
CSE 143 Java

Collections

2/5/2008

(c) University of Washington

11-1

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 patterns
 - regularize interface to increase commonality
 - factor them out into common interfaces, abstract classes
 - Their implementation will raise issues previously swept under the rug: efficiency

2/5/2008

(c) University of Washington

11-2

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

2/5/2008

(c) University of Washington

11-3

Java 2 Collection Interfaces

- Key interfaces in Java 1.2 and above:
 - **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

2/5/2008

(c) University of Washington

11-4

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)

2/5/2008

(c) University of Washington

11-5

Footnote: Pre-Java 2 Collections

- Java 1.0 and 1.1 had different collection classes
 - still retained because they are widely used in existing 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
- *New programs should use the new classes only*

2/5/2008

(c) University of Washington

11-6

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

2/5/2008

(c) University of Washington

11-7

interface Iterator

- 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
 - 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.]
- Standard usage pattern:

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

2/5/2008

(c) University of Washington

11-8

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
}

2/5/2008

(c) University of Washington

11-9

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 over any collection, even those without a `get(int)` operation
 - encapsulates the tedious details of iterating, indexing
- CSE143 style rule: use iterator pattern
 - Unless there are compelling reasons to use a counting loop

2/5/2008

(c) University of Washington

11-10

Collection Contents: Objects

- All Java Collections store Objects
 - cannot specify a particular type of object
 - cannot store primitive types
- 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.
- Values returned from Collections must be cast to their specific subtype

```
Integer age = new Integer(21);
ArrayList ageList = new ArrayList();
ageList.add(0, age);
Integer ageAgain = ageList.get(0); //syntax error!
```

2/5/2008

(c) University of Washington

11-11

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 *Starwars* 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

2/5/2008

(c) University of Washington

11-12

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 $0 \leq \text{pos} < \text{size}()$

2/5/2008

(c) University of Washington

11-13

interface **ListIterator** extends Iterator

- 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**(Object o) – to replace the current element with something else
 - add**(Object o) – to insert an element after the current element

2/5/2008

(c) University of Washington

11-14

List Implementations

- **ArrayList** – internal data structure is an array
 - Fast iterating
 - Fast access to individual elements (using **get**(int), **set**(int, Object))
 - Slow add/remove, particularly in the middle of the list
- **LinkedList** – internal data structure is a linked list
 - Fast iterating
 - Slow access to individual elements (using **get**(int), **set**(int, Object))
 - 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

2/5/2008

(c) University of Washington

11-15

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 will always return set elements in a specified order
 - Requires that elements implement the **compareTo**(Object) method, returning a negative, 0, or positive number to mean $<$, $=$, or $>$, respectively

2/5/2008

(c) University of Washington

11-16

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>

2/5/2008

(c) University of Washington

11-17

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
}
```

2/5/2008

(c) University of Washington

11-18

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 able to be compared, using `compareTo`
- Sorting affects the order in which keys and values are iterated through
 - `keySet()` returns a `SortedSet`
 - `values()` returns an ordered collection

2/5/2008

(c) University of Washington

11-19

Preview of Coming Attractions

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

2/5/2008

(c) University of Washington

11-20