
CSE 143 Java

Collections

Reading: Ch. 12 (mostly review), Ch. 21

7/26/2004

(c) 2001-4, University of Washington

13-1

Collections

- Most programs need to store and access collections of data
- Collections are worth studying because...
 - They are widely useful in programming
 - They provide examples of the OO approach to design and implementation
 - identify common pattern
 - regularize interface to increase commonality
 - factor them out into common interfaces, abstract classes
 - Their implementation will raise issues previously swept under the rug, particularly efficiency

7/26/2004

(c) 2001-4, University of Washington

13-2

Goals for Upcoming 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
- Along the way, point out useful applications

7/26/2004

(c) 2001-4, University of Washington

13-3

Collections in Java

- Java has a variety of facilities to support programming with collections
 - Most are part of package `java.util`
 - Most were introduced in Java 1.2
 - Collectively referred to as the "Collections Framework"
- Lots of this you have already used!
- Primary components
 1. A set of interfaces for common container types
 2. Concrete implementations of these containers
 3. Interfaces for support operations
 4. Static utility methods
 5. Wrappers and adapters (won't be covered)

7/26/2004

(c) 2001-4, University of Washington

13-4

Where You're Coming From...

We've already used these Collections features extensively

- **ArrayList**
 - Indexed collection of Objects
 - No limit on size
 - Methods:
- **Iterator**
 - Sequence through an ArrayList
 - Methods:
- **Comparable and Comparator**
- **Sort methods**
 - Arrays.sort, Collections.sort

7/26/2004

(c) 2001-4, University of Washington

13-5

Interfaces

• Key interfaces in Java 1.2 and later:

- **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.)
*technically not an extension of Collection, but interface is generally similar
- **Iterator** – provides element-by-element access to items in a collection

7/26/2004

(c) 2001-4, University of Washington

13-6

Java 2 Collection Implementations

• Main concrete implementations of these interfaces:

- **ArrayList** implements List (using arrays)
- **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)

7/26/2004

(c) 2001-4, University of Washington

13-7

Footnote: Pre-Java 2 Collections

• Java 1.0 and 1.1 had different collection classes

- still retained because they are used in existing (old) code

• Correspondence of some classes and interfaces:

Java 1.2	Java 1.0, 1.1
ArrayList	Vector
Map	Dictionary
HashMap	HashTable
Iterator	Enumeration

- Newer classes generally lighter weight, more efficient, but very similar interfaces
- *Use the new classes only* unless you have a specific reason to use the old ones

7/26/2004

(c) 2001-4, University of Washington

13-8

interface Collection

- 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` is in the collection
[how to compare `o` with the elements already in the collection?]
`boolean add(Object o)` – ensure that `o` is in the collection, possibly adding it;
return true if collection altered; false if not. [leaves a lot unspecified...]
`boolean addAll(Collection other)` – add all elements in the other collection
`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 iterator()` – return an iterator object for this collection

- Note: much richer interface than an array

7/26/2004

(c) 2001-4, University of Washington

13-9

interface Iterator

- Provides access to elements of any collection one-by-one, even if the collection has no natural ordering (sets, maps are not ordered)

- Interface

`boolean hasNext()` – true if the iteration has more elements
`Object 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.]

7/26/2004

(c) 2001-4, University of Washington

13-10

Standard Iterator Loop Pattern

```
Collection c = ...;
Iterator iter = c.iterator();
while (iter.hasNext()) {
    Object elem = iter.next();
    ... // do something with elem
}
```

- Note similarity to generic file/stream processing loop:

```
open stream -- perhaps from file
while not at end of stream {
    read/write next data item, do something with it
}
```

7/26/2004

(c) 2001-4, University of Washington

13-11

Iterators vs. Counter Loops

- A related pattern is the *counting loop*:

```
ArrayList list = ...;
for (int i = 0; i < list.size(); i++) {
    Object elem = list.get(i);
    ... // do something with elem
}
```

- The iterator pattern is generally preferable because it...

