos, E elem) – store elem at position pos; slide elements at position pos to size( )-1 up one position
E remove(int pos) – remove item at given position; shift remaining elements down one position to fill the gap. returns Object removed
int indexOf(Object o) – return position of first occurrence of o in the list, or -1 if not found

Precondition for most of these is 0 <= pos < size( )

interface *Set<E>* extends *Collection<E>*

- As in math, a Set is an unordered collection, no duplicates
  - attempting to add an element already in the set does not change the set
  - Interface is same as Collection, but refines the specifications via comments
- interface *SortedSet<E>* extends *Set<E>*
  - Same as Set<E>, but iterators always return set elements in order
  - Requires that elements be *Comparable<T>*:
    implement the compareTo(T o) method, returning a negative, 0, or positive number to mean <=, ==, or >=, respectively

Interface *Map<k, v>*

- Collections of <key, value> pairs
  - Keys are unique. Values need not be.
- Does not extend Collection because its contract is different in important ways, but does provide similar methods
  size( ), isEmpty( ), clear( )
- Basic methods for dealing with <key, value> pairs
  V put(K key, V value) – add <key, value> to the map, replacing the previous <key, value> mapping if one exists
  V get(Object key) – return the value associated with the given key, or null if key is not present
  V remove(Object key) – remove any mapping for the given key
  boolean containsKey(Object key) – true if key appears in a <key, value> pair
  boolean containsValue(Object value) – true if value appears in a <key, value>

Maps and Iteration

Map provides methods to view contents of a map as a collection:

Set<K> **keySet**( ) – return a Set whose elements are the keys of this map
Collection<V> **values**( ) – return a Collection whose elements are the values contained in this map
[Why is one a set and the other a collection?]

Typical operation:

```
Map<K, V> map = …;
Set<K> keys = map.keySet( );
Iterator<K> iter = keys.iterator( );
while (iter.hasNext( )) {
    K key = iter.next( );
    V value = map.get(key);
    …  // do something with key and value
}
```


Preview of Coming Attractions

1. Study ways to implement these interfaces
   - Array-based vs. link-list-based vs. hash-table-based vs. tree-based
2. Compare implementations
   - What does it mean to say one implementation is "faster" than another?
   - Basic complexity theory – O( ) notation
3. Correctly apply these and other data structures in our programming