The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
8 March 2003

Notes – Collections

I. Libraries
   a. Java has lots of common libraries
   b. Someone can spend a lot of time coding a decent library and then everyone can use it.
   c. Don't try to learn the source code, just learn the specification and accept it.
   d. Many built-in libraries will be impossible to study at the source code anyway.

II. Collections
   a. Collection library is in java.util
   b. 12 concrete classes, 9 interfaces, and 5 abstract classes are included.
   c. The pre Java 1.2 version has a smaller set of classes.
   d. A collection is no more complicated than a container to hold things.
   e. Behaviors
      i. add
      ii. remove
      iii. contains (is this in here?)
      iv. size
      v. members (Iterator)
   f. Iterator
      i. for each object in the collection, do something
      ii. foreach is the key concept
   g. Why have so many classes to implement such a simple concept?
      i. Many variations on the theme.
      ii. Is the collection ordered or unordered?
      iii. How efficient is some particular type of access?
      iv. Can elements be accessed with a key?
      v. Are duplicates allowed?

III. Four Key Interfaces
   a. Define the basics of collections
   b. Collection
      i. Some of its behaviors are important; some are rarely used.
      ii. See slide CS-100-20-11
      iii. Behaviors
         1. `add(Object)`
         2. `addAll(Collection)d`
         3. `clear()`
         4. `contains(Object)`, `containsAll(Collection)` (to contain all, it must have exactly the same type and same elements as the collection given
         5. `remove(Object)`, `removeAll(Collection)`
         6. `retainAll(Collection)` removes everything except those in the given collection
         7. `toArray()` Not really as useful as might be expected
      iv. Many other (non-Java) libraries implement something similar, often called a Bag.
   c. List
      i. Ordered
      ii. Implements all the behaviors of a collection, plus some of its own.
      iii. Behaviors
         1. `add(int, Object)`
         2. `get(int)`, `set(int, Object)`
         3. `add(Object)` goes at the end.
         4. `addAll(int, Object)` inserts all in the middle.
         5. `indexOf(Object)`, `lastIndexOf(Object)`

6. `subList(int, int)d`
    a. Get a subset of the list as another list.
    b. Includes the first number.
    c. DOES NOT include the last number.
    d. subList(1, 2) returns just the first element

iv. Implementations
1. Vector
    a. Most widely used
    b. Efficient random access
2. LinkedList
    a. Doubly-linked list
    b. Random access is expensive

d. Set
i. No duplicates allowed
ii. Another subclass of Collection.
iii. Only difference is that duplicate elements are ignored. No arrors, just silently discarded.
iv. Implementations
1. HashSet
    a. Takes the same amount of time for each operation, no matter how long the list gets.
    b. Total time to traverse the list increases linearly since each next() takes the same amount of time.
2. LinkedHashSet. Keeps the same order over time.
3. TreeSet Keeps a sorted order.

v.

e. Map
i. Defines mapping from one class to another.
ii. Essentially stores key-value pairs for any data
iii. Keys form a Set (no duplicates allowed)
iv. Values form a Collection
v. Behaviors
1. `put(Object key, Object value)`
2. `get(Object key)`
3. `containsKey(Object)`
    a. Very common.
    b. Can be used before inserting to make sure no duplicate keys are attempted.
4. `containsValue(Object)`
    a. Much less common.
    b. The point of a map is to lookup objects by their keys.
5. Others
    a. Can get all the keys or all the key-value pairs as `Sets`
    b. Can get a `Collection` of all values.

vi. Implementations
1. HashMap.
    a. Similar to HashSet
    b. Constant-time operations
2. LinkedHashMap
    a. Maintain the order
    b. Usually it's in the order of insertion
    c. Can change it so it stores in the order last accessed
    d. Can derive a subclass that automatically removes oldest elements

i. Use this in combination with the ability to store in the order last accessed.
ii. Then you can keep track of only the newest "things" in some collection.

f. Iterators
   i. We've used these before, so there shouldn't be any surprises.
   ii. Behaviors
      1. `hasNext()`
      2. `next()`
      3. `remove()`
   iii. Only instantiate an iterator if you're creating your own collection class.
   iv. Otherwise you'd ask the collection to return an appropriate iterator.
   v. ListIterator implementation
      1. Adds order
      2. `add(Object)` inserts just before the current position
      3. `set(Object)` replaces the previous element (makes sense if you've just used next() to get the old value)
      4. `previous()`, `hasPrevious()`
      5. `nextIndex()`, `previousIndex()`

IV. Abstractions
   a. foreach x in collection
      i. Iterator provides this abstraction
      ii. Many variations on the theme
   b. Insert / Remove / Change
      i. Improve upon the basics by adding manipulation of the data.
      ii. Collection and its derivations implement this.
   c. All implementations of these essentially stem from the five basic interfaces.

V. Aggregates
   a. Explicit Aggregates
      i. The program specifically takes action to insert values.
      ii. All collections fit this description
   b. Predicate-Defined Aggregates
      i. Ask for the set of _____
      ii. Eg: "The collection of all people named Smith"
      iii. SQL queries fit this description.
      iv. Still uses the foreach abstraction
      v. Must have an underlying explicit aggregate at some bottom level from which the subset can be pulled.
      vi. No inherent support for this in Java, so use an Iterator like EventDateIterator (see code for assignment 2)
   c. Iterators provide the basic aggregation abstraction
      i. If you don't care about the data's source (explicit, predicate-defined), take Iterator as your argument.
      ii. If you want to modify the data, or get size(), membership(), etc, use a Collection.
      iii. Think carefully about this choice when you're writing the code.
   d. Implementing iterators for predicate-defined aggregates
      i. Such iterators will usually be built on top of other iterators.
      ii. Look through the underlying iterator for the next appropriate element; store and return that.

VI. Iterator Example
   a. Write an iterator to walk an array
   b. int _index = 0; Object[] _values;
   c. Invariant: _index >= 0 && _index <= _values.length() && _values != null
   d. Abstraction Function

i. _index < _values.length() means there's more values to find.
ii. _index == _values.length() means there aren't any more values.
iii. Abstraction function can depend on the rep invariant.

e. hasNext()
i. return (_index < _values.length());

f. next()
i. if (!hasNext()) return _values [_index++];
ii. throw new NoSuchElementException();

g. remove() just throws UnsupportedOperationException()

VII. Changing the Collection While Iterating

a. What happens to the iterator if the underlying collection is changed?
b. If an element is added
i. The behavior is undefined!
ii. This can create mild problems.
c. If an element is removed
i. This can get nasty.
ii. You could get output of the element you removed if the iterator has already pulled it out.
iii. You could start throwing exceptions that aren't meant to be thrown.
d. Cure
i. Use the operations on the iterator, not on the collection
ii. remove() on Iterator
iii. set(), add() on ListIterator
iv. Obviously the ability to do this depends on having the right type of iterator for the job.
e. Fail-Fast
i. Will be guaranteed to fail.
ii. That means there's a predictable behavior if something goes wrong.
iii. All standard Collection classes are fail-fast.
iv. Sun has said "don't absolutely rely on the exception, since fringe cases may slip through."

VIII. Summary

a. There are only two fundamental abstractions
b. Everything else is variations on the theme.
c. Iterator provides foreach
d. Collection adds insert / remove
e. List adds ordering
f. Set adds duplicate checking
g. Map adds keys
h. Remember those five.
i. Always write code in terms of those five! Only use a subclass next to the word 'new', so that the rest of the code doesn't care.