- works for any collection, even those without a `get(int)` operation
- encapsulates the tedious details of iterating, indexing
- is efficient – `get()` is not fast in some collections, even if available

- CSE143 style rule: use iterator pattern

- Unless there are compelling reasons to use a counting loop

7/26/2004

(c) 2001-4, University of Washington

13-12

Collection Contents: Objects

- All Java Collections store Objects
- Cannot store primitive types directly
 - Use wrapper classes if needed
- Values returned from Collections must be cast back to a type

```
Integer age = new Integer(21);
ArrayList ageList = new ArrayList();
ageList.add(0, age);
Integer ageAgain = ageList.get(0); // type error!
Object ageAgain = ageList.get(0); // correct – but not always useful!
Integer ageAgain = (Integer) ageList.get(0); // correct and useful
```
- Contrast: Arrays are declared with a single, specific element type
 - Could be any type: Object, primitive type, interface, abstract class, concrete class, another array, etc.

7/26/2004

(c) 2001-4, University of Washington

13-13

Lists as Collections

- In some collections, there is no natural order
 - Toys in a toybox, 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

7/26/2004

(c) 2001-4, University of Washington

13-14

interface List extends Collection

- Following are included in all Java Lists (and some other Collection types):
 - Object **get**(int pos) – return element at position pos
 - boolean **set**(int pos, Object elem) – store elem at position pos
 - boolean **add**(int pos, Object elem) – store elem at position pos; slide elements at position pos to size()-1 up one position to the right
 - Object **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 $0 \leq \text{pos} < \text{size}()$

7/26/2004

(c) 2001-4, University of Washington

13-15

interface ListIterator extends Iterator

- The **iterator()** method for a List actually returns an instance of ListIterator (extends Iterator)
 - Can also use **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(Object o)** – to replace the current element with something else
 - add(Object o)** – to insert an element after the current element

7/26/2004

(c) 2001-4, University of Washington

13-16

List Implementations

- **ArrayList** – internal data structure is an array
 - Fast iterating
 - Fast access to individual elements (`get(int)`, `set(int, Object)`)
 - Slow add/remove except at the end of the list
- **LinkedList** – internal data structure is a linked list
 - Fast iterating
 - Slow access to individual elements (`get(int)`, `set(int, Object)`)
 - Fast add/remove, even in the middle of the list
- We'll dissect both forms of implementation shortly

7/26/2004

(c) 2001-4, University of Washington

13-17

interface **Set** extends Collection

- 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** extends Set
 - Same as Set, but iterators always return set elements in order
 - Requires that elements be Comparable: implement the `compareTo(Object)` method, returning a negative, 0, or positive number to mean `<=`, `==`, or `>=`, respectively

7/26/2004

(c) 2001-4, University of Washington

13-18

interface **Map**

- Collections of <key, value> pairs
 - keys are unique, but values need not be
- Doesn't extend Collection, but does provide similar methods `size()`, `isEmpty()`, `clear()`
- Basic methods for dealing with <key, value> pairs:
 - Object `put(Object key, Object value)` – add <key, value> to the map, replacing the previous <key, value> mapping if one exists
 - void `putAll(Map other)` – put all <key, value> pairs from other into this map
 - Object `get(Object key)` – return the value associated with the given key, or null if key is not present
 - Object `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>

7/26/2004

(c) 2001-4, University of Washington

13-19

Maps and Iteration

- Map provides methods to view contents of a map as a collection:
 - Set `keySet()` – return a **Set** whose elements are the keys of this map
 - Collection `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

```
Map map = ...;
Set keys = map.keySet();
Iterator iter = keys.iterator();
while (iter.hasNext()) {
    Object key = iter.next();
    Object value = map.get(key);
    ... // do something with key and value
}
```

7/26/2004

(c) 2001-4, University of Washington

13-20

interface **SortedMap** 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
- Sorting affects the order in which keys and values are iterated through
 - keySet() returns a **SortedSet**

7/26/2004

(c) 2001-4, University of Washington

13-21

Interoperating

- It is relatively easy to shift between various collections
- One collection to another:
 - addAll(Collection)
- Arrays to Lists:
 - static List Arrays.asList(array)
- Lists to Arrays:
 - Object[] aList.toArray(List)

7/26/2004

(c) 2001-4, University of Washington

13-22

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

7/26/2004

(c) 2001-4, University of Washington

13-23