Collection Interfaces

- **Collection** – a collection of objects
- **List** extends Collection – ordered sequence of objects (first, second, third, …); duplicates allowed
- **Set** extends Collection – unordered collection of objects; duplicates suppressed
- **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

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
  - Yields unattractive code
    ```java
    ArrayList a = new ArrayList();
    a.add("First element"); // a string
    String s = (String) a.get(0); // need a cast
    ```
- In Java 5.0, a Collection (or a Map) specifies the static type of its elements
  - Better code
    ```java
    ArrayList<String> a = new ArrayList<String>();
    a.add("First element");
    String s = a.get(0); // no cast
    ```
- With Generics, the compiler can do some type checking.
  ```java
  ArrayList<String> a = new ArrayList<String>();
  a.add(new Oval()); // doesn't compile
  ```

Goals for Next Several Lectures

- Survey different kinds of collections, focusing on their interfaces
  - Lists, sets, maps
  - Iterators over collections
- Then look at different possible implementations
  - Arrays, linked lists, hash tables, trees
  - Mix-and-match implementations to interfaces
- Compare implementations for efficiency
  - How do we measure efficiency?
  - Implementation tradeoffs

Java 2 Collection Implementations

- Main concrete implementations of these interfaces:
  - **ArrayList** implements List (using arrays underneath)
  - **LinkedList** implements List (using linked lists)
  - **HashSet** implements Set (using hash tables)
  - **TreeSet** implements Set (using trees)
  - **HashMap** implements Map (using hash tables)
  - **TreeMap** implements Map (using trees)
Java 5.0: Boxing

• Before Java 5.0, a primitive type could not be put directly in a collection
  
  ```java
  ArrayList a = new ArrayList();
  int i = 3;
  a.add(i);// NO!
  a.add(new Integer(i)); // OK
  ```

• The distinction between primitive and reference types is still present in Java 5.0. But, the details are hidden from the programmer.
  
  ```java
  ArrayList<Integer> a = new ArrayList<Integer>();
  int i = 3;
  a.add(i);// OK: i is boxed into an Integer object
  int k = a.get(0); // OK: the arraylist element is unboxed
  ```

interface Collection<E>

• Basic methods available on most collections (E is the generic type of the collection):
  
  ```java
  int size() – # of items currently in the collection
  boolean isEmpty() – (size() == 0)
  boolean contains(Object o) – true if o is in the collection
  [how to compare o with the elements already in the collection?]
  boolean add(E e) – ensure that e is in the collection, possibly adding it;
  return true if collection altered; false if not. [leaves a lot unspecified...]
  boolean addAll(Collection<E> other) – add all elements in the other collection
  (actually not the exact signature...)
  boolean remove(Object o) – remove one o from the collection, if present;
  return true if something was actually removed
  void clear() – remove all elements
  Iterator<E> iterator() – return an iterator object for this collection
  ```

interface Iterator<E>

• Provides access to elements of any collection one by one, even if the collection has no natural ordering (sets, maps)
  
  ```java
  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. [Optional: some collections don’t support this.]
  ```

Iterators vs. Counter Loops

• A related pattern is the counting loop:
  
  ```java
  ArrayList<E> list = …;
  for(int i = 0; i < list.size(); i++) {
    E elem = list.get(i);
    // do something with elem
  }
  ```

• The iterator pattern is generally preferable because it...
  • works over any collection, even those without a get(int) operation
  • encapsulates the tedious details of iterating, indexing

Still more abstraction: for( : )

• Can even use an iterator without asking for one
  
  ```java
  ArrayList<E> list = …;
  for(E elem : list) {
    // do something with elem
  }
  ```

• CSC143 style rule: use the iterator pattern (with an actual iterator or in the form of the above for loop). It is a good illustration of the concept of abstraction.
  • Unless there are compelling reasons to use a counting loop (e.g. initialization)
  • Note: the for( : ) statement works for arrays as well (anything that is Iterable)
  ```java
  int a = new int[10];
  for(int k : a) { // do something with k
  ```
Lists as Collections

• In some collections, there is no natural order
  • Leaves on a tree, grocery items in a bag, grains of sand on the beach
• In other collections, the order of elements is natural and important
  • Chapters of a book, floors in a building, people camping out to buy Star Wars tickets
• Lists are collections where the elements have an order
  • Each element has a definite position (first, second, third, …)
  • positions are generally numbered from 0

interface List<E> extends Collection<E>

• Following are included in all Java Lists (and some other Collection types):
  E get(int pos) – return element at position pos
  boolean add(int pos, E elem) – store elem at position pos
  boolean add(int pos, E elem) – store elem at position pos; slide elements
  at position pos to size() - 1 up one position to the right
  E remove(int pos) – remove item at given position; shift remaining
  elements to the left to fill the gap; return the removed element
  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 ListIterator<E> extends Iterator<E>

• The iterator( ) method for a List returns an instance of ListIterator
• Can also send listIterator(int pos) to get a ListIterator starting at the
given position in the list
• ListIterator returns objects in the list collection in the order
  they appear in the collection
• Supports additional methods:
  hasPrevious( ), previous( ) – for iterating backwards through a list
  set( E e) – to replace the current element with something else
  add( E e) – to insert an element after the current element

List Implementations

• ArrayList<E> – internal data structure is an array
  • Fast iterating
  • Fast access to individual elements (using get(int), set(int, E))
  • Slow add/remove, particularly in the middle of the list
• LinkedList<E> – internal data structure is a linked list
  • Fast iterating
  • Slow access to individual elements (using get(int), set(int, E))
  • Fast add/remove, even in the middle of the list if via iterator
• A bit later in the course we’ll dissect both forms of implementation

interface Set<E> extends Collection<E>

• As in math, a Set is an unordered collection, with no
duplicate elements
• attempting to add an element already in the set does not change the set
• Interface is same as Collection, but refines the specifications
  • The specs are in the form of comments
• interface SortedSet<E> extends Set<E>
• Same as Set, but iterators will always return set elements in a
specified order
• Requires that elements be Comparable: implement the
compareTo(  ) method, returning a negative, 0, or positive
number to mean <, =, or >, respectively

interface SortedSet<E> extends Set<E>

• Collections of <key, value> pairs
  • keys are unique, but values need not be
  • Doesn’t extend Collection, but does provide similar methods
  • 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
    void putAll(Map<K, V> other) – put all <key, value> pairs from other into this map
    V get(K key) – return the value associated with the given key, or null
    if key is not present
    V remove(K 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?]
- To iterate through the keys or values or both, grab one of these collections, and then iterate through that
  ```java
  Map<K, V> map = ...;
  Set<K> keys = map.keySet();
  for (K key : keys) {
    V value = map.get(key);
    // do something with key and value
  }
  ```

interface SortedMap<K, V> extends Map

- SortedMap can be used for maps where we want to store key/value pairs in order of their keys
  - Requires keys to be Comparable, using compareTo, or comparable with a Comparator.
  - Sorting affects the order in which keys and values are iterated through
    - `keySet()` returns a `SortedSet<K>`
    - `values()` returns an ordered `Collection<V>`

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. Use these and other data structures in our programming